Western Graduate&PostdoctoralStudies

Western University

Scholarship@Western

Electronic Thesis and Dissertation Repository

8-14-2024 10:30 AM

Approximation Algorithms for High Multiplicity Strip Packing, Thief Orienteering, and K-Median

Andrew Bloch-Hansen,

Supervisor: Solis-Oba, Roberto, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science
© Andrew Bloch-Hansen 2024

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Theory and Algorithms Commons

Recommended Citation

Bloch-Hansen, Andrew, "Approximation Algorithms for High Multiplicity Strip Packing, Thief Orienteering, and K-Median" (2024). *Electronic Thesis and Dissertation Repository*. 10338.
https://ir.lib.uwo.ca/etd/10338

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

This thesis investigates three research objectives on three different problems: (1) algorithm design for problems in which the input can be grouped into a small number of classes, demonstrated on the high multiplicity strip packing problem; (2) algorithm design for problems with multiple interdependent sub-problems, demonstrated on the thief orienteering problem; and (3) algorithm design using neural networks with few layers, demonstrated on the k-median problem.

(1) The two-dimensional strip packing problem consists of packing in a rectangular strip of width 1 and minimum height a set of n rectangles, where each rectangle has width $0 < w \leq 1$ and height $0 < h \leq h_{max}$. We consider the high-multiplicity version of the problem in which there are only K different types of rectangles. For the case when $K = 3$, we give an algorithm that produces solutions requiring at most height $\frac{3}{2}h_{max} + \epsilon$ plus the height of an optimal solution, where ϵ is any positive constant. For the case when $K = 4$, we give an algorithm yielding solutions of height at most $\frac{7}{3}h_{max} + \epsilon$ plus the height of an optimal solution. For the case when $K > 3$, we give an algorithm that gives solutions of height at most $\lfloor \frac{3}{4}K \rfloor + 1 + \epsilon$ plus the height of an optimal solution.

(2) We consider routing an agent called a *thief* through a weighted graph $G = (V, E)$ from a start vertex s to an end vertex t. A set I of items each with weight w_i and profit p_i is distributed among $V \setminus \{s, t\}$. The thief, who has a knapsack of capacity W, must follow a simple path from s to t within a given time T while packing in the knapsack a set of items taken from the vertices along the path of total weight at most W and maximum profit. The travel time across an edge depends on the edge length and current knapsack load.

The thief orienteering problem (ThOP) is a generalization of the orienteering problem, the longest path problem, and the 0-1 knapsack problem. We prove that there exists no approximation algorithm for ThOP with constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$, and we present a polynomial-time approximation scheme (PTAS) for ThOP when G is directed and acyclic (DAG) that produces solutions using time at most $T(1 + \epsilon)$ for any constant $\epsilon > 0$. We show how to transform instances of ThOP on outerplanar and series-parallel graphs into equivalent instances of ThOP on DAGs; therefore, yielding a PTAS for these graph classes as well. We present a fully polynomial-time approximation scheme (FPTAS) for ThOP on arbitrary undirected graphs where the travel time depends only on the lengths of the edges and T is the length of a shortest path from s to t plus a constant K. Finally, we present a FPTAS for a version of the problem where the input graph is a clique.

(3) The k-median problem (KMP) is a classical clustering problem where given n locations one wants to select k locations such that the total distance between every non-selected location and its nearest selected location is minimized. We present a modified Hopfield network for KMP and experimentally evaluate it against several neural networks and local search algorithms, demonstrating that our algorithm produces solutions very quickly with competitive approximation ratios. We describe several improvements to our algorithm which could help us match the state-of-the-art algorithms.

Keywords LP-relaxation, combinatorial optimization, two-dimensional strip packing, high multiplicity, approximation algorithm, thief orienteering problem, knapsack problem, dynamic programming, approximation scheme, series-parallel, outerplanar, hopfield networks, k-median problem, neural networks, local search

Summary for Lay Audience

Many real-life applications require computing solutions to complex optimization problems: Using a computer requires solving a series of scheduling problems to coordinate the execution of its various processes, ordering a package from Amazon requires solving a variety of packing and scheduling problems to transport and deliver products, using Google Maps to give directions requires solving network routing problems to provide an optimum path, and so on. Moreover, businesses in fields such as manufacturing, supply chain management, and other operations research fields are faced with an even larger number of complicated optimization problems to solve; modeling each business's unique industrial challenge requires adding a variety of constraints to classical problem formulations and hence the same algorithm cannot be used to solve all of the problems, even if the problems are closely related. Research that creates new and improved algorithms for optimization problems enables many of today's technologies. Hence, the design and analysis of efficient algorithms for these problems is of critical importance.

This thesis focuses on the design and analysis of efficient algorithms for optimization problems. In particular, we investigate three research objectives on three different problems: (1) algorithm design for problems in which the input can be grouped into a small number of classes, demonstrated on the high multiplicity strip packing problem; (2) algorithm design for problems with multiple interdependent sub-problems, demonstrated on the thief orienteering problem; and (3) algorithm design using neural networks with few layers, demonstrated on the k-median problem.

(1) We design an approximation algorithm for the high multiplicity strip packing problem, which involves efficiently placing rectangles inside of a bigger rectangle, that takes advantage of the small number of input classes to outperform the more general algorithms that treat all the input items individually.

(2) We design an approximation algorithm for a restricted version of the thief orienteering problem, which involves selecting a route between two points and collecting items along that route and placing them in a knapsack, that can compute solutions arbitrarily close to an optimal solution at the expense of taking longer to compute solutions.

(3) We design a modified Hopfield neural network for the k-median problem, which involves identifying a number of cluster centers such that the input locations are all assigned to a nearby cluster, and perform an experimental evaluation against other local search algorithms and neural networks.

We conclude this thesis with a discussion of future research that could improve or extend our work.

Co-Authorship Statement

This thesis consists of four research papers that have already been published, are under review, or are in preparation. Roberto Solis-Oba contributed research ideas to the design and analysis of each of the contributions and helped review each of the manuscripts. The contributions of the other co-authors are listed below. The list of authors for each paper uses alphabetical order.

- Chapter 3 includes an article titled "High Multiplicity Strip Packing with Three Rectangle Types" co-authored by Roberto Solis-Oba and Andy Yu. A preliminary version of this paper is published in [14], and the full version is currently under review at the journal Theory of Computing Systems. Andy Yu contributed to an earlier version of the algorithm and Andrew Bloch-Hansen contributed with the design and analysis of the final algorithm, implementation and experimental evaluations, creation of all figures, and manuscript preparation and review.
- Chapter 4 includes an article titled "Algorithms for the Thief Orienteering Problem on Directed Acyclic Graphs" co-authored by Daniel R. Page and Roberto Solis-Oba. A preliminary version of this paper is published in [12], and the full version is currently under review at the journal Theoretical Computer Science. Daniel R. Page contributed to the inapproximability results and helped to review the manuscript and Andrew Bloch-Hansen contributed to the design and analysis of the algorithms, creation of all figures, and manuscript preparation and review.
- Chapter 5 includes an article titled "The Thief Orienteering Problem on Series-Parallel Graphs" co-authored by Roberto Solis-Oba. A preliminary version of this paper is published in [13], and the full version is currently under review at the Journal Acta Informatica. Andrew Bloch-Hansen contributed to the design and analysis of the algorithms, creation of all figures, and manuscript preparation and review.
- Chapter 6 includes an article titled "A Modified Hopfield Network for the K-Median Problem" co-authored by Cody Rossiter and Roberto Solis-Oba. This paper has not yet been submitted for peer review. Cody Rossiter contributed to an earlier version of the algorithm and Andrew Bloch-Hansen contributed with the design and analysis of the final algorithm, implementation and experimental evaluations, creation of all figures, and manuscript preparation and review.

Acknowledgements

First, I want to thank my supervisor Professor Roberto Solis-Oba for his guidance and mentoring over the past eight years. During this time, I have learned a great deal from him about teaching, researching, writing, and most of all, not giving up when the algorithm does not seem to work the way I want it to. Roberto has taught me a number of creative ways to tackle problems when all else seems lost. Thank you so much Roberto for all of your help and for giving me the opportunity to work with you.

I want to thank my co-authors Nasim Samei, Daniel R. Page, and Cody Rossiter, and my co-workers Tanner Bohn and Alex Brandt, for their enlightening discussions and suggestions.

I want to thank the Department of Computer Science at Western University. I have spent many years learning and growing at this university, and I am very grateful for the time that I have spent here. In particular, I would like to thank Janice Wiersma and Ange Muir for their guidance on pretty much everything throughout graduate school, and I would like to thank Professor Dan Lizotte and Professor Marc Moreno Maza for their guidance through UWORCS, through my research, and finally through job searching.

I want to thank all of the students that I have ever taught. Thank you for giving me the opportunity to teach you, and thank you to those that taught me in return. Many of you have helped shape me into the person that I am today.

I want to thank my parents. While they might not have been as interested in my research as I was, they patiently listened to all of my algorithmic ramblings as I tried to design things that actually worked, and they excitedly came to my presentations to cheer me on. I couldn't have come this far without the support and love from my parents.

Finally, to Linxiao Wang: Thank you for all the support you provided to help me on my journey through graduate school. And thank you for pulling me out of my shell and introducing me to whole new worlds of traveling, language, pets, cooking, and so much more. My life is so much fuller with you in it.

Table of Contents

Abstract **ii**

Summary for Lay Audience **iii**

Co-Authorship Statement **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **xi**

List of Figures **xiii**

1 Introduction **1**
- 1.1 Background 2
  - 1.1.1 Optimization Problems 2
  - 1.1.2 Complexity Theory 2
  - 1.1.3 Approximation Algorithms 3
- 1.2 Motivation 4
  - 1.2.1 Modeling Industrial Challenges 4
  - 1.2.2 New Algorithm Design Techniques 5
- 1.3 Algorithm Design Techniques 5
  - 1.3.1 Greedy 6
  - 1.3.2 Local Search 6
  - 1.3.3 Dynamic Programming 6
  - 1.3.4 Linear Program Rounding 7
- 1.4 Contributions 7
  - 1.4.1 High Multiplicity Strip Packing 7
  - 1.4.2 Thief Orienteering on Directed Graphs 8
  - 1.4.3 Thief Orienteering on Undirected Graphs and Special Graph Classes 9
  - 1.4.4 Modified Hopfield Network for K-Median 9
  - 1.4.5 Awards, Publications, and Preprints 10

2 Literature Review **11**
- 2.1 High Multiplicity Strip Packing 11
  - 2.1.1 Introduction 11
  - 2.1.2 The Bin Packing Problem 12
  - 2.1.3 The Cutting Stock Problem 14
  - 2.1.4 The Rectangle Packing Problem 16
  - 2.1.5 The Two-Dimensional Strip Packing Problem 17
  - 2.1.6 High Multiplicity Problems 20
- 2.2 Thief Orienteering 21

2.2.1 Introduction 21
2.2.2 The 0-1 Knapsack Problem 22
2.2.3 The Orienteering Problem 24
2.2.4 The Thief Orienteering Problem 25
2.2.5 The Packing While Traveling Problem 27
2.2.6 The Traveling Thief Problem 28
2.3 Hopfield Networks and the K-Median Problem 30
2.3.1 The K-Median Problem 31
2.3.2 Hopfield Networks for Other Problems 34

3 Paper 1: High Multiplicity Strip Packing 36
3.1 Introduction 37
3.2 Solving FSP in Polynomial time 38
3.3 Algorithm for HMSP with Three Rectangle Types 39
3.3.1 Two Rectangle Types Per Configuration 42
Ordering the Configurations 43
Pairing Configurations 44
Rounding Fractional Rectangles 46
Rounding Fractional Rectangles: One Wide Rectangle 47
Rounding Fractional Rectangles: Two Wide Rectangles 50
Rounding Fractional Rectangles: Some S_{Casei} is Empty 53
3.3.2 Three Rectangle Types in a Configuration 55
Ordering the Configurations 55
Rounding Fractional Rectangles: One Wide Rectangle 56
Rounding Fractional Rectangles: Two Wide Rectangles 57
3.3.3 Fewer Than Three Configurations 61
3.3.4 Differing Number of Rectangle Types in Each Configuration 61
3.4 Polynomial Time Implementation 62
3.5 4-Type Algorithm 63
3.5.1 Grouping Vertical Sections 63
3.5.2 Case1: $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} \leq 1$ 63
3.5.3 Case2: $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} > 1$ and $f_{1(i)} + f_{2(i)} + f_{3(i)} \leq 1$ 64
3.5.4 Case3: $f_{1(i)} + f_{2(i)} + f_{3(i)} > 1$ and $f_{1(i)} + f_{2(i)} \leq 1$ 64
3.5.5 Case4: $f_{1(i)} + f_{2(i)} > 1$ 65
3.6 K-Type Algorithm 65
3.6.1 Pairing Configurations 65
3.6.2 Processing Fractional Rectangles 66
3.6.3 Packing Rectangles into the Regions $R_{j,j+1}$ 67
3.6.4 Packing the Split Rectangles 68
3.7 Experimental Results 69
3.7.1 Input Data 69
3.7.2 Test Cases 70
3.7.3 Experimental Results 70
3.7.4 Final Observation 73
3.8 References 74

4 Paper 2: Thief Orienteering on Directed Graphs 76
4.1 Introduction . . . 78
4.2 Inapproximability . . . 80
4.3 Algorithm for Thief Orienteering on DAGs . . . 82
  4.3.1 Profit Table . . . 82
  4.3.2 UpdateProfitTable . . . 84
    Selecting a Solution from the Profit Table . . . 84
  4.3.3 Algorithm Analysis . . . 87
4.4 PTAS for Thief Orienteering on DAGs . . . 88
  4.4.1 PTAS Analysis . . . 91
4.5 Thief Orienteering on Undirected Graphs . . . 95
  4.5.1 Building the Lists of Detours . . . 98
  4.5.2 Transforming the Problem into a Shortest Paths Problems . . . 99
4.6 Thief Orienteering on Cliques . . . 105
  4.6.1 Clique Profit Table . . . 106
  4.6.2 Computing the Clique Profit Table . . . 106
  4.6.3 Algorithm Analysis . . . 107
4.7 References . . . 108

5 Paper 3: Thief Orienteering on Undirected Graphs 111
5.1 Introduction . . . 112
5.2 Thief Orienteering on Outerplanar Graphs . . . 114
  5.2.1 Cut Vertices . . . 115
  5.2.2 Transforming $\mathcal{G}_{\ell_x}$ into a DAG . . . 116
    Adding Directed Chords to $\mathcal{G}$ . . . 117
    Connecting Duplicate Vertices to the Top and Bottom Paths . . . 117
  5.2.3 Algorithm Analysis . . . 119
5.3 Thief Orienteering on Series-Parallel Graphs . . . 125
  5.3.1 Removing Vertices with Degree 1 . . . 125
  5.3.2 Cut Vertices . . . 125
  5.3.3 Transforming $\mathcal{G}_{\ell_x}$ into a DAG . . . 126
    Step 1: Edges Close to s and t . . . 128
    Step 2: Transforming a Parallel Subgraph G_i of $\mathcal{G}_{\ell_x}$ into Several DAGs . . . 129
    Step 3: Edges Connecting Nested Parallel Graphs Containing s or t . 132
5.4 Algorithm Analysis . . . 136
5.5 References . . . 153

6 Paper 4: A Modified Hopfield Network for the K Median Problem 155
6.1 Introduction . . . 156
6.2 A Typical Hopfield Network Formulation . . . 158
6.3 Our Modified Hopfield Network Model . . . 162
6.4 The Modified Hopfield Network Algorithm . . . 164
  6.4.1 Finding the Best Facility Neuron to Activate . . . 166
6.5 Experimental Results . . . 170

6.5.1 Input Data . . . 171
6.5.2 Testing Environment . . . 172
6.5.3 Initialization Strategies . . . 172
6.5.4 Performing Multiple Runs . . . 175
6.5.5 Comparing Against the Neural Networks . . . 175
6.5.6 Comparing Against the Local Search Algorithms . . . 177
6.5.7 Approximation Ratio Versus Number of Facilities . . . 178
6.6 Analysis . . . 180
6.6.1 Initialization Strategies . . . 180
6.6.2 Performing Multiple Runs . . . 182
6.6.3 Comparing Against the Neural Networks . . . 182
6.6.4 Comparing Against the Local Search Algorithms . . . 183
6.6.5 Single-Swap Versus Multi-Swap . . . 184
6.6.6 Approximation Ratio Versus Number of Facilities . . . 185
6.7 Conclusion . . . 186
6.8 References . . . 187

7 Conclusions 190
7.1 Summary . . . 190
7.2 Future Work . . . 191
7.2.1 HMSP . . . 191
Improvements to the Algorithm for the Three Type Problem . . . 191
Improvements to the Algorithm for the Four Type Problem . . . 192
Improvements to the Algorithm for the K Type Problem . . . 192
K Type Problem Constrained to a Fixed Number of Configurations . 193
Integrality Gap of the Linear Program . . . 194
Experimental Evaluation of the Algorithms . . . 194
Three-Dimensional High Multiplicity Strip Packing . . . 195
High Multiplicity Rectangle Packing . . . 195
High Multiplicity Scheduling Problems . . . 196
7.2.2 Thief Orienteering Problem . . . 197
Exceeding the Time Limit . . . 197
Transforming Additional Graph Classes to DAGs . . . 198
Experimental Evaluation of the Algorithms for Industrial Challenges 198
ThOP Variants . . . 198
Approximation Algorithms for TTP . . . 199
7.2.3 Modified Hopfield Network for K-Median . . . 199
Swapping Multiple Facilities . . . 199
Improving the Inner Value Metric . . . 200
Improving the Neuron Update Function . . . 201
Simulated Annealing . . . 202
Learning from Previous Solutions . . . 202
Analyzing the Approximation Ratio . . . 203
Experimental Results using GPUs . . . 203
Modified Integer Program for K-Median . . . 203

Broader Experimental Evaluations 204
K-Median Problem Variants 204
Other Optimization Problems 205

Bibliography **206**

Curriculum Vitae **220**

List of Tables

2.1 Significant algorithms, and their performances, for the bin packing problem. 13

2.2 Significant algorithms, and their performances, for the cutting stock problem. 15

2.3 Significant algorithms, and their performances, for the rectangle packing problem. 17

2.4 Significant algorithms, and their performances, for the strip packing problem. 19

2.5 Significant FPTAS for the knapsack problem. The notation $\tilde{O}()$ is a variation of the big-O that ignores logarithmic factors and Ω defines a lower bound. 23

2.6 Significant algorithms, and their performances, for the orienteering problem. 26

2.7 All algorithms for the thief orienteering problem. 27

2.8 Significant algorithms, and their performances, for the packing while traveling problem. 27

2.9 Significant algorithms, and their performances, for the orienteering problem. 29

2.10 Local search algorithms and neural networks designed for the k-median problem. 32

2.11 Hopfield networks being applied to optimization problems. 34

3.1 One thousand trials. Widths are generated from the interval $[0.01, 0.05]$, heights are generated from the interval $[0.90, 1]$. The results are separated into categories depending on how many different rectangle types appear in each configuration, and within these categories the mean average height increase, minimum height increase, and maximum height increase is listed with respect to the height of an optimal fractional packing. Note that in all 1000 trials the fractional packing contained three configurations. 72

6.1 Matrix representation of the neurons in a three location k-median problem. The first column contains only the facility neuron variables. Each row begins with a facility neuron and the following neurons are the client-facility neurons that can be served by that facility. 159

6.2 Matrix representation of the neurons in a three location k-median problem. The neuron state variables V_1 to V_n correspond to the F_i facility neuron variables and the remaining neuron state variables V_{n+1} to V_{n^2+n} correspond to the $x_{i,j}$ client-facility neuron variables. 160

6.3 Initialization Strategies: Ratios and runtimes of our modified Hopfield network for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We use three different initialization strategies. The best performance for each test case is bolded. 174

6.4 Performing Multiple Runs: Ratios and runtimes of our modified Hopfield network for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. 176

6.5 Comparing Against the Neural Networks: Ratios and runtimes of the solutions computed by the neural networks for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We allotted our algorithm 40 runs and the other neural networks 4 runs. The best performance for each test case is bolded. 177
6.6 Comparing Against the Local Search Algorithms: Ratios and runtimes of the solutions computed by the local search algorithms for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We allotted our algorithm 100 runs and the other local search algorithms 5 seconds. The best performance for each test case is bolded. For the algorithms of Arya et al. and Cohen-Addad et al., both the single-swap (s) and multi-swap (m) results are listed. 178
6.7 Approximation Ratio Versus Number of Facilities: Ratios and runtimes of the solutions computed by the neural networks for a selection of the test results from USCA312. The best performance for each test case is bolded. . 179
6.8 Approximation Ratio Versus Number of Facilities: Ratios and runtimes of the solutions computed by the local search algorithms for a selection of the test results from USCA312. The best performance for each test case is bolded. For the algorithms of Arya et al. and Cohen-Addad et al., both the single-swap (s) and multi-swap (m) results are listed. 179

List of Figures

2.1 The bin packing problem: given a set $A = \{a_1, a_2, ..., a_n\}$ of n items, pack them into the smallest possible number of bins. 13

2.2 a) In the bin packing problem n sizes of items must be given in the input. b) In the cutting stock problem only K sizes of items and K multiplicities must be given in the input. Even if the value of n is very large, if the value of K is small the size of the input for the cutting stock problem is small. . . 15

2.3 The rectangle packing problem: given a set $A = \{a_1, a_2, ..., a_n\}$ of n items, pack the maxmimum subset of them into a rectangular container. 16

2.4 The strip packing problem: given n rectangles, pack them all in to a rectangular strip with minimum height. 18

2.5 The knapsack problem: given a set $I = \{i_1, i_2, ..., i_n\}$ of n items, pack a subset of them with maximum profit into a knapsack. 23

2.6 An entry $A[j]$ in the knapsack profit table is a list of tuples (w, p) that are restricted to selecting subsets of only the first j items. 24

2.7 An instance of the k-median problem: given a graph $G = (V, E)$ with n vertices and nonnegative length d_{ij} for every edge, choose $k = 2$ vertices as medians that minimizes the sum of distances from each vertex to its nearest median. 31

3.1 A configuration with *base configuration* (4,2,3). The fractional rectangles are shaded in a darker color. 39

3.2 (a) Replacing the fractional rectangles in S_{Common} with whole rectangles increases the height of the packing by at most 1. (b) Within each configuration, the rectangles in $S_{Uncommon}$ are sorted according to their fractional values. $S_{Uncommon}$ consists of four vertical sections: s_1, s_2, s_3, and s_4. 41

3.3 The fractional rectangles in C_1 with fractional values f_{1l} and f_{1r} define $B_{1,1}$, the fractional rectangles in C_2 with fractional values f_{2l} and f_{2r} define $B_{2,3}$, and the fractional rectangles in C_3 with fractional values f_{3l} and f_{3r} define $B_{1,2}$. 43

3.4 (a) C_1 has been flipped upside down and the sets F_1 and F_2 are highlighted in blue and green, respectively. (b) Rectangles in C_1 have been shifted upwards to create the region C_{A1} to hold the rectangles in $F_1 \cup F_2$ and the rest of the fractional rectangles have been rounded up. 45

3.5 $numWide = 1$ and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$. 47

3.6 $numWide = 1$ and $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$. 50

3.7 $numWide = 2$, $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, $(1 - f_{3r})h_{3r} > \frac{1}{2}$, $(1 - f_{2l})h_{2l} > \frac{1}{2}$, and $(1 - f_{1l})h_{1l} > \frac{1}{2}$. 52

3.8 $numWide = 2$, $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, $(1 - f_{3r})h_{3r} > \frac{1}{2}$, $(1 - f_{2l})h_{2l} > \frac{1}{2}$, $(1 - f_{1l})h_{1l} > \frac{1}{2}$, and $f_{1l} \geq f_{2l} \geq f_{3r}$. 53

3.9 If $S = S_{Case1} \cup S_{Case2}$ and $numWide = 0$, then there is exactly one vertical section $s_i \in S_{Case2}$. (a) The largest fraction is more than $\frac{1}{2}$, and (b) the largest fraction is less than $\frac{1}{2}$. 54

3.10 None of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 1$, $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, and (a) $(1-a)h_a \leq \frac{1}{2}$ and (b) $(1-a)h_a > \frac{1}{2}$ so $f > \frac{1}{2}$. 57
3.11 $S = S_{Case1} \cup S_{Case2} \cup S_{Case3}$, $numWide = 2$, and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$. (a) C_2's leftmost fractional value is packed within S_{Case2} and $(1-a)h_a > \frac{1}{2}$. (b) C_3's leftmost fractional value is packed within S_{Case2} and $(1-a)h_a > \frac{1}{2}$, $(1-e)h_e < \frac{1}{2}$, and $f > \frac{1}{2}$. (c) C_2 has only two rectangle types and $(1-a)h_a > \frac{1}{2}$, $(1-e)h_e < \frac{1}{2}$, and $f > \frac{1}{2}$. 58
3.12 None of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 2$, $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, and (a) $a > \frac{1}{2}$ and (b) $a < \frac{1}{2}$ so $c > \frac{1}{2}$ and $d > \frac{1}{2}$. 59
3.13 When there are only two configurations, a vertical section is classified as S_{Case1} if $f_{1(i)} + f_{2(i)} \leq 1$ and classified as S_{Case2} if $f_{1(i)} + f_{2(i)} > 1$. 61
3.14 When $K = 4$ and the fractional packing has exactly four configurations, our algorithm partitions the packing into at most 4 cases. 64
3.15 Configurations with wide rectangle types are adjacent and near the bottom. . 66
3.16 The K configurations are stacked; if there is an odd number of configurations, the topmost configuration is rounded up instead of paired. After creating the regions $R_{j,j+1}$, the rectangles whose widths are greater than half the width of the strip are again grouped at the bottom of the configuration. . 67
3.17 The configurations have been paired, the fractional rectangles have been processed, and the split rectangles have been made whole and packed in regions placed at the top of the packing. 68
3.18 Average height increase with respect to optimal fractional packing. Each data line represents a different value used for w when selecting the rectangle widths and the x-axis shows the value used for h when selecting the height from the interval $[1 - 0.1h, 0.9 - 0.1h]$. From left to right the chart shows taller to shorter rectangles, and from top to bottom the series of lines show narrower to wider rectangles. 71

4.1 An example of a DAG that has two possible paths from s to t. In each vertex u there is a triplet (index, profit, weight) for each item in I_u. 82
4.2 Let σ be the path from s to t that includes the blue edges. The red edges belong to a shortest path (of length $s.dist = 100$) between s and t, the green edges belong to a shortest path (of length $v_3.dist = 50$) between v_3 and t, and the purple edge belongs to a shortest path (of length $v_5.dist = 60$) between v_5 and t. The path σ decomposes into the unique list $D = D_1, D_2$ of detours where $D_1 = \{s, v_2\}$ and $D_2 = \{v_3, v_5, v_6\}$. 97

5.1 An example outerplanar graph that has vertices of degree 1. Note that s is re-named, vertices other than s or t with degree 1 are removed, and the cut vertices c_1, c_2, and c_3 are labelled. 115
5.2 The outerplanar graph from Figure 5.1 after identifying the subgraphs $\mathcal{G}_{\ell_x}$. . 115

5.3 An example of a subgraph $\mathcal{G}_{\ell_x}$ after indexing the vertices; the graph consists of a cycle with chords. Vertex 2 has two incident chords: the first chord is red and the second chord is green. 116

5.4 Directing the chord $(1,6)$ of Figure 5.3: vertices $\hat{1}_1$ and $\hat{6}_1$ are added to $\mathcal{G}$ along with directed chords $(1,\hat{6}_1)$ and $(6,\hat{1}_1)$. 118

5.5 Directing the chords incident on vertex 2 of Figure 5.3. 118

5.6 The chords of vertex 1 span vertices 6 to 8. The chords of vertex 8 span vertices 1 to 4. The chords of vertex 4 span vertices 8 and 9. Vertices 1, 6, and 8 are located at the start of a span. Vertices 2, 3, and 7 are located in the middle of a span. Vertices 4, 8, and 9 are located at the end of a span. 118

5.7 Processing the chord $(1,6)$: the directed edges $(\hat{1}_1,2)$ and $(\hat{6}_1,7)$ are added. 120

5.8 Connecting the duplicate vertices for vertex 2. Note that a copy of 8, $\hat{8}_1$, was created, even though vertex 8 has no incident chords, because 8 is in the middle of a span from vertex 2. 120

5.9 A path σ in G that routes through the red non-chord $(s,6)$, the blue chords $(6,1)$ and $(1,7)$, the blue non-chords $(7,8)$ and $(8,9)$, the green chord $(9,2)$, and the green non-chords $(2,3)$, $(3,4)$, $(4,5)$, and $(5,t)$. 120

5.10 The path σ' in $\mathcal{G}$ corresponding to σ' in Figure 5.9 routes through the red non-chord $(s,6)$, the blue chords $(6,\hat{1}_1)$ and $(\hat{1}_1,\hat{7}_1)$, the blue non-chords $(\hat{7}_1,8)$ and $(8,9)$, the green chord $(9,\hat{2}_2)$, and the green non-chords $(\hat{2}_2,\hat{3}_1)$, $(\hat{3}_1,\hat{4}_1)$, $(\hat{4}_1,5)$, and $(5,t)$. 121

5.11 A simple cycle $C : u, v_1, v_2, ..., v_z, u$ in G, where u is the only vertex of C in the top path, is not encoded as a directed cycle in $\mathcal{G}$ because the algorithm does not add any directed edges from the duplicate vertices of $v_1, v_2, ..., v_z$ back to u or any of its duplicates. The original vertices are shaded red and the part of C that exists in $\mathcal{G}$ is shaded green. 122

5.12 Identifying the subgraphs $\mathcal{G}_{\ell_x}$ in a series-parallel graph. 126

5.13 A subgraph $\mathcal{G}_{\ell_x}$ that does not contain either s or t (left) and so the edges are directed from s to t (right). 127

5.14 A subgraph $\mathcal{G}_{\ell_x}$ containing several parallel subgraphs. Both s and t are contained in $\mathcal{G}_{\ell_x}$, so there are no other subgraphs $\mathcal{G}_{\ell_x}$ as there are no cut vertices in a shortest path from s to t. 127

5.15 Edges in the lowest indexed maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ containing s or t are directed away from s and towards t. 129

5.16 (Left) The maximal parallel subgraph G_i is shown before being processed by Step 2; $\hat{G}_i$ consists of a single connected parallel subgraph (shown in blue). (Right) After G_i has been processed by Step 2 the simple paths between a'_i and b'_i are directed from a'_i to b'_i and the simple paths between b''_i and a''_i are directed from b''_i to a''_i. Any edges incident on the original terminals a_i and b_i before processing G_i are now incident on the corresponding terminals of G'_i and G''_i. 129

5.17 (Left) The maximal parallel subgraph $G_i = G_4$ is shown before being processed by Step 2 and since the maximal parallel subgraphs G_1, G_2, and G_3 are each contained within G_4 (so each is transformed into DAGs before G_4 is processed) then $\hat{G}_i$ consists of multiple components. The vertices and edges of $\hat{G}_{4_1}$, $\hat{G}_{4_2}$, and $\hat{G}_{4_3}$ are shown in blue. (Right) After G_i has been processed by Step 2, the components of $\hat{G}_i$ have been duplicated and their incident edges have been directed. 130
5.18 (Top) Two adjacent maximal parallel undirected subgraphs G_1 and G_2 were originally connected by a single edge between terminals b_2 and a_1, shown in red. (Middle) Since the maximal parallel subgraphs are processed in increasing order of index, G_1 was transformed into two DAGs first, and the edge between b_2 and a_1 transformed into the edges between b_2 and a'_1 and a''_1. (Bottom) After G_2 has been transformed into two DAGs, there is a single edge between b'_2 and a'_1 and a single edge between b''_2 and a''_1. 132
5.19 This maximal parallel subgraph G_i contains two parallel components pc_{i1} (blue) and pc_{i2} (red), each of which contain three maximal parallel subgraphs. Note that the parallel component pc_{i1} contains s and the parallel component pc_{i2} contains t. 133
5.20 The parallel component pc_{i1} from Figure 5.19 after processing Step 2 but before processing Step 3. 133
5.21 The parallel component pc_{i1} of Figure 5.19 contains s but not t and so some of the DAGs for each G_j of pc_{i1} have been deleted and some of the edges (highlighted in red) have been added. 134
5.22 The parallel component pc_{i2} of Figure 5.19 contains t but not s and so some of the edges for each G'_j of pc_{i2} and G''_j of pc_{i2} (highlighted in red) have been added. 134
5.23 A parallel component pc_{i3} containing both s and t, but not in the same maximal parallel subgraph G_j. The edges highlighted in red have been added. 135
5.24 A parallel component pc_{ist} of a maximal parallel subgraph G_i contains DAGs for each maximal parallel subgraph G_j contained within G_i. Since pc_{ist} contains both s and t within a maximal parallel subgraph G_j of G_i, the edges highlighted in red have been added. 137
5.25 Parallel component pc_{i1} from Figure 5.24 is shown as if the terminals of G_s were entangled with the adjacent parallel subgraphs. (Top-left) The parallel component is shown is it appears in the undirected graph G. (Top-right) Terminal a_s and its entangled vertex are highlighted in orange and terminal b_s and its entangled vertex are highlighted in purple. (Bottom-left) Some of the adjacent parallel subgraphs adjacent to G_s are deleted and hence it is clear which entangled vertices to merge. (Bottom-right) If $G_s = G_t$, then after changing the edges during Step 3 the entangled pairs need to be adjusted so as to not form a cycle. The pairs of vertices of the same colors will be merged together. 137

5.26 When t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1, the segment $<t_1, t_2>$ exits G_s. (Left) G_s does not contain t and so b'_s is the vertex corresponding to t_1. (Right) G_s contains t and so b''_s is the vertex corresponding to t_1. 142
5.27 When t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1, and the segment $<t_1, t_2>$ stays in G_k. (Left) Segment $<t_1, t_2>$ exits into an adjacent maximal parallel subgraph G_q. (Right) Segment $<t_1, t_2>$ exits into a maximal parallel subgraph G_p. 143
5.28 When t_1 (purple) is to the right of s and t_2 (red) is to the left of t_1, the segment $<t_1, t_2>$ is still contained within G_s. (Left) G_s does not contain t and so b''_s is the vertex corresponding to t_1. (Right) G_s contains t and so b''_s is the vertex corresponding to t_1. 143
5.29 When t_1 (purple) is to the left of s and t_2 (red) is to the left of t_1, the segment $<t_1, t_2>$ exits G_s. (Left) G_s does not contain t and so a''_s is the vertex corresponding to t_1. (Right) G_s contains t and so a'_s is the vertex corresponding to t_1. 144
5.30 When t_1 (purple) is to the left of s and t_2 (red) is to the right of t_1, the segment $<t_1, t_2>$ is still contained within G_s. (Left) G_s does not contain t and so a'_s is the vertex corresponding to t_1. (Right) G_s contains t and so a'_s is the vertex corresponding to t_1. 144
5.31 $s = a_s$ and so the vertices of G_s were not duplicated. Vertice t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1. The segment $<t_1, t_2>$ exits G_s.145
5.32 $s = a_s$ and so the vertices of G_s were not duplicated. t_1 (purple) is to the right of s and t_2 (red) is to the left of t_1. (Left) G_s does not contain t and so b_s is the vertex corresponding to t_1. (Right) G_s contains t and so b_s is the vertex corresponding to t_1 and $t_2 = t$ as this is the only feasible path. 146
5.33 t_1 (purple) is to the left of s and t_2 (red) is to the right of t_1; observe that the segment $<s, t_1>$ exits G_s into the maximal parallel subgraph G_p containing G_s. (Left) G_p does not contain t and so a'_i is the vertex corresponding to t_1. (Right) G_p contains both s and t and so a''_i is the vertex corresponding to t_1. 146
5.34 $s = a_s$ and so the vertices of G_s were not duplicated. t_1 (purple) is to the left of s and t_2 (red) is to the left of t_1. (Top-left) The segment $<s, t_1>$ exits G_s into a maximal parallel subgraph G_p containing G_s and the vertex a''_p corresponds to t_1. (Bottom-left) The segment $<s, t_1>$ exits G_s and enters an adjacent maximal parallel subgraph G_q so the vertex b''_q corresponds to t_1. (Top-right) The segment $<s, t_1>$ exits G_s into a maximal parallel subgraph G_p containing G_s and the vertex a''_p corresponds to t_1. (Bottom-right) The segment $<s, t_1>$ exits G_s and enters an adjacent maximal parallel subgraph G_q so the vertex b''_q corresponds to t_1. 147
5.35 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the right of t_{i+1}. b'_r is the vertex that corresponds to t_{i+1}. 148

5.36 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the left of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. (Left) t_{i+1} is a terminal of G_s which also contains t and so b'_s is the vertex that corresponds to t_{i+1}. Note that in this case $t_{i+2} = t$ due to how we renamed the terminals t_i. (Right) In all other cases, b''_r is the vertex that corresponds to t_{i+1}. 149
5.37 t_{i+1} (purple) is the left terminal a_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the right of t_{i+1}. (Left) t_{i+1} is a terminal of G_s which contains t and so a''_s is the vertex that corresponds to t_{i+1}. (Right) In all other cases, a'_r is the vertex that corresponds to t_{i+1}. 149
5.38 t_{i+1} (purple) is the left terminal a_r of G_r, t_{i+1} is to the left of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. a''_r is the vertex that corresponds to t_{i+1}. 150
5.39 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. (Top-left) The segment $<t_i, t_{i+1}>$ exits a parallel component containing s and so the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1}. (Bottom-left) The segment $<t_i, t_{i+1}>$ exits a parallel component not containing s and so the terminal b'_r of G''_r is the vertex that corresponds to t_{i+1}. (Top-right) G_r contains both s and t and the segment $<t_i, t_{i+1}>$ exits a parallel component containing s and so the terminal b'_r of G''_r is the vertex that corresponds to t_{i+1}. (Bottom-right) G_r contains both s and t and the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s and so the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1}. 151

6.1 The modified Hopfield network is represented using two matrices: The first matrix contains the inner values and the second matrix contains the activation values. 162
6.2 Continuing from the example in Figure 6.1, facility 1 had the lowest inner value and was deactivated. During Algorithm **??**, the first column in *innerValues* is set to 0, the inner values of the client-facility neurons are updated, and the inner values of client-facility neurons corresponding to any of the $k - 1$ remaining active facilities are set to 0 (colored in red). . . . 168
6.3 Candidate facilities should not be assigned clients if they cannot serve them better than the remaining $k - 1$ active facilities, and so the inner values of client-facility neurons in a column j that are less than or equal to the maximum inner value from an active facility in that column are set to 0 (colored in red). 169
6.4 Candidate facilities that are closer to a client than the closest active facility represent an improvement to the solution, and so the inner values of client-facility neurons in a column j that are greater than the maximum inner value from an active facility in that column are increased (colored in blue). 169
6.5 After computing the inner values of the client-facility neurons corresponding to inactive facilities, the best candidate facility is the facility neuron with the largest inner value (colored in blue). 170

Chapter 1

1 Introduction

Many real-life applications require computing solutions to complex optimization problems: Using a computer requires solving a series of scheduling problems to coordinate the execution of its various processes, ordering a package from Amazon requires solving a variety of packing and scheduling problems to transport and deliver products, using Google Maps to give directions requires solving network routing problems to provide an optimum path, and so on. Moreover, businesses in fields such as manufacturing, supply chain management, and other operations research fields are faced with an even larger number of complicated optimization problems to solve; modeling each business's unique industrial challenge requires adding a variety of constraints to classical problem formulations and hence the same algorithm cannot be used to solve all of the problems, even if the problems are closely related. Research that creates new and improved algorithms for optimization problems enables many of today's technologies. Hence, the design and analysis of efficient algorithms for these kinds of problems is of critical importance.

In this thesis we focus on the design and analysis of efficient algorithms for optimization problems. We explore network, packing, resource allocation, and clustering optimization problems. As part of this core research focus, we investigate three specific objectives:

- **Objective 1:** Create algorithm design techniques for solving problems in which the input can be grouped into a small number of classes and take advantage of this property to find efficient solutions.
- **Objective 2:** Create algorithm design techniques for solving optimization problems that contain multiple interdependent sub-problems that accurately model real-life optimization problems.
- **Objective 3:** Create algorithm design techniques for neural networks with few layers to efficiently solve optimization problems.

In Chapter 1.1 we provide background on optimization problems, computational complexity theory, and approximation algorithms. Next, in Chapter 1.2 we provide the motivation of our research and in Chapter 1.3 we describe some core algorithm design techniques that can be applied to optimization problems. Finally, in Chapter 1.4 we summarize the problems we investigated and describe our contributions.

This thesis is presented in the integrated-article format. In Chapter 2 we provide literature review corresponding to all of our contributions. Chapters 3-6 are integrated articles from relevant topics completed during the duration of the author's PhD: Chapter 3 investigates Objective 1, Chapters 4 and 5 investigate Objective 2, and Chapter 6 investigates Objective 3, and each of the articles provide detailed descriptions of the novel work. Chapter 7 concludes the thesis and discusses many possible future research directions. The included articles are:

- **Chapter 3:** High Multiplicity Strip Packing.
- **Chapter 4:** Thief Orienteering on Directed Graphs.
- **Chapter 5:** Thief Orienteering on Undirected Graphs.
- **Chapter 6:** A Modified Hopfield Network for K-Median.

1.1 Background

1.1.1 Optimization Problems

Problems in which the goal is to find the best solution from all possible solutions are called *optimization problems*. Often, solutions must satisfy a number of constraints to be considered a *feasible solution*. The standard form of a *continuous optimization problem* is:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) \leq 0, \text{ for all } i = 1, ..., m \\ & h_j(x) = 0, \text{ for all } j = 1, ..., p \\ \text{and} & x \in \mathbb{R}^n \end{array}$$

where x is a vector of variables, f is an *objective function*, $g_i(x) \leq 0$ is an *inequality constraint*, and $h_j(x) = 0$ is an *equality constraint*. Finding the best solution from all feasible solutions is equivalent to minimizing the objective function. Maximization problems can be solved by negating the objective function.

Continuous optimization problems allow the variables to take on an infinite number of values and hence there are an infinite number of feasible solutions. In *discrete optimization problems*, the solution space (the set of all possible solutions) includes a finite number of feasible solutions. A *combinatorial optimization problem* is a discrete optimization problem in which the variables used to express a feasible solution take on values from a *discrete* set.

All of the above formats of optimization problems are typically very difficult to solve, as trying to choose the best solution from among many possible solutions can take a large amount of computational resources.

1.1.2 Complexity Theory

Many algorithms that aim to compute exact solutions to optimization problems are found to take an exponentially increasing amount of time as the size of the optimization problem increases. In the field of *computational complexity theory*, optimization problems are grouped into different classes depending on the amount of computational resources required, such as the amount of storage or the time needed to compute a solution. The *time complexity* function measures the number of computational operations performed by an algorithm. The *space complexity* function measures the amount of memory required by an algorithm.

A *decision problem* is a problem with a yes or no answer, and all optimization problems have an equivalent decision problem; for example, for the optimization problem of finding the shortest path between two vertices u and v, the equivalent decision problem is to ask whether there is a path between u and v of length at most k, where $k \geq 0$. A *deterministic* algorithm always produces the same output for a given input and the complexity class P is the set of decision problems that can be solved exactly by deterministic algorithms in *polynomial time*, which means the time complexity function of the algorithm is polynomial with respect to the size of the input to the problem (in terms of the number of bits).

Let there exist an *oracle* that can be consulted during an algorithm to determine the optimal choices to make during each step in order to select an optimal solution. The complexity class NP is the set of decision problems that can be solved exactly in polynomial time by *non-deterministic* algorithms that consult the oracle in order to perform the minimum number of computations to select an optimal solution. It is currently unknown whether P = NP.

A decision problem P_1 *reduces* to a decision problem P_2 if there is a polynomial time algorithm that transforms instances p_1 of P_1 to instances p_2 of P_2 such that the solution to p_1 is yes if and only if the solution to p_2 is also yes. The complexity class NP-complete is the set of decision problems in NP such that for a problem p in NP-complete, every other problem in NP can be reduced to p in polynomial time. In other words, the existence of a deterministic polynomial time algorithm that exactly solves a problem in NP-complete implies the existence of deterministic polynomial time algorithms that exactly solve every problem in NP.

A problem P_1 is in the complexity class NP-hard if there is a problem P_2 in NP-complete that can be reduced to P_1 in polynomial time. Note that NP-hard problems do not need to be in NP. There are currently no known deterministic polynomial time algorithms that compute exact solutions to NP-complete and NP-hard problems.

For some problems, it is not known whether the problem belongs to the complexity class NP-hard. Many researchers devote their time to proving the complexity of a problem. One common approach to proving that a problem P_1 is NP-hard is to show that P_1 reduces to another problem P_2, where P_2 is already known to be NP-hard. As the logic goes, if P_1 is not NP-hard, then polynomial-time algorithms designed for P_1 could indirectly solve P_2 in polynomial time as well via the transformation. This argument forms the basis of a proof that P_1 is also NP-hard. We employ a similar tactic in Chapter 4 to prove the complexity of the thief orienteering problem.

The complexity class APX is the set of problems in NP for which there are efficient algorithms that can produce solutions that are within a constant factor of the optimal solutions. Since many important problems belong to the complexity class NP-hard, researchers often design *approximation algorithms* to at least be able to compute approximation solutions within a reasonable timeframe.

1.1.3 Approximation Algorithms

An α-approximation algorithm for some optimization problem is a polynomial-time algorithm that, for all instances of the problem, produces a solution whose value is within a factor of α of the value of an optimal solution; careful and detailed analysis is typically needed to show that an approximation algorithm indeed has this worst-case performance. In this situation, we say that the α-approximation algorithm has an *approximation ratio* of α. A common convention is that $\alpha > 1$ for minimization problems and $\alpha < 1$ for maximization problems.

In this thesis, we often characterize the optimal solution for a problem on instance I as $OPT(I)$, and we let $SOL(I)$ be a solution output by an approximation algorithm for the same instance I. Hence, the approximation ratio is computed for minimization problems using the ratio $\alpha = \frac{SOL(I)}{OPT(I)}$ and computed for maximization problems using the ratio $\alpha = \frac{OPT(I)}{SOL(I)}$.

It is important to provide the distinction between *heuristics* and approximation algorithms. Heuristics are algorithms that typically produce approximate solutions but do not provide approximation ratios; these algorithms are usually studied empirically using experimental evaluation of the running times of the algorithm on a variety of inputs. Heuristics might not have a corresponding approximation ratio due to either the underlying strategy being not well understood or the approach being very complicated to analyze; however, if an approximation ratio is proven for a heuristic algorithm, that algorithm is then considered to be an approximation algorithm. The design and analysis of approximation algorithms includes mathematical rigor to provide performance guarentees on the algorithms. This thesis mainly is concerned with the design and analysis of approximation algorithms.

A *polynomial-time approximation scheme* (PTAS) is an algorithm that accepts as input a parameter $\epsilon > 0$ and produces solutions such that $\alpha = (1 + \epsilon)$ (for minimization problems) or $\alpha = (1 - \epsilon)$ (for maximization problems). When the running time of an approximation scheme is polynomial in $\frac{1}{\epsilon}$, it is called a *fully polynomial-time approximation scheme* (FPTAS).

Asymptotic analysis describes the performance of an approximation algorithm as the value of an optimal solution becomes very large. Many approximation ratios have the form $A + \frac{C}{OPT(I)}$, where A is a factor not dependent on $OPT(I)$ called the *asymptotic approximation ratio* and C is an *additive constant*. When $OPT(I)$ is small, the additive constant of the approximation ratio might be significant; however, in the asymptotic setting, additive constants are insignificant.

1.2 Motivation

1.2.1 Modeling Industrial Challenges

Each of our research objectives stems from a real industrial challenge that can be modeled using the problem formulations that we study.

In Chapter 3, we study algorithm design techniques for solving problems in which the input can be grouped into a small number of classes. Many industrial packing and scheduling problems have a small number of distinct object classes compared to the total number of objects. For example, consider the scenario of a large retail business transporting additional stock of its most popular products. Problem formulations that group the input model important industrial challenges and hence designing algorithms for these problems is of great value.

In Chapters 4 and 5, we study algorithm design techniques for solving optimization problems that contain multiple interdependent sub-problems. A problem is said to have two or more interdependent sub-problems if the solution for one sub-problem influences the quality of the solution for the other sub-problems [17]. Many problems in manufacturing or supply chains have multiple constraints, layers of complexity, and/or contain a combination of interdependent sub-problems that are not reflected in any of the classical formulations of the closest related optimization problems. In some cases, these constraints can be so critical as to significantly alter the approach taken to solve the problem [88]. These problems need to be considered as a whole – without optimizing their sub-problems individually. However, most classical problems do not include many problems with interdependent sub-problems. There is a gap between current optimization research problems and their real-world applications, and so there is a strong motivation to research optimization problem formulations that can model practical problems of modern importance.

In Chapter 6, we apply algorithm design techniques using neural networks with few layers to efficiently solve optimization problems. Machine Learning techniques can be applied to metaheuristics to improve their performance in solving tasks without being explicitly programmed for each individual optimization problem by helping to select the best subordinate heuristic, helping to tune the parameters, and helping to evolve the solutions [98]. The fields of Communications and Signal Processing in particular have seen great success in applying machine learning techniques to overcome the randomness in real-life datasets that are too large for humans to find meaningful patterns [41].

1.2.2 New Algorithm Design Techniques

Beyond the industrial applications of our research, we also aim to contribute new algorithm design techniques.

Algorithms that assume the input contains a small number of item classes can apply algorithmic techniques that produce solutions that are often of better quality as compared to the solutions produced by algorithms for the non-high multiplicity versions of the problems. For example, in the Bin Packing Problem (BPP) the goal is to pack the input items into a minimum number of bins. The best approximation ratio for BPP is 3/2 unless $\mathsf{P} = \mathsf{NP}$; however, an algorithm that assumes the input of BPP can be partitioned into K item classes solves BPP exactly in polynomial time [69].

Many of the classical formulations of NP-hard optimization problems, which are problems that have many feasible solutions but require an exceedingly large number of computational resources to compute an optimal solution, were studied in restricted formats to simplify the complexity of the problems. Problems were simplified by removing difficult to satisfy constraints or by adding restrictive constraints that make the problems easier. Research was conducted this way because these optimization problems are very difficult; however, this decision led to the design of algorithms that were not able to readily model complex industrial challenges. Over the last 50 years the design and analysis of algorithms for NP-hard optimization problems has grown and hence a recent trend has involved studying optimization problems with a larger number of difficult to satisfy constraints.

1.3 Algorithm Design Techniques

As described above, researchers often want to design algorithms that can produce fast and accurate feasible solutions to optimization problems. Below we list some core design techniques that appear in the design of many approximation algorithms; in fact, we make use of almost all of these techniques in the rest of this thesis.

1.3.1 Greedy

Greedy algorithms make a sequence of decisions in order to build up a solution, where each decision is intended to be the optimal choice at that particular moment (and without regard to future decisions). Greedy algorithms are often easy to design and have low running times in comparison to other algorithms designed for the same problem. For these reasons, it is not a bad idea when designing algorithms for a new problem to begin with a greedy approach. In the worst case, this initial algorithm design can serve as a benchmark for comparison of future algorithm designs, and in the best case, the greedy algorithm might actually perform quite well.

The problem with greedy algorithms is that making a series of choices that are optimal in-the-moment often leads to getting stuck in *local optimal solutions*. These local optima represent optimal solutions within a neighboring set of candidate solutions, but they are not necessarily a *global optimal solution* (a solution with the best value).

1.3.2 Local Search

Local search algorithms transition from feasible solution to feasible solution by making a small (or local) change to the solution, sometimes referred to as a *swap operation*. Determining which part of the solution is to be swapped out, and what part of the solution space is to be swapped in, can differ drastically from one algorithm design to the next, but the general principles of this approach suggest that the swap operation makes only a small change to the solution in order to improve its quality. Similar to what was described for greedy approaches, local search algorithms often get stuck in local optimal solutions.

The main difference between greedy and local search algorithms are that greedy approaches start with an empty solution and gradually build up to a feasible solution; in contrast, local search algorithms start with a feasible solution and are only permitted to transition to other feasible solutions. Depending on the design of the local search algorithm, the running time can be quite large; hence, great care must be taken to design polynomial-time local search algorithms.

1.3.3 Dynamic Programming

Dynamic programming algorithms divide problems into subproblems, store optimal solutions to these subproblems in a table, and then piece together the optimal solutions to the subproblems in order to find an optimal solution to the original problem.

When applying a dynamic programming approach to a problem with a *overlapping subproblem* characteristic, identified by finding the same subproblems over and over again as part of the original problem (and performing repetitive work to solve those repeating subproblems), a time-memory trade-off is achieved: at the expense of writing information down in a table (using storage space), time can be saved by looking up previously calculated results stored in the table instead of repeating those calculations.

When applying a dynamic programming approach to a problem without a *optimal substructure* characteristic, identified when the optimal solution to a subproblem is part of an optimal solution to the original problem, infeasible results can be returned; therefore, dynamic programming algorithms should only be used on optimization problems that exhibit optimal substructure.

1.3.4 Linear Program Rounding

A *linear program* can be expressed in the general form:

$$\begin{array}{lr} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b, \\ \text{and} & x \geq 0 \end{array}$$

where x is a vector of variables, c and b are vectors of coefficients, and A is a matrix of coefficients. The expression $c^T x$ is known as the objective function. We can use a linear program to solve minimization problems by negating the objective function. The inequalities $Ax \leq b$ and $x \geq 0$ are constraints on the mathematical model.

An *integer program* is a linear program where all the variables are restricted to be integers. The problem of finding optimal solutions for integer programs is NP-hard [100]; however, solutions for linear programs can be computed in polynomial time [105].

Linear program rounding (LP-rounding) is described by the following process:

1. Formulate an NP-hard problem as an integer program and relax the integrality constraints of the integer program to obtain a linear program
2. Solve the linear program
3. Transform the solution of the linear program into a feasible solution for the NP-hard problem by rounding the values of non-integral variables to integer values

Some of the design decisions in a LP-rounding algorithm include the precise details of formulating the integer and linear programs, and whether the transformation to a feasible solution is performed randomly or using deterministic decisions.

1.4 Contributions

The contributions of this thesis apply several of the algorithm design techniques described in Section 1.3 to investigate the research objectives proposed in Chapter 1.

1.4.1 High Multiplicity Strip Packing

The two-dimensional strip packing problem consists of packing in a rectangular strip of width 1 and minimum height a set of n rectangles, where each rectangle has width $0 < w \leq 1$ and height $0 < h \leq h_{max}$. We consider the high-multiplicity version of the problem (HMSP) in which there are only K different types of rectangles.

HMSP can be relaxed to the two-dimensional fractional strip packing problem, which permits horizontal cuts on the rectangles, and can be represented using the configuration linear program, which is a technique that uses a variable for each possible multiset of rectangles whose total width fits in the rectangular strip. We use the algorithm of Karmarkar and Karp [99] to compute a basic feasible solution to the linear program in polynomial time of height at most $LIN + \epsilon$, where LIN is the height of an optimal fractional packing and $\epsilon > 0$. Basic feasible solutions to this linear program consist of a set of at most K configurations stacked on top of one another, and a simple algorithm can round the fractional rectangles in a basic feasible solution to whole rectangles to produce solutions of height at most $LIN + K + \epsilon$.

Our algorithm takes as input the fractional packing computed by the linear program and applies different rounding techniques to the fractional rectangles depending on whether their heights are small or large. Large fractional rectangles can be made whole within further increasing the height of the packing by a large amount but small fractional rectangles need to be merged together and packed elsewhere in the packing.

For the case when $K = 3$, we give an algorithm that produces solutions requiring at most height $\frac{3}{2}h_{max} + \epsilon$ plus the height of an optimal solution, where ϵ is any positive constant. For the case when $K = 4$, we give an algorithm yielding solutions of height at most $\frac{7}{3}h_{max} + \epsilon$ plus the height of an optimal solution. For the case when $K > 3$, we give an algorithm that gives solutions of height at most $\lfloor \frac{3}{4}K \rfloor + 1 + \epsilon$ plus the height of an optimal solution.

1.4.2 Thief Orienteering on Directed Graphs

We consider the scenario of routing an agent called a *thief* through a weighted graph $G = (V, E)$ from a start vertex s to an end vertex t. A set I of items each with weight w_i and profit p_i is distributed among $V \setminus \{s, t\}$. The thief, who has a knapsack of capacity W, must follow a simple path from s to t within a given time T while packing in the knapsack a set of items taken from the vertices along the path of total weight at most W and maximum profit. The travel time across an edge depends on the edge length and current knapsack load. The thief orienteering problem (ThOP) is a generalization of the orienteering problem, the longest path problem, and the 0-1 knapsack problem. We prove that there exists no approximation algorithm for ThOP with constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$.

We adapt a classic dynamic programming approach for the 0-1 knapsack problem that builds a profit table containing the weights and profits of the item subsets so that it can represent the path along the graph G from which the items were collected and it can store the travel times needed to collect each item subset. For directed and acyclic graphs we can visit the vertices in topological order to ensure that we only update the profit table a single time for each vertex. To keep the size of the profit table small, the parameters of the problem (weight, profit, and traveling time) need to be rounded carefully to ensure that the our algorithm produces solutions close to optimal ones.

We present a polynomial-time approximation scheme (PTAS) for ThOP when G is directed and acyclic that produces solutions that use time at most $T(1 + \epsilon)$ for any constant $\epsilon > 0$. We also present a fully polynomial-time approximation scheme (FPTAS) for ThOP on arbitrary undirected graphs where the travel time depends only on the lengths of the edges and T is the length of a shortest path from s to t plus a constant K. Finally, we present a FPTAS for a restricted version of the problem where the input graph is a clique.

1.4.3 Thief Orienteering on Undirected Graphs and Special Graph Classes

A *planar* graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. An *outerplanar* graph is a planar graph for which all of its vertices belong to the outer face.

A *series-parallel* graph $G = (V, E, t_1, t_2)$ has two terminal vertices t_1, t_2 and is defined inductively:

- $G = (\{t_1, t_2\}, (t_1, t_2), t_1, t_2)$ is series-parallel.
- if $G_1 = (V_1, E_1, t_1, t_2)$ and $G_2 = (V_2, E_2, t'_1, t'_2)$ are series-parallel then the *series composition* $G = (V_1 \cup V_2, E_1 \cup E_2, t_1, t'_2)$ is series-parallel if $t_2 = t'_1$.
- if $G_1 = (V_1, E_1, t_1, t_2)$ and $G_2 = (V_2, E_2, t'_1, t'_2)$ are series-parallel then the *parallel composition* $G = (V_1 \cup V_2, E_1 \cup E_2, t_1, t_2)$ is series-parallel if $t_1 = t'_1$ and $t_2 = t'_2$. A graph created from a parallel composition is a *parallel graph*.

We take advantage of the properties of outerplanar and series-parallel graphs in order to transform them into the desired DAGs using a polynomial number of additional vertices and edges such that the DAG has the same set of simple paths between s and t as the original undirected graph. For example, the endpoints of *chords* of an outerplanar graph (the edges that do not belong to the outer face) are quite restricted due to the constraint that the chords cannot intersect each other. Additionally, within a series-parallel graph many of the vertices have degree at most 2; only the vertices that serve as a terminal for some series-parallel subgraph can have degree of 3 or more, and hence directing many of the paths is less complicated.

We give polynomial-time algorithms for transforming instances of the problem on outerplanar and series-parallel graphs into equivalent instances of the thief orienteering problem on directed acyclic graphs; therefore, yielding polynomial-time approximation schemes for the thief orienteering problem on these graph classes that produce solutions using at most time $T(1 + \epsilon)$ for any $\epsilon > 0$.

1.4.4 Modified Hopfield Network for K-Median

The k-median problem (KMP) is a classical clustering problem where given n locations one wants to select k locations such that the total distance between every non-selected location and its nearest selected location is minimized. We present a modified Hopfield network for KMP and experimentally evaluate it against several neural networks and local search algorithms.

By designing a new neuron update function, our algorithm uses a local search approach; however, instead of randomly selecting facilities and clients to swap, our algorithm uses a metric called *inner value* to determine which facilities are the least valuable to a solution and should be swapped out and which clients are the most valuable to a solution and should become facilities. We show that using this metric allows our network to produce accurate solutions very quickly. Moreover, we show how we can adapt our algorithm so that multiple facilities could be swapped out in a single swap operation.

1.4.5 Awards, Publications, and Preprints

The work in this thesis is directly related to the receipt of an NSERC award and several manuscripts, listed below.

Awards

Alexander Graham Bell Canada Graduate Scholarship - Doctoral

Publications

Bloch-Hansen, A., Solis-Oba, R. (2024). The thief orienteering problem on series-parallel graphs. *Proceedings of the 8th International Symposium on Combinatorial Optimization (ISCO)* (pp. 248-262), https://doi.org/10.1007/978-3-031-60924-4_19.

Bloch-Hansen, A., Page, D. R., Solis-Oba, R. (2023). A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs. *Proceedings of the 34th International Workshop on Combinatorial Algorithms (IWOCA)* (pp. 87-98), https://doi.org/10.1007/978-3-031-34347-6_8.

Bloch-Hansen, A., Solis-Oba, R., Yu, A. (2022). High multiplicity strip packing with three rectangle types. *Proceedings of the 7th International Symposium on Combinatorial Optimization (ISCO)* (pp. 215-227), https://doi.org/10.1007/978-3-031-18530-4_16.

Preprints

Bloch-Hansen, A., Rossiter, C., Solis-Oba, R. (2024). A modified hopfield network for the k-median problem. *In preparation*.

Bloch-Hansen, A., Solis-Oba, R. (2024). The thief orienteering problem on series-parallel graphs. *Under review at the Journal Acta Informatica*, https://doi.org/10.21203/rs.3.rs-4208548/v1.

Bloch-Hansen, A., Page, D. R., Solis-Oba, R. (2023). Algorithms for the thief orienteering problem on directed acyclic graphs. *Under review at the Journal of Theoretical Computer Science*.

Bloch-Hansen, A., Solis-Oba, R., Yu, A. (2022). High multiplicity strip packing with three rectangle types. *Under review at the Journal Theory of Computing Systems*, https://doi.org/10.21203/rs.3.rs
1976126/v1.

Chapter 2

2 Literature Review

In this chapter we provide additional literature review (beyond what is included in the papers) for the four contributions of the thesis.

2.1 High Multiplicity Strip Packing

2.1.1 Introduction

Packing problems typically involve maximizing the number of objects that can be placed into a container or minimizing the number of containers needed to hold a set of objects. Many industrial problems can be modeled as packing problems involving rectangles and squares. Solutions for rectangle packing problems are useful, for example, for loading pallets and shipping containers for storage and transport, for designing transistor layouts for computer chips, and for optimizing workflow in a workplace.

There is a quantifiable difference between good and bad solutions in the industrial environment. Wasted space during storage and transport costs companies resources: time might need to be spent re-packing containers, additional containers might need to be shipped, or product might be lost if certain weight restrictions are not satisfied. Many companies still use trial-and-error approaches while packing items for storage and transport.

High multiplicity problems have their input partitioned into relatively few groups that consist of identical objects. High multiplicity problems are important because the number of distinct object types is typically small in practice. Additionally, algorithms for high multiplicity problems generally produce solutions closer to optimal than algorithms for non high multiplicity problems.

In Chapter 3, we consider the *two-dimensional high multiplicity strip packing problem* (HMSP): given K distinct rectangle types, where each rectangle type T_i has n_i rectangles each with width $0 < w_i \leq 1$ and height $0 < h_i \leq 1$, the goal is to pack these rectangles into a strip of width 1 without rotating or overlapping the rectangles such that the total height of the packing is minimized. In this thesis we focus on the cases when there are three distinct rectangle types and when there are four distinct rectangle types.

Very little research has been done on HMSP; however, we find HMSP important and worth studying because the non high multiplicity version of the problem, the two-dimensional strip packing problem, has been researched extensively and finds applications in areas such as wood and glass cutting, storage and transportation, and paging of articles in newspapers [117]. Moreover, we note that some other famous packing problems, such as the bin packing problem, has had its high multiplicity variant (the cutting stock problem) studied extensively.

Efficient algorithms for packing problems translate into improved industrial practices that reduce company expenses and improve product delivery. Furthermore, algorithm design techniques specifically developed for rectangle and square packing problems have contributed to the design of efficient algorithms for other types of optimization problems. As research into packing problems expands, more industrial problems will be solved using these algorithms and more packing and optimization problems will be able to leverage the algorithmic techniques that are discovered.

Packing problems come in many shapes and sizes, and small variations in the definition of a packing problem can produce seemingly similar but many times very different problems. We review the literature on several classic packing problems: the bin packing problem, the cutting stock problem, the rectangle packing problem, and the strip packing problem. The algorithm design and analytical techniques used on these related packing problems both inform the approaches used in this thesis and also help direct our future research directions.

The rectangle packing and strip packing problems are very similar to HMSP; each of these problems involves packing rectangles into a rectangular container. The bin packing and cutting stock problems are fundamental problems in optimization, and the relationship between them is similar to the relationship between the strip packing problem and HMSP: the latter problem is a high multiplicity version of the former. We describe the term high multiplicity in more detail later on in this section.

2.1.2 The Bin Packing Problem

The *bin packing problem* (BP) (see Figure 2.1) is one of the earliest packing problems considered in the literature: the problem is NP-hard [60] and no approximation algorithm for it can have approximation ratio smaller than $\frac{3}{2}$ unless P = NP. It has the following definition:

Definition Given a set $A = \{a_1, a_2, ..., a_n\}$ of n items, each of size $0 < size(a_i) \leq 1$, pack all the items into the smallest possible number of unit capacity bins.

A practical application of this problem encodes transport trucks as bins and products as items. In this model, the size of an item is the product's weight, and the capacity of a bin is the truck's carrying capacity. By minimizing the total number of bins needed to hold all the items, we also minimize the number of transport trucks needed to carry all of the product; therefore, algorithms that minimize the number of required bins can help a transport company minimize its delivery expenses.

The bin packing problem is theoretically significant: as one of the earlier NP-hard problems studied many of the approaches that are used to evaluate the performance of approximation algorithms were first designed for algorithms for the bin packing problem [35].

Over the many years of research on BP, a large variety of algorithms have been designed (see Table 2.1). Some of the most important ones are the following:

- **First-Fit.** The items are packed from the input one at a time into the first bin in which they fit.
- **Best-Fit.** The items are packed from the input one at a time into the best bin in terms of maximizing the space used in the bins.
- **First-Fit Decreasing.** Same as First-Fit but the items are first sorted in non-increasing order.
- **Best-Fit Decreasing.** Same as Best-Fit but the items are first sorted in non-increasing order.

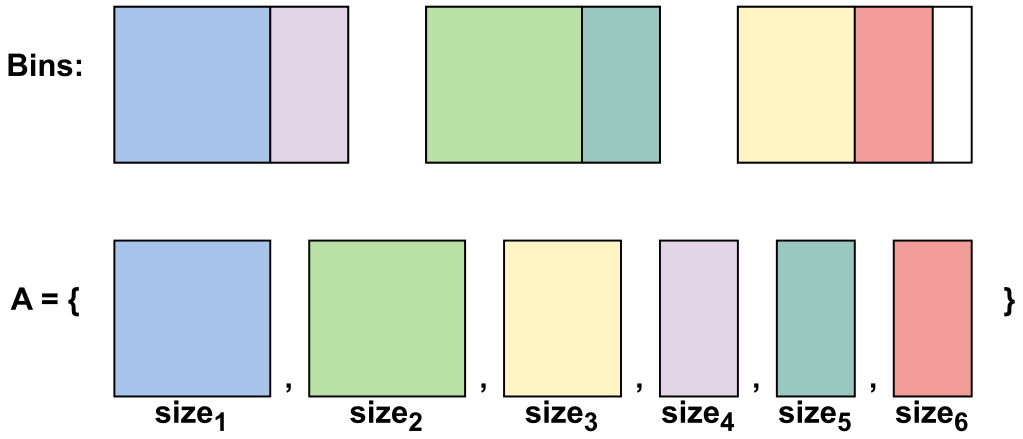


Figure 2.1: The bin packing problem: given a set $A = \{a_1, a_2, ..., a_n\}$ of n items, pack them into the smallest possible number of bins.

- **Refined First-Fit Decreasing.** The items are partitioned into groups depending on their sizes. Only items within the same group are packed into the same bins together, but the packing still follows the rules of First-Fit Decreasing.
- **Best Two-Fit.** The items are packed according to First-Fit Decreasing. Then, if a bin contains more than a single item, the algorithm checks if the smallest item in the bin can be replaced by two un-packed items in order to fill the bin more fully. The two items with the largest size that can replace the single item are chosen.

There are several common techniques used in these algorithms, such as sorting the items, partitioning items into groups based on their size, and using different algorithms for different item groups. These techniques are widely used on many other packing problems.

Approximation Algorithms		
Year	**Authors**	**Approximation Ratio**
1974	Johnson, Demers, Ullman, Garey, Graham [95]	$\frac{3}{2}$
Asymptotic Approximation Algorithms		
Year	**Authors**	**Approximation Ratio**
1980	Yao [174]	$\frac{11}{9} - \frac{10^{-7}}{OPT(I)}$
1985	Johnson and Garey [96]	$\frac{71}{60}$
1991	Friesen and Langston [57]	$\frac{71}{60}$
Polynomial Time Approximation Schemes		
Year	**Authors**	**Approximation Ratio**
1981	De la Vega and Lueker [166]	$1 + \epsilon$
1982	Karmarkar and Karp [99]	$1 + O(\frac{\log^2 OPT(I)}{OPT(I)})$
2013	Rothvoß [146]	$1 + O(\frac{\log OPT(I) * \log\log OPT(I)}{OPT(I)})$

Table 2.1: Significant algorithms, and their performances, for the bin packing problem.

In addition to the classic bin packing problem, there are several variants of the problem that have been proposed. These variants change the definition of the problem by allowing rotations, allowing overlaps, allowing the dimensions of the bins to be adjusted, and packing irregularly shaped objects, to name a few. In many instances, these variants have applications in niche situations. Since the algorithm of Johnson et al. [95] has matched the theoretical bound on the best possible (unless $\mathsf{P} = \mathsf{NP}$) approximation ratio of $\frac{3}{2}$, research either works to improve the running time of these approximation algorithms, the running time of exact algorithms, or the performance of algorithms for problem variants.

2.1.3 The Cutting Stock Problem

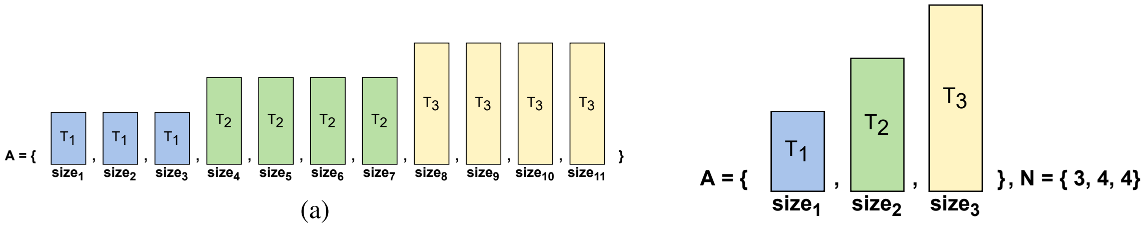
The *cutting stock problem* (CS) first appeared in the literature under the name of the *trim problem* in 1957 [48] and some of the first work on solving this problem was done by Gilmore and Gomory [66–68] who formulated this problem as an integer program. The cutting stock problem is known to be NP-hard and no approximation algorithm for it can have approximation ratio smaller than $\frac{3}{2}$ unless $\mathsf{P} = \mathsf{NP}$ [90]. It has the following definition:

Definition Given a set $A = \{a_1, a_2, ..., a_K\}$ of K item types and a set $N = \{n_1, n_2, ..., n_K\}$ of K item multiplicities, where all n_i items of type a_i have size $0 < size(a_i) \leq 1$, pack all the items into the smallest possible number of unit capacity bins.

This problem can model many industrial challenges of cutting raw materials into smaller units for customers. For example, a business might stock standard length pieces of wood, but customers might request non-standard lengths of wood. The business wants to minimize the cost of cutting their standard material to the requested size. A solution to the cutting stock problem is equivalent to selecting the minimum number of standard-length materials needed to fulfill the customers orders.

Note that the cutting stock problem is equal to the bin packing problem where the input objects are partitioned into K types. However, there is a significant difference between the cutting stock problem and the bin packing problem; while the input to the bin packing problem consists of n item sizes, the input to the cutting stock problem consists of only K item sizes and K item multiplicities. Thus, when the value of n is large and the value of K is small the size of the input for the cutting stock problem is small compared to the size of the input for bin packing (see Figure 2.2). This is a very important observation as we wish to design algorithms whose running times are polynomial in the size of the input.

Recall that an algorithm is efficient if its time complexity is polynomial in its input size. Algorithms for the bin packing problem that consider individually each item would not run in polynomial time if applied to the cutting stock problem. Bin packing algorithms are given n items as input, so considering each item individually incurs a number of operations that depends on the value of n, the size of the input; however, cutting stock algorithms are given K item types as input, so when the total number of items is larger than K, considering each item individually incurs a number of operations that depends on the value of n, but a polynomial time cutting stock algorithm must incur a number of operations that polynomially depends on the value of K, the size of the input. When n is much larger than K, the time complexities of bin packing algorithms applied to the cutting stock problem can be


A = {
T1, T1, T1, T2, T2, T2, T2, T3, T3, T3, T3
}
size1 size2 size3 size4 size5 size6 size7 size8 size9 size10 size11
A = {
T1, T2, T3
}, N = { 3, 4, 4}
size1 size2 size3

(a)

(b)

Figure 2.2: a) In the bin packing problem n sizes of items must be given in the input. b) In the cutting stock problem only K sizes of items and K multiplicities must be given in the input. Even if the value of n is very large, if the value of K is small the size of the input for the cutting stock problem is small.

exponential in their input sizes. Polynomial time algorithms for the cutting stock problem must consider each object type and not individual objects.

In 1985, Marcotte [121] showed that when $K = 2$ the cutting stock problem has a particular property called *integer round-up*. To have the integer round-up property, the optimal value of an integer programming formulation of the problem must be the least integer greater than or equal to an optimal value of its linear programming relaxation. Orlin [133] used this property to design a polynomial time algorithm that solves cutting stock when $K = 2$ exactly. Mcormick et al. [123] later improved upon the running time of the algorithm to $O(\log^2 \beta \log n)$. However, algorithms for cutting stock when $K > 2$ proved to be more difficult and Table 2.2 shows the progression over the years.

Fixed K Algorithms		
Year	**Authors**	**Additive Constant**
2005	Filippi and Agnetis [53]	$K - 2$
2007	Filippi and Agnetis [52]	$1 \ (2 < K \leq 6)$
2007	Filippi and Agnetis [52]	$1 + \lfloor \frac{K-1}{3} \rfloor \ (K > 6)$
2010	Jansen and Solis-Oba [90]	1
2013	Goemans and Rothvoß [69]	0
Not Fixed K Algorithms		
Year	**Authors**	**Additive Constant**
1982	Karmarkar and Karp [99]	$O(\log^2 K)$
2013	Rothvoß [146]	$O(\log K * \log\log K)$

Table 2.2: Significant algorithms, and their performances, for the cutting stock problem.

Most algorithms for CS have been linear program based algorithms that extend the work of Gilmore and Gomory [66–68]. Therefore, most of the techniques proposed have been related to transforming fractional solutions into integer solutions while minimizing the increase to the number of bins that this causes. Since the problem has been solved for the case when K is constant, future work attempts to improve the running time of these algorithms, design algorithms for arbitrary values of K, or focus on variants of the problem.

2.1.4 The Rectangle Packing Problem

The *rectangle packing problem* (RP) (see Figure 2.3) is NP-hard [5] and has been studied since the early 1980's. It has the following definition:

Definition Given a set $A = \{a_1, a_2, ..., a_n\}$ of n rectangles, where each rectangle a_i has width w_i and height h_i, and a rectangular container of width W and height H, pack the maximum subset of rectangles from A into the rectangular container, without rotating or overlapping any of the rectangles.

RP is very useful in modeling the problem of cutting rectangular patterns out of raw material. Consider a rectangular piece of fabric of width W and height H, and a set $A = \{a_1, a_2, ..., a_n\}$ of n rectangular cutting patterns, where each cutting pattern a_i has width w_i and height h_i. Solving the rectangle packing problem is equivalent to maximizing the number of patterns that can be cut from the rectangular piece of fabric, which in turn leads to a cost-effective strategy of cutting the fabric.

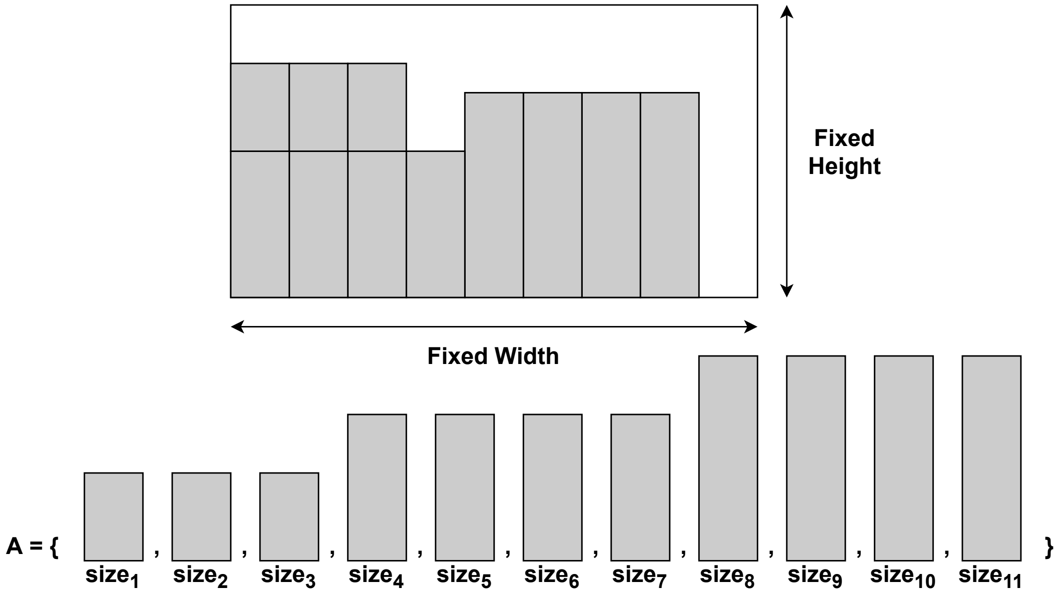


Figure 2.3: The rectangle packing problem: given a set $A = \{a_1, a_2, ..., a_n\}$ of n items, pack the maxmimum subset of them into a rectangular container.

Similarly to the bin packing problem, algorithms such as FFD can be modified to work on RP. When FFD is applied to the two-dimensional rectangle packing problem, it is considered to be a *level oriented* algorithm. In a level oriented algorithm, the bottom of the rectangular container is considered the first level of the packing. Rectangles are packed in the first level until a rectangle r is found that is too wide to be packed in the remaining space at the bottom of the container. The bottom of the second level of the packing is defined by a horizontal line drawn from the top of the tallest rectangle packed in the first level. Subsequent levels are defined in the same way. When FFD is applied to the two-dimensional rectangle packing problem, it has an approximation ratio of 1.7 [31].

Surprisingly, much of the work done on RP focuses on its variations, such as packing rectangles with profits or packing squares (see Table 2.3). Moreover, many of these algorithms approximate very close to optimal solutions and quite a few PTAS have been presented.

Approximation Algorithms		
Year	**Authors**	**Approximation Ratio**
1983	Baker et al. [5]	$\frac{4}{3}$ (unit-weight squares)
2004	Caprara and Monaci [22]	$3 + \epsilon$ (rectangles with profits)
2007	Jansen and Zhang [91]	$2 + \epsilon$ (rectangles with profits)
2009	Harren [78]	$\frac{5}{4} + \epsilon$ (squares with profits)
Polynomial Time Approximation Schemes		
Year	**Authors**	**Approximation Ratio**
2005	Fishkin et al. [56]	$1 + \epsilon$ (rectangles with profits into square container)
2005	Fishkin et al. [56]	$1 + \epsilon$ (squares with profits equal to area)
2005	Fishkin et al. [55]	$1 + \epsilon$ (squares with augmentation)
2008	Jansen and Solis-Oba [89]	$1 + \epsilon$ (squares with profits)
2012	Lan et al. [108]	$\frac{k^2+3k+2}{k^2}$ (rectangle side length at most $\frac{1}{k}$)
2012	Lan et al. [108]	$1 + \epsilon$ (squares without profits)

Table 2.3: Significant algorithms, and their performances, for the rectangle packing problem.

2.1.5 The Two-Dimensional Strip Packing Problem

The *two-dimensional strip packing problem* (SP) (see Figure 2.4) has the following definition:

Definition Given n rectangles with widths $w_1, w_2, ..., w_n$ and heights $h_1, h_2, ..., h_n$, where $0 < w_i \leq 1$ and $0 < h_i \leq 1$ for $i = 1, 2, ..., n$, the goal is to pack all the rectangles without rotations or overlaps in a rectangular strip of width 1 and minimum height.

SP, while similar to RP, allows the rectangular container to have unbounded height; this means that all items in SP can always be packed, while some items in RP might be left unpacked. Operations researchers have long known that packing problems such as SP apply to industrial or commercial situations in which objects are packed on floors, shelves, wooden pallets, trucks, etc., where two specific dimensions of the objects needed to be optimized. However, the applications of SP go well beyond physically packing objects into containers. As early as the 1960's, Codd [34] defined an application of SP to scheduling memory access for multiprogrammed computer systems, and by the 1970's, Garey and Graham [62] had applied SP to scheduling tasks for multiprocessor computer systems. SP has continued to be an important and well-studied problem with modern applications in areas as diverse as resource allocation, scheduling, manufacturing, and transportation.

When the heights of all rectangles in the input to SP are equal, the problem is equivalent to the bin packing problem; therefore, since the bin packing problem is NP-hard, SP is also NP-hard. Thus, SP can not be approximated with approximation ratio better than $\frac{3}{2}$ unless P = NP.

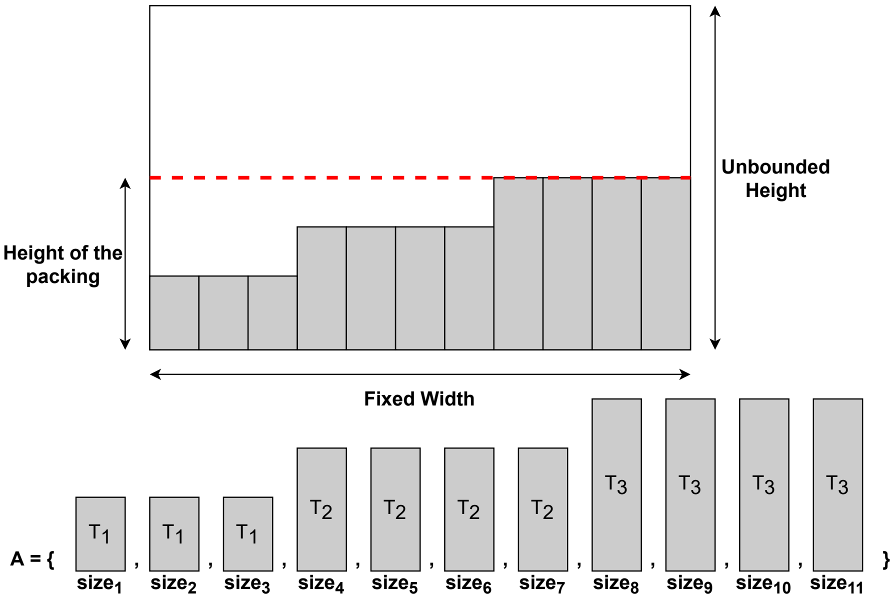


Figure 2.4: The strip packing problem: given n rectangles, pack them all in to a rectangular strip with minimum height.

Algorithms for SP steadily progressed over the years seeing many improvements to their approximation ratios, until recently in 2014 Harren et al. [79] presented the current best-known algorithm whose approximation ratio is $\frac{5}{3}$ (see Table 2.4).

Note that while Harren et al. [79] have the current best-known approximation ratio for SP, and while they have stated that they do not see how to use their recent technique to improve their algorithm, they still believe that it is possible to achieve an approximation ratio of $\frac{3}{2}$. Towards that end, recent researchers have been considering SP in a restricted sense, such as Nadiradze and Wiese's [128] algorithm that computes solutions of height at most $(1.4 + \epsilon)OPT(I)$ in pseudo-polynomial time, Galvez et al.ś [59] algorithm that computes solutions of height at most $(\frac{4}{3} + \epsilon)OPT(I)$ in pseudo-polynomial time, and Jansen and Rau's [92] algorithm that computes solutions of height at most $(\frac{4}{3}+\epsilon)OPT(I)$ in pseudo-polynomial time (but runs significantly faster than Galvez et al.ś algorithm), where $\epsilon > 0$ is a constant. As far as we are aware, these are the cutting-edge algorithms for SP.

As seen in Table 2.4, a large number of papers have been published on SP; however, as seen below, many of these papers recycle old techniques. Here is a list of a few of the most important algorithms:

- **Bottom-Up Left-Justified.** This describes a family of algorithms which all follow the same pattern: rectangles are packed one at a time in the lowest part of the packing that is wide enough to fit them and pushed towards the left wall of the strip as far as possible. The only difference in the algorithms of this family are the order in which the rectangles are packed, with rectangles being sorted by non-increasing width being favoured by Baker et al. [6].
- **First-Fit Decreasing-Height.** This is a *Level-Oriented* algorithm; within a level, all rectangles are packed such that their bottom edge lies on a common line with the

<table>
<tr><th colspan="3">Approximation Algorithms</th></tr>
<tr><th>Year</th><th>Authors</th><th>Approximation Ratio</th></tr>
<tr><td>1980</td><td>Baker et al. [6]</td><td>3</td></tr>
<tr><td>1980</td><td>Coffman et al. [36]</td><td>2.7</td></tr>
<tr><td>1980</td><td>Sleator [156]</td><td>2.5</td></tr>
<tr><td>1994</td><td>Schiermeyer [151]</td><td>2</td></tr>
<tr><td>1997</td><td>Steinberg [157]</td><td>2</td></tr>
<tr><td>2009</td><td>Harren and Van Stee [80]</td><td>1.9396</td></tr>
<tr><td>2014</td><td>Harren et al. [79]</td><td>$\frac{5}{3} + \epsilon$</td></tr>
<tr><th colspan="3">Asymptotic Approximation Algorithms</th></tr>
<tr><th>Year</th><th>Authors</th><th>Approximation Ratio</th></tr>
<tr><td>1980</td><td>Baker et al. [6]</td><td>3</td></tr>
<tr><td>1981</td><td>Coffman et al. [36]</td><td>$\frac{3}{2}$</td></tr>
<tr><td>1981</td><td>Golan [70]</td><td>$\frac{4}{3}$</td></tr>
<tr><td>1981</td><td>Baker and Katseff [4]</td><td>$\frac{5}{4}$</td></tr>
<tr><th colspan="3">Polynomial Time Approximation Schemes</th></tr>
<tr><th>Year</th><th>Authors</th><th>Approximation Ratio</th></tr>
<tr><td>1998</td><td>De la Vega and Zissimopoulos [43]</td><td>$1 + \epsilon$ (no narrow or wide items)</td></tr>
<tr><td>2000</td><td>Kenyon and Remila [104]</td><td>$(1 + \epsilon)OPT(I) + O(\frac{1}{\epsilon^2})$</td></tr>
<tr><td>2009</td><td>Jansen and Solis-Oba [93]</td><td>$(1 + \epsilon)OPT(I) + 1$</td></tr>
</table>

Table 2.4: Significant algorithms, and their performances, for the strip packing problem.

base of the level and all rectangles are pushed towards the left wall of the strip as far as possible. When no more rectangles can be packed within a level, the next level is created with its base at the point of the tallest rectangle in the previous level. In the First-Fit version of the algorithm, rectangles are packed one at a time within the lowest level where it fits. Note that level-oriented algorithms are adapted from bin packing algorithms, as each level can be viewed as a bin.

- **Split Fit.** This algorithm partitions the input rectangles into wide and narrow groups. The wide rectangles are packed first one on top of the other, while the narrow rectangles might be packed beside the wide ones if they fit, or they might be packed above otherwise. This is of the most popular techniques across all of the later algorithms, with some algorithms creating many partitions of the input and using different packing techniques for each group.
- **Reverse-Fit.** This algorithm is similar to Split-Fit in the sense that it packs wide and narrow rectangles differently, but the core technique featured in this algorithm is the idea of packing a level that is left-justified and packing a second level that is right-justified and dropping the second level on the first. This technique is used on rectangles that are all narrow and sorted by non-increasing height; therefore, the right-most rectangles of the second level can frequently fit in the empty space left above the right-most rectangles of the first level.
- **Steinberg's Algorithm.** This algorithm provided an elegant way to use the previous techniques of partitioning the input into groups of differently sized rectangles. Multi-

ple packing techniques are created and the algorithm matches a technique to a group of rectangles if certain constraints are met. Steinberg's algorithm is frequently used as a sub-routine in the state-of-the-art algorithms.

- **BCS Structural Lemma.** This lemma is just as popular as Steinberg's algorithm in the state-of-the-art algorithms, and is used to show that there exists efficient packings that possess a ”nice structure”.

Note that the best-known algorithm by Harren et al. uses Steinbergs's algorithm and the BCS lemma as sub-routines and re-arranges the rectangles to make a slightly better packing.

Proof techniques for many of the algorithms described above often try to prove a similar property: that the packing produced by their algorithm is more than half full. Authors have proven this property in different ways. Baker et al. [6] proved that any arbitrarily drawn horizontal line in their packing intersects less empty space than occupied space. Coffman et al. [36] used a different approach; they bounded the total width of the rectangles on a level including the first rectangle on the next level.

As SP is the closest related packing problem to our own topic, HMSP, it is important to review the algorithmic techniques and proofs of correctness used to address that problem to find inspiration for our own work. Indeed, our own algorithms share several techniques with some of the algorithms listed above, such as sorting rectangles, using different techniques depending on the sizes of the rectangles, and carefully considering the possible structures of the packing.

2.1.6 High Multiplicity Problems

Practical problems that can be solved using packing algorithms often include a small number of different types of items. For example, a company might have a shipment containing only a few different products, but they ship these products in bulk. Therefore, a packing algorithm can group identical products together and pack the groups, instead of packing individual items.

As defined by Hochbaum and Shamir [83], a *high multiplicity* problem has its input ”partitioned into relatively few groups (or types), and in each group all the inputs are identical.” The number of items within a type is called the *multiplicity* of that type. Note that high multiplicity problems represent the input of a problem very compactly, as we only need a list of types and a list of multiplicities and not a list of the individual items in the input. For example, consider the difference between the bin packing and cutting stock problems described above: the bin packing problem takes as input a list of n items, while the cutting stock problem takes as input a list of K types and a list of K multiplicities. Ideally, an algorithm for a high multiplicity problem takes advantage of the fact that there are many items with identical dimensions to find a better solution than a general algorithm that considers every item to be unique.

Each of the problems listed above has high multiplicity variants. For example, as described above, the cutting stock problem is the high multiplicity variant of the bin packing problem. However, we could easily group the input by item types for the rectangle packing and strip packing problems as well. Several approximation algorithms for high multiplicity problems produce solutions closer to the optimal ones than approximation algorithms for their respective general problems.

Even before Hochbaum and Shamir coined the term “high multiplicity”, researchers were exploring problem variants that include groups consisting of identical items. Not all papers describe these problem variants as ”high multiplicity” either, which can make searching for them difficult. For example, in 1980 Psaraftis [140] worked on a job scheduling problem where there were K distinct groups of jobs and jobs within each group are identical. Instead of referring to the problem as a high multiplicity variant of job scheduling, Psaraftis simply calls his problem Sequencing Groups of Identical Jobs. In 1990, Dessouky et al. [45] presented an algorithm for scheduling identical jobs on uniform parallel machines; it is important to note however, that while Dessouky et al. worked on what is essentially a high multiplicity problem, the time complexity of their algorithm was not a polynomial function of a high multiplicity encoding of the input. As we have discussed before in the context of the cutting stock problem, it is challenging to design a polynomial algorithm when the input is so compact.

Hochbaum and Shamir [83] introduced several high multiplicity scheduling problems such as the ”weighted number of tardy jobs problem” and the ”total weighted tardiness problem”, which both accept groups of identical jobs as input. Some high multiplicity problems have been shown to be polynomially solvable, such as the high multiplicity variants of single-machine scheduling with earliness and tardiness [33] and the multiprocessor scheduling problem with 2 job lengths [123], while others have been proven to be NP-hard, such as the high multiplicity variant of minimizing service and operation costs of periodic scheduling [8]. Plenty of research is being conducted on high multiplicity variants of different scheduling problems.

Beyond packing and scheduling problems, high multiplicity variants appear in graph problems as well. In 1982, Lengauer [112] created a compact ”high multiplicity” representation for circuits in very large scale integrated circuitry (VLSI). Galperin and Wigderson [58] expanded this representation to include graphs, which established a new avenue of research in what is essentially high multiplicity graph problems. The compact graph representation, and its application to graph problems, has been advanced through the likes of Lengauer and Wanke [113], Lozano and Balcazar [118], Papadimitriou and Yannakakis [136], and Turan [163], to name a few. An example high multiplicity graph problem would be a variant of the traveling salesman problem where there are only a few cities but they are each visited a large number of times [40].

2.2 Thief Orienteering

2.2.1 Introduction

Network routing problems typically involve selecting a path P from some vertex u to some vertex v in order to minimize objectives such as (*i*) the length of P, (*ii*) the time needed to traverse P, (*iii*) the cost of traversing P, and so on. Examples of problems that can be modeled as network routing problems include planning routes for vehicles to take, routing signals over telephone networks, and planning factory workflows [103].

Using the above examples, poor solutions to network routing problems could result in longer commutes, unstable telephone network coverage, slow internet connections, and wasted money through sub-optimal workflow. To make matters more complicated, network routing problems often involve selecting multiple paths at the same time for different entities to traverse and hence a large number of people or systems could be impacted by these poor solutions.

Multi-objective optimization refers to problems that involve more than one objective function that need to be simultaneously optimized, and often these objectives conflict with each other. In the above example of selecting a path P from u to v, the goal of a multi-objective optimization problem might be to minimize both travel time and cost of traversing P; however, the fastest paths from u to v might incur a high cost.

A problem is said to have two or more *interdependent* sub-problems if the solution for one sub-problem influences the quality of the solution for the other sub-problems [17]. Problems with multiple interdependent sub-problems are important because many real-life problems have multiple constraints, layers of complexity, and/or contain a combination of interdependent sub-problems, and so there is a strong motivation to research optimization problem formulations that can model practical problems of modern importance.

In Chapters 4 and 5, we consider the *thief orienteering problem* (ThOP), which includes the 0-1 knapsack problem and the orienteering problem as sub-problems (we introduce each of these sub-problems further below). Very little research has been done on ThOP; however, we find ThOP important and worth studying because many classical problem formulations do not include multiple interdependent sub-problems, in many cases algorithms for classical problems cannot be re-used for these sub-problems, and yet ThOP can model important industrial challenges.

The aim of this thesis's literature review on ThOP is not to provide an exhaustive list of approaches that have been proposed for this problem; rather, there are some key points that we wish to make: (*i*) introduce the sub-problems of ThOP, (*ii*) describe a particular FPTAS for the 0-1 knapsack problem (*iii*) call to attention the traveling thief problem [17], and (*iv*) compare in detail how our approach in Chapter 4 differs from a FPTAS that was designed for the packing while traveling problem [139].

2.2.2 The 0-1 Knapsack Problem

The *0-1 knapsack problem* (KP) (see Figure 2.5) is a classic NP-hard [101] problem that has been studied since the 1950's and over a thousand papers have been published on it and its related variants [20]. It is also one of the sub-problems included in ThOP. It has the following definition:

Definition Given a set $I = \{i_1, i_2, ..., i_n\}$ of n items, where each item i_j has profit p_j and weight w_j, pack a subset S of the items into a knapsack with capacity W such that $\sum_{i_j \in S} w_j \leq W$ and $\sum_{i_j \in S} p_j$ is maximized.

This problem has many practical applications in storage and transportation, but a less obvious application includes capital budgeting [130]: In this application, the capacity W of the knapsack represents an expenditure limit, and the goal is to select a subset of projects to invest in, where each project has a particular expense (the weight) and also an expected future return (the profit). By selecting a subset of items with maximum profit, we also choose which projects to invest in that maximizes the potential future earnings while not exceeding the current budget.

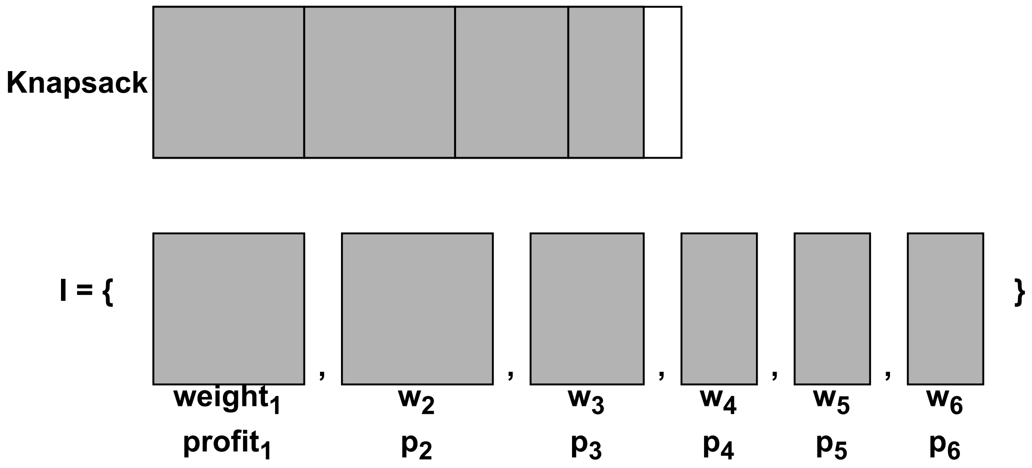


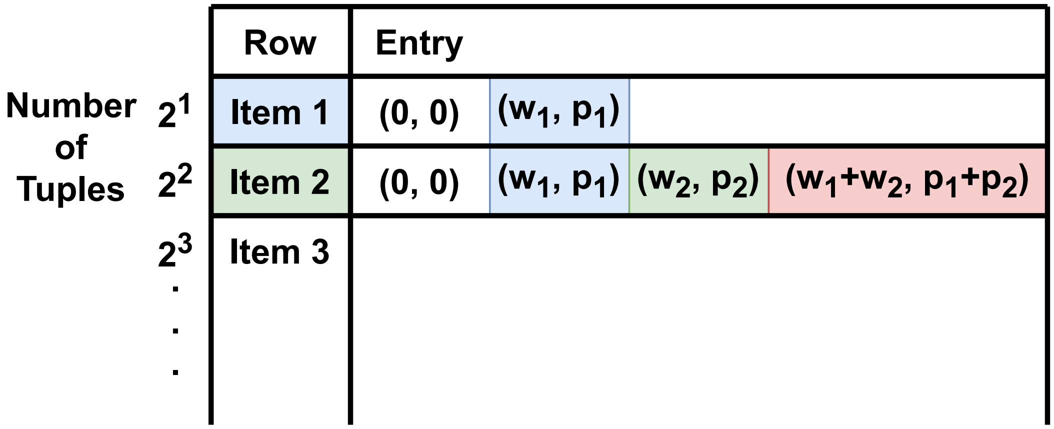
Figure 2.5: The knapsack problem: given a set $I = \{i_1, i_2, ..., i_n\}$ of n items, pack a subset of them with maximum profit into a knapsack.

If the reader is interested in seeing the full body of related work on KP and its variants, we direct the reader to the recent surveys [19, 20]. However, since ThOP specifically includes the 0-1 knapsack problem as a sub-problem, and because in Chapters 4 and 5 we modify a simple FPTAS for KP, we restrict our commentary to fully-polynomial time approximation schemes for KP (see Table 2.5 for some recent results), and in particular to a simple FPTAS that we make use of in Chapters 4 and 5.

Polynomial Time Approximation Schemes		
Year	**Authors**	**Running Time**
1975	Sahni [147]	$n^{O(1/\epsilon)}$
1975	Ibarra and Kim [87]	$O(n \log n + (\frac{1}{\epsilon})^4 \log \frac{1}{\epsilon})$
1979	Lawler [110]	$O(n \log n + (\frac{1}{\epsilon})^4)$
2004	Kellerer and Pferschy [102]	$O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^3 \log^2 \frac{1}{\epsilon})$
2015	Rhee [143]	$O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^{\frac{5}{2}} \log^3 \frac{1}{\epsilon})$ (randomized)
2018	Chan [26]	$O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^{\frac{12}{5}} / 2^{\Omega(\sqrt{\log(\frac{1}{\epsilon})})})$
2019	Jin [94]	$O(n \log \frac{1}{\epsilon} + (\frac{1}{\epsilon})^{\frac{9}{4}} / 2^{\Omega(\sqrt{\log(\frac{1}{\epsilon})})})$
2023	Deng et al. [44]	$\tilde{O}(n + (\frac{1}{\epsilon})^{\frac{11}{5}} / 2^{\Omega(\sqrt{\log(\frac{1}{\epsilon})})})$ (randomized)
2024	Chen et al. [30]	$\tilde{O}(n + (\frac{1}{\epsilon})^2)$

Table 2.5: Significant FPTAS for the knapsack problem. The notation $\tilde{O}()$ is a variation of the big-O that ignores logarithmic factors and Ω defines a lower bound.

In Chapters 4 and 5, we use a very simple FPTAS in order to keep the analysis of our algorithm simple. Williamson and Shmoys [170] describe a very simple dynamic programming approach to KP that builds a table where each entry $A[j]$ is a list of tuples (w, p), for all items $i_j \in I$. A tuple (w, p) in the list of $A[j]$ indicates that there is a subset S of the first j items of I such that the weight of the items in S is $w \leq W$ and the profit of the items in S is p (see Figure 2.6).



Number of Tuples	Row	Entry			
2^1	Item 1	(0, 0)	(w_1, p_1)		
2^2	Item 2	(0, 0)	(w_1, p_1)	(w_2, p_2)	(w_1+w_2, p_1+p_2)
2^3	Item 3				
.					
.					
.					

Figure 2.6: An entry $A[j]$ in the knapsack profit table is a list of tuples (w, p) that are restricted to selecting subsets of only the first j items.

The tuples in the list of entry $A[1]$ of Williamson and Shmoys's profit table include the empty set $(0, 0)$ with no profit or weight and the set containing only the first item (w_1, p_1) with the weight and profit of item i_1. To build the list of tuples in the entry $A[j]$ corresponding to item i_j, for $j = 2, 3, ..., |I|$, first the tuples from entry $A[j-1]$ are copied to entry $A[j]$ and then for each tuple in $A[j-1]$ a new tuple is created and appended to the list at entry $A[j]$ that includes the old tuple plus the item i_j (as long as including i_j does not exceed the carrying capacity W).

A tuple (w, p) *dominates* another tuples (w', p') if $p \geq p'$ and $w \leq w'$. Dominated tuples are removed from each list of A such that no tuple in the list of each entry $A[j]$ dominates another tuple in the same list. Therefore, it is assumed that each list $A[j]$ keeps only one tuple for each unique profit value - the tuple with the lowest weight.

To make a FPTAS, the profit of each item is rounded down to the nearest multiple of $\frac{\epsilon P_{max}}{n}$, where P_{max} is the largest profit among the items. Since dominated tuples are removed, the number of tuples in each entry of the table is at most $O(\frac{n^2}{\epsilon})$ and hence the running time of the algorithm is $O(\frac{n^3}{\epsilon})$.

2.2.3 The Orienteering Problem

The *orienteering problem* (OP) was introduced in 1987 by Golden, Levy, and Vohra [71], based on an outdoor sport where locations are visited along a path from a start point to an end point. Tsiligirides [162] designed several algorithms for this problem under the name of the generalized traveling salesman problem. Golden et al. [71] show that OP is NP-hard by reducing it to the traveling-salesman problem. It has the following definition:

Definition Given a weighted graph $G = (V, E)$ with n vertices, where each vertex u_i has a score s_i and vertex u_1 represents the start vertex and u_n represents the end vertex, select a simple path from the start to end vertices that maximizes the total score but does not exceed the time limit T, where the time it takes to travel over edge (u, v) is the length of (u, v).

A practical application of this problem includes modeling a truck that delivers fuel to a large number of customers on a daily basis [71]. The vertices of the graph represent the customer locations, the edges between vertices represent the cost of traveling between customers, and the time limit represents a day of work. A customer's fuel supply is critical and hence a measure of urgency can be assigned to each customer; therefore, by maximizing the score of the OP, we also visit the most critical customers each day to avoid any customers from running out of fuel.

Since the score of a vertex is independent to the time it takes to travel to that vertex, it can be difficult to select vertices that belong to an optimal solution [63]. The most difficult instances of OP are when an optimal solution includes roughly half of the total number of vertices, as this involves the largest number of feasible solutions [164].

Research on OP includes heuristics, approximation algorithms, and exact algorithms, as well as research into restricted versions of the problem and variants such as the team orienteering problem, the time dependant orienteering problem, and the orienteering problem with time windows. We discuss some of these problem variants in Chapter 7 as potential future directions for ThOP. For a broader list of related work, we direct the reader to recent surveys [73, 165]. Table 2.6 lists several of the most influential algorithms for OP; however, as stated previously our particular approach modifies an algorithm for KP instead of modifying an algorithm for OP, so we do not discuss these algorithms in detail, instead, we simply wish to inform the reader of the problem definition.

2.2.4 The Thief Orienteering Problem

The *thief orienteering problem* (ThOP) was introduced in 2018 by Santos and Chagas [150] by combining the orienteering problem and the 0-1 knapsack problem, which are both NP-hard problems. Later in Chapter 4, we show that ThOP is also NP-hard. ThOP has the following definition:

Definition Given a weighted graph $G = (V, E)$ with n vertices, where two vertices are designated the *start* and *end* vertices, and a set I of items stored in the vertices of V, where each item $i_j \in I$ has a non-negative weight w_j and profit p_j, select a path from the start vertex to the end vertex, while collecting items from each vertex to maximize the profit in the knapsack without exceeding its carrying capacity W or the time limit T. The time needed to travel from city to city depends on the current weight in the knapsack.

A practical application of this problem includes modeling the collection of goods from customers for proper disposal [150]. Each customer product has a benefit, and the vehicle driver has limited storage capacity and limited working hours. Additionally, the vehicle speed slows down as it collects more products. By optimizing the solution to ThOP, we also pick up the products from the customers with the most benefit each day.

Table 2.7 shows all of the related work on ThOP; note that beside this thesis's contributions, the rest of the research has been on heuristics. This is likely due to the difficulty

Exact Algorithms		
Year	**Authors**	**Algorithm Type**
1990	Laporte and Martello [109]	branch-and-bound
1992	Ramesh et al. [141]	branch-and-bound
1994	Leifer and Rosenwein [111]	cutting plane
1998	Fischetti et al. [54]	branch-and-cut
1998	Gendreau et al. [63]	branch-and-cut
Heuristics		
1984	Tsiligirides [162]	monte-carlo and vehicle routing
1987	Golden et al. [71]	centre-of-gravity
1991	Ramesh and Brown [142]	four-phase
1996	Chao et al. [27]	five-phase
1998	Gendreau et al. [64]	tabu search
2001	Tasgetiren [159]	genetic
2002	Liang et al. [116]	ant colony
2009	Schilde et al. [152]	pareto ant colony
2010	Sevkli and Sevilgen [153, 154]	strengthened particle swarm
2012	Liang et al. [115]	multi-level neighborhood search
2014	Camposet al. [21]	greedy randomized adaptive search
2015	Marinakis et al. [122]	memetic-greedy randomized adaptive search
2018	Kobeaga et al. [106]	evolutionary
2019	Santini [149]	adaptive neighbourhood search
2023	He et al. [81]	genetic
Approximiation Algorithms		
1998	Arkin et al. [2]	$(2 + \epsilon)$-approximation (Euclidean)
2003	Blum et al. [15]	4-approximation
2004	Bansal et al. [7]	3-approximation
2007	Blum et al. [16]	PTAS (rooted orienteering)
2008	Chen and Har-Peled [29]	PTAS (fixed-dimensional Euclidean)
2012	Chekuri et al. [28]	$(2 + \epsilon)$-approximation
2022	Gottlieb et al. [72]	PTAS (rooted orienteering)

Table 2.6: Significant algorithms, and their performances, for the orienteering problem.

in designing approximation algorithms for ThOP with guaranteed approximation rations, which we discuss further in Chapter 4. In order for us to design our approximation algorithms, we needed to impose additional restrictions on ThOP such as the input graph being from a particular graph class, or the thief's travel time over an edge being independent of the weight in the knapsack.

Since the approaches used in the heuristic algorithms by other researchers are so drastically different than the approach used in this thesis, we do not think it would be very useful to describe in detail how the heuristics work. Instead, we offer another problem, the packing while traveling problem, that has some overlap in methodology to our own results.

Heuristics		
Year	**Authors**	**Algorithm Type**
2018	Santos and Chagas [150]	local search and genetic
2020	Chagas and Wagner [23]	ant colony
2020	Faêda and Santos [50]	genetic
2022	Chagas and Wagner [24]	swarm inteligence and randomization
2023	Huynh et al. [86]	ant colony and local search
Approximation Algorithms		
2023	Bloch-Hansen et al. [12]	PTAS (DAGs)
2024	Bloch-Hansen and Solis-Oba [13]	PTAS (series-parallel graphs)

Table 2.7: All algorithms for the thief orienteering problem.

2.2.5 The Packing While Traveling Problem

The *packing while traveling problem* (PWTP) was introduced in 2017 by Polyakovskiy and Neumann [139] and was shown to be NP-hard. PWTP is very similar to ThOP, but in PWTP the thief must follow a fixed path. PWTP has the following definition:

Definition Given is a weighted graph $G = (V, E)$, a sequence $N = \{u_1, u_2, ..., u_{n+1}\}$ of unique vertices, and a set I of items, each with weight w_i and profit p_i, distributed among the first n cities. There is a single vehicle that visits the cities of N in sequence and may collect any items when it visits a city, as long as the selected items do not exceed its carrying capacity W. The time needed to travel from vertex to vertex depends on the current weight in the vehicle. The vehicle has a constant rent rate defining how much money needs to be paid per unit time, and so the transportation cost of the selected items is the product of the rent rate and the total traveling time. The goal is to select a subset of I of weight not exceeding W such that the difference between the profit of the selected items and the transportation cost is maximized.

A practical application of this problem includes modeling a supplier that travels a single major route and has to decide whether or not to purchase goods for future sales [139]. Deciding to include goods incurs a transportation cost and affects the total travel time along the route, which may incur addtional expenses if the vehicle was rented. By optimizing the solution to PWTP, we also select the optimal goods to purchase along that single route.

Approximation Algorithms		
Year	**Authors**	**Algorithm Type**
2019	Neumann et al. [131]	FPTAS
Exact Algorithms		
2017	Polyakovskiy and Neumann [139]	mixed-integer and branch-infer-and-bound
Heuristics		
2019	Roostapour et al. [144]	evolutionary

Table 2.8: Significant algorithms, and their performances, for the packing while traveling problem.

As shown in Table 2.8, there is not a lot of related work on PWTP. We wish to highlight the work of Neumann et al. [131] because, out of all of the related work we have discussed so far, this is the most similar to our contributions in Chapter 4. Below we describe how our contributions extend upon and differ from the results of Neumann et al. It might help the reader to first familiarize themselves with Chapter 4 before reading this comparison.

The objective function of PWTP can take on positive and negative values and Neumann et al. [131] show that there is no polynomial-time algorithm with finite approximation ratio for the original objective function of PWTP unless $\mathsf{P} = \mathsf{NP}$. Instead, they consider a different objective function for PWTP in which the objective function is the amount that can be gained over the cost when the vehicle travels empty. Similar to our approach, they first design an exact algorithm using dynamic programming and then they round down the values of the profits of the items to keep the running time polynomial.

Observe a key difference between PWTP and ThOP: in ThOP there is a time limit and any solution that exceeds the time limit is infeasible; in contrast, in PWTP there is no time limit but rather solutions that take more time have a higher transportation cost. This difference is significant when the profits are rounded, because two tuples in the PWTP table that have both been rounded to the same benefit (profit minus transportation cost) are both feasible solutions and since Neumann et al. take the tuple with the least weight they are guaranteed to keep the best tuple. In contrast, for two tuples in ThOP that have been rounded to the same value of profit and weight, it is possible that only one of those tuples represents a feasible solution that satisfies the time limit, and hence an algorithm that discards the wrong tuple can produce solutions that are arbitrarily far below an optimal solution.

A second key difference between PWTP and ThOP concerns rounding the weights of the items. Neumann et al. do not need to round the weights of the items in order to keep the size of their table polynomial: If multiple tuples have the same benefit, they keep the tuple with the lowest weight. In ThOP, when multiple tuples have the same profit, we need to keep one tuple per unique weight: the tuple with the smallest travel time. These additional tuples need to be kept because a solution using the tuple with the smallest weight might exceed the time limit (for example, by carrying the items for a long distance).

Our approach differs from Neumann et al.ś approach in several ways. First, since ThOP does not include a fixed route, our algorithm needs to store information relating to which path the thief takes. Second, since in ThOP the travel time does not directly impact the profit, we need to explicitly track the time needed to carry item subsets along each path, whereas Neumann et al.ś algorithm implicitly includes this information by transforming the travel time into a travel cost and including it with the profit. Finally, the PTAS that we present in Chapter 4 produces solutions that might slightly exceed the time limit in order to guarantee that our algorithm selects enough of the items from an optimal solution in order to get a near maximum profit.

2.2.6 The Traveling Thief Problem

The *traveling thief problem* (TTP) was introduced in 2013 by Bonyadi, Michalewicz, and Barone [17] by combining the traveling-salesman problem and the 0-1 knapsack problem, which are both classic NP-hard problems, and hence TTP is NP-hard. TTP differs from ThOP due to the fact that in TTP every vertex in the graph must be visited. It is not as closely related to ThOP as PWTP was, but we mention in it this section anyways as it is yet another routing problem that includes the 0-1 knapsack problem as a sub-problem. It has the following definition:

Definition Given is a weighted graph $G = (V, E)$ with n vertices, where one vertex is designated the *start* vertex, and a set I of items stored in the vertices of V, where each item $i_j \in I$ has a non-negative weight w_j and profit p_j. There is a thief that begins at the start vertex and visits each vertex exactly once and returns to the start vertex, while collecting items from visited cities whose total weight does not exceed the carrying capacity W of a knapsack. The time needed to travel from city to city depends on the current weight in the knapsack. The knapsack has a constant rent rate defining how much money needs to be paid per unit time, and so the transportation cost of the selected items is the product of the rent rate and the total traveling time. The goal is to select a subset of I of weight not exceeding W such that the difference between the profit of the selected items and the transportation cost is maximized.

As shown in Table 2.9, which lists a sample of the related work, there are many more heuristics designed for TTP than exact algorithms. For further reading, we direct the reader to a survey on the heuristics for TPP [82].

Heuristics		
Year	**Authors**	**Algorithm Type**
2014	Polyakovskiy et al. [138]	local search and evolutionary
2014	Faulkner et al. [51]	local search
2014	Bonyadi et al. [18]	solve sub-problems independently
2016	Mei et al. [124]	co-evolution and memetic
2016	Wagner [167]	ant swarm
2017	Blank et al. [10]	greedy
2018	Wagner et al. [168]	regression, clustering, and classification
2018	Yafrani and Ahiod [49]	hill climbing and simulated annealing
2018	Yafrani et al. [173]	hill climbing
2020	Maity and Das [119]	local search
2021	Myszkowski and Laszczyk [127]	diversity based selection
2022	Nikfarjam et al. [132]	quality diversity and evolutionary
2022	Chagas and Wagner [25]	weighted-sum
Exact Algorithms		
2017	Wu et al. [172]	dynamic and constraint programming

Table 2.9: Significant algorithms, and their performances, for the orienteering problem.

2.3 Hopfield Networks and the K-Median Problem

Machine learning techniques are sometimes integrated into meta-heuristics for solving combinatorial optimization problems. *Metaheuristics* are general-purpose algorithms that can be applied to a large variety of problems, including optimization problems [98], and include a family of methods that guide local improvement procedures while exploring the solution space [65]. These metaheuristics often generate large volumes of data that describe candidate solutions, such as the sequence of search operators (e.g. for a local search algorithm, the initial solution and the sequence of swap operations), the local optima, and other values, and so machine learning techniques are often used to help filter through the large amount of information produced by the metaheuristics.

The use of machine learning techniques described above is in aid to other heuristics; it is also possible that machine learning techniques are used as the primary approach to solving optimization problems. Recent research in communications and signal processing has produced neural networks that perform well for solving optimization problems in signal processing when the data is random, dynamic, and mathematically complex [41], and future research directions have been suggested for applying learning-based techniques to other types of optimization problems.

Neural networks are a type of machine learning technique that consist of processing units called *neurons* that are connected to each other and can accept input and produce output. These neurons are named after the neurons in the brain, and similarly to how our brains work, neural networks aggregate the input and output of many neurons to solve complex tasks. The connections between neurons can be given a weight that influences the strength with which they influence each other, and networks choose the connection weights by a special process called training. Neurons in a network are typically organized into a sequence of *layers*, where each layer receives as its input the output of the previous layer. Many types of neural networks contain a large number of layers hence are quite complicated to analyze.

In Chapter 6 we investigate a special kind of neural network called a *Hopfield neural network* that contains only a single layer; such a network is not very complicated and could be analyzed easier. In contrast to the previously described neural networks that receive input into a layer and produce output, a Hopfield network does not output anything; rather, state values of the neurons correspond to the values of a solution. The transition between states in a Hopfield network is driven by an energy function that is carefully designed for each problem to reflect their particular constraints and carried out through neuron updates. Neurons periodically update their state values based on the state values of the other neurons in the network and the connection weights for each neuron pair. Network states that violate the problem constraints should add large amounts of energy to the network, and hence network states with lower energy correspond to solutions that are closer to an optimal one. A Hopfield network is said to be stabilized when the state values of the neurons remain unchanged during update functions, which occurs when the energy of the network cannot be lowered any further; note that stabilization can happen when the network has only reached a local optimal solution.

In Chapter 6 we present a modified Hopfield network for the k-median problem; our network uses a local search approach, so in the following sections we aim to: (*i*) present existing local search and neural network approaches to the k-median problem and (*ii*) discuss other problems that Hopfield networks have been applied to.

2.3.1 The K-Median Problem

The *k-median problem* (KMP) (see Figure 2.7) is a classic clustering problem and is NP-hard [61]. It has the following definition:

Definition Given a graph $G = (V, E)$ with n vertices and nonnegative length d_{ij} for every edge (i, j), choose k vertices as medians that minimizes the sum of distances from each vertex to its nearest median.

A practical application models the optimal placement of facilities among a collection of clients. In this application, k facilities should be opened in a particular location in order to serve a list of nearby clients. By selecting the medians that minimize the sum of distances from each vertex to its nearest median, we also choose locations to open the facilities so that the total distance from the clients to their nearest facility is minimized.

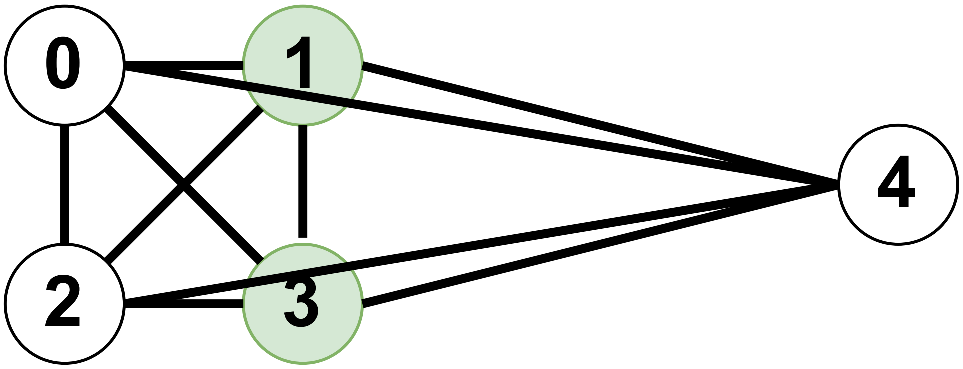


Figure 2.7: An instance of the k-median problem: given a graph $G = (V, E)$ with n vertices and nonnegative length d_{ij} for every edge, choose $k = 2$ vertices as medians that minimizes the sum of distances from each vertex to its nearest median.

Readers interested in seeing related work on KMP are directed to the surveys in [42, 134, 158]. Since in Chapter 6 the algorithm we describe is a combination of neural network and local search algorithm, in this section we restrict our commentary to the most closely related literature (see Table 2.10).

The algorithm of Arya et al. [3] transitions from feasible solution to feasible solution by considering *swap operations*, where each operation swaps a facility with a client by turning the facility into a client and the client into a facility. When the algorithm considers only a single swap at a time (a single facility is swapped with a single client), Arya et al. prove that the approximation ratio is 5, but when the algorithm is allowed to swap p facilities with p clients, Arya et al. prove that the approximation ratio is $3 + \frac{2}{p}$.

Cohen-Addad et al. [37] augment Arya et al.ś multi-swap local search algorithm by changing the objective function: In the new scheme, the sum of distances from each client

Local Search Algorithms		
Year	**Authors**	**Approximation Ratio**
2000	Korupolu et al. [107]	$k(1 + \epsilon)$ (metric k-median)
2001	Arya et al. [3]	5 (single swap) and $3 + \epsilon$ (multi swap)
2006	Pan and Zhu [135]	5
2018	Peng et al. [137]	$3 + \epsilon$
2019	Cohen-Addad et al. [38]	$1 + \epsilon$ (planar graphs)
2022	Cohen-Addad et al. [37]	$2.836 + \epsilon$
2023	Cohen-Addad et al. [39]	$2.671 + \epsilon$
Neural Networks		
2002	Merino and Perez [47]	Hopfield network
2003	Merino et al. [125]	recurrent neural network
2008	Domínguez and Muñoz [46]	recurrent neural network
2012	Shamsipoor et al. [155]	Hopfield network (capacitated k-median)
2016	Mishrra and Barman [126]	Hopfield network
2020	Haralampiev [75–77]	Hopfield network and boltzmann machine
2023	Rossiter [145]	modified Hopfield network

Table 2.10: Local search algorithms and neural networks designed for the k-median problem.

to both its closest facility **and** its second-closest facility is minimized. This new objective can avoid some of the local optima that the previous local search algorithms produced. This work was further improved by carefully selecting an initial solution for the local search algorithm [39].

The algorithm of Pan and Zhu [135], runs the algorithm of Arya et al. p times and constructs a reduced version of the problem that omits any facilities common to the p solutions and any clients served by those facilities. The motivation behind this approach is that the common facilities from even a few local optima have a high likelihood of belonging to a global optimal solution and hence should be kept; furthermore, solving the reduced problem is less complicated and can help move past the local optimal where the algorithm might get stuck. This algorithm outperforms Arya et al.ś algorithm in practice.

On the machine learning side of KMP, many neural networks have been used to help pre-process the input in order to reduce the number of candidate solutions [160], and neural networks have been used to enhance the solutions output by other heuristics [129], but only a couple of neural networks have been specifically designed to solve KMP as a standalone approach.

Domínguez and Muñoz [47] were the first to apply a Hopfield network approach to KMP: They designed a single layer network that was divided into two disjoint groups of neurons, where one group of neurons represented potential facilities and another group represented potential clients. Furthermore, within each group of neurons, they were further divided into sub-groups where only a single neuron within the same sub-group could be active at the same time, which meant that when the network stabilized the solution was always feasible. The network begins with a random solution and then, by updating the values of the neurons and computing the energy of the network it interchanges the active neuron within a group of neurons with the neuron with the maximum input. Their Hopfield network was compared against two other simple algorithms, an interchange algorithm [161] and a random search algorithm, with the Hopfield network producing the best solutions in the smallest amount of time.

Merino et al. [125] proposed three different recurrent neural networks for KMP, all of which were built similarily to [47] in terms of having separate neurons for facilities and clients. In the first approach, they let their neural network solve KMP several times and take the best solution. In the second approach, their neural network is initialized so that all vertices are considered to be facilities and serve themselves. Iteratively, the two facilities that are closest to each other are compared and one of them is deactivated, until there is only *k* facilities active. In the third approach, the neural network contains only a single active facility; iteratively, the client that is furthest from any active facility is turned into a facility, until there are *k* facilities. Like the previous work, the neural networks were compared against the interchange algorithm [161], and the authors concluded that the performance of their three models depended on the size of the network and the value of *k*.

Domínguez and Muñoz [46] proposed several recurrent neural networks to improve upon their previous Hopfield network, and performed experiments on the well-known benchmark instances of the OR Library [9]. They compared their networks against the variable neighborhood search (VNS) [74] and demonstrated results that outperform VNS when the computational time is limited.

Haralampiev [75–77] built a neural network architecture for some facility location problems (including KMP) with the goal of solving them with only minimal parameter tuning. Haralampiev built the network using similar rules as the previous works, dividing the neurons into groups for facilities and clients and enforcing rules on sub-groups to ensure that a client is served by exactly one facility at a time. The main difference in Haralampiev's network is a *temperature* variable that influences whether a neuron is activated or deactivated; by using the temperature variable, the network can explore additional solutions even if they reduce the quality of the solution.

Rossiter's [145] modified Hopfield network is the closest work to that which is presented in Chapter 6. Rossiter's network maintains two sets of values for each neuron, the activation value and the inner value. Rossiter's network is initialized with all facilities receiving activation values that are arbitrarily close to 1 (indicating that each facility is active). The inner values of the client neurons are computed using the product of a facilities activation value and the distance between a client neuron and a facility neuron. Iteratively, a facility is randomly chosen to be updated; if that facility currently has one of the top k inner values then its activation value is set to 1, and otherwise its activation value is set to 0. After updating the activation value of a facility neuron, the values of the rest of the neurons are updated further based on the connections to the facility neurons. The algorithm terminates once there are *k* facilities active. Rossiter compares the modified Hopfield network against the neural network of Haralampiev [75–77] and the local search algorithm of Arya et al. [3].

2.3.2 Hopfield Networks for Other Problems

Hopfield [84] introduced the Hopfield neural network in 1982 to implement content addressable memory, which is a memory capable of storing items and retrieving an item based on only partial information. The proposed neural network contained only a single fully-connected layer of neurons that perform asynchronous parallel processing.

Hopfield and Tank [85] later applied their network to solve combinatorial optimization problems, proposing a general energy function for Hopfield networks with n neurons:

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij}V_iV_j - \sum_{i=1}^{n} V_iI_i$$

Where V_i are the neuron state variables, T is a matrix of neuron connection weights, and I_i is an input bias to add input to neuron i. Table 2.11 lists several problems for which a Hopfield neural network has been designed for.

Hopfield Neural Networks		
Year	**Authors**	**Problem**
1982	Hopfield [84]	content addressable memory
1985	Hopfield and Tank [85]	traveling salesman problem
1988	Wilson and Pawley [171]	traveling salesman problem
1990	Aiyer et al. [1]	traveling salesman problem
1990	Kamgar-Parsi and Kamgar-Parsi [97]	traveling salesman problem
1990	Kamgar-Parsi and Kamgar-Parsi [97]	k-means
1996	Liang [114]	quadratic assignment problem
2004	Chen et al. [32]	vertex cover
2004	Salcedo-Sanz and Yao [148]	terminal assignment problem
2009	Wang et al. [169]	maximum diversity
2020	Manjunath et al. [120]	traveling salesman, vertex cover, maximum cut

Table 2.11: Hopfield networks being applied to optimization problems.

Hopfield and Tank [85] designed a Hopfield network for the traveling salesman problem by representing each city by n neurons to indicate where in the tour the city should be visited. Since in the traveling salesman problem each city should be visited exactly one time, then in the Hopfield network exactly one neuron out of the group of n neurons that correspond to a specific city should be active. The resulting network had n^2 neurons and was represented in a n by n matrix. Hopfield and Tank encoded the problem constraints into the energy function by adding additional energy into the network if (*i*) the sum of the neuron state values within a row of the matrix is not 1, (*ii*) the sum of the values within a column is not 1, and (*iii*) the sum of values in the matrix is not n. Several researchers [1, 97, 171] later attempted to recreate Hopfield and Tank's network for the traveling salesman problem and noted that the formulation returned many low-quality and infeasible solutions.

Kamgar-Parsi and Kamgar-Parsi [97] claimed that Hopfield networks are better suited for clustering problems, and created a k by n matrix representation and energy function specific to the k-means problem following the guidelines of Hopfield and Tank, where each vertex was represented by k neurons to indicate which cluster it should belong to and violations of the constraints of the problem add additional energy into the network. The Hopfield network performed much better on the k-means problem than it had on the traveling salesman problem (in running time, solution quality, and scalability), which Kamgar-Parsi and Kamgar-Parsi attributed to requiring a smaller matrix and having simpler constraints to impose on that matrix.

Several modifications to the Hopfield network have been presented, such as including an additional group of neurons [114], including additional processing steps after the network stabilizes [32], and integrating the Hopfield network with genetic algorithms [148], to try and produce feasible solutions more often or to avoid becoming stuck in local optima.

Chapter 3

3 Paper 1: High Multiplicity Strip Packing

A preliminary version of this paper was first published in 2022 (pp. 215-227) in the Proceedings of the 7th International Symposium on Combinatorial Optimization (ISCO) by Springer Nature [14]. The Version of Record for the preliminary paper is available online at: https://doi.org/10.1007/978-3-031-18530-4_16. An extended version of this paper was later submitted to the journal of Theory and Computing Systems and is currently under review.

This paper investigates the case when the input of a problem can be grouped into a small number of classes and whether approximation algorithms can take advantage of this case to outperform the more general algorithms that treat all the input items individually. This researched is explored in the context of the strip packing problem.

High Multiplicity Strip Packing with Three Rectangle Types

Abstract

The two-dimensional strip packing problem consists of packing in a rectangular strip of width 1 and minimum height a set of n rectangles, where each rectangle has width $0 < w \leq 1$ and height $0 < h \leq h_{max}$. We consider the high-multiplicity version of the problem in which there are only K different types of rectangles. For the case when $K = 3$, we give an algorithm that produces solutions requiring at most height $\frac{3}{2}h_{max} + \epsilon$ plus the height of an optimal solution, where ϵ is any positive constant. For the case when $K = 4$, we give an algorithm yielding solutions of height at most $\frac{7}{3}h_{max} + \epsilon$ plus the height of an optimal solution. For the case when $K > 3$, we give an algorithm that gives solutions of height at most $\lfloor \frac{3}{4}K \rfloor + 1 + \epsilon$ plus the height of an optimal solution.

Keywords: LP-relaxation, two-dimensional strip packing, high multiplicity, approximation algorithm

1 Introduction

The two-dimensional *strip packing problem* (SP) is defined as follows.

Definition 1 Given n rectangles with widths w_1, w_2, ..., w_n and heights h_1, h_2, ..., h_n, where $0 < w_i \leq 1$ for $i = 1, 2, ..., n$, the goal is to pack all the rectangles without rotations or overlaps in a rectangular strip of width 1 and minimum height.

This is a well-studied problem with applications in areas as diverse as resource allocation, scheduling, manufacturing, and transportation, among others. SP is equivalent to the classical bin packing problem if all rectangles have the same height, and since the bin packing problem is NP-hard [7] then SP is also NP-hard; therefore, the best possible approximation ratio achievable in polynomial time for SP is $\frac{3}{2}$ unless P = NP.

Baker et al. [1] designed the first approximation algorithm for SP which has approximation ratio 3. Coffman et al. [4] presented an algorithm with approximation ratio 2.7, Sleator [15] improved the approximation ratio to 2.5, and Schiermeyer [14] and Steinberg [17] further reduced the approximation ratio to

2. Harren and Van Stee [8] later presented an algorithm with approximation ratio 1.9396. The best known approximation algorithm for SP is from Harren et al. [9] with approximation ratio $\frac{5}{3}+\epsilon$. Several Asymptotic Polynomial Time Approximation Schemes (APTAS) have been presented as well: Kenyon and Rémila [13] gave an APTAS with an additive constant of $O(\frac{1}{\epsilon^2})$, and Jansen and Solis-Oba [11] improved Kenyon and Rémila's additive constant to 1. Sviridenko [16] presented a polynomial time algorithm that computes a solution of value $OPT + O(\sqrt{OPT \log OPT})$, where OPT is the value of an optimal solution.

In this paper we study the two-dimensional *high multiplicity strip packing problem* (HMSP), in which there is only a fixed number K of different rectangle types. A preliminary version of this paper was published at the proceedings of the 7th International Symposium on Combinatorial Optimization (ISCO 2022) [3]. This paper extends the previous work by including all proofs of correctness for the algorithm for the case when $K = 3$, an algorithm for the case when $K = 4$, an algorithm for any fixed value for K, and experimental results for our algorithm for the case when $K = 3$.

Note that the input to HMSP can be described using a list of only $3K$ numbers: the width w_i, height h_i, and number n_i of rectangles of each type T_i. Therefore, a challenging issue faced when designing an approximation algorithm for the problem is to ensure that its running time is a polynomial function of the size of the input. Observe that even describing a feasible solution for the problem using a polylogarithmic number of bits is not trivial as this requires specifying the positions of n rectangles in the packing; therefore, it is unknown whether HMSP belongs to the class NP.

We present an algorithm for HMSP for the case when $K = 3$ that computes solutions of value at most $OPT + \frac{3}{2}h_{max} + \epsilon$, where OPT is the value of an optimum solution, h_{max} is the height of the tallest rectangle, and ϵ is a positive constant. In addition we performed an experimental evaluation of this algorithm and our results show that our algorithm performs much better than the above theoretical upper bound. This algorithm is an improvement over the works of Yu and Solis-Oba [18] and Bloch-Hansen and Solis-Oba [2] whose algorithms computed solutions of value at most $OPT + \frac{5}{3}h_{max} + \epsilon$.

Our approach uses a formulation of HMSP that allows fractional rectangles in the solution called the two-dimensional *fractional strip packing problem* (FSP). We show that a solution for FSP can be converted into a solution for HMSP by a careful shifting, re-shaping, and combining of the fractional rectangles to form whole rectangles while increasing the height of the solution by at most $\frac{3}{2}h_{max} + \epsilon$. Our analysis is nearly tight as it is not hard to see that there are instances for which the corresponding fractional and integral solutions differ by h_{max}.

We also give two additional algorithms: (i) an algorithm for the case when $K = 4$ that computes solutions of value at most $OPT + \frac{7}{3}h_{max} + \epsilon$ and (ii) an algorithm that for any fixed K computes solutions of height at most $OPT + \lfloor \frac{3}{4}Kh_{max} \rfloor + h_{max} + \epsilon$.

The rest of the paper is organized in the following way. In Section 2 we describe how to compute a near optimum solution for FSP. In Sections 3-5 we present our algorithm for the case when $K = 3$. In Section 6 we describe a polynomial time implementation of the algorithm. In Section 7 we present our algorithm for the case when $K = 4$. In Section 8 we describe an algorithm for the case when $K > 3$. Finally, in Section 9 we describe our experimental results for the case when $K = 3$.

2 Solving FSP in Polynomial Time

HMSP can be relaxed to the two-dimensional *fractional strip packing problem* (FSP) by allowing horizontal cuts on the rectangles. A solution to FSP consists of a set of *configurations*. A *base configuration* C_j consists of a multiset of rectangle types whose total width is at most 1 (see Figure 1). A base configuration can be specified by indicating the number of rectangles of each type T_i in it. For example, the base configuration shown in Figure 1 consists of 4 rectangles of type T_1, 2 rectangles of type T_2, and 3 rectangles of type T_3, so that base configuration can be represented with the tuple (4,2,3).

A group of rectangles following a base configuration can be stacked on top of each other as shown in Figure 1, so that any horizontal line parallel to the base of the strip drawn across any part of the group will intersect the same multiset of rectangle types. This group of rectangles is called a *configuration*. A vertical line drawn across any part of a configuration will intersect either only rectangles of the same type, or empty space. The height of a vertical line intersecting rectangles of a configuration is called the height of the configuration. The configurations are stacked one on top of the other to form a fractional packing. Note that the number of possible configurations is $O(n^K)$.

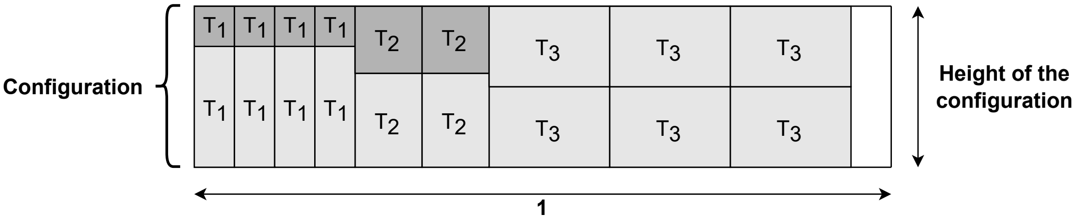


Fig. 1: A configuration with *base configuration* (4,2,3). The fractional rectangles are shaded in a darker color.

For a configuration C_j let $n_{i,j}$ be the number of rectangles of type T_i in its base configuration, for $i = 1, 2, ..., k$. Let x_j be a variable denoting the height of C_j. Let J be the set of all possible configurations. FSP can be expressed as the following linear program, hereafter referred to as linear program (1):

$$\begin{aligned} \text{Minimize: } & \sum_{C_j \in J} x_j \\ \text{Subject to: } & \sum_{C_j \in J} x_j n_{i,j} \geq n_i h_i, \text{ for each rectangle type } T_i \qquad (1) \\ & x_j \geq 0, \text{ for each } j \in J \end{aligned}$$

where n_i is the number of rectangles of type T_i and h_i is the height of each rectangle of type T_i. The objective function is to minimize the total height of the packing.

We denote with $OPT(I)$ the height of an optimal packing for instance I of HMSP and denote with $LIN(I)$ an optimal solution to the corresponding instance of FSP. It is not hard to see that $LIN(I) \leq OPT(I)$.

Note that FSP is identical to the fractional bin packing problem; in the latter problem a base configuration is a set of items that fit within a single bin and a solution to linear program (1) gives the fractional number of bins needed to pack all the items. Therefore, we can use an algorithm of Karmarkar and Karp [12] to compute a basic feasible solution for linear program (1) in time $O(K^9 \log K \log^2 \frac{K}{\epsilon})$ of value at most $LIN(I) + \epsilon$ for any fixed $\epsilon > 0$.

In any basic feasible solution, the number of nonzero variables is at most the number of constraints [10]. Thus, the number of nonzero variables, and therefore, the number of configurations used in a basic feasible solution for linear program (1) is at most the number of rectangle types, K.

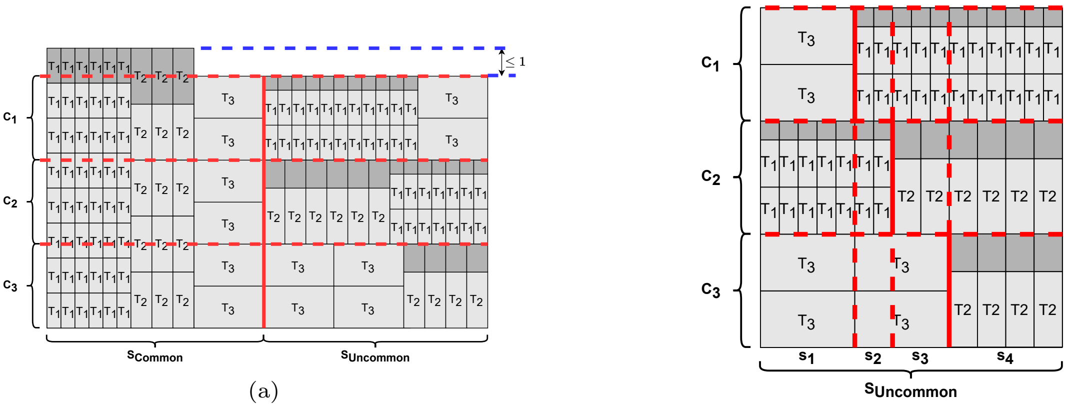
A simple algorithm for HMSP is to compute a basic feasible solution for linear program (1) and replace each fractional rectangle with a whole one of the corresponding type, shifting surrounding rectangles upwards as needed. Since a basic feasible solution for (1) uses at most K configurations and replacing the fractional rectangles with whole ones increases the height of a configuration by at most h_{max}, this algorithm computes a solution to HMSP of height at most $OPT(I) + Kh_{max} + \epsilon$.

3 Algorithm for HMSP with Three Rectangle Types

When $K = 3$ a basic feasible solution for linear program (1) consists of at most three configurations. Our algorithm performs several steps: 1) the fractional solution of the linear program is divided in two parts: S_{Common} and $S_{Uncommon}$, and the fractional rectangles in S_{Common} are replaced with whole ones of the corresponding types; 2) in $S_{Uncommon}$ the rectangles in each configuration are sorted and $S_{Uncommon}$ is further partitioned into vertical sections; 3) the vertical sections are grouped according to the heights of the fractional rectangles in them; and 4) the fractional rectangles in each group are combined and/or replaced with whole ones depending on their heights.

We assume for now that the fractional solution computed by solving linear program (1) consists of three configurations: C_1, C_2, and C_3. We show in later sections how to deal with the cases when the fractional solution has two or one configuration.

Step 1: Partitioning the Packing. For notational simplicity, in the sequel we assume $h_{max} = 1$. The three configurations of the solution for linear program (1) are stacked one on top of the other as shown in Figure 2a. Rectangles are rearranged horizontally within the configurations so that rectangles of the same type appearing in all three configurations are placed together in a section on the left side of the packing called S_{Common}. In the remaining portion of the packing, called $S_{Uncommon}$, each rectangle type is packed in at most 2 configurations (see Figure 2b), as the common rectangle types in the 3 configurations are in S_{Common}.



(a)

(b)

Fig. 2: (a) Replacing the fractional rectangles in S_{Common} with whole rectangles increases the height of the packing by at most 1. (b) Within each configuration, the rectangles in $S_{Uncommon}$ are sorted according to their fractional values. $S_{Uncommon}$ consists of four vertical sections: s_1, s_2, s_3, and s_4.

The fractional rectangles in S_{Common} are replaced with whole rectangles of the corresponding types, increasing the height of the packing by at most 1 (see Figure 2a). In the sequel, we discuss only how to process the fractional rectangles in $S_{Uncommon}$.

Step 2: Sorting $S_{Uncommon}$. Within each configuration, we place the fractional rectangles in $S_{Uncommon}$ at the top of the configuration. Let r be a fractional rectangle. The ratio between the height of r and the height of a rectangle of the same type as r is called the *fractional value* of r. We sort the rectangles so that fractional rectangles are sorted in non-decreasing order of their fractional values (see Figure 2b).

Step 3: Grouping Vertical Sections. We draw a vertical line at each point where two rectangles of different types are packed side-by-side within a configuration. These vertical lines partition $S_{Uncommon}$ into *vertical sections* (see Figure 2b). Vertical sections are indexed from left to right starting at index

1 for the leftmost section. Within some vertical section s_i, let $C_{1(i)}$, $C_{2(i)}$, and $C_{3(i)}$ refer to the part of C_1, C_2, and C_3 that is located within s_i, respectively.

Within a vertical section s_i, each configuration has a single rectangle type. Let $f_{1(i)}$, $f_{2(i)}$, and $f_{3(i)}$ represent the fractional values of the fractional rectangles packed in s_i at the top of C_1, C_2, and C_3, respectively.

We classify the vertical sections $s_i \in S_{Uncommon}$ into three cases, depending on the three fractional values $f_{1(i)}$, $f_{2(i)}$, and $f_{3(i)}$ as follows:

- S_{Case1} includes all sections s_i such that $f_{1(i)} + f_{2(i)} + f_{3(i)} \leq 1$.
- S_{Case2} includes all sections s_i such that $f_{1(i)} + f_{2(i)} + f_{3(i)} > 1$ and either $f_{1(i)} + f_{2(i)} \leq 1$, $f_{1(i)} + f_{3(i)} \leq 1$, or $f_{2(i)} + f_{3(i)} \leq 1$.
- S_{Case3} includes all sections s_i such that $f_{1(i)} + f_{2(i)} > 1$, $f_{1(i)} + f_{3(i)} > 1$, and $f_{2(i)} + f_{3(i)} > 1$. Note that for each $s_i \in S_{Case3}$

$$f_{1(i)} + f_{2(i)} + f_{3(i)} > \frac{3}{2} \tag{2}$$

Intuitively, vertical sections classified into S_{Case3} contain fractional rectangles with large fractional values and hence if they were to be replaced with whole rectangles of the corresponding type (as we did in S_{Common}) then the height of the packing would increase by only a small amount. In contrast, replacing the fractional rectangles with whole ones in vertical sections classified into S_{Case1} should be avoided as this would cause a large increase in the height of the packing; instead, the small fractional rectangles in S_{Case1} are combined and placed in another region of the packing. We use a mixture of these two techniques to process fractional rectangles belonging to S_{Case2}.

Step 4: Combining and/or Rounding Fractional Rectangles into Whole Ones. We carefully decide which fractional rectangles to combine into whole rectangles and which to replace with whole rectangles depending on the number of rectangle types in each configuration and the fractional values of the fractional rectangles. Therefore, Step 4 of our algorithm consists of several cases and the remaining subsections describe how our algorithm computes an integer packing for each of them.

Each one of the configurations C_1, C_2, and C_3 in $S_{Uncommon}$ could have one, two, or three different rectangle types. First we consider the cases when (i) each configuration has two rectangle types and (ii) one configuration has three types, another has two types, and the last one has one type, as these are the most complex cases for getting rid of the fractional rectangles.

3.1 Two Rectangle Types Per Configuration

We denote with $B_{i,j}$, for $i,j = 1,2,3$, a vertical line that separates two adjacent vertical sections belonging one to S_{Casei} and the other to S_{Casej} (see Figure 3). In the sequel, we say that the rectangles in a configuration *define* boundary $B_{i,j}$ if the rectangles adjacent to $B_{i,j}$ on the left and right sides are of different types. A rectangle r might intersect vertical sections of two or more cases; hereafter, we call such a rectangle a *vertically split* rectangle (see Figure 3).

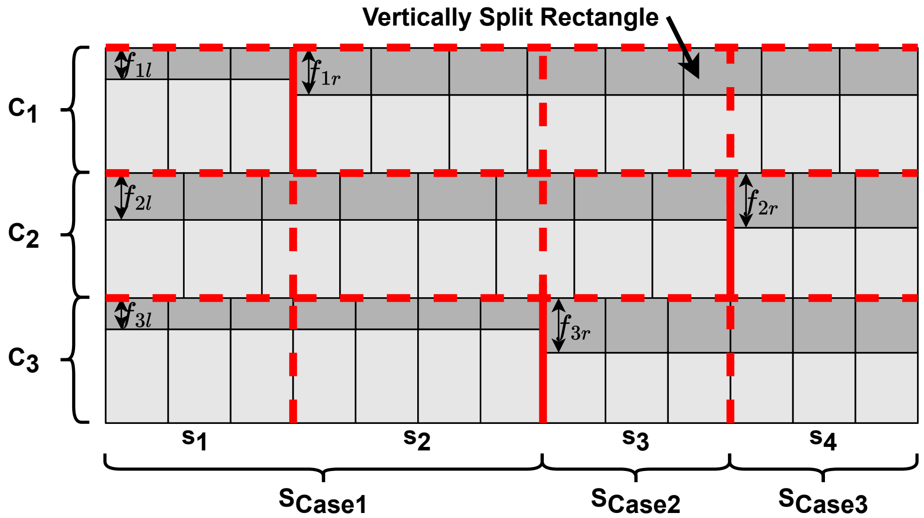


Fig. 3: The fractional rectangles in C_1 with fractional values f_{1l} and f_{1r} define $B_{1,1}$, the fractional rectangles in C_2 with fractional values f_{2l} and f_{2r} define $B_{2,3}$, and the fractional rectangles in C_3 with fractional values f_{3l} and f_{3r} define $B_{1,2}$.

Since in this section we assume that within $S_{Uncommon}$ each configuration contains exactly two different rectangle types, then within $S_{Uncommon}$ the rectangles in each configuration define at most one boundary $B_{i,j}$.

3.1.1 Ordering the Configurations

If no vertical sections are classified into S_{Casei}, for some $i = 1, 2, 3$, we say that S_{Casei} is empty. We order the configurations as follows:

- If S_{Case3} is empty or S_{Case2} is empty then order the configurations so that the rectangles in the bottom configuration define $B_{1,2}$ or $B_{1,3}$, respectively.
- Otherwise, order the configurations so that the rectangles in the middle configuration define $B_{2,3}$ and if S_{Case1} and S_{Case2} are not empty the rectangles in the bottom configuration must define $B_{1,2}$. Note that the rectangles in the middle configuration cannot define $B_{2,3}$ and $B_{1,2}$ because the middle configuration has only rectangles of two different types.

After ordering the configurations as above, let the configuration packed at the top be C_1, the one in the middle be C_2, and the one at the bottom be C_3 (see Figure 3). If we re-order the configurations later on, we will not re-name them; for example, if we re-order the configurations such that C_1 and C_3 swap positions, then C_1 would now be on the bottom.

Having the rectangles in C_2 define $B_{2,3}$, if possible, allows flexibility for shifting the rectangles in $C_2 \cap S_{Case3}$ as we show; for some of our algorithm's cases we shift these rectangles downwards into empty space if the rectangles in $C_3 \cap S_{Case3}$ take up less height than the rectangles in $C_3 \cap S_{Case2}$. Therefore, ordering the configurations as described above is important to our algorithm.

Let f_{il} and f_{ir} be the fractional values of the leftmost and rightmost fractional rectangles in configuration C_i, respectively, for each $i = 1, 2, 3$ (see Figure 3).

We use a variable called *numWide* to track how many wide rectangle types appear in a packing, where a rectangle type is considered to be wide if it is the leftmost type in its configuration and it is packed, at least partially, within S_{Case2} or S_{Case3}. Because the sum of fractional values of fractional rectangles in a vertical section belonging to S_{Case1} can be arbitrarily small, but the sum of fractional values in vertical sections belonging to S_{Case2} or S_{Case3} are at least 1, the presence (or absence) of these wide rectangle types is important in deciding whether we can re-use the empty space left behind when fractional rectangles are shifted around in S_{Case1} and S_{Case2} as we later explain.

We initialize variable *numWide* to 0. If any fractional rectangles with fractional value f_{1l} are packed within any vertical section of S_{Case2} or S_{Case3}, we increase the value of *numWide* by one. If any fractional rectangles with fractional value f_{2l} are packed within any section of S_{Case2} or S_{Case3}, we increase the value of *numWide* by one.

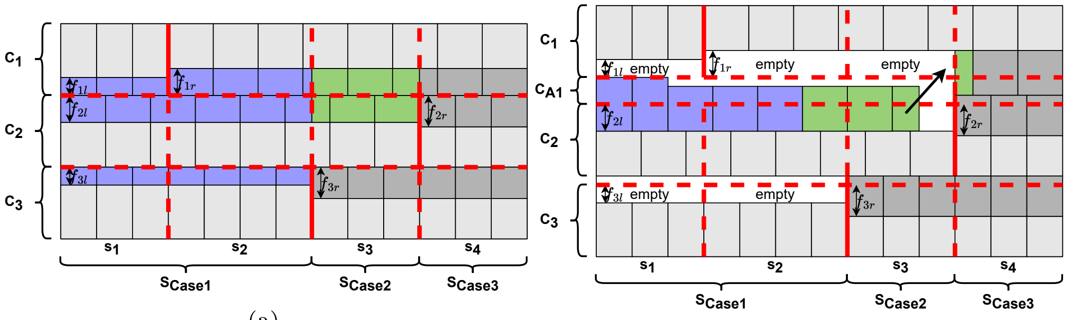
3.1.2 Pairing Configurations

Our algorithm for processing fractional rectangles sometimes needs to *pair* the two configurations at the top of the packing. To explain how configurations are packed, assume that C_1 is the configuration at the top of the packing and C_2 is the middle configuration. When pairing C_1 with C_2, we flip C_1 upside down (see Figure 4a). Let F_1 be the set of fractional rectangles in each vertical section $s_i \in S_{Case1}$, and let F_2 be the set of fractional rectangles from C_1 and C_2 in each vertical section $s_i \in S_{Case2}$ where $f_{1(i)} + f_{2(i)} \leq 1$. We remove the sets F_1 and F_2 from their original positions in the packing. If $F_1 \cup F_2$ is not empty we shift up the remaining rectangles in C_1 so that the tops of the topmost rectangles in C_1 lie on a common line and the distance between C_1 and C_2 in vertical section s_1 is 1. This creates a region in S_{Case1} and S_{Case2} of height at most 1 between C_1 and C_2 where we will pack F_1 and F_2; we call this region C_{A1} (see Figure 4b). If $F_1 \cup F_2$ is empty, then region C_{A1} has initial height zero, but its height might be increased later as explained below.

We re-shape each fractional rectangle $r \in F_1 \cup F_2$ so that its area does not change but it has the full height of a rectangle of the same type as r.

Lemma 1 *Let C_1 and C_2 be paired as described above. The re-shaped fractional rectangles in $F_1 \cup F_2$ can be packed in region C_{A1}.*

Proof Let vertical section $s_i \in S_{Case1}$ have width W_i and let $C_{A1(i)}$ be the part of C_{A1} within s_i. The total empty area A_i in $C_{A1(i)}$ is $A_i \geq W_i * 1 = W_i$. Since each of $C_{1(i)}$, $C_{2(i)}$, and $C_{3(i)}$ has only one fractional rectangle type, the total area a_i of the fractional rectangles in $C_{1(i)}$, $C_{2(i)}$, and $C_{3(i)}$ is



(a)

(b)

Fig. 4: (a) C_1 has been flipped upside down and the sets F_1 and F_2 are highlighted in blue and green, respectively. (b) Rectangles in C_1 have been shifted upwards to create the region C_{A1} to hold the rectangles in $F_1 \cup F_2$ and the rest of the fractional rectangles have been rounded up.

$$a_i \leq (W_i * f_{1(i)}) + (W_i * f_{2(i)}) + (W_i * f_{3(i)}) \leq W_i \leq A_i,$$

as the height of each rectangle is at most 1 and $f_{1(i)} + f_{2(i)} + f_{3(i)} \leq 1$ for $s_i \in S_{Case1}$.

A similar argument can be made for the vertical sections $s_i \in S_{Case2}$ for which $f_{1(i)} + f_{2(i)} \leq 1$. □

Observe how in Figure 4b the fractional rectangles with fractional value f_{1r} are not *wide* (as they are not leftmost in their configuration) and so the empty space left behind after moving these fractional rectangles to C_{A1} does not have the same width as C_{A1}. This is important because the empty space left behind from moving the non-wide fractional rectangles with fractional value f_{1l} (that belong only to S_{Case1}) can have an arbitrarily small height, and hence the empty space in C_1 left over after moving the fractional rectangles in S_{Case1} and S_{Case2} to C_{A1} cannot be included in C_{A1} as only the height of the shorter of the two empty spaces (from fractional rectangles with fractional value f_{1l} and f_{1r}) can be added to C_{A1}, as we do not know the number and heights of the rectangles that will be packed in C_{A1}.

In contrast, in Figure 4b if the fractional rectangles with fractional value f_{1l} were wide, then the empty space left behind from moving these wide fractional rectangles would extend to the left side of $S_{Uncommon}$ and since $f_{1r} > f_{1l}$ the empty space remaining after moving the rightmost rectangles to C_{A1} would have at least the same height, and so the empty space from C_1 left over after moving the fractional rectangles in S_{Case1} and S_{Case2} has height $f_{1l}h_{1l}$ and width equal to C_{A1} making C_{A1} taller. Since the fractional values of wide rectangles appear in a vertical section belonging to S_{Case2} and/or S_{Case3}, then their fractional values are bounded as stated in the definitions of S_{Case2} and/or S_{Case3}, ensuring that the increase in height to C_{A1} is not arbitrarily small.

Corollary 1 *After re-shaping the fractional rectangles in $F_1 \cup F_2$ we can pack them in C_{A1} so that there is at most one fractional rectangle of each type in C_{A1}.*

Proof We combine the fractional rectangles in $F_1 \cup F_2$ such that a whole rectangle is formed whenever a sufficient number of pieces of the same type have been packed. When fractional rectangles of the same type do not form a whole rectangle, they merge to become one larger fractional rectangle. Therefore, at most one fractional rectangle of each type may remain. By Lemma 1 the rectangles can be packed in C_{A1}. □

We *round up* the fractional rectangles from C_1 and C_2 in each vertical section $s_i \in S_{Case2}$ where $f_{1(i)} + f_{2(i)} > 1$ and for each vertical section $s_i \in S_{Case3}$. Rounding up a fractional rectangle r means replacing it with a whole rectangle of the same type as r and shifting rectangles up as needed to make room for the whole rectangle. When shifting rectangles from C_1 we need ensure that the tops of the topmost rectangles in C_1 lie on a common line. Finally, we round up the fractional rectangles in $C_3 \cap S_{Case2}$ (see Figure 4b).

Note that after pairing two configurations and re-shaping rectangles some whole rectangles might be vertically split by the boundaries $B_{1,2}$ and $B_{2,3}$. Because of the way in which region C_{A1} was defined, the two pieces of a whole rectangle that is vertically split by any of those boundaries are placed side-by-side forming a whole rectangle. However, pieces of fractional rectangles that are vertically split might be placed in different parts of the packing. Later we show how to shift these fractional pieces to form whole rectangles.

3.1.3 Rounding Fractional Rectangles

Recall that the variable *numWide* counts the number of wide rectangles as defined earlier and that if no vertical sections belong to a particular S_{Casei} we say that S_{Casei} is empty. We provide different algorithms for rounding fractional rectangles into whole ones based on which of S_{Case1}, S_{Case2}, and S_{Case3} are not empty and what the value of *numWide* is.

Lemma 2 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, then numWide > 0.*

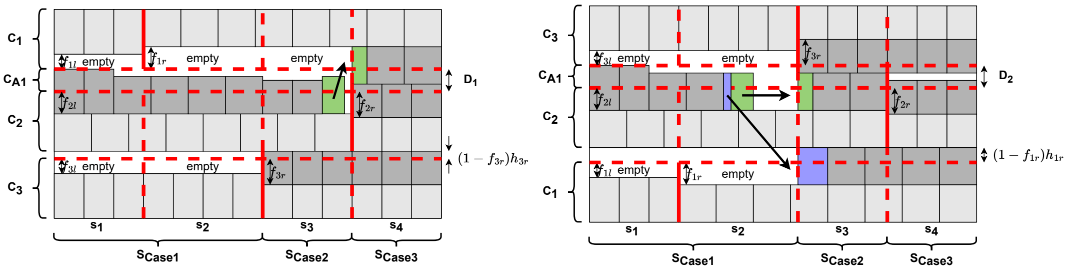
Proof Assume that *numWide* $= 0$ and none of S_{Case1}, S_{Case2}, and S_{Case3} are empty. Because of how we ordered the configurations, the rectangles in C_2 define the boundary $B_{2,3}$ and therefore the fractional rectangles in C_2 with fractional value f_{2l} are packed in S_{Case2}, contradicting the assumption that *numWide* $= 0$. □

By Lemma 2, we do not need consider the case when none of S_{Case1}, S_{Case2}, and S_{Case3} are empty and *numWide* $= 0$. For the rest of the cases in this subsection, the intuition behind each approach is that we need to identify which of the fractional rectangles belonging to S_{Case2} should be rounded up, and which of the empty spaces leftover after moving fractional rectangles in S_{Case2} can be used to place rectangles without increasing the total height of the packing by more than $\frac{1}{2}$. To see how we do this, note that the sum of fractional values in vertical sections v belonging to S_{Case2} is more than 1, and so if one of the fractional values in v is small then another of the fractional values must be large; therefore, if the large fractional rectangles are wide then they can be moved to C_{A1} and the empty space left behind after moving them can be included in C_{A1}, and if the large fractional rectangles are not wide they can be rounded up to increase the height of the packing by only a small amount.

For simplicity and without loss of generality, in the sequel we assume that none of the configurations computed by solving linear program (1) contain any empty space, so the width of the base configuration of each configuration C is equal to 1. Additionally, we assume that for some configuration C_i, if its leftmost and rightmost rectangle types t_{il} and t_{ir} are both in $S_{Case1} \cup S_{Case2}$, then $f_{il}h_{il} < f_{ir}h_{ir}$ where h_{il} and h_{ir} are the corresponding heights of the whole rectangles of type t_{il} and t_{ir}, respectively. Note that if the opposite is true the analysis is very similar, so we omit it.

3.1.4 Rounding Fractional Rectangles: One Wide Rectangle

Let h_{il} and h_{ir} be the height of the rectangles corresponding to fractional values f_{il} and f_{ir}, respectively, for $i = 1, 2, 3$. Because of the way the configurations were ordered, C_2 must contain the wide rectangle type with fractional value f_{2l} as the rectangles in C_2 define $B_{2,3}$. If f_{2l} is large enough then rounding up the wide fractional rectangles is a good solution; otherwise, the empty space left by moving the wide rectangles to C_{A1} increases the height of C_{A1} by $f_{2l}h_{2l}$; note that if the wide rectangle's fractional value is small, then another fractional value in S_{Case2} must be large and can therefore be rounded up instead.



(a) (b)

Fig. 5: $numWide = 1$ and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$.

Lemma 3 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 1$, and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

Proof Our algorithm will produce two solutions and choose the one with shorter height. For the first solution, pair C_1 and C_2 and re-shape, pack, and round rectangles as explained in Section 3.1.2 (see Figure 5a). The height increase in S_{Case1} and S_{Case2} caused by creating C_{A1} is at most $h_1 - f_{1l}h_{1l} - f_{2l}h_{2l} \leq h_1 - f_{2l}h_{2l}$, where $h_1 = max\{h_{1l}, h_{1r}, h_{2l}, h_{3l}\}$ (note that C_{A1} re-uses the space that was occupied by the fractional rectangles of fractional values f_{1l} and f_{2l}).

In S_{Case3}, the height increase caused by rounding up the fractional rectangles with fractional values f_{1r} and f_{2r} is at most $(1 - f_{1r})h_{1r} + (1 - f_{2r})h_{2r}$; hence the height increase caused by pairing C_1 and C_2 is at most $D_1 = max\{h_1 - f_{2l}h_{2l}, (1 - f_{1r})h_{1r} + (1 - f_{2r})h_{2r}\}$. The height increase caused by rounding up the fractional rectangles in C_3 with fractional value f_{3r} is at most $(1 - f_{3r})h_{3r}$ (see Figure 5a). Therefore, the total height increase is at most $max\{\Delta_A, \Delta_B\}$, where $\Delta_A = h_1 - f_{2l}h_{2l} + (1 - f_{3r})h_{3r} \leq 2 - f_{3r} - f_{2l}h_{2l}$, as $h_1 \leq 1$ and $h_{3r} \leq 1$ and $\Delta_B = (1 - f_{1r})h_{1r} + (1 - f_{2r})h_{2r} + (1 - f_{3r})h_{3r} \leq 3 - f_{1r} - f_{2r} - f_{3r} < \frac{3}{2}$ as $h_{1r} \leq 1, h_{2r} \leq 1$, and $f_{1r} + f_{2r} + f_{3r} > \frac{3}{2}$ by (2).

For the second solution, re-order the configurations so that fractional rectangles with fractional value f_{1l} appear in the bottom configuration, and fractional rectangles with fractional value f_{3l} appear in the top configuration, then pair C_2 and C_3 (note that these are now the top two configurations) and re-shape, pack, and round rectangles as explained in Section 3.1.2 (see Figure 5b). We only consider the case when $f_{3r} + f_{2l} > 1$ (see Figure 5b); the case when $f_{3r} + f_{2l} \leq 1$ is similar.

The height increase caused by creating C_{A1} is at most $h_2 - f_{2l}h_{2l} - f_{3l}h_{3l}$, where $h_2 = max\{h_{1l}, h_{1r}, h_{2l}, h_{3l}\}$. In S_{Case2} and S_{Case3}, the height increase caused by rounding up the fractional rectangles with fractional values f_{2l}, f_{2r}, and f_{3r} is at most $max\{(1 - f_{2l})h_{2l}, (1 - f_{2r})h_{2r}\} + (1 - f_{3r})h_{3r}$; hence the height increase caused by pairing C_2 and C_3 is at most $D_2 = max\{h_2 - f_{2l}h_{2l} - f_{3l}h_{3l}, max\{(1 - f_{2l})h_{2l}, (1 - f_{2r})h_{2r}\} + (1 - f_{3r})h_{3r}\}$. The height increase caused by rounding up fractional rectangles in C_1 with fractional value f_{1r} is at most $(1-f_{1r})h_{1r}$. Therefore, the total height increase is at most $max\{\Delta_C, \Delta_D\}$, where $\Delta_C = (1 - f_{1r})h_{1r} + h_2 - f_{2l}h_{2l} - f_{3l}h_{3l}$ and $\Delta_D = (1-f_{1r})h_{1r} + max\{(1-f_{2l})h_{2l}, (1-f_{2r})h_{2r}\} + (1-f_{3r})h_{3r}$.

Selecting the better of the two solutions produces an increase in the height of the solution by $max\{min\{\Delta_A, \Delta_C\}, min\{\Delta_A, \Delta_D\}, min\{\Delta_B, \Delta_C\}, min\{\Delta_B, \Delta_D\}\}$.

- $min\{\Delta_A, \Delta_C\}$: $\Delta_A = 2 - f_{3r} - f_{2l}h_{2l}$ and $\Delta_C = (1 - f_{1r})h_{1r} + max\{h_{2l}, h_{3l}\} - f_{2l}h_{2l} - f_{3l}h_{3l}$. Note that $\Delta_A \leq 2 - f_{3r}$ and since $h_{1r}, h_{2l}, h_{3l} \leq 1$ then $\Delta_C \leq (1 - f_{1r}) + (1 - f_{3l}) = 2 - f_{1r} - f_{3l}$. Since $f_{3r} + f_{1r} > 1$ as fractional rectangles with fractional values f_{3r} and f_{1r} appear in S_{Case3} then either $f_{3r} > \frac{1}{2}$ or $f_{1r} > \frac{1}{2}$ and so $min\{\Delta_A, \Delta_C\} \leq \frac{3}{2}$.
- $min\{\Delta_A, \Delta_D\}$: $\Delta_A = 2 - f_{3r} - f_{2l}h_{2l}$ and $\Delta_D = (1 - f_{1r})h_{1r} + max\{(1 - f_{2l})h_{2l}, (1 - f_{2r})h_{2r}\} + (1 - f_{3r})h_{3r}$. Recall our assumption that $f_{3r} + f_{2l} > 1$ (the case when $f_{3r} + f_{2l} \leq 1$ is similar), therefore either $f_{3r} > \frac{1}{2}$ or $f_{2l} > \frac{1}{2}$. If $f_{3r} > \frac{1}{2}$ then $\Delta_A < \frac{3}{2} - f_{2l}h_{2l} < \frac{3}{2}$. If $f_{2l} > \frac{1}{2}$ then $\Delta_D \leq (1 - f_{1r}) + (1 - f_{3r}) + max\{1 - f_{2l}, 1 - f_{2r}\} = 2 - f_{1r} - f_{3r} + max\{1 - f_{2l}, 1 - f_{2r}\}$:
  - If $1 - f_{2l} > 1 - f_{2r}$ then $\Delta_D \leq 3 - f_{1r} - f_{3r} - f_{2l} < 3 - \frac{1}{2} - f_{1r} - f_{3r} < \frac{3}{2}$ as $f_{1r} + f_{3r} > 1$.
  - If $1 - f_{2r} > 1 - f_{2l}$ then $\Delta_D \leq 3 - f_{1r} - f_{2r} - f_{3r} \leq \frac{3}{2}$ by (2).

  Therefore, $min\{\Delta_A, \Delta_D\} \leq \frac{3}{2}$.
- $min\{\Delta_B, \Delta_C\} \leq \frac{3}{2}$ and $min\{\Delta_B, \Delta_D\} \leq \frac{3}{2}$ because $\Delta_B \leq \frac{3}{2}$.

□

Observe that in the first solution, depicted in Figure 5a, there might be a fractional rectangle r in C_1 that is vertically split by $B_{2,3}$ such that one piece of r is re-shaped and packed as explained in Section 3.1.2, while the other piece is rounded up to the height of a rectangle of the same type as r. These pieces are marked in Figure 5a. Note that the two pieces can be placed beside each other to form a whole rectangle without further increasing the height of the packing. Similarly the fractional rectangles in Figure 5b can be combined to form whole rectangles without affecting the height of the packing. In the sequel we will not explicitly explain how fractional rectangles that are vertically split are combined to form whole rectangles, instead the figures will show how to do this.

Lemma 4 *If none of* S_{Case1}, S_{Case2}, *and* S_{Case3} *are empty and* $numWide = 1$, *then* C_1*'s rectangles cannot define* $B_{2,2}$, $B_{2,3}$, *or* $B_{3,3}$.

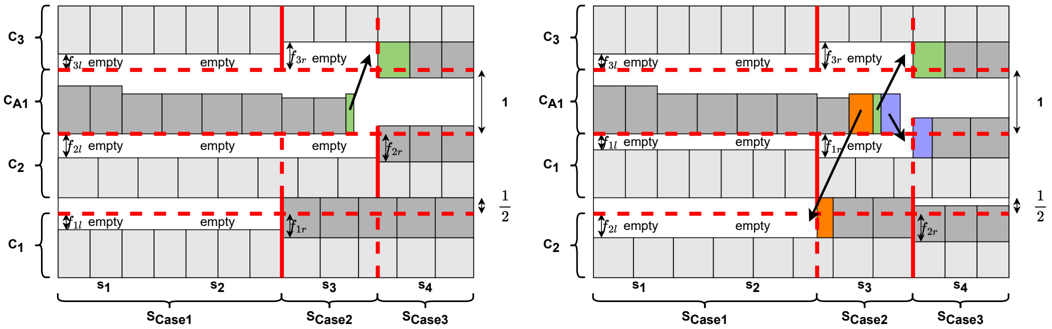
Proof Note that if C_1's rectangles defined $B_{2,2}$, then the value of $numWide$ would be 2 because rectangles in C_1 with fractional value f_{1l} would appear in S_{Case2} and since the rectangles in C_2 define $B_{2,3}$ then rectangles with fractional value f_{2l} would also appear in S_{Case2}. Similarly, it is not possible that C_1's rectangles define boundaries $B_{2,3}$ or $B_{3,3}$. □

Lemma 5 *If none of* S_{Case1}, S_{Case2}, *and* S_{Case3} *are empty,* $numWide = 1$, *and* $f_{1(i)} + f_{2(i)} > 1$ *for at least one* $s_i \in S_{Case2}$, *then there is an algorithm that produces an integer packing of height at most* $\frac{3}{2}$ *plus the value of the solution for linear program (1).*

Proof By Lemma 4, C_1's rectangles cannot define $B_{2,2}$, $B_{2,3}$, or $B_{3,3}$. Note that if C_1's rectangles defined $B_{1,1}$, then $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$ since the rectangles in C_2 define $B_{2,3}$ and thus fractional rectangles with fractional values f_{1r} and f_{2l} would appear within S_{Case1} and so $f_{1r} + f_{2l}$ would be at most 1. Therefore, the rectangles in C_1 and C_3 must create a coinciding boundary $B_{1,2}$ so that $f_{1r} + f_{2l}$ could be larger than 1, as required by the Lemma.

Since $f_{1r} + f_{2l} > 1$, then $f_{1r} > \frac{1}{2}$ and/or $f_{2l} > \frac{1}{2}$. If $f_{1r} > f_{2l}$, then re-order the configurations so that fractional rectangles with fractional value f_{1r} appear in the bottom configuration. Pair C_2 and C_3 and re-shape, pack, and round rectangles as explained Section 3.1.2. The height increase caused by pairing C_2 and C_3 is at most 1: for sections s_i where $f_{1(i)} + f_{2(i)} \leq 1$ the height increase caused by creating C_{A1} is at most 1, and for sections s_i where $f_{1(i)} + f_{2(i)} > 1$ the height increase caused by rounding up $f_{1(i)}$ and $f_{2(i)}$ is also at most 1. The height increase caused by rounding up fractional rectangles with fractional value f_{1r} is at most $\frac{1}{2}$ (see Figure 6a) so the total height increase is at most $\frac{3}{2}$.

If $f_{2l} > f_{1r}$, then re-order the configurations so that fractional rectangles with fractional value f_{2l} appear in the bottom configuration, pair C_1 and C_3, and re-shape, pack, and round rectangles as explained in Section 3.1.2. The height increase caused by pairing C_1 and C_3 is at most 1 and the height increase caused by rounding up fractional rectangles with fractional value f_{2l} is at most $\frac{1}{2}$ (see Figure 6b). □



(a) (b)

Fig. 6: $numWide = 1$ and $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$.

3.1.5 Rounding Fractional Rectangles: Two Wide Rectangles

Because of the way the configurations were ordered, when there are two wide rectangle types in $S_{Uncommon}$ C_1 must contain a wide rectangle type with fractional value f_{1l} and C_2 must contain a second wide rectangle type with fractional value f_{2l}, as the rectangles in C_3 define $B_{1,2}$. Either the fractional rectangles with fractional value f_{3r} can be rounded up and the two wide fractional rectangle types with fractional values f_{1l} and f_{2l} are moved to C_{A1} and the empty space left behind after moving these wide rectangles can be included in C_{A1}, or one of the wide rectangle types can be rounded up and the other wide fractional rectangle type is moved to C_{A1} and the empty space left behind after moving these wide rectangles can be included in C_{A1}.

Lemma 6 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 2$, and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

Proof Note that the rectangles in C_1 cannot create $B_{1,1}$ as otherwise $numWide < 2$.

- If $(1 - f_{3r})h_{3r} \leq \frac{1}{2}$ then re-order the configurations so that fractional rectangles with fractional value f_{3r} appear in the bottom configuration. If $(1 - f_{3r})h_{3r} > \frac{1}{2}$:
  - If $(1 - f_{1l})h_{1l} \leq \frac{1}{2}$ re-order the configurations so that fractional rectangles with fractional value f_{1l} appear in the bottom configuration.

  - If $(1-f_{2l})h_{2l} \leq \frac{1}{2}$ re-order the configurations so that fractional rectangles with fractional value f_{2l} appear in the bottom configuration.

  Pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2. These solutions are very similar as that in Figure 6a. The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up the fractional rectangles in the bottom configuration is at most $\frac{1}{2}$. To see this note that if $(1-f_{3r})h_{3r} > \frac{1}{2}$ then $h_{3r} > \frac{1}{2}$ and $f_{3r} < \frac{1}{2}$ as $h_{3r} \leq 1$; furthermore, $f_{1r} > \frac{1}{2}$ and $f_{2r} > \frac{1}{2}$ since $f_{1r} + f_{3r} > 1$ and $f_{2r} + f_{3r} > 1$ (as fractional values f_{1r}, f_{2r}, and f_{3r} appear in S_{Case3}). So the total height increase is at most $\frac{3}{2}$.

- If $(1-f_{3r})h_{3r} > \frac{1}{2}$, $(1-f_{2l})h_{2l} > \frac{1}{2}$, and $(1-f_{1l})h_{1l} > \frac{1}{2}$, then $h_{1l} > \frac{1}{2}$, $h_{2l} > \frac{1}{2}$, and $h_{3r} > \frac{1}{2}$. Pair C_1 and C_2 and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2 (see Figure 7). The height increase in S_{Case1} and S_{Case2} caused by creating C_{A1} is at most $h_1 - f_{1l}h_{1l} - f_{2l}h_{2l}$, where $h_1 = max\{h_{1l}, h_{1r}, h_{2l}, h_{3l}\}$. In S_{Case3}, the height increase caused by rounding up the fractional rectangles with fractional values f_{1r} and f_{2r} is at most $(1-f_{1r})h_{1r} + (1-f_{2r})h_{2r}$; hence the height increase caused by pairing C_1 and C_2 is at most $D_1 = max\{h_1 - f_{1l}h_{1l} - f_{2l}h_{2l}, (1-f_{1r})h_{1r} + (1-f_{2r})h_{2r}\}$. The height increase caused by rounding up the fractional rectangles in C_3 with fractional value f_{3r} is at most $(1-f_{3r})h_{3r}$. Therefore, the total height increase is at most $max\{\Delta_A, \Delta_B\}$, where:

  - $\Delta_A = h_1 - f_{1l}h_{1l} - f_{2l}h_{2l} + (1-f_{3r})h_{3r} = h_1 + h_{3r} - (f_{1l}h_{1l} + f_{2l}h_{2l} + f_{3r}h_{3r}) \leq 2 - \frac{1}{2}(f_{1l} + f_{2l} + f_{3r}) < \frac{3}{2}$, as $h_1 \leq 1$, $h_{1l} > \frac{1}{2}$, $h_{2l} > \frac{1}{2}$, $1 \geq h_{3r} > \frac{1}{2}$, and $f_{1l} + f_{2l} + f_{3r} > 1$, and
  - $\Delta_B = (1-f_{1r})h_{1r} + (1-f_{2r})h_{2r} + (1-f_{3r})h_{3r} \leq (1-f_{1r}) + (1-f_{2r}) + (1-f_{3r}) = 3 - f_{1r} - f_{2r} - f_{3r} < \frac{3}{2}$ as $h_{1r} \leq 1, h_{2r} \leq 1$, $h_{3r} \leq 1$, and $f_{1r} + f_{2r} + f_{3r} > \frac{3}{2}$ by (2).

□

Lemma 7 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, numWide $= 2$, and $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

Proof Note that if C_1's rectangles defined $B_{2,3}$ or $B_{3,3}$, then $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$ since the rectangles in C_2 define $B_{2,3}$ and therefore rectangles with fractional values f_{1l} and f_{2l} would appear within S_{Case1} (so $f_{1l}+f_{2l} \leq 1$) and they would be the only fractional values in $S_{Case2} \cap (C_1 \cup C_2)$. Additionally, note that C_1's rectangles cannot define $B_{1,1}$ or $B_{1,2}$, as otherwise *numWide* could not have value 2. Therefore, C_1's rectangles must define boundary $B_{2,2}$ so that $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$ and $f_{1r} + f_{2l} > 1$, as required by the Lemma.

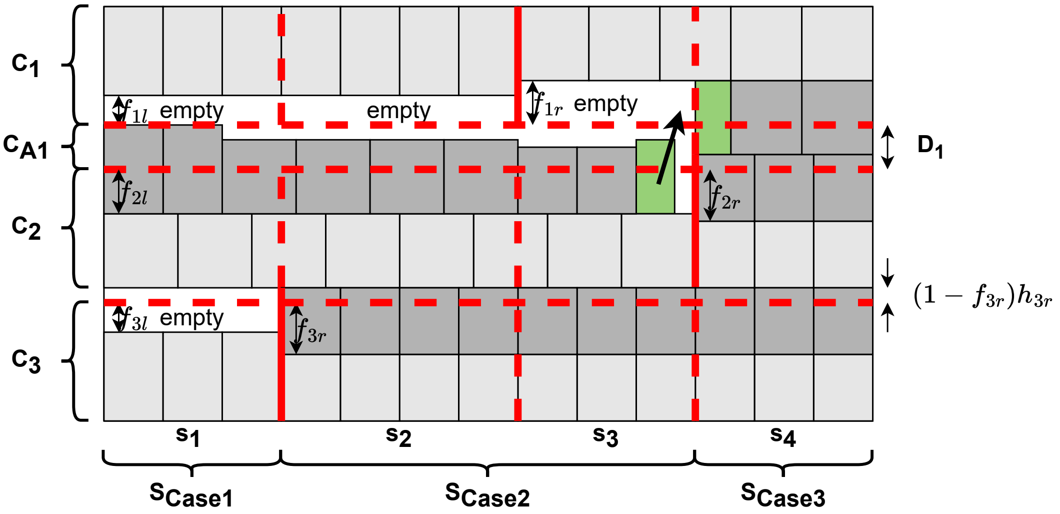


Fig. 7: $numWide = 2$, $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, $(1-f_{3r})h_{3r} > \frac{1}{2}$, $(1-f_{2l})h_{2l} > \frac{1}{2}$, and $(1-f_{1l})h_{1l} > \frac{1}{2}$.

Similar to the analysis in the proof of Lemma 6, if $(1-f_{3r})h_{3r} \leq \frac{1}{2}$, or if $(1-f_{3r})h_{3r} > \frac{1}{2}$ and $(1-f_{2l})h_{2l} \leq \frac{1}{2}$ or $(1-f_{1l})h_{1l} \leq \frac{1}{2}$, then we re-order the configurations so that the fractional value f_{3r}, f_{2l}, or f_{1l} appears in the bottom configuration, respectively. The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles in the bottom configuration is at most $\frac{1}{2}$, and so the total height increase is at most $\frac{3}{2}$.

Hence, we only need to consider the case when $(1-f_{3r})h_{3r} > \frac{1}{2}$, $(1-f_{2l})h_{2l} > \frac{1}{2}$, and $(1-f_{1l})h_{1l} > \frac{1}{2}$. Note that then $f_{3r} < \frac{1}{2}$, $f_{2l} < \frac{1}{2}$, $f_{1l} < \frac{1}{2}$, $h_{2l} > \frac{1}{2}$, $h_{1l} > \frac{1}{2}$, and $f_{1r} > \frac{1}{2}$ as $f_{1r} + f_{2l} > 1$. Consider the fractional values f_{1l}, f_{2l}, and f_{3r}, and re-order the configurations so that the two largest fractional values among them are in the bottom and middle configurations. If $f_{1l} \geq f_{2l} \geq f_{3r}$ or $f_{2l} \geq f_{1l} \geq f_{3r}$, then ensure that fractional value f_{1l} appears in the bottom configuration. If $f_{1l} \geq f_{3r} \geq f_{2l}$, $f_{3r} \geq f_{1l} \geq f_{2l}$, $f_{2l} \geq f_{3r} \geq f_{1l}$, or $f_{3r} \geq f_{2l} \geq f_{1l}$, then ensure that fractional value f_{3r} appears in the bottom configuration. Pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2. We consider below just the case when $f_{1l} \geq f_{2l} \geq f_{3r}$; the other cases are similar.

Observe that $f_{1l} + f_{2l} > \frac{2}{3}$ as $f_{1l} + f_{2l} + f_{3r} > 1$ (see Figure 8). The height increase in S_{Case1} and S_{Case2} caused by creating C_{A1} is at most $h_1 - f_{2l}h_{2l} - f_{3l}h_{3l} \leq h_1 - f_{2l}h_{2l}$, where $h_1 = max\{h_{1l}, h_{2l}, h_{3l}, h_{3r}\}$. In S_{Case3}, the height increase caused by rounding up the fractional rectangles with fractional values f_{2r} and f_{3r} is at most $(1-f_{2r})h_{2r} + (1-f_{3r})h_{3r}$.

Note that $(1-f_{1l})h_{1l} > (1-f_{1r})h_{1r}$, as $f_{1r} > \frac{1}{2}$ and so $(1-f_{1r})h_{1r} \leq \frac{1}{2}$ but $(1-f_{1l})h_{1l} > \frac{1}{2}$. Thus the fractional rectangles with fractional value f_{2r} (and the whole rectangles of the same type beneath them in the middle configuration) can be shifted downwards into the empty space above the fractional rectangles with fractional value f_{1r} (see Figure 8). Hence, the height increase caused from pairing the top two configurations is at most $D_1 = max\{h_1 - f_{2l}h_{2l}, (1-f_{2r})h_{2r} + (1-f_{3r})h_{3r} - ((1-f_{1l})h_{1l} - (1-f_{1r})h_{1r})\}$.

The height increase caused by rounding up the fractional rectangles in the bottom configuration with fractional value f_{1l} is at most $(1 - f_{1l})h_{1l}$. Therefore, the total height increase is at most $max\{\Delta_A, \Delta_B\}$, where:

- $\Delta_B = (1-f_{2r})h_{2r} + (1-f_{3r})h_{3r} - ((1-f_{1l})h_{1l} - (1-f_{1r})h_{1r}) + (1-f_{1l})h_{1l} \leq (1-f_{2r}) + (1-f_{3r}) + (1-f_{1r}) = 3 - f_{1r} - f_{2r} - f_{3r} \leq \frac{3}{2}$ by (2), as h_{1r}, h_{2r}, and h_{3r} are at most 1.
- $\Delta_A = h_1 - f_{2l}h_{2l} + (1 - f_{1l})h_{1l} \leq 2 - f_{1l} - f_{2l}h_{2l}$ as $h_1 \leq 1$ and $h_{1l} \leq 1$. Since Δ_A is a decreasing function on $f_{1l} + f_{2l}$ and $f_{1l} + f_{2l} > \frac{2}{3}$ then an upper bound for the value of Δ_A can be obtained when $f_{1l} + f_{2l} = \frac{2}{3}$ and so $f_{1l} = \frac{2}{3} - f_{2l}$, therefore $\Delta_A \leq 2 - \frac{2}{3} + f_{2l} - f_{2l}h_{2l} = \frac{4}{3} + f_{2l} - f_{2l}h_{2l} < \frac{4}{3} + f_{2l} - \frac{f_{2l}}{2(1-f_{2l})}$ because $(1 - f_{2l})h_{2l} > \frac{1}{2}$. Then $\Delta_A < \frac{4}{3} + \frac{f_{2l} - 2f_{2l}^2}{2(1-f_{2l})}$. The right hand side of this inequality takes its maximum value when $f_{2l} = 1 - \frac{\sqrt{2}}{2}$ and so $\Delta_A < \frac{4}{3} + \frac{3}{2} - \sqrt{2} = \frac{17}{6} - \sqrt{2} < \frac{3}{2}$.

□

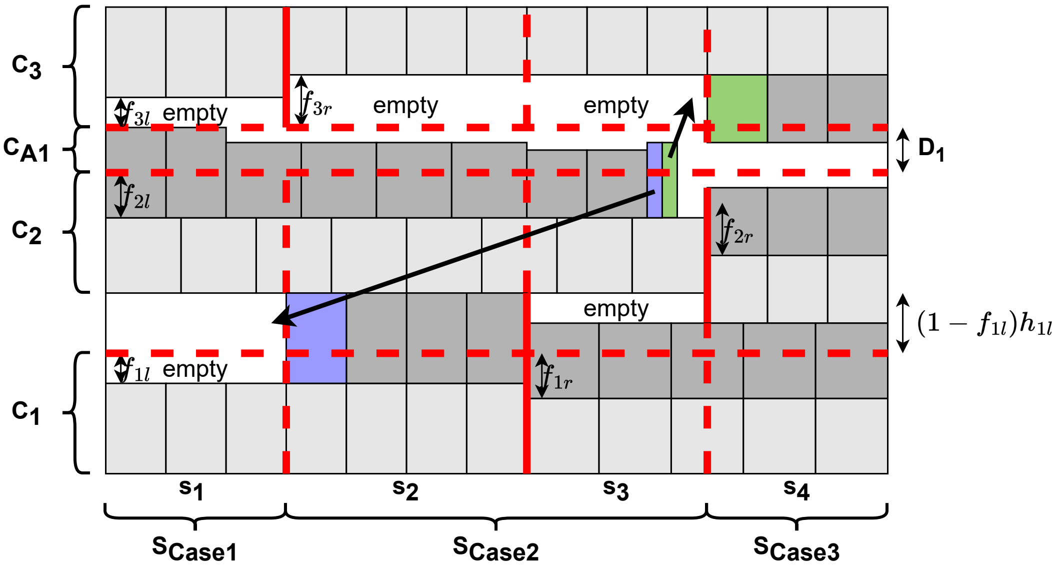
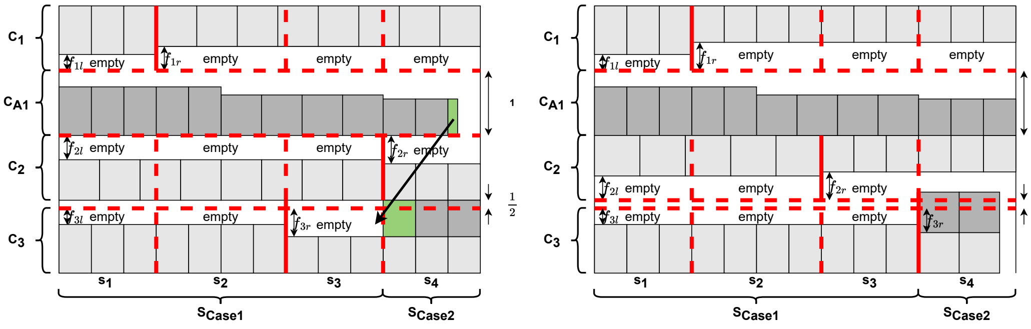


Fig. 8: $numWide = 2$, $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, $(1-f_{3r})h_{3r} > \frac{1}{2}$, $(1-f_{2l})h_{2l} > \frac{1}{2}$, $(1-f_{1l})h_{1l} > \frac{1}{2}$, and $f_{1l} \geq f_{2l} \geq f_{3r}$.

3.1.6 Rounding Fractional Rectangles: Some S_{Casei} is Empty

If there is some S_{Casei} that is empty, then the problem becomes simpler and we address the remaining cases as follows:

- If S_{Case1} is not empty, but S_{Case2} and S_{Case3} are both empty, pair C_1 and C_2 re-shape and pack all fractional rectangles in C_{A1} as explained in Section 3.1.2. Therefore we obtain a solution of height at most 1 plus the value of the solution for linear program (1).



(a) (b)

Fig. 9: If $S = S_{Case1} \cup S_{Case2}$ and $numWide = 0$, then there is exactly one vertical section $s_i \in S_{Case2}$. (a) The largest fraction is more than $\frac{1}{2}$, and (b) the largest fraction is less than $\frac{1}{2}$.

- If S_{Case2} is not empty, but S_{Case1} and S_{Case3} are both empty, then $numWide > 0$ and we can use Lemmas 3-7.
- If S_{Case3} is not empty, but S_{Case1} and S_{Case2} are both empty, round up all fractional rectangles. Since $f_{1(i)} + f_{2(i)} + f_{3(i)} > \frac{3}{2}$ for every vertical section $s_i \in S_{Case3}$ then we obtain a solution of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).
- If both S_{Case1} and S_{Case2} are not empty, but S_{Case3} is empty, and $numWide = 0$, there must be only one vertical section $s_i \in S_{Case2}$ as otherwise the boundary $B_{2,2}$ must exist, but that would mean that at least one fractional rectangle with fractional value a or c must be within S_{Case2} and therefore $numWide$ would have value larger than zero.
  - If the largest fractional value in the section $s_i \in S_{Case2}$ is more than $\frac{1}{2}$, re-order the configurations so that the fractional rectangles with that fractional value appear in the bottom configuration and then pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2 (see Figure 9a). The height increase caused by creating C_{A1} is at most 1 and the height increase caused by rounding up the fractional rectangles in the bottom configuration is at most $\frac{1}{2}$.
  - Otherwise, re-order the configurations such that for the section $s_i \in S_{Case2}$ the fractional value in the top configuration in S_{Case2} is the smallest, and the boundary defined by the rectangles in the bottom configuration does not occur to the left of the boundary defined by the rectangles in the middle configuration. Pair the top two configurations and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2, then flip the middle configuration upside down. Note that the rounded-up rectangles in S_{Case2} in the bottom configuration can use the empty space left behind by the fractional rectangles in the middle configuration (see Figure 9b).

  The height increase caused by creating C_{A1} is at most 1. Note that since the middle configuration is flipped upside down, the empty space leftover after removing the fractional rectangles from the middle configuration can be used by the rounded up rectangles in the bottom configuration; therefore, the height increase caused by rounding up the fractional rectangles in the bottom configuration is at most $\frac{1}{2}$, as the fractional values the middle and bottom configurations sum to more than $\frac{1}{2}$, so the total height increase is at most $\frac{3}{2}$.

- If both S_{Case1} and S_{Case2} are not empty, but S_{Case3} is empty, and $numWide > 0$, then we can use Lemmas 3-7.
- If both S_{Case1} and S_{Case3} are not empty, but S_{Case2} is empty, and $numWide = 0$, then re-order the configurations so that the fractional rectangles in the bottom configuration are the largest of fractional values f_{1r}, f_{2r}, and f_{3r}. Note that fractional values f_{1l} and f_{2l} are not within S_{Case3}, as $numWide = 0$, and the rectangles in the bottom configuration create $B_{1,3}$, so fractional value f_{3l} is not within S_{Case3} either. Pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2. The height increase from pairing the top two configurations is at most 1, and the height increase from rounding up the fractional rectangles in S_{Case3} in the bottom configuration is at most $\frac{1}{2}$, so the total height increase is at most $\frac{3}{2}$.
- If both S_{Case1} and S_{Case3} are not empty, but S_{Case2} is empty, and $numWide > 0$, then we can use Lemmas 3-7.
- If both S_{Case2} and S_{Case3} are not empty, but S_{Case1} is empty, then $numWide = 2$, and we can use Lemmas 3-7.

Theorem 1 *If $K = 3$ and the fractional solution computed by solving linear program (1) has exactly three configurations, and if each of those configurations has exactly two different rectangle types in $S_{Uncommon}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

3.2 Three Rectangle Types in a Configuration

In this section we consider the case when the fractional solution obtained from solving linear program (1) has three configurations, one configuration has exactly three rectangle types, one configuration has exactly two rectangle types, and one configuration has exactly one rectangle type. The algorithms described in this section are modifications of the algorithms described in the previous section to account for the existence of a configuration with three rectangle types. Note that only a single configuration in $S_{Uncommon}$ can pack three rectangle types, as otherwise at least one of the rectangle types would be common to all three configurations.

3.2.1 Ordering the Configurations

We order the configurations as follows:

- The top configuration contains only one rectangle type.
- The rectangles in the middle configuration define $B_{2,3}$, if it exists. Note that the rectangles in the middle configuration can define both $B_{1,2}$ and $B_{2,3}$ if it contains three rectangle types.

After ordering the configurations as above, let the configuration packed at the top be C_1, the one in the middle be C_2, and the one at the bottom be C_3.

Let a be the fractional value of the fractional rectangles in C_1. If C_2 contains three rectangle types, then let b, c, and d be the fractional values in C_2 and let e and f be the fractional rectangles in C_3. Otherwise, if C_2 contains two rectangle types, then let b and c be the fractional values in C_2 and let d, e, and f be the fractional rectangles in C_3.

Initialize variable *numWide* to 0. Increase *numWide* in the following way:

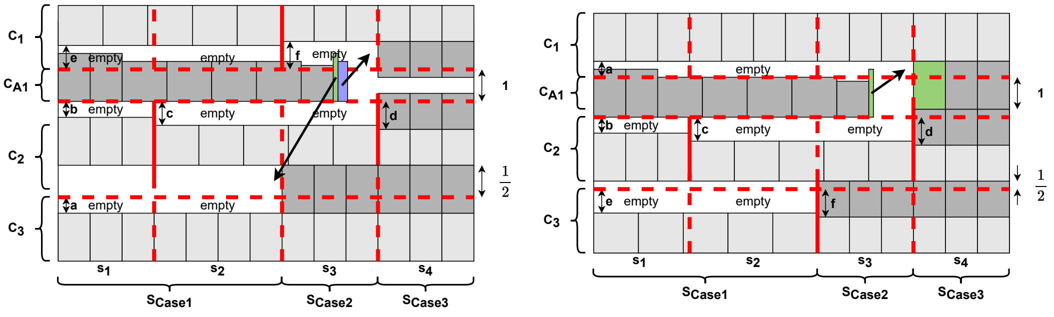
- If any fractional rectangles with fractional value a are packed within any vertical section of S_{Case2} or S_{Case3}, increase the value of *numWide* by one.
- If any fractional rectangles with fractional value b are packed within any vertical section of S_{Case2} or S_{Case3}, increase the value of *numWide* by one.
- If C_2 contains three rectangle types:
  - If any fractional rectangles with fractional value e are packed within any vertical section of S_{Case2} or S_{Case3}, increase the value of *numWide* by one.
- If C_2 contains two rectangle types:
  - If any fractional rectangles with fractional value d are packed within any vertical section of S_{Case2} or S_{Case3}, increase the value of *numWide* by one.

We provide different algorithms for rounding fractional rectangles into whole ones based on which of S_{Case1}, S_{Case2}, and S_{Case3} are not empty and what the value of $numWide$ is.

3.2.2 Rounding Fractional Rectangles: One Wide Rectangle

Note that because C_1 has only one rectangle type, if none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, then $numWide > 0$. Therefore, the fractional rectangles with fractional value a are wide and either they can be rounded up or the empty space remaining after moving them to C_{A1} can be included in C_{A1}.

Lemma 8 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 1$ and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*



(a) (b)

Fig. 10: None of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 1$, $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, and (a) $(1-a)h_a \leq \frac{1}{2}$ and (b) $(1-a)h_a > \frac{1}{2}$ so $f > \frac{1}{2}$.

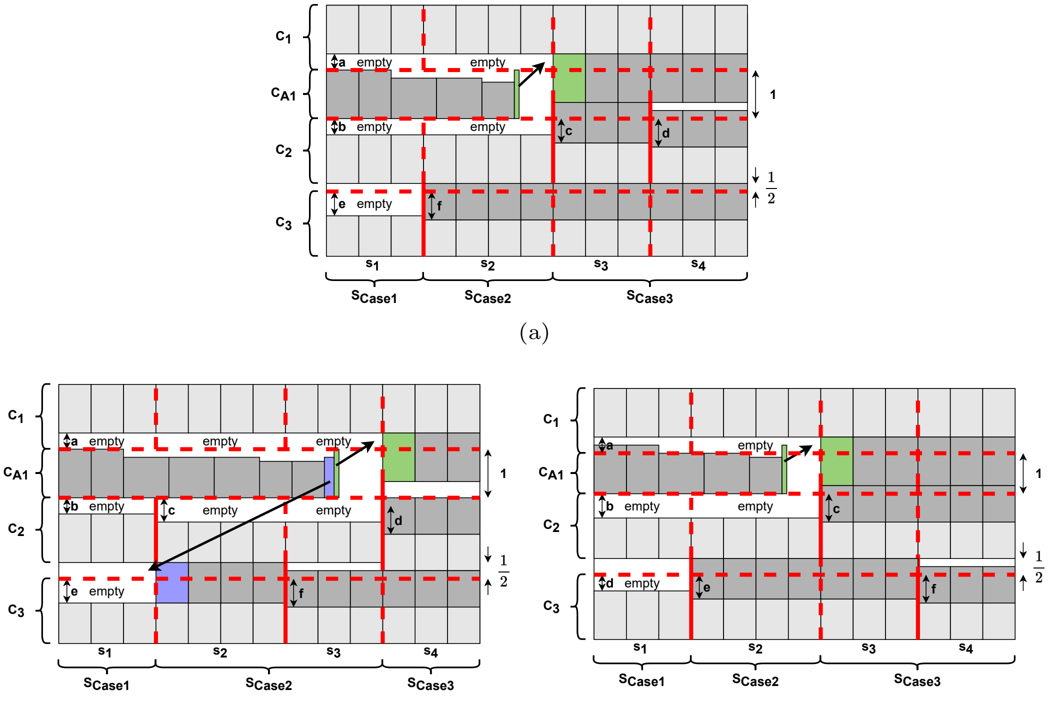
Proof Since the rectangles in C_2 define $B_{2,3}$, if C_2 had only two rectangle types then $numWide > 1$; hence C_2 must have three rectangle types. So, in C_2 fractional rectangles with fractional value b are in S_{Case1} (but not S_{Case2}, as otherwise $numWide > 1$), fractional rectangles with fractional value c are in S_{Case2} (but not S_{Case3}, as these rectangles define $B_{2,3}$), and only the fractional rectangles with fractional value d are in S_{Case3}. In C_3 fractional value e cannot be within S_{Case2} or S_{Case3}, as otherwise $numWide > 1$.

If $(1-a)h_a \leq \frac{1}{2}$ then re-order the configurations so that fractional rectangles with fractional value a appear in the bottom configuration (see Figure 10a); otherwise $(1-a)h_a > \frac{1}{2}$, so $a < \frac{1}{2}$ and $f > \frac{1}{2}$ because $a+f > 1$ as both fractional values appear in S_{Case3} (see Figure 10b). Pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2. The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles with fractional value a (if $(1-a)h_a \leq \frac{1}{2}$) or f (if $(1-a)h_a > \frac{1}{2}$) is at most $\frac{1}{2}$ so the total height increase is at most $\frac{3}{2}$. □

Note that by the proof of Lemma 8, C_2 must have three rectangle types and the rectangles in C_2 do not define the boundary $B_{2,2}$ (and neither do the rectangles in C_1 or C_3 define this boundary); therefore, there is only one vertical section in S_{Case2} and since $a+f > 1$ (as the fractional rectangles with fractional values a and f both appear in a vertical section in S_{Case3}) then $a + c \leq 1$ and hence we do not need to consider the case when $numWide = 1$ and $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$.

3.2.3 Rounding Fractional Rectangles: Two Wide Rectangles

Similar to the algorithm in Section 3.1.5, we try to take advantage of the presence of two wide rectangles by moving these fractional rectangles to C_{A1} and including in C_{A1} the empty space left behind after moving these rectangles. Additionally, since the fractional rectangles with fractional value a belong to a vertical section in S_{Case3} we know that the sum of a and the rightmost fractional values of C_2 or C_3 must be more than 1.



(a)

(b)

(c)

Fig. 11: $S = S_{Case1} \cup S_{Case2} \cup S_{Case3}$, $numWide = 2$, and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$. (a) C_2's leftmost fractional value is packed within S_{Case2} and $(1-a)h_a > \frac{1}{2}$. (b) C_3's leftmost fractional value is packed within S_{Case2} and $(1-a)h_a > \frac{1}{2}$, $(1-e)h_e < \frac{1}{2}$, and $f > \frac{1}{2}$. (c) C_2 has only two rectangle types and $(1-a)h_a > \frac{1}{2}$, $(1-e)h_e < \frac{1}{2}$, and $f > \frac{1}{2}$.

Lemma 9 *If none of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 2$ and $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

Proof First assume that C_2 has three rectangle types and C_3 has two rectangle types. We need to consider two cases.

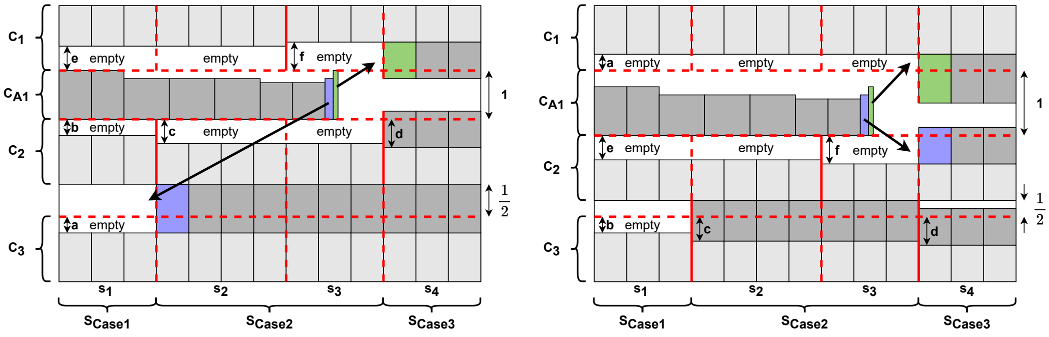
- C_2's leftmost fraction is packed within S_{Case2}. Therefore, C_2 cannot create either $B_{1,1}$ or $B_{1,2}$, C_3 must define $B_{1,2}$, and fractional value e does not appear in S_{Case2} or S_{Case3} (see Figure 11a). We process the fractional rectangles using the same approach as in Lemma 8.
- C_3's leftmost fraction is packed within S_{Case2}. Therefore, C_2 must define $B_{1,2}$ and C_3 could create either $B_{2,2}$, $B_{2,3}$, or $B_{3,3}$ (see Figure 11b). If $(1-a)h_a \leq \frac{1}{2}$ then re-order the configurations so that fractional rectangles with fractional value a appear in the bottom configuration. Otherwise, if $(1-a)h_a > \frac{1}{2}$ and $(1-c)h_c \leq \frac{1}{2}$ (or $(1-e)h_e \leq \frac{1}{2}$), then re-order the configurations so that fractional rectangles with fractional value c (or e)

appear in the bottom configuration. Pair the top two configurations, and reshape, pack, and round fractional rectangles as explained in Section 3.1.2. The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles in the bottom configuration is at most $\frac{1}{2}$. To see this note that if $(1-a)h_a > \frac{1}{2}$ then $h_a > \frac{1}{2}$ and $a < \frac{1}{2}$ as $h_a \leq 1$; therefore, when this happens $d > \frac{1}{2}$ and $f > \frac{1}{2}$ since $a + d > 1$ and $a + f > 1$ (as fractional values a, d, and f appear in S_{Case3}). So the total height increase is at most $\frac{3}{2}$. Hence, we only need to consider the case when $(1-a)h_a > \frac{1}{2}$, $(1-c)h_c > \frac{1}{2}$, and $(1-e)h_e > \frac{1}{2}$, which can be addressed using the approach from the proof of Lemma 7 and it increases the height of the packing by at most $\frac{3}{2}$.

Assume now that C_2 has two rectangle types and C_3 has three rectangle types. Note that fractional rectangles in C_2 with fractional value b are packed within S_{Case2}, because the rectangles in C_2 define boundary $B_{2,3}$, and so C_3 must create $B_{1,2}$ (see Figure 11c).

Again, similar to the analysis above, if $(1-a)h_a \leq \frac{1}{2}$ then re-order the configurations so that fractional rectangles with fractional value a appear in the bottom configuration. Otherwise, if $(1-a)h_a > \frac{1}{2}$ and $(1-b)h_b \leq \frac{1}{2}$ (or $(1-e)h_e \leq \frac{1}{2}$), then re-order the configurations so that fractional rectangles with fractional value b (or e) appear in the bottom configuration. Pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2. The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles in the bottom configuration is at most $\frac{1}{2}$. To see this note that if $(1-a)h_a > \frac{1}{2}$ then $h_a > \frac{1}{2}$ and $a < \frac{1}{2}$ as $h_a \leq 1$; therefore, when this happens $c > \frac{1}{2}$ and $f > \frac{1}{2}$ since $a + c > 1$ and $a + f > 1$ (as fractional values a, c, and f appear in S_{Case3}). So the total height increase is at most $\frac{3}{2}$.

Finally, when $(1-a)h_a > \frac{1}{2}$, $(1-b)h_b > \frac{1}{2}$, and $(1-e)h_e > \frac{1}{2}$, using the approach from the proof of Lemma 7 increases the height of the packing by at most $\frac{3}{2}$. □



(a) (b)

Fig. 12: None of S_{Case1}, S_{Case2}, and S_{Case3} are empty, $numWide = 2$, $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, and (a) $a > \frac{1}{2}$ and (b) $a < \frac{1}{2}$ so $c > \frac{1}{2}$ and $d > \frac{1}{2}$.

Lemma 10 *If $numWide = 2$, and $f_{1(i)} + f_{2(i)} > 1$ for at least one vertical section $s_i \in S_{Case2}$, then there is an algorithm that produces an integer packing of height at most $\frac{3}{2}$ plus the value of the solution for linear program (1).*

Proof Note that if C_2 has two rectangle types then $f_{1(i)} + f_{2(i)} \leq 1$ for all vertical sections $s_i \in S_{Case2}$ since the rectangles in C_2 define boundary $B_{2,3}$ and so fractional values a and b would appear within S_{Case1} and S_{Case2} and so $a + b$ would be at most 1; therefore, C_2 must have three rectangle types.

Additionally, note that since C_2 has three rectangle types, its rectangles must also define $B_{1,2}$ or $B_{2,2}$, as otherwise $f_{1(i)} + f_{2(i)} \leq 1$ for all sections $s_i \in S_{Case2}$. The rectangles in C_2 cannot define $B_{1,1}$ as then the rectangles in C_3 would have to define $B_{1,2}$ and so $numWide$ would have value 1.

We first consider when the rectangles in C_2 define $B_{1,2}$, which means that the rectangles in C_3 define either $B_{2,2}$, $B_{2,3}$, or $B_{3,3}$, all of which are handled the same way.

Since $a + c > 1$, then $a > \frac{1}{2}$ and/or $c > \frac{1}{2}$.

- If $a > \frac{1}{2}$ then re-order the configurations so that fractional rectangles with fractional value a appear in the bottom configuration, pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2 (see Figure 12a). The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles with fractional value a is at most $\frac{1}{2}$ so the total height increase is at most $\frac{3}{2}$.
- If $a < \frac{1}{2}$ then $c > \frac{1}{2}$; also $d > \frac{1}{2}$, and $f > \frac{1}{2}$ as fractional values a, d, and f appear in S_{Case3}. Re-order the configurations so that fractional rectangles with fractional value c appear in the bottom configuration, pair the top two configurations, and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2 (see Figure 12b). The height increase caused by pairing the top two configurations is at most 1. The height increase caused by rounding up fractional rectangles with fractional values c and d is at most $\frac{1}{2}$ so the total height increase is at most $\frac{3}{2}$.

For the case when the rectangles in C_2 define $B_{2,2}$, which means that the rectangles in C_3 define $B_{1,2}$, we use the same approach as in Lemma 8. □

When only S_{Case1} is not empty, or when only S_{Case3} is not empty, we can use the algorithms described in Section 3.1 for the same cases. Note that when only S_{Case2} or S_{Case3} are empty, then $numWide > 0$ since there is a configuration containing a single rectangle type, and hence the algorithms from Lemmas 8-10 can be used. When only S_{Case2} is not empty or when only S_{Case1} is empty, then $numWide > 0$ and the algorithms from Lemmas 8-10 can be used.

Theorem 2 *If $K = 3$ and the fractional solution computed by solving linear program (1) has exactly three configurations, one configuration has three rectangle types, one configuration has two rectangle types, and one configuration has only one rectangle* *type, then there is an algorithm that produces an integer packing of height at most* $\frac{3}{2}$ *plus the value of the solution for linear program (1).*

3.3 Fewer Than Three Configurations

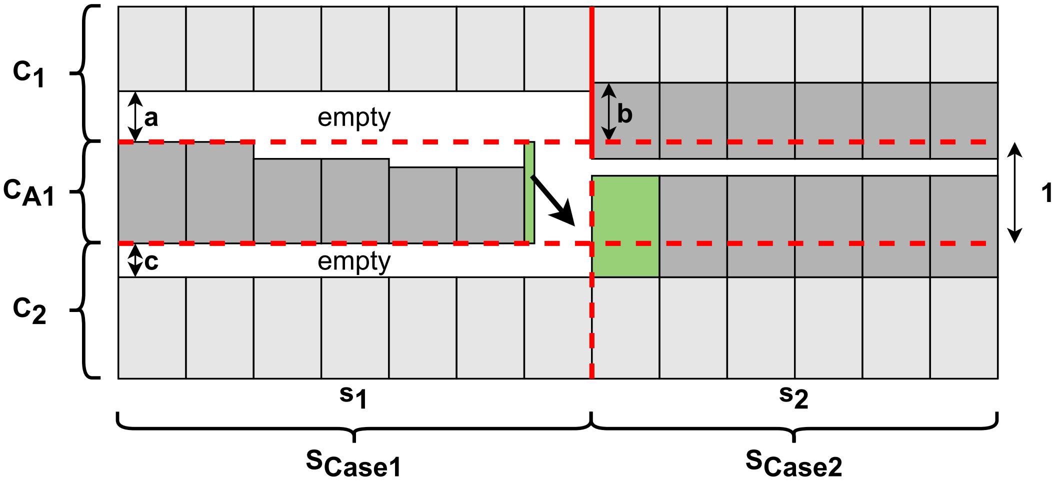


Fig. 13: When there are only two configurations, a vertical section is classified as S_{Case1} if $f_{1(i)} + f_{2(i)} \leq 1$ and classified as S_{Case2} if $f_{1(i)} + f_{2(i)} > 1$.

We have described how to round a fractional packing with exactly three configurations computed by solving linear program (1). When the fractional packing has fewer than three configurations we need to group the vertical sections in a different manner. When there are only two configurations, a vertical section s_i is classified as S_{Case1} if $f_{1(i)} + f_{2(i)} \leq 1$ and classified as S_{Case2} if $f_{1(i)} + f_{2(i)} > 1$. Pair the two configurations and re-shape, pack, and round fractional rectangles as explained in Section 3.1.2 (see Figure 13). Note that the height increase caused by pairing the two configurations is at most 1.

When there is only one configuration, all fractional rectangles are rounded up for a height increase of the packing of at most 1.

3.4 Differing Number of Rectangle Types in Each Configuration

We have described how to round $S_{Uncommon}$ when each configuration has exactly two rectangle types, or when one configuration has three rectangle types, one configuration has two rectangle types, and one configuration has only one rectangle type. In all of the other remaining possible combinations of the number of rectangle types in each configuration there is at least one configuration with only a single rectangle type, and so the approach used in Sections 5.1-5.6 can be applied for all of these remaining cases.

4 Polynomial Time Implementation

Recall that the input to HMSP is represented as a list of $3K$ numbers, not a list specifying the dimensions of n rectangles; therefore, any algorithm that specifies individual locations of rectangles in a solution for HMSP will not run in polynomial time.

We represent a configuration as a list of $O(K)$ numbers: for $1 \leq i \leq K$ we specify the rectangle type T_i, the number of rectangles of type T_i packed side-by-side, and the number of rectangles of type T_i packed on top of each other (note that this last number might not be integer).

Since there are at most K configurations, and we create at most one additional configuration by creating C_{A1} during the rounding process, then at most $O(K^2)$ numbers are needed to specify the packing in $S_{Uncommon}$. Similarly, the packing in S_{Common} is specified using at most $O(K)$ numbers, for a total of $O(K^2)$ numbers to specify the entire packing.

The number of rectangles of type T_i that are packed side-by-side in S_{Common} is equal to the minimum of the number of rectangles of type T_i that are packed in each of C_1, C_2, and C_3. The number of rectangles of type T_i that are packed vertically in S_{Common} is equal to the rounded up sum of the number of rectangles of type T_i that are packed one-on-top of the other in each of C_1, C_2, and C_3. Therefore, finding the number of rectangles of each type that belong in S_{Common} requires $O(K^2)$ operations.

Processing S_{Common} requires $O(K)$ operations as for $1 \leq i \leq K$ our algorithm only needs to round up the fractional values for each rectangle type T_i. Sorting the rectangles in each configuration in $S_{Uncommon}$ by their fractional values requires $O(K^2)$ operations.

Ordering the configurations as specified in Sections 4 and 5 requires $O(K)$ operations, computing the value of the *numWide* variable requires $O(K)$ operations, and checking which of the cases specified in the lemmas of Sections 4 and 5 are present in the fractional packing requires $O(K)$ operations. Re-shaping, packing, and rounding fractional rectangles as described in Section 3.1.2 requires $O(K^2)$ operations. Finally, packing leftover vertically split fractional rectangles as shown in the figures requires $O(K)$ operations.

Note that the above analysis holds regardless of how many rectangle types are in each configuration of $S_{Uncommon}$.

Theorem 3 *There is a polynomial time algorithm for HMSP with three rectangle types that computes solutions of value at most* $OPT + \frac{3}{2} + \epsilon$ *for* $\epsilon > 0$.

Proof As shown in Section 2, an optimal fractional solution to FSP can be computed in polynomial time. Our algorithm transforms fractional packings obtained by solving linear program (1) into integer packings with height of at most $\frac{3}{2} + \epsilon$ plus the height of the corresponding fractional packing, where ϵ is a positive constant. Finally, as shown above our algorithm can be implemented in polynomial time. ☐

5 4-Type Algorithm

When $K = 4$ a basic feasible solution for linear program (1) consists of at most four configurations. Our algorithm for this case performs the same four steps as for the case when $K = 3$.

When there are only one or two configurations, the fractional rectangles can be rounded as described in Section 3.3. Note that when $K = 4$ but there are only three configurations in the fractional solution of linear program (1), we cannot use our 3-type algorithm described above, as that algorithm takes advantage of where the at most three case boundaries $B_{i,j}, i \neq j$ are located, but when $K = 4$ there can be up to eight boundaries, and these are not accounted for in the algorithms we described above. Therefore, when $K = 4$ but there are three configurations, we pair the top two configurations as described in Section 3.1.2 and round up the fractional rectangles in the bottom configuration to produce a packing of height at most 2 plus the value of the solution for linear program (1). In the sequel we only consider the case where the solution of linear program (1) has 4 configurations.

5.1 Grouping Vertical Sections

Recall that within a vertical section s_i, each configuration has a single rectangle type. Let i be the smallest section index for which the sum of the smallest three fractions in section s_i is more than 1, if such a section exists; otherwise, we set $i = 0$. We order the configurations so that $f_{1(i)} \leq f_{2(i)} \leq f_{3(i)} \leq f_{4(i)}$, where $f_{1(i)}$, $f_{2(i)}$, $f_{3(i)}$, and $f_{4(i)}$ represent the fractional values of the fractional rectangles packed in s_i of C_1, C_2, C_3, and C_4, respectively.

We classify the vertical sections $s_i \in S_{Uncommon}$ into 4 cases, depending on the four fractional values $f_{1(i)}$, $f_{2(i)}$, $f_{3(i)}$, and $f_{4(i)}$ as follows:

- S_{Case1} includes all sections s_i such that $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} \leq 1$.
- S_{Case2} includes all sections s_i such that $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} > 1$ and $f_{1(i)} + f_{2(i)} + f_{3(i)} \leq 1$.
- S_{Case3} includes all sections s_i such that $f_{1(i)} + f_{2(i)} + f_{3(i)} > 1$ and $f_{1(i)} + f_{2(i)} \leq 1$.
- S_{Case4} includes all sections s_i such that $f_{1(i)} + f_{2(i)} > 1$.

If no section s_i exists for which the sum of the smallest three fractions is more than 1, then cases S_{Case3} and S_{Case4} will be empty.

5.2 Case1: $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} \leq 1$

For every section $s_i \in S_{Case1}$, we remove the fractional rectangles in C_1, C_2, C_3, and C_4 (see Figure 14), including the parts r_{Case1} for vertically split fractional rectangles. We re-shape the fractional rectangles so that they have the full height of a rectangle of the same type but only a fraction of its width, and then we pack them side-by-side in C_{A1}: to create C_{A1} all rectangles in C_1 are shifted upwards, including rectangles in S_{Case2}, S_{Case3}, and S_{Case4}, until there is empty space of height 1 between C_1 and C_2 in section s_i. After shifting

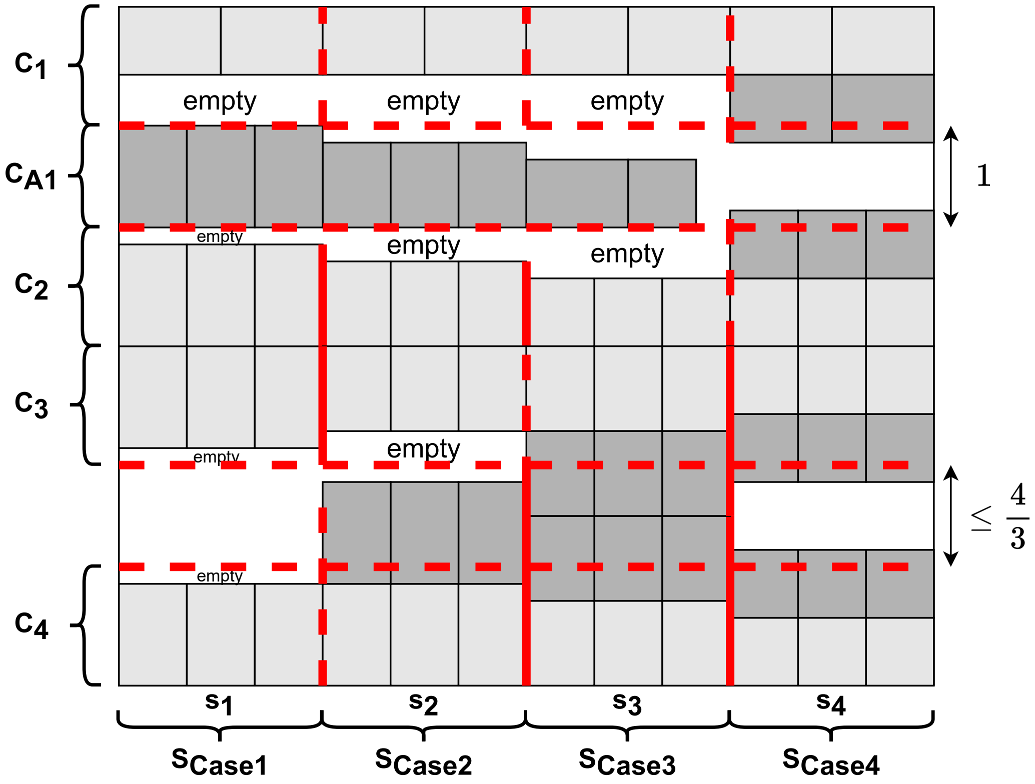


Fig. 14: When $K = 4$ and the fractional packing has exactly four configurations, our algorithm partitions the packing into at most 4 cases.

the rectangles the tops of the topmost rectangles in C_1 must lie on a common line.

The creation of C_{A1} increases the height of the packing by at most 1.

5.3 Case2: $f_{1(i)} + f_{2(i)} + f_{3(i)} + f_{4(i)} > 1$ and $f_{1(i)} + f_{2(i)} + f_{3(i)} \leq 1$

For every section $s_i \in S_{Case2}$, we remove the fractional rectangles in C_1, C_2, and C_3 (see Figure 14), including the parts r_{Case2} for vertically split fractional rectangles. We re-shape the fractional rectangles so that they have the full height of a rectangle of the same type but only a fraction of its width, and then we pack them side-by-side in C_{A1} as described above. Fractional rectangles in C_4 are rounded up.

The creation of C_{A1} and rounding fractional rectangles in C_4 increases the height of the packing by at most 2.

5.4 Case3: $f_{1(i)} + f_{2(i)} + f_{3(i)} > 1$ and $f_{1(i)} + f_{2(i)} \leq 1$

For every section $s_i \in S_{Case3}$, we remove the fractional rectangles in C_1 and C_2 (see Figure 14), including the parts r_{Case3} for vertically split fractional rectangles. We re-shape the fractional rectangles so that they have the full height of a rectangle of the same type but only a fraction of its width, and then we pack them side-by-side in C_{A1} as described above. Fractional rectangles in C_3 and C_4 are rounded up.

Note that we ordered the configurations based on the fractional values of the fractional rectangles in the leftmost section s_i of S_{Case3}, so $f_{1(i)} + f_{2(i)} + f_{3(i)} > 1$ and $f_{1(i)} \leq f_{2(i)} \leq f_{3(i)} \leq f_{4(i)}$. Hence, $f_{4(i)} \geq f_{3(i)} > \frac{1}{3}$; therefore, rounding up the fractional rectangles in C_3 and C_4 increases the height of the packing by at most $\frac{4}{3}$, and when including the height increase caused by creating C_{A1} the total height increase for this case is at most $\frac{7}{3}$.

5.5 Case4: $f_{1(i)} + f_{2(i)} > 1$

For every section $s_i \in S_{Case4}$ the fractional rectangles in C_1 and C_2 are rounded up, increasing the height of the packing by at most 1 (see Figure 14). Additionally, the fractional rectangles in C_3 and C_4 are rounded up, and by the same reasoning shown for S_{Case3}, the height increase is at most $\frac{4}{3}$. Therefore, the height increase for this case is at most $\frac{7}{3}$.

Theorem 4 *If $K = 4$ there is an algorithm that produces an integer packing of height at most $\frac{7}{3}$ plus the value of the solution for linear program (1).*

6 K-Type Algorithm

Our algorithm for the case when $K > 4$ also performs four steps. The first two steps are the same as the cases when $K = 3$ and $K = 4$. However, we perform one additional pre-processing step: if there are any rectangle types whose widths are greater than half the width of the strip, we place these rectangles leftmost within their configurations. Additionally, we order the configurations so that configurations containing these wide rectangles are placed at the bottom of the packing so that two configurations containing wide rectangles of the same type are put in adjacent positions. Observe that this ensures that wide rectangles are whole (see Figure 15).

6.1 Pairing Configurations

If there are an odd number of configurations, let C_0 be the configuration at the top of the packing, and let each subsequently lower configuration be C_1, C_2, ..., C_{K-1}, respectively. Otherwise, if there are an even number of configurations, let them be C_1, C_2, ..., C_K, respectively, from top to bottom. Pair configurations C_{2i-1} and C_{2i} for $i = 1, 2, ..., \lfloor \frac{K}{2} \rfloor$. Add a region $R_{j,j+1}$ of height 1 between each pair of configurations C_j and C_{j+1}, shifting rectangles upwards as necessary, but ensure that the rectangles whose widths are greater than half the width of the strip still remain at the bottom of the packing (see Figure 16). If there is an odd number of configurations, the final configuration (topmost) will simply have all of its fractional rectangles rounded up.

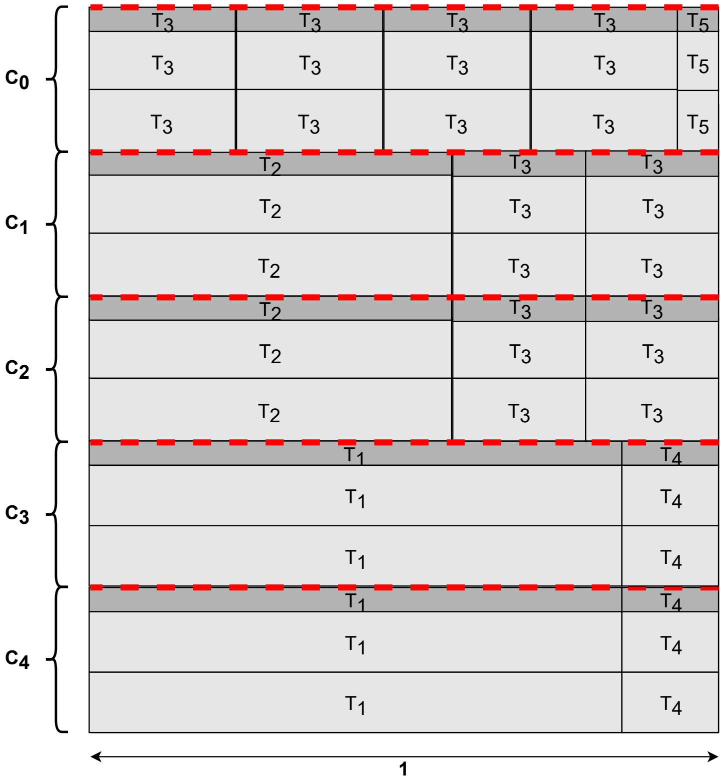


Fig. 15: Configurations with wide rectangle types are adjacent and near the bottom.

For every paired configurations C_j and C_{j+1}, a vertical section s_i is classified as S_{Case1} if $f_{j(i)} + f_{j+1(i)} \leq 1$ and classified as S_{Case2} if $f_{j(i)} + f_{j+1(i)} > 1$.

Note that since rectangle types whose widths are greater than half the width of the strip were packed together at the bottom of the packing and are already whole, these rectangles are not considered for the remainder of this section.

6.2 Processing Fractional Rectangles

Consider paired configurations C_j and C_{j+1}. For any vertically split fractional rectangle $r \in S_{Case1} \cap S_{Case2}$, put the fractional piece r_{Case2} of r located in S_{Case2} into a set F. Round up the remaining fractional rectangles contained in S_{Case2}.

Add to the set F all the fractional rectangles from each vertical section $s_i \in S_{Case1}$. Using fractional rectangles from F, form as many whole rectangles as possible. Note that all fractional rectangles in S_{Common} and S_{Case2} (excluding the pieces r_{Case2}) are rounded up and therefore represent an integer number of whole rectangles of each type. Since there was an integer number of whole rectangles given as input to HMSP, then the fractional rectangles in F must yield an integer number of whole rectangles of each type. Therefore,

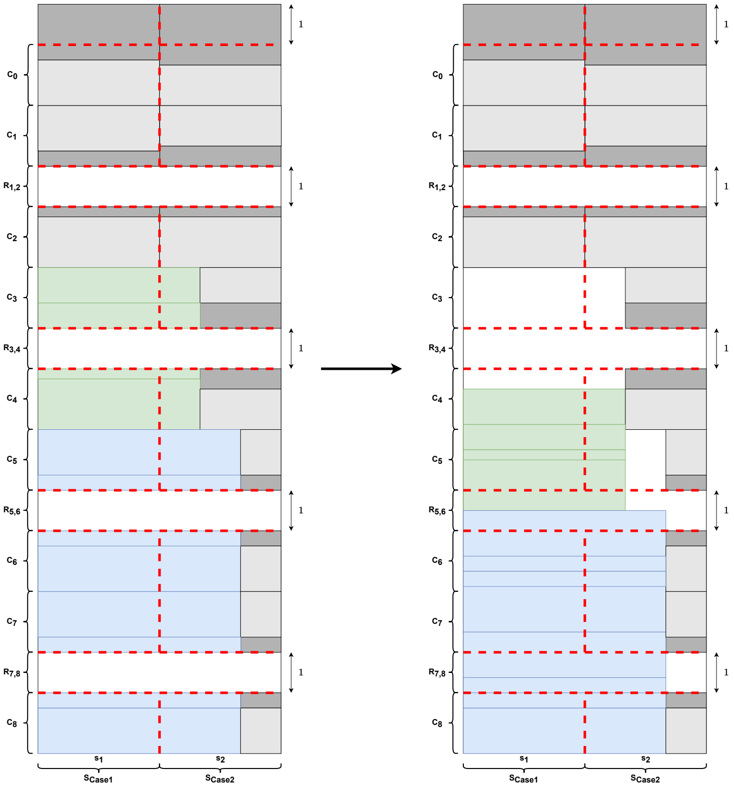


Fig. 16: The K configurations are stacked; if there is an odd number of configurations, the topmost configuration is rounded up instead of paired. After creating the regions $R_{j,j+1}$, the rectangles whose widths are greater than half the width of the strip are again grouped at the bottom of the configuration.

any leftover fractional rectangles in F must have been used to round up other rectangles and can be discarded.

6.3 Packing Rectangles into the Regions $R_{j,j+1}$

Consider one by one the regions $R_{j,j+1}$. Pack the rectangles from F one by one into $R_{j,j+1}$ until the next rectangle r does not fit. Split r and pack in $R_{j,j+1}$ the largest fraction of r that fits; the other piece of r is put back in F. Note that either $R_{j,j+1}$ is completely full (width-wise) or the set F is empty. If F is not empty, continue packing rectangles from F starting with the fractional piece of r, if any, into the remaining regions in the same manner. Note that the rectangles from F must fit within these regions as we did not leave empty space (width-wise) in any region and the total width of the rectangles in F was at most the total width of all the regions combined.

Lemma 11 *After packing the whole rectangles from F into the regions $R_{j,j+1}$ as described above, at most $\lfloor \frac{K}{2} \rfloor - 1$ rectangles were split.*

Proof When rectangles from F are packed into the first region $R_{j,j+1}$, at most one fractional rectangle is leftover (the final rectangle that did not fit in the region). This fractional rectangle combines with the fractional rectangle packed at the beginning of the next region to form a whole rectangle. Combining the fractional rectangle located at the end of a region with the fractional rectangle located at the beginning of the next region accounts for $\lfloor \frac{K}{2} \rfloor - 1$ whole rectangles, as the final region will only have a fractional rectangle at the beginning of the region and not at the end of it. □

6.4 Packing the Split Rectangles

We create additional regions R_1, R_2, ..., $R_{\lfloor \frac{1}{4}K \rfloor}$ at the top of the packing of width equal to the width of the strip to pack the rectangles that were split (see Figure 17). These regions have width 1, the same as the rectangular strip, instead of having just the width of $S_{Uncommon}$. Since the height of S_{Common} is increased by at most 1, then as long as $K > 2$ these regions are located above S_{Common}. Note that S_{Common} could be empty, so that the width of $S_{Uncommon}$ is equal to the width of the full strip.

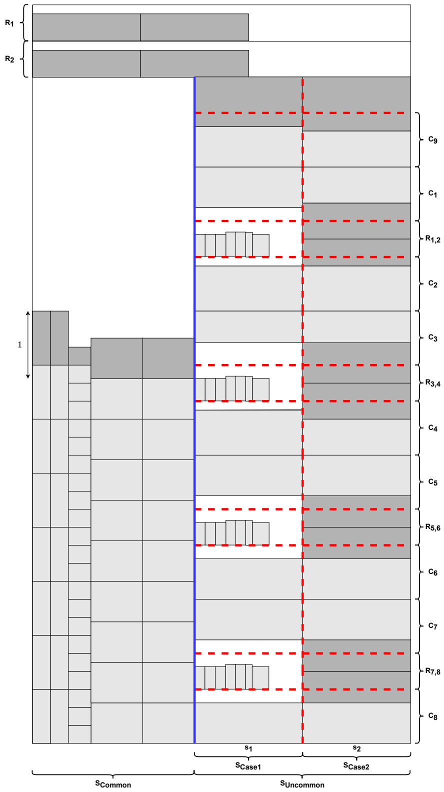


Fig. 17: The configurations have been paired, the fractional rectangles have been processed, and the split rectangles have been made whole and packed in regions placed at the top of the packing.

Lemma 12 *The* $\lfloor \frac{K}{2} \rfloor - 1$ *split rectangles can be packed using at most* $\lfloor \frac{K}{4} \rfloor$ *additional regions of height* 1*.*

Proof Note that none of the split rectangles are wide, hence we can pack at least two of these split rectangles into each region. Since there are fewer than $\lfloor \frac{K}{2} \rfloor$ whole rectangles, packing them at least two to a region will use at most $\lfloor \frac{K}{4} \rfloor$ additional regions. □

Note that if K is even, then the height increase of $S_{Uncommon}$ is at most $\lfloor \frac{K}{2} \rfloor$ from the regions created between each pair of configurations and an additional $\lfloor \frac{K}{4} \rfloor$ from the final regions added to the top of the packing. If K is odd, then the height increase of $S_{Uncommon}$ is at most $\lfloor \frac{K}{2} \rfloor + \lfloor \frac{K}{4} \rfloor + 1$, where the term 1 is from rounding up the un-paired configuration.

Theorem 5 *If* $K > 3$ *then there is an algorithm that produces an integer packing of height at most* $OPT + \lfloor \frac{3}{4}K \rfloor + 1$ *plus the value of the solution for linear program (1).*

7 Experimental Results

We compared our rectangle packing algorithm for the case when the input contains three types of rectangles with the fractional packings produced by solving linear program (1). We implemented our algorithm for 3 rectangle types using Java. The commercial integer and linear program solver Cplex 12.7, configured using default settings, was used to compute optimal fractional solutions. For each test instance we pre-computed the list of possible *base configurations* to provide to the linear program.

Our algorithm for three rectangle types produces integer packings of height at most $\frac{3}{2}h_{max} + \epsilon$ plus the height of the fractional packing where ϵ is a positive constant, but as we show, its experimental performance is much better than its theoretical upper bound.

7.1 Input Data

We used randomly generated sets of rectangles of 3 types to evaluate our algorithm. Note that the running time of our algorithm depends on K, so the number of rectangles in the input does not have much effect on the running time of the algorithm. For each rectangle type, we randomly generate a width, a height, and a multiplicity, but we performed different tests changing the intervals over which we selected the random values. The width and height of the rectangles were always rounded to two decimal places. For every test case we generated one thousand trials.

The structure of the fractional packing impacts how well our algorithm performs. When all of the heights of the fractional rectangles are nearly the full height of their corresponding rectangle types, our algorithm simply rounds them up and so it computes near-optimum solutions. In contrast, when some of the heights of the fractional rectangles are much smaller than the heights of their corresponding rectangle types, our algorithm needs to apply a combination of rounding techniques usually leading to solutions with larger heights. Therefore, when analyzing our experimental results, we divide the test cases into groups based on the structure of the fractional packing.

7.2 Test Cases

We studied the impact that rectangle type width has on the performance of our algorithm. For $w = 1, 2, ..., 10$, we generated packings where the upper bound on the randomly generated widths was $\frac{1}{w}$. For example, when $w = 5$ the widths of the rectangle types were randomly generated from the interval $[0.01, 0.20]$. The running time of our algorithm quickly increases when we decrease the upper bound for the rectangle widths because the need to pre-compute the base configurations to be given as input to Cplex, so we limited the maximum value of w to be 10 for the majority of our test cases. We also performed a smaller number of experiments where the widths of the rectangle types were randomly generated from the interval $[0.01, 0.05]$.

We also studied the impact that rectangle type height has on the performance of our algorithm. For each value of w noted above, we chose height intervals of size 0.10, 0.25, 0.50, and 1. We used the following height intervals (the minimum height is 0.01):

- $[0.9 - 0.1h, 1 - 0.1h]$ for $h = 0, 1, ..., 9$.
- $[0.75 - 0.25h, 1 - 0.25h]$ for $h = 0, 1, 2, 3$.
- $[0.50 - 0.50h, 1 - 0.50h]$ for $h = 0, 1$.
- $[0.01, 1]$.

7.3 Experimental Results

In this section we present only a sample of our experimental results, but the observations that we make in this section will cover all of our experiments [1].

Figure 18 shows how the widths and heights of the rectangle types impact the height of the packings computed by our algorithm. On the x-axis, each label represents the value of h used for that test case, and the height was randomly generated using the interval $[0.9 - 0.1h, 1 - 0.1h]$. On the y-axis, each label represents the mean of the difference between the height of the packing computed by our algorithm and the height of the fractional packing obtained by solving linear program (1); the mean is taken over the thousand tests performed for each value of w and h. Each individual line on the figure represents the change in value of h for a particular value of w (recall that the width of the rectangles was randomly generated using the interval $[0.01, \frac{1}{w}]$). So, looking at the chart from left to right shows results of our test cases of taller

---

[1] The complete results are available at www.csd.uwo.ca/~ablochha/2DHMSPP_Journal_RawData.pdf

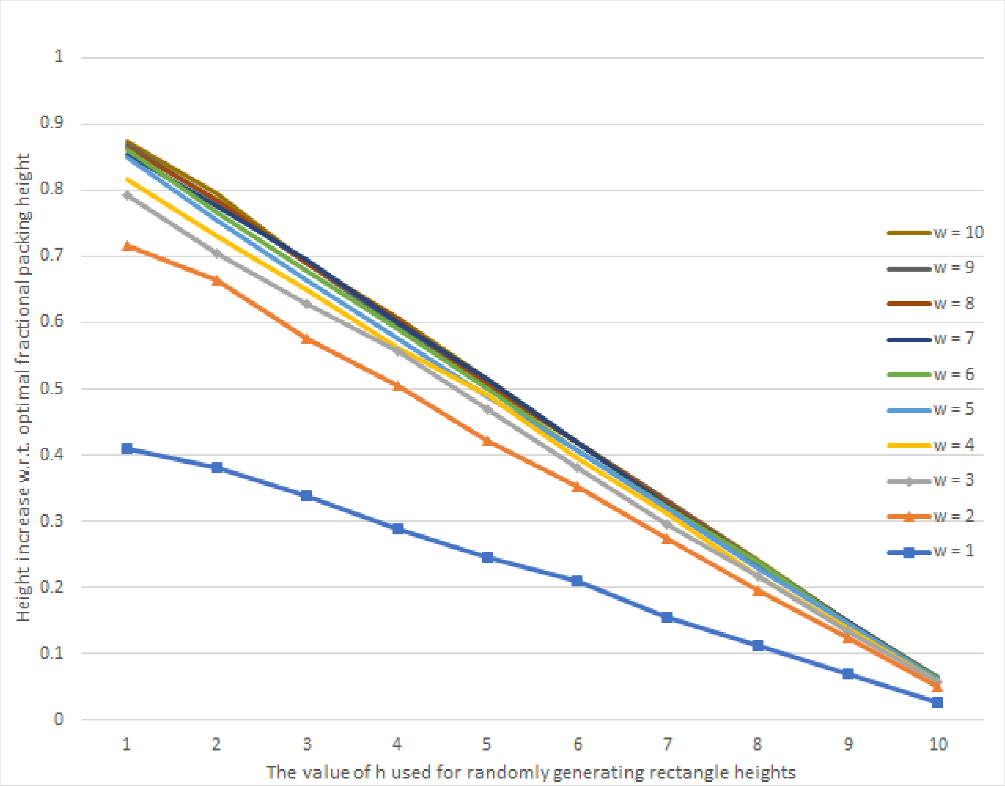


Fig. 18: Average height increase with respect to optimal fractional packing. Each data line represents a different value used for w when selecting the rectangle widths and the x-axis shows the value used for h when selecting the height from the interval $[1-0.1h, 0.9-0.1h]$. From left to right the chart shows taller to shorter rectangles, and from top to bottom the series of lines show narrower to wider rectangles.

to shorter rectangles, and looking at the series of lines from top to bottom shows results of our test cases of narrower to wider rectangles.

Rectangle Heights. The height increase in a packing caused by rounding up a fractional rectangle depends on the height of the rectangle type (see Figure 18). Instances generated using shorter rectangle types resulted in our algorithm producing solutions that were closer to the optimal fractional solutions. The correlation between the height of the rectangles and the height increase of the packing was observed in each of our tests cases. Note that if the heights of all the rectangle types are the same, then the fractional values for each fractional rectangle will also be the same (see the full results), which leads to a simpler problem.

Rectangle Widths. Our results do not include the trivial case when all rectangle types have widths larger than $\frac{1}{2}$; however, we did consider cases when some of the rectangle types have widths larger than $\frac{1}{2}$. Within a particular height interval, the instances that contained rectangles wider than $\frac{1}{2}$ have the lowest height increase with respect to the optimal fractional packing. To see this, observe in Figure 18 the data points corresponding to $h = 1$ on the x-axis;

Test Case	Description	Trials	Avg	Min	Max
1	Configurations have 3, 2, and 1 rectangle types	163	0.913	0.48	1.3
2	Configurations have 3, 1, and 1 rectangle types	41	0.834	0.48	1.16
3	Configurations have 2, 2, and 2 rectangle types	389	0.941	0.34	1.38
4	Configurations have 2, 2, and 1 rectangle types	326	0.877	0.19	1.35
5	Configurations have 2, 1, and 1 rectangle types	71	0.828	0.44	1.26
6	Configurations have 1, 1, and 1 rectangle types	10	0.718	0.15	0.98
7	Our Algorithm Total	1000	0.901	0.15	1.38
8	Simple Algorithm Total	1000	2.041	0.65	2.9

Table 1: One thousand trials. Widths are generated from the interval $[0.01, 0.05]$, heights are generated from the interval $[0.90, 1]$. The results are separated into categories depending on how many different rectangle types appear in each configuration, and within these categories the mean average height increase, minimum height increase, and maximum height increase is listed with respect to the height of an optimal fractional packing. Note that in all 1000 trials the fractional packing contained three configurations.

the bottommost line represents the results where $w = 1$ and the maximum rectangle width was 1, and lines above it represent larger and larger value of w. Our results show that for many of the fractional packings that include one or more rectangle types wider than $\frac{1}{2}$ either each of the configurations contain a single rectangle type, S_{Case2} and S_{Case3} are both empty, or there are fewer than three configurations (see the full results). Each of these situations are simple to solve and most of their solutions increase the height by less than 1.

As we reduced the maximum width allowed for each rectangle type, the solutions computed by our algorithm had heights that were further away from the height of an optimal fractional packing. In the full results we can see that the fraction of the instances that had three configurations increased when we reduced the maximum width (recall that the theoretical upper bound on the height increase is worse for rounding three configurations). For the instances where the maximum rectangle width was $\frac{1}{10}$ and the height was from the interval $[0.90, 1]$, the average height increase of our algorithm was 0.874; in contrast, when the maximum rectangle width was 1 and the height was from the interval $[0.90, 1]$, the average height increase of our algorithm was 0.409.

To get instances that pushed our algorithm towards its theoretical upper bound, we generated inputs that contained rectangle types that are tall and narrow. In Table 1 we show results for a test case where rectangle widths were randomly chosen from the interval $[0.01, 0.05]$ and heights from $[0.90, 1]$. Under the column labeled "Description" we give a description of the data that is included for that test case. For example, test case 1 includes 163 trials in which there was a configuration with three different rectangle types, another configuration with two different rectangle types, and a configuration with only a single rectangle type. Test case 7 includes all of the results from the 1000 trials using our algorithm, while test case 8 includes all of the results from the 1000 trials using a simple algorithm that only rounds up fractional rectangles. Under the "Trials" column, we list the number of instances that are included in each test case, and under the "Avg", "Min", and "Max" columns we list the mean average height increase, minimum height increase, and maximum height increase, respectively, within each test case with respect to the height of an optimal fractional packing. Note that in all 1000 instances shown in Table 1 all fractional packings contained three configurations.

The results shown in Table 1 include some of the largest height increases with respect to the fractional packing that we were able to produce in our experiments. Observe that in test cases 2, 5, and 6, when at least two of the three configurations have only a single rectangle type, the problem becomes simpler: as explained in Section 3.1 the boundaries between the cases are defined by the configuration that has multiple rectangle types, and often one of the configurations with a single rectangle type can be rounded up without increasing the height of the packing by a large amount. As seen in the table, 122 of the 1000 trials had this simpler structure (test cases 2, 5, and 6) and for them the average height increases (0.834, 0.828, and 0.718, respectively) were the lowest within the table.

The instances from Table 1 that have the highest average height increases are in test cases 1 and 3, with height increases of 0.913 and 0.941, respectively. Recall that these test cases are the most complicated versions of the problem and required multiple different strategies to transform the fractional rectangles into whole ones. As seen in the table, 552 of the 1000 trials had this more complicated structure, which represents a majority of the instances. We did not perform additional experiments with even narrower rectangles because of the increased running time.

7.4 Final Observation

We compared our HMSP algorithm for three rectangle types against optimal fractional solutions computed by Cplex. Even though our algorithm has a worst case performance of $1.5 + \epsilon$ plus the height of an optimal fractional packing, its average performance was better. Our algorithm produces solutions that are closest to the optimal fractional packings on instances where the rectangle types are short and wide and produces solutions that are furthest from the optimal where the rectangles are tall and narrow. Moreover, for instances that have at most two configurations our algorithm performs significantly better than when there are three configurations.

References

[1] Baker, B., Coffman, E., and Rivest, R.: Orthogonal packings in two dimensions. SIAM Journal on Computing **9**(4), 846-855 (1980).

[2] Bloch-Hansen, A.: High multiplicity strip packing. Msc. Thesis, Western University (2019).

[3] Bloch-Hansen, A., Solis-Oba, R., Yu, A.: High multiplicity strip packing with three rectangle types. Paper presented at Combinatorial Optimization: 7th International Symposium, ISCO 2022, Online, 2022.

[4] Coffman, E., Garey, M., Johnson, D., and Tarjan, R.: Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing **9**(4), 808-826 (1980).

[5] De La Vega, W., Lueker, G.: Bin packing can be solved within $1 + \epsilon$ in linear time. Combinatorica **1**(4), 349-355 (1981).

[6] De La Vega, F., and Zissimopoulos, V.: An approximation scheme for strip packing of rectangles with bounded dimensions. Discrete Applied Mathematics **82**(1-3), 93-101 (1998).

[7] Garey, M., and Johnson, D.: Computers and intractability. Vol. 174. San Francisco: freeman (1979).

[8] Harren, R., van Stee, R.: Improved absolute approximation ratios for two-dimensional packing problems. Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 177-189. Springer, (2009).

[9] Harren, R., Jansen, K, Prädel, L, Van Stee, R.: A $(\frac{5}{3} + \epsilon)$-approximation for strip packing. Computational Geometry **47**(2), 248–267 (2014).

[10] Karloff, H.: Linear programming. Springer Science & Business Media (2008).

[11] Jansen, K., and Solis-Oba, R.: Rectangle packing with one-dimensional resource augmentation. Discrete Optimization **6**(3), 310-323 (2009).

[12] Karmarkar, N., Karp, R.: An efficient approximation scheme for the one-dimensional bin-packing problem. In: 23rd Annual Symposium on Foundations of Computer Science, 312–320. IEEE (1982).

[13] Kenyon, C., and Rémila, E.: A near-optimal solution to a two-dimensional cutting stock problem. Mathematics of Operations Research **25**(4), 645-656 (2000).

[14] Schiermeyer, I.: Reverse-fit: A 2-optimal algorithm for packing rectangles. European Symposium on Algorithms, 290-299. Springer, Berlin, Heidelberg (1994).

[15] Sleator, D.: A 2.5 times optimal algorithm for packing in two dimensions. Information Processing Letters. **10**(1), 37-40 (1980).

[16] Sviridenko, M.: A note on the Kenyon-Remila strip-packing algorithm. Information Processing Letters. **112**(1-2), 10-12 (2012).

[17] Steinberg, A.: A strip-packing algorithm with absolute performance bound 2. SIAM Journal on Computing **26**(2), 401-409 (1997).

[18] Yu, A.: High Multiplicity Strip Packing Problem With Three Rectangle Types. Msc. Thesis, Western University (2019).

Chapter 4

4 Paper 2: Thief Orienteering on Directed Graphs

A preliminary version of this paper was first published in 2023 (pp. 87-98) in the Proceedings of the 34th International Workshop on Combinatorial Algorithms (IWOCA) by Springer Nature [12]. The Version of Record for the preliminary paper is available online at: https://doi.org/10.1007/978-3-031-34347-6_8. An extended version of this paper was later submitted to the journal of Theoretical Computer Science and is currently under review.

This paper investigates optimization problems that contain multiple interdependent subproblems and presents several approximation algorithms for restricted versions of the thief orienteering problem.

Algorithms for the Thief Orienteering Problem on Directed Acyclic Graphs

Abstract

We consider the scenario of routing an agent called a *thief* through a weighted graph $G = (V, E)$ from a start vertex s to an end vertex t. A set I of items each with weight w_i and profit p_i is distributed among $V \setminus \{s, t\}$. The thief, who has a knapsack of capacity W, must follow a simple path from s to t within a given time T while packing in the knapsack a set of items taken from the vertices along the path of total weight at most W and maximum profit. The travel time across an edge depends on the edge length and current knapsack load.

The thief orienteering problem (ThOP) is a generalization of the orienteering problem, the longest path problem, and the 0-1 knapsack problem. We prove that there exists no approximation algorithm for ThOP with constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$, and we present a polynomial-time approximation scheme (PTAS) for ThOP when G is directed and acyclic that produces solutions that use time at most $T(1 + \epsilon)$ for any constant $\epsilon > 0$. We also present a fully polynomial-time approximation scheme (FPTAS) for ThOP on arbitrary undirected graphs where the travel time depends only on the lengths of the edges and T is the length of a shortest path from s to t plus a constant K. Finally, we present a FPTAS for a restricted version of the problem where the input graph is a clique.

Keywords: thief orienteering problem, knapsack problem, dynamic programming, approximation algorithm, approximation scheme

1. Introduction

The *thief orienteering problem* (ThOP) is defined as follows. Let $G = (V, E)$ be a weighted graph with n vertices, where two vertices $s, t \in V$ are designated the *start* and *end* vertices, and every edge $e = (u, v) \in E$ has a length $d_{u,v} \in \mathbb{Q}^+$. In addition, let there be a set I of items, where each item $i_j \in I$ has a non-negative integer weight w_j and profit p_j. Each vertex $u \in V \setminus \{s,t\}$ stores a subset $S_u \subseteq I$ of items such that $S_u \cap S_v = \emptyset$ for all $u \neq v$ and $\bigcup_{u \in V \setminus \{s,t\}} S_u = I$. There is an agent called a thief that has a knapsack with capacity $W \in \mathbb{Z}^+$ and the goal of the problem is for the thief to travel a simple path from s to t within a given time $T \in \mathbb{Q}^+$ while collecting items in the knapsack taken from the vertices along the path of total weight at most W and maximum total profit.

The amount of time needed to travel between two adjacent vertices u, v depends on the length of the edge connecting them and on the weight of the items in the knapsack when the edge is traveled; specifically, the travel time between adjacent vertices u and v is $d_{u,v}/\mathcal{V}$ where $\mathcal{V} = \mathcal{V}_{\max} - w(\mathcal{V}_{\max} - \mathcal{V}_{\min})/W$, w is the current weight of the items in the knapsack, and $\mathcal{V}_{\min}$ and $\mathcal{V}_{\max}$ are the minimum and maximum velocities of the thief.

ThOP is a generalization of the orienteering problem [9], the longest path problem, and the 0-1 knapsack problem, so it is NP-hard. The problem was first formulated by Santos and Chagas [16] in 2018, and they provided the first two heuristics for ThOP: An iterated local search algorithm and a biased random-key genetic algorithm. In 2020, Faêda and Santos [6] presented a genetic algorithm for ThOP. Chagas and Wagner [4] designed a heuristic using an ant colony algorithm which they later further improved [5].

While ThOP is a relatively new problem, the closely related family of travelling problems, such as the travelling thief problem [3] and some variants of orienteering [17], are well-studied and have applications in areas as diverse as route planning [7, 8, 12], circuit design [3], and logistics [11], among others.

In 2017, Polyakovskiy and Neumann [14] introduced the packing while travelling problem (PWTP). This is a problem similar to ThOP, but in PWTP the thief must follow a fixed path and the goal is to maximize the difference between the profit of the items collected in the knapsack and the transportation cost. Polyakovskiy and Neumann provided two exact algorithms for PWTP, one based on mixed-integer programming and another on branch-infer-and-bound. In 2019, Roostapour et al. [15] presented three evolutionary algorithms on variations of PWTP. Most recently, Neumann et al. [13] presented an exact dynamic programming algorithm and a fully polynomial-time approximation scheme (FPTAS) for PWTP. Their dynamic programming algorithm, unfortunately, cannot be applied to ThOP because in ThOP we must bound both the total weight of the items and the travelling time of the thief.

To the best of our knowledge, study on ThOP to this date has focused on the design of heuristics and no previous work has presented an approximation algorithm for it.

In this paper we show that ThOP cannot be approximated within a constant factor unless $\mathsf{P} = \mathsf{NP}$ and we present a dynamic programming-based polynomial-time approximation scheme (PTAS) for ThOP when the input graphs are directed acyclic graphs (DAGs) and the time used by the thief is at most $T(1+\epsilon)$ for any constant $\epsilon > 0$.

There are several challenges involved in the design of a PTAS for ThOP on DAGs. To achieve polynomial running time the parameters of the problem (weight, profit, and travelling time) need to be rounded to keep the size of the dynamic programming table polynomial. Rounding weights needs to be done carefully because (1) rounding the weights of small weight items with high profit can yield solutions with profit much smaller than the optimum one and (2) rounding the weights of items located far away from the destination vertex can cause large errors in the travelling times.

We solve the first problem through enumeration by ensuring that a constant number of the items with largest profit in an optimum solution belong also to the solution computed by our algorithm. We solve the second problem by using actual item weights and not rounded item weights when computing travelling times, and by allowing the thief slightly more time to travel from s to t.

Variations of our algorithm yield FPTAS on special versions of ThOP on arbitrary undirected graphs, like (i) the case when $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, edge lengths are integer, and T is equal to the length L of a shortest path from s to t plus a constant K, and (ii) the case when $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, edges have unit length, and the input graph is a clique. This latter case is a generalization of the knapsack problem when the items are partitioned into groups and we must select items only from a certain number of these groups.

The rest of the paper is organized in the following way. We begin in Section 2 by proving the inaproximability of ThOP. In Section 3 we present an exact algorithm for ThOP on DAGs using dynamic programming and in Section 4, we transform this algorithm into a PTAS that produces solutions that use time at most $T(1+\epsilon)$. In Section 5 we show our algorithm can be converted to a FPTAS for a restricted version of ThOP on undirected graphs. In Section 6 we present our FPTAS for ThOP on cliques. A preliminary version of this paper appeared in [1], and this extended version includes the complete proofs for the lemmas and theorems that were omitted in [1], including a correction to Lemma 2 and Theorem 3 as our algorithm needs time $T(1+\epsilon)$ and not time T. We also include the algorithm for the case when the input graph is a clique with unit length edges.

2. Inapproximability

We first show that even when the input graph is a clique, each vertex has at most one item, and edge lengths are only 1 or 2, ThOP has no PTAS unless $\mathsf{P} = \mathsf{NP}$. To prove this we show that ThOP is a generalization of the rooted orienteering problem.

In the *rooted orienteering problem* (OP) we are given a weighted complete graph $G = (V, E)$ with metric distances $w : E \to \mathbb{Z}^+$, vertex profits $\pi : V \to \mathbb{Z}_{\geq 0}$, root vertex $s' \in V$, and a path length limit $L \in \mathbb{Z}_{\geq 0}$. The goal in this problem is to compute a path starting at s' of total length at most L for which the sum of the profits of the vertices in the path is maximum.

Theorem 1. *There is no PTAS for ThOP unless* $\mathsf{P} = \mathsf{NP}$. *This is true even when the input graph is a clique, every vertex has at most one item, and edge lengths are only* 1 *or* 2.

Proof. Blum et al. [2] proved that there is no PTAS for OP even if the input graph is a clique, edge lengths are 1 or 2, and the root vertex has zero profit. Consider an instance of this restricted version of OP. We create an instance of ThOP on $\{1, 2\}$-metrics, by performing the following steps:

1. Set s to be the root vertex and add an ending vertex t to the input graph $G = (V, E)$ of OP.
2. Add an edge from every vertex of G to t of length 1.
3. Assign velocities $v_{max} = v_{min} = 1$, knapsack capacity $W = |V|$, and time limit $T = L + 1$.
4. Finally, for each vertex $v \in V$ where profit $\pi(v) > 0$, add an item i to I with weight $w_i = 1$ and profit $p_i = \pi(v)$, and place i as the only item in the subset I_v of items associated with vertex v.

A solution for this instance of ThOP yields a solution of the same value for the corresponding instance of OP, as a solution for ThOP includes all items stored at the vertices of the selected path. Hence, a PTAS for ThOP would be a PTAS for OP. $\square$

To build to our main result in this section, we consider the optimization version of the classic *longest path problem*. In this problem, we are given an arbitrary simple connected unweighted graph $G = (V, E)$, and the goal is to compute a path P in G with a maximum number of edges. The longest path problem has no approximation algorithm with constant approximation ratio, unless $\mathsf{P} = \mathsf{NP}$ [10]. Furthermore, for any $\epsilon > 0$, there is no algorithm for the longest path problem with approximation ratio $O(\log^{1-\epsilon} n)$, unless $\mathsf{NP} \in \mathsf{DTIME}(2^{O(\log^{1-\epsilon} n)})$.

Theorem 2. *There is no approximation algorithm for ThOP with constant approximation ratio, unless* $\mathsf{P} = \mathsf{NP}$*. Furthermore, for any* $\epsilon > 0$*, there is no approximation algorithm for ThOP with approximation ratio* $2^{O(\log^{1-\epsilon} n)}$ *unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(2^{O(\log^{1-\epsilon} n)})$*. These hardness results hold even if the input graph has bounded degree, all edges have unit length, and each vertex stores only one item of unit weight and profit.*

Proof. The longest path problem [10] is a special case of ThOP, where s and t are the endpoints of a longest path in the input graph $G = (V, E)$, every edge has length 1, every vertex $u \in V \setminus \{s, t\}$ stores one item of weight 1 and profit 1, the capacity of the knapsack is $W = |V| - 2$, the bound on the time is $T = |V| - 1$, and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$. Since there are $O(|V|^2)$ possible choices for the endpoints of a longest path in G then the inapproximability properties of the longest path problem [10] apply also to ThOP. □

The fractional version of ThOP allows the thief to select a fraction of each item.

Corollary 1. *The fractional version of ThOP cannot be approximated in polynomial time within any constant factor unless* $\mathsf{P} = \mathsf{NP}$.

Proof. Consider the same reduction as in the proof of Theorem 2. Note than an optimal fractional solution must collect the whole items stored in the vertices of an optimal path. □

3. Algorithm for Thief Orienteering on DAGs

To simplify the description of our algorithms, for each vertex u that does not store any items we add to I a new dummy item of weight 0 and profit 0 and store it in u; hence, every vertex stores at least one item (see Figure 1).

We can assume that the minimum and maximum velocities $\mathcal{V}_{\min}$ and $\mathcal{V}_{\max}$ are δ and 1, respectively, where $\delta \in \mathbb{Q}^+$ and $\delta \leq 1$. Then, the *travel time* from vertex u to vertex v when the knapsack's total weight is w just prior to leaving vertex u is equal to $d_{u,v}/\eta$, where $\eta = 1 - \frac{(1-\delta)w}{W}$.

Since DAGs have no cycles, every path from s to t is a simple path. We index the vertices of the graph using a topological ordering. We delete from G all vertices that are unreachable from s and all vertices that cannot reach t. For a vertex u, let $u.index$ be the index of the vertex as determined by the topological ordering.

Note that s and t have the lowest and highest indices, respectively. Additionally, observe that the indices of the vertices encountered along a simple path from s to any vertex u appear in increasing order. We index the items stored in the vertices so that item indices are unique and items in vertex u have smaller indices than items in vertex v if $u.index < v.index$. Let the items stored in the vertices of the input graph G be $i_1, i_2, ..., i_{|I|}$.

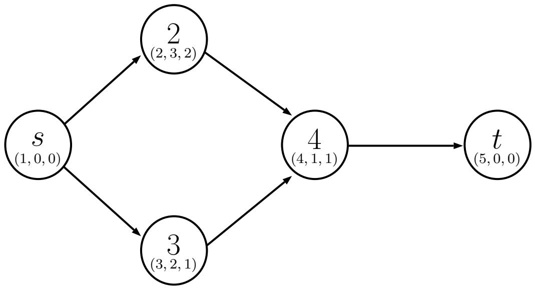


Figure 1: An example of a DAG that has two possible paths from s to t. In each vertex u there is a triplet (index, profit, weight) for each item in I_u.

We define the *parents* of a vertex u to be the vertices v such that (v, u) is a directed edge of G. Our algorithm for ThOP on DAGs is shown in Algorithm 1.

3.1. Profit Table

Let S be a subset of items. In the sequel, we define the *travel time* of S to a vertex u as the minimum time needed by the thief to collect all items in S while travelling along a simple path p_{su} from s to u that includes all the vertices storing the items in S. Path p_{su} is called a *fastest path* of S to u. Additionally, we define the *total travel time* of S as the travel time of S to t

Algorithm 1 ThOPDAG$(G = (V, E), W, T, s, t, I, \delta)$

```
Input: DAG $G$, knapsack capacity $W$, time limit $T$, start vertex $s$, end vertex $t$, vertex item assignments $I$, and minimum velocity $\delta$.
Output: An optimum solution for ThOP.
Delete from $G$ vertices unreachable from $s$ and vertices that cannot reach $t$.
Compute a topological ordering for $G$.
Index $V$ by increasing topological ordering and index items as described above.
Let $A$ be an empty profit table. Set $A[1] = (0, 0, 0, -1)$.
for $i = s.index$ to $t.index$ do
    Let $u$ be the vertex with index $i$.
    Call $UpdateProfitTable(W, T, \delta, A, u)$.
end for
Return $BuildKnapsack(A, I)$.
```

and the *total travel time of* S *through* u as the travel time of S to t using a simple path that includes u.

Definition 1. *Let* $S_z = \{i_1, ..., i_z\}$ *for all* $z = 1, ..., |I|$, *and let vertex* u *hold item* i_z. *A subset* S *of* S_z *is a feasible subset with respect to* u *if it has weight* $w_S = \sum_{i_j \in S} w_j \leq W$ *and total travel time through* u *at most* T.

Our algorithm builds a *profit table* A where each entry $A[j]$, for $j = 1, ..., |I|$, corresponds to the item i_j with index j, and $A[j]$ is a list of tuples $(w, p, time, prev)$. Let u be the vertex that contains item i_j; a tuple $(w, p, time, prev)$ in the list of $A[j]$ indicates that there is a subset S of S_j such that:

- the weight of the items in S is $w \leq W$,
- the profit of the items in S is p,
- the travel time of S to u is $time \leq T$,
- a fastest path of S to u includes a vertex storing i_{prev}. Note that item i_{prev} does not need to be in S.

A tuple $(w, p, time, prev)$ *dominates* a tuple $(w', p', time', prev')$ if $p \geq p'$, $w \leq w'$, and $time \leq time'$. We remove dominated tuples from each list of A such that no tuple in the list of each entry $A[j]$ dominates another tuple in the same list. Therefore, we can assume each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their profits, (ii) there might be multiple tuples with the same profit and these tuples are sorted in non-decreasing order of their weights, and (iii) if several tuples in $A[j]$ have the same profit p and weight w, only the tuple with the smallest value of $time$ is kept in $A[j]$.

3.2. UpdateProfitTable

Algorithm 2 shows how we update the profit table with the items stored in vertex u. The start vertex $s \in V$ has no parents and holds a single item i_1 of weight and profit 0; therefore, we initialize $A[1]$ to store the tuple $(0, 0, 0, -1)$.

When two (or more) different paths from s to t are routed through some intermediate vertex u, it is vital that subsets of items corresponding to each of the paths are recorded correctly: The entries in the profit table A must represent the item subsets of each path from s to u, but none of the tuples in A should contain information from items stored in vertices from disjoint sections of two different paths.

Every item is stored in a unique vertex; therefore, by including in each tuple $(w, p, time, prev)$ the value $prev$, which indicates that the tuple was built using a tuple from the entry $A[prev]$ corresponding to item i_{prev}, the profit table stores the necessary information to determine which of the parents of a vertex u were used to create the tuples at entries $A[j]$ of the items i_j stored in u.

Observe that travel times need to be computed when creating the tuples for the first item of a vertex u; however, this travel time is the same for the tuples corresponding to the other items stored in u.

3.2.1. Selecting a Solution from the Profit Table

Recall that the vertex with the largest index is t; therefore, t is visited last by Algorithm 1 and its last item $i_{|I|}$ corresponds to the final entry in the profit table A. Therefore, after executing Algorithm 1 the last entry in A contains the list of all dominating tuples whose weights are at most W and total travel times are at most T. Within this list there exists a tuple $(w^*, p^*, time^*, prev^*)$ with maximum profit, and this tuple represents a

Algorithm 2 UpdateProfitTable(W, T, δ, A, u)

```
Input: Knapsack capacity $W$, time limit $T$, minimum velocity $\delta$, profit
  table $A$, and vertex $u$.
Output: The entries of the profit table $A$ corresponding to $u$'s items are
  updated to represent the subsets of items found along all paths from $s$
  to $u$.
for each item $i_j$ of $u$ do
    Let $w$ be the weight and $p$ the profit of $i_j$.
    if $i_j$ is $u$'s first item then
        $A[j] = \emptyset$.
        for each parent $v$ of $u$ do
            Let $id_v$ be the index of $v$'s last item.
            for each $(w', p', time', prev') \in A[id_v]$ do
                Let $d_{u,v}$ be the distance from $v$ to $u$.
                Let $\eta = 1 - (1 - \delta)w'/W$.
                Let $travel = \frac{d_{u,v}}{\eta}$.
                if $time' + travel \leq T$ then
                    Append $(w', p', time' + travel, id_v)$ to $A[j]$.
                end if
            end for
        end for
    else
        Copy all tuples of $A[j-1]$ into $A[j]$.
        For each tuple $(w', p', time', prev')$ in $A[j]$, $prev' = j - 1$.
    end if
    for each $(w', p', time', prev') \in A[j]$ do
        if $w + w' \leq W$ then
            Append $(w + w', p + p', time', prev')$ to $A[j]$.
        end if
    end for
    Remove dominated tuples from $A[j]$.
end for
```

feasible subset S of $S_{|I|}$ with respect to t with maximum profit. Algorithm 3 shows how to recover S from the information stored in the profit table.

Algorithm 3 considers one item at a time, starting with t's last item $i_{|I|}$ and determines whether $i_{|I|}$ belongs in the solution. The next item considered

Algorithm 3 BuildKnapsack(A, I)

```
Input: Profit table $A$ and vertex item assignments $I$.
Output: The subset of items along a simple path from $s$ to $t$ with the most profit.
$path = \emptyset$, $knapsack = \emptyset$.
Let $j = |I|$, the index of $t$'s last item, and let $u = t$.
Let $(w, p, time, prev)$ be the tuple with maximum profit from $A[j]$.
while $j \geq 0$ do
    Let $\eta = 1 - (1 - \delta)w/W$.
    If $u \notin path$, prepend $u$ to $path$.
    Let $w_j$ be the weight of item $i_j$ with index $j$ and $p_j$ be its profit.
    if $u$ is the vertex where the item with index $j - 1$ is located then
        if a tuple $(w, p, time, prev')$ is in $A[j - 1]$ then
            $(w, p, time, prev) = (w, p, time, prev')$.
        else
            Append the item $i_j$ with index $j$ to $knapsack$.
            $(w, p, time, prev) = (w - w_j, p - p_j, time, prev')$.
        end if
        $j = j - 1$.
    else
        Let $u_\alpha$ be the vertex where item $prev$ is located.
        Let $d_{u_\alpha,u}$ be the distance from $u_\alpha$ to $u$.
        if a tuple $(w, p, time' = time - \frac{d_{u_\alpha,u}}{\eta}, prev')$ is in $A[prev]$ then
            $j = prev$.
            $(w, p, time, prev) = (w, p, time', prev')$.
        else
            Append the item $i_j$ with index $j$ to $knapsack$.
            $j = prev$.
            $(w, p, time, prev) = (w - w_j, p - p_j, time - \frac{d_{u_\alpha,u}}{\eta}, prev')$.
        end if
    end if
end while
return $path$.
```

by the algorithm is i_{prev^*}, and so on. Consider an item i_j located at some vertex u. Let entry $A[j]$ contain a tuple $\tau = (w, p, time, prev)$. If item i_{j-1} is also located at u, then a tuple $(w, p, time, prev')$ in $A[j-1]$ indicates that i_j is not in the (partial) solution corresponding to tuple τ. If $A[j-1]$ corresponds to an item not located at u, then let the vertex u_α store item i_{prev}; then a tuple $(w, p, time', prev')$ in $A[prev]$ with $time' = time - \frac{d_{u_\alpha,u}}{\eta}$ indicates that i_j was not used to create tuple τ.

If there is no tuple $(w, p, time', prev')$ in $A[j-1]$ (or $A[prev]$) as described above then i_j belongs in the (partial) solution corresponding to τ.

3.3. Algorithm Analysis

Recall that the items are indexed such that for two items with indices h and j where $h < j$, item i_h must belong to a vertex whose index is less than or equal to the index of the vertex containing item i_j.

To prove that our algorithm is correct we must prove that each entry $A[z]$ of the profit table is such that for every feasible subset S of S_z with respect to the vertex holding item i_z the entry $A[z]$ contains either (i) a tuple $(w_S, p_S, time_S, prev)$, where $w_S = \sum_{i_j \in S} w_j$, $p_S = \sum_{i_j \in S} p_j$, and $time_S$ is the travel time of S to the vertex u storing i_z, or (ii) a tuple $(w', p', time', prev')$ that dominates $(w_S, p_S, time_S, prev)$. This implies that A contains a tuple representing a simple path from s to t whose vertices store a maximum profit set S^* of items of weight at most W and total travel time at most T.

Lemma 1. *Let* $1 \leq z \leq |I|$ *and let vertex* u *hold item* i_z. *For each feasible subset* S *of* S_z *with respect to* u *there is a tuple* $(w, p, time, prev)$ *in the profit table* A *at entry* $A[z]$ *such that* $w \leq w_S = \sum_{i_j \in S} w_j$ *and* $p \geq p_S = \sum_{i_j \in S} p_j$.

Proof. We use a proof by induction on the number of entries of the profit table A. The base case is trivial as there are only two feasible subsets of S_1 with respect to s, and so $A[1]$ stores the tuple $(0, 0, 0, -1)$.

Assume that the lemma holds true for every feasible subset S of S_q with respect to the vertex holding item i_q for all $q = 1, ..., z-1$. Let S be a feasible subset of S_z with respect to the vertex holding item i_z; we show that there is a tuple $(w, p, time, prev) \in A[z]$ such that $w \leq w_S$ and $p \geq p_S$. Let u be the vertex storing i_z. We consider two cases:

- Case 1: Item $i_z \in S$. Let $S' = S - \{i_z\}$.
  - If i_z is u's first item, let u_α be a parent of u, let i_α be the last item at u_α, and let $time_\alpha$ be the travel time of S' from u_α to u. By the induction hypothesis, there is a tuple $(w', p', time', prev')$ in $A[\alpha]$ such that $w' \leq w_{S'}$ and $p' \geq p_{S'}$. Lines 7 to 17 and 22

to 26 in Algorithm 2 consider all parents u_α of u and add tuples $(w' + w_z, p' + p_z, time' + time_\alpha, prev')$ to $A[z]$ where $w' + w_z \leq w_{S'} + w_z = w_S$, $p' + p_z \geq p_{S'} + p_z = p_S$, and $time' + time_\alpha \leq T$.

  - If i_z is not u's first item, then item i_{z-1} is also located at u. By the induction hypothesis, there is a tuple $(w', p', time', prev')$ in $A[z-1]$ such that $w' \leq w_{S'}$ and $p \geq p_{S'}$. Lines 19 to 26 in Algorithm 2 add the tuple $(w' + w_z, p' + p_z, time', prev')$ to $A[z]$.

- Case 2: Item $i_z \notin S$. Since S is a feasible subset with respect to u, by the induction hypothesis, either (i) there is a tuple $(w, p, time, prev)$ in $A[z-1]$ such that $w \leq w_S$ and $p \geq p_S$ that Algorithm 2 would have copied to $A[z]$ in lines 19-20 if i_z is not u's first item, or (ii) there is a tuple $(w, p, time, prev)$ in $A[\alpha]$ (where i_α is the last item in some vertex u_α as defined above) such that $w \leq w_S$ and $p \geq p_S$ that Algorithm 2 would have copied to $A[z]$ in lines 7-17 if i_z is u's first item. $\square$

4. PTAS for Thief Orienteering on DAGs

Algorithm 1 might not run in polynomial time because the size of the profit table might become too large. We can convert Algorithm 1 into a PTAS by carefully rounding down the profit and rounding up the weight and travel times associated with each item.

Note that if we simply use rounded weights in the profit table described in Section 3.1, then we might introduce a very large error to the travel times computed by Algorithm 2. To prevent this, we modify the profit table so that every entry $A[j]$ holds a list of tuples $(w_r, w_t, p, time_r, prev)$, where each tuple indicates that there is a subset S of S_j in which the sum of their rounded weights is w_r, the sum of their true weights is w_t, the sum of their rounded profits is p, the rounded travel time of S to the vertex u holding item i_j is $time_r$, and the path of S to u corresponding to this tuple includes the vertex which contains i_{prev}. Let P_{max} be the maximum profit of an item that can be transported within the allotted time T from the vertex that initially stored it to the destination vertex. Our PTAS is described in Algorithm 4.

Algorithm *UpdateProfitTable** is a slight modification of Algorithm 2 that is described in Algorithm 5. Travel times are rounded up to the nearest multiple of $\frac{\epsilon T}{n}$.

A tuple $(w_r, w_t, p, time_r, prev)$ dominates a tuple $(w'_r, w'_t, p', time'_r, prev')$ if $p \geq p'$, $w_r \leq w'_r$, $time_r \leq time'_r$, and $w_t \leq w'_t$. Therefore, we can assume

Algorithm 4 ThOPDAGPTAS$(G = (V, E), W, T, s, t, I, \delta, \epsilon)$

```
Input: DAG $G$, knapsack capacity $W$, time limit $T$, start vertex $s$, end vertex $t$, item assignments $I$, minimum velocity $\delta$, and constant $\epsilon > 0$.
Output: A solution for ThOP of profit at least $(1 - 3\epsilon)OPT$ that uses time at most $T(1 + \epsilon)$, where $OPT$ is the value of an optimum solution.
Delete from $G$ vertices unreachable from $s$ and vertices that cannot reach $t$.
Compute a topological ordering for $G$.
Index $V$ by increasing topological ordering and index items as described in Section 3.
Let $K = \frac{1}{\epsilon}$.
Let $\mathbb{S}$ be the set of all feasible subsets $S$ of $S_{|I|}$ with respect to $t$ such that $|S| \leq K$.
for each $S \in \mathbb{S}$ do
    Let $A$ be an empty profit table. Set $A[1] = (0, 0, 0, 0, -1)$.
    Let $W' = W - \sum_{i_j \in S} w_j$.
    Round down the profit of each item in $I - S$ to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$.
    Round up the weight of each item in $I - S$ to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$.
    for $i = s.index$ to $t.index$ do
        Let $u$ be the vertex with index $i$.
        Call $UpdateProfitTable^*(W, T, \delta, A, u, S)$.
    end for
end for
Select the profit table $A^*$ storing the tuple with maximum profit.
Return $BuildKnapsack(A^*, I)$.
```

each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their rounded profits, (ii) there might be multiple tuples with the same rounded profit and these tuples are sorted in non-decreasing order of their rounded weights, (iii) there might be multiple tuples with the same rounded weight and these tuples are sorted in non-decreasing order of their rounded travel times, and (iv) if several tuples in $A[j]$ have the same rounded values of profit, weight, and travel time, only the tuple with the smallest true weight is kept in $A[j]$.

Algorithm 5 UpdateProfitTable*(W, T, δ, A, u, S)

```
Input: Knapsack capacity $W$, time limit $T$, minimum velocity $\delta$, profit
  table $A$, vertex $u$, and item subset $S$.
Output: The entries of the profit table $A$ corresponding to $u$'s items are
  updated to represent the subsets of items found along all paths from $s$
  to $u$.
for each item $i_j$ of $u$ do
    Let $w_t$, $w_r$, and $p_r$ be the weight, rounded weight, and rounded profit
      of $i_j$, respectively. If $i_j \in S$, set $w_r = w_t$ and $p_r =$ profit of $i_j$.
    if $i_j$ is $u$'s first item then
        $A[j] = \emptyset$.
        for each parent $v$ of $u$ do
            Let $id_v$ be the index of $v$'s last item.
            for each $(w'_r, w'_t, p'_r, time'_r, prev') \in A[id_v]$ do
                Let $d_{u,v}$ be the distance from $v$ to $u$.
                Let $\eta = 1 - (1 - \delta)w'_t/W$.
                Let $travel = \frac{d_{u,v}}{\eta}$.
                if $time'_r + travel \leq T(1 + \epsilon)$ then
                    Append $(w'_r, w'_t, p'_r, t'_r, id_v)$ to $A[j]$, where $t'_r$ is the
                      nearest multiple of $\frac{\epsilon T}{n}$ that is larger than or equal to
                      $time'_r + travel$.
                end if
            end for
        end for
    else
        Copy all tuples of $A[j - 1]$ into $A[j]$.
        For each $(w'_r, w'_t, p'_r, time'_r, prev')$ in $A[j]$, set $prev' = j - 1$.
    end if
    for each $(w'_r, w'_t, p'_r, time'_r, prev') \in A[j]$ do
        if $w_r + w'_r \leq W$ then
            Append $(w_r + w'_r, w_t + w'_t, p_r + p'_r, time'_r, prev')$ to $A[j]$.
        end if
    end for
    if $i_j \in S$ then remove tuples that do not include $i_j$ from $A[j]$.
    Remove dominated tuples from $A[j]$.
end for
```

4.1. PTAS Analysis

Since the weights of items are rounded up, the solution produced by Algorithm 4 might have unused space where the algorithm could not fit any rounded items. However, an optimal solution would not leave empty space if there were items that could be placed in the knapsack while still travelling from s to t in at most T time, so we need to bound the maximum profit lost due to the rounding of the weights and profits, and we also need to bound the increase in travel time caused by the rounding.

Lemma 2. *For any constant* $\epsilon > 0$*, Algorithm 4 computes a feasible solution with profit at least* $(1-3\epsilon)OPT$ *that uses travel time at most* $(1+2\epsilon)T$*, where* OPT *is the profit of an optimum solution.*

Proof. Let S_{OPT} be the set of items in an optimum solution and let $OPT = \sum_{i_j \in S_{OPT}} p_j$. If $|S_{OPT}| \leq K$ then Algorithm 4 computes an optimum solution; hence, for the rest of the proof we assume that $|S_{OPT}| > K$. Let S_K be the set of the K items with largest profit from S_{OPT}, where $K = \frac{1}{\epsilon}$, and let $W' = W - \sum_{i_j \in S_K} w_j$. Let S_A be the set of items in the solution selected by our algorithm, and let $SOL = \sum_{i_j \in S_A} p_j$.

In the sequel, we will use w'_j and p'_j to refer to the rounded weight and profit of item i_j, and w_j and p_j to refer to the true weight and profit of item i_j. Given a set X of items let $weight(X) = \sum_{i_j \in X} w_j$, $weight'(X) = \sum_{i_j \in X} w'_j$, $profit(X) = \sum_{i_j \in X} p_j$, and $profit'(X) = \sum_{i_j \in X} p'_j$. We let $profit'(S_K) = profit(S_K)$ and $weight'(S_K) = weight(S_K)$.

For this proof a subset S of $S_{|I|}$ is feasible if $weight'(S) \leq W$ and the total travel time of S using the real weights of the items in S (not their rounded weights) is at most $(1+\epsilon)T$. Recall that our algorithm considers solutions that include every possible feasible subset of at most K items in the knapsack. Therefore, our algorithm must have included S_K in the knapsack in one of the iterations and filled the remainder of the knapsack using the profit table. Let S^*_A be the solution computed by our algorithm in the iteration where it chose to include the items of S_K in the knapsack, and let $SOL^* = \sum_{i_j \in S^*_A} p_j$. Since our algorithm returns the best solution that it found over all iterations, then $SOL \geq SOL^*$.

To compare SOL^* to OPT, we round up the weight of each item in $S_{OPT} - S_K$ to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$; note that the weights and profits of the items in S_K are not rounded.

Rounding up the weight of a single item increases the weight of that item by at most $\frac{\epsilon W'}{|I|^2}$, so $weight'(S_{OPT}) \leq weight(S_{OPT}) + \frac{\epsilon W'}{|I|} \leq W + \frac{\epsilon W'}{|I|}$ as $|S_{OPT}| \leq |I|$. Let A_{OPT} be a subset of $S_{OPT} - S_K$ with $weight'(A_{OPT}) \leq W'$ and maximum rounded profit.

Note that $A_{OPT} \cup S_K$ is a feasible set of items. Let $\hat{i}_1, \hat{i}_2, ..., \hat{i}_D$ be the set of items in $A_{OPT} \cup S_K$ listed in increasing order of index, and let $\hat{C}_j$ be the set formed by the first j items of $A_{OPT} \cup S_K$ for all $j = 0, 1, ..., D$. We show that in the entry of the profit table corresponding to item $\hat{i}_j$ our algorithm must have included a tuple $\tau^* = (w_r^*, w_t^*, p^*, time_r^*, prev^*)$ that corresponds to a feasible subset such that $w_t^* \leq weight(\hat{C}_j)$, $w_r^* \leq weight'(\hat{C}_j)$, $p^* \geq profit'(\hat{C}_j)$, and $time_r^*$ is at most $(1+\epsilon)time_j + \epsilon T\frac{m_j}{n}$, where m_j is the number of vertices in the path selected by the optimum solution from s to the vertex u storing the tuple τ and $time_j$ is the time needed by the optimum solution to transport the items in $\hat{C}_j$ to u. To show that this tuple exists, we use a proof by induction on the number of items in $\hat{C}_j$.

The base case, when $j = 0$, so $|\hat{C}_0| = 0$, is trivial as our algorithm adds to the profit table the entry $(0, 0, 0, 0, -1)$. Assume then that the claim is true for $\hat{C}_j$ with $j < D$. In $\hat{C}_{j+1}$, $\hat{i}_{j+1}$ is the item with largest index. By the induction hypothesis, there is an entry of the profit table corresponding to $\hat{i}_j$ storing a tuple $\tau^* = (w_r^*, w_t^*, p^*, time_r^*, prev^*)$ as above.

Let u be the vertex storing $\hat{i}_j$, v be the vertex storing $\hat{i}_{j+1}$, and let $d_{u,v}$ be the distance between u and v in the path P selected by OPT from s to v. Our algorithm considers transporting the items corresponding to the tuple τ^* from u to v and that would add a tuple $(w_r'', w_t'', p'', time_r'', prev'')$ to the entry $A[\pi(\hat{i}_{j+1})]$ corresponding to item $\hat{i}_{j+1}$, where by the induction hypothesis $w_t'' = w_t^* \leq weight(\hat{C}_j)$. Since our algorithm removes dominated tuples from the profit table, then entry $A[\pi(\hat{i}_{j+1})]$ must store a tuple $\tau^\sim = (w_r^\sim, w_t^\sim, p^\sim, time_r^\sim, prev^\sim)$ such that $w_r^\sim \leq w_r''$, $w_t^\sim \leq w_t''$, $p^\sim \geq p''$, and $time_r^\sim \leq time_r''$. Since $w_t^\sim \leq w_t'' \leq weight(\hat{C}_j)$, then the time $time_{uv}^\sim$ that the thief needs to carry weight w_t'' from u to v following the path selected by OPT is at most the time $time_{uv}$ needed in the optimum solution to carry the items in $\hat{C}_j$ from vertex u to vertex v.

In line 14 of Algorithm 5 we round the traveling times, and since a path between u and v has $m_{uv} \leq n$ vertices, then the number of times that we round traveling times when computing the tuples needed to produce $\tau^\sim$ from τ^* is $m_j + m_{uv} \leq n$ and so $time_r^\sim \leq time_r^* + time_{uv} + \epsilon T\frac{m_{uv}}{n}$. Hence, by the induction hypothesis, $time_r^\sim \leq (1+\epsilon)time_j + time_{uv} + \epsilon T\frac{m_j + m_{uv}}{n} \leq$

$(1+\epsilon)\mathit{time}_{j+1} + \epsilon T\frac{m_j+m_{uv}}{n}$, where $\mathit{time}_{j+1} = \mathit{time}_j + \mathit{time}_{uv}$ is the time needed by the optimum solution to transport $\hat{C}_{j+1}$ from s to v.

After adding item $\hat{i}_{j+1}$ to the tuple $\tau^{\sim}$ and removing duplicated tuples, the profit table must include a tuple $(w_r^{\#}, w_t^{\#}, p^{\#}, \mathit{time}_r^{\#}, \mathit{prev}^{\#})$ such that $w_r^{\#} \leq w_r^{+} + \mathit{weight}'(\hat{i}_{i+j}) \leq \mathit{weight}'(\hat{C}_{j+1})$, $p^{\#} \geq \mathit{profit}'(\hat{C}_{j+1})$, $\mathit{time}_r^{\#} \leq (1+\epsilon)\mathit{time}_{j+1} + \epsilon T\frac{m_j+m_{uv}}{n}$, and $w_t^{\#} \leq \mathit{weight}(\hat{C}_{j+1})$, as for tuples with the same rounded weight, profit, and travel times the algorithm selects that with smallest true weight.

Hence, our algorithm will compute a tuple $(w_r, w_t, p, \mathit{time}_r, \mathit{prev})$, where $w_r \leq \mathit{weight}'(A_{OPT} \cup S_K)$, $w_t \leq \mathit{weight}(A_{OPT} \cup S_K)$, $p \geq \mathit{profit}'(A_{OPT} \cup S_K)$, and $\mathit{time}_r \leq (1+\epsilon)\mathit{time}_D + \epsilon T\frac{n}{n} \leq (1+2\epsilon)T$, where $\mathit{time}_D \leq T$ is the time needed by the optimum solution to transport the items in $A_{OPT} \cup S_K$ from s to t.

Since

$$\mathit{SOL}^* \geq p^* \geq \mathit{profit}'(A_{OPT} \cup S_K) \tag{1}$$

we need to bound $\mathit{profit}'(A_{OPT} \cup S_K)$.

Let $S_{OPT}^{-} = S_{OPT} - S_K - A_{OPT}$, this set includes the items in the optimum solution whose profit is not included in the right hand side of (1). Note that if S_{OPT}^{-} is empty, then $S_K \cup A_{OPT} = S_{OPT}$ and so $\mathit{SOL}^* \geq \mathit{profit}'(S_{OPT})$. If S_{OPT}^{-} is not empty, we show that $\mathit{weight}'(S_{OPT}^{-}) \leq \frac{\epsilon W'}{|I|} + w_L$, where w_L is the largest weight of the items in S_{OPT}^{-}. To see this, recall that $\mathit{weight}'(S_{OPT}) \leq W + \frac{\epsilon W'}{|I|}$ and note that $\mathit{weight}(S_K) + \mathit{weight}'(A_{OPT}) \leq W$, but by the way in which A_{OPT} was defined the empty space that $S_K \cup A_{OPT}$ leave in the knapsack is not large enough to fit the rounded weight of another item from S_{OPT}^{-}. Therefore,

$$\begin{aligned}
\mathit{weight}'(S_{OPT}^{-}) &= \mathit{weight}'(S_{OPT}) - (\mathit{weight}(S_K) + \mathit{weight}'(A_{OPT})) \\
&\leq W + \frac{\epsilon W'}{|I|} - (W - w_L) \\
&= \frac{\epsilon W'}{|I|} + w_L
\end{aligned} \tag{2}$$

Now we bound $\mathit{profit}'(S_{OPT}^{-})$. If S_{OPT}^{-} consists of only one item i_j, then since the least profitable item in S_K has profit at most $\frac{1}{K}OPT$ and i_j is not in S_K, then $p_j \leq \frac{1}{K}OPT$, which means that $\mathit{profit}'(S_{OPT}^{-}) \leq \frac{1}{K}OPT =$

ϵOPT. If S^-_{OPT} consists of two or more items, we can partition S^-_{OPT} into the singleton $\{i_L\}$ consisting of the item with largest weight w_L and the set i_* of remaining items, which by (2) has $\mathit{weight}'(i_*) \leq \frac{\epsilon W'}{|I|}$.

Item i_L is not in S_K, so it has profit $p_L \leq \frac{1}{K} OPT$. As for i_* we show that $\mathit{profit}'(i_*) \leq \frac{1}{K} OPT$:

- $W' - \mathit{weight}'(A_{OPT}) < \frac{\epsilon W'}{|I|} = \frac{W'}{K|I|}$, as otherwise the items i_* would have been included in A_{OPT}, and so

$$\mathit{weight}'(A_{OPT}) = \sum_{i_j \in A_{OPT}} w'_j > W' - \frac{W'}{K|I|} > \frac{K-1}{K} W', \text{ as } |I| \geq 1 \quad (3)$$

- There is at least one item i_ψ in A_{OPT} with $w'_\psi \geq \frac{W'}{K|I|}$. To see this, note that if all of the items in A_{OPT} had weight strictly less than $\frac{W'}{K|I|}$, then $\sum_{i_j \in A_{OPT}} w'_j < \frac{W'}{K|I|} |A_{OPT}| < \frac{1}{K} W'$, as $|A_{OPT}| \leq |I|$, which contradicts (3).
- By definition, A_{OPT} includes items from $S_{OPT} - S_K$ with $\mathit{weight}'(A_{OPT}) \leq W'$ and maximum profit, and since the items i_* are not in A_{OPT} then $\mathit{profit}'(i_*) \leq p'_\psi$, as otherwise the items i_* would be in A_{OPT} instead of i_ψ. Since item i_ψ is not in S_K then $p'_\psi \leq \frac{1}{K} OPT$ and so $\mathit{profit}'(i_*) \leq \frac{1}{K} OPT$.

Therefore, $\mathit{profit}'(S^-_{OPT}) = \mathit{profit}'(i_L) + \mathit{profit}'(i_*) \leq \frac{2}{K} OPT$. Since $S_{OPT} = S_K \cup A_{OPT} \cup S^-_{OPT}$ then $\mathit{profit}'(A_{OPT}) + \mathit{profit}'(S_K) = \mathit{profit}'(S_{OPT}) - \mathit{profit}'(S^-_{OPT}) \geq \mathit{profit}'(S_{OPT}) - \frac{2}{K} OPT$. By (1),

$$\begin{aligned} SOL^* &\geq \mathit{profit}'(A_{OPT}) + \mathit{profit}'(S_K) \\ &\geq \mathit{profit}'(S_{OPT}) - \frac{2}{K} OPT \\ &\geq \mathit{profit}(S_{OPT}) - \frac{\epsilon P_{max}}{|I|} |S_{OPT}| - \frac{2}{K} OPT \\ &\geq OPT - \epsilon P_{max} - 2\epsilon OPT \geq (1 - 3\epsilon) OPT \end{aligned}$$

Since $SOL \geq SOL^*$, then $SOL \geq (1 - 3\epsilon) OPT$. □

Theorem 3. *There is a PTAS for ThOP on DAGs that produces solutions with travel time at most $T(1+\epsilon)$ for any constant $\epsilon > 0$.*

Proof. As shown by Lemma 2, Algorithm 4 computes solutions with profit at least $(1-3\epsilon)OPT$. The profit table A has $|I|$ entries, and each entry $A[j]$ can have at most $O(P_r W_r T_r)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items, W_r is the number of different values for the rounded weight of any subset of items of total weight at most W, and T_r is the maximum number of rounded travel times. Since profits are rounded down to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$ then P_r is $O(\frac{|I|^2}{\epsilon})$. Since weights of items not in the selected feasible subsets S are rounded up to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$, then W_r is $O(\frac{|I|^2}{\epsilon})$; T_r is $O(\frac{n}{\epsilon})$.

Algorithm 4 iterates through the list at each entry $A[j]$ exactly once if i_j is not the last item in a vertex u; if i_j is the last item in a vertex u, then our algorithm iterates through the list at entry $A[j]$ once for each outgoing edge of u. Thus, for each feasible subset S with at most K items the algorithm loops through $|I|$ rows in the profit table, iterates over a particular row at most $n = |V|$ times, and each row can have at most $O(\frac{|I|^4 n}{\epsilon^3})$ tuples in it; since the number of feasible subsets S is $O(|I|^{\frac{1}{\epsilon}})$ the running time is $O(\frac{n^2}{\epsilon^3}|I|^{5+\frac{1}{\epsilon}})$. □

Corollary 2. *There is a PTAS for ThOP on DAGs when* $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ *that produces solutions with travel time at most* T.

Proof. Without loss of generality we can assume that $\mathcal{V}_{\min} = \mathcal{V}_{\max} = 1$, so the travel time between vertices u and v is $d_{u,v}$. Therefore, in Algorithms 4 and 5 we do not need to keep rounded travel times or real item weights in the tuples. Hence, a tuple $(w_r, p, time, prev)$ dominates a tuple $(w'_r, p', time', prev')$ if $p \geq p'$, $w_r \leq w'_r$, and $time \leq time'$. Since the algorithm now uses real travel times, the solution produced by the algorithm uses total travel time at most T. □

5. Thief Orienteering on Undirected Graphs

In this section we consider the case when the input graph $G = (V, E)$ is undirected and has integer edge lengths of at least 1 and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$. Notice that if $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ the travel time for any edge is equal to the length of the edge and it is independent of the weight of the items in the knapsack. First, we consider when T is equal to the length L of a shortest path from s to t.

Theorem 4. *There is a FPTAS for ThOP where* $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ *and the time* T *is equal to the length* L *of a shortest path from* s *to* t.

Proof. Delete from G every edge (u, v) that does not belong to a shortest path from s to t. If the input graph G is undirected, direct the remaining edges towards the endpoint closer to t. Note that G is now a DAG. We modify Algorithm 1 so that it rounds down the profit of each item to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$, and a tuple with profit p dominates a tuple with profit p' if $p \geq p'$ and $w \leq w'$. Observe that since the paths from s to t are shortest paths, the travel times for both of these tuples must be the same, and hence the weights and travel times of the items do not need to be rounded up.

Furthermore, this modified Algorithm 1 does not need to consider all possible feasible subsets S of at most K items because in Section 4 we needed to consider all these subsets to bound the maximum profit loss due to the rounding of the weights. Hence, this modified Algorithm 1 is a FPTAS. □

In the rest of this section we show that Theorem 4 can be extended to the case when $T = L + K$, where $K > 0$ is a integer constant. In the sequel we assume that the input graph is undirected. It is not hard to see how our ideas can be extended to directed graphs. Note that when $T = L + K$ we cannot simply transform G into a DAG using the same process as described in the proof of Theorem 4, as in this version of the problem a feasible solution might include edges that do not belong to a shortest path between s and t.

Let $u.dist$ be the shortest distance from vertex u to t, let σ be a path from s to t with length at most $L + K$, and let the vertices in σ be $s = u_1, u_2, u_3, ..., u_{|\sigma|} = t$. If σ includes an edge (u_i, u_{i+1}) that does not belong to any shortest path from u_i to t or, more formally, if $u_i.dist < u_{i+1}.dist + d_{u_i,u_{i+1}}$, then we say that (u_i, u_{i+1}) is part of a *detour*.

Definition 2. *A simple path σ from s to t decomposes into a unique list $D = D_1, D_2, ..., D_{n_D}$ of vertex disjoint detours, where D might be empty. Each detour D_i forms a longest subpath of σ where no edge (u, v) of D_i belongs to a shortest path from u to t, for all $i = 1, 2, ..., n_D$. All edges (u, v) in σ that do not belong to a shortest path from u to t belong to exactly one detour; hence, the list D is unique.*

We show a few examples of detours in Figure 2; note that we have omitted the items information. Next, we show that the number of edges that belong to detours is at most the constant K. This fact will help us to reduce the running time of our algorithm, as it enumerates all possible sets of detours.

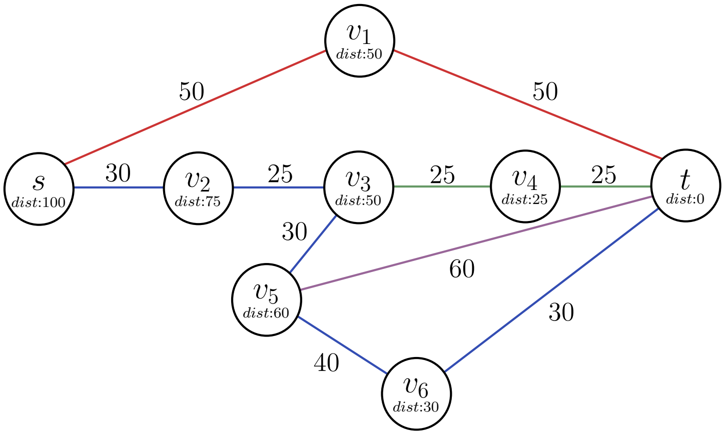


Figure 2: Let σ be the path from s to t that includes the blue edges. The red edges belong to a shortest path (of length $s.dist = 100$) between s and t, the green edges belong to a shortest path (of length $v_3.dist = 50$) between v_3 and t, and the purple edge belongs to a shortest path (of length $v_5.dist = 60$) between v_5 and t. The path σ decomposes into the unique list $D = D_1, D_2$ of detours where $D_1 = \{s, v_2\}$ and $D_2 = \{v_3, v_5, v_6\}$.

Lemma 3. *A simple path σ from s to t of length at most $L + K$ has at most K edges (u, v) such that (u, v) is in a detour.*

Proof. We use a proof by contradiction. Assume that path σ from s to t of length at most $L + K$ includes edges $(u_1, v_1), (u_2, v_2), ..., (u_a, v_a)$ with $a > K$ that do not belong to any shortest path from u_i to t. Let $\ell(u, v)$ be the length of the subpath of σ from u to v and let $u_{a+1} = t$. Then $d_{u_i,v_i} + \ell(v_i, u_{i+1}) + u_{i+1}.dist > u_i.dist$, for all $i = 1, 2, ..., a$. Since $\ell(u_i, u_{i+1}) = d_{u_i,v_i} + \ell(v_i, u_{i+1})$ then $\ell(u_i, u_{i+1}) \geq u_i.dist - u_{i+1}.dist + 1$ as all edges have length at least 1.

Therefore, $\ell(s, t) = \ell(s, u_1) + \sum_{i=1}^{a} \ell(u_i, u_{i+1}) \geq \ell(s, u_1) + \sum_{i=1}^{a}(u_i.dist - u_{i+1}.dist + 1) = \ell(s, u_1) + u_1.dist + a \geq L + a$. Since the length $\ell(s, t)$ of σ is at most $L + K$ but $a > K$ then we have reached a contradiction. $\square$

Corollary 3. *A simple path σ from s to t of length at most $L+K$ decomposes into a list D containing at most K detours.*

Note that by the definition of a detour, for a simple path σ from s to t the edges (u, v) that do not belong to any detours must belong to a shortest path from u to t. We show next how to decompose σ into alternating subpaths of detours and subpaths that are shortest paths between their first and last vertices.

Corollary 4. *A simple path σ from s to t of length at most $L+K$ decomposes into subpaths $\sigma = P_1, D_1, P_2, D_2, ..., P_r, D_r, P_{r+1}$ where $r \leq K$, each D_i is a detour, and each P_i is a shortest path between its first and last vertices.*

Proof. Let $P_i = u_{i_1}, u_{i_2}, ..., u_{i_e}$. Each edge $(u_{i_j}, u_{i_{j+1}})$ is in a shortest path from u_i to t as otherwise the edge would belong to a detour. Hence, $u_{i_j}.dist = d_{u_{i_j},u_{i_{j+1}}} + u_{i_{j+1}}.dist$ for each $i = 1, ..., e-1$. Therefore, the length of P_i is $\sum_{j=1}^{e-1} d_{u_{i_j},u_{i_{j+1}}} = \sum_{j=1}^{e-1}(u_{i_j}.dist - u_{i_{j+1}}.dist) = u_{i_1}.dist - u_{i_e}.dist$ and thus P_i is a shortest path between u_{i_1} and u_{i_e}.

The remaining subpaths in σ are the detours $D_1, D_2, ..., D_{n_D}$ and by Corollary 3, $n_D \leq K$. □

In the sequel we say that a path σ from s to t *decomposes* into a list D of detours.

5.1. Building the Lists of Detours

We define the *profit of a path* as the maximum profit achievable by collecting items of total weight at most W along that path.

Let σ^* be a simple path from s to t with maximum profit of length at most $L+K$. If σ^* is a shortest path from s to t, then Theorem 4 provides an FPTAS for it. Hence, in the sequel we assume that σ^* is not a shortest path from s to t, and so it must have at least one detour. The path σ^* decomposes into a unique list $D^* = D_1^*, D_2^*, ..., D_{n_{D^*}}^*$ of n_{D^*} detours, where $n_D^* \leq K$ (by Corollary 3), D has at most K edges (by Lemma 3), and the subpaths of σ^* that do not belong to any detours are shortest paths between their first and last vertices (by Corollary 4).

Since we do not know how many detours are in D^*, or which vertices are in each of them, our algorithm to compute a path σ from s to t of length at most $L+K$ and near maximum profit will generate the set $\mathcal{D}$ of all possible *valid* lists of detours as shown in Algorithm 6.

Definition 3. *A list D of detours $D_1, D_2, ..., D_{n_D}$ is valid if there exists a simple path σ from s to t of length at most $L+K$ that decomposes into D.*

Note that D^* must be included within the set $\mathcal{D}$ computed by Algorithm 6. For each list $D \in \mathcal{D}$ we will create a DAG G_D containing all the paths from s to t of length at most $L+K$ that go through the detours in D. Algorithm 4 will then be used on these DAGs, and the highest profit path found will be output.

Algorithm 6 GenerateDetours($G = (V, E)$)

```
**Input:** Graph $G$.
**Output:** The set $\mathcal{D}$ of all valid detours for paths from $s$ to $t$ in $G$.
Let $\mathcal{D} = \emptyset$.
for each subset $S_V$ of $V$ such that $2 \leq |S_V| \leq 2K$ do
    for $n_D = 1$ to $K$ do
        for each possible partitioning of $S_V$ into a list $D$ of $n_D$ detours do
            Compute a shortest path $\sigma$ from $s$ to $t$ in $G$ that decomposes into $D$.
            if $\sigma$ has length at most $L + K$ then
                $\mathcal{D} = \mathcal{D} \cup D$.
            end if
        end for
    end for
end for
Output $\mathcal{D}$.
```

5.2. Transforming the Problem into a Shortest Paths Problem

We transform the problem of finding in a given graph G a simple path with maximum profit from s to t of length at most $L+K$ into the problem of finding in a collection of directed graphs shortest paths with maximum profit from s to t. We create a graph G_D from G with the following properties: (i) for each simple path σ in G from s to t of length at most $L+K$ that decomposes into a valid list D of detours there is an equivalent shortest path in G_D from s to t, and (ii) for every shortest path from s to t in G_D there is an equivalent simple path σ in G of length at most $L+K$ that decomposes into a valid list D of detours.

Definition 4. *An undirected path σ from s to t in G and a directed path σ_D from s to t in G_D are equivalent if they include the same vertices and edges.*

To ensure that all paths in G from s to t that travel through D are shortest paths in G_D we will carefully reduce the lengths of the edges in D (allowing some of them to have negative lengths) so that a shortest path in G_D must travel through D. Algorithm 7 describes how to construct G_D. For any detour D_i in D, an *internal* vertex of D_i is any vertex that is not the first or last vertex of D_i.

Algorithm 7 DetourToDAG($G = (V, E), D, s, t, I$)

```
Input: Graph $G$, valid list of detours $D$, start vertex $s$, end vertex $t$, and item assignments $I$.
Output: A DAG $G_D$ containing paths from $s$ to $t$ equivalent to the paths from $s$ to $t$ in $G$ that travel through all of the detours in $D$.
Let $G_D$ be a copy of $G$, with the same item assignments $I$.
For each vertex $u \in G_D$ compute $u.dist$.
for each internal vertex $u_i \in D_i$, for all $D_i \in D$ do
    Delete from $G_D$ all edges incident on $u_i$ except for $(u_{i-1}, u_i)$
    and $(u_i, u_{i+1})$, where $u_{i-1}$ is the vertex preceding $u_i$ in $D_i$
    and $u_i$ is the vertex preceding $u_{i+1}$ in $D_i$.
end for
Let $\sigma$ be a simple path from $s$ to $t$ that decomposes into $D$.
for each $D_i \in D$ do
    Direct the edges in $D_i$ from its first to its last vertex according to the order in which the vertices appear in $\sigma$.
end for
Index the directed edges $E_D = e_1, e_2, ..., e_{m_D}$ of the detours in $D$ in the order they appear in $\sigma$.
for directed edge $e_i$ with endpoints $(u_i, v_i)$, for all $i = 1, 2, ..., m_D$ do
    $d_{u_i,v_i} = u_i.dist - v_i.dist - 1$.
end for
Delete any edges $(u, v)$ in $G_D$ that do not belong to any detours of $D$ such that $u.dist = v.dist$.
Direct any remaining undirected edges in $G_D$ towards the endpoint closer to $t$.
Output the DAG $G_D$.
```

Corollary 5. *Let D be a valid list of detours. For each simple path σ in G from s to t that decomposes into D, the graph G_D output by Algorithm 7 contains all the edges of σ and hence σ is a path in G_D from s to t.*

Proof. Recall that a path σ in G from s to t that decomposes into D has the form $P_1, D_1, P_2, D_2, ..., P_r, D_r, P_{r+1}$, where each D_i is a detour in D and each P_j is a shortest path between its first and last vertices for all $i = 1, 2, ..., r$ and $j = 1, 2, ..., r + 1$. Since initially G_D was a copy of G and the only edges that are deleted from G_D are edges not in detours incident on internal vertices of detours or edges (u, v) not in D for which $u.dist = v.dist$, then all the edges of σ are also directed in G_D. $\square$

Let σ be a simple path from s to t in G of length at most $L + K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. By Corollary 5, σ is a path in G_D from s to t. To avoid confusion, let σ_D denote the directed version of path σ when referring to the graph G_D. For any two vertices u and v of σ, let $\ell(u, v)$ be the length of the subpath in σ between u and v and let $\ell_D(u, v)$ be the length of the subpath in σ_D between u and v. Let $d'_{u,v}$ be the length of edge (u, v) in G_D. For a vertex u, let $u.dist$ be the length of a shortest path in G from u to t; note that Algorithm 7 does not change the value $u.dist$, the algorithm only modifies the lengths of edges in G_D and not those in G.

Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. We will prove in the next lemma that for edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist - v_z.dist - z$, for all $z = 1, 2, ..., m_D$.

Lemma 4. *Let σ be a simple path from s to t in G of length at most $L + K$ that decomposes into the non-empty valid list D of detours, let G_D be the graph output by Algorithm 7, and let σ_D denote the directed version of path σ when referring to the graph G_D. Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. For edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist - v_z.dist - z$, for all $z = 1, 2, ..., m_D$.*

Proof. We use a proof by induction on the edges in E_D. For the base case we consider the first edge e_1 with endpoints (u_1, v_1). In line 14 of Algorithm 7 edge (u_1, v_1) is assigned length $d'_{u_1,v_1} = u_1.dist - v_1.dist - 1$ (note that in G_D we allow negative lengths but only for the edges in the detours). By Corollary 4, $\ell(s, u_1) = s.dist - u_1.dist$ (as the subpath of σ from s to u_1 is a shortest path between s and u_1) and since all edges in the subpath of σ_D from s to u_1 have the same lengths as the corresponding edges in σ, then $\ell_D(s, u_1) = s.dist - u_1.dist$, and since $\ell_D(s, v_1) = \ell_D(s, u_1) + d'_{u_1,v_1}$, then $\ell_D(s, v_1) = s.dist - u_1.dist + u_1.dist - v_1.dist - 1 = s.dist - v_1.dist - 1$.

Assume now that the claim holds true for every edge e_q in E_D with endpoints (u_q, v_q), for all $q = 1, 2, ..., j - 1$, so that $\ell_D(s, v_q) = s.dist - v_q.dist - q$. We show for the edge e_j in E_D with endpoints (u_j, v_j) that $\ell_D(s, v_j) = s.dist - v_j.dist - j$.

Edge e_j with endpoints (u_j, v_j) is assigned length $d'_{u_j,v_j} = u_j.dist - v_j.dist - 1$ in line 14 of Algorithm 7. Let e_j be part of detour D_i. We need to consider two cases:

- Case 1: Edge e_j is the first edge of D_i. Since $\ell_D(s, v_j) = \ell_D(s, v_{j-1}) + \ell_D(v_{j-1}, u_j) + d'_{u_j,v_j}$, by the induction hypothesis $\ell_D(s, v_{j-1}) = s.dist - v_{j-1}.dist-(j-1)$. By Corollary 4 $\ell(v_{j-1}, u_j) = v_{j-1}.dist-u_j.dist$ (as the subpath of σ from v_{j-1} to u_j is a shortest path between these vertices) and all edges in the subpath of σ_D from v_{j-1} to u_j have the same lengths as the corresponding edges in σ, then $\ell_D(v_{j-1}, u_j) = v_{j-1}.dist - u_j.dist$. Since $d'_{u_j,v_j} = u_j.dist - v_j.dist - 1$, then $\ell_D(s, v_j) = s.dist - v_{j-1}.dist - (j-1) + v_{j-1}.dist - u_j.dist + u_j.dist - v_j.dist - 1 = s.dist - v_j.dist - j$.
- Case 2: Edge e_j is not the first edge of D_i. Since $\ell_D(s, v_j) = \ell_D(s, u_j) + d'_{u_j,v_j}$, and by the induction hypothesis $\ell_D(s, u_j) = s.dist - u_j.dist - (j-1)$, then $\ell_D(s, v_j) = s.dist - u_j.dist - (j-1) + u_j.dist - v_j.dist - 1 = s.dist - v_j.dist - j$.

□

Corollary 6. *Let σ be a simple path from s to t in G of length at most $L + K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. The path σ_D in G_D has length $L - m_D$, where m_D is the number of edges in D.*

Proof. Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. By Lemma 4 for edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist-v_z.dist-z$, for all $z = 1, 2, ..., m_D$. Then, since $\ell_D(s, t) = \ell_D(s, v_{m_D}) + \ell_D(v_{m_D}, t)$, where $e_{m_D} = (u_{m_D}, v_{m_D})$ and by Corollary 4 $\ell(v_{m_D}, t) = v_{m_D}.dist$ (as the subpath of σ from v_{m_D} to t contains no edges that belong to a detour) and since the edges in the subpath of σ_D from v_{m_D} to t have the same lengths as the corresponding edges in σ, then $\ell_D(v_{m_D}, t) = v_{m_D}.dist$; therefore, σ_D has length $\ell_D(s, t) = \ell_D(s, v_{m_D}) + \ell_D(v_{m_D}, t) = s.dist - v_{m_D}.dist - m_D + v_{m_D}.dist = s.dist - m_D = L - m_D$. □

By Lemma 4 and Corollary 6, we can see that the length of a shortest path in G_D from s to t is less than the length of a shortest path in G from s to t. This is important, because we do not want the shortest paths in G from s to t which do not decompose into D to also be shortest paths in G_D from s to t. We show that all the shortest paths in G_D from s to t must decompose into D as this is critical to our algorithm.

Lemma 5. *Let σ be a simple path from s to t in G of length at most $L+K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. The path σ_D is a shortest path in G_D from s to t and every shortest path in G_D from s to t decomposes into D.*

Proof. By Corollary 6, the path σ_D in G_D from s to t that corresponds to the path σ has length $L - m_D$, where L is the length of a shortest path in G and m_D is the number of edges in D.

Assume, for the sake of contradiction, that there exists a path σ'_D in G_D from s to t with length smaller than $L - m_D$. Since there are only m_D edges that belong to detours, and the edges in σ'_D that belong to detours are the only edges whose lengths were changed with respect to the lengths of the edges in the corresponding path σ' in G, then it must be the case that for some detour edge (u, v) the algorithm set its length to a value smaller than $u.dist - v.dist - 1$. However, this is not possible because line 14 of Algorithm 7 sets the length of each detour edge (u, v) to $u.dist - v.dist - 1$, and so σ_D is a shortest path from s to t in G_D.

Additionally, assume for the sake of contradiction, that there exists a shortest path σ'_D in G_D from s to t that does not include all the edges in D. Since only m_D edges belong to detours, and the edges in G_D that belong to detours are the only edges whose lengths were changed with respect to the lengths of the corresponding edges in G, then it must be the case that for some detour edge (u, v) the algorithm set its length to a value smaller than $u.dist - v.dist - 1$. However, this is not possible because line 14 of Algorithm 7 sets the length of each detour edge (u, v) to $u.dist - v.dist - 1$, and so every shortest path in G_D from s to t decomposes into D. $\square$

Corollary 7. *Let D be a valid list of detours in G. For the graph G_D output by Algorithm 7 corresponding to D (i) for every path σ' from s to t in G of length $\Delta \leq L + K$ that decomposes into D, G_D has an equivalent shortest path σ'_D from s to t that decomposes into D and (ii) for every shortest path σ'_D from s to t in G_D there is an equivalent path σ' from s to t in G of length $\Delta \leq L + K$ that decomposes into D.*

Proof. (i) Let σ' be a path from s to t in G of length $\Delta \leq L + K$ that decomposes into D. By Corollary 5 σ' is also a path of G_D from s to t that decomposes into D, and by Lemma 5 σ' is a shortest path in G_D from s to t.

Since G_D has the same vertices, edges, and item assignments as G then the two paths σ' in G and G_D are equivalent.

(ii) By Lemma 5, σ'_D decomposes into D. Since Algorithm 7 initially creates G_D by copying G and it does not add any extra edges to G_D, then σ'_D is also a path in G from s to t that decomposes into D. Since D is a valid list of detours, then the length of the path σ'_D in G is at most $L + K$. □

Algorithm 8 ThOPLengthLPlusK($G = (V, E), W, T, K, s, t, I, \epsilon$)

```
Input: Graph $G$, knapsack capacity $W$, time limit $T \leq L + K$, constant $K$, start vertex $s$, end vertex $t$, item assignments $I$, and constant $\epsilon > 0$, where $L$ is the length of a shortest path from $s$ to $t$.
Output: A path with length at most $T$ from $s$ to $t$ with profit at least $(1 - \epsilon)$ times the maximum possible profit.
Let $bestPath$ be an empty path.
Let $\mathcal{D}$ be the set of all possible valid lists of detours as described in Section 5.1.
for each $D \in \mathcal{D}$ do
    $G_D = DetourToDAG(G, D, s, t, I)$.
    Let $D$ have $m_D$ edges and let $T' = L - m_D$.
    $path = ThOPDAGPTAS^*(G_D, W, T', s, t, I, 1, \epsilon)$.
    if profit of $path >$ profit of $bestPath$ then
        $bestPath = path$.
    end if
end for
Return $bestPath$.
```

Our algorithm for ThOP where $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and the time T is equal to the length L of a shortest path from s to t plus a constant K is described in Algorithm 8. Since some edges in G_D might have negative lengths algorithm *ThOPDAGPTAS** uses a slightly modified *UpdateProfitTable* algorithm that discards the condition $time' + travel \leq T$ on line 13 of Algorithm 2. Therefore, the profit tables produced by Algorithm *ThOPDAGPTAS** might contain infeasible solutions whose total travel time is greater than T, but this is not a problem since we select the best solution that does not exceed the time limit T.

Theorem 5. *There is a FPTAS for ThOP where* $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ *and the time* T *is equal to the length* L *of a shortest path from* s *to* t *plus a constant* K.

Proof. Let σ^* be a path from s to t with length $\Delta \leq L + K$ with maximum profit, and let D^* be the list of detours that σ^* decomposes into. The list D^* is valid; therefore, since Algorithm 8 considers all possible valid lists of detours, then Algorithm 8 considers the list of detours D^* in one of its iterations.

By Corollary 4 and Corollary 7 the problem of finding a path in G from s to t that decomposes into the list D of detours, has length at most $L+K$, and has maximum profit is equivalent to finding a shortest path in G_D from s to t with maximum profit. For every valid list of detours, and in particular, for the list D^*, Algorithm 8 finds a path in G from s to t that decomposes into D^*, has length at most $L+K$, and has profit at least $(1-\epsilon)OPT$, where OPT is the value of an optimum solution.

Note that when the profit tables produced by Algorithm *ThOPDAGPTAS** contain multiple tuples with the same profit, only the tuple with the smallest weight is kept in the profit table, as $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and paths from s to t are shortest paths and so the travel time times for these tuples must be the same, and hence the weights and travel times of the items do not need to be rounded up; therefore, each profit table produced by Algorithm *ThOPDAGPTAS** has at most $O(P_r)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items. Moreover, since the item weights are not rounded then Algorithm *ThOPDAGPTAS** does not need to iterate over all possible subsets using at most K items. Hence, Algorithm *ThOPDAGPTAS** has time complexity $O(n|I|^2)$.

Furthermore, the number of possible lists of detours is polynomial with respect to the constant K and verifying whether a list of detours is valid requires computing a constant number of shortest paths, so Algorithm 8 is a FPTAS. □

6. Thief Orienteering on Cliques

In this section we study a version of ThOP where the graph is a clique, every edge has length 1, and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ so the travel time of an edge does not depend on the current load in the backpack. A clique is a graph where every pair of vertices is connected by an edge. Observe that solving ThOP on a clique is equivalent to finding the $T-1$ vertices that a path σ from s to t in G should include for the thief to collect items of maximum value and total weight at most W.

Note that since all edges have length 1 and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, the order in which a path σ traverses the vertices does not matter when computing the best path. Therefore, we index the vertices in V from 1 to $|V|$ assigning to s index 1 and t index $|V|$.

6.1. Clique Profit Table

Definition 5. *Let $S_z = \{i_1, ..., i_z\}$ for all $z = 1, ..., |I|$, and let vertex u hold item i_z. A subset S of S_z is a feasible subset (for ThOP on cliques) if it has weight $w_S = \sum w_j \leq W$ and the items in S belong to a most $T - 1$ different vertices.*

Our algorithm builds a *clique profit table* where each entry $A[j]$, for $j = 1, ..., |I|$, corresponds to the item i_j with index j, and $A[j]$ is a list of tuples (w, p, ν). A tuple (w, p, ν) in the list of $A[j]$ indicates that there is a subset S of S_j such that:

- the items in S are stored in the vertices in ν, where $|\nu| \leq i$,
- the weight of the items in S is $w \leq W$, and
- the profit of the items in S is p.

A tuple (w, p, ν) *dominates* a tuple (w', p', ν') if $p \geq p'$, $w \leq w'$, and $|\nu| \leq |\nu'|$. We remove dominated tuples from each list of A such that no tuple in the list of each entry $A[j]$ dominates another tuple in the same list. Therefore, we can assume each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their profits, (ii) there might be multiple tuples with the same profit and these tuples are sorted in non-decreasing order of their weights, and (iii) if several tuples in $A[j]$ have the same profit p and weight w, only the tuple with the smallest value of $|\nu|$ is kept in $A[j]$.

6.2. Computing the Clique Profit Table

Algorithm 9 shows how we update the clique profit table with the items stored in vertex u. The start vertex s holds a single item i_1 of weight and profit 0; therefore, we initialize $A[1]$ to store the tuple $(0, 0, \emptyset)$.

Algorithm 9 UpdateCliqueProfitTables(W, T, A, u)

```
Input: Knapsack capacity $W$, time $T$, clique profit table $A$, and vertex $u$.
Output: The entries of the clique profit table are updated to represent the subsets of items taken from at most $T-1$ vertices along a path from $s$ to $t$ that routes through $u$.
for each item $i_j$ of $u$ do
    Let $w$ be the weight and $p$ the profit of $i_j$.
    Copy all tuples of $A[j-1]$ into $A[j]$.
    for each tuple $(w', p', \nu') \in A[j]$ do
        if $(u \in \nu')$ OR $(u \notin \nu'$ AND $|\nu'| < T-1)$ then
            if $w + w' \leq W$ then
                Append $(w + w', p + p', \nu' \cup u)$ to $A[j]$.
            end if
        end if
    end for
    Remove dominated tuples from $A[j]$.
end for
```

6.3. Algorithm Analysis

Our algorithm for ThOP on cliques is shown below. The algorithm computes a path σ from s to t and a set S of items such that the sum of profits of the items is maximum, the sum of weights of the items is at most W, σ contains at most $T+1$ vertices (including s and t), and the items from S are located at the vertices in σ. Algorithm *BuildKnapsack* recovers the set of items from the selected path with maximum profit by using a similar approach as in Algorithm 3.

A similar inductive proof as to the proof of Lemma 1 shows that for each feasible subset S of S_z the clique profit table contains a tuple (w_S, p_S, ν_S) (or a dominating tuple (w'_S, p'_S, ν'_S)) at entry $A[z]$, where $w_S = \sum_{i_j \in S} w_j$, $p_S = \sum_{i_j \in S} p_j$, $|\nu_S| \leq T-1$, and ν_S is the set of vertices that stores the items in S.

Theorem 6. *There is a FPTAS for ThOP when the graph G is a clique, every edge has length* 1*, and* $\mathcal{V}_{\min} = \mathcal{V}_{\max}$.

Proof. We modify Algorithm 10 so that it rounds down the profit of each item to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$. Note that each entry of the clique profit

Algorithm 10 ThOPClique($G = (V, E), W, T, s, t, I$)

```
Input: Clique graph $G$, knapsack capacity $W$, time limit $T$, start vertex $s$, end vertex $t$, and item assignments $I$.
Output: An optimum solution for ThOP.
Index vertices and items as described in Section 3.
Let $A$ be an empty clique profit table. Set $A[1] = (0, 0, \emptyset)$.
for $i = 2$ to $|V| - 1$ do
    Let $u$ be the vertex with index $i$.
    Call $UpdateCliqueProfitTables(W, T, A, u)$.
end for
Return $BuildKnapsack(A, I)$.
```

table contains at most T tuples with the same profit (one for each value of $|\nu|$), and the weights of the items do not need to be rounded up; therefore, each entry in the clique profit table has at most $O(P_r T)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items and $T < n$ is the time limit. Hence, this modified Algorithm 10 has time complexity $O(\frac{|I|}{\epsilon} n^2)$ and so it is a FPTAS. ☐

References

[1] Bloch-Hansen, A., Page, D., and Solis-Oba, R.: A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs. In International Workshop on Combinatorial Algorithms (IWOCA), pp. 87-98. Springer, 2023.

[2] Blum, A., Chawla, S., Karger, D., Lane, T., Meyerson, A., and Minkoff, M.: Approximation algorithms for orienteering and discounted-reward TSP. SIAM Journal on Computing **37**(2), pp. 653-670, 2007.

[3] Bonyadi, M., Michalewicz, Z., and Barone, L.: The travelling thief problem: the first step in the transition from theoretical problems to realistic problems. In IEEE Congress on Evolutionary Computation (CEC), pp. 1037-1044. IEEE, 2013.

[4] Chagas, J., Wagner, M.: Ants can orienteer a thief in their robbery. Operations Research Letters **48**(6), pp. 708-714. 2020.

[5] Chagas, J., Wagner, M.: Efficiently solving the thief orienteering problem with a max-min ant colony optimization approach. Optimization Letters **16**(8), pp. 2313-2331. 2022.

[6] Faêda, L. and Santos, A.: A genetic algorithm for the thief orienteering problem. In 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1-8. IEEE, 2020.

[7] Fang, S., Lu, E., and Tseng, V.: Trip recommendation with multiple user constraints by integrating point-of-interests and travel packages. In IEEE 15th International Conference on Mobile Data Management, vol. 1, pp. 33-42. IEEE, 2014.

[8] Freeman, N., Keskin, B., and Çapar, İ.: Attractive orienteering problem with proximity and timing interactions. European Journal of Operational Research **266**(1), pp. 354-370. 2018.

[9] Golden, B., Levy, L., and Vohra, R.: The orienteering problem. Naval Research Logistics (NRL) **34**(3), pp. 307-318. 1987.

[10] Karger, R., Motwani, R., and Ramkumar, G.: On approximating the longest path in a graph. Algorithmica **18**(1), pp. 82-98. Springer, 1997.

[11] Konstantakopoulos, G., Gayialis, S., and Kechagias, E.: Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification. Operational Research, pp. 1-30. 2020.

[12] Lin, C., Choy, K., Ho, G., Chung, S., and Lam, H.: Survey of green vehicle routing problem: past and future trends. Expert Systems with Applications **41**(4), pp. 1118-1138. 2014.

[13] Neumann, F., Polyakovskiy, S., Skutella, M., Stougie, L., and Wu, J.: A fully polynomial time approximation scheme for packing while traveling. In International Symposium on Algorithmic Aspects of Cloud Computing, pp. 59-72. Springer, Cham, 2019.

[14] Polyakovskiy, S. and Neumann, F.: The packing while traveling problem. European Journal of Operational Research **258**(2), pp. 424-439. 2017.

[15] Roostapour, V., Pourhassan, M., and Neumann, F.: Analysis of baseline evolutionary algorithms for the packing while travelling problem. In Proceedings of the 1th ACM/SIGEVO Conference on Foundations of Genetic Algorithms, pp. 124-132. 2019.

[16] Santos, A. and Chagas, J.: The thief orienteering problem: formulation and heuristic approaches. In IEEE Congress on Evolutionary Computation (CEC), pp. 1-9. IEEE, 2018.

[17] Vansteenwegen, P., Souffriau, W., and Van Oudheusden, D.: The orienteering problem: A survey. European Journal of Operational Research **209**(1), pp. 1-10, 2011.

Chapter 5

5 Paper 3: Thief Orienteering on Undirected Graphs

A preliminary version of this paper was first published in 2024 (pp. 248-262) in the Proceedings of the 8th International Symposium on Combinatorial Optimization (ISCO) by Springer [13]. The Version of Record for the preliminary paper is available online at: https://doi.org/10.1007/978-3-031-60924-4_19. An extended version of this paper was later submitted to the journal Acta Informatica and is currently under review.

This paper investigates techniques for transforming instances of the thief orienteering problem on undirected outerplanar and series-parallel graphs into equivalent instances of the thief orienteering problem on directed acyclic graphs.

The Thief Orienteering Problem on Series-Parallel Graphs

Abstract

In the thief orienteering problem an agent called a *thief* carries a knapsack of capacity W and has a time limit T to collect a set of items of total weight at most W and maximum profit along a simple path in a weighted graph $G = (V, E)$ from a start vertex s to an end vertex t. There is a set I of items each with weight w_i and profit p_i that are distributed among $V \setminus \{s, t\}$. The time needed by the thief to travel an edge depends on the length of the edge and the weight of the items in the knapsack at the moment when the edge is traversed.

There is a polynomial-time approximation scheme for the thief orienteering problem on directed acyclic graphs that produces solutions that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$. We give a polynomial-time algorithm for transforming instances of the problem on outerplanar and series-parallel graphs into equivalent instances of the thief orienteering problem on directed acyclic graphs; therefore, yielding a polynomial-time approximation scheme for the thief orienteering problem on these graph classes that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$.

Keywords: thief orienteering problem, knapsack problem, approximation algorithm, approximation scheme, outerplanar, series-parallel

1 Introduction

Let $G = (V, E)$ be a weighted graph with n vertices, where two vertices $s, t \in V$ are designated the *start* and *end* vertices, respectively. Let there be a set I of items, where each item $i_j \in I$ has a non-negative integer weight w_j and profit p_j. Each vertex $u \in V \setminus \{s, t\}$ stores a subset $S_u \subseteq I$ of items such that $S_u \cap S_v = \emptyset$ for all $u \neq v$ and $\bigcup_{u \in V \setminus \{s,t\}} S_u = I$. Additionally, every edge $e = (u, v) \in E$ has a length $d_{u,v} \in \mathbb{Q}^+$.

In the *thief orienteering problem* (ThOP) the goal is for an agent called a *thief* to travel a simple path in G between s and t within a given time $T \in \mathbb{Q}^+$ while collecting items in a knapsack with capacity $W \in \mathbb{Z}^+$ taken from the vertices along the path of total weight at most W and maximum total profit.

The time needed to travel between two adjacent vertices u, v depends on the length of the edge connecting them and on the weight of the items in the knapsack when the edge is traveled; specifically, the travel time between adjacent vertices u and v is $d_{u,v}/\mathcal{V}$ where $\mathcal{V} = \mathcal{V}_{\max} - w(\mathcal{V}_{\max} - \mathcal{V}_{\min})/W$, w is the current weight of the items in the knapsack, and $\mathcal{V}_{\min}$ and $\mathcal{V}_{\max}$ are the minimum and maximum velocities of the thief.

ThOP is a generalization of the knapsack and longest path problems that has not been extensively studied, but related travelling problems such as the travelling thief problem [4] and some variants of orienteering [12] are well-studied and have applications in areas such as route planning [10] and circuit design [4].

ThOP was first formulated in 2018 by Santos and Chagas [13] and several heuristics have since been designed for it [5, 6, 9]. In 2023, Bloch-Hansen et al. [3] proved that there exists no approximation algorithm for the thief orienteering problem with constant approximation ratio unless $P = NP$, and they presented a polynomial-time approximation scheme (PTAS) for ThOP when the input graph G is directed and acyclic (DAG) that produces solutions that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$.

In this paper we consider ThOP on two special classes of graphs: outerplanar and series-parallel. These kinds of graphs have applications in diverse areas such as scheduling [15], chemoinformatics [14], VLSI [7], physics [2], and electrical circuits [8]. Our motivation for studying ThOP on this graph class was production optimization, an important problem in the manufacturing industry.

The manufacturing of a product might require several stages involving different resources, equipment, and personnel. The goal of production optimization is to design and schedule the stages needed to manufacture a product at minimum cost. A directed graph can be used to model the different manufacturing possibilities for a product, with vertices representing manufacturing stages and edges denoting the order in which the stages need to be performed [16]. Every vertex has attributes like cost of the manufacturing stage, resources needed, completion time, and personnel required. Edges indicate time needed to move the (partial) product from one stage to another. The best plan for manufacturing a product then corresponds to a path in the directed graph from the vertex corresponding to the start of the manufacturing process to the vertex corresponding to the completion of the product. This best path must satisfy several constraints like production time, resource cost, and number of personnel needed to fabricate the product, and so this problem can be modelled with the thief orienteering problem. Due to the sequential nature of many manufacturing processes, the different production plans corresponding to the various ways to manufacture a process can be conveniently modelled using series-parallel graphs [1].

Another application for ThOP on series-parallel graphs includes system reliability, which is fundamental in system design. A series system consists of a sequence S of subsystems, and the failure of any subsystem causes the failure of the entire system. To prevent this, redundant subsystems are added in parallel to ensure a minimum level of reliability. These systems are called series-parallel systems [11]. To determine the optimal number of redundant subsystems, a series-parallel graph can be used to model the problem. Each vertex in the graph corresponds to a different number of possible copies of each subsystem. Vertices have costs indicating the cost of the redundant subsystems and they also have profits indicating the reliability achieved with the corresponding redundant systems. An edge from a vertex u to an adjacent vertex v in the sequence S is given length equal to the number of subsystems represented by u. The goal is to find a minimum cost path from a vertex s representing the beginning of the sequence S to a vertex corresponding to the end of the sequence that achieves a failure rate no larger than a maximum bound W, and that uses at most a given number T of additional subsystems. This problem can also be modelled with the thief orienteering problem on series-parallel graphs.

Our strategy for dealing with outerplanar and series-parallel graphs is to first transform them into DAGs and then to use the PTAS of [3] that produces solutions of time at most $(1+\epsilon)T$. The transformation into DAGs is not easy as we need to preserve all simple paths from s to t while avoiding the formation of cycles. To achieve this, we create copies of the vertices and edges of the input graph and carefully select the directions of the edges so every simple undirected path in the input graph has a corresponding, equivalent directed path in the DAGs produced by our algorithm.

The main challenge in achieving polynomial running time when transforming this class of graphs into DAGs is to preserve all the different paths from s to t without adding a very large number of vertices and edges or creating any cycles. We show how to overcome these challenges by exploiting the special structures of outerplanar and series-parallel graphs. For series-parallel graphs we need to create at most two copies of each vertex and edge, but assigning the correct directions to the edges is not easy. For outerplanar graphs we need to create several copies of every vertex and edge.

In the rest of the paper we present our algorithm to transform instances of ThOP on series-parallel graphs into equivalent instances of ThOP on DAGs.

In Sections 2 and 3 we present our algorithms to transform instances of ThOP on outerplanar and series-parallel graphs into equivalent instances of ThOP on DAGs.

2 Thief Orienteering on Outerplanar Graphs

A *planar* graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. An *outerplanar* graph is a planar graph for which all of its vertices belong to the outer face.

Let G be an undirected outerplanar graph. Consider each vertex u, $u \neq s$, $u \neq t$, in G that has degree 1. A simple path from s to t cannot route through u. Therefore, we can delete these vertices u, and any subsequent vertices $v \neq s$, $v \neq t$ with degree 1 that are exposed from this process.

If s has degree 1 we direct all edges away from s along a simple path to the closest vertex v (or t if t is closer) that has degree at least 3 and rename v

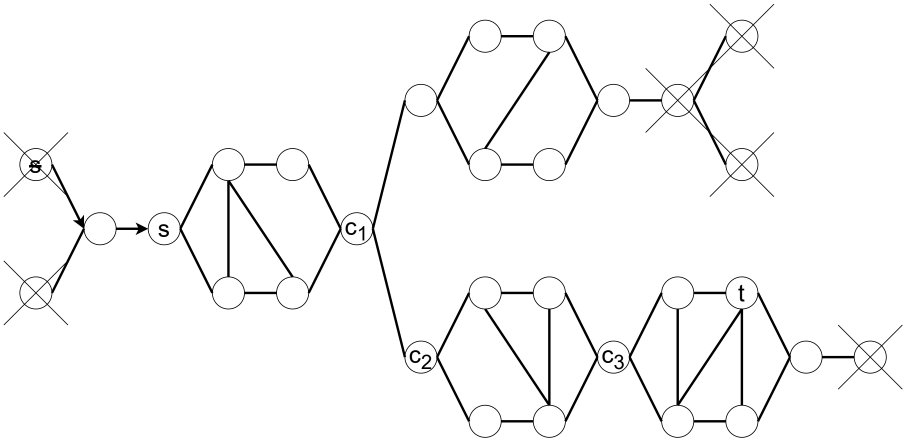


Fig. 1 An example outerplanar graph that has vertices of degree 1. Note that s is re-named, vertices other than s or t with degree 1 are removed, and the cut vertices c_1, c_2, and c_3 are labelled.

to s. Similarly, if t has degree 1 let v (or s if s is closer) be the closest vertex to t with degree at least 3; direct all edges away from v along the simple path to t and rename v to t (see Figure 1).

Note that for the cases when (i) G is a tree, (ii) there is only one simple path in G from s to t, or (iii) all vertices in G have degree less than 3, then directing the edges from s to t transforms the instance of ThOP on G into an equivalent instance on a DAG. Therefore, in the sequel we assume that at least one vertex in G has degree at least 3 and that there is more than one simple path in G from s to t.

2.1 Cut Vertices

A *cut vertex* is a vertex whose removal from a connected graph G disconnects it into at least 2 non-empty connected components. Consider a shortest path p between s and t. Let c_1, c_2, ..., c_{k-1} be the cut vertices, other than s and t, along p in increasing order of distance to s (see Figure 1). Let $c_0 = s$ and $c_k = t$. For each c_x from $x = 1, 2, ..., k$, we define the graph $\mathcal{G}_{\ell_x}$ that includes the cut vertices c_{x-1} and c_x and all vertices and edges in all simple paths between c_{x-1} and c_x (see Figure 2).

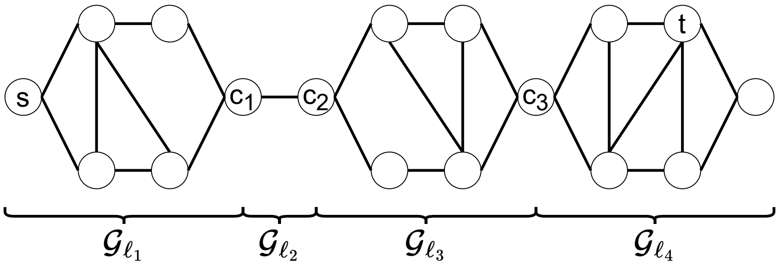


Fig. 2 The outerplanar graph from Figure 1 after identifying the subgraphs $\mathcal{G}_{\ell_x}$.

2.2 Transforming $\mathcal{G}_{\ell_x}$ into a DAG

Consider one of the graphs $\mathcal{G}_{\ell_x}$. For the purpose of transforming $\mathcal{G}_{\ell_x}$ into a DAG, if s is not in $\mathcal{G}_{\ell_x}$ we temporarily let $s = c_{x-1}$. Additionally, if t is not in $\mathcal{G}_{\ell_x}$, then we temporarily let $t = c_x$. Observe that either in $\mathcal{G}_{\ell_x}$ s and t are adjacent or $\mathcal{G}_{\ell_x}$ has a cycle containing s and t. In the former case, $\mathcal{G}_{\ell_x}$ is trivially transformed into the desired DAG by directing the edge from s to t, so we assume below that $\mathcal{G}_{\ell_x}$ contains a cycle. Moreover, if $\mathcal{G}_{\ell_x}$ contains no chords, then all of the edges in $\mathcal{G}_{\ell_x}$ can be directed away from s and towards t to form an equivalent instance of ThOP on a DAG. Therefore, we assume that $\mathcal{G}_{\ell_x}$ contains at least one chord.

Draw the graph $\mathcal{G}_{\ell_x}$ so that its vertices lie on a circle and s and t are on the left and right sides of the circle, respectively. We index the vertices in $\mathcal{G}_{\ell_x}$ as follows: (i) s has index 1, (ii) in a clockwise-manner, increasingly index all the vertices around the circle starting from s and up to but excluding t, (iii) in a counter-clockwise-manner, continue indexing all the vertices around the circle from s to t, so that t has the largest index. Hence, the vertices in the top portion of the circle are $s = 1, 2, 3, ..., i, t$ and the vertices in the bottom portion are $s = 1, i+1, i+2, ..., t-1, t$ (see Figure 3). The path $s, 2, 3, ..., i, t$ is called the *top path* and $s, i+1, ..., t-1, t$ is called the *bottom path*. The top and bottom paths form the *outer circle*. The index of a vertex u will be denoted as $u.index$.

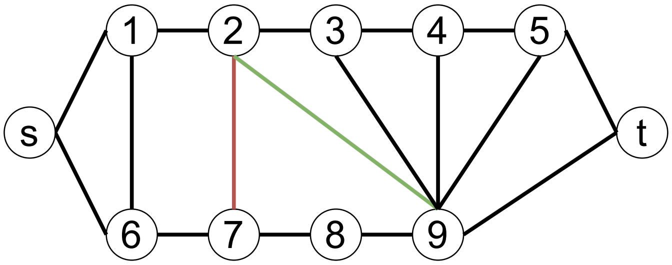


Fig. 3 An example of a subgraph $\mathcal{G}_{\ell_x}$ after indexing the vertices; the graph consists of a cycle with chords. Vertex 2 has two incident chords: the first chord is red and the second chord is green.

A vertex u might have several chords incident on it. We call the chord (u, v) incident on u for which v has the lowest index of the endpoints of the chords incident on u, the *first chord* of u. Similarly, the *second chord* of u is the chord (u, v') where v' has the second lowest index among the other endpoints of the chords incident on u, and so on.

We transform $\mathcal{G}_{\ell_x} = (V_x, E_x)$ into a DAG by first creating a graph $\overrightarrow{G}_{\ell_x} = (\mathcal{V}_x, \mathcal{E}_x)$ where $\mathcal{V}_x = V_x$ and $\mathcal{E}_x$ contains the non-chord edges in E_x. We direct the edges of $\mathcal{E}_x$ away from s and toward t; then we add new vertices and edges to $\mathcal{E}_x$ as explained below.

2.2.1 Adding Directed Chords to $\mathcal{G}$

The undirected chords of $\mathcal{G}_{\ell_x}$ must be transformed into directed chords of $\mathcal{G}$ carefully to ensure that $\mathcal{G}$ is acyclic, all simple paths in $\mathcal{G}_{\ell_x}$ between s and t also exist in $\mathcal{G}$, and no additional simple paths between s and t exist in $\mathcal{G}$ that did not exist in $\mathcal{G}_{\ell_x}$.

For an undirected chord (u, v) in $\mathcal{G}_{\ell_x}$, the thief might take a path that travels the chord from u to v or a path that travels it from v to u. In order to include both paths in $\overrightarrow{G}_{\ell_x}$, we add to $\overrightarrow{G}_{\ell_x}$ several copies of u and v. For each vertex $u \in V_x$ incident on z different chords, we add to $\mathcal{V}_x$ z copies of u: $\hat{u}_1$, $\hat{u}_2$, ..., $\hat{u}_z$; we call u an *original vertex* and each $\hat{u}_i$ is called a *duplicate vertex*. Each one of these duplicates of u stores the same set of items as u.

For each undirected chord (u, v) in $\mathcal{G}_{\ell_x}$ we add to $\overrightarrow{G}_{\ell_x}$ two different types of directed chords that correspond to the different types of simple paths from s to t in $\mathcal{G}_{\ell_x}$ that traverse (u, v). The first type of directed chord has the form $(u, \hat{v}_j)$, where u is an original vertex and $\hat{v}_j$ is a duplicate vertex. The second type of directed chord has the form $(\hat{u}_i, \hat{v}_j)$, where $\hat{u}_i$ and $\hat{v}_j$ are duplicate vertices. We add these two types of directed chords to $\overrightarrow{G}_{\ell_x}$ as follows:

- (a) For every undirected chord (u, v) in $\mathcal{G}_{\ell_x}$, we add two directed chords of the first type to $\overrightarrow{G}_{\ell_x}$: the directed chords $(u, \hat{v}_j)$ and $(v, \hat{u}_i)$, where (u, v) is v's j^{th} chord and u's i^{th} chord. The lengths of these two directed chords are both $d_{u,v}$, the length of the undirected chord. These chords allow the thief to travel from u to v or from v to u. In Figure 4 we show an example of the first type of directed chords: chords $(1, \hat{6}_1)$ and $(6, \hat{1}_1)$.
- (b) For every undirected chord (u, v) in $\mathcal{G}_{\ell_x}$ that is the i^{th} chord of u, where $i > 1$, and the first chord of v (note that (u, v) cannot be the j^{th} chord of v with $j > 1$ because the input graph is outerplanar), we add the directed chords $(\hat{u}_h, \hat{v}_1)$ for every $h < i$, each of length $d_{u,v}$, to $\overrightarrow{G}_{\ell_x}$. For example, for chord $(2, 9)$ in Figure 3 the algorithm adds the directed chord $(\hat{2}_1, \hat{9}_1)$ as shown in Figure 5 because $(2, 9)$ is the second chord of vertex 2.
  Note that if vertex u has several chords we need to create several duplicates of u, where the i^{th} duplicate $\hat{u}_i$ is incident to directed chords of the form $(\hat{u}_i, \hat{v}_1)$ where vertices $\hat{v}_1$ are the $i + 1$, $i + 2$, ..., neighbors of u. These directed chords allow directed paths in $\overrightarrow{G}_{\ell_x}$ corresponding to paths in $\mathcal{G}_{\ell_x}$ that go through two chords (v, u), (u, w) incident on u while preventing these directed paths from going back to vertex u or one of its duplicates, hence avoiding the creation of cycles (for us, a cycle is a path that includes two copies of the same vertex).

2.2.2 Connecting Duplicate Vertices to the Top and Bottom Paths

Let a vertex v contain several chords in its chord list where the endpoints of these chords have indices $i, i + x, i + y, ..., i + z$. In the sequel, we say that the

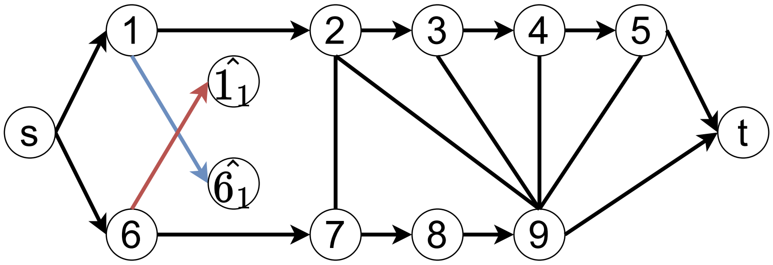


Fig. 4 Directing the chord $(1,6)$ of Figure 3: vertices $\hat{1}_1$ and $\hat{6}_1$ are added to $\mathcal{G}$ along with directed chords $(1,\hat{6}_1)$ and $(6,\hat{1}_1)$.

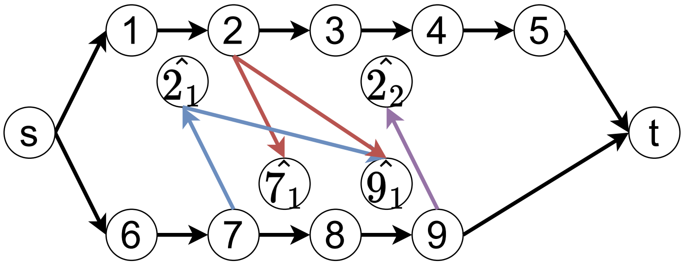


Fig. 5 Directing the chords incident on vertex 2 of Figure 3.

chords of v *span* the vertices with indices from i to $i+z$; the start of the span of v's chords is i and the end of the span of v's chords is $i+z$. Note that v's chords do not need to connect to every index from i to $i+z$, but for any missing index q the corresponding vertex u does not have any incident chords on it. For example, in Figure 3 vertex 2 has a span from vertex 7 to 9, but vertex 8 has no incident chords.

For v's first chord (v,u'), vertex u' is called the *start of v's span*. For v's last chord (v,u''), vertex u'' is called the *end of v's span*. All vertices u in $\mathcal{G}_{\ell_x}$ such that $u'.index < u.index < u''.index$ are in the *middle of v's span*.

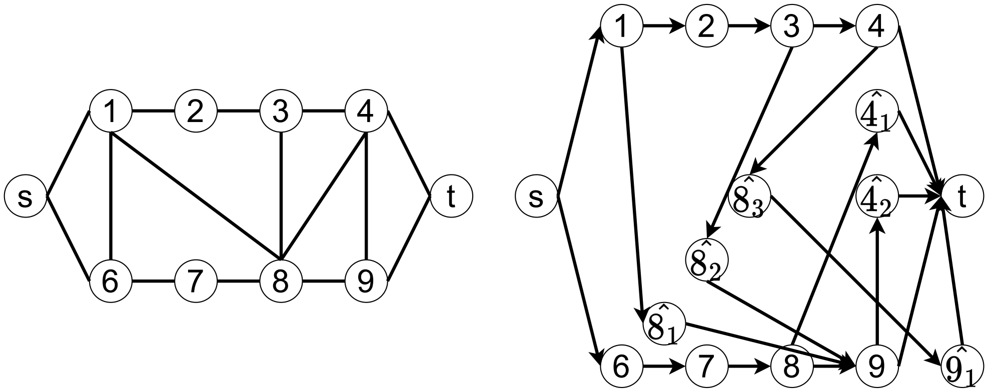


Fig. 6 The chords of vertex 1 span vertices 6 to 8. The chords of vertex 8 span vertices 1 to 4. The chords of vertex 4 span vertices 8 and 9. Vertices 1, 6, and 8 are located at the start of a span. Vertices 2, 3, and 7 are located in the middle of a span. Vertices 4, 8, and 9 are located at the end of a span.

Let the chords of v span the vertices from i to $i+z$ and let u be incident on j chords, where $i \leq u.index \leq i+z$. Let vertex q be the vertex following vertex u in the outer circle: So either $q.index = u.index + 1$, or u is adjacent to t and hence $q = t$. We add the following directed edges to $\overrightarrow{G}_{\ell_x}$:

- (a) If vertex u is the start of v's span: For each duplicate vertex $\hat{u}_i$ where $i < j$ add a directed edge $(\hat{u}_i, q)$ with length $d_{u,q}$ and for $\hat{u}_j$ add a directed edge $(\hat{u}_j, \hat{q}_1)$ with length $d_{u,q}$. Note that if $\mathcal{G}_{\ell_x}$ has a node v incident on chords (v, u) and (v, q) then the directed edges $(v, \hat{u}_j)$ and $(q, \hat{v}_2)$ are in $\overrightarrow{G}_{\ell_x}$ and so we add the directed edge $(\hat{u}_j, \hat{q}_1)$ to $\overrightarrow{G}_{\ell_x}$ instead of edge $(\hat{u}_j, q)$ as this latter edge would create a cycle. For example, in Figure 6 (left) vertex 8 is located at the start of the span of the chords of vertex 4; vertex 8 has 3 chords in its chord list so $j = 3$, and hence for the duplicates $\hat{8}_1$ and $\hat{8}_2$ we add the directed edges $(\hat{8}_1, 9)$ and $(\hat{8}_2, 9)$ and for the duplicate $\hat{8}_3$ we add the directed edge $(\hat{8}_3, \hat{9}_1)$. Observe that we do not add the directed edge $(\hat{8}_3, 9)$ because then the path $4, \hat{8}_3, 9, \hat{4}_2$ would visit the items in vertex 4 twice.
- (b) If u is in the middle of v's span: If u has one incident chord (note that u cannot have more than one incident chord), then add a directed edge $(\hat{u}_1, \hat{q}_1)$ with length $d_{u,q}$. If u has no incident chords, then we add duplicate vertex $\hat{u}_1$ and a directed edge $(\hat{u}_1, \hat{q}_1)$ with length $d_{u,q}$ (see in Figure 8 chord $(\hat{8}_1, \hat{9}_1)$). As explained above, these edges are incident on $\hat{q}_1$ and not on q to avoid the creation of cycles.
- (c) If u is the end of v's span: For each duplicate vertex $\hat{u}_i$ add the directed edge $(\hat{u}_i, q)$ with length $d_{u,q}$ (see in Figure 7 chords $(\hat{1}_1, 2)$ and $(\hat{6}_1, 7)$). Since q is not in v's span, directed edge $(\hat{u}_i, q)$ cannot create cycles.

We illustrate the above algorithm using several examples. In Figure 7 we show how the duplicate vertices $\hat{1}_1$ and $\hat{6}_1$ are connected to the top and bottom paths.

In Figure 8 we show how the duplicate vertices for vertex 2 are connected to the top and bottom paths. Vertex 7 is the start of a span, so vertex $\hat{7}_1$ cannot directly connect to vertex 8; instead, vertex $\hat{7}_1$ must connect to vertex $\hat{8}_1$, as described in the first rule above. Vertex 8 has no chords incident on it, but since it is in the middle of a span, the duplicate vertex $\hat{8}_1$ must be created and we add a directed edge from it to vertex $\hat{9}_1$, as described in the second rule. Finally, vertex 9 is at the end of a span, so by the third rule we add the directed edge $(\hat{9}_1, t)$.

Note that vertex 2 is the start of a span from the chords of vertex 9; therefore, by the first rule since vertex 2 has two chords in its chord list we add directed edges $(\hat{2}_1, 3)$ and $(\hat{2}_2, \hat{3}_1)$.

2.3 Algorithm Analysis

Algorithm 1 transforms an outerplanar graph $G = (V, E)$ into a DAG $\mathcal{G}$.

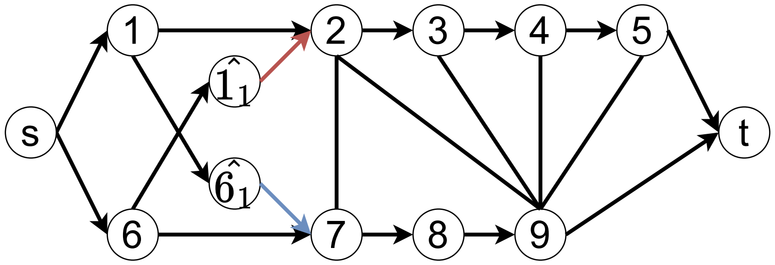


Fig. 7 Processing the chord $(1, 6)$: the directed edges $(\hat{1}_1, 2)$ and $(\hat{6}_1, 7)$ are added.

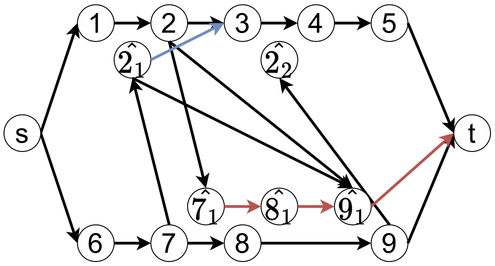


Fig. 8 Connecting the duplicate vertices for vertex 2. Note that a copy of 8, $\hat{8}_1$, was created, even though vertex 8 has no incident chords, because 8 is in the middle of a span from vertex 2.

Algorithm 1 OuterPlanarToDAG($G = (V, E)$, s, t, I)

```
**Input:** Outerplanar graph $G$, start vertex $s$, end vertex $t$, and item assignments $I$.
**Output:** A DAG $\mathcal{G}$ containing paths from $s$ to $t$ equivalent to those in $G$.
Process vertices with degree 1 as described at the beginning of Section 2.
Let $p$ be a shortest path from $s$ to $t$.
Let $c_1$, $c_2$, ..., $c_{k-1}$ be $p$'s cut vertices.
for each $x = 1$ to $k - 1$ do
    Create $\mathcal{G}_{\ell_x}$ and transform it into a DAG as described in Section 2.2.
end for
Output the DAG $\mathcal{G}$ created by transforming each $\mathcal{G}_{\ell_x}$.
```

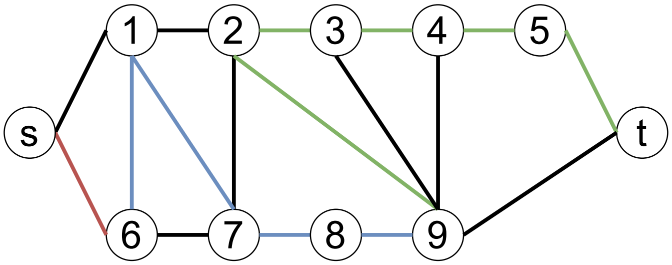


Fig. 9 A path σ in G that routes through the red non-chord $(s, 6)$, the blue chords $(6, 1)$ and $(1, 7)$, the blue non-chords $(7, 8)$ and $(8, 9)$, the green chord $(9, 2)$, and the green non-chords $(2, 3)$, $(3, 4)$, $(4, 5)$, and $(5, t)$.

Definition 1 An undirected path σ from s to t in G and a directed path σ' from s to t in $\mathcal{G}$ are *equivalent* if both have the same number of vertices, each vertex in σ

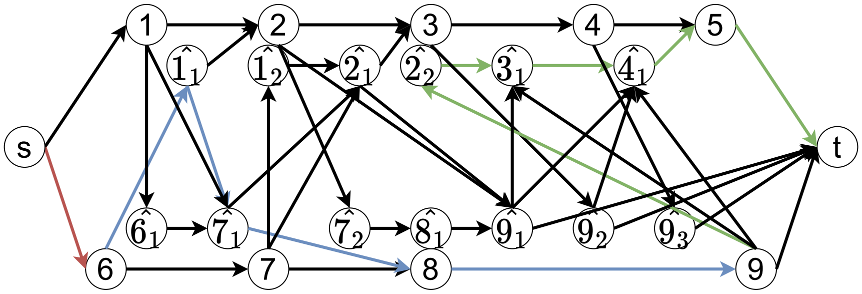


Fig. 10 The path σ' in $\mathcal{G}$ corresponding to σ' in Figure 9 routes through the red non-chord $(s, 6)$, the blue chords $(6, \hat{1}_1)$ and $(\hat{1}_1, \hat{7}_1)$, the blue non-chords $(\hat{7}_1, 8)$ and $(8, 9)$, the green chord $(9, \hat{2}_2)$, and the green non-chords $(\hat{2}_2, \hat{3}_1)$, $(\hat{3}_1, \hat{4}_1)$, $(\hat{4}_1, 5)$, and $(5, t)$.

and the corresponding vertex in σ' store the same set of items, and every edge in σ and its corresponding edge in σ' have the same length.

Lemma 1 *Algorithm 1 transforms an instance of ThOP on an undirected outerplanar graph G into an instance of ThOP on a directed graph $\mathcal{G}$ with no cycles, so all paths from s to t in $\mathcal{G}$ are simple.*

Proof Algorithm 1 transformed the graph G into the directed graph $\mathcal{G}$ by adding several duplicate vertices corresponding to existing vertices in G and several duplicate edges corresponding to existing edges in G and then directing all the edges. Note that two vertices in $\mathcal{G}$ are adjacent only if the corresponding vertices in G are also adjacent.

To prove that there are no cycles in $\mathcal{G}$, we show that there are two possible forms of cycles in $\mathcal{G}$ and prove that neither exist. Since all directed edges in $\mathcal{G}$ correspond to undirected edges in G, a cycle in $\mathcal{G}$ would be either (1) two anti-parallel directed edges in $\mathcal{G}$ corresponding to an undirected edge (u, v) in G or (2) a directed cycle in $\mathcal{G}$ corresponding to a cycle in G.

(1) For each undirected non-chord (u, v) in G there are no corresponding anti-parallel directed edges in $\mathcal{G}$ because Algorithm 1 directs all non-chords added to $\mathcal{G}$ towards a vertex with a higher index. For each undirected chord (u, v) in G, two types of directed chords are added to $\mathcal{G}$. For the first type, the two directed chords $(u, \hat{v}_j)$ and $(v, \hat{u}_i)$ are added to $\mathcal{G}$, and since these two edges end at different duplicate vertices they do not create anti-parallel directed chords. For the second type, for each undirected chord (u, v) in G that is the i^{th} chord of u, where $i > 1$, and the first chord of v the directed chords $(\hat{u}_h, \hat{v}_1)$, for every $h < i$, are added. Observe that this process does not add anti-parallel directed chords between two vertices in $\mathcal{G}$, because that would require that (u, v) is the j^{th} chord of v with $j > 1$.

(2) Consider a cycle C of G that contains edges (u, v) and (u', v') where u and v are in the top path and u' and v' are in the bottom path; then, the directed edges added to $\mathcal{G}$ corresponding to C cannot form a cycle as non-chord edges are always directed toward a vertex with larger index. Hence, the only cycles of G that could create a directed cycle in $\mathcal{G}$ are those where only non-chord edges are either in the top path or in the bottom path. Consider a simple cycle $C : u, v_1, v_2, ..., v_z, u$ where u is the only vertex of C in the top path (the case where u is in the bottom path is similar). Then, Algorithm 1 will add the directed edges shown in Figure 11 for

this cycle C. As the figure shows, there are no directed cycles because (i) every edge directed from the top path to the bottom path ends at a duplicate vertex; (ii) vertices v_1, v_2, ..., v_{z-1} have only one chord so each duplicate vertex $\hat{v}_i$ has only one outgoing edge, and these edges are all non-chords that end at duplicate vertices; and (iii) every edge directed from the bottom path to the top path starts at an original vertex.

Therefore, since there are no anti-parallel directed edges or directed cycles in $\mathcal{G}$, then all paths from s to t in $\mathcal{G}$ are simple.

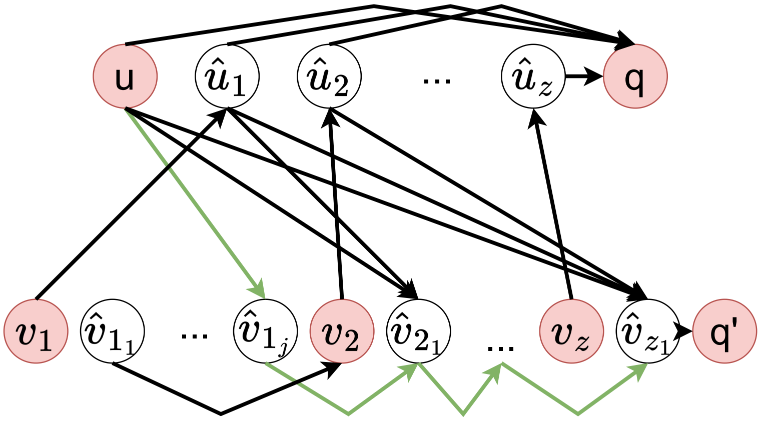


Fig. 11 A simple cycle $C : u, v_1, v_2, ..., v_z, u$ in G, where u is the only vertex of C in the top path, is not encoded as a directed cycle in $\mathcal{G}$ because the algorithm does not add any directed edges from the duplicate vertices of $v_1, v_2, ..., v_z$ back to u or any of its duplicates. The original vertices are shaded red and the part of C that exists in $\mathcal{G}$ is shaded green.

□

Lemma 2 *Algorithm 1 transforms an instance of ThOP on an undirected outerplanar graph G into an instance of ThOP on a directed graph $\mathcal{G}$ such that for every simple path σ from s to t in G, there is an equivalent directed path σ' from s to t in $\mathcal{G}$.*

Proof We prove that for every simple path σ in G from s to t, there is an equivalent directed path σ' in $\mathcal{G}$ from s to t. To do this, we show how to select the directed edges in $\mathcal{G}$ that correspond to the undirected edges in σ and we show that these directed edges form a path from s to t in $\mathcal{G}$.

Consider a simple path σ in G from s to t. We construct an equivalent path σ' in $\mathcal{G}$ by considering the edges in σ one-by-one and including the corresponding edges in σ' in the following manner:

- If the next edge (u, v) in σ is v's j^{th} chord:
  - If the edge most recently included in σ' is a directed chord $(w, \hat{u}_h)$ of the first type, then (u, v) is the k^{th} chord of u with $k > h \geq 1$ and hence (u, v) is the first chord of v and so the algorithm in Section 2.2.1 (b) added to $\mathcal{G}$ the directed edge $(\hat{u}_h, \hat{v}_1)$, which is included in σ'. For example, in Figure 9 after the chord $(6, \hat{1}_1)$ is included in σ', the directed chord $(\hat{1}_1, \hat{7}_1)$ is included in σ'.

  - If the edge most recently included in σ' is a directed chord $(\hat{w}_i, \hat{u}_h)$ of the second type, then $h = 1$ (by the algorithm in Section 2.2.1 (b)) and so (u, v) is not the first chord of u. Therefore, the algorithm in Section 2.2.1 (b) added to $\mathcal{G}$ the directed edge $(\hat{u}_1, \hat{v}_1)$, which is included in σ'.
  - If σ' is empty or if the edge most recently included in σ' is a non-chord edge (w, u) where u is an original vertex, then the directed chord $(u, \hat{v}_j)$ is included in σ': Since u is an original vertex and (u, v) is the j^{th} chord of v, then the algorithm in Section 2.2.1 (a) added to $\mathcal{G}$ the directed chord $(u, \hat{v}_j)$ of the first type. For example, in Figure 9 after the non-chord edge $(s, 6)$ is included in σ', the directed chord $(6, \hat{1}_1)$ is included in σ'.
  - If the edge most recently included in σ' is a non-chord edge $(w, \hat{u}_h)$, where $\hat{u}_h$ is a duplicate vertex, then $h = 1$ (by the algorithm in Section 2.2.2 (b)) and so (u, v) is not the first chord of u. Therefore, the algorithm in Section 2.2.1 (b) added to $\mathcal{G}$ the directed edge $(\hat{u}_1, \hat{v}_1)$, which is included in σ'.

- If the next edge (u, v) in σ is a non-chord edge:
  - If σ' is empty or if the edge most recently included in σ' is a non-chord edge (w, u) where u is an original vertex, then the directed edge (u, v) is included in σ'; since the algorithm in Section 2.2.1 included all of the original non-chord edges from G, then (u, v) must be in $\mathcal{G}$. For example, in Figure 9 after the non-chord edge $(\hat{4}_1, 5)$ is included in σ', the directed non-chord edge $(5, t)$ is included in σ'.
  - Otherwise, if the edge most recently included in σ' is $(w, \hat{u}_i)$ where $\hat{u}_i$ is a duplicate vertex we consider several cases:
    - Let vertex u have j chords in its chord list. If u is located at the start of a span and $i < j$ then the directed non-chord edge $(\hat{u}_i, v)$ is included in σ'; otherwise, if $i = j$ the directed non-chord edge $(\hat{u}_j, \hat{v}_1)$ is included in σ'. Note that the algorithm in Section 2.2.2 (a) added these edges to $\mathcal{G}$. For example, in Figure 9 after the directed chord $(\hat{9}_1, \hat{2}_2)$ is included in σ', the directed non-chord edge $(\hat{2}_2, \hat{3}_1)$ is included in σ'.
    - If u is in the middle of a span then i must be 1 so the directed edge $(\hat{u}_1, \hat{v}_1)$ is included in σ'; note that the algorithm in Section 2.2.2 (b) added this edge to $\mathcal{G}$. For example, in Figure 9 after the directed non-chord edge $(\hat{7}_1, \hat{8}_1)$ is included in σ' (or alternatively, $(\hat{2}_2, \hat{3}_1)$), the directed non-chord edge $(\hat{8}_1, \hat{9}_1)$ is included in σ' (alternatively, $(\hat{3}_1, \hat{4}_1)$).
    - Otherwise, the directed non-chord edge $(\hat{u}_i, v)$ is included in σ'; note that the algorithm in Section 2.2.2 (c) added this edge to $\mathcal{G}$. For example, in Figure 9 after the directed chord $(\hat{3}_1, \hat{4}_1)$ is included in σ', the directed non-chord edge $(\hat{4}_1, 5)$ is included in σ'.

Each of the duplicate vertices created by Algorithm 1 stores the same items as the corresponding vertices in G. Moreover, each of the directed edges created by Algorithm 1 have the same length as the corresponding edges in G. As shown above, for any path σ in G from s to t, the corresponding edges in σ' form a path from s to

t (see Figures 9 and 10). Therefore, for any simple path σ in G from s to t, there is an equivalent simple path σ' in $\mathcal{G}$ from s to t. □

Lemma 3 *Algorithm 1 transforms an instance of ThOP on an undirected outerplanar graph G into an instance of ThOP on a directed graph $\mathcal{G}$ such that for every directed path σ' from s to t in $\mathcal{G}$, there is an equivalent simple path σ from s to t in G.*

Proof Note that the duplicate vertices created by the algorithm in Section 2.2.1 have the same index as the corresponding original vertices. We now prove that a path σ' in $\mathcal{G}$ cannot visit multiple vertices with the same index; to see this, we use a proof by contradiction.

Assume that a path σ' exists in $\mathcal{G}$ that starts and ends at vertices with the same index. Since two vertices in $\mathcal{G}$ are adjacent only if the corresponding vertices in G are also adjacent, and since there are no self-loops in G, then it must be the case that the path σ in G corresponding to σ' is a cycle C. Since non-chord edges in $\mathcal{G}$ are directed to the endpoint with the higher index, then C cannot have one edge belonging to the top path and another edge belonging to the bottom path; therefore, C must be of the form $u, v_1, v_2, ..., v_z, u$ where u is the only vertex of C in the top path (the case where u is in the bottom path is similar).

As previously described in the proof of Lemma 1, Algorithm 1 creates a directed graph $\mathcal{G}$ with the directed edges shown in Figure 11 for the above cycle C. Observe that C does not induce in $\mathcal{G}$ a path starting and ending at vertices with the same index as u as explained at the end of the proof of Lemma 1.

Since there are no cycles in $\mathcal{G}$, a path σ' from s to t in $\mathcal{G}$ cannot visit multiple vertices with the same index, and Algorithm 1 only adds directed edges to $\mathcal{G}$ corresponding to existing edges in G, then for any directed path σ' from s to t in $\mathcal{G}$, selecting the corresponding undirected edges in G forms an equivalent simple path σ from s to t. □

Theorem 1 *There is a PTAS for the thief orienteering problem when the graph G is an undirected outerplanar graph that produces solutions that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$.*

Proof Algorithm 1 transforms an undirected outerplanar graph G into a DAG $\mathcal{G}$ by creating at most two additional vertices per chord, at most one additional vertex for each vertex $u \in V$ incident on no chords, and at most $degree(u)$ additional edges per vertex $u \in V$, and so Algorithm 1 runs in polynomial time. By Theorem 2 in [3], Algorithm 3 in [3] is a PTAS for the thief orienteering problem when the input graph G is a DAG that produces solutions that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$. Therefore, by Lemmas 1, 2, and 3 the combination of algorithms 1 and 3 [3] is a PTAS for the thief orienteering problem when the input graph G is an undirected outerplanar graph that produces solutions that use time at most $T(1 + \epsilon)$ for any $\epsilon > 0$. □

3 Thief Orienteering on Series-Parallel Graphs

A series-parallel graph $G = (V, E, t_1, t_2)$ has two terminal vertices t_1, t_2 and is defined inductively:

- $G = (\{t_1, t_2\}, (t_1, t_2), t_1, t_2)$ is series-parallel.
- if $G_1 = (V_1, E_1, t_1, t_2)$ and $G_2 = (V_2, E_2, t'_1, t'_2)$ are series-parallel then the *series composition* $G = (V_1 \cup V_2, E_1 \cup E_2, t_1, t'_2)$ is series-parallel if $t_2 = t'_1$.
- if $G_1 = (V_1, E_1, t_1, t_2)$ and $G_2 = (V_2, E_2, t'_1, t'_2)$ are series-parallel then the *parallel composition* $G = (V_1 \cup V_2, E_1 \cup E_2, t_1, t_2)$ is series-parallel if $t_1 = t'_1$ and $t_2 = t'_2$. A graph created from a parallel composition is a *parallel graph*.

We can take advantage of the properties of a series-parallel graph G in order to transform it into the desired DAG $\mathcal{G}$ using a polynomial number of additional vertices and edges such that the DAG has the same set of simple paths between s and t as G. Note that the start and end vertices s and t might not be the two terminals in a series-parallel graph.

3.1 Removing Vertices with Degree 1

Let G be an undirected series-parallel graph. If there is only one simple path from s to t in G or if all vertices in G have degree less than 3, then directing the edges away from s and towards t produces the desired DAG. Therefore, in the sequel we assume that at least one vertex in G has degree 3 or higher and that there is more than one simple path in G from s to t.

Consider a vertex u of degree 1, where $u \neq s$ and $u \neq t$. A simple path from s to t cannot route through u, so we delete these vertices u, and any subsequent vertices $v \neq s$, $v \neq t$ with degree 1 that are exposed from this process.

However, if s or t have degree 1 they must be kept in the graph. If s has degree 1 we direct away from s all edges along a simple path to the closest vertex v of degree at least 3; we rename v to s as any path from s to t must route through v. Similarly, if t has degree 1 let v be the closest vertex to t with degree at least 3; direct all edges away from v along the simple path to t and rename v to t.

3.2 Cut Vertices

A *cut vertex* is a vertex whose removal from a connected graph G disconnects it into at least 2 non-empty connected components.

Consider a shortest path p between s and t. Let c_1, c_2, ..., c_{k-1} be the cut vertices, other than s and t, along p in increasing order of distance to s (see Figure 12). Let $c_0 = s$ and $c_k = t$. If the removal of c_0 splits G into two connected components, we delete the component that does not contain t. Similarly, if the removal of c_k splits G into two connected components, we delete the component that does not contain s. If deleting these components causes s or t to have degree 1, then the process described in Section 3.1 is used to rename them.

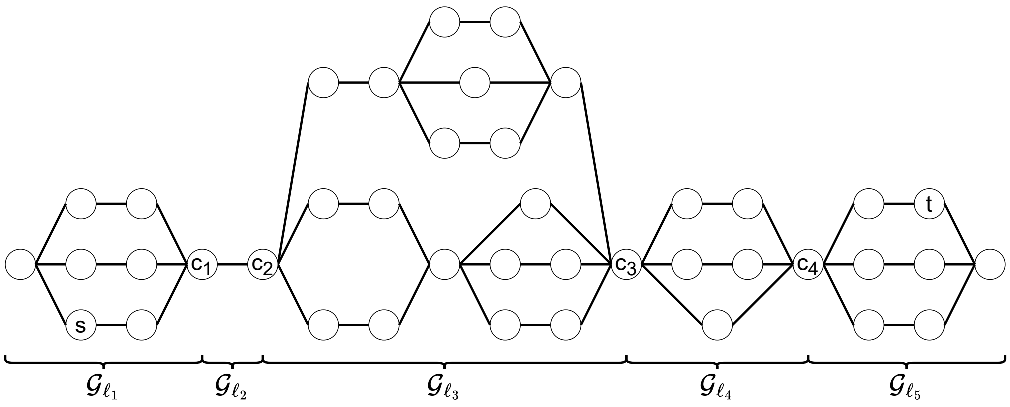


Fig. 12 Identifying the subgraphs $\mathcal{G}_{\ell_x}$ in a series-parallel graph.

For each c_x from $x = 1, 2, ..., k$, we define the graph $\mathcal{G}_{\ell_x}$ that includes the vertices c_{x-1} and c_x and all vertices and edges in all simple paths between c_{x-1} and c_x. We transform each graph $\mathcal{G}_{\ell_x}$ into a DAG as described below. We then combine these DAGs to produce the DAG $\mathcal{G}$ for the input graph G.

3.3 Transforming $\mathcal{G}_{\ell_x}$ into a DAG

The undirected edges of $\mathcal{G}_{\ell_x}$ must be transformed into directed edges of $\mathcal{G}$ carefully to ensure that $\mathcal{G}$ is acyclic, all simple paths in $\mathcal{G}_{\ell_x}$ also exist in $\mathcal{G}$, and no additional simple paths exist in $\mathcal{G}$ that did not exist in $\mathcal{G}_{\ell_x}$.

Definition 2 An undirected path σ from s to t in G and a directed path σ' from s to t in $\mathcal{G}$ are *equivalent* if both have the same number of vertices, each vertex in σ and the corresponding vertex in σ' store the same set of items, and every edge in σ and its corresponding edge in σ' have the same length.

Consider one of the graphs $\mathcal{G}_{\ell_x}$. If all vertices in $\mathcal{G}_{\ell_x}$ have degree at most 2, directing all edges of $\mathcal{G}_{\ell_x}$ away from c_{x-1} and towards c_x creates a DAG with one simple path that is equivalent to the one simple path in G from c_{x-1} to c_x. So, in the sequel we assume that each graph $\mathcal{G}_{\ell_x}$ contains at least one vertex of degree larger than 2 and so either $\mathcal{G}_{\ell_x}$ is a parallel graph or it contains a parallel sub graph.

Lemma 4 *If neither s nor t are in $\mathcal{G}_{\ell_x}$, then transforming $\mathcal{G}_{\ell_x}$ into a DAG that contains paths from c_{x-1} to c_x equivalent to those paths in G from c_{x-1} to c_x is achieved by simply directing all edges in $\mathcal{G}_{\ell_x}$ away from c_{x-1} and towards c_x.*

Proof Assume that a simple path σ in G from c_{x-1} to c_x viewed as a directed path includes an edge (u, v) directed towards c_{x-1} (consider, for example, the edge (u, v) on the left side of Figure 13). Since σ is traversed from c_{x-1} to c_x the only way in which this edge (u, v) exists in σ is if σ traverses through a parallel subgraph G_i of

$\mathcal{G}_{\ell_x}$, reaches the terminal b_i of G_i closest to c_x and then continues to vertex u. But then note that to reach b_i the path σ must first traverse through the other terminal a_i of G_i; hence it would be impossible for σ to go from v to c_x without going again through one of the terminals of G_i. □

For the rest of this section we assume that one or both of s and t are in $\mathcal{G}_{\ell_x}$ (see Figure 14).

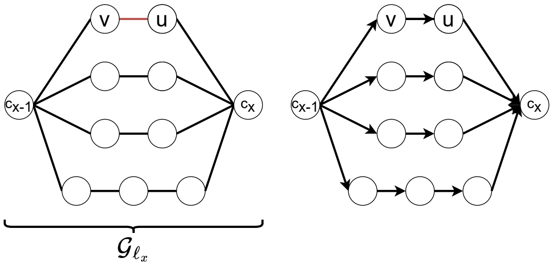


Fig. 13 A subgraph $\mathcal{G}_{\ell_x}$ that does not contain either s or t (left) and so the edges are directed from s to t (right).

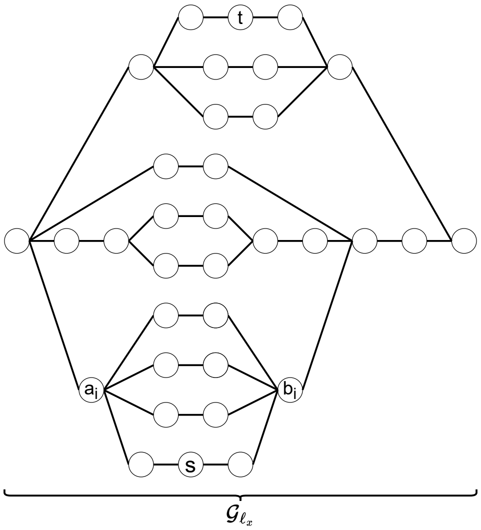


Fig. 14 A subgraph $\mathcal{G}_{\ell_x}$ containing several parallel subgraphs. Both s and t are contained in $\mathcal{G}_{\ell_x}$, so there are no other subgraphs $\mathcal{G}_{\ell_x}$ as there are no cut vertices in a shortest path from s to t.

A parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ with terminals a_i and b_i is a *maximal* parallel subgraph of $\mathcal{G}_{\ell_x}$ if there is no other parallel subgraph of $\mathcal{G}_{\ell_x}$ with terminals a_i and b_i that contains G_i. For example, in Figure 14 the parallel subgraph G_i with terminals a_i and b_i includes 11 vertices and 13 edges.

Let G_1, G_2, ..., G_p be the maximal parallel subgraphs of $\mathcal{G}_{\ell_x}$ indexed such that if G_j is a subgraph of G_i then $1 \leq j < i \leq p$. Note that G_j and G_i can have at most one terminal in common. To transform $\mathcal{G}_{\ell_x}$ into a DAG, we process the maximal parallel subgraphs G_1, ..., G_p in $\mathcal{G}_{\ell_x}$ in increasing order of index. If $\mathcal{G}_{\ell_x}$ contains both s and t, then any path σ from s to t in G must stay within $\mathcal{G}_{\ell_x}$ and so the rest of G can be deleted.

In the sequel we might refer to multiple different parallel subgraphs of $\mathcal{G}_{\ell_x}$ at the same time, so we describe here a consistent notation. The index i corresponds to the maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ that is currently being processed, the index j corresponds to a maximal parallel subgraph G_j contained within G_i, and the index k corresponds to a maximal parallel subgraph G_k that contains G_i. Hence, when processing G_i all the maximal parallel subgraphs with indices $j < i$ have already been processed and we have yet to process the maximal parallel subgraphs with indices $k > i$. Additionally, note that both s and t must be in at least one parallel subgraph because if, for example, s is a vertex of degree 1 or 2 in a graph $\mathcal{G}_{\ell_x}$ then, as described in Section 3.1, s would be renamed to a vertex of degree at least 3. Let G_s and G_t be the smallest indexed maximal parallel subgraphs that contain s and t, respectively.

To simplify the description of our algorithm we first modify $\mathcal{G}_{\ell_x}$ as follows. If vertex a is a common terminal of two or more maximal parallel subgraphs G_{a_1}, G_{a_2}, ..., G_{a_r}, then we modify $\mathcal{G}_{\ell_x}$ by replacing a with r vertices a_1, a_2, ..., a_r. Vertex a_g, for all $g = 1, 2, ..., r$, is adjacent to all neighbours u of a in G_{a_g} that are not in any other maximal subgraph G_{a_h}, $h \neq g$; the length of edge (a_g, u) is the same as the length of (a, u). Each one of these vertices a_i has the same set of items as a and vertices a_h and a_{h+1} are connected with an edge (a_h, a_{h+1}) of length 0 for all $h = 1, 2, ..., r - 1$. Note that this transformation creates some paths in which the same item appears multiple times. We will fix this later by merging all vertices a_h back into a single vertex a. We say that terminal vertices a_1, a_2, ..., a_r are *entangled.*

Important Note. We always assume that for each maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ the terminal a_i is on the left and terminal b_i is on the right, as shown in the figures.

Below we explain how to transform $\mathcal{G}_{\ell_x}$ into a DAG using three steps.

3.3.1 Step 1: Edges Close to s and t

Consider the maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ with terminals a_i and b_i. If $G_i = G_s$, then for each simple path p from s to a_i that does not route through either t or b_i, the edges of p are directed away from s and towards a_i. Similarly, for each simple path p from s to b_i that does not route through either t or a_i, the edges of p are directed away from s and towards b_i (see Figure 15).

If $G_i = G_t$, then direct to t the edges in each simple path p from a_i to t and from b_i to t that does not route through either s, a_i, or b_i (see Figure 15).

Finally, if $G_i = G_s = G_t$, then for each simple path p from s to t that does not route through either a_i or b_i, the edges in p are directed away from s and towards t (see Figure 15).

Note that for some of the maximal parallel subgraphs G_i of $\mathcal{G}_{\ell_x}$, Step 1 might not direct any edges.

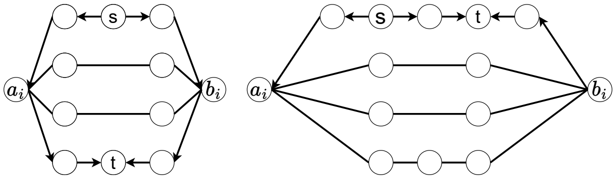


Fig. 15 Edges in the lowest indexed maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ containing s or t are directed away from s and towards t.

3.3.2 Step 2: Transforming a Parallel Subgraph G_i of $\mathcal{G}_{\ell_x}$ into Several DAGs

We transform the maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ with terminals a_i and b_i into several DAGs, as explained below. Note that if s or t is a terminal of G_i then all the edges of G_i have been directed already in Step 1 and hence we do not need to perform Step 2; therefore, for the remainder of this section we assume that neither s nor t is a terminal of G_i.

(a) Splitting G_i Let $\hat{G}_i = (\hat{V}_i, \hat{E}_i, a_i, b_i)$ be the subgraph of G_i that includes the terminals a_i and b_i of G_i, all vertices of G_i that do not belong to any maximal parallel subgraph G_j of G_i, and all undirected edges between these vertices. For example, in the left side of Figure 16 $\hat{G}_i$ does not include the simple paths from s to the terminals or from the terminals to t, as these edges were directed during Step 1.

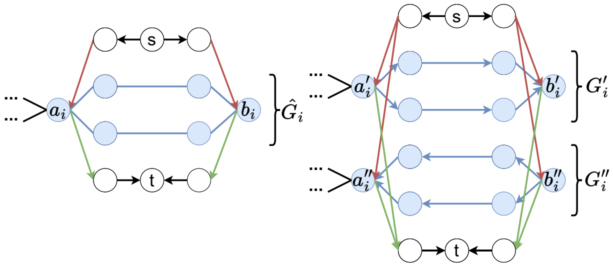


Fig. 16 (Left) The maximal parallel subgraph G_i is shown before being processed by Step 2; $\hat{G}_i$ consists of a single connected parallel subgraph (shown in blue). (Right) After G_i has been processed by Step 2 the simple paths between a'_i and b'_i are directed from a'_i to b'_i and the simple paths between b''_i and a''_i are directed from b''_i to a''_i. Any edges incident on the original terminals a_i and b_i before processing G_i are now incident on the corresponding terminals of G'_i and G''_i.

The subgraph $\hat{G}_i$ consists of one or more connected components. Multiple components might be present if G_i contains multiple maximal parallel subgraphs; since each maximal parallel subgraph G_j of G_i must have been processed before G_i, $\hat{G}_i$ does not include the directed edges of the DAGs created for the maximal subgraphs G_j. Let the connected components of $\hat{G}_i$ be $\hat{G}_{i_1}$, $\hat{G}_{i_2}$, ..., $\hat{G}_{i_q}$ (see Figure 17).

We create two copies of each $\hat{G}_{i_w}$: G'_{i_w} and G''_{i_w}. Each of the edges in these copies has the same length as the corresponding edge in $\hat{G}_{i_w}$, and the two copies u' and u'' of a vertex u in $\hat{G}_{i_w}$, store the same items as u. We call u'

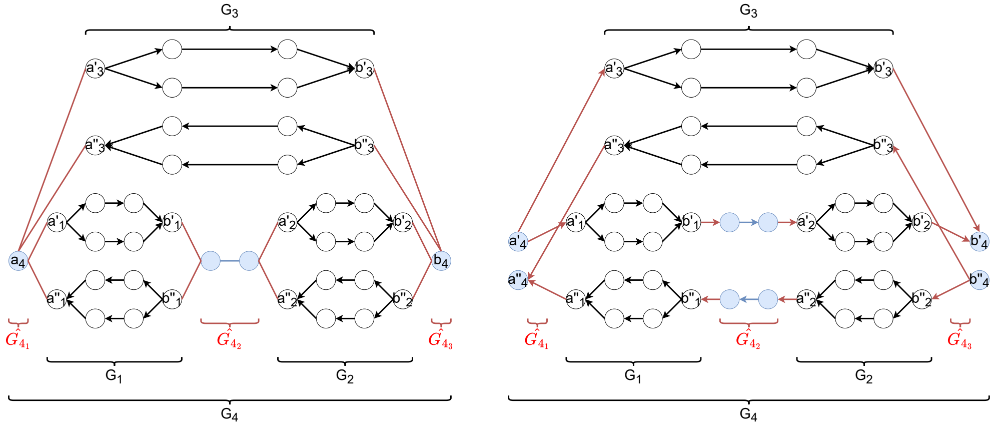


Fig. 17 (Left) The maximal parallel subgraph $G_i = G_4$ is shown before being processed by Step 2 and since the maximal parallel subgraphs G_1, G_2, and G_3 are each contained within G_4 (so each is transformed into DAGs before G_4 is processed) then $\hat{G}_i$ consists of multiple components. The vertices and edges of $\hat{G}_{4_1}$, $\hat{G}_{4_2}$, and $\hat{G}_{4_3}$ are shown in blue. (Right) After G_i has been processed by Step 2, the components of $\hat{G}_i$ have been duplicated and their incident edges have been directed.

and u'' duplicate vertices. Note that this step duplicates the terminals a_i and b_i of G_i.

For the remainder of the description of the algorithm, we use the following notation. A vertex v' marked with the prime symbol ($'$) represents the copy of a vertex v that belongs to the first copy G'_ℓ of a subgraph $\hat{G}_\ell$ and a vertex v'' marked with the double prime symbol ($''$) represents the copy of vertex v that belongs to the second copy G''_ℓ of $\hat{G}_\ell$.

For each undirected edge (u, v) where u is in $\hat{G}_{i_w}$ and v is in G_i but not in $\hat{G}_{i_w}$ (see the red edges in Figure 17), we delete (u, v) from G_i and proceed as follows:

- If v is not a duplicate vertex (this occurs when s or t is a terminal of a maximal parallel subgraph G_j contained within G_i) then add the undirected edge (u', v) to G'_{i_w} and add the undirected edge (u'', v) to G''_{i_w}.
- If v is a duplicate vertex, then after removing both edges (u, v') and (u, v'') add the undirected edge (u', v') to G'_{i_w} and add the undirected edge (u'', v'') to G''_{i_w}.

(b) Processing the terminals of G_i. For each vertex v contained in G_i but not in $\hat{G}_i$, that is adjacent to a_i or b_i, we proceed as follows:

- For each directed edge (a_i, v) of G_i (remember that some edges of G_i were assigned a direction in Step 1) delete (a_i, v) and add directed edges (a'_i, v) to G'_{i_1} and (a''_i, v) to G''_{i_1}.
- For each directed edge (v, a_i) of G_i delete (v, a_i) and add directed edges (v, a'_i) to G'_{i_1} and (v, a''_i) to G''_{i_1}.
- For each directed edge (b_i, v) of G_i delete (b_i, v) and add directed edges (b'_i, v) to G'_{i_q} and (b''_i, v) to G''_{i_q}.

- For each directed edge (v, b_i) of G_i delete (v, b_i) and add directed edges (v, b'_i) to G'_{i_q} and (v, b''_i) to G''_{i_q}.

For example, in Figure 16 (left) the red and green directed edges are incident on a_i and b_i but in Figure 16 (right) the red and green directed edges (which have been duplicated) are incident on a'_i, a''_i, b'_i, and b''_i.

We refer to the set of subgraphs G'_{i_1}, G'_{i_2}, ..., G'_{i_q} simply as G'_i, and all the subgraphs G''_{i_1}, G''_{i_2}, ..., G''_{i_q} as G''_i.

(c) Transforming G'_i and G''_i into DAGs. If G_i contains both s and t and they are connected by an undirected edge (note that this can only happen if s is a terminal of a maximal parallel subgraph contained within G_i and t is a terminal of a different maximal parallel subgraph contained within G_i, as otherwise the edge would have already been directed in Step 1) then this edge is directed towards t. Then, we transform each G'_i and G''_i into DAGs by directing every undirected edge in G'_i from left to right and every undirected edge in G''_i from right to left (see Figure 16).

(d) Connecting G'_i and G''_i to $\mathcal{G}_{\ell_x}$. The above steps (a) - (c) allow paths to traverse from the left terminal of G'_i to its right terminal and from the right terminal of G''_i to its left terminal.

After the edges of G'_i and G''_i have been directed, for any undirected edges (u, v) where u is a terminal of G_i and v is not in G_i edges need to be added to the directed graphs G'_i and G''_i to connect them to the rest of $\mathcal{G}_{\ell_x}$. These edges will allow paths to traverse between the newly created DAGs G'_i and G''_i and any adjacent maximal parallel subgraphs and/or any maximal parallel subgraph G_k containing G_i.

Note that if G_k contains G_i, it might also contain other maximal parallel subgraphs that were already transformed into DAGs, so some of the edges that we will add to G'_i and G''_i to connect them to G_k will be incident on vertices that have already been processed and some others will be incident on vertices that have not been processed yet (see Figure 18).

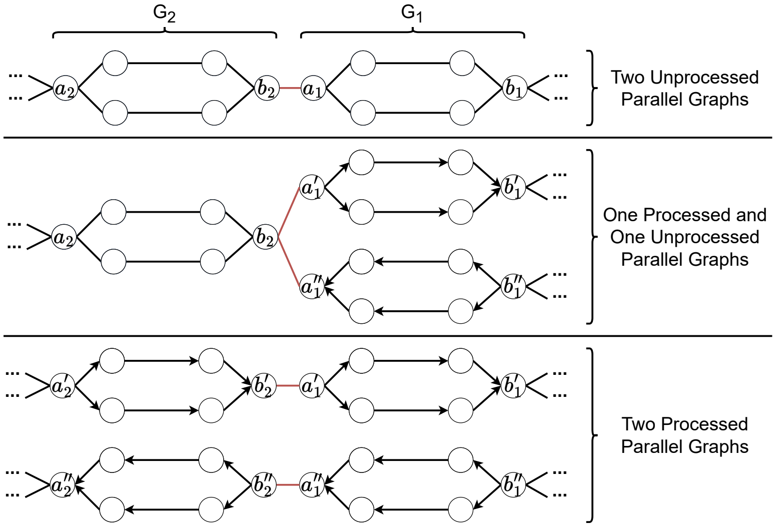
Observation 1 Each terminal a_i and b_i of a maximal parallel subgraph G_i of $\mathcal{G}_{\ell_x}$ has at most one neighbor v that is contained in $\mathcal{G}_{\ell_x}$ but not contained in G_i.

Observation 1 follows from the definitions of series and parallel compositions and from the creation of entangled vertices for shared terminals.

We add undirected edges to G'_i and G''_i to connect them to $\mathcal{G}_{\ell_x}$ as follows:

- For all undirected edges (a_i, v') and (a_i, v'') where v' and v'' are not in G_i, delete (a_i, v') and (a_i, v'') and add undirected edges (a'_i, v') to G'_i and (a''_i, v'') to G''_i (see the bottom image in Figure 18). Note that the vertices v' and v'' correspond to duplicates of a vertex that has already been processed.
- For each undirected edge (v, a_i) where v has not yet been processed and v is not in G_i, add undirected edges (a'_i, v) and (a''_i, v) to G'_i and G''_i, respectively (see the middle image in Figure 18).

- The same two steps above are repeated for each vertex v adjacent to b_i that is not in G_i, and the new edges are incident to b_i' and b_i''. Each $\hat{G}_{i_w}$ is then deleted, so that the terminals a_i and b_i and all undirected edges of $\hat{G}_i$ are removed. Note that the duplicate copies of each of these vertices and edges still exist in G_i' and G_i''.



G2
G1
a2
b2
a1
b1
Two Unprocessed Parallel Graphs
a'1
b'1
a''1
b''1
One Processed and One Unprocessed Parallel Graphs
a'2
b'2
a''2
b''2
Two Processed Parallel Graphs

Fig. 18 (Top) Two adjacent maximal parallel undirected subgraphs G_1 and G_2 were originally connected by a single edge between terminals b_2 and a_1, shown in red. (Middle) Since the maximal parallel subgraphs are processed in increasing order of index, G_1 was transformed into two DAGs first, and the edge between b_2 and a_1 transformed into the edges between b_2 and a_1' and a_1''. (Bottom) After G_2 has been transformed into two DAGs, there is a single edge between b_2' and a_1' and a single edge between b_2'' and a_1''.

3.3.3 Step 3: Edges Connecting Nested Parallel Graphs Containing s or t

When G_i contains s and/or t, we need to change some of the edges in the above DAGs created for G_i; however, the specific edges that need to be modified depend on where in G_i vertices s and t are located.

Recall that since G_i is a maximal parallel subgraph, it was created from the parallel composition of a group of series-parallel subgraphs G_{i_1}, G_{i_2}, ..., G_{i_m}. Let the *parallel components* pc_{i1}, pc_{i2}, ..., pc_{im} of G_i be these series-parallel subgraphs, excluding the terminals of G_i (see Figure 19). Note that a parallel component of G_i might contain multiple maximal parallel subgraphs.

If a parallel component pc_{ih} either contains no maximal parallel subgraphs or it does not contain s or t, then the edges in the DAGs for pc_{ih} have already been correctly directed in Steps 1 and 2. Therefore, we only consider parallel components pc_{ih} of G_i that contain at least one maximal parallel subgraph and s and/or t. Each maximal parallel subgraph G_j in a parallel component pc_{ih} of G_i was previously processed, and so the two terminals a_j and b_j of G_j

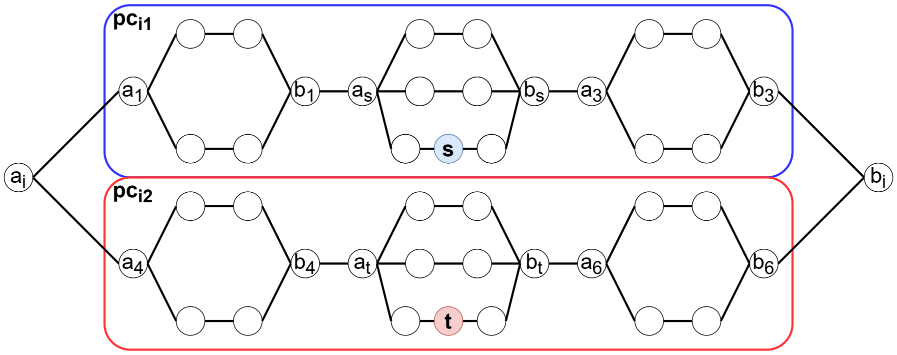


Fig. 19 This maximal parallel subgraph G_i contains two parallel components pc_{i1} (blue) and pc_{i2} (red), each of which contain three maximal parallel subgraphs. Note that the parallel component pc_{i1} contains s and the parallel component pc_{i2} contains t.

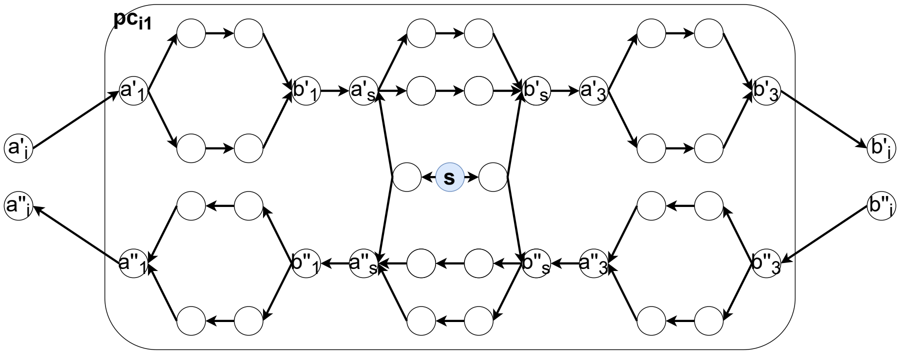


Fig. 20 The parallel component pc_{i1} from Figure 19 after processing Step 2 but before processing Step 3.

were transformed into four terminals a'_j, a''_j, b'_j, and b''_j of two DAGs G'_j and G''_j (see Figure 20).

Intuitively, the algorithm that we describe below modifies the edges of the DAGs corresponding to a parallel component pc_{ih} of G_i containing s or t such that (i) when a maximal parallel subgraph G_j of pc_{ih} contains s but not t, paths in the DAGs corresponding to G_i are directed outwards away from G_j; (ii) when a maximal parallel subgraph G_j of pc_{ih} contains t but not s, paths in the DAGs corresponding to G_i are directed inwards towards G_j; and (iii) when a maximal parallel subgraph G_j of pc_{ih} contains both s and t, paths in the DAGs corresponding to G_i can travel away from s through G_j and possibly through several maximal parallel subgraphs containing G_j in increasing order of index, then traverse exactly one parallel component of one of these maximal parallel subgraphs, and finally return to t. Note that in a series-parallel graph a simple path cannot start at s, travel through maximal parallel subgraphs in increasing order of index, then in decreasing order of index, and then again in increasing order as such a path would have to traverse twice through some terminal of a maximal parallel subgraph.

If a parallel component pc_{ih} of G_i contains s or t, we modify the DAGs corresponding to the parallel component pc_{ih} as described below.

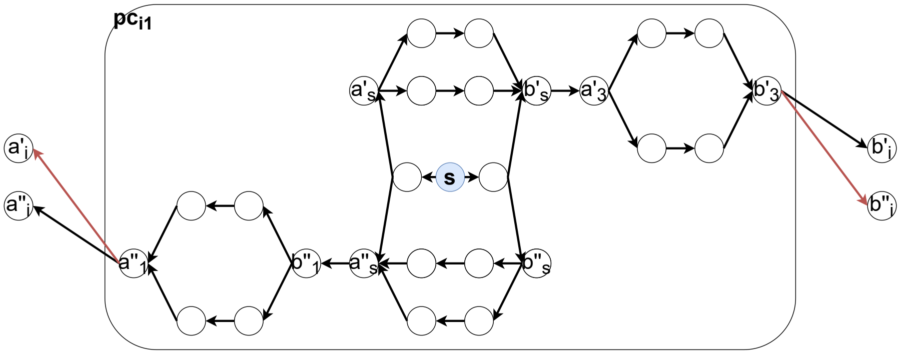


Fig. 21 The parallel component pc_{i1} of Figure 19 contains s but not t and so some of the DAGs for each G_j of pc_{i1} have been deleted and some of the edges (highlighted in red) have been added.

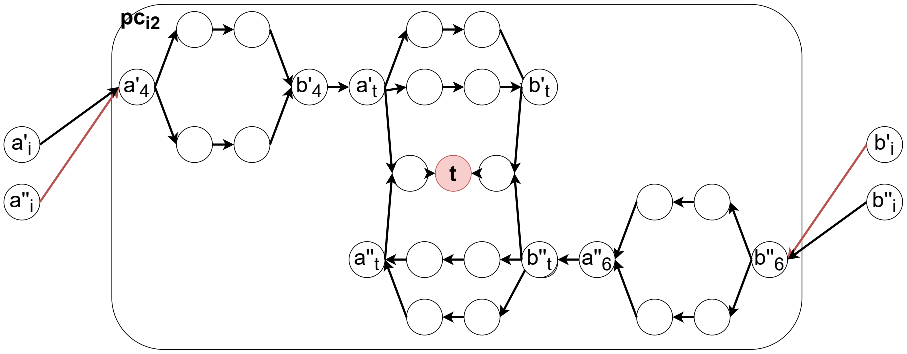


Fig. 22 The parallel component pc_{i2} of Figure 19 contains t but not s and so some of the edges for each G'_j of pc_{i2} and G''_j of pc_{i2} (highlighted in red) have been added.

(a) No maximal parallel subgraph contains s and t. If pc_{ih} contains at least one of s and t, but no maximal parallel subgraph G_j of pc_{ih} contains both s and t, then:

- If a simple path σ_1 exists between the terminal a_i of G_i and a terminal of G_s that does not traverse through G_t or b_i, we add a directed edge (u''_a, a'_i) (see the left maximal parallel subgraphs in Figures 21 and 23), where u_a is the single vertex of pc_{ih} adjacent to a_i (see Observation 1). Then, we delete G'_j for each maximal parallel subgraph G_j of pc_{ih} reachable by σ_1 such that $G_j \neq G_s$ and G_s is not contained within G_j. These steps allow paths to traverse from s to the left terminal of G_i and then onto the right terminal of G_i without creating cycles.
  Similarly, if a simple path σ_2 exists between b_i and G_s that does not traverse through G_t or a_i, we add a directed edge (u'_b, b''_i) (see the right maximal parallel subgraph in Figure 21), where u_b is the vertex of pc_{ih} adjacent to b_i. Then, we delete G''_j for each maximal parallel subgraph G_j of pc_{ih} reachable by σ_2 such that $G_j \neq G_s$ and G_s is not contained within G_j. These steps allow paths to traverse from s to the right terminal of G_i and then onto the left terminal of G_i without creating cycles.

- If a simple path σ_1 exists between a_i and a terminal of G_t that does not traverse through G_s or b_i, we add a directed edge (a''_i, u'_a) (see the left maximal parallel subgraph in Figure 22), where u_a is the vertex of pc_{ih} adjacent to a_i. Then, we delete G''_j for each maximal parallel subgraph G_j of pc_{ih} reachable by σ_1 such that $G_j \neq G_t$ and G_t is not contained within G_j. If a simple path σ_2 exists between b_i and G_t that does not traverse through G_s or a_i, we add a directed edge (b'_i, u''_b) (see the right parallel subgraphs in Figures 22 and 23), where u_b is the vertex of pc_{ih} adjacent to b_i, and we delete G'_j for each maximal parallel subgraph G_j of pc_{ih} reachable by σ_2 such that $G_j \neq G_t$ and G_t is not contained within G_j. These two changes allow paths to traverse both terminals of G_i before reaching t.
- We delete G''_j for each maximal parallel subgraph G_j contained within pc_{ih} such that $G_j \neq G_s$, $G_j \neq G_t$, G_s and G_t are not contained within G_j, and G_j is reachable from a simple path between G_s and G_t that does not traverse through either a_i or b_i. This step simplifies the DAGs as we are not interested in paths from t to s.

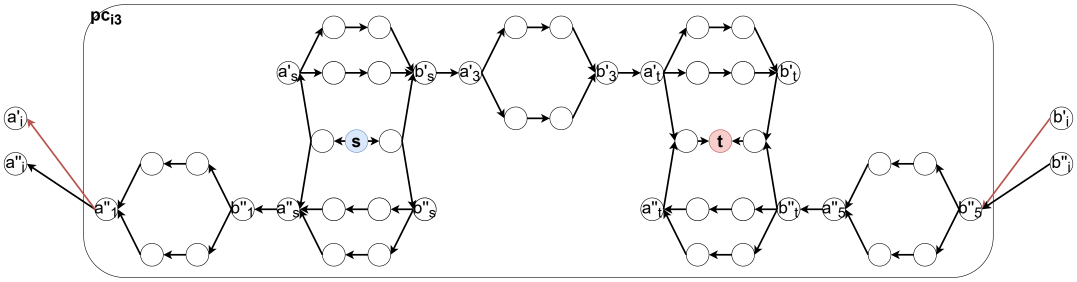


Fig. 23 A parallel component pc_{i3} containing both s and t, but not in the same maximal parallel subgraph G_j. The edges highlighted in red have been added.

(b) A maximal parallel subgraph contains s and t. If pc_{ih} contains both s and t within the same maximal parallel subgraph G_j, then we need to change a few edges incident on the terminals of G_i (see Figure 24):

- For each directed edge (u'', a''_i), where u'' is in G''_i but not in pc_{ih}, delete the edge and add the directed edge (a''_i, u'), where u' is the other duplicate of u''. This allows paths to traverse from s towards the left terminal a''_i of G''_i to then go towards the right through a different parallel component pc'_{im}.
- For each directed edge (b''_i, u''), where u'' is in G''_i but not in pc_{ih}, delete the edge and add the directed edge (u', b''_i), where u' is the other duplicate of u''. These last two changes allow paths from s to a''_i to then traverse through another parallel component pc'_{im} to reach b''_i and then return to t. Note that a path σ' from s to t traveling through b''_i must have already traveled through a''_i, so the edges (u'', a''_i) and (b''_i, u'') cannot be in σ'.
- For each directed edge (u', b'_i), where u' is in G'_i but not in pc_{ih}, delete the edge and add the directed edge (b'_i, u''). This allows paths that traverse from

s towards the right terminal b'_i of G'_i to then go towards the left through a different parallel component pc''_{im}.

- For each directed edge (a'_i, u'), where u' is in G'_i but not in pc_{ih}, delete the edge and add the directed edge (u'', a'_i). These last two changes allow paths from s to b'_i to then traverse left to reach a'_i and then return to t. Note that a path σ' from s to t traveling through a'_i must have already traveled through b'_i, so the edges (u', b'_i) and (a'_i, u') cannot be in σ'.
- Finally, if G_i is the lowest indexed maximal parallel subgraph that contains G_s, $G_s = G_t$, and neither s nor t is a terminal of G_s, then we delete the edges (u', a'_s), (a''_s, u''), (b'_s, u'), and (u'', b''_s), where u' is in G'_i and u'' is in G''_i but neither are in G_s. Then, add the directed edges (a'_s, u''), (u', a''_s), (u'', b'_s), and (b''_s, u'); these edges allow paths from s that traverse both terminals of G'_i or G''_i to reach t (see Figure 24).
  These changes are done because a path σ from s to t in G entering G_s through b_s must have traveled through a_s and must then travel to t, and so the edges (a''_s, u'') and (u'', b''_s) cannot be in the DAGs as they create cycles. To see this, note that if the edges (a''_s, u'') and (u'', b''_s) existed then a path σ' that exits G''_s from a''_s and traverses through some parallel component pc''_{ih} to a''_i and then traverses to the right to b''_i would then continue towards b''_s, which then can reach a''_s, creating a cycle. Similarly, (u', a'_s) and (b'_s, u') cannot be in the DAGs.

After a maximal parallel subgraph G_i has been transformed into the DAGs G'_i and G''_i, the remaining undirected edges from G_i and any isolated vertices are deleted. Additionally, for any vertex for which all its entangled vertices are contained within G_i, these vertices are merged into a single vertex.

If a_s is an entangled vertex with the right terminal b_h of a maximal parallel subgraph G_h that appears to the left of G_s, then because of the above changes when entangled vertices are merged, a'_s will be merged with b''_h (instead of b'_h) and a''_s will be merged with b'_h (see the bottom-right of Figure 25). Similarly, if b_s is an entangled vertex with the left terminal a_h of the maximal parallel subgraph G_h to the right of G_s, then b'_s will be merged with a''_h and b''_s will be merged with a'_h.

Note that the two DAGs G'_i and G''_i might each contain a vertex corresponding to every vertex of G_i. Additionally, G'_i and G''_i might each contain a directed edge corresponding to every undirected edge of G_i. Therefore, G'_i and G''_i have at most twice as many vertices and edges as G_i.

4 Algorithm Analysis

Algorithm 2 transforms an undirected series-parallel graph $G = (V, E)$ into a DAG $\mathcal{G}$.

We say that a directed path σ *enters* a maximal parallel subgraph $G_i = (V_i, E_i, a_i, b_i)$ when σ includes an edge where one endpoint is either a_i or b_i, the other endpoint u is in G_i, and the edge is directed towards u. Similarly, we say that a directed path σ *exits* G_i when it includes an edge where one endpoint

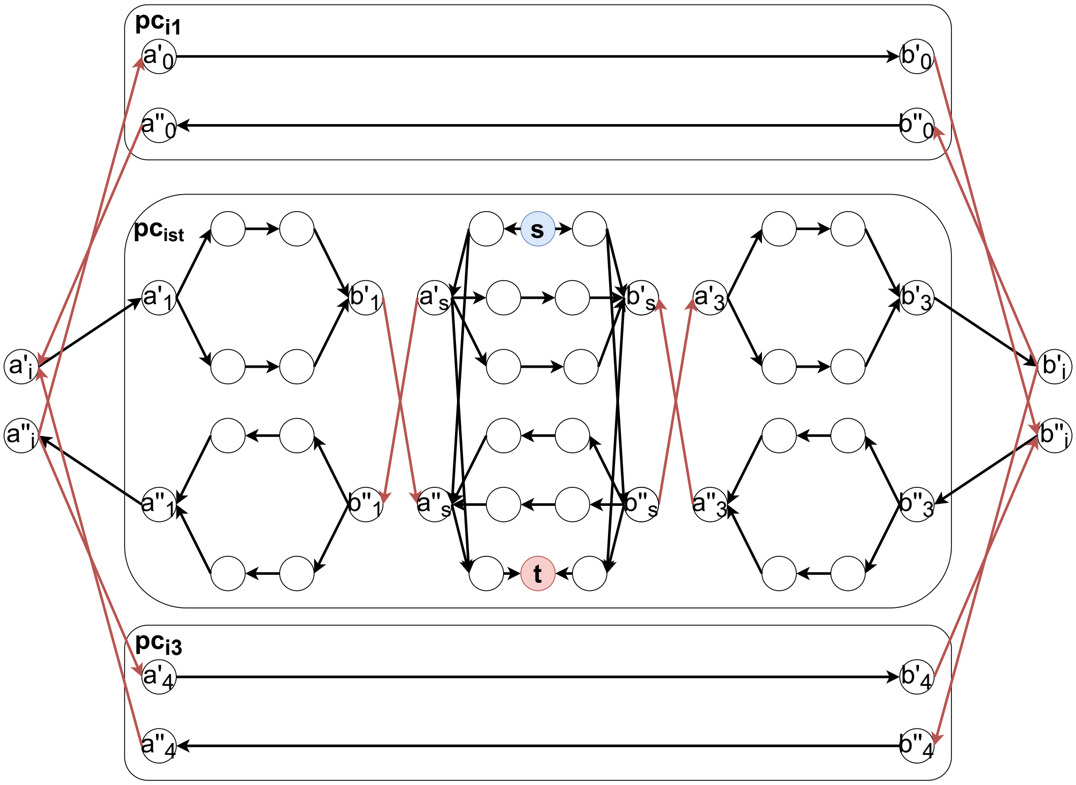


Fig. 24 A parallel component pc_{ist} of a maximal parallel subgraph G_i contains DAGs for each maximal parallel subgraph G_j contained within G_i. Since pc_{ist} contains both s and t within a maximal parallel subgraph G_j of G_i, the edges highlighted in red have been added.

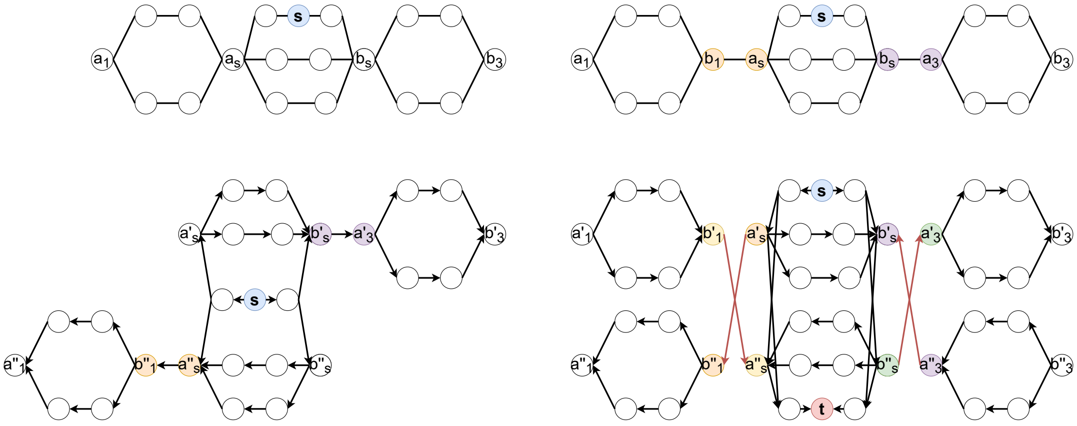


Fig. 25 Parallel component pc_{i1} from Figure 24 is shown as if the terminals of G_s were entangled with the adjacent parallel subgraphs. (Top-left) The parallel component is shown is it appears in the undirected graph G. (Top-right) Terminal a_s and its entangled vertex are highlighted in orange and terminal b_s and its entangled vertex are highlighted in purple. (Bottom-left) Some of the adjacent parallel subgraphs adjacent to G_s are deleted and hence it is clear which entangled vertices to merge. (Bottom-right) If $G_s = G_t$, then after changing the edges during Step 3 the entangled pairs need to be adjusted so as to not form a cycle. The pairs of vertices of the same colors will be merged together.

is either a_i or b_i, the other endpoint u is not in G_i, and the edge is directed to u. Finally, we say that a path σ *traverses* through G_i if σ both enters and

Algorithm 2 SeriesParallelToDAG($G = (V, E)$, s, t, I)

```
Input: Series-Parallel graph $G$, start vertex $s$, end vertex $t$, and item assignments $I$.
Output: A DAG $\mathcal{G}$ containing paths from $s$ to $t$ equivalent to the paths from $s$ to $t$ in $G$.
Process vertices with degree 1 as described at the beginning of Section 3.
Let $p$ be a shortest path from $s$ to $t$ and let $c_1$, $c_2$, ..., $c_{k-1}$ be its cut vertices.
for each $x = 1$ to $k$ do
    Build $\mathcal{G}_{\ell_x}$ as described in Section 3.2 and index the maximal parallel subgraphs $G_1$, ..., $G_p$ of $\mathcal{G}_{\ell_x}$ such that if $G_j$ is a subgraph of $G_i$, then $j < i$.
    for each $i = 1$ to $p$ do
        Transform $G_i$ into a directed graph as described in Section 3.3.
Output the DAG $\mathcal{G}$ created by joining the DAGs for the graphs $\mathcal{G}_{\ell_x}$.
```

exits G_i. We extend the notation of entering, exiting, and traversing maximal parallel subgraphs to parallel components as well.

Algorithm 2 transforms an undirected series-parallel graph G into a directed graph $\mathcal{G}$ by adding duplicate vertices and edges of existing vertices and edges in G and then directing all the edges. Note that two vertices in $\mathcal{G}$ are only adjacent if the corresponding vertices in G are also adjacent.

For each maximal parallel subgraph G_i of G Algorithm 2 creates two directed graphs G'_i and G''_i, where a'_i and b'_i are the terminals of G'_i and a''_i and b''_i are the terminals of G''_i. Recall that we assume that when drawing G'_i and G''_i the terminals a'_i and a''_i are always drawn on the left and the terminals b'_i and b''_i are drawn on the right. We use pc_{ih} to refer to the h^{th} parallel component in G_i, and we use pc'_{ih} and pc''_{ih} to refer to the corresponding directed subgraphs created for pc_{ih} by the algorithm.

Lemma 5 *Algorithm 2 transforms an instance of ThOP on an undirected series-parallel graph G into an instance of ThOP on a DAG $\mathcal{G}$, so all paths from s to t in $\mathcal{G}$ are simple.*

Proof To prove that there are no cycles in $\mathcal{G}$, we note that since all directed edges in $\mathcal{G}$ correspond to undirected edges in G, a cycle in $\mathcal{G}$ would be either (1) two anti-parallel directed edges in $\mathcal{G}$ corresponding to an undirected edge (u, v) in G or (2) a directed cycle in $\mathcal{G}$ corresponding to a cycle in G.

(1) For each undirected edge (u, v) in G there are no corresponding anti-parallel directed edges in $\mathcal{G}$: When Algorithm 2 creates two copies (u', v') and (u'', v'') of an undirected edge (u, v) of G, these edges are incident on different copies of u and v, thus they do not form a cycle.

(2) For each simple cycle C in G, we show that the corresponding directed edges in $\mathcal{G}$ do not form a cycle. Consider a simple cycle C in G. Observe that C must be fully contained within some parallel subgraph. Let G_i be the lowest indexed maximal

parallel subgraph that contains C. Note that C must include both terminals a_i and b_i of G_i and so C contains a path that traverses a parallel component pc_{ih} of G_i from a_i to b_i and a path that traverses a different parallel component pc_{im} of G_i from b_i to a_i.

To show that the directed edges in $\mathcal{G}$ corresponding to C do not form a cycle, we must consider five cases depending on whether the parallel subgraph G_i contains s, t, or neither:

- If G_i does not contain s or t, then either (i) the algorithm in Lemma 4 directs the edges in $\mathcal{G}$ corresponding to the edges in G_i away from c_{x-1} and toward c_x and this does not form cycles, or (ii) the algorithm in Section 3.3.2 directs the edges in G'_i and G''_i corresponding to G_i; in this case, there is a path σ' in $\mathcal{G}$ from a'_i to b'_i but since Algorithm 2 does not add any incident edges to b'_i directed towards vertices inside G'_i or G''_i then σ' cannot be extended to traverse back to a'_i to create a cycle (see Figure 18 (bottom)). Similarly, a path σ' in $\mathcal{G}$ from b''_i to a''_i cannot be extended to create a cycle.
- If G_i contains s but not t, then by statement (ii) in the preceding argument there is a path σ' in $\mathcal{G}$ from a'_i to b'_i, but there are no incident edges to b'_i directed towards vertices inside G'_i or G''_i (see Figure 21). Similarly, a path σ' in $\mathcal{G}$ from b''_i to a''_i cannot be extended to create a cycle.
- If G_i contains t but not s, then by the same statement (ii) above there is a path σ' in $\mathcal{G}$ from a'_i to b'_i but the algorithm in Section 3.3.3(a) ensures that the only incident edges to b'_i directed towards G'_i or G''_i lead only to t, and so σ' cannot be extended to create a cycle (see Figure 22). Similarly, a path σ' in $\mathcal{G}$ from b''_i to a''_i cannot be extended to create a cycle.
- If G_i contains both s and t but no maximal parallel subgraph G_j contained within G_i contains both s and t, then the above arguments show that there is path σ' in $\mathcal{G}$ from a'_i to b'_i (or from b''_i to a''_i) but σ' cannot be extended to create a cycle (see Figure 23).
- If G_i contains both s and t in the same maximal parallel subgraph G_j contained within G_i, then there exists no path σ' in $\mathcal{G}$ from a'_i to b'_i, as all paths from a'_i are directed towards t. To see this, note that the algorithms in Section 3.3.3(b) add only one outgoing directed edge in $\mathcal{G}$ incident to a'_i and it leads to a''_s, whose only outgoing directed edge leads to t (see Figure 24). However, there is a path σ' from a''_i to b''_i because the algorithms in Section 3.3.3(b) add edges from a''_i towards each of the other parallel components of G_i that lead to b''_i (see Figure 24). If there is also a path σ'' in $\mathcal{G}$ from b''_i that enters G''_i then σ'' leads to t, but since in this case the algorithms in Section 3.3.1 and 3.3.2(b) does not add any incident edges to t directed towards a''_i then σ'' cannot be extended to create a cycle.
  Similarly, a path σ' in $\mathcal{G}$ from b'_i to a'_i corresponding to a subpath in C from b_i to a_i cannot be extended to create a cycle.

□

Lemma 6 *Algorithm 2 transforms an instance of ThOP on an undirected series-parallel graph G into an instance of ThOP on a DAG $\mathcal{G}$ such that for every simple* *path σ from s to t in G, there is an equivalent (see Definition 2) directed path σ' from s to t in $\mathcal{G}$.*

Proof Consider a simple path σ from s to t in G and let t_1, t_2, ..., t_d be the terminals of maximal parallel subgraphs of G included in σ in the order in which they appear in a traversal of σ from s to t. These terminals split σ into a set of subpaths called *segments*: $<t_0, t_1>$, $<t_1, t_2>$, ..., $<t_d, t_{d+1}>$, where $t_0 = s$ and $t_{d+1} = t$. We gradually construct a directed path σ' of $\mathcal{G}$ equivalent to σ by considering one by one these segments. Throughout this construction we maintain the invariant that for the subpath of σ from s to any t_i there is an equivalent directed path in $\mathcal{G}$ from s to a vertex corresponding to t_i that can be extended to include a directed subpath equivalent to the next segment $<t_i, t_{i+1}>$.

Recall that after partitioning G into the subgraphs $\mathcal{G}_{\ell_x}$, if a subgraph $\mathcal{G}_{\ell_x}$ contains no maximal parallel subgraphs or if neither s nor t are in $\mathcal{G}_{\ell_x}$ then the edges in $\mathcal{G}_{\ell_x}$ are directed from its left terminal to its right terminal, and so all segments $<t_i, t_{i+1}>$ belonging to one of these subgraphs $\mathcal{G}_{\ell_x}$ can be contracted to a single segment; moreover, adjacent contracted segments can be further contracted to a single segment. Hence for the remainder of this proof, we consider only subgraphs $\mathcal{G}_{\ell_x}$ that have at least one maximal parallel subgraph that contains s and/or t.

If terminal $t_1 \neq t$ is in G_s but is not a terminal of G_s, then observe that G_s contains within it at least one maximal parallel subgraph G_j, t_1 is either the terminal a_j or b_j, and G_j lies on a simple path between the terminals of G_s. The edges in G_j are directed away from s by the algorithm in Section 3.3.1 and hence G_j would not have been duplicated by the algorithm in Section 3.3.2. In this case we rename t_1 so that it is either a terminal of G_s or t, whichever occurs first in σ. Similarly, if $t_d \neq s$ is in G_t but is not a terminal of G_t, we rename t_d so that it is either a terminal of G_t or s, whichever occurs first in σ. The remaining terminals are renamed so that t_2 is the next terminal included in σ after t_1 in the order of a traversal of σ from s to t. Hence for the remainder of this proof, if $t_1 \neq t$ we assume it is a terminal of G_s, and if $t_d \neq s$ we assume it is a terminal of G_t.

Recall that the algorithm in Section 3.3.2(a) constructs the DAGs G'_j and G''_j corresponding to a parallel subgraph G_j by creating two copies of G_j; therefore, there can be two duplicate vertices in $\mathcal{G}$ that correspond to each terminal t_i of σ. Hence, when selecting a directed path in $\mathcal{G}$ equivalent to a segment $<t_{i+1}, t_{i+2}>$ we must carefully consider which of the two vertices in $\mathcal{G}$ corresponding to each terminal needs to be chosen.

We use a proof by induction on the terminals t_i that the invariant holds for all subpaths of σ from s to t_i.

We first show that the invariant holds for a simple path from s to t in G that is contained in G_s and for the first segment $<s, t_1>$ of σ.

1. If the simple path σ from s to t in G is contained within G_s, then the algorithm in Section 3.3.1 adds directed edges to $\mathcal{G}$ from s to vertices corresponding to the terminals of G_s and from s to t; the algorithms in Section 3.3.2(a)-(c) create two DAGs G'_s and G''_s containing paths between the vertices corresponding to terminals of G_s and from these vertices to t, ensuring that regardless of whether σ traverses through zero, one, or two terminals of G_s in $\mathcal{G}$ there is a path σ' equivalent to σ (see Figure 16).

2. If the first segment of $\mathcal{G}$ is $<s, t_1>$, where s is not a terminal of G_s, then t_1 must be a terminal of G_s; therefore, the algorithms in Sections 3.3.1 and 3.3.2(a)-(c) create paths in $\mathcal{G}$ from s to t'_1 and t''_1 (the duplicates of t_1 created for the DAGs G'_s and G''_s), and each of these paths is equivalent to $<s, t_1>$. Since only one of these two directed paths can be extended to an equivalent path for the next segment $<t_1, t_2>$, we show how to select the correct path so that the invariant holds, by considering four cases: either (a) t_1 is to the right of s and t_2 is to the right of t_1, (b) t_1 is to the right of s and t_2 is to the left of t_1, (c) t_1 is to the left of s and t_2 is to the left of t_1, and (d) t_1 is to the left of s and t_2 is to the right of t_1.

   Given a path σ from s to t, we determine whether a terminal t_i is to the left or to the right of terminal t_{i+1} as follows:

   - If t_i and t_{i+1} are in the same maximal parallel subgraph, let G_h be the smallest maximal parallel subgraph containing t_i and t_{i+1}.
     - If t_i and t_{i+1} are terminals of G_h then the terminal corresponding to a_h is to the left of the other one.
     - If t_i is a terminal of G_h and t_{i+1} is not: if t_i corresponds to a_h then t_i is to the left of t_{i+1}; otherwise, t_i is to the right of t_{i+1}. Similarly, if only t_{i+1} is a terminal of G_h: if t_{i+1} corresponds to a_h then t_{i+1} is to the left of t_i; otherwise, t_{i+1} is to the right of t_i.
     - If neither t_i or t_{i+1} are terminals of G_h, then t_i must be a terminal of a maximal parallel subgraph G_j contained within G_h and t_{i+1} must be a terminal of a different maximal parallel subgraph contained within G_h that is adjacent to G_j. If t_i corresponds to the terminal a_j then t_{i+1} is to the left of t_i; otherwise, t_i is to the left of t_{i+1}.
   - Otherwise, if t_i and t_{i+1} are not in the same maximal parallel subgraph. If t_i corresponds to the terminal a_p of a maximal parallel subgraph G_p and t_{i+1} corresponds to the terminal b_q of a maximal parallel subgraph G_q then t_{i+1} is to the left of t_i; otherwise t_i is to the left of t_{i+1}.
     Similarly, we can determine if s is to the left of t_1 and if t is to the left of t_d.

   (a) If t_1 is to the right of s and t_2 is to the right of t_1 then the segment $<t_1, t_2>$ must exit G_s and so t_1 must be either vertex b'_s or vertex b''_s. If G_s contains both s and t then the terminal b''_s of G''_s is the vertex corresponding to t_1; otherwise, the terminal b'_s of G'_s corresponds to t_1 (see Figure 26).

   Let G_k be the largest-indexed maximal parallel subgraph containing G_s. To show that the directed path corresponding to $<s, t_1>$ can be extended to a vertex corresponding to t_2 we consider three cases: (i) $t_2 = t$, (ii) the segment $<t_1, t_2>$ exits G_k, and (iii) the segment $<t_1, t_2>$ stays in G_k.

   (i) If $t_2 = t$, then G_s does not contain t, because the segment $<t_1, t_2>$ exits G_s (see Figure 26). The algorithms in Sections 3.3.1 and 3.3.2(b)

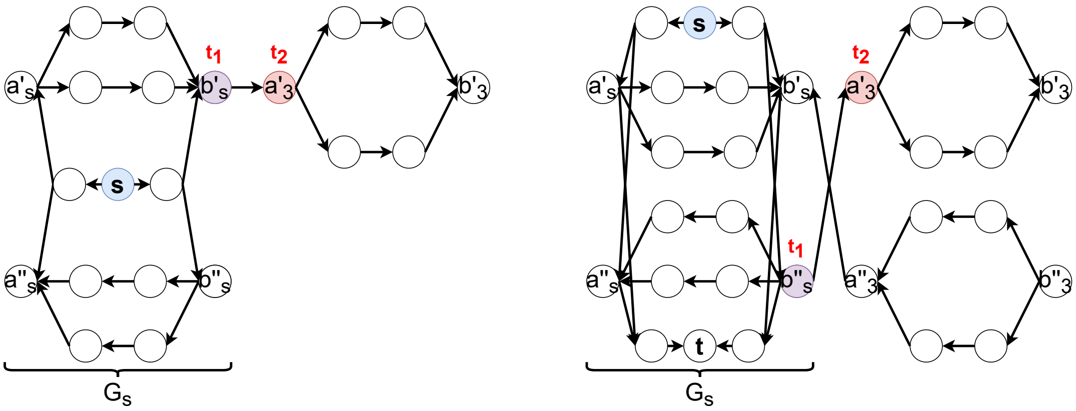


Fig. 26 When t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1, the segment $<t_1, t_2>$ exits G_s. (Left) G_s does not contain t and so b'_s is the vertex corresponding to t_1. (Right) G_s contains t and so b''_s is the vertex corresponding to t_1.

and (c) direct the simple paths in $\mathcal{G}$ from b'_s to t, ensuring the existence of a path in $\mathcal{G}$ equivalent to $<t_1, t>$ that extends from $<s, t_1>$.

(ii) If $<t_1, t_2>$ exits G_k then the terminal b_s of G_s is also a terminal of G_k and so the segment $<t_1, t_2>$ enters an adjacent series-parallel subgraph $\mathcal{G}_{\ell_x}$ (note that in this case σ can not return to G_k and hence G_k does not contain t). So, t_1 must be vertex b'_s. If $\mathcal{G}_{\ell_x}$ does not contain t, then the algorithms in Lemma 4 direct all simple paths in $\mathcal{G}_{\ell_x}$ from b'_s to t_2, ensuring the existence of a path in $\mathcal{G}$ equivalent to $<t_1, t_2>$ that extends from $<s, t_1>$. If $\mathcal{G}_{\ell_x}$ contains t, then the algorithms in Section 3.3.2 direct the simple paths in $\mathcal{G}_{\ell_x}$ from b'_s to t_2, ensuring the existence of a path in $\mathcal{G}$ equivalent to $<t_1, t_2>$ that extends from $<s, t_1>$.

(iii) If $<t_1, t_2>$ stays within G_k then the terminal b_s of G_s is not a terminal of G_k, and $<t_1, t_2>$ either exits G_s into a maximal parallel subgraph G_p containing G_s or exits G_s into a maximal parallel subgraph G_q adjacent to G_s (see Figure 27). In either case, the algorithms in Section 3.3.2(a)-(c) create the directed graphs G'_p and G''_p or the directed graphs G'_q and G''_q and the algorithm in Section 3.3.2(d) connects G'_s to G'_p or G'_q. If G_s contains t, then the algorithm in Section 3.3.3(b) connects b''_s to G'_p or G'_q. This ensures the existence of a path in $\mathcal{G}$ equivalent to $<t_1, t_2>$ that extends from $<s, t_1>$.

(b) If t_1 is to the right of s and t_2 is to the left of t_1, then the terminal b''_s of G''_s is the vertex that corresponds to t_1 (see Figure 28). If the terminal b_s of G_s is also a terminal of a maximal parallel subgraph G_r containing G_s, then additional vertices were added by the algorithm as described in Section 3.3 before beginning any of the steps so that b'_s, b''_s, b'_r, and b''_r were entangled, but then b''_s and b'_r were merged together by the algorithm in Section 3.3.3 and so the terminal b''_s is incident on edges directed towards the left that enter the other parallel components of G_i (that do not contain s).

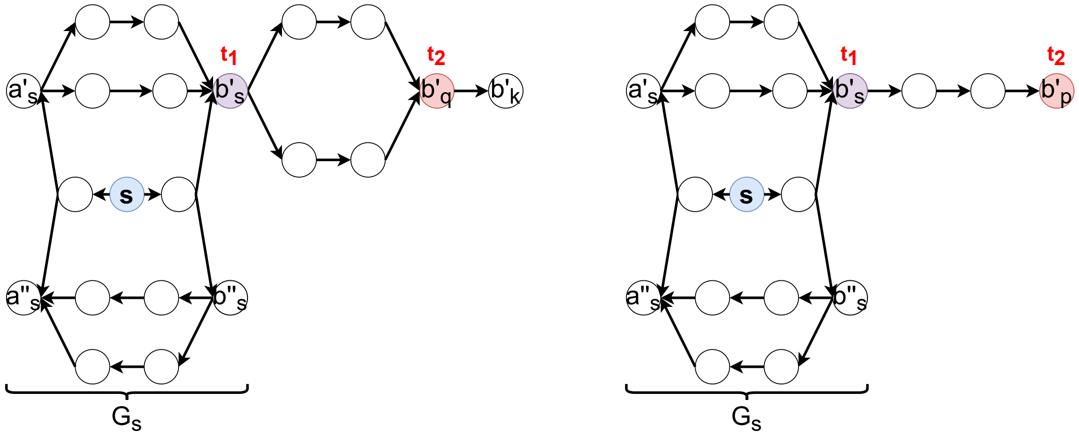


Fig. 27 When t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1, and the segment $<t_1, t_2>$ stays in G_k. (Left) Segment $<t_1, t_2>$ exits into an adjacent maximal parallel subgraph G_q. (Right) Segment $<t_1, t_2>$ exits into a maximal parallel subgraph G_p.

The algorithms in Sections 3.3.2(a)-(d) direct edges from b''_s to the left toward either the vertices corresponding to the terminal a_s or a_r, thus segment $<s, t_1>$ can be extended to $<t_1, t_2>$.

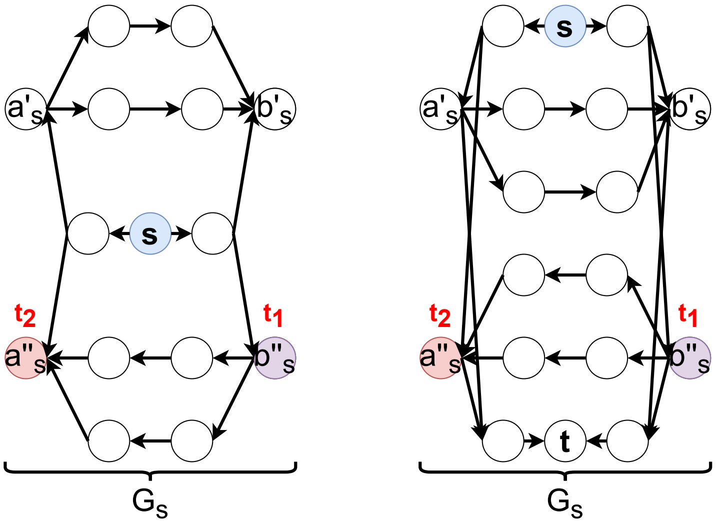


Fig. 28 When t_1 (purple) is to the right of s and t_2 (red) is to the left of t_1, the segment $<t_1, t_2>$ is still contained within G_s. (Left) G_s does not contain t and so b''_s is the vertex corresponding to t_1. (Right) G_s contains t and so b''_s is the vertex corresponding to t_1.

(c) If t_1 is to the left of s and t_2 is to the left of t_1: If G_s contains both s and t then the terminal a'_s of G'_s is the vertex corresponding to t_1; otherwise, the terminal a''_s of G''_s is the vertex that corresponds to t_1 (see Figure 29). The proof to show that for path $<s, t_1>$, $<t_1, t_2>$ there is an equivalent path in $\mathcal{G}$ is similar to the proof for the above case (a).

(d) If t_1 is to the left of s and t_2 is to the right of t_1, then the terminal a'_i of G'_s is the vertex that corresponds to t_1 (see Figure 30). The proof to

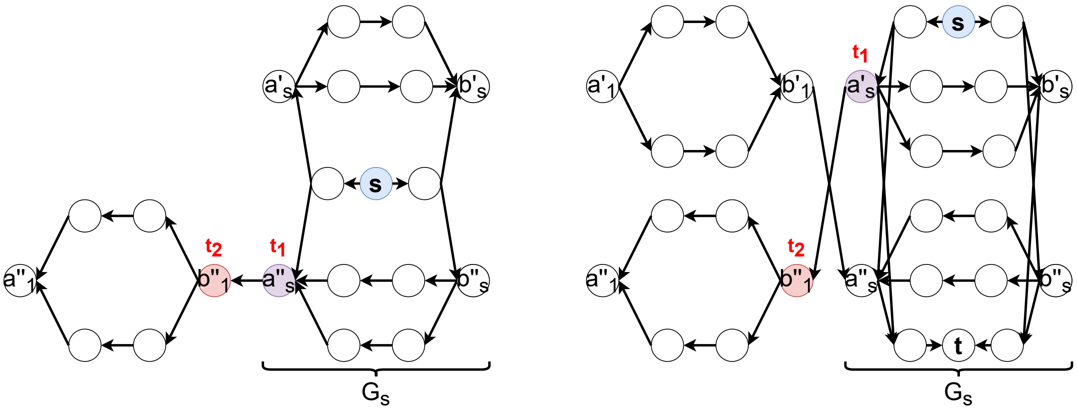


Fig. 29 When t_1 (purple) is to the left of s and t_2 (red) is to the left of t_1, the segment $<t_1, t_2>$ exits G_s. (Left) G_s does not contain t and so a''_s is the vertex corresponding to t_1. (Right) G_s contains t and so a'_s is the vertex corresponding to t_1.

show that for path $<s, t_1>$, $<t_1, t_2>$ there is an equivalent path in $\mathcal{G}$ is similar to the proof for the above case (b).

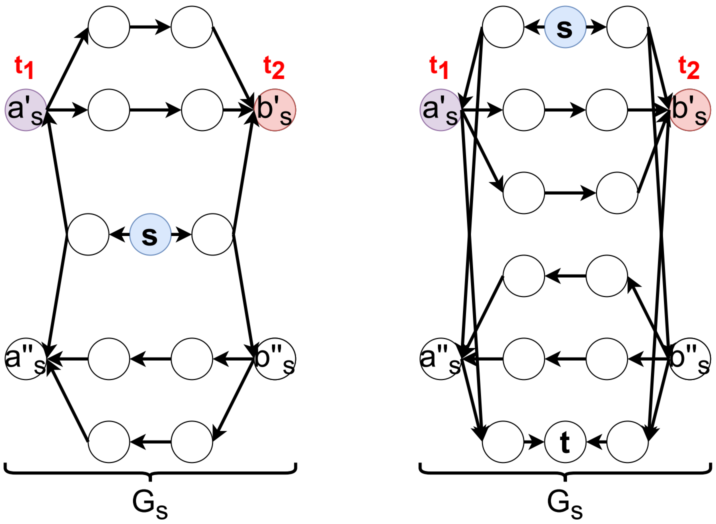


Fig. 30 When t_1 (purple) is to the left of s and t_2 (red) is to the right of t_1, the segment $<t_1, t_2>$ is still contained within G_s. (Left) G_s does not contain t and so a'_s is the vertex corresponding to t_1. (Right) G_s contains t and so a'_s is the vertex corresponding to t_1.

3. If the first segment is $<s, t_1>$, and s is a terminal of G_s we consider two cases. If $t_1 = t$ or t_1 is the other terminal of G_s, then the algorithm in Section 3.3.1 directs the paths from s to the other terminal of G_s and from s to t if t is contained in G_s, ensuring that the path in $\mathcal{G}$ from s to t_1 is equivalent to the corresponding subpath in σ and that this path can be extended to $<t_1, t_2>$.

   When t_1 is not t or a terminal of G_s, the algorithms in Sections 3.3.2 and 3.3.3 ensure the existence of two paths in $\mathcal{G}$ equivalent to $<s, t_1>$: one path from s to t'_1 and one from s to t''_1. Since only one of these directed paths can be extended to an equivalent path for the next segment $<t_1, t_2>$, we show

how to select the correct one so that the invariant holds, by considering eight cases. Let G_p be the lowest-indexed series-parallel subgraph containing G_s (if such a series-parallel subgraph exists) and let G_q be a maximal parallel subgraph adjacent to G_s (if such a maximal parallel subgraph exists).

(a) $s = a_s$, t_1 is to the right of s, and t_2 is to the right of t_1. Since $s = a_s$ the algorithms in Section 3.3.2 do not duplicate any of the vertices in G_s (see Figure 31). Therefore, t_1 must be vertex b_s. If $t_2 = t$ then if G_s is contained within another maximal parallel subgraph then the algorithm in Section 3.3.2(b) directed the edge from b_s to t and if G_s is not contained within another maximal parallel subgraph then one of the subgraphs $\mathcal{G}_{\ell_x}$ consists of only the two vertices t_1 and t_2 and the algorithm at the beginning of Section 3.3 directed the path away from s and towards t, ensuring that in $\mathcal{G}$ the path from s to t_1 can be continued towards t_2. Otherwise, if $t_2 \neq t$ then the algorithm in Section 3.3.2(d) ensures that in $\mathcal{G}$ the path from s to t_1 can be continued towards t_2.

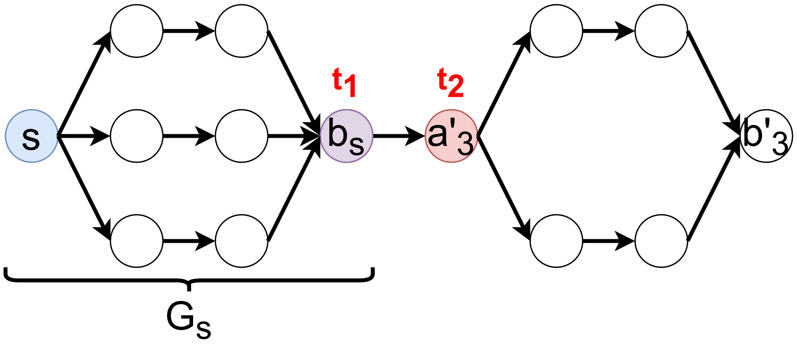


Fig. 31 $s = a_s$ and so the vertices of G_s were not duplicated. Vertice t_1 (purple) is to the right of s and t_2 (red) is to the right of t_1. The segment $<t_1, t_2>$ exits G_s.

(b) If $s = a_s$, t_1 is to the right of s, and t_2 is to the left of t_1 then t_1 must be b_s because of the properties of series-parallel graphs as $<s, t_1>$, $<t_1, t_2>$ is part of a simple path σ from s to t. If $t_2 = t$ then the algorithm in Section 3.3.1 ensures the directed path from s to t if t is in G_s, and the algorithm in Section 3.3.2(b) ensures the directed path from s to t if t is not in G_s. If $t_2 \neq t$ then t_1 must be also a terminal of a maximal parallel subgraph containing G_s and so the algorithms in Sections 3.3.2(a)-(c) ensure a directed path in $\mathcal{G}$ equivalent to $<s, t_1>$, $<t_1, t_2>$.

To avoid being repetitive, from this point on we will not continue making reference to the algorithms in Section 3.3, but we will explain what the terminals t_1 and t_2 are and we will make reference to a figure illustrating each case.

(c) $s = a_s$, t_1 is to the left of s, and t_2 is to the right of t_1. Let G_p be the smallest indexed maximal parallel subgraph containing t_1. Then, vertex t_1 must be terminal a_p of G_p as t_2 is to the right of t_1. If G_p contains both s and t, then the terminal a''_p of G''_p is the vertex corresponding to

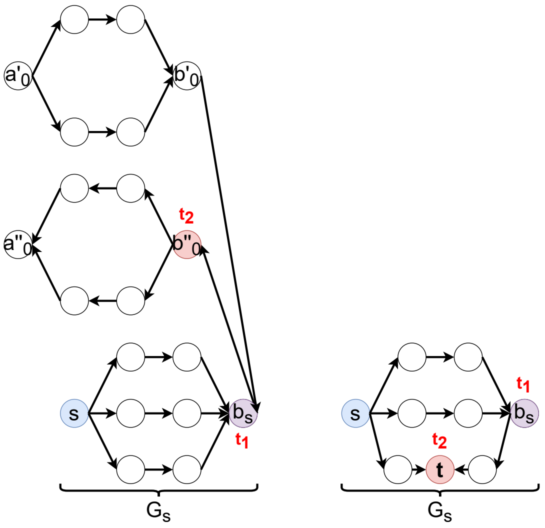


Fig. 32 $s = a_s$ and so the vertices of G_s were not duplicated. t_1 (purple) is to the right of s and t_2 (red) is to the left of t_1. (Left) G_s does not contain t and so b_s is the vertex corresponding to t_1. (Right) G_s contains t and so b_s is the vertex corresponding to t_1 and $t_2 = t$ as this is the only feasible path.

t_1; otherwise, the terminal a'_p of G'_p is the vertex that corresponds to t_1 (see Figure 33).

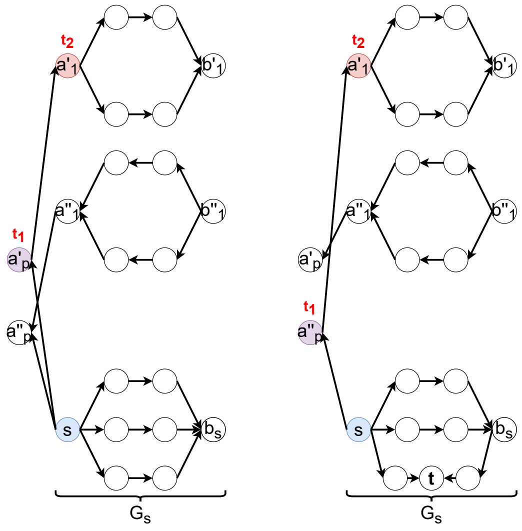


Fig. 33 t_1 (purple) is to the left of s and t_2 (red) is to the right of t_1; observe that the segment $<s, t_1>$ exits G_s into the maximal parallel subgraph G_p containing G_s. (Left) G_p does not contain t and so a'_i is the vertex corresponding to t_1. (Right) G_p contains both s and t and so a''_i is the vertex corresponding to t_1.

(d) $s = a_s$, t_1 is to the left of s, and t_2 is to the left of t_1. If the segment $<s, t_1>$ enters a maximal parallel subgraph G_q that does not contain G_s, then either the vertex b''_q corresponds to t_1, or the vertex a''_q corresponds to t_1 if G_s and G_q have a common terminal; otherwise, if the segment

$<s, t_1>$ enters a maximal parallel subgraph G_p containing G_s then the vertex a''_p corresponds to t_1 (see Figure 34).

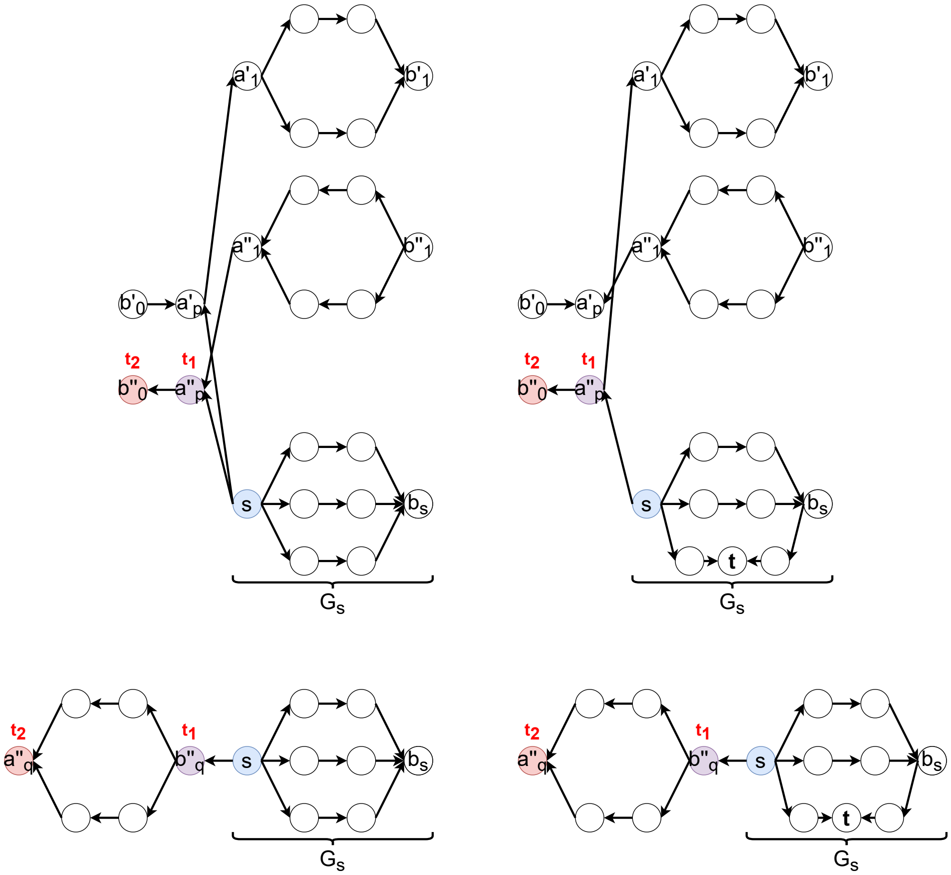


Fig. 34 $s = a_s$ and so the vertices of G_s were not duplicated. t_1 (purple) is to the left of s and t_2 (red) is to the left of t_1. (Top-left) The segment $<s, t_1>$ exits G_s into a maximal parallel subgraph G_p containing G_s and the vertex a''_p corresponds to t_1. (Bottom-left) The segment $<s, t_1>$ exits G_s and enters an adjacent maximal parallel subgraph G_q so the vertex b''_q corresponds to t_1. (Top-right) The segment $<s, t_1>$ exits G_s into a maximal parallel subgraph G_p containing G_s and the vertex a''_p corresponds to t_1. (Bottom-right) The segment $<s, t_1>$ exits G_s and enters an adjacent maximal parallel subgraph G_q so the vertex b''_q corresponds to t_1.

(e) The remaining four cases when $s = b_s$ are simply mirror images of the cases when $s = a_s$.

Assume that a directed path σ' equivalent to the subpath of σ from s to t_i has been selected that can be extended to a vertex of $\mathcal{G}$ corresponding to t_{i+1}. We show how to construct a directed path equivalent to the subpath of σ from s to t_{i+1} that can be extended to a vertex corresponding to t_{i+2}.

1. If $t_{i+1} = t$, then by the assumption a directed path σ' equivalent to σ exists.
2. If $t_{i+1} \neq t$, then by the assumption, σ' can be extended to reach a vertex of $\mathcal{G}$ corresponding to t_{i+1}; however, $\mathcal{G}$ can have two duplicate vertices corresponding to t_{i+1}. Since only one of these duplicate vertices can extend

the directed path σ' to a vertex corresponding to t_{i+2}, we show how to select the duplicate vertex in $\mathcal{G}$ that corresponds to t_{i+1} so that the invariant holds, by considering eight cases.

(a) If t_{i+1} is the right terminal b_r of some maximal parallel subgraph G_r of G, t_{i+1} is to the right of t_i, and t_{i+2} is to the right of t_{i+1}, then the terminal b'_r of G'_r is the vertex that corresponds to t_{i+1} (see Figure 35). The algorithms in Sections 3.3.1 and 3.3.2 and Lemma 4 direct the simple paths in $\mathcal{G}$ from b'_r to the right ensuring the existence of a path in $\mathcal{G}$ equivalent to $<t_{i+1}, t_{i+2}>$ that extends from $<t_i, t_{i+1}>$.

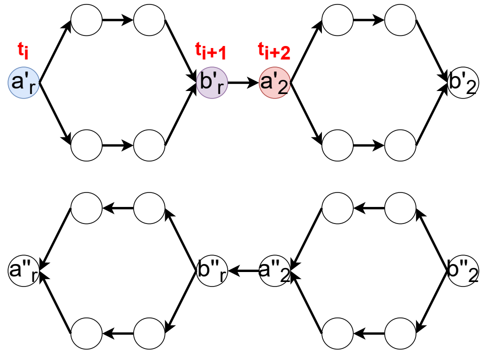


Fig. 35 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the right of t_{i+1}. b'_r is the vertex that corresponds to t_{i+1}.

(b) If t_{i+1} is the right terminal b_r of a maximal parallel subgraph G_r, t_{i+1} is to the left of t_i, and t_{i+2} is to the left of t_{i+1}.

- If t_{i+1} is a terminal of G_s which contains both s and t, then the terminal b'_s of G'_s is the vertex corresponding to t_{i+1} and $t_{i+2} = t$ (see Figure 36 (left)). Note that even if the terminal a_s is common with the terminal b_r of G_r of a maximal parallel subgraph adjacent to G_s, t_{i+1} cannot correspond to a_s because that implies that either $t_i = s$, but the terminals t_i were not defined to include s, or $t_i = b_s$, but then both terminals of G_s have been traversed and the path σ can not return to t.
- Otherwise, the terminal b''_r of G'_r is the vertex that corresponds to t_{i+1} (see Figure 36 (right)). Note that if t_{i+1} corresponds to the terminal a_s of G_s which contains only s, and if the terminal a_s is common with the terminal b_r of G_r of a maximal parallel subgraph adjacent to G_s, then vertex b''_r is still the vertex that corresponds to t_{i+1}.

Note that if t_{i+1} is the terminal b_r of G_r, G_r contains G_s, and G_s contains t, then it must be the case that the path σ has already traversed from s to a_i before reaching t_{i+1}. To see this, note that the segment $<t_i, t_{i+1}>$ enters G_r, which implies that σ previously exited G_r, but σ could not have exited G_r using b_r as then this terminal would have been visited twice. Hence, σ exited G_r using a_r. Therefore, σ has traversed both terminals of G_r and thus must traverse the parallel component of G_r that contains t (as otherwise σ could never reach t). The algorithm in

Section 3.3.3(b) kept the edge incident on b''_r that is directed towards G''_r and hence in this case there exists a path in $\mathcal{G}$ equivalent to $<t_{i+1}, t_{i+2}>$ that extends from $<t_i, t_{i+1}>$.

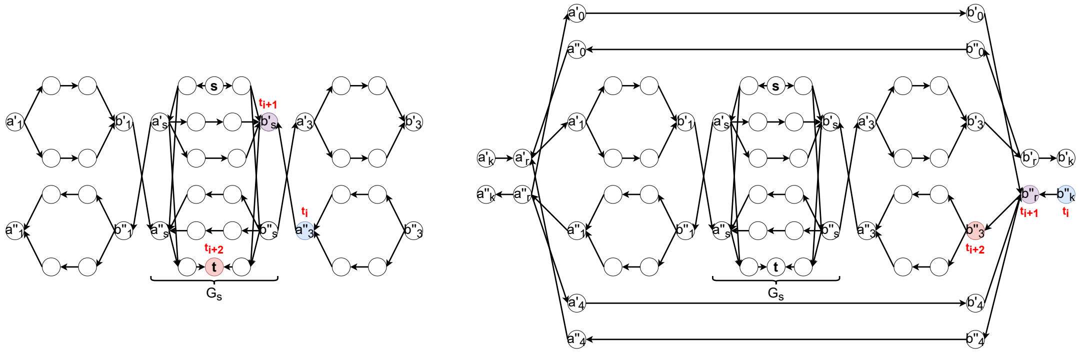


Fig. 36 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the left of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. (Left) t_{i+1} is a terminal of G_s which also contains t and so b'_s is the vertex that corresponds to t_{i+1}. Note that in this case $t_{i+2} = t$ due to how we renamed the terminals t_i. (Right) In all other cases, b''_r is the vertex that corresponds to t_{i+1}.

(c) If t_{i+1} is the left terminal a_r of G_r, t_{i+1} is to the right of t_i, and t_{i+2} is to the right of t_{i+1}.
   - If t_{i+1} is a terminal of G_s which contains both s and t, then the terminal a''_s of G''_s is the vertex corresponding to t_{i+1} (and $t_2 = t$, see Figure 37 (Left)).
   - Otherwise, the terminal a'_r of G'_r is the vertex that corresponds to t_{i+1} (see Figure 37 (Right)).

   Note that this case is symmetric to case (b) above, and so the proof is similar.

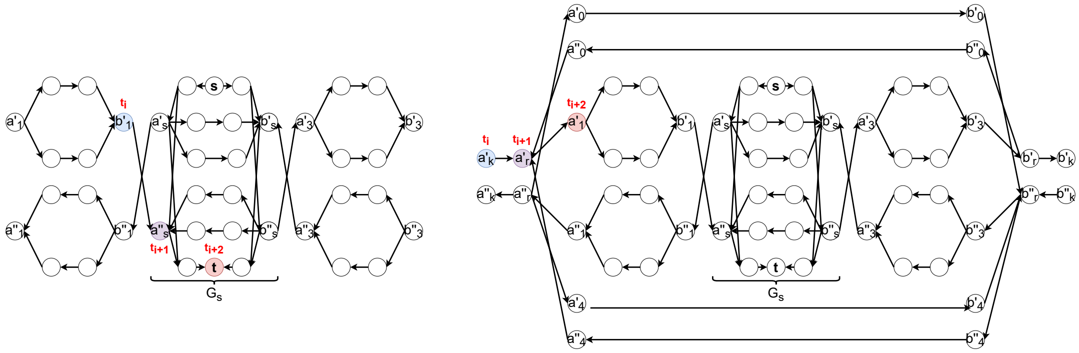


Fig. 37 t_{i+1} (purple) is the left terminal a_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the right of t_{i+1}. (Left) t_{i+1} is a terminal of G_s which contains t and so a''_s is the vertex that corresponds to t_{i+1}. (Right) In all other cases, a'_r is the vertex that corresponds to t_{i+1}.

(d) If t_{i+1} is the left terminal a_r of a maximal parallel subgraph G_r, t_{i+1} is to the left of t_i, and t_{i+2} is to the left of t_{i+1} then the terminal a''_r of

G''_r is the vertex that corresponds to t_{i+1} (see Figure 38). Note that this case is symmetric to case (a) above.

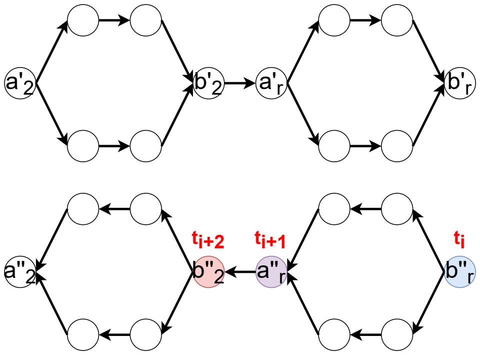


Fig. 38 t_{i+1} (purple) is the left terminal a_r of G_r, t_{i+1} is to the left of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. a''_r is the vertex that corresponds to t_{i+1}.

(e) If t_{i+1} is the right terminal b_r of a maximal parallel subgraph G_r, t_{i+1} is to the right of t_i, and t_{i+2} is to the left of t_{i+1} then there are four possibilities:
  (i) If G_r contains both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component containing s then the terminal b'_r of G'_r is the vertex that corresponds to t_{i+1} (see Figure 39 (Top-right)).
  (ii) If G_r contains both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s then the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1} (see Figure 39 (Bottom-right)).
  (iii) If G_r does not contain both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component containing s then the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1} (see Figure 39 (Top-left)).
  (iv) If G_r does not contain both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s then the terminal b'_r of G'_r is the vertex that corresponds to t_{i+1} (note that in this case the segment $<t_i, t_{i+1}>$ must enter a parallel component containing t) (see Figure 39 (bottom-left)).

(f) If t_{i+1} is the left terminal a_r of a maximal parallel subgraph G_r, t_{i+1} is to the right of t_i, and t_{i+2} is to the left of t_{i+1}, then it must be the case that the terminal a_r is common with the terminal b_p of a maximal parallel subgraph G_p adjacent to G_r. To see this, note that the a terminal is the left terminal of a maximal parallel graph and any path approaching the a terminal (that is not common with any other b terminal) from the left side must continue to the right. Therefore, t_{i+1} is also the right terminal b_p and hence this case is identical to case (e) above.

(g) If t_{i+1} is the left terminal a_r of a maximal parallel subgraph G_r, t_{i+1} is to the left of t_i, and t_{i+2} is to the right of t_{i+1} then there are four

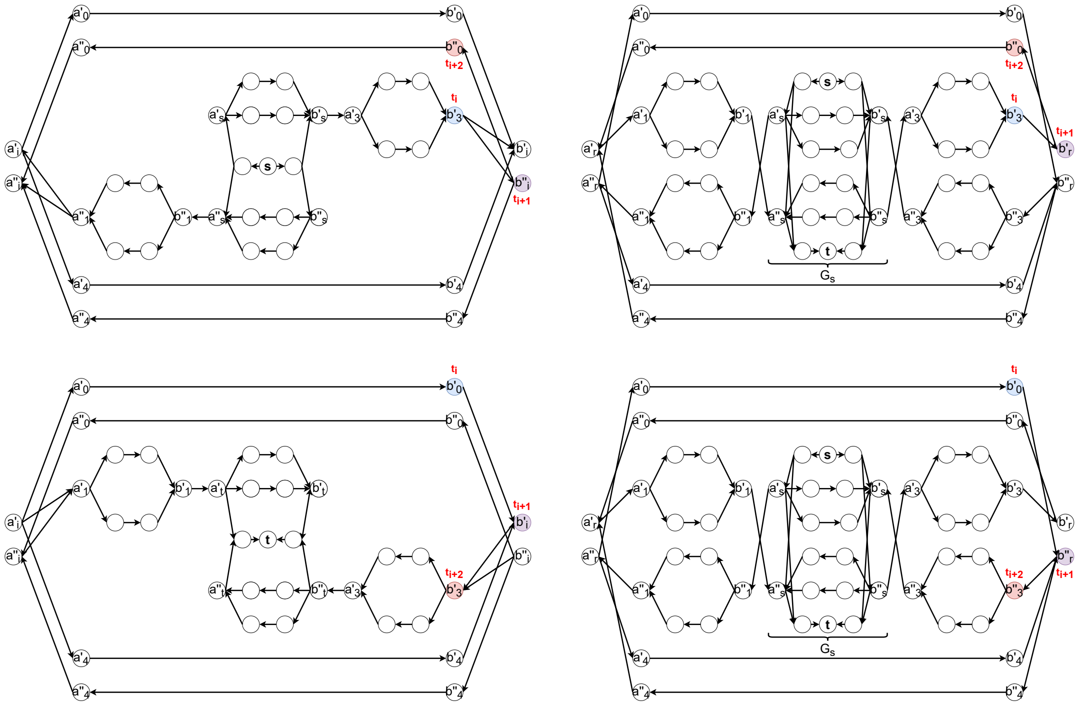


Fig. 39 t_{i+1} (purple) is the right terminal b_r of G_r, t_{i+1} is to the right of t_i (blue), and t_{i+2} (red) is to the left of t_{i+1}. (Top-left) The segment $<t_i, t_{i+1}>$ exits a parallel component containing s and so the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1}. (Bottom-left) The segment $<t_i, t_{i+1}>$ exits a parallel component not containing s and so the terminal b'_r of G''_r is the vertex that corresponds to t_{i+1}. (Top-right) G_r contains both s and t and the segment $<t_i, t_{i+1}>$ exits a parallel component containing s and so the terminal b'_r of G''_r is the vertex that corresponds to t_{i+1}. (Bottom-right) G_r contains both s and t and the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s and so the terminal b''_r of G''_r is the vertex that corresponds to t_{i+1}.

possibilities. Note that this is symmetric to case (e) above and hence we omit the figures.

(i) If G_r contains both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component containing s then the terminal a''_r of G''_r is the vertex that corresponds to t_{i+1}.

(ii) If G_r contains both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s then the terminal a'_r of G'_r is the vertex that corresponds to t_{i+1}.

(iii) If G_r does not contain both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component containing s then the terminal a'_r of G'_r is the vertex that corresponds to t_{i+1}.

(iv) If G_r does not contain both s and t in the same maximal parallel subgraph and if the segment $<t_i, t_{i+1}>$ exits a parallel component not containing s then the terminal a''_r of G''_r is the vertex that corresponds to t_{i+1} (note that in this case the segment $<t_i, t_{i+1}>$ must enter a parallel component containing t).

(h) If t_{i+1} is the right terminal b_r of a maximal parallel subgraph G_r, t_{i+1} is to the left of t_i, and t_{i+2} is to the right of t_{i+1}, then it must be the case that the terminal b_r is common with the terminal a_p of a maximal parallel subgraph G_p adjacent to G_r. To see this, note that the b terminal is the right terminal of a maximal parallel graph and any path approaching the b terminal (this is not common with any other a terminal) from the left side must continue to the right. Therefore, t_{i+1} is also the left terminal a_p and hence this case is identical to case (g) above.

Algorithm 2 creates at most two vertices per vertex in G, and the vertices in $\mathcal{G}$ store the same items as the corresponding vertices in G. Moreover, Algorithm 2 creates at most two directed edges for each edge in G, and the edges in $\mathcal{G}$ have the same length as the corresponding edges in G. As shown above, for any path σ in G from s to t, there is an equivalent simple path σ' in $\mathcal{G}$ from s to t. □

Lemma 7 *Algorithm 2 transforms an instance of ThOP on an undirected series-parallel graph G into an instance of ThOP on a directed graph $\mathcal{G}$ such that for every directed path σ' from s to t in $\mathcal{G}$, there is an equivalent simple path σ from s to t in G.*

Proof Given a path σ' from s to t in $\mathcal{G}$ we include in σ the undirected edges corresponding to the directed edges in σ' and so we only need to show that σ' does not traverse through two vertices storing the same set of items.

Recall that Algorithm 2 makes at most two duplicate vertices corresponding to each vertex in G. For a maximal parallel subgraph G_i of G, one of these duplicate vertices v' is in G'_i and the other duplicate vertex v'' is in G''_i. If G_i does not contain s or t, then no path from s to t in $\mathcal{G}$ could visit both v' and v'' because Algorithm 2 does not add any edges to $\mathcal{G}$ for a path σ' to exist that traverses from G'_i to G''_i or from G''_i to G'_i.

Recall that for parallel components pc_{ih} in a maximal parallel subgraph G_i that contain s or t, but not both within the same maximal parallel subgraph G_j, the algorithm in Section 3.3.3(a) deletes one of G'_j or G''_j for each maximal parallel subgraph G_j in G_i, except G_s and G_t; hence the only vertices for which their two duplicate copies exist in G_i are located in the DAGs for G_s and G_t. For G'_s and G''_s, a path σ' from s to t must either traverse and exit G'_s or traverse and exit G''_s; in either case, σ' cannot traverse through the two duplicate vertices of a vertex of G_s. Similarly, a path σ' from s to t must either traverse G'_t and then go to t or traverse G''_t and then go to t; in either case, σ' cannot traverse through the two duplicate vertices of the same vertex of G_t.

For a parallel component pc_{ih} in G_i that contain s and t within the same maximal parallel subgraph G_j, a path σ' from s to t must traverse one of the parallel components pc'_{ih} or pc''_{ih} created for parallel component pc_{ih}, but σ' cannot traverse both. To see this, observe that if σ' traverses pc''_{ih} toward the left terminal a''_i of G''_i, then by the algorithm in Section 3.3.3(b) σ' cannot reach pc'_{ih}, but it can traverse to the right to a different parallel component pc'_{im} and onto b''_i, from where it still cannot reach pc'_{ih}. Note that when σ' traverses from b''_i back into pc''_{ih}, σ' continues toward t and this does not create a cycle. Similarly, σ' can traverse pc'_{ih} to the right terminal b'_i from where it cannot reach pc''_{ih} but it can traverse to the left to a different parallel component pc''_{im} and onto a'_i, from where it must continue toward t.

Therefore, σ' is unable to traverse multiple duplicate vertices that correspond to the same vertex in G. ☐

Theorem 2 *There is a PTAS for the thief orienteering problem when the graph* G *is an undirected series-parallel graph that produces solutions that use time at most* $T(1+\epsilon)$ *for any* $\epsilon > 0$.

Proof Algorithm 2 transforms an undirected series-parallel graph G into a DAG $\mathcal{G}$ by creating at most one additional vertex for each vertex in G and creating two directed edges for each undirected edge in G, and so Algorithm 2 runs in polynomial time. By Theorem 2 in [3], Algorithm 3 in [3] is a PTAS for the thief orienteering problem when the input graph G is a DAG that produces solutions that use time at most $T(1+\epsilon)$ for any $\epsilon > 0$. Therefore, by Lemmas 5, 6, and 7 the combination of Algorithm 2 and Algorithm 3 in [3] is a PTAS for the thief orienteering problem when the input graph G is an undirected series-parallel graph that produces solutions that use time at most $T(1+\epsilon)$ for any $\epsilon > 0$. ☐

References

[1] Abdelkader, R., Abdelkader, Z., Mustapha, R., and Yamani, M.: Optimal allocation of reliability in series parallel production system. In Search Algorithms for Engineering Optimization, InTechOpen, Croatia, pp. 241-258, 2013.

[2] Batra, D., Gallagher, A., Parikh, D., and Chen, T.: Beyond trees: MRF inference via outer-planar decomposition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 2496-2503, 2010.

[3] Bloch-Hansen, A., Page, D., and Solis-Oba, R.: A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs. In International Workshop on Combinatorial Algorithms (IWOCA), pp. 87-98. Springer, 2023.

[4] Bonyadi, M., Michalewicz, Z., and Barone, L.: The travelling thief problem: the first step in the transition from theoretical problems to realistic problems. In IEEE Congress on Evolutionary Computation (CEC), pp. 1037-1044. IEEE, 2013.

[5] Chagas, J., Wagner, M.: Ants can orienteer a thief in their robbery. Operations Research Letters **48**(6), pp. 708-714. 2020.

[6] Chagas, J., Wagner, M.: Efficiently solving the thief orienteering problem with a max-min ant colony optimization approach. Optimization Letters **16**(8), pp. 2313-2331. 2022.

[7] Chung, F., Leighton, F., and Rosenberg, A.: Embedding graphs in books: a layout problem with applications to VLSI design. SIAM Journal on Algebraic Discrete Methods 1(**8**), pp. 33-58, 1987.

[8] Duffin, R.: Topology of series-parallel networks. Journal of Mathematical Analysis and Applications 2(**10**), pp. 303-318, 1965.

[9] Faêda, L. and Santos, A.: A genetic algorithm for the thief orienteering problem. In 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1-8. IEEE, 2020.

[10] Freeman, N., Keskin, B., and Çapar, İ.: Attractive orienteering problem with proximity and timing interactions. European Journal of Operational Research **266**(1), pp. 354-370. 2018.

[11] Gago, J., Hartillo, I., Puerto, J., and Ucha, J.: Exact cost minimization of a series-parallel reliable system with multiple component choices using an algebraic method. Computers and Operations Research **40**(11), pp. 2752-2759, 2013.

[12] Gunawan, A., Lau, H., and Vansteenwegen, P.: Orienteering problem: a survey of recent variants, solution approaches and applications. European Journal of Operatinal Research **255**(2), pp. 315-332, 2016.

[13] Santos, A. and Chagas, J.: The thief orienteering problem: Formulation and heuristic approaches. In IEEE Congress on Evolutionary Computation, pp. 1-9., 2018.

[14] Schietgat, L., Ramon, J., and Bruynooghe, M.: A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. Annals of Mathematics and Artificial Intelligence **69**, pp. 343-376, 2013.

[15] Valdes, J: Parsing flowcharts and series-parallel graphs. Ph.D. dissertation, Standford University, 1978.

[16] Wang, J., Wu, X., and Fan, X.: A two-stage ant colony optimization approach based on a directed graph for process planning. International Journal of Advanced Manufacturing Technology, 80, pp. 839-850, 2015.

Chapter 6

6 Paper 4: A Modified Hopfield Network for the K Median Problem

This paper is currently in preparation.

This paper investigates a non-traditional technique, the Hopfield network, and its application to the k-median problem. A local search algorithm designed by modifying the typical Hopfield network model is presented and experimentally evaluated against neural networks and local search algorithms from the literature, and our modified Hopfield network is shown to produce solutions very quickly with competitive approximation ratios.

A Modified Hopfield Network for the K-Median Problem

Abstract. The k-median problem is a classical clustering problem where given n locations one wants to select k locations such that the total distance between every non-selected location and its nearest selected location is minimized. The problem has a large number of applications in a variety of fields, including network design, resource allocation, and data mining.

We present a modified Hopfield network for the k-median problem. Our main contribution is the design of a synchronous neuron update function that acts like a single-swap local search algorithm. We experimentally evaluate our modified Hopfield network against Rossiter's modified Hopfield network, Haralampiev's competition-based neural network, and the local search algorithms of Arya et al., Pan and Zhu, and Cohan-Addad et al. We show that our modified Hopfield network converges to local optimal in very few iterations and hence has runtimes almost fifty times smaller than Rossiter's network and hundreds of times smaller than Haralampiev's network. The solutions produced by our algorithm are often better than the other neural networks and single-swap local search algorithms, and we propose how to implement a multi-swap version of our algorithm.

Keywords: Hopfield Networks · K-Median Problem · Neural Networks · Local Search · Combinatorial Optimization · Approximation Algorithms.

1 Introduction

The *k-median problem* (KMP) is defined as follows. Given a graph $G = (V, E)$ with n vertices and nonnegative distance $d_{i,j}$ for every edge (i, j) such that $d_{i,j} = d_{j,i}$, select k vertices, called facilities or medians, to minimize the sum of distances between every non-selected vertex and the facility closest to it. The problem has the following integer programming formulation:

$$\text{Minimize} : \sum_{i=1}^{n} \sum_{j=1}^{n} d_{i,j} x_{i,j} \tag{1a}$$

$$\text{Subject to} : \sum_{j=1}^{n} x_{i,j} = 1 \, , i = 1, ...n \tag{1b}$$

$$\sum_{i=1}^{n} F_i = k \tag{1c}$$

$$x_{i,j} \leq F_i \, , i = 1, ...n; \; j = 1, ...n \tag{1d}$$

$$F_i \, , x_{i,j} \in \{0, 1\} \tag{1e}$$

where n is the number of vertices in the graph, k is the number of facilities or medians, $d_{i,j}$ is the distance (cost) of vertex i serving vertex j, $F_i = 1$ represents that vertex i is a facility, and $x_{i,j} = 1$ represents that vertex j is being served by vertex i.

Equation (1a) represents the goal of minimizing the total distance from the clients to their nearest facility, equation (1b) requires that every vertex is served by exactly one other vertex, equation (1c) requires that exactly k facilities are selected, equation (1d) requires that only active facilities can serve other vertices, and equation (1e) requires that the variables F_i and $x_{i,j}$ are restricted to the values 0 and 1.

KMP is a well-known problem in the facility location literature [13]; the k selected vertices represent facilities and the non-selected vertices are the clients to be served. Not only is KMP applicable to a variety of fields such as network design [1], data mining [24], and web access [20], but KMP is also critical to several modern unsupervised and semi-supervised machine learning models [9]. KMP is known to be NP-hard [21], and Jain et al. [19] proved that KMP cannot be approximated within a factor strictly less than $1 + \frac{2}{e} \sim 1.736$, unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{O(\log \log n)})$.

Many of the successful approximation algorithms for KMP are either based on linear program rounding or local search algorithms. On the linear program rounding side, the current best result is the 2.670-approximation of Cohen-Addad et al. [10], which improved on the 2.675-approximation of Byrka et al. [5], and the 2.732-approximation of Li and Svensson [22]. All of the above approaches involve rounding fractional solutions produced by either linear programs or the convex combination of two integer solutions. For the euclidean k-median problem, Cohen-Addad et al. [9] have improved the approximation ratio to 2.406.

On the local search algorithm side, the current best result is the $(2.836 + \epsilon)$-approximation of Cohen-Addad et al. [8], which improved on the $(3 + \frac{2}{p})$-approximation of Arya et al. [2], where p is the number of facilities swapped out in one swap operation; whereas Arya et al. perform arbitrary multi-swaps, Cohen-Addad et al. constructed a new objective function that considers both the closest and second-closest facility for each client. Peng et al. [30] also created a local search algorithm that achieves a $(3 + \frac{2}{p})$-approximation ratio, and Pan and Zhu [28] presented a local search algorithm that performs well in practice but no analysis was given as to its approximation ratio.

Arya et al. [2] have also shown that any local search algorithm that performs *single swap operations* to produce a solution that cannot be further improved by any single-swap has an approximation ratio of at most 5. A swap operation transforms the current solution s into a new feasible solution s' by removing a facility from the solution s and adding a new facility.

There have also been many heuristics designed for KMP: the interchange algorithm [34], tree search [6], Kohonen maps [23], and more. Additionally, several neural networks have been designed for KMP: Merino, Muñoz-Pérez, and Jerez-Aragonés [11, 25, 26] designed a Hopfield network and several recurrent neural networks, Mishrra and Barman [27] designed two Hopfield networks, Haralampiev [14–16] designed a competition-based neural network, and Rossiter [32] designed a modified Hopfield network. Our work builds on the algorithm of Rossiter: We identify some flaws in Rossiter's algorithm, we design a new neuron update function, and we improve on the (already fast) speed of Rossiter's network.

In this paper, we present a modified Hopfield network for KMP and we show that in practice it produces solutions that are very close to optimal solutions. Our modified Hopfield network runs almost fifty times faster than Rossiter's [32] and several hundred times faster than Haralampiev's [14–16]. By running our algorithm several times and improving upon the best solutions that it finds, our algorithm produces approximation ratios that are often better than the other neural networks and single-swap local search algorithms.

The rest of the paper is organized in the following way: In Section 2 we describe a typical Hopfield neural network formulation for KMP, in Section 3 we present our modified Hopfield network model, in Section 4 we describe our network's algorithm, in Section 5 we present our experimental results, in Section 6 we explain the performances of the algorithms, and in Section 7 we provide our conclusions and future research directions.

2 A Typical Hopfield Network Formulation

Hopfield [17] first introduced the Hopfield network in 1982 as a neural network with a single layer consisting of artificial neurons. Each neuron i updates itself at regular time intervals according to Equation (2) based on the values of the other neurons in the network, the weights of the connections between neuron i and the rest of the neurons in the network, and the threshold value θ_i for neuron i. Assume that neurons have updated their activation values at time t, then time $t + 1$ is the next time that the neurons will update themselves. The activation value (output value) of neuron i at time $t + 1$ is:

$$V_i(t+1) = \begin{cases} 1 & \text{if } \sum_{j=1}^{N} T_{i,j} V_j(t) \geq \theta_i \,, \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where N is the number of neurons in the network, $T_{i,j}$ is the weight of the neuron connection between neurons i and j, $V_j(t)$ is the activation value of the neuron j at time t, and θ_i is the threshold value of the neuron i.

The activation values of the neurons in the network describe a unique configuration or state of the network; when neural networks are designed to represent optimization problems, the state of the network corresponds to a solution to the optimization problem, but the solution is not necessarily a feasible solution. Hopfield and Tank [18] showed that a Hopfield network can optimize a function of the following form:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{i,j} V_i V_j - \sum_{i=1}^{N} V_i I_i \tag{3}$$

where N is the number of neurons, V_i is the neuron state variable, $T_{i,j}$ is the neuron connection matrix, and I_i is the input bias. Equation (3) is called an *energy function*, and the basic idea is that desirable network states should correspond to low energy configurations, which in term should represent good solutions for the optimization problem being modeled. The energy function is a description of the activation values of the neurons (neurons whose activation values are 1 are said to be *active*) in the network; therefore, attempting to minimize the energy in a neural network described by a well-designed energy function should cause the network to reach a state where the active neurons represent high-quality feasible solutions. Equation (3) is a Lyapunov function, and its main property is that it always decreases or remains constant as the neurons are updated.

Hopfield networks implemented using Equation (3) tend to become stuck in local optimal solutions that often do not even represent feasible solutions to the corresponding optimization problem. Equation (3) can be improved by including the problem constraints in such a way that network states that violate the constraints have increased energy; therefore, the only way to reach low-energy states is to find feasible solutions.

A neural network can model KMP by using a single layer of neurons divided into two groups: The first group of neurons, corresponding to the variables F_i from equation (1), represent whether vertex j is an active facility, and the second group of neurons, corresponding to the variables $x_{i,j}$ from equation (1), represent whether client i is being served by facility j.

F_1	$x_{1,1}$	$x_{2,1}$	$x_{3,1}$
F_2	$x_{1,2}$	$x_{2,2}$	$x_{3,2}$
F_3	$x_{1,3}$	$x_{2,3}$	$x_{3,3}$

Table 1: Matrix representation of the neurons in a three location k-median problem. The first column contains only the facility neuron variables. Each row begins with a facility neuron and the following neurons are the client-facility neurons that can be served by that facility.

This results in $N = n^2 + n$ neurons and can be visualized in a tabular format as an $n \times (n+1)$ neuron matrix (see Table 1). The first column of the matrix contains the *facility neurons* that correspond to the variables F_i and represent whether a given vertex is an active facility. The remaining columns of the matrix contain the *client-facility neurons* that correspond to the variables $x_{i,j}$ and represents whether vertex j is being served by vertex i.

Let $V(i,j)$ be a function that maps the client-facility neuron variable $x_{i,j}$ to its corresponding neuron state variable V_h. We define this function as $V(i,j) = V_{n*i+j}$ where n is the number of vertices (see Table 2). The indices for the facility neuron variables F_i range from 1 to n so they can be represented using V_i from 1 to n.

$$\begin{vmatrix} V_1 & V_4 & V_7 & V_{10} \\ V_2 & V_5 & V_8 & V_{11} \\ V_3 & V_6 & V_9 & V_{12} \end{vmatrix}$$

Table 2: Matrix representation of the neurons in a three location k-median problem. The neuron state variables V_1 to V_n correspond to the F_i facility neuron variables and the remaining neuron state variables V_{n+1} to V_{n^2+n} correspond to the $x_{i,j}$ client-facility neuron variables.

To encourage low energy states to correspond to feasible solutions, constraint terms can be added to Equation (3):

$$\begin{aligned} E =& C_1 \sum_{i=1}^{n} \sum_{j=1}^{n} V(i,j) D_{i,j} && \text{(4a)} \\ &+ C_2 ((\sum_{i=1}^{n} V_i) - k)^2 && \text{(4b)} \\ &+ C_3 \sum_{i=1}^{n} ((\sum_{j=1}^{n} V(i,j)) - 1)^2 && \text{(4c)} \\ &+ C_4 ((\sum_{i=1}^{n} \sum_{j=1}^{n} V(i,j)) - n)^2 && \text{(4d)} \\ &+ C_5 \sum_{i=1}^{n} \sum_{j=1}^{n} (1 - V_i) V(j,i) && \text{(4e)} \end{aligned}$$

where $V(i,j)$ corresponds to the client-facility neuron state variable representing client i being served by facility j, $V(i)$ corresponds to the facility neuron state variable representing facility i, and the terms C_1 through C_5 are constants, chosen by the network designer, that amplify the amount of energy added to the network when the constraint is violated.

The first constraint, represented by Equation (4a), adds energy to the network corresponding to the distance between each facility and the clients that it serves. Larger distances between clients and facilities results in more energy added to the network and so achieving low-energy configurations requires assigned clients to nearby facilities.

The second constraint, represented by Equation (4b), adds energy to the network unless exactly k facilities neurons are *active* (facilities are active if their activation value is 1). Since KMP requires exactly k facilities to be selected, having more than k or less than k facilities active represents an infeasible solution and hence this constraint helps low-energy configurations to correspond to feasible solutions to KMP.

The third constraint, represented by Equation (4c), adds energy to the network unless a client is served by exactly one facility. KMP does not allow fractional solutions in which a client might be partially served by several facilities and hence each client must be assigned to exactly one facility to be a part of a feasible solution.

The fourth constraint, represented by Equation (4d), adds energy to the network unless exactly n client-facility neurons are active (facilities are considered clients that are served by themselves). Since KMP requires each client to be assigned to a facility, having less than n clients assigned to facilities represents an infeasible solution and hence this constraint helps low-energy configurations to correspond to feasible solutions to KMP.

Finally, the fifth constraint, represented by Equation (4e), adds energy to the network if a client-facility neuron $x_{i,j}$ is active but the corresponding facility neuron F_i is not active. In feasible solutions, only active facilities can serve the clients.

If Equation (4) is rearranged into the form of Equation (3), we can derive the connection values T and the input bias values I. Since the constraint terms are defined for subsets of neurons, the connection matrix T is defined by a piecewise function:

$$T_{i,j} = \begin{cases} -2C_2 & \text{if } i \text{ and } j \text{ are facility neurons} \\ 2C_5 & \text{if } i \text{ is a facility neuron and } j \text{ is a client-facility neuron served by } i \\ -2C_3 - 2C_4 & \text{if } i \text{ and } j \text{ are client-facility neurons in the same column} \\ -2C_4 & \text{if } i \text{ and } j \text{ are client-facility neurons in different columns} \\ 0 & \text{otherwise} \end{cases}$$

and the input bias values are defined as:

$$I_i = \begin{cases} 2C_2k & \text{if } i \text{ is a facility neuron} \\ -C_1D(i) + 2C_3 + 2C_4n - C_5 & \text{if } i \text{ is a client-facility neuron} \end{cases}$$

where $D(i)$ is a distance function that takes a client-facility neuron i and returns the distance between the represented client and facility and is defined as $D(i) = D_{\lfloor (i-1)/n \rfloor, ((i-1) mod n)+1}$.

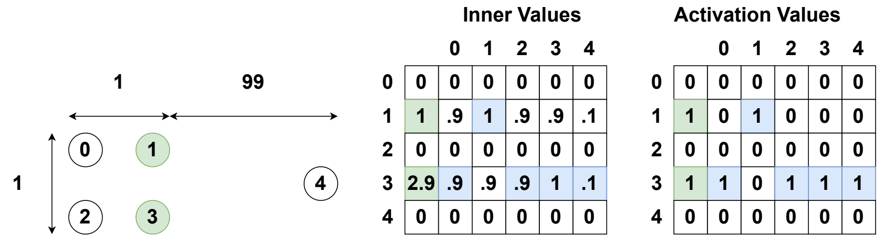
At this point, we would like to point out that a typical hopfield network would asynchronously update its neurons using Equation (2), attempting to minimize Equation (4), and when the energy function cannot be decreased any further the network configuration would correspond to a feasible solution to KMP with (hopefully) good quality. Our proposed neural neural network does not behave this way.

3 Our Modified Hopfield Network Model

Our proposed neural network makes several modifications to the Hopfield network formulated in Section 2. First, while a typical Hopfield network's neurons asynchronously update according to the equations that we described in Section 2, our Hopfield network is updated in a synchronous manner, which we describe in detail in Section 4.

In our network design, each neuron will contain two variables: the *activation value* and the *inner value*. The activation value is used to determine which facilities are active or whether a client is being served by a particular facility, and the inner value represents how close clients are to facilities and vice-versa. We show an example in Figure 1 using a small graph $G = (V, E)$ with $n = 5$ vertices in which vertices with indices 0 through 3 form a small cluster such that they each have distance 1 from each other, and the vertex with index 4 is located much further away. Let the vertices with indices 1 and 3 be the currently active facilities.

Looking at the activation values in Figure 1 (the rightmost matrix), we can see that in the first column of the matrix (corresponding to the facility neuron values F_i) the facility neuron variables F_1 and F_3 (colored in green) are both set to 1, indicating that these are the active facilities. The remaining columns of the activation value matrix represent the client-facility neuron variables, and we can see that $x_{1,1}$, $x_{3,0}$, $x_{3,2}$, $x_{3,3}$, and $x_{3,4}$ (colored in blue) are all set to 1, indicating that facility 1 serves itself and facility 3 serves clients 0, 2, 3, and 4.



Inner Values

		0	1	2	3	4
0	0	0	0	0	0	0
1	1	.9	1	.9	.9	.1
2	0	0	0	0	0	0
3	2.9	.9	.9	.9	1	.1
4	0	0	0	0	0	0

Activation Values

		0	1	2	3	4
0	0	0	0	0	0	0
1	1	0	1	0	0	0
2	0	0	0	0	0	0
3	1	1	0	1	1	1
4	0	0	0	0	0	0

Fig. 1: The modified Hopfield network is represented using two matrices: The first matrix contains the inner values and the second matrix contains the activation values.

Looking at the inner values in Figure 1 (the leftmost matrix), we can see that the facility neuron variables have values 1 and 2.9. The inner values of these facility neurons represent how close the facilities are to the clients they currently serve; in other words, how valuable these facilities might be to a potential solution. Consider facility f_3: The vertices with indices 0, 1, and 2 are all a distance of 1 away from f_3 and hence can be served at a low cost, which is represented in the inner value matrix as a high inner value of 0.9. Facility f_3 can serve itself at no cost and so the client-facility neuron variable $x_{3,3}$ has the maximum value of 1, but the vertex with index 4 is a distance of 99 away and so the inner value of the client-facility neuron $x_{3,4}$ is only 0.1. The inner value of f_3 is the sum of the client-facility inner values that are assigned to f_3; note that the activation value of $x_{3,1}$ is 0 because that vertex is not assigned to f_3 (it is a facility that serves itself) and so the inner value of $x_{3,1}$ is not added to the inner value of f_3.

The connection values for our modified Hopfield network are not derived from the energy function; instead, we have carefully chosen how the neurons can update each other, and we do not add an input bias I to our neurons. There are several different ways that the neuron connections can be influenced, and we introduce the most basic way first.

In the first column of the inner value matrix in Figure 1, the facility neuron variables corresponding to the active facilities have the only non-zero values; this is because we calculate the inner value of an active facility neuron F_i as the dot product of the cells $x_{i,1}$, $x_{i,2}$, ..., $x_{i,n}$ from both the inner value and activation value matrices. More specifically,

$$innerValue[i,0] = \sum_{j=0}^{n} innerValue[i,j+1] \times activationValue[i,j+1]$$

Note that when a vertex with index j is served by a facility with index i we refer to its corresponding client-facility neuron as $x_{i,j}$, but this neuron is located in the matrices at column $j+1$ because column 0 is reserved for the facility neurons.

We can also see that in the rest of the columns of the matrix, the client-facility neuron variables corresponding to the active facilities have the only non-zero variables; this is because we calculate the inner value of a client-facility neuron $x_{i,j}$ such that

$$innerValue[i,j+1] = activationValue[i,0] \times D'[i,j]$$

where the matrix D' stores normalized distances between the vertices.

To normalize the distance values, we scale each of the distances $d_{i,j}$ between vertex pairs to the range $[0,1]$ and then subtract the resulting value from 1. Normalizing the distances in this manner allows for intuitive matrix manipulations. For example, activation values for active and inactive facilities are 1 and 0, respectively, and so multiplying these activation values by the distance to a potential client creates either a non-zero inner value (for client-facility neurons corresponding to active facilities) or a zero inner value (for client-facility neurons corresponding to inactive facilities).

Facilities that serve themselves do so at no cost. Using the original distances, consider a facility f_a that only serves itself versus a facility f_b that serves itself and several nearby clients: The most valuable facility in this example is clearly f_b as it is serving multiple neighbours, but facility f_a is contributing less total cost to the solution, which might make facility f_a seem more desirable since our goal is to minimize the total cost. In contrast, using our normalized distances, facility f_a has an inner value of 1 (for serving itself) but facility f_b would have a much higher inner value because each close client that it serves would increase its inner value by a large amount.

Normalizing the distances in this manner and calculating the inner value as described above makes it intuitively clear which facilities are contributing the most to a potential solution. Additionally, having zeros for inner values corresponding to client-facility neurons for inactive facilities is much simpler than storing large values (or infinity) to represent inactive client-facilities using the original distances.

Another modification that we made to the Hopfield network is related to how constraints are imposed on the network: In a typical Hopfield network, the neuron's activation values steer the network towards a lower energy function, where network states that violate the problem constraints add additional energy to the network. Our network does not do that; instead, we incorporate the constraint terms from Equation (4) directly into our neuron update operations. For example, instead of a client-facility neuron activating because its input crossing a threshold, it activates because it has the largest inner value in its column of the inner value matrix. Our network still transitions between different states and stabilizes at low energy feasible solutions for KMP, but we find that it converges on local optima in fewer iterations than the Hopfield networks from previous researchers.

In the next section we describe our synchronous algorithm that coordinates the neuron updates, enforces the problem constraints, and drives the network towards lower energy states.

4 The Modified Hopfield Network Algorithm

Our modified Hopfield network algorithm synchronously updates neurons in a series of rounds by applying a carefully constructed sequence of manipulations to the inner values. Algorithm 1 shows a high-level description of what occurs in each round of the algorithm. We start by initializing our modified Hopfield network to a feasible solution to KMP by randomly picking k of the facility neuron variables F_i and setting their values to 1. Then, in a loop, our algorithm (*i*) deactivates the active facility with the lowest inner value, (*ii*) updates the inner values of all the neurons, and (*iii*) activates the inactive facility with the highest inner value. If, after activating the next facility, the solution was not an improvement over the previous solution, our algorithm terminates. Hence, our algorithm is a single swap local search algorithm that transitions from feasible solution to feasible solution by removing one facility and adding one facility.

Algorithm 1 ModifiedHopfield(G, k)

```
Input: Graph $G = (V, E)$ and the number $k$ of facilities.
Output: A feasible solution for KMP.
Compute the normalized distance matrix $D'$.
Initialize $activationValue[][]$ with $k$ random active facilities.
Initialize $innerValue[][]$ to zero.
Initialize $stabilized$ to false.
Call $UpdateNeurons(activationValue, innerValue, D')$.
while $stabilized$ is $false$ do
    Let $prevEnergy$ equal the sum of the first column of $innerValue$.
    Deactivate the facility neuron $F_{low}$ with the lowest inner value.
    Call $FindBestFacility(activationValue, innerValue, D', F_{low})$.
    Activate the inactive facility $F_{high}$ with the highest inner value.
    Call $UpdateNeurons(activationValue, innerValue, D')$.
    Let $energy$ equal the sum of the first column of $innerValue$.
    if $prevEnergy \geq energy$ then
        $stabilized$ is $true$.
        $activationValue[low, 0] = 1$; $activationValue[high, 0] = 0$.
    end if
end while
Return the set of $k$ active facilities.
```

Recall that each neuron has two values, its activation value and its inner value. We store these values in the $n \times n + 1$ matrices $activationValue$ and $innerValue$, such that the first column of the matrix holds the values for the facility neurons and the remaining columns of the matrix hold the values for the client-facility neurons. The matrices can be interpreted so that row i begins with the facility neuron and continues with all of the client-facility neurons corresponding to that facility (all the clients that the facility could potentially serve), and column $j > 0$ contains all the client-facility neurons corresponding to that client (all neurons correspond to the same vertex in G and the column contains one neuron for each facility it might be served by).

The method $UpdateNeurons(activationValue, innerValue)$ is quite simple and updates the client-facility inner values, client-facility activation values, and facility inner values, in that order:

- **Updating the client-facility inner values:** For each client-facility neuron $x_{i,j}$, set $innerValue[i, j+1] = activationValue[i, 0] \times D'_{i,j}$.
- **Updating the client-facility activation values:** For the client-facility neuron $x_{i,j}$ in the column $j+1$ with the highest inner value, set $activationValue[i, j+1] = 1$; for the rest of the client-facility neurons in column $j+1$, set their $activationValue$ to 0.

- **Updating the facility inner values:** For the facility neuron F_i, set $innerValue[i,0] = \sum_{j=0}^{n} innerValue[i,j+1] \times activationValue[i,j+1]$.

Our modified Hopfield network algorithm transitions from feasible solution to feasible solution by deactivating a facility and then activating a different facility, so that after performing this facility swap operation there is k active facilities. When k facilities are active, observe that after calling method *UpdateNeurons*, updating the client-facility inner values produces non-zero inner values only for client-facility neurons within rows of $innerValue$ corresponding to active facilities, and hence if there were k active facilities prior to calling this method then there are exactly k rows of non-zero inner values after invoking the method. Then, updating the client-facility activation values enforces the constraint that each client is assigned to exactly one facility, as we only activate the single client-facility neuron in each column with the largest inner value, and also this enforces the constraint that exactly n clients are assigned to facilities. Note also that each client-facility neuron must be assigned to an active facility. Finally, updating the facility inner values assigns inner value to the active facility neurons corresponding to the sum of distances of assigned client-facility neurons, hence the sum of column 0 of $innerValue$ represents the energy of our modified Hopfield network, which we are trying to maximize. Therefore, these methods satisfy all the constraints from Equation (4).

When exactly k facilities are active, the inner values of the client-facility neurons are updated based on the distance from each client to each active facility. The client j is assigned to the facility closest to client j by identifying the client-facility neuron with the largest value in column $j+1$ of $innerValue$ and setting the corresponding cell in $activationValue$ to 1 and setting the rest of the values in that column of $activationValue$ to 0 (in the case of ties, the facility with the lowest index is assigned the client). Therefore, after updating the inner values of the facility neurons, a facility f_a with a lower inner value than a facility f_b means that the clients that f_a is serving are located further away from f_a than the distance between f_b and its clients.

We deactivate the facility neuron with the lowest inner value on line 10 of Algorithm 1 (in the case of ties, the facility with the lowest index is deactivated), as this suggests that it is serving clients the furthest away. Next, we show how to select the facility to activate.

4.1 Finding the Best Facility Neuron to Activate

Our algorithm for finding the best facility neuron to activate is shown in Algorithm 2. Algorithm 2 starts by setting all the inner values of the facility neurons to 0, as we will re-compute them in a specific way to select which facility we want to activate. In line 4, we zero out the inner values of client-facility neurons corresponding to the deactivated facility as those clients can no longer be served by that facility and hence one of the remaining $k-1$ active facilities will now have the highest inner value for serving those clients.

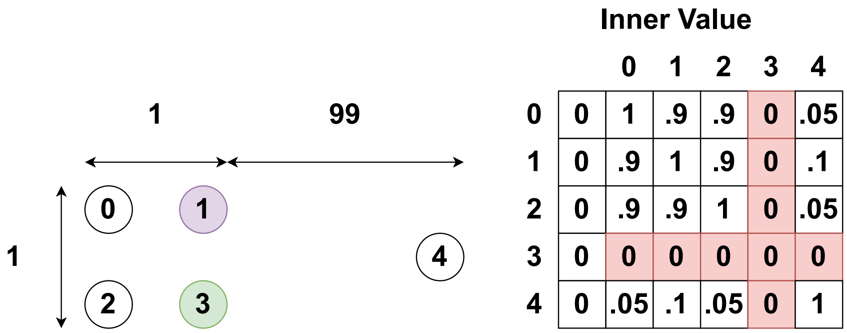
Algorithm 2 FindBestFacility($activationValue, innerValue, D', F_{low}$)

```
1: Input: The activation value matrix $activationValue$, the inner value matrix
   $innerValue$, the normalized distances $D'$, and the deactivated facility $F_{low}$.
2: Output: The inner values of the facility neurons are updated.
3: Set the facility neuron inner values to 0.
4: Set all values in row $low$ of $innerValue$ to 0.
5: Let $maxValues$ be an array holding the maximum inner value for columns $j > 0$.
6: Let $facilityValues$ be an array holding facility activation values where inactive
   facilities have value 1 and active facilities have value 0.
7: Compute the client-facility inner values using $facilityValues$.
8: Zero out columns in $innerValue$ corresponding to active facilities.
9: for each value $v$ in $innerValues$ from column 1 to $n + 1$ do
10:     Let $v$ be in row $i$ and column $j$.
11:     if $v > maxValues[j]$ then
12:         $v = v + (v - maxValues[j])$.
13:     else
14:         $v = 0$.
15:     end if
16:     $innerValues[i, j] = v$.
17: end for
18: for $i = 0$ to $n$ do
19:     $innerValues[i, 0] = \sum_{j=0}^{n} innerValues[i, j + 1]$.
20: end for
```

During calls to Algorithm 2 there are only $k - 1$ active facilities. Since these facilities will belong to the solution after finishing Algorithm 2, we focus on computing the inner values of client-facility neurons corresponding to inactive facilities. To identify the best candidate facility, we apply a series of manipulations to the inner value matrix according to three logical rules:

1. The $k - 1$ active facilities will serve themselves, and so we zero out the inner value of client-facility neurons in rows and columns corresponding to any of the $k - 1$ active facilities,
2. A candidate facility will only serve a client if it is closer to that client than the closest of the $k - 1$ active facilities, and so we zero out the inner value of client-facility neurons if their inner value is less than or equal to the largest inner value of the $k - 1$ active facilities in the same column, and
3. A candidate facility that is closer to a client than the closest active facility represents an improvement to the solution, and so we increase the inner value of client-facility neurons if their inner value is greater than the largest inner value of an active facility in the same column.

1) Facilities serve themselves. In lines 6-8 of Algorithm 2 the variable $facilityValues$ is set to 1 for inactive facilities and 0 for active facilities. When this modified $facilityValues$ is used to update $innerValues$, it creates zeros for the inner values of client-facility neurons corresponding to rows and columns of the remaining $k-1$ active facilities, since those facilities will serve themselves and



Inner Value

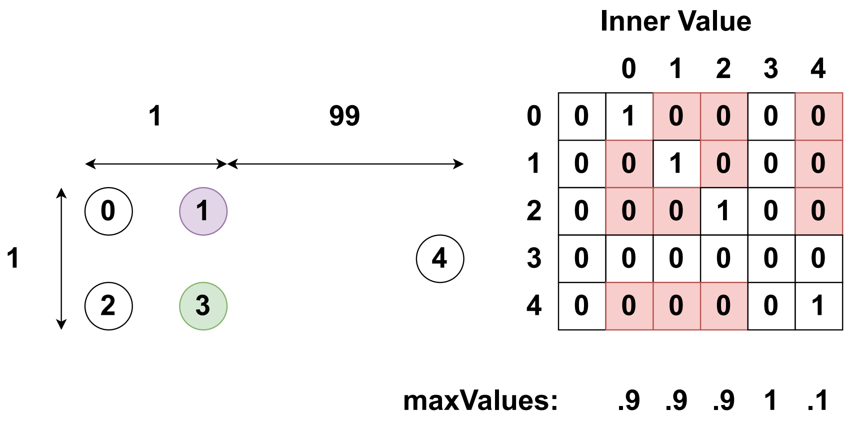
	0	1	2	3	4
0	0	1	.9	.9	0	.05
1	0	.9	1	.9	0	.1
2	0	.9	.9	1	0	.05
3	0	0	0	0	0	0
4	0	.05	.1	.05	0	1

Fig. 2: Continuing from the example in Figure 1, facility 1 had the lowest inner value and was deactivated. During Algorithm 2, the first column in $innerValues$ is set to 0, the inner values of the client-facility neurons are updated, and the inner values of client-facility neurons corresponding to any of the $k-1$ remaining active facilities are set to 0 (colored in red).

hence should not contribute inner value to any candidate facilities, and creates non-zero values for inner values in rows and columns corresponding to inactive facilities (see Figure 2). This means that potential facilities are evaluated only on how well they can serve the client locations.

2) Candidate facility underperforms active facility. In lines 11-15 of Algorithm 2 the inner value v of a client-facility neuron x in column j is compared against the maximum inner value from the remaining $k-1$ active facilities in column j, which was stored in the variable $maxValues[j]$ in line 5. If v is at most $maxValues[j]$, then the client-facility neuron x is not closer to facility i than it is to the remaining $k-1$ active facilities, and so assigning client j to facility i would not improve the solution. Hence, $innerValue[i,j]$ is set to 0 (see Figure 3). This means that a potential facility f is evaluated only based on the clients that would actually be assigned to f and not based on the closeness of f to its nearby clients.

3) Candidate facility outperforms active facility. In lines 11-15 of Algorithm 2 if v is greater than $maxValues[j]$, then the client-facility neuron x is closer to facility i than any of the remaining $k-1$ active facilities, and so assigning client j to facility i would improve the solution. By increasing the inner value by the difference between $innerValues[i,j]$ and $maxValues[j]$, the magnitude of the improvement is reflected in the updated inner value (see Figure 4). For example, in Figure 4 choosing to activate facility 4 is an improvement to the solution because facility 4 can serve itself at no cost, but in the previous solution facility 3 was serving facility 4 at a large cost (and this improvement takes into account the fact that client 1 now needs to be served by some facility at a non-zero cost).

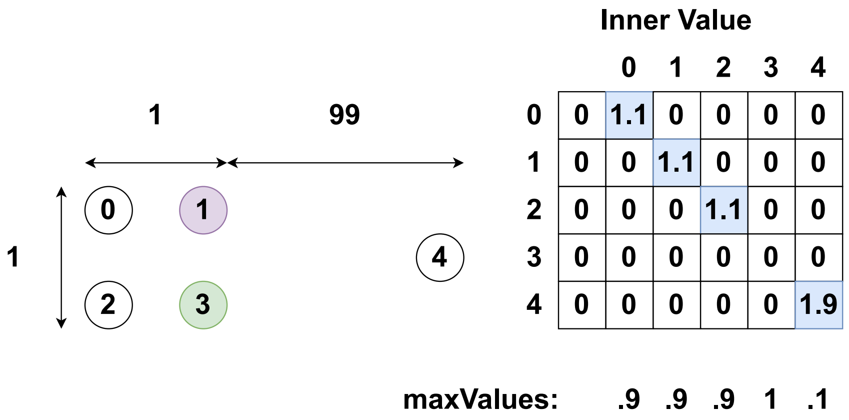


Inner Value

	0	1	2	3	4
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	1

maxValues: .9 .9 .9 1 .1

Fig. 3: Candidate facilities should not be assigned clients if they cannot serve them better than the remaining $k-1$ active facilities, and so the inner values of client-facility neurons in a column j that are less than or equal to the maximum inner value from an active facility in that column are set to 0 (colored in red).



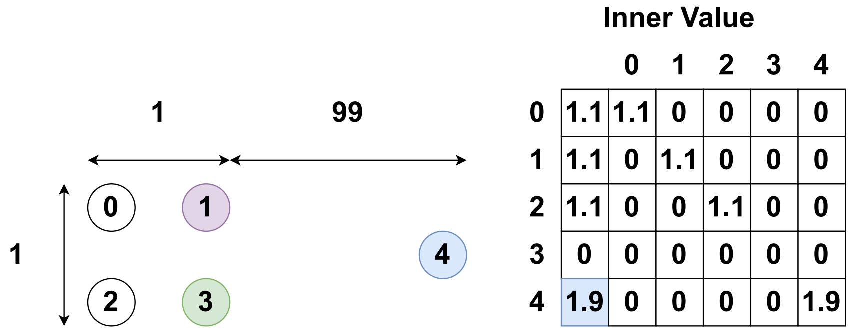
Inner Value

	0	1	2	3	4
0	0	1.1	0	0	0	0
1	0	0	1.1	0	0	0
2	0	0	0	1.1	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	1.9

maxValues: .9 .9 .9 1 .1

Fig. 4: Candidate facilities that are closer to a client than the closest active facility represent an improvement to the solution, and so the inner values of client-facility neurons in a column j that are greater than the maximum inner value from an active facility in that column are increased (colored in blue).

Finally, in lines 18-20 the inner value of a facility neuron F_i is set to the sum of the inner values of the client-facility neurons in row i (see Figure 5). Note that we do not take the dot product with the activation value matrix because we have not assigned clients to facilities yet.



		0	1	2	3	4
0	1.1	1.1	0	0	0	0
1	1.1	0	1.1	0	0	0
2	1.1	0	0	1.1	0	0
3	0	0	0	0	0	0
4	1.9	0	0	0	0	1.9

Fig. 5: After computing the inner values of the client-facility neurons corresponding to inactive facilities, the best candidate facility is the facility neuron with the largest inner value (colored in blue).

Note since we start with k active facilities, we only ever deactivate one facility at a time, and we always re-activate one facility, then at the end of each iteration of the algorithm there always are exactly k active facilities.

5 Experimental Results

We compared our modified Hopfield network with five other algorithms for KMP: Arya et al.'s single-swap local search algorithm [2], the local search algorithm of Pan and Zhu [28], the local search algorithm of Cohen-Addad et al. [8], the competition-based neuron network of Haralampiev [14–16], and the modified Hopfield network of Rossiter [32]. We give a brief description of how each of these algorithms work below.

Arya et al.'s Local Search. The algorithm begins with a random feasible solution for KMP. In the single-swap algorithm, for a given solution S a client i of S is considered for activation as a facility and each currently active facility j in S is considered for deactivation (so that one client is swapping roles with one facility). For each such swap, the cost of the resulting solution S' is calculated, and the best solution is selected. To ensure that this algorithm runs in polynomial time, we implemented the suggestion of Cohen-Addad and Mathieu [7] and terminate the algorithm if after considering swapping each client i with each facility in S no solution S' can be created such that $cost(S') \leq (1-1/n)cost(S)$. In the multi-swap algorithm, we randomly select two of the active facilities and two of the clients and swap them.

Pan and Zhu's Local Search. The single-swap local search algorithm of Arya et al. is run three times and the selected facilities from each solution are recorded. The set of common facilities F_c across the three solutions are identified, and a new reduced input is created that does not include the vertices from F_c or any of the clients whose closest facility is in F_c. The single-swap algorithm is run on this reduced input, and the facilities in the solution to this smaller problem are combined with F_c and used as the starting point for one more run of the single-swap algorithm.

Cohen-Addad et al.'s Local Search. Arya et al.'s local search algorithm is used, but for the purposes of calculating the cost of the solution, a new objective function is used: The cost associated with a particular facility i includes not only the clients closest to i, but also the clients whose second closest facility is i. This new objective function is used throughout the algorithm in order to select which swap operations to perform. We show results for both a single-swap and a multi-swap version of this algorithm.

Haralampiev's Competition-Based Neural Network. This neural network uses $2 \times n \times k$ neurons to represent KMP, where the first set C of $n \times k$ neurons represent each client being assigned to one of k clusters and the second set F of $n \times k$ neurons represent whether a facility is assigned to one of the k clusters. The set F is divided into subgroups of size n where each subgroup contains neurons for every vertex from the input but only one neuron from a subgroup can be active at a time (corresponding to the active facility for a particular cluster). The set C is divided into subgroups of size k, where only one neuron from a subgroup can be active at a time, and corresponds to the assignment of a client to a particular cluster. The facility neuron i assigned to cluster j takes input from all the client neurons that are also assigned to cluster j, using the distance between client and facility as the connection weight. With a high probability, the facility neuron from cluster j with the lowest value is activated (corresponding to the lowest cost assignment of clients to facilities); however, with low probability the algorithm can activate a facility that does not have the lowest value, which is done to help the network avoid getting stuck in local optimal solutions.

Rossiter's Modified Hopfield Network. Similar to our network, Rossiter uses $n + n^2$ neurons to represent KMP, where each neuron has both inner values and activation values. However, this network is initialized so that all facility neurons are *nearly* active with activation values that are arbitrarily close to 1, and the inner values and activation values of each of the client-facility neurons are updated based on these facility activation values. Repeatedly, a facility neuron is randomly chosen to be updated: If a facility neuron's inner value is in the top k facility inner values then its activation value is increased to 1, and otherwise its activation value is decreased to 0. The algorithm terminates when there are k active facilities.

5.1 Input Data

We used a variety of inputs obtained from network benchmarks for KMP, the traveling salesman problem, networks constructed from North-American cities, and randomly generated networks. We give a brief description of how each input was generated below.

Random-Small. We randomly generated networks where the number n of vertices was one of $[20, 30, 40, 50, 60, 70, 80, 90, 100]$ and the value of k was one of $[2, 3, 4, 5, 6, 7, 8, 9, 10]$. For each n and k pair we performed 100 tests.

Random-Large. We randomly generated networks where n was one of $[500, 600, 700, 800]$ and k was one of $[20, 30, 40, 50]$. For each n and k pair we performed 3 tests.

Cities USCA312. We used John Burkardt's USCA312 dataset [4] which provides a list of 312 cities across North-America. The dataset converts a city's longitude and latitude into an equivalent x and y coordinate pair which we used to construct the graph. We created tests from this dataset by randomly selecting n vertices, where n was any multiple of 10 between 30 and 300, and using k values between 2 and 10. For each n and k pair we performed 100 tests.

OR-Library P-Median. We used the dataset for KMP from the OR-Library [3] which contains 40 tests labeled pmed1 through pmed40. The n values range from 100 to 900 and the k values range from 5 to 200. For each n and k pair we performed a single test.

TSPLib. We used the TSPLib dataset [31] which contains inputs for the traveling salesman problem. Each input has over 1000 locations specified by x and y coordinates, and the dataset was adapted for KMP by García et al. [12]. There are several tests per problem with k values ranging from 5 to 5000.

For each test case that we report, we also provide the values of n and k. For inputs where the optimal solution was not known, we used the Coin-OR solver [33] to produce optimal solutions.

5.2 Testing Environment

We implemented the algorithms using python making extensive use of the PyTorch framework [29], which allows the efficient expression of matrices as tensors. This greatly improves performance as we structured most of the calculations as matrix operations which can be executed in parallel. The experiments were performed on a computer using an Intel Core i5-8300H 2.30GHz (4 cores) with 8GB of RAM.

5.3 Initialization Strategies

We begin by presenting the results of our modified Hopfield network using three different strategies to select the active facilities at the beginning of Algorithm 1. We use the results from these experiments to determine the best initialization strategy.

In Section 4 we described how to initialize our modified Hopfield network by activating k random facilities. We observed in our preliminary testing that our algorithm was very fast, and so our network could be re-initialized multiple times in order to perform multiple "runs" of the network using different sets of initial facilities each time, where the start point of a run is the initialization of the active facilities and the end point of a run is when a neuron update fails to improve the solution. Our motivation in developing our initialization strategies was to try and use previous solutions to identify valuable facilities to include in future runs of the algorithm. In Table 3 we provide results for three different initialization strategies, described below:

Random Initialization. This is the initialization strategy described in Section 4. Using this strategy, each run of our modified Hopfield network randomly selects k facilities to use as the starting point of the network. No information is shared between different runs of the algorithm. This initialization strategy can be viewed as the baseline by which our next two strategies can be compared against.

Tally Initialization. In this initialization strategy, we draw inspiration from the algorithm of Pan and Zhu [28], where they identify the common facilities selected from multiple solutions, construct a reduced problem that omits those common facilities and clients served by them, and find a set of good facilities for the reduced problem. While Pan and Zhu identify their common facilities over three runs of their algorithm, we instead maintain a frequency table that counts how many times each facility appears in a solution returned by our algorithm, as we found that in three runs our network often did not have any facilities common to all three solutions. Let r represent a parameter for the number of runs. We build the frequency table over the first r runs and remove the top $\lfloor k/4 \rfloor$ facilities to create the reduced problem. Then, we initialize a reduced version of the problem (using random sets of starting facilities from the remaining vertices) and solve it another r times, storing the set of facilities that produced the best solution to the reduced problem. Finally, we perform one last run initialized using the $3k/4$ best facilities from the reduced problem and the top $k/4$ facilities from the frequency table.

Best Facility Initialization. In this initialization strategy, our goal is to address a weak point of our algorithm: We designed a single-swap algorithm that can get stuck in local optimal solutions when improvements require swapping more than one facility. To strengthen our algorithm, each run is initialized using the facilities from our best solution produced so far (the very first run uses k random facilities), but we deactivate several of the facilities with the lowest inner values (we keep either $k-3$ or $\lfloor 0.9k \rfloor$ facilities, whichever is fewer, but keeping at least 1 facility). A random set of facilities is then activated to bring the total to k active facilities.

We include a selection of the results here[1]. In each table, the values in a single column correspond to the same algorithm, with the name of the algorithm listed at the top of the column. The rows are labeled with the type of input used and the n and k pair, followed by statistics for each algorithm.

For each test we report two statistics per algorithm: the approximation ratio and the total runtime in seconds. The approximation ratio is calculated as S/S^* where S is the value of the solution computed by an algorithm and S^* is the value of an optimal solution. The total runtime of a test is calculated by starting a timer when an algorithm begins initialization and ending the timer when a solution is returned. When there are multiple tests for a single n and k pair we return the mean average approximation ratio and average runtime of each algorithm for the n and k pair.

---

[1] The complete results are available at www.csd.uwo.ca/∼ablochha/ThesisHopfieldResults.zip

			40 Runs (random)		40 Runs (tally)		40 Runs (best)	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
random-small	20	5	1.02	0.05	1.03	0.13	**1.02**	**0.05**
	50	5	1.05	0.08	1.04	0.21	**1.04**	**0.08**
	100	5	1.05	0.13	1.05	0.35	**1.05**	**0.13**
random-large	500	50	1.14	1.04	1.12	2.89	**1.10**	**0.55**
	600	50	1.12	1.85	1.12	4.30	**1.06**	**0.89**
	700	50	1.13	2.50	1.12	6.04	**1.08**	**1.13**
	800	50	1.13	3.61	1.12	8.24	**1.07**	**1.58**
USCA312	100	10	1.22	0.12	1.22	0.36	**1.21**	**0.11**
	200	10	1.19	0.22	1.18	0.68	**1.16**	**0.21**
	300	10	1.19	0.30	1.19	0.93	**1.16**	**0.27**
pmed	500	167	1.48	0.66	1.51	2.56	**1.38**	**0.48**
	600	200	1.47	1.21	1.50	3.49	**1.34**	**0.72**
	800	80	1.13	3.37	1.13	11.99	**1.08**	**1.77**
	900	90	1.11	7.67	1.12	19.02	**1.08**	**2.32**
TSPLib	1304	500	1.55	20.21	1.52	61.76	**1.39**	**8.61**
	1400	500	3.08	**6.64**	**2.89**	16.76	3.21	6.77
	1432	500	1.07	56.73	1.09	115.08	**1.05**	**21.25**

Table 3: Initialization Strategies: Ratios and runtimes of our modified Hopfield network for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We use three different initialization strategies. The best performance for each test case is bolded.

The intuition behind the tally initialization strategy is that the most commonly occurring facilities across many solutions are valuable and might also belong to an optimal solution. Additionally, as seen in the results presented in this section, the approximation ratios produced by our modified Hopfield network are best when the value of k is smaller, and hence our network should be able to produce high-quality solutions for the reduced problem. The intuition behind the best facility initialization is to essentially allow our network to continue from a previous run after performing a multi-swap operation to remove the least valuable facilities.

We show the results of our three initialization strategies in Table 3, with the best approximation ratio and runtime for each test case bolded. The standard deviations of the approximation ratios are fairly similar for each initialization strategy at roughly 0.02. The standard deviations for each individual test case can be found in the full results.

It is clear that our best performing initialization strategy is best facility initialization, as it not only produces the lowest approximation ratios but also computes solutions in the least amount of time. In particular, for the tests on the large random graphs with $n = 800$ and $k = 50$, the best facility initialization had mean approximation ratios of 1.07 and runtimes of 1.15 seconds, compared to the random initialization with mean ratios of 1.13 and runtimes of 3.61 seconds. The tally initialization consistently performed the worst both in terms of approximation ratios and runtimes, with the exception of the input from the TSPLib with $n = 1400$ and $k = 500$ where its approximation ratio was the best, but we interpret this as just a random occurrence due to how each algorithm incorporates a random selection of facilities.

For all of the remaining test cases, we report results of our algorithm using the best facility initialization strategy.

5.4 Performing Multiple Runs

Each of the three initialization strategies described above benefits from performing multiple network runs: Random initialization can result in poor choices in starting facilities and so additional runs allow a variety of facilities to be selected, tally initialization benefits from performing multiple runs to find the most frequently selected facilities, and best facility initialization essentially performs a random multi-swap on the best solution seen so far which also benefits from trying multiple swaps.

Table 4 shows the results of running our algorithm differing amounts of times using the best facility initialization strategy. The standard deviations are as large as 0.09 when only 5 runs of the algorithm are performed, shrink to roughly 0.06 with 10 runs, and are reduced to roughly 0.04 and 0.02 when 20 and 40 runs are performed, respectively.

As expected, running the algorithm more times produces better results (see Table 4); however, for some of the test cases, such as the small random graphs, USCA312, and TSPLib, the improvement over performing 20 runs is minimal, while for the larger random graphs and the pmed dataset the improvements are still significant (an improvement of 0.03 for $n = 600$ and $k = 50$). Because our algorithm can perform 40 runs very quickly, we decided to use this number of runs to compare against the other algorithms instead of increasing the number of runs even further. As we see in the following tables, our algorithm is often still faster than the competing algorithms using 40 runs.

5.5 Comparing Against the Neural Networks

For each test using the neural networks, we allotted Rossiter's modified Hopfield network and Haralampiev's competition-based neural network 4 runs and the best solution found was returned, and we allotted our modified Hopfield network 40 runs as we previously determined this was enough runs to allow our best facility initialization strategy to improve the best found solutions. Note that our network completes its 40 runs faster than the other neural networks complete

			5 Runs		10 Runs		20 Runs		40 Runs	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
random-small	20	5	1.06	0.01	1.05	0.01	1.03	0.03	**1.02**	0.05
	50	5	1.09	0.01	1.07	0.02	1.05	0.04	**1.04**	0.08
	100	5	1.09	0.01	1.07	0.03	1.06	0.06	**1.05**	0.13
random-large	500	50	1.16	0.08	1.13	0.16	1.10	0.29	**1.10**	0.55
	600	50	1.15	0.10	1.09	0.23	1.09	0.42	**1.06**	0.89
	700	50	1.11	0.18	1.10	0.31	1.08	0.59	**1.08**	1.13
	800	50	1.11	0.26	1.11	0.38	1.09	0.80	**1.07**	1.58
USCA312	100	10	1.27	0.02	1.24	0.03	1.22	0.06	**1.21**	0.11
	200	10	1.21	0.03	1.20	0.05	1.17	0.10	**1.16**	0.21
	300	10	1.21	0.04	1.19	0.07	1.17	0.14	**1.16**	0.27
pmed	500	167	1.54	0.07	1.47	0.12	1.41	0.27	**1.38**	0.48
	600	200	1.50	0.11	1.40	0.20	1.36	0.38	**1.34**	0.72
	800	80	1.09	0.29	1.10	0.50	1.09	0.70	**1.08**	1.77
	900	90	1.13	0.39	1.11	0.56	1.10	1.16	**1.08**	2.32
TSPLib	1304	500	1.49	1.30	1.46	1.85	1.43	3.67	**1.39**	8.61
	1400	500	3.24	0.60	3.43	1.31	**3.12**	1.94	3.21	6.77
	1432	500	1.06	4.41	1.06	5.58	1.06	10.34	**1.05**	21.25

Table 4: Performing Multiple Runs: Ratios and runtimes of our modified Hopfield network for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets.

their 4 runs, and that Haralampiev's algorithm could not complete the TSPLib test cases within a reasonable timeframe.

Table 5 shows the results of performing tests on the neural networks, with the best approximation ratio and runtime for each test case bolded. While Haralampiev's algorithm produced many of the best approximation ratios for these selected test cases, the time it takes to perform only 4 runs of the algorithm is over a hundred times longer than the time used by our algorithm for many of the inputs. The prohibitively long runtime of Haralampiev's network prevented us from including its results on the TSPLib dataset. Despite the amount of extra time Haralampiev's network was allotted to solve each input, our algorithm managed to produce solutions with approximation ratios that were close and even a few solutions that were better (the larger pmed inputs).

Rossiter's algorithm computed solutions to the small random graphs the quickest (given that we performed ten times as many network runs), but as the size of the graph increased it required more time than our algorithm to produce its solutions, to the point where on the large random graphs with $n = 800$ and $k = 50$ Rossiter's algorithm's 4 runs took five times longer than our algorithm's 40 runs. For the largest input graph with $n = 13509$ and $k = 5000$, Rossiter's algorithm took 6 hours per run while our algorithm took only 30 seconds.

			Our Network (40)		Rossiter (4)		Haralampiev (4)	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
random-small	20	5	**1.02**	0.05	1.10	**0.02**	1.04	0.09
	50	5	1.04	0.08	1.07	**0.06**	**1.02**	0.41
	100	5	1.05	0.13	1.07	**0.12**	**1.01**	1.07
random-large	500	50	1.10	**0.55**	1.10	1.62	**1.09**	81.48
	600	50	**1.06**	**0.89**	1.11	2.63	1.06	127.00
	700	50	1.08	**1.13**	1.10	4.45	**1.06**	150.43
	800	50	1.07	**1.58**	1.09	8.08	**1.05**	192.40
USCA312	100	10	1.21	**0.11**	1.25	0.12	**1.16**	0.71
	200	10	1.16	**0.21**	1.21	0.32	**1.10**	2.45
	300	10	1.16	**0.27**	1.18	0.62	**1.10**	5.60
pmed	500	167	1.38	**0.48**	1.39	1.78	**1.31**	83.92
	600	200	1.34	**0.72**	**1.33**	2.67	1.36	89.69
	800	80	**1.08**	**1.77**	1.09	6.13	1.15	54.56
	900	90	**1.08**	**2.32**	1.10	10.67	1.14	67.08
TSPLib	1304	500	1.39	**8.61**	**1.38**	36.71	-	-
	1400	500	3.21	**6.77**	**1.81**	43.08	-	-
	1432	500	**1.05**	**21.25**	1.09	55.19	-	-
	13509	5000	1.96	**1364.86**	**1.69**	57219	-	-

Table 5: Comparing Against the Neural Networks: Ratios and runtimes of the solutions computed by the neural networks for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We allotted our algorithm 40 runs and the other neural networks 4 runs. The best performance for each test case is bolded.

5.6 Comparing Against the Local Search Algorithms

For each test using the local search algorithms, we allotted a runtime of 5 seconds on all but the largest graphs for the competing algorithms and continued to run our network 40 times. For the TSPLib inputs with 1000 vertices, we allotted the local search algorithms 100 seconds, and for the input with 13509 vertices, we allotted the local search algorithms 10000 seconds, in order to give these algorithms more time than our algorithm took. Given the current solution S, if the local search algorithms consider all possible swap operations and fail to find a new solution of cost at most $(1 - 1/n)cost(S)$, they will terminate before their given time limit.

We compared both the single-swap and multi-swap (we allowed two facilities to swap at once) algorithms of Arya et al. and Cohen-Addad et al. The multi-swap algorithms have much higher runtimes, so in order to complete the tests in a reasonable time we allotted these algorithms a maximum of one second on the small random graphs and the USCA312 dataset. In the tables, we divide these results using a slash; for example, the result 1.06/1.03 indicates that the single-swap algorithm produced solutions with mean approximation ratio 1.06 and the multi-swap algorithm produces solutions with ratio 1.03.

Table 6 shows the results of performing the tests using the local search algorithms, with the best approximation ratio and runtime for each test case bolded. For all but the USCA312 dataset, our modified Hopfield network outperformed the single-swap local search algorithms. When the local search algorithms were allowed to swap out two facilities at a time, Arya et al's multi-swap algorithm produced approximation ratios lower than ours on a number of the test cases, and the performance of the multi-swap algorithms were significantly better on some of the larger TSPLib inputs.

			Our Network (40)		Arya (s/m)		Pan		Cohen-Addad (s/m)	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
random-small	20	5	**1.02**	0.05	1.06/1.03	**0.01**/1.00	1.07	0.03	1.08/1.04	0.02/1.00
	50	5	1.04	0.08	1.04/**1.01**	**0.03**/1.00	1.03	0.11	1.04/1.03	0.08/1.00
	100	5	1.05	0.13	1.02/**1.01**	**0.09**/1.00	1.02	0.29	1.03/1.03	0.29/1.00
random-large	500	50	1.10	**0.55**	1.14/**1.07**	5.00/5.00	1.12	4.84	1.19/1.10	5.00/5.00
	600	50	**1.06**	**0.89**	1.20/1.09	5.00/5.00	1.16	3.08	1.20/1.09	5.00/5.00
	700	50	**1.08**	**1.13**	1.21/1.09	5.00/5.00	1.17	5.96	1.23/1.10	5.00/5.00
	800	50	**1.07**	**1.58**	1.22/1.11	5.00/5.00	1.24	3.14	1.22/1.11	5.00/5.00
USCA312	100	10	1.21	**0.11**	**1.03**/1.03	0.24/1.00	1.03	0.97	1.04/1.05	0.58/1.00
	200	10	1.16	**0.21**	**1.02**/1.04	0.58/1.00	1.02	2.80	1.03/1.05	1.45/1.00
	300	10	1.16	**0.27**	**1.01**/1.04	1.48/1.00	1.01	4.31	1.03/1.05	2.75/1.00
pmed	500	167	1.38	**0.48**	1.58/**1.11**	5.00/5.00	1.34	8.35	1.55/1.11	5.00/5.00
	600	200	1.34	**0.72**	1.52/**1.09**	5.00/5.00	1.40	9.90	1.62/1.12	5.00/5.00
	800	80	**1.08**	**1.77**	1.30/1.08	5.00/5.00	1.22	13.46	1.35/1.09	5.00/5.00
	900	90	**1.08**	**2.32**	1.36/1.09	5.00/5.00	1.23	15.81	1.39/1.08	5.00/5.00
TSPLib	1304	500	1.39	**8.61**	2.16/**1.20**	100/100	1.79	123.67	2.02/1.21	100/100
	1400	500	3.21	**6.77**	3.10/**1.47**	100/100	3.23	129.42	3.35/1.50	100/100
	1432	500	1.05	**21.25**	1.43/**1.03**	100/100	1.29	130.49	1.40/1.05	100/100
	13509	5000	1.96	1364.86	2.34/1.84	1000/1000	2.25	4337	2.32/1.89	1000/1000

Table 6: Comparing Against the Local Search Algorithms: Ratios and runtimes of the solutions computed by the local search algorithms for a selection of the test results for the randomized, North-American cities, k-median, and traveling salesman problem datasets. We allotted our algorithm 100 runs and the other local search algorithms 5 seconds. The best performance for each test case is bolded. For the algorithms of Arya et al. and Cohen-Addad et al., both the single-swap (s) and multi-swap (m) results are listed.

5.7 Approximation Ratio Versus Number of Facilities

When looking at Table 5, we noticed that for the large random graphs the neural networks all consistently performed better when the number of vertices increased but the number of facilities stayed the same. We noticed the same trend for the USCA312 inputs as well. To observe the impact of the number of facilities, we include in Tables 7 and 8 one last set of results using the inputs from USCA312 where $n = 30$.

			Our Network (40)		Rossiter (4)		Haralampiev (4)	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
USCA312	30	2	**1.02**	0.04	1.08	**0.02**	1.02	0.02
	30	3	**1.05**	0.05	1.12	**0.03**	1.05	0.03
	30	4	**1.08**	0.05	1.16	**0.03**	1.13	0.04
	30	5	**1.09**	0.06	1.23	**0.03**	1.13	0.05
	30	6	**1.12**	0.06	1.31	**0.03**	1.23	0.05
	30	7	**1.22**	0.06	1.44	**0.03**	1.39	0.06
	30	8	**1.14**	0.06	1.29	**0.03**	1.29	0.06
	30	9	**1.25**	0.06	1.57	**0.03**	1.46	0.07
	30	10	**1.27**	0.06	1.68	**0.03**	1.51	0.08

Table 7: Approximation Ratio Versus Number of Facilities: Ratios and runtimes of the solutions computed by the neural networks for a selection of the test results from USCA312. The best performance for each test case is bolded.

			Our Network (40)		Arya (s/m)		Pan		Cohen-Addad (s/m)	
Test	n	k	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)	Ratio	Time (s)
USCA312	30	2	1.02	0.04	1.02/**1.02**	**0.01**/1.00	1.02	0.02	1.03/1.03	0.02/1.00
	30	3	1.05	0.05	1.04/**1.02**	**0.01**/1.00	1.04	0.03	1.05/1.03	0.02/1.00
	30	4	1.08	0.05	1.04/**1.02**	**0.01**/1.00	1.04	0.04	1.06/1.03	0.03/1.00
	30	5	1.09	0.06	1.05/**1.02**	**0.01**/1.00	1.05	0.06	1.06/1.03	0.04/1.00
	30	6	1.12	0.06	1.05/**1.02**	**0.02**/1.00	1.04	0.08	1.05/1.03	0.04/1.00
	30	7	1.22	0.06	1.04/**1.02**	**0.03**/1.00	1.04	0.10	1.06/1.04	0.07/1.00
	30	8	1.14	0.06	1.06/**1.02**	**0.02**/1.00	1.05	0.10	1.08/1.04	0.06/1.00
	30	9	1.25	0.06	1.06/**1.02**	**0.03**/1.00	1.05	0.12	1.08/1.04	0.09/1.00
	30	10	1.27	0.06	1.06/**1.02**	**0.04**/1.00	1.05	0.13	1.09/1.05	0.10/1.00

Table 8: Approximation Ratio Versus Number of Facilities: Ratios and runtimes of the solutions computed by the local search algorithms for a selection of the test results from USCA312. The best performance for each test case is bolded. For the algorithms of Arya et al. and Cohen-Addad et al., both the single-swap (s) and multi-swap (m) results are listed.

As shown in Table 7, all of the neural network approaches produce worse approximation ratios when the number of facilities increases. Note that when $n = 30$ and $k = 2$, our algorithm produced the same mean approximation ratios as Haralampiev's network of 1.02, but when k was increased to 10, our algorithm performed significantly better with mean ratios of 1.27 compared to Haralampiev's 1.51. For this particular set of inputs, our modified Hopfield network outperformed the other neural networks.

As shown in Table 8, the single-swap local search algorithms produce worse approximation ratios when the number of facilities increases, but this same effect is not nearly as pronounced (or even existent) in the multi-swap algorithms. Arya et al.'s multi-swap algorithm produced the best results for these tests, but note that a single run of the algorithm (which was terminated due to hitting the time limit) took a hundred times longer than Arya et al.'s single-swap algorithm (which terminated before the time limit due to not finding any significant improvements).

6 Analysis

We provide observations on the performances of the different algorithms on the inputs and the parameters that we chose for our algorithm. While interpreting the results, please note that even though our runtime might sometimes exceed that of the other algorithms, it is because we perform many more runs than the other algorithms.

6.1 Initialization Strategies

In Table 3 we presented results of our modified Hopfield algorithm using a variety of strategies for choosing the initial active facilities during each run of our algorithm. In our first initialization strategy we simply randomly chose k facilities. Due to the speed of our algorithm, our network was able to complete many runs in the same time as the competing algorithms could complete far fewer runs. However, this random initialization strategy did not take advantage of the solutions from previous runs to help improve the future runs. To attempt to use information from previous network runs to aid the future runs, we tried two more initialization strategies: The tally initialization strategy and the best facility initialization strategy.

The intuition behind the tally initialization strategy is threefold: (*i*) the most frequently appearing facilities from solutions produced after random initializations are likely valuable facilities that might belong to optimal solutions, as suggested by Pan and Zhu [28], and so these facilities are desirable and should be included in future runs of the algorithm; (*ii*) the most frequent facilities from our frequency table represent facilities that were part of a local optimal solution, and by selecting several of these facilities and combining them with a different set of other facilities, we can increase the chances of breaking through the local optima; and (*iii*) our algorithm performs better when the number k of facilities is smaller, and so once the most frequent facilities from the tally have been removed to create the reduced problem, our algorithm should find solutions to the reduced problem with good approximation ratios.

In practice, the tally initialization strategy did not perform as well as we had hoped. When Pan and Zhu tried their approach, all of the facilities in their set of common facilities appear together in each solution they produced; in contrast, we do not know whether the most frequently selected facilities appeared alongside any of the other frequently selected facilities. It could, for example, be the case that two frequently selected facilities are actually very close to each other and either one or the other of these two facilities appear in every solution but both facilities should not be active in the same solution, and yet our algorithm might select them both because they are both so frequently selected. We believe that in order for this strategy to become valuable, we might need to change how we determine which facilities to select.

The intuition behind the best facility initialization strategy is similar: The best solution found so far was a local optimal solution for which our algorithm could not improve any further with its neuron update function, and hence by including several facilities with the top inner value from the best solution combined with a set of random facilities, we can attempt to break through the local optima. Essentially, this initialization strategy allows our network to perform a single multi-swap operation between runs where multiple facilities with the lowest inner value are swapped out and a random set of facilities are swapped in.

Table 3 shows that this strategy almost always produced the best approximation ratios out of all the proposed strategies. This is likely due to reducing some of the randomness of selecting facilities and instead trying to prioritize selecting facilities that were previously shown to be valuable. Moreover, this initialization strategy produced solutions faster than the other strategies; the speed increase is due to a smaller number of iterations being performed before reaching a local optima, likely because the initial set of facilities in each run includes many facilities that were part of a different local optimal solution.

We notice an outlier approximation ratio on the TSPLib input with $n = 1400$ and $k = 500$, where the ratio (3.21) is significantly higher than the rest of the test cases. We believe what happened is that the "best facilities" that were being re-used in each run of the network were actually not very good facilities at all, but because the graph is fairly large, our algorithm was able to keep generating improved solutions (even if only minimally improved) using the same set of "best facilities" and therefore was unable to make a better set of initial facilities. We could avoid this type of result in the future by requiring improvements to pass a specific threshold, otherwise we could re-randomize the initial facilities in the next run.

Since the best facility initialization produced the lowest approximation ratios and runtimes, we used this strategy for the remaining test cases, but we believe that other techniques could be applied to improve the approximation ratio of our algorithm.

6.2 Performing Multiple Runs

Since our algorithm can complete a single run very quickly, we are able to perform more runs of our network than the competing algorithms within the same timeframe. The speed of our algorithm is a core strength of our modified Hopfield network, and taking advantage of the speed allows our algorithm to compete against the other algorithms by learning from previous runs.

The results in Table 4 show that running our algorithm multiple times and taking the best solution results in lower approximation ratios. We tried performing up to 100 runs of the network, but the benefit to the approximation ratio started to decline after performing 40 runs, so we decided to compare the results of our algorithm using 40 runs against the competing algorithms.

We note that while our algorithm did not benefit very much from adding additional runs past 40, this is related to the specific initialization strategy used to assign the starting facilities. For example, we previously described an addition to the best facility initialization strategy where if a significant improvement is not found, then the next run will be initialized using random facilities. This would allow the algorithm to pursue a group of facilities but later abandon them if it cannot improve the solution further using that initial group. A strategy like this would require more runs to be effective, and so we could consider performing network runs until a solution is produced that is better than some threshold t of our choosing.

6.3 Comparing Against the Neural Networks

As seen in the previous tables, the runtimes of our Modified hopfield network are very low. The greatest strength of the modified Hopfield network of Rossiter [32] was its speed, and our algorithm is already ten to fifty times faster depending on the size of the input. We attribute the speed of our network to the small number of iterations required to converge to a local optimal solution. For example, for a graph with $n = 500$, our algorithm terminates after $5 - 30$ iterations while Rossiter's algorithm terminates after $2000 - 3000$ iterations.

Haralampiev's algorithm produces solutions that are closest to the optimal for some of the test cases, but as shown in Table 7 there are many test cases in which our algorithm performs significantly better. Haralampiev takes significantly more time to produce solutions than any of the other algorithms that we tested. For the smaller inputs that we used, such as random-small and USCA312 where $n < 300$, the runtime of Haralampiev's network only takes a few seconds and therefore makes for a good choice of algorithm; however, for larger inputs where $n > 500$ the runtime of Haralampiev's network becomes significantly slower than the modified Hopfield networks. We did not even run Haralampiev's network on the TSPLib dataset because the tests could not be completed within several days of runtime.

We briefly discuss how some of the characteristics of each algorithm might influence their performance.

Haralampiev. Haralampiev's network flips the activation value of a facility according to the temperature parameter, which can allow the algorithm to explore beyond solutions that would otherwise produce local optima for the modified Hopfield networks. This feature of the algorithm allows it to produce solutions that are much closer to optimal solutions, but it causes an increased number of iterations of the algorithm because sometimes the network transitions to solutions with higher costs before eventually finding lower cost solutions. This strategy is effective for avoiding getting stuck in local optima but has a drastic increase on the runtime of the algorithm.

Rossiter. Rossiter's modified Hopfield network runs significantly slower than our modified Hopfield network, and we attribute this to the number of iterations required for a single run of the network to reach a local optimal solution. As described previously, the facility neurons of Rossiter's modified Hopfield network are initialized arbitrarily close to 1 and the algorithm terminates when there are k facility neurons with activation value 1. In the worst case, all n of Rossiter's facility neurons can have their activation values first increased to 1, for n iterations, before $n - k$ facility neurons have their activation values decreased to 0, for a total of $2n - k$ iterations. Moreover, Rossiter's algorithm randomly chooses which facility to update in each iteration and sometimes a facility is chosen but its activation value is not modified, and hence the algorithm sometimes performs many multiples of n iterations before stabilizing. This large number of iterations causes the runtime of Rossiter's network to be significantly larger than our network, as our network converges to local optimal solutions in less than 50 iterations even when $n > 1000$.

The approximation ratio of Rossiter's modified Hopfield network can be arbitrarily large. When Rossiter's network is first initialized, all of the facility neurons are essentially active and hence the facilities all serve themselves to achieve an inner value of 1. As previously described, when a facility neuron is updated, its activation value is increased to 1 if its inner value is in the top k inner values; however, in the first iteration of the algorithm, since all of the facility neurons have the same inner value, the first facility to be activated is essentially randomly chosen. Since Rossiter includes no mechanism to deactivate a facility neuron, this can result in approximation ratios that are arbitrarily large. This might explain some of the results where Rossiter's approximation ratios are significantly higher than ours, such as on the traveling salesman problems (see the Appendix for more results).

6.4 Comparing Against the Local Search Algorithms

The local search algorithms can also quickly produce solutions close to the optimal when the values of n and k are small, with the algorithm of Pan and Zhu achieving the best approximation ratios when comparing only the single-swap algorithms. When two facilities can be swapped out in a single operation, the algorithm of Arya et al. outperforms the algorithms of Pan and Zhu and Cohen-Addad et al. While our modified Hopfield network frequently outperforms the single-swap algorithms in using much less time, it is itself outperformed by the much slower multi-swap algorithms.

We briefly discuss how some of the characteristics of each algorithm might influence their performance.

Arya. Arya et al.'s algorithm will exhaustively search all combinations of client and facility swaps if the improvement threshold is not met, which means that, depending on which swap operations are improving, this algorithm can spend a lot of time without actually transitioning to another solution. This is more often seen on the multi-swap version of the algorithm, where the algorithm terminated due to reaching the time limit, while the single-swap version of the algorithm often terminated due to not finding any more significant improvements. When only a single facility is being swapped out, there are nk possible combinations of swaps, but when two facilities are swapped out, there are n^2k^2 possible combinations of swaps.

Pan. Pan's algorithm consistently produces solutions that are closest to the optimal out of the three basic local search algorithms implemented. This is likely due to the fact that Pan's algorithm correctly identifies high value facilities that appear in multiple local optima. This algorithm runs slower than Arya's algorithm due to having to run Arya's algorithm five times: Three times to create the list of common facilities, a fourth time on the reduced problem where the common facilities are excluded, and a fifth time using the common facilities plus the solution to the reduced problem.

Cohen-Addad. This algorithm consistently performs the worst of the three basic local search algorithms implemented. The runtime is likely increased by solving a more complicated objective function, which may cause it to perform fewer iterations within the timeframe we allotted it for our experiments.

Our Modified Hopfield Network. Our algorithm converges to local optimal solutions in a small number of iterations. Intuitively, our algorithm attempts to identify the best single-swap operation during each iteration, and so it makes sense that it should outperform the algorithm of Arya et al., which simply tries many combinations of swaps and accepts the first significant improvement that it finds. However, since the results showcase our algorithm when it only swaps a single facility at a time, our network gets stuck in more local optimal solutions than the multi-swap algorithms.

6.5 Single-Swap Versus Multi-Swap

Arya et al. proved that single-swap local search algorithms that produce solutions that cannot be further improved by any single-swap have approximation ratio at most 5 and multi-swap local search algorithms that cannot be further improved by any p-swap have approximation ratio at most $(3 + \frac{2}{p})$, where p is the number of facilities swapped out in one swap operation; so it can be expected that the multi-swap algorithms would perform better in practice. Our results confirm this intuition: Arya et al.'s multi-swap algorithm outperformed its single-swap version and the single-swap algorithms of Pan and Zhu and Cohen-Addad et al. Additionally, Arya et al.'s multi-swap algorithm frequently outperformed our modified Hopfield network.

Local search algorithms that can only swap out a single facility per operation can transition to solutions in which no further improvements can be found by changing only a single facility. Such solutions are not necessarily optimal; solutions exist for which the only improvements differ by two or more facilities. Therefore, single-swap local search algorithms can get stuck in local optimal solutions that multi-swap local search algorithms could improve upon. This is especially noticeable in Table 8 for the USCA312 inputs with only 30 vertices, as Arya et al.'s single-swap algorithm consistently terminates in only a fraction of a second due to not finding significant improvements, but Arya et al.'s multi-swap algorithm is able to consistently find solutions with better approximation ratios.

The performance of the multi-swap algorithms is even more noticeable on some of the TSPLib inputs, as shown in Table 6, where the multi-swap algorithm produces solutions with better approximation ratios compared to the solutions produced by the single-swap algorithm. However, the multi-swap local search algorithm of Arya et al. requires a significant amount of extra time. The algorithm is not complicated, it simply randomly selects two facilities to deactivate and randomly selects two clients to activate; however, there are many combinations of choosing two facilities and the algorithm might have to evaluate many sets before it finds an improvement.

6.6 Approximation Ratio Versus Number of Facilities

Tables 7 and 8 show that the neural networks and the single-swap local search algorithms all produce solutions with worse approximation ratios when the number of facilities is increased while the multi-swap local search algorithms tend to produce equally good solutions no matter the number of facilities.

Our modified Hopfield network deactivates the facility with the lowest inner value, as this suggests that the facility is the least valuable to the solution, but it is not necessarily true that the best single-swap operation results from deactivating this particular facility. For example, consider an input which contains a large cluster and a single outlier vertex that is located far away from the cluster. It is reasonable to believe that in an optimal solution the outlier should be a facility that serves itself but is not assigned any other clients. This facility might appear as the facility with the lowest inner value in our algorithm and deactivating it might not find any replacement facilities that improve the solution; however, it is possible that swapping one of the facilities located in the cluster could lead to an improvement, even though they did not have the lowest inner value in our network. Hence, our algorithm might become stuck in local optimal solutions even when single-swap operations still exist that could improve the solution. In contrast, Arya et al.'s single-swap local search algorithm simply tries many combinations of swap operations, and in the above example it is reasonable to think that this algorithm could find a better solution than our modified Hopfield network.

We believe that when the number k of facilities increases, the probability of our algorithm getting stuck in local optimal solutions, where the facility with the lowest inner value is not the best facility to swap out, also increases. Future improvements to the way in which a facility neurons inner value is calculated could lead to performance improvements in these situations.

7 Conclusion

In the k-median problem we are given n locations and want to select k locations such that the total distance between each unselected location and its nearest selected location is minimized. We present a modified Hopfield network for KMP and compare it against several local search and neural network algorithms.

We empirically show that our modified Hopfield network converges to local optimal solutions very quickly; the time needed to perform a single run of our algorithm is the least compared to all of the algorithms implemented, and this enabled us to perform many more runs of our algorithm than the competing algorithms for each test case, while still maintaining comparable total runtimes. Even though both Rossiter's algorithm and our algorithm are both modified Hopfield networks that use some similar techniques (such as inner value), our algorithm consistently produced solutions with better approximation ratios using less time as compared to Rossiter's algorithm. Haralampiev's algorithm was able to produce better solutions than us for some of the test cases but required much more time to do so, such as the large random graphs and the USCA312 inputs with larger numbers of vertices, but our algorithm was able to produce better solutions for other test cases.

When compared against the single-swap local search algorithms, our modified Hopfield network often produced solutions with better approximation ratios using less time, except on the USCA312 dataset. When we allowed Arya et al.'s local search algorithm to swap out two facilities at a time, this multi-swap algorithm began to outperform our algorithm on a number of the test cases. In some instances, such as the large graphs from the TSPLib dataset, the multi-swap algorithm produced ratios less than 1.5 while our algorithm produced ratios more than 3.0.

Given the increase in solution quality observed when allowing the local search algorithms to swap out multiple facilities at a time, we believe that our algorithm could be greatly improved by allowing it to swap out multiple facilities. Careful consideration will be required to implement an efficient multi-swap operation, as after deactivating the worst facilities our current algorithm relies on the *inner value* of the facility neurons to select replacements, and attempting to activate multiple facilities based on their inner value can be deceiving: Two candidate facilities might receive inner value for serving each other or for both serving a particular client, but if they were both to be activated they should serve themselves and a particular client can only be assigned to one of them.

Simple modifications to Algorithms 1 and 2 transform our algorithm into a multi-swap local search algorithm. To perform a p-swap, where p facilities are swapped with clients, the algorithm is modified as follows: (i) the p active facilities with the lowest inner value are deactivated; (ii) the inner values of the inactive candidate facilities are calculated as previously described in Algorithm 2; (iii) one-at-a-time, the inactive facility f with the highest inner value is recorded, its distance to each client it would potentially serve is used to update the array $maxValues$, the rows and columns of the inner value matrix corresponding to f are zeroed-out, and the inner values of the client-facility neurons are re-calculated, until p candidate facilities have been selected; and (iv) if activating the p facilities does not produce a better solution then the previous set of facilities is restored and the algorithm terminates, otherwise the algorithm continues.

By activating the p facilities with the highest inner value one-at-a-time, updating the array $maxValues$ between activating each facility, and re-calculating the inner values of the client-facility neurons, we solve the issue where the inner value of multiple facility neurons could include serving the same client (or facilities serving each other). The benefit of performing a multi-swap operation in our modified Hopfield network is that we will not need to try multiple swap operations before accepting a new set of facilities to activate; rather, correctly updating the neuron's inner values will inform our algorithm exactly which facilities should be selected in the new solution. Because we update the values of our neurons using efficient matrix calculations, the runtime of our algorithm should not increase by much in order to perform these multi-swaps. We believe this strategy can perform multi-swap operations faster than the multi-swap local search algorithm of Arya et al., because Arya et al.'s multi-swap algorithm is quite simple and might try a potentially very large number of swap operations before accepting a new solution.

References

1. Andrews, M. and Zhang, L.: The access network design problem. In Proceedings of the 39th Annual Symposium on Foundations of Computer Science, pp. 40-49. 1998.
2. Arya, V., Garg, N., Khandekar, R., Meyerson, A., Munagala, K., and Pandit, V.: Local search heuristic for k-median and facility location problems. In Proceedings of the Thirty-Third annual ACM Symposium on Theory of Computing, pp. 21-29. 2001.
3. Beasley, J.: OR-Library: Distributing test problems by electronic mail. Journal of the Operational Research Society, **41**(11), pp. 1069-1072. 1990.
4. Burkardt, J.: Cities-city distance datasets. https//people.math.sc.edu/Burkardt/datasets/cities/cities.html. 2011.
5. Byrka, J., Pensyl, T., Rybicki, B., Srinivasn, A., and Trinh, K.: An improved approximation for k-median and positive correlation in budgeted optimization.
6. Christofides, N. and Beasley, J.: A tree search algorithm for the problem p-mediates. European Journal of Operational Research, **10**, pp. 196-204. 1982.
7. Cohen-Addad, V. and Mathieu, C.: Effectiveness of local search for geometric optimization. In Proceedings of the 31st International Symposium on Computational Geometry. 2015.

8. Cohen-Addad, V., Gupta, A., Hu, L., Oh, H., and Saulpic, D.: An improved local search algorithm for k-median. In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1556-1612. 2022.
9. Cohen-Addad, V., Esfandiari, H., Mirrokni, V., and Narayanan, S.: Improved approximations for euclidean k-means and k-median, via nested quasi-independent sets. In Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing, pp. 1621-1628. 2022.
10. Cohen-Addad, V., Grandoni, F., Lee, E., and Schwiegelshohn, C.: Breaching the 2 LMP approximation barrier for facility location with applications to k-median. In Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 940-986. 2023.
11. Domínguez, E. and Muñoz, J.: A neural model for the p-median problem. Journal of Computers and Operations Research, **35**(2), pp. 404-416. 2008.
12. García, S., Labbé, M., and Marín, A.: Solving large p-median problems with a radius formulation. INFORMS Journal on Computing, **23**(4), pp. 546-556. 2011.
13. Hakimi, L.: Optimum distribution of switch centers in a communication network and some related graph theoretic problems. Journal of Operations Research, **13**(3), pp. 462-475. 1965.
14. Haralampiev, V.: Neural networks for facility location problems. Annual of Sofia University St. Kliment Ohridski, Faculty of Mathematics and Informatics, **106**, pp. 3-10. 2019.
15. Haralampiev, V.: Theoretical justification of a neural network approach to combinatorial optimization. In Proceedings of the 21st International Conference on Computer Systems and Technologies, pp. 74-77. 2020.
16. Haralampiev, V.: Neural network approaches for a facility location problem. Journal of Mathematical Modeling, **4**(1), pp. 3-6. 2020.
17. Hopfield, J.: Neural networks and physical systems with emergent collective computational abilities. In Proceedings of the National Academy of Sciences, **79**(8), pp. 2554-2558. 1982.
18. Hopfield, J. and Tank, D.: "Neural" computation of decisions in optimization problems. Journal of Biological Cybernetics, **52**(3), pp. 141-152. 1985.
19. Jain, K., Mahdian, M., and Saberi, A.: A new greedy approach for facility location problems. In Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing, pp. 731-740. 2002.
20. Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L.: On the placement of internet instrumentation. In Proceedings of IEEE INFOCOM 2000, pp. 26-30. 2000.
21. Kariv, O.: An algorithmic approach to network location problems. Part 1: the p-centers. SIAM Journal of Applied Math, **37**(3), pp. 513-538. 1979.
22. Li, S. and Svensson, O.: Approximating k-median via pseudo-approximation. In Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, pp. 901-910. 2013.
23. Lozano, S., Guerrero, F., Onieva, L., and Larrañeta, J.: Kohonen maps for solving a class of location-allocation problems. European Journal of Operational Research, **108**(1), pp. 106-117. 1998.
24. Mangasarian, O., Fayyad, U., and Bradley, P.: Mathematical programming for data mining: formulations and challenges. Microsoft Technical Report. 1998.
25. Merino, E. and Perez, J.: An efficient neural network algorithm for the p-median problem. Lecture Notes in Computer Science, pp. 460-469. 2002.
26. Merino, E., Perez, J., and Jerez-Aragonés, J.: Neural network algorithms for the p-median problem. ESANN, pp. 385-392. 2003.

27. Mishrra, S. and Barman, S.: Cost reduction techniques in p-median method using artificial neural network. JSM Bioinformatics, Genomics, and Proteomics. 2016.
28. Pan, R. and Zhu, D.: An efficient local search algorithm for k-median problem. In Proceedings of the 6th WSEAS International Conference on Applied Computer Science, pp. 19-24. 2006.
29. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems, 32. 2019.
30. Peng, X., Xia, X., Zhu, R., Lin, L., Gao, H., and He, P.: A comparative performance analysis of evolutionary algorithms on k-median and facility location probelsm. Journal of Soft Computing, **22**(23), pp. 7787-7796. 2018.
31. Reinelt, G.: TSPLIB-A traveling salesman problem library. ORSA Journal on Computing, **3**(4), pp. 376-384. 1991.
32. Rossiter, C.: A modified hopfield network for the k-median problem. Master's thesis, Western University. 2023.
33. Saltzman, M.: COIN-OR: An open-source library for optimization. Programming Languages and Systems in Computational Economics and Finance. 2002.
34. Teitz, M. and Bart, P.: Heuristic methods for estimating the generalized vertex median of a weighted graph. Journal of Operations Research, **16**(5), pp. 955-961. 1968.

Chapter 7

7 Conclusions

In this thesis we investigated three specific research objectives which resulted in approximation algorithms for the high multiplicity strip packing problem (HMSP) and thief orienteering problem (ThOP), and a heuristic for the k-median problem (KMP). In the case of the approximation algorithms, our work represents the first known approximation algorithms for these problems.

Below we provide a summary of the results in Section 7.1 and discuss future research directions for each problem in Section 7.2.

7.1 Summary

In Chapter 3 we gave an algorithm for HMSP for the case when $K = 3$ that produces solutions requiring at most height $\frac{3}{2}h_{max} + \epsilon$ plus the height of an optimal solution, where ϵ is any positive constant and h_{max} is the height of the tallest rectangle, and for the case when $K = 4$, an algorithm that produces solutions requiring at most height $\frac{7}{3}h_{max} + \epsilon$ plus the height of an optimal solution. Additionally, we gave an algorithm for the K-type problem that produces solutions requiring at most height $\lfloor \frac{3}{4}K \rfloor + 1 + \epsilon$ plus the height of an optimal solution. These are the current best-known results for HMSP.

In Chapter 4 we proved that there exists no approximation algorithm for ThOP with constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$, and we presented a PTAS for ThOP when the input graph G is directed and acyclic that produces solutions that use time at most $T(1 + \epsilon)$ for any constant $\epsilon > 0$. We also presented a FPTAS for ThOP on arbitrary undirected graphs where the travel time depends only on the lengths of the edges and the time limit T is the length of a shortest path from s to t plus a constant K. Finally, we presented a FPTAS for a restricted version of ThOP when the input graph G is a clique. These are the first approximation algorithms that have been designed for ThOP.

In Chapter 5 we gave an algorithm that transforms instances of ThOP on arbitrary undirected outerplanar graphs into equivalent instances of ThOP on DAGs in polynomial time, and another algorithm that transforms instances of ThOP on arbitrary undirected series-parallel graphs into equivalent instances of ThOP on DAGs in polynomial time. These algorithms allow our PTAS from Chapter 4 to produce solutions for undirected outerplanar graphs and undirected series-parallel graphs that use time at most $T(1 + \epsilon)$.

In Chapter 6 we presented a modified Hopfield network for KMP that uses a local search approach. By using a simple neuron update function based on the inner value metric, our algorithm quickly produces solutions that are close to optimal. We experimentally evaluated our algorithm against several neural networks and local search algorithms and demonstrated that our algorithm was the fastest and produced solutions with competitve approximation ratios. Our algorithm was outperformed, with respected to approximation ratios, by the multi-swap local search algorithms; however, we propose future improvements to our algorithm so that it can also swap multiple facilities in a single swap operation.

7.2 Future Work

We conclude this thesis by presenting some future work for each of the topics studied. In Section 7.2.1 we talk about our work on high multiplicity strip packing, in Section 7.2.2 we talk about our work on thief orienteering, and in Section 7.2.3 we talk about our work on Hopfield networks.

7.2.1 HMSP

In this section we discuss improvements to the algorithms presented in Chapter 3, discuss limitations of the approaches that we used, and describe additional algorithms and experiments that could be performed.

Improvements to the Algorithm for the Three Type Problem

In Chapter 3 we described how we solved HMSP by first solving FSP and then transforming fractional rectangles into whole ones. We showed a trivial algorithm that simply replaces all fractional rectangles with whole ones, shifting surrounding rectangles upwards as necessary, that produces solutions of height at most $LIN(I) + K + \epsilon$, where ϵ is a positive constant and $LIN(I)$ is the height of an optimal fractional solution. When there are at most three rectangle types, we improved upon this with a simple algorithm that produces solutions of height at most $LIN(I) + \frac{5}{3} + \epsilon$, and then we showed more complicated techniques that can be used to produce solutions of height at most $LIN(I) + \frac{3}{2} + \epsilon$. Since it is easy to see that there are some inputs for which the best rounding of the fractional solution produces a height of $LIN(I) + 1 + \epsilon$, a natural question is whether we can improve our algorithm further.

Perhaps the best way to attempt these improvements is to carefully consider the cases of our algorithms and try to find improvements that work for a single case. If such an improvement can be found, we should investigate whether that approach would work on some of the remaining cases, and if not, then we should identify the limitations of our approach. In such a way, we could either (1) prove that there exists specific cases that can not be improved, which shows that our algorithm is tight, or (2) find an improvement to our algorithm.

More specifically, we wish to discover whether some variation of our algorithm can produce solutions of height at most $LIN(I) + 1 + \epsilon$. For example, we currently round up all fractional rectangles in S_{Case3}, in which the sum of all fractional values is at least $\frac{3}{2}$; however, an improved algorithm would be restricted to only rounding up all fractional rectangles if the sum of their fractional values is at least 2, so that the height increase of rounding up these fractional rectangles is at most 1.

Making this change to the definition of S_{Case3} is the easy part; the hard part is determining how to round the fractional rectangles in the vertical sections whose fractional values sum between 1 and 2. Some possible approaches include attempting to guarantee that the empty space left behind by fractional rectangles packed in C_{A1} sums to at least 1, or ensuring that we only round up "sufficiently large" rectangles. It is entirely possible that our algorithm cannot be improved, but we would like to be able to prove that.

Improvements to the Algorithm for the Four Type Problem

We described an algorithm for HMSP for the case when $K = 4$ that produces solutions of height at most $LIN(I) + \frac{7}{3} + \epsilon$, where ϵ is a positive constant and $LIN(I)$ is the height of an optimal fractional solution. This algorithm was a natural extension our first simple algorithm for HMSP for the case when $K = 3$ in the sense that (1) in S_{Case1} all fractional rectangles are packed in C_{A1}, (2) in S_{Case2} the bottom fractional rectangles are rounded up and the top fractional rectangles are packed in C_{A1}, and (3) in S_{Case3} all fractional rectangles are rounded up. Note that this algorithm does not use any of the empty space left behind when fractional rectangles are re-located to C_{A1}.

Therefore, we can try to improve upon this algorithm by borrowing a few ideas from our improved three type algorithm. Here are how the ideas might be adapted to the four type problem:

- Divide the algorithm into several algorithms that assume a given number of rectangle types in each configuration. For example, one algorithm could assume that C_1 has four rectangle types, C_2 has four rectangle types, C_3 has three rectangle types, and C_4 has one rectangle type. Another algorithm could assume that each configuration has three rectangle types, which again is similar to our improved algorithm for the three type problem.
- Leverage the existence of a ”wide” leftmost rectangle, that is at least partially contained within S_{Case2} or S_{Case3}, in order to calculate how much empty space is left behind when fractional rectangles are re-located to C_{A1}. Infer properties of the packing, such as the presence or absence of vertical sections in a particular case, if such a “wide” rectangle does not exist.

It is reasonable to think that we can use some of the existing empty space, and that the height of this empty space might be as large as $\frac{1}{3}$. In this case, we could design an algorithm that produces solutions of height strictly less than $LIN(I) + \frac{7}{3} + \epsilon$, which would be an improvement.

Improvements to the Algorithm for the K Type Problem

We described an algorithm for HSMP when K is constant. Recall that a simple algorithm could produce a solution to HMSP of height at most $LIN(I) + K + \epsilon$, where ϵ is a positive constant and $LIN(I)$ is the height of an optimal solution to FSP, and our K-type algorithm produces solutions of height at most $LIN(I) + \lfloor \frac{3}{4}K \rfloor + 1 + \epsilon$.

Our best algorithm for $K = 4$ produces solutions of height at most $LIN(I) + \frac{7}{3} + \epsilon$. If, for cases of HMSP when $K > 4$, we simply applied this algorithm on four of the possible configurations and rounded up the remaining configurations (simply replace fractional rectangles with whole rectangles, shifting other rectangles as necessary), then this algorithm would produce solutions of height at most $LIN(I) + (K - \frac{5}{3}) + \epsilon$ for all $K \geq 4$, which is strictly an improvement over the simple algorithm described above.

Recall that in the algorithms we described in the previous chapters, we sometimes take fractional rectangles whose fractional values sum to at most 1 and pack them in a single region of height 1. Intuition suggests that we could take fractional rectangles whose fractional values sum to at most 2 and pack them in two regions that each have height 2;

however, this is not necessarily the case. Depending on the dimensions and multiplicities of the fractional rectangles, it is possible that the fractional rectangles can not be packed evenly in the two regions. More specifically, the situation might arise where the only way to pack all of the rectangles into the two regions is for one of the rectangles to be split with one of its pieces packed in the first region and its other piece packed in the second region. Note that in this situation there is no way to ensure an integer packing.

Therefore, we can not take the easy approach to the K type problem by defining this case: ”if the sum of the fractional values is at most $\frac{K}{2}$ then pack the fractional pieces in $\frac{K}{2}$ regions that each have height 1”, together with the case: ”if the sum of the fractional values is more than $\frac{K}{2}$ then round up the fractional rectangles”; such an algorithm could produce solutions of height at most $LIN(I) + \frac{K}{2} + \epsilon$ for any even value for K. Unfortunately, this algorithm could not guarantee that the solutions are always integer packings.

Moreover, it is even difficult to pack fractional rectangles whose fractional values sum to at most $\frac{3}{2}$ into two regions that each have height 1. Note that if we attempt to pack the fractional rectangles into too many regions, then the height increases by too much. For example, when $K = 4$, if we take a group of fractional rectangles whose fractional values sum to at most 2 and pack them into into four regions that each have height 1, then the height would increased beyond what our current algorithm achieves.

However, perhaps it would be possible to borrow ideas from our algorithm for the three type problem and make use of the empty space left behind when small fractional rectangles are moved. Perhaps we could prove that as K gets larger the empty space left behind after moving small fractional rectangles becomes larger as well, which would lead to a good algorithm for the K type problem.

K Type Problem Constrained to a Fixed Number of Configurations

Recall from Chapter 3 that when there are K rectangle types, the solution output by the linear program might have any number of configurations ranging from 1 to K. So, we could consider a variant of the problem where K is large, but the number of configurations is, for example, only three. While this makes the problem easier than the more general K type problem, note that is still more complicated than the regular three type problem.

The algorithm for the three type problem that produces solutions of height at most $LIN(I) + \frac{5}{3} + \epsilon$ [11] would work for this variant; however, our improved algorithm for the three type problem would not work, because it makes assumptions about the structure of the packing (such as each configuration having at most two rectangle types) that would not hold true if $K > 3$. These assumptions are critical because they allow us to calculate the size of the empty space left when small fractional rectangles are moved to C_{A1}. When we allow more than three rectangles types to be packed in the three configurations, there could be many more vertical divisions that separate different vertical sections within S_{Case1} as compared to the three type problem, which negatively interferes with the amount of empty space available for C_{A1}.

However, it is a good exercise to consider this variant of the problem, because it forces us to design an algorithm that does not make assumptions about how many rectangle types exist in the input. This is an important step towards building an algorithm for the K type problem, as we actually can not extend our best three type algorithm beyond $K = 3$, even if we use the strategy described above where the first three configurations are rounded as in our improved three type algorithm and the remaining configurations are rounded according to the simple algorithm.

Integrality Gap of the Linear Program

Recall from Chapter 3 that we defined a linear program that represents the fractional strip packing problem and that an optimal solution to the fractional strip packing problem might have smaller height than the corresponding optimal integer solution. This is because the fractional solution might horizontally cut some rectangles and thus reduce the height by spreading the fractional pieces out width-wise. The *integrality gap* refers to the maximum difference of heights between an optimal fractional solution and its corresponding optimal integer solution.

It is easy to construct an instance of HMSP such that the difference in heights between an optimal fractional solution and its corresponding optimal integer solution is just shy of 1: consider an input consisting of exactly one rectangle whose height is 1 and whose width approaches 0. An optimal integer solution must pack this rectangle as-is resulting in a height of 1, but an optimal fractional solution can horizontally cut the rectangle into fractional pieces whose heights approach 0 and pack them side-by-side resulting in a height that approaches 0. Therefore, the difference in heights of these two packings approaches 1, which is a lower bound on the integrality gap. With our improved algorithm for the three type problem we have established an upper bound on the integrality gap, as our algorithm produces solutions of height at most $LIN(I) + \frac{3}{2} + \epsilon$.

Note that the problem of closing the integrality gap is related to our work on improving our algorithm for the three type problem: if we can prove that the integrality gap is $\frac{3}{2}$, then we have also proved that our algorithm is tight and can not be improved. In contrast, if we can improve our algorithm for the three type problem to produce solutions of height at most $LIN(I) + 1 + \epsilon$, then we have also proved that the integrality gap is 1. Since we do not currently know how to improve our algorithm, and since we do currently know how to prove the integrality gap, then we have two different directions to try pursuing in order to achieve the same goal.

Experimental Evaluation of the Algorithms

We have mentioned several times throughout this thesis that high multiplicity algorithms generally perform better than their non high multiplicity counterparts, but this is simply an observation based on the approximations ratios of these algorithms. There is value in conducting experimental evaluations that compare not only different high multiplicity algorithms for the same problem against each other, but also high multiplicity algorithms against their non high multiplicity counterparts.

We implemented our algorithms, as described in Chapter 3, using the commercial integer and linear program solver CPLEX by IBM to solve the fractional strip packing problem. This allowed us to make our first observations for the three type problem, such as the difference in running times between a simple algorithm and our improved algorithm and the difference in solutions between our algorithm and an optimal solution.

Next, we can implement several of the algorithms for the non high multiplicity strip packing problem, such as the algorithm by Harren et al. [79]. By performing experiements for a variety of values of K, we can determine whether our three type algorithm outperforms the algorithm of Harren et al. and we can better analyze our K type algorithm.

Three-Dimensional High Multiplicity Strip Packing

A natural step beyond HMSP is to consider the same problem in three dimensions. In the *three-dimensional high multiplicity strip packing problem* (3DHMSP), the length and width of the container are fixed and the goal is to minimize the height needed to pack all cuboids from the input. This variant of the problem models physical object packing much better than the two dimensional version of the problem.

3DHMSP can be approached in a similar way as HSMP: by rounding the solution output by a linear program. Recall that for HMSP we rounded solutions obtained from solving the fractional strip packing problem. These solutions consisted of configurations, which in turn consist of base configurations.

3DHMSP can be relaxed to the *three-dimension fractional strip packing problem* (3DFSP), which permits horizontal cuts on the cuboids; a solution to 3DFSP might have a lower height due to the possibility of packing fractional pieces in regions where whole rectangles cannot fit. A linear program can be formulated for 3DFSP similarly to the linear program for FSP shown in Chapter 2: all cuboids from the input must be packed in exactly one configuration each, all configurations used in the solution to 3DFSP must have a positive height, all configurations' lengths and widths must be at most the length and width of the container, and the objective function must minimize the total heights of the configurations.

Therefore, a solution to 3DFSP consists of a set of configurations. A *base configuration* C_j consists of a multi set of cuboid types whose total width is at most 1 and whose total length is at most 1, i.e., a base configuration is a set of cuboids that can be packed in the container without needed to put any of the cuboids on top of another cuboid. A group of cuboids following a base configuration can be stacked on top of each other so that any horizontal plane parallel to the base of the container across any part of the group will intersect the same multiset of cuboid types. A vertical line drawn across any part of the configuration will intersect either only cuboids of the same type, or empty space. A fractional solution might include configurations whose uppermost cuboids have been cut on a plane that is parallel to the base. The height of a vertical line intersecting cuboids of a configuration is called the height of the configuration.

If an optimal solution to such a linear program can be found in polynomial time, then we might be able to adapt our algorithm for HMSP to work for 3DHMSP. Some points of concern include the difficulty of finding a base configuration that fits in a 1 x 1 bin.

High Multiplicity Rectangle Packing

We reviewed some of the related work on the rectangle packing problem in Chapter 2, but we could not find any literature on the *high multiplicity rectangle packing problem*

(HMRP). HMRP has the following definition:

Definition Given K distinct rectangle types, where each rectangle type Ti has n_i rectangles each with width $0 < w_i \leq 1$ and height $0 < h_i \leq 1$, and a rectangular container of width W and height H, pack the maximum subset of the rectangles into the rectangular container, without rotating or overlapping any of the rectangles.

Like the other high multiplicity problems mentioned previous, the input to HMRP is compact: it can be described using a list of only $3K + 2$ numbers: the width w_i, height h_i, number n_i of rectangles of each type T_i, and the width and height of the rectangular container.

We suggest an approach to solving it based on our work on HMSP: we start by solving the corresponding FSP. The constraints of the linear program must be modified such that the rectangles packed in each configuration have width at most that of the rectangular container. Note that a solution to FSP might need to be adjusted in two ways to become a feasible solution to HMRP: (1) any fractional rectangles will have to be transformed into whole ones and (2) the height of a solution to FSP might be greater than the height of the rectangular container.

We can use rounding techniques similar to what we described in Chapter 3. If the resulting packing does not fit within the rectangular container, several strategies could be used:

- A careful consideration of the structure of the packing, as in the work of Harren et al. [79], could be used to shift rectangles within the rectangular container until they fit.
- A configuration could be selected to be removed from the rectangular container or specific rectangles could be selected to be removed from the rectangular container so that all remaining rectangles fit.

Alternatively, consideration could be put towards selecting a different formulation for the linear program and other rounding schemes.

High Multiplicity Scheduling Problems

Note the similarity between packing problems and scheduling problems. Some scheduling problem on identical machines can be represented as a series of rectangular containers (one for each machine) with fixed width and unbounded length, and the input jobs are rectangles that need to be packed in the containers. The length of the packing within one of these containers represents the amount of time needed to process the jobs scheduled on the corresponding machine, and the maximum length from any of these containers represents the time needed to complete all of the jobs.

Observe that when these containers are oriented vertically and placed side-by-side, they resemble the strip packing problem. However, a clear distinction between scheduling problems and the strip packing problem exists: if a rectangle's width is larger than the width of a rectangular container representing a machine, then the implication is that the job represented by that rectangle must be scheduled on two or more machines in parallel. Therefore, depending on the definition of the scheduling problem under consideration, all of the rectangles in the input might have the same width (if no jobs can be scheduled in parallel on multiple machines). Such a problem essentially becomes a one-dimensional problem.

Similarities are often drawn between scheduling problems and the bin packing problem, as the rectangular containers representing the machines are similar to bins. However, there is a critical difference: in the bin packing problem the goal is to minimize the number of fixed-sized bins, but in some scheduling problems there is one bin per machine, each with unbounded height, and the goal is to minimize the maximum height of the packing across all of the bins.

Therefore, while many similarities exist between scheduling problems, strip packing problems, and bin packing problems, scheduling problems are actually a distinct set of problems. Moreover, there are a large number of different scheduling problems based on whether the jobs are identical, the machines are identical, the jobs take differing amounts of time on different machines, certain jobs must be sequenced in particular orders, and many other factors.

One way for a scheduling problem to have a high multiplicity encoding is when there are relatively few distinct job lengths (encoded as rectangle heights) in the input, but there are many other ways to make a high multiplicity scheduling problem such as having few distinct machine processing powers, few distinct delayed start times, and so on. Our work on HMSP might be applicable to some variants of high multiplicity scheduling problems.

7.2.2 Thief Orienteering Problem

In this section we discuss improvements to the algorithms presented in Chapters 4 and 5, discuss limitations of the approaches that we used, and describe additional algorithms and experiments that could be performed.

Exceeding the Time Limit

In Chapter 4, the PTAS that we presented produces solutions that use time at most $T(1+\epsilon)$. While this exceeds the time limit, the problem is still interesting because many practical applications will allow the time limit to be exceeded in exchange for a profit penalty; for example, in the case of ThOP modeling a vehicle routing problem where the vehicle is collecting goods from different locations, the vehicle might be late to service future customers and hence those customers could be issued a refund. However, a future area of research could investigate whether the PTAS could produce solutions without exceeding the time limit.

The PTAS that we designed exceeds the time limit because in order to keep the size of the profit table polynomial we need to discard some tuples. Two tuples that have the same rounded profit and rounded weight are equivalent with respect to the carrying capacity of the knapsack, but because travel time is dependent on the true weight of the tuple, these two tuples might differ in terms of their travel times.

If we were able to design a metric that could determine which of the tuples with the same rounded profit and rounded weight is objectively the best, we would not need to exceed the time limit. Alternatively, if we were able to prove that we only need to keep a polynomial number of tuples that all share the same rounded weight and rounded profit, we would not need to exceed the time limit. Finally, another approach could be to decide on a specific metric by which to keep only a single tuple and update the analysis to bound the amount of profit lost by missing out on items due to traveling too slow with respect to the specific metric used.

Transforming Additional Graph Classes to DAGs

In Chapter 5, we transformed instances of ThOP on undirected outerplanar and series-parallel graphs into equivalent instances of ThOP on DAGs. Our strategy involved adding vertices and edges and using the structures of outerplanar and series-parallel graphs to inform us on how many additional vertices and edges are needed and which vertices can be incident on each other. For example, our algorithm for transforming instances of ThOP on undirected outerplanar graphs into equivalent instances of ThOP on DAGs created additional vertices depending on the number of chords incident on a vertex. In contrast, our algorithm for transforming series-parallel graphs creates DAGs with at most twice as many vertices as the input graph.

Future research can investigate additional classes of graphs and take advantage of their structures in order to transform them into DAGs while creating only a small number of additional vertices and edges. Each additional graph class that we can successfully transform increases the usefulness of our PTAS from Chapter 4. Beyond investigating special kinds of graphs, we also aim to investigate approximation algorithms for ThOP on arbitrary undirected graphs.

Experimental Evaluation of the Algorithms for Industrial Challenges

In Chapter 5, we described two applications that can be modeled using ThOP on series-parallel graphs: production optimization and systems reliability. Many researchers currently perform experimental evaluations of their algorithms for these applications on well-known benchmarks. We could implement both our PTAS from Chapter 4 and our series-parallel transformation from Chapter 5 and perform experimental evaluations to see how our approach compares to the existing literature.

Because our PTAS currently exceeds the time limit T, we would need to decide how best to remove items from the solutions produced by our algorithm in order to reduce the time. A simple approach could remove items with the smallest profit until we find a feasible solution, but this might not be the best way to reduce the total travel time. Future research could investigate how best to keep the largest possible profit of the solutions if we need to reduce their total travel time.

ThOP Variants

The orienteering problem has multiple popular variants: The team orienteering problem, the orienteering problem with time windows, and the time dependent orienteering problem. Future research can investigate these variants for ThOP.

The team thief orienteering problem considers the problem where there are M thiefs that all need to select a path from s to t and each item can only be collected by a single thief. The goal in the problem is to produce a solution where all of the thiefs travel to t within the time limit, none of the thiefs exceed their knapsack's carrying capacity, and the total profit among all of the thiefs is maximized. This adds complexity because in addition to building the profit table as before, an algorithm would need to consider placing a particular item into each of the M thief's knapsacks.

The thief orienteering problem with time windows considers the problem when items can only be collected from a particular vertex during a predefined time window. The thief can choose to travel to a vertex and wait until the items become available, but if the thief arrives after the window closes the items will not be collectable. This adds addtional complexity, because the profit table will need to contain more tuples to represent the possibilitity that the thief chooses to (*i*) not wait at vertex u and (*ii*) wait at vertex u until the time window opens.

The time dependent thief orienteering problem considers the problem when the travel time across edge (u, v) changes depending on what time it is. For example, this models the practical scenario of traffic congestion on highways during particular times of the day. For each edge (u, v), the distance $d_{u,v}$ is predetermined based on the departure time of vertex u. This adds complexity because now it might be more efficient for the thief to wait at a particular vertex u in order to achieve a better departure time, and that requires keeping more tuples in the profit table.

Each of the above problem variants are well-studied with respect to the orienteering problem, and would be valuable directions for future research on ThOP.

Approximation Algorithms for TTP

As shown in Chapter 2, there are currently no approximation algorithms for TTP. This is likely due to an inability to approximate TTP within a constant factor unless $\mathsf{P} = \mathsf{NP}$. However, using the techniques developed in Chapter 4, we could investigate whether it is possible to design a PTAS for a restricted version of TTP. Then, similar to what we did in Chapter 5, we could aim to generalize the PTAS. Finally, we could implement these algorithms and compare them against the existing heuristics for TTP.

7.2.3 Modified Hopfield Network for K-Median

In Chapter 6, we presented our modified Hopfield network algorithm that swaps out one facility at a time to transition from feasible solution to feasible solution. Our algorithm was arguably the best performing neural network and also the best performing single-swap local search algorithm, but the multi-swap local search algorithms of Arya et al. and Cohan-Addad et al. produced solutions with better approximation ratios (albeit with increased runtimes). We have identified several areas in which we can improve our algorithm in order to make it more competitive.

Swapping Multiple Facilities

Our algorithm currently deactivates a single facility and then, given that there are $k-1$ facilities still active, finds the best facility to activate. Performing only single swaps sometimes prevents our algorithm from being able to access a global optimal solution, because it is possible that from the current network state that two facilities would need to be deactivated and two inactive facilities would need to be activated in order to reach a global optimal solution, but by only deactivating either of them it could be possible that none of the other inactive facilities could be activated to improve the solution. In this case, our algorithm, after deactivating one of the active facilities, would re-activate that same facility again and then terminate, instead of transitioning to the global optimal solution. In other words, the two facilities would have to be swapped at the same time.

Without adjusting our current algorithm, if we deactivated two facilities and attempted to find two replacements, then two candidate facilities that are currently deactivated and nearby to each other could possibly both get high inner values, which might suggest that we could activate them both, but it could be the case that once one of them is activated that the other facility neuron's inner value decreases. This can occur because they were both able to serve the same clients (and serve each other), but once one of these facility neurons is activated the other facility neuron will no longer receive any inner value unless it can actually serve the clients at a lower cost.

To avoid this issue, we propose the following modifications to the neuron update function. After deactivating the q facilities with the lowest inner values, identify the replacement facilities one-at-a-time: (*i*) after identifying the first (and best) replacement facility f using our previous algorithm, the array $maxValues$ should be updated to reflect any clients that are best served by f; (*ii*) the rows and the columns of the inner value matrix corresponding to f should be zeroed-out (to avoid selecting f again); and (*iii*) the inner values of the client-facility neurons corresponding to the remaining candidate facilities should be re-calculated given that f will be an active facility in the next solution.

These modifications will increase the runtime of our algorithm slightly, but not by the amount observed by the multi-swap algorithm of Arya et al. Because we implement our algorithm using efficient matrix calculations, and since the inner values inform our algorithm which facilities to activate, we do not need to try large numbers of swap operations like the simpler local search algorithms. Changing our algorithm to a multi-swap algorithm might help us improve its approximation ratio from 5 to $(3+\frac{2}{p})$, where p is the number of facilities swapped in on swap operation, if we can prove that when our algorithm terminates there does not exist any remaining p-swaps that could improve the solution.

Improving the Inner Value Metric

We created the *inner value* metric to represent how valuable a facility is in terms of how close it is to its clients. In our modified Hopfield network we deactivate the facility with the lowest inner value; intuitively, we want to replace the facility that is least valuable to the solution with a more valuable facility. When k facilities are active the inner values of the facility neurons are calculated as:

$$innerValue[i,0] = \sum_{j=0}^{n} innerValue[i, j+1] \times activationValue[i, j+1]$$

However, this definition of inner value does not lead to the facility with the lowest inner value being the least valuable to a solution. Consider a graph that has a large cluster of vertices that are close together and a single vertex located far away from the cluster. Let there be several facilities active in the cluster and let the single distant vertex also be an active facility f. It is easy to imagine that the facilities in the middle of the cluster might have high inner values (much larger than 1) and facility f might have an inner value of exactly 1 (because it only serves itself). Therefore, our algorithm would deactivate f because it has the lowest inner value.

In this example, there are several active facilities within the cluster and so a candidate facility f' to replace f would also be located in the cluster and hence any improvement to the solution would likely be small, because all of the clients within the cluster are already assigned to facilities that are nearby. However, because facility f is so distant from the cluster, the cost of the solution will be much higher if f is a client being served by a facility within the cluster, and hence our algorithm would reactivate f. Since f was deactivated and then re-activated, the swap operation would not improve the solution and so our algorithm would reach a local optima and terminate. Observe that it is entirely possible that an improvement could be found by changing which facilities are active within the cluster, as long as f remains active as well.

To address this issue, the inner value metric needs to consider the cost of replacing a facility. Facility f had a low inner value because it did not serve any additional clients, but deactivating f has a large cost to the solution because none of the other facilities can serve it cheaply. We can borrow from the techniques of Cohen-Addad et al. [37] and identify the second-closest facility for each client. Under the new definition, the inner value of f would be 1 (for serving itself) plus the difference between 1 and the distance to the second-closest facility. Note that we used a similar technique in our neuron update function, but we need to apply it again when determining which facility to deactivate.

Adding this improvement to the algorithm should cause our network to get stuck in fewer local optimal solutions and to find more improving swap operations, leading to better approximation ratios. Since we plan to transform our algorithm into a multi-swap local search algorithm, this improvement might also allow us to get close to the approximation ratio of Cohen-Addad et al. [37].

Improving the Neuron Update Function

When our modified Hopfield network has k active facilities, we assign clients to their closest facility. If a client is equally close to multiple facilities, we assign it to the facility with the lowest index. Note that this tie-breaking rule might produce unintended results. Consider that facility f_1 and f_2 are both equally close to client c_1, but facility f_2 serves another client c_2 and c_2 is closer to f_2 than to f_1. When breaking ties using the lowest index, client c_1 is assigned to facility f_1 and hence f_1 and f_2 both serve themselves and a single additional client each. However, it is easy to imagine that a better solution might be to have facility f_2 serve both c_1 and c_2; this would cause facility f_1 to have a lower (and more accurate) inner value and so the algorithm might replace it with a more valuable facility.

The intuition behind our improved tie-breaking strategy is to assign clients to the facility that is already serving the most clients. The difficulty of this strategy is that we are trying to determine which facility has been assigned the most clients when we have not finished assigning the clients yet. We believe that in most inputs, tie-breaking will be a infrequent (but important) edge case, and so we propose assigning clients to facilities over the course of two passes: In the first pass, we iterate over the clients and either assign them to their closest facility or add them to a tie-breaking list and continue with the next clients, and in the second pass, we iterate over the tie-breaking list and assign clients to their closest facility that currently has the highest inner value. Ties could occur again, and in this case we could either assign them to the lowest index or repeat the process, but we believe just trying to break ties in this manner a single time might improve the quality of the solutions.

Simulated Annealing

As mentioned above, since our algorithm performs single swap operations it can get stuck in local optimal solutions where swapping out only a single facility does not produce a better solution. Another approach to avoid getting stuck in these local optima, besides performing multi swaps, is to borrow a technique from Haralampiev [76] (and many others) and include a *temperature* parameter in our algorithm. This temperature parameter can be used to allow our algorithm to perform swap operations that do not improve the quality of the solution, and tuning the parameter can control the probability at which the algorithm might make these swaps. The benefit of this approach is that a sequence of single swaps could be performed that do not improve the solution quality but transition the state of the network closer to a global optimal solution.

Allowing our algorithm to perform swaps that reduce the quality of the solution will undoubtedly cause the running time to increase, as it will perform a larger number of iterations before getting stuck in a local optimal solution. However, as shown in Chapter 6, our current algorithm reaches local optima in a small number of iterations and hence runs very quickly. We can add a temperature parameter to our algorithm and still remain competitive with respect to the running time of the other algorithms that we compared it against.

Learning from Previous Solutions

We implemented several initialization strategies in our modified Hopfield network that attempt to identify the most valuable facilities from previous solutions and include them in future solutions. Our best performing initialization strategy, the best facility initialization, activates the facilities with the highest inner value from the best solution found so far and randomly selects the remaining facilities. This strategy essentially allows a multi-swap of the less valuable facilities to attempt to break free of a local optimal solution.

Given that we plan on implementing a multi-swap algorithm, we might want to re-visit the initialization strategies to design a better technique, as the usefulness of a single multi-swap between runs will decrease. We can borrow from the ideas of Rossiter [145]: Rossiter created a search tree where a parent vertex p contains a set of k facilities and the children of p contain solutions that are restricted to using less facilities or clients. In other words, the search tree allows the size of the problem to be reduced in order to further explore the value of selecting specific facilities. If the solutions contained along a specific path of the tree are not high-quality enough, then the next runs of the network can explore a different path of the search tree.

This idea combines the best parts of the initialization strategies that we tried: reducing the problem size and exploring multiple solutions that include a core set of valuable facilities. With the speed of our algorithm, we will be able to implement an extension such as the search tree and still produce solutions within a fast timeframe.

Analyzing the Approximation Ratio

Arya et al. [3] proved that any single swap local search algorithm for the k-median problem that produces solutions that cannot be further improved by a single-swap operation has an approximation ratio of 5. Even though our approach uses a modified Hopfield network, the core method involves transitioning from a feasible solution to another feasible solution by deactivating a single facility and activating another facility, which means that our approach is also a single swap local search algorithm. However, we currently cannot ensure that the solutions produced by our modified Hopfield network cannot be further improved by some additional single-swap operation; for example, see the discussion above on the improvements to the inner value metric and the neuron update function.

If we can prove that the suggested improvements to our algorithm guarantee that our local optimal solutions cannot be further improved with single-swap operations, then it follows that our algorithm has an approximation ratio of 5. Note that in Arya et al.'s analysis they consider a sequence of arbitrary swaps, whereas our algorithm uses the inner value metric to select the swap; therefore, in the future we can investigate whether there is a tighter bound on our algorithm's approximation ratio.

Additionally, once we implement p-swaps to our algorithm, which allow p facilities to be swapped with p clients, we will need to show that the solutions produced by our algorithm cannot be further improved by any p-swap. Such a proof would mean that our algorithm would have an approximation ratio of $(3 + \frac{2}{p})$.

Experimental Results using GPUs

The experimental results that we presented in Chapter 6 were produced using a computer that was not using a GPU. However, our modified Hopfield network is implemented using the pytorch library which enables efficient matrix calculations that take advantage of the GPU. The runtimes of our algorithm could be significantly sped up if we performed experimental results using a computer with a GPU.

Modified Integer Program for K-Median

Our modified Hopfield network uses $n+n^2$ neurons based on the integer program presented in Chapter 6. Domínguez and Muñoz [46] presented a different integer program formulation:

$$\text{Minimize} : \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{q=1}^{k} d_{i,j} x_{i,q} y_{j,q} \tag{7.1a}$$

$$\text{Subject to} : \sum_{q=1}^{k} x_{i,q} = 1 \, , i = 1, ...n \tag{7.1b}$$

$$\text{Subject to} : \sum_{j=1}^{n} y_{j,q} = 1 \, , i = 1, ...k \tag{7.1c}$$

where n is the number of locations in KMP, k is the number of facilities, $d_{i,j}$ is the distance between location i and facility j, and

$$x_{i,q} = \begin{cases} 1 & \text{if } i \text{ is assigned to the cluster } q, \\ 0 & \text{otherwise} \end{cases}$$

$$y_{j,q} = \begin{cases} 1 & \text{if } j \text{ is the center of the cluster } q, \\ 0 & \text{otherwise} \end{cases}$$

The $n \times k$ facility neurons $y_{j,q}$ represent whether a particular vertex j is a facility assigned to cluster q and the $n \times k$ client-facility neurons $x_{i,j}$ represent whether a particular vertex i is a client assigned to cluster q. Constraint (7.1b) requires that a client is only assigned to a single cluster and constraint (7.1c) requires that a cluster is only assigned a single facility. If $2 \times n \times k < n + n^2$ then this integer program formulation will result in our modified Hopfield network using fewer neurons and hence a smaller matrix representation, which could speed up our algorithm. Domínguez and Muñoz [46] show that integer program (7.1) is equivalent to the integer program described in Chapter 6.

Broader Experimental Evaluations

Our current experimental evaluation compares the performance of the algorithms of Arya et al. [3], Pan and Zhu [135], Cohen-Addad et al. [37], Haralampiev [76], and Rossiter [145]. However, it might be useful to also include in our evaluation the algorithms of Merino and Perez [47], Merino et al. [125], and Domínguez and Muñoz [46], as these networks are very similar to Hopfield networks.

Additionally, because we plan to improve our algorithm in a variety of ways, we believe the performance of our algorithm will outperform the competing neural network and local search algorithms. Therefore, we plan to also experimentally evaluate its performance against the state-of-the-art linear program rounding algorithms, which currently have the best theoretical bounds.

K-Median Problem Variants

The k-median problem is a classic member of the facility location family of problems. In the *uncapacitated facility location problem*, instead of requiring k facilities to be active, there is a cost associated with activating a facility and the goal is to minimize the total cost to serve all the clients. In the *capacitated facility location problem*, each facility has an upper bound on the number of clients it can serve. There is also a *capacitated k-median problem* where exactly *k* facilities should be selected but they each have an upper bound on the number of clients that they can serve.

Future research can investigate how to modify the Hopfield network to activate a flexible number of facilities and incorporate their activation costs into the energy function. For the capacitated problem, consideration is required to determine how best to assign clients to facilities in the network when an activate facility already is full; determining which of the clients that are currently being served should be swapped out could have large impacts on the solution quality.

Other Optimization Problems

Kamgar-Parsi and Kamgar-Parsi [97] have previously designed a Hopfield network for the k-means problem. Future research could apply our modified Hopfield network to not only the k-means problem, but to other popular clustering problems such as the k-medoids problem and variations of these problems.

In addition to investigating more clustering problems, in the future we can use our techniques to design a new modified Hopfield network for other optimization problems such as the traveling salesman problem and packing problems.

References

[1] Sreeram VB Aiyer, Mahesan Niranjan, and Frank Fallside. A theoretical investigation into the performance of the hopfield model. *IEEE transactions on neural networks*, 1(2):204–215, 1990.

[2] Esther M Arkin, Joseph SB Mitchell, and Giri Narasimhan. Resource-constrained geometric network optimization. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 307–316, 1998.

[3] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristic for k-median and facility location problems. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 21–29, 2001.

[4] B. Baker, D. Brown, and H. Katseff. A $\frac{5}{4}$ algorithm for two-dimensional packing. *Journal of Algorithms*, 2(4):348–368, 1981.

[5] B. Baker, R. Calderbank, E. Coffman, and J. Lagarias. Approximation algorithms for maximizing the number of squares packed into a rectangle. *SIAM Journal on Algebraic Discrete Methods*, 4(3):383–397, 1983.

[6] Brenda S Baker, Edward G Coffman, Jr, and Ronald L Rivest. Orthogonal packings in two dimensions. *SIAM Journal on computing*, 9(4):846–855, 1980.

[7] Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Adam Meyerson. Approximation algorithms for deadline-tsp and vehicle routing with time-windows. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 166–174, 2004.

[8] Amotz Bar-Noy, Randeep Bhatia, Joseph Naor, and Baruch Schieber. Minimizing service and operation costs of periodic scheduling. *Mathematics of Operations Research*, 27(3):518–544, 2002.

[9] John E Beasley. Or-library: distributing test problems by electronic mail. *Journal of the operational research society*, 41(11):1069–1072, 1990.

[10] Julian Blank, Kalyanmoy Deb, and Sanaz Mostaghim. Solving the bi-objective traveling thief problem with multi-objective evolutionary algorithms. In *Evolutionary Multi-Criterion Optimization: 9th International Conference, EMO 2017, Münster, Germany, March 19-22, 2017, Proceedings 9*, pages 46–60. Springer, 2017.

[11] Andrew Bloch-Hansen. High multiplicity strip packing. Master's thesis, The University of Western Ontario (Canada), 2019.

[12] Andrew Bloch-Hansen, Daniel R Page, and Roberto Solis-Oba. A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs. In *International Workshop on Combinatorial Algorithms*, pages 87–98. Springer, 2023.

[13] Andrew Bloch-Hansen and Roberto Solis-Oba. The thief orienteering problem on series-parallel graphs. In *International Symposium on Combinatorial Optimization*, pages 248–262. Springer, 2024.

[14] Andrew Bloch-Hansen, Roberto Solis-Oba, and Andy Yu. High multiplicity strip packing with three rectangle types. In *International Symposium on Combinatorial Optimization*, pages 215–227. Springer, 2022.

[15] Avrim Blum, Shuchi Chawla, David R Karger, Terran Lane, Adam Meyerson, and Maria Minkoff. Approximation algorithms for orienteering and discounted-reward tsp. In *44th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–55, 2003.

[16] Avrim Blum, Shuchi Chawla, David R Karger, Terran Lane, Adam Meyerson, and Maria Minkoff. Approximation algorithms for orienteering and discounted-reward tsp. *SIAM Journal on Computing*, 37(2):653–670, 2007.

[17] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *2013 IEEE Congress on Evolutionary Computation*, pages 1037–1044. IEEE, 2013.

[18] Mohammad Reza Bonyadi, Zbigniew Michalewicz, Michal Roman Przybylek, and Adam Wierzbicki. Socially inspired algorithms for the travelling thief problem. In *Proceedings of the 2014 annual conference on genetic and evolutionary computation*, pages 421–428, 2014.

[19] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems—an overview of recent advances. part i: Single knapsack problems. *Computers & Operations Research*, 143:105692, 2022.

[20] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems—an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems. *Computers & Operations Research*, 143:105693, 2022.

[21] Vicente Campos, Rafael Martí, Jesús Sánchez-Oro, and Abraham Duarte. Grasp with path relinking for the orienteering problem. *Journal of the Operational Research Society*, 65:1800–1813, 2014.

[22] A. Caprara and M. Monaci. On the two-dimensional knapsack problem. *Operations Research Letters*, 32(1):5–14, 2004.

[23] Jonatas BC Chagas and Markus Wagner. Ants can orienteer a thief in their robbery. *Operations Research Letters*, 48(6):708–714, 2020.

[24] Jonatas BC Chagas and Markus Wagner. Efficiently solving the thief orienteering problem with a max–min ant colony optimization approach. *Optimization Letters*, 16(8):2313–2331, 2022.

[25] Jonatas BC Chagas and Markus Wagner. A weighted-sum method for solving the bi-objective traveling thief problem. *Computers & Operations Research*, 138:105560, 2022.

[26] Timothy M Chan. Approximation schemes for 0-1 knapsack. In *1st Symposium on Simplicity in Algorithms (SOSA 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.

[27] I-Ming Chao, Bruce L Golden, and Edward A Wasil. A fast and effective heuristic for the orienteering problem. *European journal of operational research*, 88(3):475–489, 1996.

[28] Chandra Chekuri, Nitish Korula, and Martin Pál. Improved algorithms for orienteering and related problems. *ACM Transactions on Algorithms (TALG)*, 8(3):1–27, 2012.

[29] Ke Chen and Sariel Har-Peled. The orienteering problem in the plane revisited. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 247–254, 2006.

[30] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. A nearly quadratic-time fptas for knapsack. *arXiv preprint arXiv:2308.07821*, 2023.

[31] P. Chen, Y. Chen, M. Goel, and F. Mang. Approximation of two-dimensional rectangle packing. *CS270 Project Report*, 1999.

[32] Xiaoming Chen, Zheng Tang, Xinshun Xu, Songsong Li, Guangpu Xia, Ziliang Zong, and Jiahai Wang. An hopfield network learning for minimum vertex cover problem. In *SICE 2004 Annual Conference*, volume 2, pages 1150–1155. IEEE, 2004.

[33] John J Clifford and Marc E Posner. High multiplicity in earliness-tardiness scheduling. *Operations Research*, 48(5):788–800, 2000.

[34] Edgar Frank Codd. Multiprogram scheduling: Parts 1 and 2. introduction and theory. *Communications of the ACM*, 3(6):347–350, 1960.

[35] E. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin packing: a survey. *Approximation Algorithms for NP-hard Problems*, pages 46–93, 1996.

[36] Edward G Coffman, Jr, Michael R Garey, David S Johnson, and Robert Endre Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[37] Vincent Cohen-Addad, Anupam Gupta, Lunjia Hu, Hoon Oh, and David Saulpic. An improved local search algorithm for k-median. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1556–1612. SIAM, 2022.

[38] Vincent Cohen-Addad, Philip N Klein, and Claire Mathieu. Local search yields approximation schemes for k-means and k-median in euclidean and minor-free metrics. *SIAM Journal on Computing*, 48(2):644–667, 2019.

[39] Vincent Cohen-Addad Viallat, Fabrizio Grandoni, Euiwoong Lee, and Chris Schwiegelshohn. Breaching the 2 lmp approximation barrier for facility location with applications to k-median. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 940–986. SIAM, 2023.

[40] Stavros S Cosmadakis and Christos H Papadimitriou. The traveling salesman problem with many visits to few cities. *SIAM Journal on Computing*, 13(1):99–108, 1984.

[41] Hayssam Dahrouj, Rawan Alghamdi, Hibatallah Alwazani, Sarah Bahanshal, Alaa Alameer Ahmad, Alice Faisal, Rahaf Shalabi, Reem Alhadrami, Abdulhamit Subasi, Malak T Al-Nory, et al. An overview of machine learning-based techniques for solving optimization problems in communications and signal processing. *IEEE Access*, 9:74908–74938, 2021.

[42] Mark S Daskin and Kayse Lee Maass. The p-median problem. In *Location science*, pages 21–45. Springer, 2015.

[43] W Fernandez de La Vega and Vassilis Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, 82(1-3):93–101, 1998.

[44] Mingyang Deng, Ce Jin, and Xiao Mao. Approximating knapsack and partition via dense subset sums. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2961–2979. SIAM, 2023.

[45] Mohamed I Dessouky, Ben J Lageweg, Jan Karel Lenstra, and Steef L van de Velde. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44(3):115–123, 1990.

[46] Enrique Domínguez and José Muñoz. A neural model for the p-median problem. *Computers & Operations Research*, 35(2):404–416, 2008.

[47] Enrique Dominguez Merino and José Muñoz Perez. An efficient neural network algorithm for the p-median problem. In *Ibero-American Conference on Artificial Intelligence*, pages 460–469. Springer, 2002.

[48] K. Eisemann. The trim problem. *Management Science*, 3(3):279–284, 1957.

[49] Mohamed El Yafrani and Belaïd Ahiod. Efficiently solving the traveling thief problem using hill climbing and simulated annealing. *Information Sciences*, 432:231–244, 2018.

[50] Leonardo M Faêda and André G Santos. A genetic algorithm for the thief orienteering problem. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[51] Hayden Faulkner, Sergey Polyakovskiy, Tom Schultz, and Markus Wagner. Approximate approaches to the traveling thief problem. In *Proceedings of the 2015 annual conference on genetic and evolutionary computation*, pages 385–392, 2015.

[52] C. Filippi. On the bin packing problem with a fixed number of object weights. *European Journal of Operational Research*, 181(1):117–126, 2007.

[53] C. Filippi and A. Agnetis. An asymptotically exact algorithm for the high-multiplicity bin packing problem. *Mathematical Programming*, 104(1):21–37, 2005.

[54] Matteo Fischetti, Juan Jose Salazar Gonzalez, and Paolo Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2):133–148, 1998.

[55] A. Fishkin, O. Gerber, K. Jansen, and R. Solis-Oba. On packing squares with resource augmentation: Maximizing the profit. In *Proceedings of the 2005 Australasian Symposium on Theory of Computing-Volume 41*, pages 61–67. Australian Computer Society, Inc., 2005.

[56] A. Fishkin, O. Gerber, K. Jansen, and R. Solis-Oba. Packing weighted rectangles into a square. In *International Symposium on Mathematical Foundations of Computer Science*, pages 352–363. Springer, 2005.

[57] D. Friesen and M. Langston. Analysis of a compound bin packing algorithm. *SIAM Journal on Discrete Mathematics*, 4(1):61–79, 1991.

[58] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56(3):183–198, 1983.

[59] Waldo Gálvez, Fabrizio Grandoni, Salvatore Ingala, and Arindam Khan. Improved pseudo-polynomial-time approximation for strip packing. *arXiv preprint arXiv:1801.07541*, 2018.

[60] M. Garey and D. Johnson. “strong”np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978.

[61] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[62] MR Garey and RL Graham. Bounds on multiprocessing scheduling with resource constraints. *Siam, Jour. Comput*, 4(2):187, 1975.

[63] Michel Gendreau, Gilbert Laporte, and Frederic Semet. A branch-and-cut algorithm for the undirected selective traveling salesman problem. *Networks: An International Journal*, 32(4):263–273, 1998.

[64] Michel Gendreau, Gilbert Laporte, and Frédéric Semet. A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research*, 106(2-3):539–545, 1998.

[65] Michel Gendreau, Jean-Yves Potvin, et al. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[66] P. Gilmore and R. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[67] P. Gilmore and R. Gomory. A linear programming approach to the cutting stock problem—part ii. *Operations Research*, 11(6):863–888, 1963.

[68] P. Gilmore and R. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations Research*, 13(1):94–120, 1965.

[69] M. Goemans and T. Rothvoß. Polynomiality for bin packing with a constant number of item types. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 830–839. SIAM, 2014.

[70] Igal Golan. Performance bounds for orthogonal oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 10(3):571–582, 1981.

[71] Bruce L Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3):307–318, 1987.

[72] Lee-Ad Gottlieb, Robert Krauthgamer, and Havana Rika. Faster algorithms for orienteering and k-tsp. *Theoretical Computer Science*, 914:73–83, 2022.

[73] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332, 2016.

[74] Pierre Hansen and Nenad Mladenović. Variable neighborhood search for the p-median. *Location Science*, 5(4):207–226, 1997.

[75] Vladislav Haralampiev. Neural networks for facility location problems. *Annual of Sofia University St. Kliment Ohridski. Faculty of Mathematics and Informatics*, 106:3–10, 2019.

[76] Vladislav Haralampiev. Neural network approaches for a facility location problem. *Mathematical Modeling*, 4(1):3–6, 2020.

[77] Vladislav Haralampiev. Theoretical justification of a neural network approach to combinatorial optimization. In *Proceedings of the 21st International Conference on Computer Systems and Technologies*, pages 74–77, 2020.

[78] R. Harren. Approximation algorithms for orthogonal packing problems for hypercubes. *Theoretical Computer Science*, 410(44):4504–4532, 2009.

[79] Rolf Harren, Klaus Jansen, Lars Prädel, and Rob Van Stee. A (5/3+ ε)-approximation for strip packing. *Computational Geometry*, 47(2):248–267, 2014.

[80] Rolf Harren and Rob van Stee. Improved absolute approximation ratios for two-dimensional packing problems. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 177–189. Springer, 2009.

[81] Pengfei He, Jin-Kao Hao, and Qinghua Wu. Hybrid genetic algorithm for undirected traveling salesman problems with profits. *Networks*, 82(3):189–221, 2023.

[82] Daniel Herring, Michael Kirley, and Xin Yao. A comparative study of evolutionary approaches to the bi-objective dynamic travelling thief problem. *Swarm and Evolutionary Computation*, 84:101433, 2024.

[83] D. Hochbaum and R. Shamir. Strongly polynomial algorithms for the high multiplicity scheduling problem. *Operations Research*, 39(4):648–653, 1991.

[84] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[85] John J Hopfield and David W Tank. “neural” computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[86] Vu Huynh, The Viet Le, and Ngoc Hoang Luong. Self-adaptive ant system with hierarchical clustering for the thief orienteering problem. In *Proceedings of the 12th International Symposium on Information and Communication Technology*, pages 723–730, 2023.

[87] O. Ibarra and C. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[88] Maksud Ibrahimov, Arvind Mohais, Sven Schellenberg, and Zbigniew Michalewicz. Evolutionary approaches for supply chain optimisation: Part i: single and two-component supply chains. *International Journal of Intelligent Computing and Cybernetics*, 5(4):444–472, 2012.

[89] K. Jansen and R. Solis-Oba. A polynomial time approximation scheme for the square packing problem. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 184–198. Springer, 2008.

[90] K. Jansen and R. Solis-Oba. An opt + 1 algorithm for the cutting stock problem with constant number of object lengths. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 438–449. Springer, 2010.

[91] K. Jansen and G. Zhang. Maximizing the total profit of rectangles packed into a rectangle. *Algorithmica*, 47(3):323–342, 2007.

[92] Klaus Jansen and Malin Rau. Improved approximation for two dimensional strip packing with polynomial bounded width. *Theoretical Computer Science*, 789:34–49, 2019.

[93] Klaus Jansen and Roberto Solis-Oba. Rectangle packing with one-dimensional resource augmentation. *Discrete Optimization*, 6(3):310–323, 2009.

[94] Ce Jin. An improved fptas for 0-1 knapsack. *arXiv preprint arXiv:1904.09562*, 2019.

[95] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[96] D. Johnson and M. Garey. A $\frac{71}{60}$ theorem for bin packing. *Journal of Complexity*, 1(1):65–106, 1985.

[97] Behzad Kamgar-Parsi and Behrooz Kamgar-Parsi. On problem solving with hopfield neural networks. *Biological Cybernetics*, 62(5):415–423, 1990.

[98] Maryam Karimi-Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, 296(2):393–422, 2022.

[99] Narendra Karmarkar and Richard M Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 312–320. IEEE, 1982.

[100] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[101] Richard Karp. Reducibility among combinatorial problems. In *Symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[102] Hans Kellerer and Ulrich Pferschy. Improved dynamic programming in connection with an fptas for the knapsack problem. *Journal of Combinatorial Optimization*, 8:5–11, 2004.

[103] Frank P Kelly. Network routing. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 337(1647):343–367, 1991.

[104] Claire Kenyon and Eric Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, 25(4):645–656, 2000.

[105] L. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[106] Gorka Kobeaga, María Merino, and Jose A Lozano. An efficient evolutionary algorithm for the orienteering problem. *Computers & Operations Research*, 90:42–59, 2018.

[107] Madhukar R Korupolu, C Greg Plaxton, and Rajmohan Rajaraman. Analysis of a local search heuristic for facility location problems. *Journal of algorithms*, 37(1):146–188, 2000.

[108] Y. Lan, G. Dósa, X. Han, C. Zhou, and A. Benko. 2d knapsack: Packing squares. *Theoretical Computer Science*, 508:35–40, 2013.

[109] Gilbert Laporte and Silvano Martello. The selective travelling salesman problem. *Discrete applied mathematics*, 26(2-3):193–207, 1990.

[110] Eugene L Lawler. Fast approximation algorithms for knapsack problems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 206–213. IEEE, 1977.

[111] Adrienne C Leifer and Moshe B Rosenwein. Strong linear programming relaxations for the orienteering problem. *European Journal of Operational Research*, 73(3):517–523, 1994.

[112] Thomas Lengauer. The complexity of compacting hierarchically specified layouts of integrated circuits. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 358–368. IEEE, 1982.

[113] Thomas Lengauer and Egon Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM Journal on Computing*, 17(6):1063–1080, 1988.

[114] Yao Liang. Combinatorial optimization by hopfield networks using adjusting neurons. *Information Sciences*, 94(1-4):261–276, 1996.

[115] Yun-Chia Liang, Sadan Kulturel-Konak, and Min-Hua Lo. A multiple-level variable neighborhood search approach to the orienteering problem. *Journal of Industrial and Production Engineering*, 30(4):238–247, 2013.

[116] Yun-Chia Liang, Sadan Kulturel-Konak, and Alice E Smith. Meta heuristics for the orienteering problem. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 1, pages 384–389. IEEE, 2002.

[117] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS journal on computing*, 11(4):345–357, 1999.

[118] Antonio Lozano and José L Balcázar. The complexity of graph problems for succinctly represented graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 277–286. Springer, 1989.

[119] Alenrex Maity and Swagatam Das. Efficient hybrid local search heuristics for solving the travelling thief problem. *Applied Soft Computing*, 93:106284, 2020.

[120] TD Manjunath, S Samarth, Nesar Prafulla, and Jyothi S Nayak. Hopfield network based approximation engine for np complete problems. In *Innovative Data Communication Technologies and Application: ICIDCA 2019*, pages 319–331. Springer, 2020.

[121] Odile Marcotte. The cutting stock problem and integer rounding. *Mathematical Programming*, 33(1):82–92, 1985.

[122] Yannis Marinakis, Michael Politis, Magdalene Marinaki, and Nikolaos Matsatsinis. A memetic-grasp algorithm for the solution of the orienteering problem. In *Modelling, Computation and Optimization in Information Systems and Management Sciences: Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences-MCO 2015-Part II*, pages 105–116. Springer, 2015.

[123] S Thomas McCormick, Scott R Smallwood, and Frits CR Spieksma. A polynomial algorithm for multiprocessor scheduling with two job lengths. *Mathematics of Operations Research*, 26(1):31–49, 2001.

[124] Yi Mei, Xiaodong Li, and Xin Yao. On investigation of interdependence between sub-problems of the travelling thief problem. *Soft Computing*, 20:157–172, 2016.

[125] Enrique Domínguez Merino, José Muñoz-Pérez, and José M Jerez-Aragonés. Neural network algorithms for the p-median problem. In *ESANN*, pages 385–392, 2003.

[126] Shillpi Mishrra and Satabdi Barman. Cost reduction techniques in p-median method using artificial neural network. 2016.

[127] Paweł B Myszkowski and Maciej Laszczyk. Diversity based selection for many-objective evolutionary optimisation problems with constraints. *Information Sciences*, 546:665–700, 2021.

[128] Giorgi Nadiradze and Andreas Wiese. On approximating strip packing with a better ratio than 3/2. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1491–1510. SIAM, 2016.

[129] Balakrishnan Nagaraj, Rajendran Arunkumar, K Nisi, and Ponnusamy Vijayakumar. Enhancement of fraternal k-median algorithm with cnn for high dropout probabilities to evolve optimal time-complexity. *Cluster Computing*, 23(3):2001–2008, 2020.

[130] George L Nemhauser and Zev Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969.

[131] Frank Neumann, Sergey Polyakovskiy, Martin Skutella, Leen Stougie, and Junhua Wu. A fully polynomial time approximation scheme for packing while traveling.

In *Algorithmic Aspects of Cloud Computing: 4th International Symposium, ALGOCLOUD 2018, Helsinki, Finland, August 20–21, 2018, Revised Selected Papers 4*, pages 59–72. Springer, 2019.

[132] Adel Nikfarjam, Aneta Neumann, and Frank Neumann. On the use of quality diversity algorithms for the traveling thief problem. In *Proceedings of the genetic and evolutionary computation conference*, pages 260–268, 2022.

[133] James B Orlin. A polynomial algorithm for integer programming covering problems satisfying the integer round-up property. *Mathematical Programming*, 22(1):231–235, 1982.

[134] Susan Hesse Owen and Mark S Daskin. Strategic facility location: A review. *European journal of operational research*, 111(3):423–447, 1998.

[135] Rui Pan and Daming Zhu. An efficient local search algorithm for k-median problem. In *Proceedings of the 6th WSEAS international conference on Applied computer science*, pages 19–24. Citeseer, 2006.

[136] Christos H Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and control*, 71(3):181–185, 1986.

[137] Xue Peng, Xiaoyun Xia, Rong Zhu, Lei Lin, Huimin Gao, and Pei He. A comparative performance analysis of evolutionary algorithms on k-median and facility location problems. *Soft Computing*, 22(23):7787–7796, 2018.

[138] Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. A comprehensive benchmark set and heuristics for the traveling thief problem. In *Proceedings of the 2014 annual conference on genetic and evolutionary computation*, pages 477–484, 2014.

[139] Sergey Polyakovskiy and Frank Neumann. The packing while traveling problem. *European Journal of Operational Research*, 258(2):424–439, 2017.

[140] Harilaos N Psaraftis. A dynamic programming approach for sequencing groups of identical jobs. *Operations Research*, 28(6):1347–1359, 1980.

[141] R Ramesh, Yong-Seok Yoon, and Mark H Karwan. An optimal algorithm for the orienteering tour problem. *ORSA Journal on Computing*, 4(2):155–165, 1992.

[142] Ram Ramesh and Kathleen M Brown. An efficient four-phase heuristic for the generalized orienteering problem. *Computers & Operations Research*, 18(2):151–165, 1991.

[143] Donguk Rhee. *Faster fully polynomial approximation schemes for knapsack problems*. PhD thesis, Massachusetts Institute of Technology, 2015.

[144] Vahid Roostapour, Mojgan Pourhassan, and Frank Neumann. Analysis of baseline evolutionary algorithms for the packing while travelling problem. In *Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*, pages 124–132, 2019.

[145] Cody Rossiter. *A Modified Hopfield Network for the K-Median Problem*. PhD thesis, The University of Western Ontario (Canada), 2023.

[146] T. Rothvoß. Approximating bin packing within $o(logopt * loglogopt)$ bins. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 20–29. IEEE, 2013.

[147] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.

[148] Sancho Salcedo-Sanz and Xin Yao. A hybrid hopfield network-genetic algorithm approach for the terminal assignment problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(6):2343–2353, 2004.

[149] Alberto Santini. An adaptive large neighbourhood search algorithm for the orienteering problem. *Expert Systems with Applications*, 123:154–167, 2019.

[150] André G Santos and Jonatas BC Chagas. The thief orienteering problem: Formulation and heuristic approaches. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–9. IEEE, 2018.

[151] Ingo Schiermeyer. Reverse-fit: A 2-optimal algorithm for packing rectangles. In *Algorithms—ESA'94: Second Annual European Symposium Utrecht, The Netherlands, September 26–28, 1994 Proceedings 2*, pages 290–299. Springer, 1994.

[152] M. Schilde, K. Doerner, R. Hartl, and G. Kiechle. Metaheuristics for the bi-objective orienteering problem. *Swarm Intelligence*, 3(3):179–201, 2009.

[153] Zülal Sevkli and F Erdogan Sevilgen. Discrete particle swarm optimization for the orienteering problem. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[154] Zülal Sevkli and F Erdogan Sevilgen. Stpso: Strengthened particle swarm optimization. *Turkish Journal of Electrical Engineering and Computer Sciences*, 18(6):1095–1114, 2010.

[155] Hengameh Shamsipoor, Mohammad Ali Sandidzadeh, and Masoud Yaghini. Solving capacitated p-median problem by a new structure of neural network. *International Journal of Industrial Engineering*, 19(8), 2012.

[156] Daniel Dominic Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Inf. Process. Lett.*, 10(1):37–40, 1980.

[157] A Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[158] Barbaros C Tansel, Richard L Francis, and Timothy J Lowe. Location on networks: A survey. part i: The p-center and p-median problems. *Management Science*, pages 482–497, 1983.

[159] M Faith Tasgetiren. A genetic algorithm with an adaptive penalty function for the orienteering problem. *Journal of Economic & Social Research*, 4(2), 2002.

[160] Dena Tayebi, Saurabh Ray, and Deepak Ajwani. Learning to prune instances of k-median and related problems. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 184–194. SIAM, 2022.

[161] Michael B Teitz and Polly Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations research*, 16(5):955–961, 1968.

[162] Theodore Tsiligirides. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35:797–809, 1984.

[163] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[164] Pieter Vansteenwegen. Planning in tourism and public transportation: Attraction selection by means of a personalised electronic tourist guide and train transfer scheduling. *4or*, 7:293–296, 2009.

[165] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2011.

[166] F. De La Vega and G. Lueker. Bin packing can be solved within $1 + \varepsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[167] Markus Wagner. Stealing items more efficiently with ants: a swarm intelligence approach to the travelling thief problem. In *Swarm Intelligence: 10th International Conference, ANTS 2016, Brussels, Belgium, September 7-9, 2016, Proceedings 10*, pages 273–281. Springer, 2016.

[168] Markus Wagner, Marius Lindauer, Mustafa Mısır, Samadhi Nallaperuma, and Frank Hutter. A case study of algorithm selection for the traveling thief problem. *Journal of Heuristics*, 24:295–320, 2018.

[169] Jiahai Wang, Yalan Zhou, Jian Yin, and Yunong Zhang. Competitive hopfield network combined with estimation of distribution for maximum diversity problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(4):1048–1066, 2009.

[170] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[171] GV Wilson and GS Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biological Cybernetics*, 58(1):63–70, 1988.

[172] Junhua Wu, Markus Wagner, Sergey Polyakovskiy, and Frank Neumann. Exact approaches for the travelling thief problem. In *Simulated Evolution and Learning: 11th International Conference, SEAL 2017, Shenzhen, China, November 10–13, 2017, Proceedings 11*, pages 110–121. Springer, 2017.

[173] Mohamed El Yafrani, Marcella SR Martins, Mehdi El Krari, Markus Wagner, Myriam RBS Delgado, Belaïd Ahiod, and Ricardo Lüders. A fitness landscape analysis of the travelling thief problem. In *Proceedings of the genetic and evolutionary computation conference*, pages 277–284, 2018.

[174] A. Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

Curriculum Vitae

Name:	Andrew Bloch-Hansen
Post-Secondary Education and Degrees:	Western University London, ON 2019 - 2024 Ph.D.
	Western University London, ON 2017 - 2019 M.Sc.
	Western University London, ON 2013 - 2017 B.Sc.
Related Work Experience:	Teaching Assistant Western University 2017 - 2024
	Limited-Duties Instructor Western University 2019-2020
	Intern Autodata Solution 2016
Honours and Awards:	Alexander Graham Bell Canada Graduate Scholarship - Doctoral 2021-2024
	University of Western Ontario Research in Computer Science (UWORCS) 2024 Best Presentation
	University of Western Ontario Research in Computer Science (UWORCS) 2023 Best Presentation
	University of Western Ontario Research in Computer Science (UWORCS) 2022 Best Presentation
	University of Western Ontario Research in Computer Science (UWORCS) 2019 Best Presentation

Publications:

Theses

Bloch-Hansen, A. (2019). High multiplicity strip packing.

Bloch-Hansen, A. (2017). An experimental comparison of the isolation heuristic and the local search approximation algorithm for the multiway cut problem.

Journal Publications

Bloch-Hansen, A., Samei, N, Solis-Oba, R. (2023). A local search approximation algorithm for the multiway cut problem. *Journal of Discrete Applied Mathematics*, 338 (pp. 8-21), https://doi.org/10.1016/j.dam.2023.05.022.

Refereed Conference Publications

Bloch-Hansen, A., Solis-Oba, R. (2024). The thief orienteering problem on series-parallel graphs. *Proceedings of the 8th International Symposium on Combinatorial Optimization (ISCO)* (pp. 248-262), https://doi.org/10.1007/978-3-031-60924-4_19.

Bloch-Hansen, A., Page, D. R., Solis-Oba, R. (2023). A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs. *Proceedings of the 34th International Workshop on Combinatorial Algorithms (IWOCA)* (pp. 87-98), https://doi.org/10.1007/978-3-031-34347-6_8.

Bloch-Hansen, A., Solis-Oba, R., Yu, A. (2022). High multiplicity strip packing with three rectangle types. *Proceedings of the 7th International Symposium on Combinatorial Optimization (ISCO)* (pp. 215-227), https://doi.org/10.1007/978-3-031-18530-4_16.

Bloch-Hansen, A., Samei, N., Solis-Oba, R. (2021). Experimental evaluation of a local search approximation algorithm for the multiway cut problem. *Proceedings of the Conference on Algorithms and Discrete Applied Mathematics (CALDAM)* (pp. 346-258), https://doi.org/10.1007/978-3-030-67899-9_28.