Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

8-20-2015 12:00 AM

Modern Optimization Algorithms and Applications: Architectural Layout Generation and Parallel Linear Programming

Bradley J. de Vlugt, *The University of Western Ontario*

Supervisor: Dr. Abdallah Shami, *The University of Western Ontario*
Joint Supervisor: Dr. Serguei Primak, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

© Bradley J. de Vlugt 2015

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

Part of the Other Electrical and Computer Engineering Commons

Recommended Citation

de Vlugt, Bradley J., "Modern Optimization Algorithms and Applications: Architectural Layout Generation and Parallel Linear Programming" (2015). *Electronic Thesis and Dissertation Repository*. 3208.
https://ir.lib.uwo.ca/etd/3208

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.

MODERN OPTIMIZATION ALGORITHMS AND APPLICATIONS: ARCHITECTURAL LAYOUT DESIGN AND PARALLEL LINEAR PROGRAMMING

(Thesis format: Integrated Article)

by

Bradley de Vlugt

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Bradley James de Vlugt 2015

Acknowlegements

I would like to acknowledge Dr. Maysam Mirahmadi and my advisors, Dr. Abdallah Shami and Dr. Serguei Primak, for their mentorship throughout this thesis. I am grateful for the creativity and wisdom they shared to help me grow as a researcher. I would like to thank my fellow lab members for their friendship during this rewarding and challenging endeavor. I would also like to thank my thesis defence committee for their insightful comments and suggestions.

To my parents, Lori and Paul de Vlugt, thank you for your guidance and encouragement. This thesis would not have been possible without you both. To my brother, Jeff de Vlugt, thank you for the jam sessions and helping me think outside the box. To my girlfriend, Carolyn Bolger, thank you for all of your love and amazing carrot cake cupcakes. To Kevin Brightwell, thank you for the interesting software design discussions and best wishes with your academic pursuits. And to all my colleagues, friends and family, thank you for your support throughout this work.

Abstract

This thesis examines two topics from the field of computational optimization; architectural layout generation and parallel linear programming. The first topic, a modern problem in heuristic optimization, focuses on deriving a general form of the optimization problem and solving it with the proposed *Evolutionary Treemap* algorithm. Tests of the algorithm's implementation within a highly scalable web application developed with Scala and the web service framework Play reveal the algorithm is effective at generated layouts in multiple styles. The second topic, a classical problem in operations research, focuses on methodologies for implementing the *Simplex Algorithm* on a parallel computer for solving large-scale linear programming problems. Implementations of the algorithm's data-parallel and task parallel forms illuminate the ideal method for accelerating a solver. The proposed *Multi-Path Simplex Algorithm* shows an average speed up of over two times that of a popular open-source solver, showing it is an effective methodology for solving linear programming problems.

Keywords: Genetic Algorithms, Procedural Generation, Architecture, Treemap, Linear Programming, OpenCL, Simplex Algorithm, Parallel Computing

Contents

Acknowlegements ii

Abstract iii

List of Figures viii

List of Tables x

List of Appendices xi

1 Introduction 1

1.1 Background 1

1.1.1 Architectural Layout Generation 2

1.1.2 The Simplex Algorithm 3

1.2 Contributions 4

1.2.1 Architectural Layout Generation 4

1.2.2 Parallel Linear Programming 4

2 Architectural Layout Generation With Evolutionary Treemap 8

2.1 Introduction 8

2.2 Literature Review 9

2.3 Formulating the Layout Generation Problem 12

2.3.1 The Standard Form 12

2.3.2 The Solution Space 14

2.3.3 The Generator Function 16

2.3.4 Optimizer Functions 17

2.3.5 The Cost Function . . . 17
2.4 Evolutionary Treemap . . . 18
2.4.1 Rectified Treemap . . . 19
2.4.2 Corridor Placement . . . 21
2.4.3 Evolutionary Optimization . . . 23
2.4.4 Measuring the Cost Value . . . 24
2.5 System Design and Implementation . . . 25
2.5.1 Representation of the User Specification . . . 27
2.5.2 Software Library Architecture . . . 28
2.5.3 API Design . . . 29
2.6 Discussion . . . 31
2.6.1 Office Buildings . . . 31
2.6.2 Open-Concept Residential Buildings . . . 33
2.6.3 Convergence Behavior . . . 35
2.6.4 Future Work . . . 35
Fixed Structures and Non-Rectangular Boundaries . . . 35
Corridor Optimizer Function . . . 39
Multi-Story Buildings . . . 40
2.7 Conclusion . . . 41
3 Parallel Linear Programming with Dense Data Structures 44
3.1 Introduction . . . 44
3.2 Literature Review . . . 45
3.3 Background . . . 47
3.3.1 The Dictionary Simplex Algorithm . . . 49
The Augmented Linear Programming Problem . . . 49
The Optimal BFS . . . 50
The Dictionary Data Structure . . . 51
The Initial Partition . . . 51
The Pricing Algorithm . . . 52

The Ratio Test 52
The Pivot Algorithm 53
The Size of the Dictionary 53
3.4 Parallel Implementation of the Dictionary Algorithm 55
3.4.1 OpenCL Software Design Architecture 56
3.5 Benchmarking Results 57
3.5.1 Speed Up 57
3.5.2 Performance versus Power 59
3.5.3 Memory Bandwidth 60
3.5.4 Future Directions 60
3.6 Conclusions 62

4 Multi-Path Parallelism in the Simplex Algorithm 65
4.1 Introduction 65
4.2 Literature Review 66
4.3 Internal Parallelism Performance Bounds 67
4.3.1 Profiling Methodology 68
4.3.2 Computing Performance Limits 68
4.4 The Multi-Path Simplex Algorithm 71
4.4.1 Algorithm Configurations 73
4.5 Profiling The Multi-Path Algorithm 74
4.6 Discussion 75
4.6.1 Future Directions 78
4.7 Conclusion 78

5 Conclusion 80
5.1 Intelligent Architectural Design 80
5.2 Parallel Linear Programming 81
5.3 Concluding Remarks 82

A SoPlex Performance Data 84

B Multi-Path Simplex Algorithm Performance Data **108**

Curriculum Vitae **120**

List of Figures

2.1 A Functional Layout for an Office Building and the Architectural Design 8

2.2 Example Treemap Representation 11

2.3 Different Mathematical Representations for a Single Room 15

2.4 Visual example of the result in Lemma 2.3.4. Partitioning the final section into the entrance and kitchen presents uncountably infinite possibilities between $y = P$ and $y = Q$ if $y \in \mathcal{R}$. 16

2.5 In larger layouts, applying the spanning tree algorithm eliminates redundant edges in the corridor to minimize wasted space 22

2.6 Effect of cost function parameters on the generated layout 26

2.7 UML Diagram of the Layout Generation Software 29

2.8 Two example office layouts generated by the *Evolutionary Treemap* algorithm based on the specification in Table 2.3 32

2.9 Residential layouts generated by the *Evolutionary Treemap* algorithm based on the specification in Table 2.4 34

2.10 Convergence of *Evolutionary Treemap* over twenty generations for an office layout with a mutation rate of 0.08 and an elite rate of 0.01 36

2.11 An example layout with a non-rectangular boundary generated with the augmented *Evolutionary Treemap* algorithm 38

2.12 An example of a corridor that contains loops which may or may not be necessary depending on the designer preferences 39

3.1 Visualization of the feasible region in a simple linear programming problem with two decision variables and four constraints. Arrows indicate a possible Simplex path. 48

3.2 The Stages of the *Simplex Algorithm* 51
3.3 Dictionary Algorithm Design Architecture 56
3.4 Speed up of the OpenCL dictionary algorithm over the sequential C++ algorithm 58
3.5 Device performance versus power analysis 59
3.6 The estimated percentage of available memory bandwidth used by each device . 61

4.1 Distribution of algorithm runtime in SoPlex for the selected test problems . . . 69
4.2 Maximum possible speedup predicted by Amdahl's Law 70
4.3 Ratio of the time taken by SoPlex to the time taken by the *Multi-Path Simplex Algorithm* for the full data set 76
4.4 Ratio of the time taken by SoPlex to the time taken by the *Multi-Path Simplex Algorithm* for problems with over 10^6 elements 77

List of Tables

2.1 Functional Layout Generation: Nomenclature for the General Form 13

2.2 Web Service API 30

2.3 Shape Specifications for the Rooms in the Medium Sized Office Layout 31

2.4 Shape Specifications for the Rooms in the Residential Home Layout 33

3.1 Hardware Device Specifications 57

A.1 SoPlex Runtime for Selected Linear Programming Test Cases 85

A.2 Average Runtime of SoPlex Functions over Selected Linear Programming Test Cases from Perf Profiling Reports 96

B.1 Averaged Performance Results for Multi-Path Simplex versus SoPlex with Randomly Selected Pricing Algorithms for 100 Trials 109

List of Appendices

SoPlex Performance Data 84

Multi-Path Simplex Algorithm Performance Data 108

Chapter 1

Introduction

Parallel and distributed computing systems present opportunities in the field of optimization from the perspective of scale and speed in both modern and classical applications. This thesis presents three articles from two optimization applications that introduce new algorithms, demonstrate scalable software design and enhance algorithms with parallelism. The first application is architectural layout generation, a modern application of optimization that utilizes stochastic algorithms to calculate the location and shape of rooms for a building floor plan. The second is linear programming with the *Simplex Algorithm*, a classical optimization algorithm that solves models with linear cost function, linear constraints and no integer variables. Both of these applications are examined herein, culminating in a set of novel contributions for each field that improve upon software performance and design. The topics are disjoint in nature but share the common theme of optimization and algorithm design.

This thesis begins with a summary on the prerequisite information from linear programming and architectural layout generation and refers interested readers to detailed texts. Summaries of the main objectives and contributions of the document follow.

1.1 Background

Parallel linear programming and architectural layout generation have seen numerous notable advancements since their conceptions and are both supported with large bodies of literature. This section provides a brief review of the pertinent literature from both applications, leaving

detailed reviews to the respective chapters. The review begins with architecture generation and concludes with the *Simplex Algorithm*.

1.1.1 Architectural Layout Generation

Industrial building renovation projects are performed by teams that consist of architects, designers, engineers and clients. The projects require allocation of spaces within an existing building by the architects and designers to meet building codes that are checked by the engineers and design specifications that are checked by the client. The full design is specified with a computer aided design (CAD) drawing that undergoes iterative refinement to ensure that all specifications, both technical and artistic, are met before construction can occur.

Many tools are available to this industry for increasing the speed at which CAD drawings can be produced. The first building design drawings were conducted by hand on drafting boards. This practice was soon replaced after the 1982 invention of AutoCAD [1] which allowed formation of the layout drawings on a computer that could more quickly be used to refine the drawings as specifications were revealed. Multiple such automated CAD tools such as SolidWorks [2], Revit [3] and MicroStation [4] are now available for use in this industry to accelerate projects. These software allow three dimensional modelling of entire construction sites. AutoDesk now provides a cloud version of AutoCAD which allows users to edit drawings in a web browser. The advancements in CAD software have greatly assisted the ability of stakeholders in construction and renovation projects to react to changing requirements in a design.

Functional layout generation is a modern topic in stochastic optimization, seeing its first literature in the 1990's [5] following advancements in computer graphics and data visualization methodologies. The majority of approaches in literature are based on stochastic optimization techniques, which optimize a data structure through mutations and measure changes with a heuristic cost function. For example, a recent technique proposed in literature generates layouts with genetic optimization [6] on a tree data structure. Other techniques are based on deterministic algorithms, such as *Squarified Treemap* [7]. This field is in infancy relative to linear programming and at present there are no commercial or open-source software that provide a

full solution for generating architectural layouts. The majority of these algorithms are tailored towards generation of residential homes, leaving additional topologies that can be explored through different formations of the problem with other optimization techniques.

1.1.2 The Simplex Algorithm

Linear Programming has been treated by numerous authors dating back to the discovery of the *Simplex Algorithm* in the 1940's by George Dantzig [8]. The algorithm transformed from the initial form proposed by Dantzig to a highly efficient, optimized form through discoveries by authors such as Forrest [9], Tomlin [10], Maros [11], and many others, resulting in enhanced efficiency and stability. The optimization can be used to solve models with linear objectives and constraints that appear in fields such as task scheduling, computer network design and radiation therapy.

High performance forms of the algorithm are available for engineers as both commercial packages, with IBM CPLEX [12], Gurobi [13], XPress [14], and open-source software, with SoPlex [15], GLPK [16], Coin-OR [17], LpSolve [18]. These commercial and open-source codes provide a wealth of opportunity for engineers to optimize linear programming models in different applications. The support for parallel processor cores available in commercial codes, as seen by the parallel switch available in CPLEX [19], has yet to be realized in the open-source code and is a resource available that could improve performance.

Recent efforts to parallelize the *Simplex Algorithm* have investigated performance of the algorithm on highly parallel processors such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). A study of the algorithm, operating with dense data structures on randomly generated problems revealed that speed ups in the order of twenty times are possible [20]. Another study that focused on implementation with FPGAs also showed potential for performance improvements over traditional sparse codes [21]. Though these results are promising, no author has implemented a parallel algorithm capable of solving the numerically challenging sparse problems found in practical modeling scenarios that shows good speed up. This is a challenge due to the inherently sequential nature of the algorithm. It consists of many sparse matrix kernels that are applied sequentially to optimize the model.

1.2 Contributions

This thesis approaches numerous topics in modern intelligent architectural layout and classical linear programming optimization. The work delves into the following open problems within the two fields: finding new layout generation algorithms, developing a software infrastructure for generating layouts, investigating linear programming on heterogeneous computing systems, and testing parallel forms of the sparse *Simplex Algorithm*. This section summarizes the main contributions in each field.

1.2.1 Architectural Layout Generation

The following contributions are presented in this thesis for the application of generating functional layouts for modern structures in the Chapter 2:

- A general formulation of layout generation as an optimization problem
- Commentary on the size and shape of the optimization problem's solution space
- The *Rectified Treemap* and *Evolutionary Treemap* algorithms
- Various heuristic methods for fine tuning layouts generated with *Evolutionary Treemap*
- A cloud based software platform for generating functional layouts
- A scalable object oriented design for layout generation software
- A web service and Application Programming Interface for the layout software

1.2.2 Parallel Linear Programming

The following contributions are presented in this thesis with regards to implementing a parallel linear programming solver based on the *Simplex Algorithm* for solving dense problems with increased speed in Chapter 3:

- An implementation of a dense linear programming solver for heterogeneous computing systems

- An OpenCL design for dense linear programming with the *Simplex Algorithm* on GPUs and FPGAs
- Performance testing for the *Simplex Algorithm* on multiple OpenCL devices

The contributions described in Chapter 4 of the thesis solve sparse linear programming problems with a task-parallel *Simplex Algorithm* and are the following:

- A novel profiling tool and methodology for analyzing linear programming software
- An upper bound for speedups from data-level parallelism in the implementation of the *Simplex Algorithm* found in the open-source code SoPlex
- A task-level parallel augmentation to the *Simplex Algorithm* found in SoPlex, the *Multi-Path Simplex Algorithm*, that surpasses the upper bound for data-level parallelism

Bibliography

[1] Autocad: Design every detail, 2015. http://www.autodesk.com/products/autocad/overview.

[2] Solidworks, 2015. http://www.solidworks.com/.

[3] Revit, 2015. http://www.autodesk.com/products/revit-family/overview.

[4] Bentley microstation, 2015. http://www.bentley.com/en-US/Products/MicroStation/.

[5] R.L. Grimsdale and C.W. Chang. Layout design language: a technique for generating layout plans. 15(2):97 – 106, 1996.

[6] Darcy Chia and Lyndon While. Automated design of architectural layouts using a multi-objective evolutionary algorithm. In Grant Dick, WillN. Browne, Peter Whigham, Mengjie Zhang, LamThu Bui, Hisao Ishibuchi, Yaochu Jin, Xiaodong Li, Yuhui Shi, Pramod Singh, KayChen Tan, and Ke Tang, editors, *Simulated Evolution and Learning*, volume 8886 of *Lecture Notes in Computer Science*, pages 760–772. Springer International Publishing, 2014.

[7] Maysam Mirahmadi and Abdallah Shami. A novel algorithm for real-time procedural generation of building floor plans. abs/1211.5842, 2012.

[8] G. B. Dantzig. Maximization of a linear function subject to linear inequalities,. In T. C. Koopmans, editor, *Activity Analysts of Production and Allocation*, pages 317–329. John Wiley and Sons, New York, 1951.

[9] JohnJ. Forrest and Donald Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57(1-3):341–374, 1992.

[10] J.A. Tomlin. On pricing and backward transformation in linear programming. *Mathematical Programming*, 6(1):42–47, 1974.

[11] Istvan Maros. *Computational Techniques of the Simplex Method*. Kluwer Academic Publishers, Norwell, MA, 2003.

[12] IBM CPLEX optimizer, 2015. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[13] Gurobi optimization, 2015. http://www.gurobi.com/.

[14] FICO xpress optimization suite, 2015.
http://www.fico.com/en/products/fico-xpress-optimization-suite.

[15] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.
http://www.zib.de/Publications/abstracts/TR-96-09/.

[16] GLPK - GNU linear programming kit, 2012. http://www.gnu.org/software/glpk/.

[17] Computational infrastructure for operations research, 2012. http://www.coin-or.org/.

[18] Lpsolve linear programming solver, 2015. http://lpsolve.sourceforge.net/.

[19] ILOG CPLEX 11.0 parameters reference manual: Parallel mode switch, 2007.
http://www-eio.upc.edu/lceio/manuals/cplex-11/html/refparameterscplex/refparameterscplex87.html.

[20] A. Hamzic, A. Huseinovic, and N. Nosovic. Implementation and performance analysis of the simplex algorithm adapted to run on commodity OpenCL enabled graphics processors. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies (ICAT)*, pages 1–7, 2011.

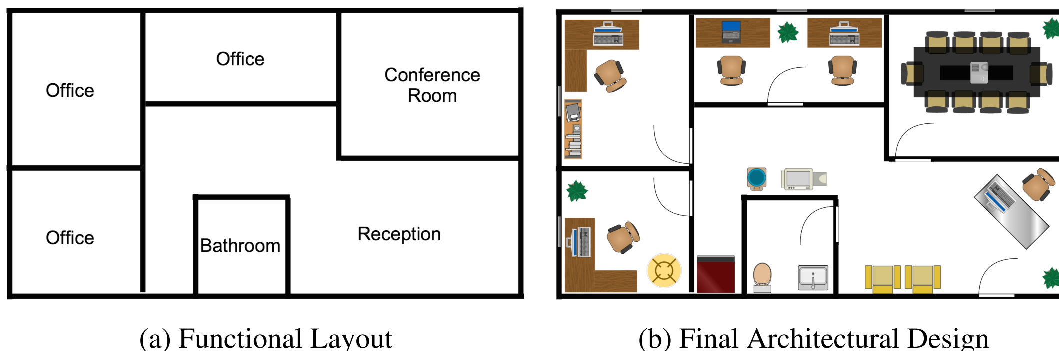
[21] S. Bayliss, C.-S. Bouganis, G.A. Constantinides, and W. Luk. An FPGA implementation of the simplex algorithm. In *IEEE International Conference on Field Programmable Technology*, pages 49–56, 2006.

Chapter 2

Architectural Layout Generation With Evolutionary Treemap

2.1 Introduction

The iterative process of building design and modeling conducted by architects transforms technical specifications into a spatial arrangement of rooms to meet building codes and aesthetic constraints. The process begins with an initial aesthetic conception that is refined by multiple stakeholders and cross-discipline engineering teams into detailed requirements for the building. The process results in complete specifications of the building for construction. These Computer Aided Design (CAD) drawings require input from multiple designers and consume a large portion of project time.



(a) Functional Layout (b) Final Architectural Design

Figure 2.1: A Functional Layout for an Office Building and the Architectural Design

This chapter presents a software that reduces the amount of time spent in the CAD design stage of a construction project. Figure 2.1a shows an example of a functional layout and Figure 2.1b presents a final architectural design based on this initial concept. The general form of the optimization problem is used to derive a new algorithm, *Evolutionary Treemap*, that can procedurally generate single floor building layouts based on a client's technical specifications and usage constraints. This algorithm is an extension to the deterministic algorithm based on *Squarified Treemap* proposed in [1]. *Evolutionary Treemap* improves the aspect ratio and spatial location of the rooms present in the generated layouts with a genetic optimization algorithm. The contributions of this chapter are a general framework from which layout generation algorithms can be constructed, the presentation of *Evolutionary Treemap* and a new visualization algorithm called *Rectified Treemap*.

This chapter begins with an overview of the literature and state of the art in layout generation. The general form of the layout generation problem is then presented and the components of a layout generation algorithm are derived. The third section presents the *Evolutionary Treemap* algorithm in general form. The fourth presents a scalable software system implemented to test the algorithm. The final section presents examples of layouts generated with the system and provides concluding remarks on the future directions of procedural generation research.

2.2 Literature Review

Computer aided design projects undertaken by engineers in industry require strict budget and schedule constraints. The estimated costs for procuring an architect for the purposes of layout design for a building is estimated to be in the order of 10% of the total design and construction costs [2]. Computer aided construction design is not the only field in which computer generated architectural building models are utilized; applications in fields such as building construction, cost estimation, and environmental simulation rely on these models. In construction, civil engineers and architects collaborate to design sets of models that facilitate cost estimation, material procurement, and construction. Professional software such as AutoCAD [3], Revit [4] and SolidWorks [5] are some examples of the many tools used by engineers to design models.

Architectural drawings take a varied amount of time based on the building complexity and size. For example, a general guideline estimated by the University of Boulder Colorado is an approximate schedule length of four to eight months based on the building's specification for a capital infrastructure project [6]. This is a large time line that requires fast, accurate designs from engineers.

In video game design, software engineers and 3D model designers create sets of virtual environments that contain detailed building layouts to immerse users in an immersive environment. For example, in the popular video game series *The Witcher* developed by CD Projekt Red [7], *Left 4 Dead* by Valve Software [8] and *Assassins Creed* by Ubisoft [9] there are vast open worlds that contain countless buildings for a user to explore. Reducing the costs of creating these CAD models could shorten the schedules for projects and thus the cost for projects in this industry. Algorithms that generate functional layouts are desirable due to the direct impact they can have on complex engineering and video game development projects.

Several approaches are proposed in literature for automated generation of functional layouts based on graphical visualization techniques such as treemap algorithms. Treemap algorithms are hierarchal shape partitioning algorithms that places a number of child rectangles with known areas into a boundary rectangle [10]. Figure 2.2 shows an example of the result of the treemap algorithm on a rectangle with four interior rectangles with areas of twenty-five, sixteen, nine, and four square feet. Treemap algorithms are originally from the field of computer visualization and partition rectangles into a number of sub-rectangles that are sized based on proportions within some data set [10].

Squarified Treemap was first proposed as an algorithm for functional layout generation in [11]. Given a target area per room, the rooms are split into functional categories and place in a *Squarified Treemap*. Next, a corridor is dynamically located via the interior points and subtracted from the adjacent rooms with polygon operators. The dynamic corridor placement algorithm was extended in [1] with an optimization technique based on a shortest path search that reduced the functional area removed from the layout's rooms by the corridor. The benefit of these treemap algorithms is that they do not require a search through a combinatorial space giving them excellent time to completion. However, due to the dynamic corridor algorithms, they can generate layouts in which rooms have undesirable aspect ratios. For example, a living

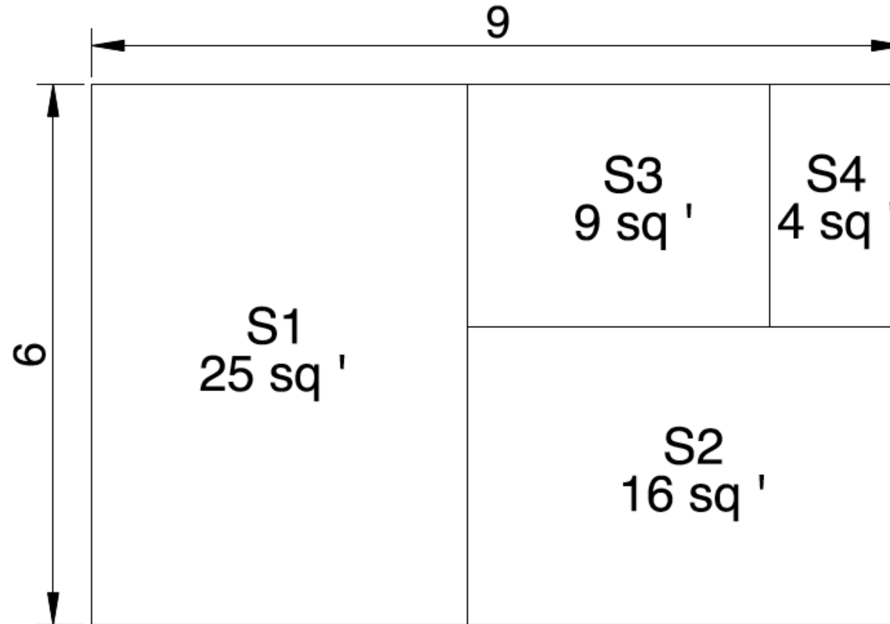


Figure 2.2: Example Treemap Representation

room in a residential home must house large furniture that could not fit in a room with a very narrow dimension even though the desired area for the room may be met by the algorithm with a room that is long and narrow.

Other approaches have been proposed based on optimization strategies such as simulated annealing [12], constraint-satisfaction [13], and genetic algorithms [14]. These strategies propose mutable data structures to represent layouts, iterate through candidate solutions and measure their quality based on a heuristic cost function. The representations vary from lists of movable edges [12], enumerated grid spaces [13], and shape grammar expressions [14]. The fitness functions are based on parameters such as room connectivity, area and aspect ratio. These algorithms have been shown to generate layouts that resemble those designed by an architect from a similar specification with a primary focus on residential styles. The performance of each algorithm depends on the solution space of the model chosen to represent a layout. For example, in [12], the solution space of the algorithm is large due to the fact that it includes any layout that can be formed from shifting walls to new locations but only a small percentage of this space is explored by the simulated annealing heuristic. This is smaller than the total number of solutions examined by the algorithm proposed in [13] due to the fact that this algo-

rithm requires examination of the entire solution space with backtracking. Algorithms that can find desirable layouts without resorting to a full search of a combinatorial space are desired to reduce the run-time of the solution and enable generation of multiple candidate solutions.

2.3 Formulating the Layout Generation Problem

As of yet, no author has presented a general language for comparing functional layout generation algorithms and the sub spaces of solutions they explore. A general representation of the algorithm allows implementation in a software system with a good object-oriented design to provide a framework for developing and testing new algorithms. This section will formulate the layout generation problem with a general language that is applicable to all of the efforts in literature from the prior section. The terminology established in this section will be used throughout the chapter to refer to the components of a layout generation algorithm. The problem is formulated in a general manner, and in later sections, the detailed components of a concrete algorithm are presented. The general form of this problem can be extended to buildings of any size and style such as multistory apartment complexes, hospitals, hotels, and residential homes. It can also be used to produce new layout generation algorithms. The nomenclature used to derive the standard form of the layout generation problem is presented in Table 2.1.

2.3.1 The Standard Form

Designing a functional layout requires spatial allocation of multiple rooms within some predefined boundary B. This boundary may be a representation of the exterior walls of the building or the plot of land on which it is to be constructed. The space of possible layouts $\mathcal{F}$ that can be formed through some spatial partition of B contains many solutions to the functional layout design problem. A candidate solution to the functional layout generation problem is defined in 2.3.1.

Definition 2.3.1. *Any layout $L \in \mathcal{F}$ that is allocated within a building based on a set of user specifications U constitutes a candidate solution. A layout L is composed of a set of rooms R. The input U contains some form of constraints, which may be fuzzy, on the properties of some*

Table 2.1: Functional Layout Generation: Nomenclature for the General Form

Symbol	Definition
B	The boundary of the layout, whether a set of walls or a plot of land
$\mathcal{F}$	The space of all possible vector layouts contained within B
L	An instance of a layout from $\mathcal{F}$
R	The set of rooms from which a layout is composed
U	The user specification that defines the set of rooms
P	The space of possible two-dimensional polygons
C	The space of possible classifiers for a room such as *Living Room* or *Office*
$\mathbf{x}$	A list of real coordinates in two-dimensions
$\mathcal{F}_z$	The space of all possible raster layouts contained within B
$\mathbf{x_z}$	A list of integer coordinates in two-dimensions
G	A generator function that reduces the space $\mathcal{F}$ to a size that can be searched
$\mathcal{F}_G$	The subset of $\mathcal{F}$ formed by the generator function
A	The cost function that describes the aesthetics of a layout
h	The value of the cost function for a particular layout
$\mathcal{F}_O$	The subset of $\mathcal{F}$ formed by an optimizer function
$\mathcal{F}_S$	The subset of $\mathcal{F}$ in the final restricted space (from optimizer or generator)

or all of each $r \in R$.

The layout generation problem can thus be phrased as an optimization that seeks the solution L that best matches U from $\mathcal{F}$ and requires the placement of R. The best layout is determined by a heuristic cost function that will be developed in later sections. To place each of the rooms, a functional classifier and location must be chosen. Creating a candidate solution to the problem thus requires forming a polygon and deciding upon a classifier for each of the rooms specified by the user in a manner that satisfies any constraints in the input. This representation will be defined as the *vector form* of a room and is formally presented in 2.3.2. Figure 2.3a presents an example of a room that is in vector form.

Definition 2.3.2. *The vector representation of a layout is a set of rooms R, where* $\forall r \in R$ $(r = \{p, c\} \mid p \in P \wedge c \in C)$ *and p is a two dimensional polygon with a list of coordinates* $x \in \mathbb{R}^2$*, P is the set of possible two dimensional polygons, c is a classifier such as "Living Room" or "Office", and C is the set of possible classifiers.*

A layout in which all rooms are in vector form is defined to be the *standard form* of a solution. This is because any other representation can be computed from this structure as will

be shown through examples in Section 2.3.2. As an example, consider a representation that relies on the assumption that the coordinates that form the polygons of the rooms are integer multiples of some unit rather than real numbers. This means that the coordinates $x = ux_z$ where $x_z \in \mathbb{Z}^2$ and the unit u is some parameter chosen by a user. The u value could for example be taken to be one meter as in [1] as this is the proposed width of the narrowest space in a layout, the corridor, or it could be taken as some fraction of an inch for the largest unit that can be practically measured when constructing the layout.

This form will be defined as the *raster form* of a layout. It can be visualized as a grid based partition of the boundary in which each block is approximately $u \times u$ as shown in Figure 2.3b and defined in 2.3.3

Definition 2.3.3. *The raster representation of a layout is a set of rooms R, where* $\forall r \in R(r = \{B_r, c\} \mid B_r \subset B_z \bigwedge c \in C)$ *and* B_r *is a set of grid blocks,* B_z *is the total set of grid blocks from partitioning the boundary into a grid with unit u, c is a classifier from the set C of possible classifiers,* $B_{r1} \cup B_{r2} \cup ... \cup B_{rn} = B_z$*, and* $B_{ri} \cap B_{rj} = \{0\}$*, .*

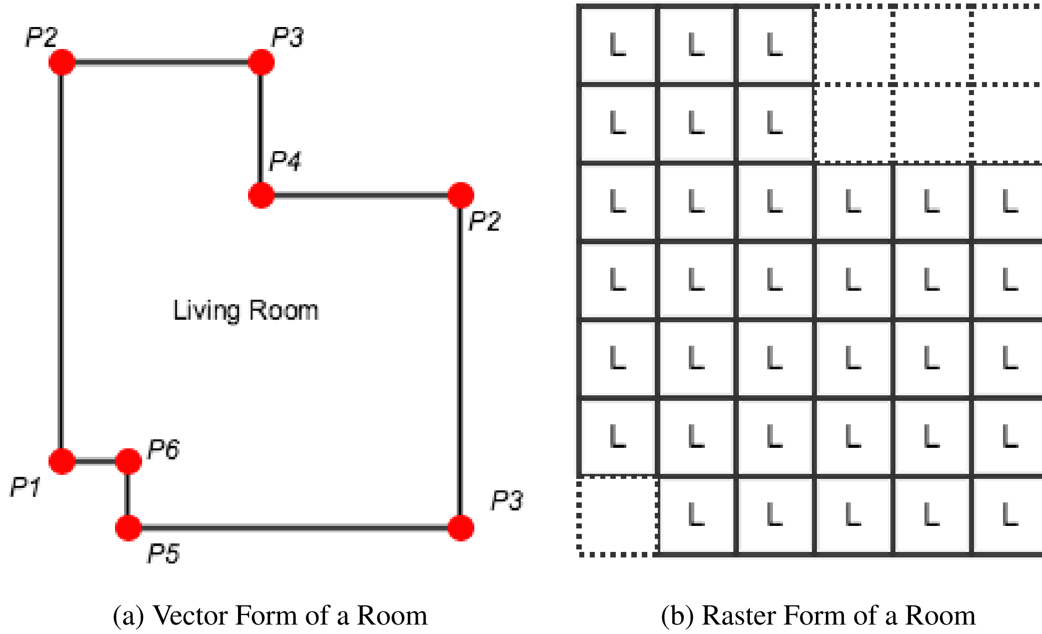
To rephrase, the raster form of a room consists of a set of grid blocks that is a subset of all grid blocks from the boundary of the layout as well as a classifier. The subsets of blocks are mutually exclusive and union to form the full boundary.

2.3.2 The Solution Space

Additional properties of the solution space must be derived before it is possible to choose an effective search strategy. The first that will be considered is the size of the solution space. The size of $\mathcal{F}$ is presented in Lemma 2.3.4 which follows from its definition as containing all possible polygonal partitions of B.

Lemma 2.3.4. *There are uncountably infinite members of the set of possible functional layouts* $\mathcal{F}$ *in standard form for a building.*

Proof. If a specific plot of land represented by a two-dimensional polygon on which a building is to be constructed is partitioned into a known number of rooms with known classifications, then there are uncountably infinite possible partitions as $x \in \mathbb{R}^2$. Figure 2.4 shows an example



(a) Vector Form of a Room (b) Raster Form of a Room

Figure 2.3: Different Mathematical Representations for a Single Room

of this, where two rooms are left to be placed in the layout. Even in this simple example, when all but two rooms have been placed, there are infinite locations along the axis of the outer boundary to place the wall that subdivides them. □

The solution space is reduced if the layouts are in raster form. Lemma 2.3.5 shows the number of possible solutions when the layouts are generated in this form.

Lemma 2.3.5. *The number of possible layouts in raster form in the solution space $\mathcal{F}_z$ where $\mathcal{F}_z \subset \mathcal{F}$ formed by partitioning B into n grid blocks is given by* $\|\mathcal{F}_z\| = \binom{n\|C\|}{n}$.

Proof. If there are n grid blocks, each $c \in C$ can be assigned a maximum of n times. There are thus $n\|C\|$ possible classifiers from which n must be chosen. Therefore the number of possible combinations is given by $\binom{n\|C\|}{n}$. □

As $u \to 0$, $n \to \infty$, $\|\mathcal{F}_z\| \to \infty$ and $\mathcal{F}_z \to \mathcal{F}$. The set of possible raster layouts thus diverges to the set of possible vector layouts as u becomes zero. This result shows that the vector form is a higher level representation that can be employed depending on the assumptions inherent to the specific layout generation problem. An algorithm that provides a solution in vector form can be converted to raster form if some minimum dimension is desired by a user but information

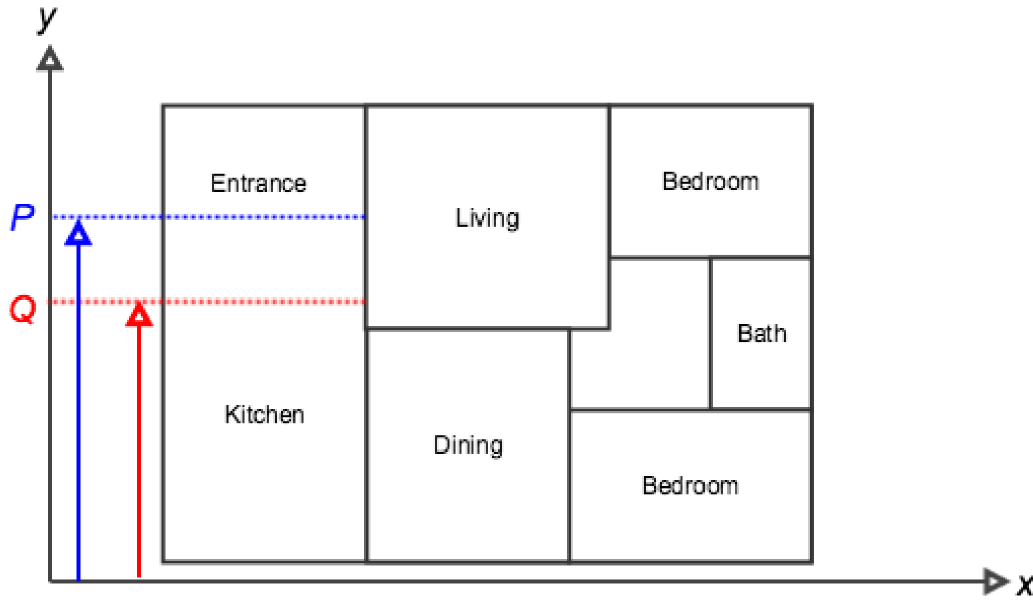


Figure 2.4: Visual example of the result in Lemma 2.3.4. Partitioning the final section into the entrance and kitchen presents uncountably infinite possibilities between $y = P$ and $y = Q$ if $y \in \mathcal{R}$.

is sacrificed in the process. This is the rationale for defining the vector representation as the standard form of a layout.

2.3.3 The Generator Function

The size of the solution space in both raster and vector forms is too large for every element to be examined. In order to solve this problem, the size of the solution space must be reduced in some manner. This will be accomplished with a *generator function* and is defined in 2.3.6.

Definition 2.3.6. *A generator is any function* G *that restricts* $\mathcal{F}$ *to a subset* $\mathcal{F}_G$ *of layouts where* $G : U \to \mathcal{F}_G$ *and* $\mathcal{F}_G$ *is finite.*

The generator function can be any algorithm that either partitions a predefined polygonal boundary for the building or iteratively refines an initial solution from $\mathcal{F}_G$ that is simple to form. The generator function affects the characteristics of the solutions that can be formed, reducing the vast space of possible room arrangements to a finite amount for a given user input.

The use of the generator function introduces the possibility that the global optimum may be excluded from $\mathcal{F}_G$. Due to the variety of types of architectural layouts that can be deemed

acceptable this is unlikely to be a concern for the majority of generator functions. A solution close to the global optimum or even a good local optimum will likely be an acceptable solution to the problem depending on the nature of the generator. The commentary presented when a generator function is proposed should acknowledge this possibility and will only be acceptable if a logical reasoning can be applied to dismiss the concern. This will be further addressed during the derivation of *Evolutionary Treemap*.

2.3.4 Optimizer Functions

A second class of functions can be used in order to alter the characteristics of $\mathcal{F}_G$ by forming a new solution space with superior results $\mathcal{F}_O$. These *optimizer functions* are defined in 2.3.7

Definition 2.3.7. *An optimizer is any function O where $O : F_G \to \mathcal{F}_O \wedge \mathcal{F}_O \subset \mathcal{F}$.*

Optimizer functions are optional components of a layout generation algorithm that can improve the results such that an architect will require less time tailoring the final result to meet a specific need. For example, the corridor placement subroutine in [1] can be considered as an optimizer function. It transforms a layout with rooms that have already been allocated into an improved layout with greater accessibility between rooms. Optimizer functions can increase the possibility that the global optimum of the optimization is found. Throughout the chapter, $\mathcal{F}_S$ will be used to refer to the final restricted solution space formed through the operation of a generator function and any number of optimization functions.

2.3.5 The Cost Function

The solutions in $\mathcal{F}_S$ must be measurable in some manner to quantify their architectural quality or desirability to a user based on their specification. The heuristic defined in 2.3.8 is a class of function that maps members $L \in \mathcal{F}_S$ to quantitative values.

Definition 2.3.8. *A layout generation cost function is any function A that quantitatively measures the aesthetics of a candidate solution h where $A : (L, U) \to h$*

Forming A requires the definition of parameters that can be computed based on the vector representation, or a simplified representation, of a layout that can combine to quantify aesthet-

ics. These parameters are highly specific to the generator function that is chosen for searching through the possible layouts. For example, one generator function may restrict $\mathcal{F}$ by omitting the possibility of forming a layout that contain rooms with very large aspect ratios and thus the aspect ratio of a room will need not be considered in the cost. Another may omit layouts that do not fit within a preallocated boundary resulting in a cost function that need not consider protrusions from the boundary as a parameter.

A layout generation algorithm is thus formed by restricting $\mathcal{F}$ to $\mathcal{F}_S$ through application of a generator function and any number of optimizer functions, then finding the global optimum of A over $\mathcal{F}_S$. The naive layout generation algorithm is thus to simply reduce the space of possible solutions to a finite set and then find the global optimum of the aesthetic cost function mapped to the set. Though this may be accomplished through any well known optimization algorithm, it is likely that, despite a finite number of members, $\mathcal{F}_S$ may be combinatorial in size. Finding the optimum value in these large cases can be accomplished with stochastic algorithms such as simulated annealing and evolutionary algorithms.

2.4 Evolutionary Treemap

The general form of a layout generation algorithm presented in the prior section provides a useful framework for introducing and comparing methodologies for automated building generation. This section will use the general form to introduce the *Evolutionary Treemap* algorithm. This algorithm is proposed to automate building interior layout design with respect to an architectural specification using genetic optimization meta-heuristics. The optimization searches through the space defined by a specialized rectangular boundary partitioning algorithm based on *Squarified Treemap* [10]. The algorithm scales to moderate numbers of rooms making it well suited for generation of small office, school, and hotel buildings. For larger numbers of rooms, the algorithm's runtime must be increased in order to find acceptable solutions. The algorithm consists of three subroutines that will be introduced in the following sections: the *Rectified Treemap*, corridor placement, and evolutionary algorithms.

2.4.1 Rectified Treemap

Evolutionary Treemap is an algorithm that restricts the solution space of possible layouts with the generator function *Rectified Treemap*. This algorithm is closely related to *Squarified Treemap* in that it partitions a rectangular boundary into a set of sub-rectangles with a recursive algorithm [10]. The difference between the two algorithms is that *Rectified Treemap* accounts for user defined aspect ratios as well as area proportions for the partition's child rectangles thereby allowing incorporation of narrow rooms in the partition.

The *Rectified Treemap* algorithm is presented in Algorithm 1. It consists of two subroutines named *rectify* and *maxError*. The first subroutine, *rectify*, is the driver function for placing rectangles within the initial rectangular boundary and is the same as the function that drives *Squarified Treemap* presented in [10]. The *rectify* function accepts a list of rectangles *rects* that have not been placed, which are structures that contain a desired area and aspect ratio for the room, a list of rectangles *row* which have already been placed in the current row and the width of the current row w. *Rectify* places the rectangles within the remaining boundary recursively, finalizing a portion that has been placed based when the maximum error of the aspect ratio of a rectangle in the row has been increased.

The *maxError* subroutine computes the difference between the desired aspect ratio for each rectangle and the aspect ratio that is expressed with the current placement. This measures the error in the room shapes with respect to the user specified area and aspect ratio. The *maxError* function is the difference between the rectified and *Squarified Treemap* algorithms. The algorithm generates the same output as *Squarified Treemap* in the special case where all of the aspect ratios are specified to be one. However, in functional layouts that require narrow rooms, such as closets, the algorithm allows different results than *Squarified Treemap* that will be closer to the desired values.

The *Rectified Treemap* generator function reduces the overall size of the solution space $\mathcal{F}$. The actual size of the restricted space $\mathcal{F}_g$ is still large, however, and thus it is likely that it will contain the global maximum or a good local maximum of the cost function. The size of the space is dependent on the number of rectangles that are placed by the algorithm. The ordering of the list *rects* changes the output of the final layout because the algorithm will always begin

Algorithm 1 Rectified Treemap

```
1: procedure RECTIFY(list rects, list row, w)
2:     c ← head(rects)
3:     if maxError(row, w) ≥maxError([row, c], w) then
4:         rectify(tail(rects), [row, c], w)
5:     else
6:         finalize(row)
7:         rectify(rects, [],width())
8:     end if
9: end procedure
10: procedure MAXERROR(list rects, w)
11:     s ←sum(rects.areas)
12:     m ← −∞
13:     for r ∈ rects do
14:         ratio ←max(w^2 r.area/s^2, s^2/w^2 r.area)
15:         error ←abs(ratio − r.aspectRatio)
16:         if error > m then
17:             m = error
18:         end if
19:     end for
20:     return m
21: end procedure
```

placement of the rooms along the smallest dimension of the input boundary. Since there are $n!$ possible orderings of this list, the size of $\mathcal{F}_g$ is therefore also $n!$. This is large enough that a desirable solution will still be present but small enough that it can be searched effectively. This result will be examined further in Section 2.4.3 when *Evolutionary Treemap's* genetic optimization heuristic is presented.

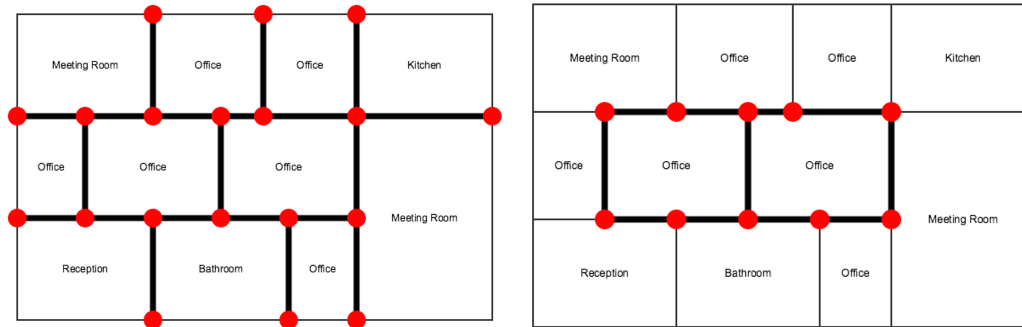
2.4.2 Corridor Placement

The layouts that result from *Rectified Treemap* approximate the input specifications from the user with respect to the aspect ratio and area of each room. The rooms must also be oriented such that each can be accessed. These *connectivity* relationships are not addressed by *Rectified Treemap*. Generating layouts that meet shape and connectivity requirements requires an optimizer function that places a corridor. This algorithm, one of the components of *Evolutionary Treemap*, maps the solution space from the *Rectified Treemap* algorithm $\mathcal{F}_g$ to a new space $\mathcal{F}_o$ where each of the solutions contains a corridor that connects the rooms.

Corridor finding algorithms based on graph theory form the basis of this work. To produce corridors in layouts that contain larger variability in the location and number of rooms, an enhancement based on a spanning tree algorithm is proposed to the graph algorithm presented in [1]. The base algorithm, as presented in [1], finds the interior edges of the layout and considers them to be graph. The algorithm then prunes The vertices that have a degree of one in this graph from the layout. Next, the algorithm subtracts a polygon formed based on the resultant edges from the layout with polygon operators.

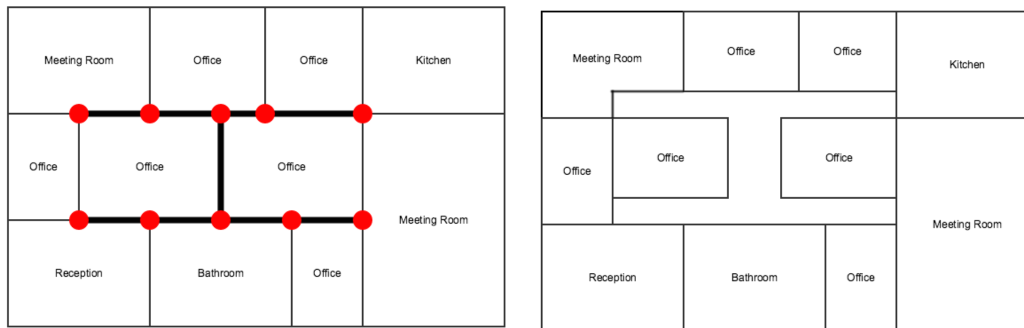
Finding the minimum spanning tree of the corridor graph helps reduce redundant corridor loops in layouts with many rooms. Figure 2.5 introduces the need for the spanning tree algorithm. In this figure, the corridor graph resulting from the pruning algorithm contains redundant loops between rooms. Including these loops in the corridor contributes to wasted space in the layout. These loops can be eliminated by finding a spanning tree in the corridor graph. Typically, the minimum spanning tree is desired as this finds the corridor with the smallest possible area and excludes the loops. Therefore, to enhance the corridor placement algorithm, a minimum spanning tree is found after the pruning stage on the interior edges to identify the

ideal corridor placement.



(a) Initial corridor graph containing all interior edges of the layout

(b) Path after the pruning algorithm from [1] is applied



(c) Path after the spanning tree algorithm is applied

(d) Final layout after corridor graph is clipped from the rooms in the layout

Figure 2.5: In larger layouts, applying the spanning tree algorithm eliminates redundant edges in the corridor to minimize wasted space

Algorithm 2 shows the full procedure that is used in *Evolutionary Treemap* to place the rooms in the layout. This function, *generateLayout*, accepts the boundary of the layout and an ordered list of structs containing room areas and identities, *rooms*. It then forms the final layout by calling *Rectified Treemap* and subtracting a corridor from the resultant set of polygons.

Algorithm 2 Generate Layout

1: **procedure** GENERATELAYOUT(**list** *rooms*, *boundary*)
2: $map \leftarrow rectifiedTreemap(rooms)$
3: $layout \leftarrow placeCorridor(map)$
4: **return** *layout*
5: **end procedure**

2.4.3 Evolutionary Optimization

The function used to generate layouts presented in Algorithm 2 accepts an ordered list of rooms and places them within a boundary. The analysis in Section 2.4.1 showed that the number of possible solutions for a specific list of rooms of size n to the *Rectified Treemap* algorithm is $n!$. Though this solution space is much smaller than infinite, it is still too large in size to be able to express every possible solution. An evolutionary optimization algorithm is proposed to search through this large solution space for an optimal layout.

Evolutionary algorithms model the problem data as a chromosome from the field of biological genetics. The algorithm begins with a population of chromosomes which are then crossed over to generate offspring, subjects the offspring to mutations, and then integrates the offspring into the population. The evolutionary algorithm considers a chromosome to be a specific ordering of the input room vector represented by random keys. This form of evolutionary algorithm, that searches through the permutations of a vector to find an optimal value is similar to solutions that have been proposed to the traveling salesman problem as in [15].

Each random key chromosome corresponds to an input to the *generateLayout* function. These chromosomes represent different permutations of the list of rooms. Algorithm 3 converts chromosomes to layouts so that their aesthetic qualities can be measured by the cost function *costValue*. The conversion algorithm accepts the list of rooms and their sort order, places them with *Rectified Treemap*, clips the corridor, and returns the cost value based on the input list of area values, *areas*, and aspect ratios, *aspect*.

Algorithm 3 Cost Function

1: **procedure** COST(**list** $rooms$, **list** $sortKeys$, **list** $areas$, **list** $aspect$,)
2: $\quad sort(rooms, sortKeys)$
3: $\quad map \leftarrow rectifiedTreemap(rooms)$
4: $\quad layout \leftarrow placeCorridor(map)$
5: $\quad$ **return** $costValue(layout, areas, aspect)$
6: **end procedure**

2.4.4 Measuring the Cost Value

The cost function for the algorithm combines a set of parameters corresponding to the user desired appearance of the layout to compute a value. Since some of the problem's constraints are guaranteed to be satisfied in *Rectified Treemap* layouts, they can be excluded from the cost value calculation. These are examples such as the rooms being contiguous and rectilinear. The parameters that are not guaranteed by the *Rectified Treemap* algorithm are the adjacency relationships and the shapes of the rooms. This is because of area removal in the corridor algorithm, the heuristic nature of the *maxError* placement function and the effect of placement order on adjacency relations. These three combine to produce rooms with aspect ratios, areas, and adjacencies that are not exactly as specified. These parameters highly depend on the ordering of the rooms as they are passed to the treemap function. For example, if two rooms are adjacent in the ordering of the specification, it is highly likely that they will be placed adjacent to each other in the layout. Additionally, the order of the rooms affects the *maxError* placement heuristic and thus the final size of the rooms. It also affects the location and number of the interior edges, which contribute to the corridor.

The cost value must consider the important values that are excluded from the *Rectified Treemap* generator algorithm to produce appealing layouts. It is thus modeled with (2.1), where S is the shape cost function, R is the list of rooms, C is the adjacency cost function, A is the list of adjacencies present in the layout and c_1 and c_2 are positive coefficients chosen by the user.

$$C(R, A) = c_1 S(R) + c_2 A_j(A) \tag{2.1}$$

The shape cost of the layout is calculated by measuring the error between the dimensions of the actual instantiation of the rooms and the user specified dimensions of the rooms as in (2.2). In this function, $U_{area}(r)$ is the user specified area of room r, $U_{aspect}(r)$ is the user specified aspect ratio of room r, $area(r)$ is the actual area of room r and $aspect(r)$ is the actual aspect ratio of room r. If room r is not a rectangle, the aspect ratio of the convex hull of the polygon is used. Lower values of this function correspond to a layout where all of the rooms are close to the correct sizes. This simultaneously attempts to minimize the area that is consumed by the

corridor as layouts that have a large corridor will take too much space that could be allocated in a room.

$$S(R) = \sum_{r \in R} \frac{U_{area}(r) - area(r)}{U_{area}(r)} + \frac{U_{aspect}(r) - aspect(r)}{U_{aspect}(r)} \tag{2.2}$$

The adjacency cost, calculated by (2.3) is simply the sum of the weights for each of the adjacent relationships that is present in the layout. These weights can either be user provided for a specific layout or as part of a larger database for room identities. Lower values of this function represent layouts that have more desirable adjacency relationships which are quantified based on the values of the weights due to the negative multiplier. For each adjacency $a \in A$ containing two room identities, the weight function returns a positive number specified by the user that ranks the importance of the adjacency between the two room identities.

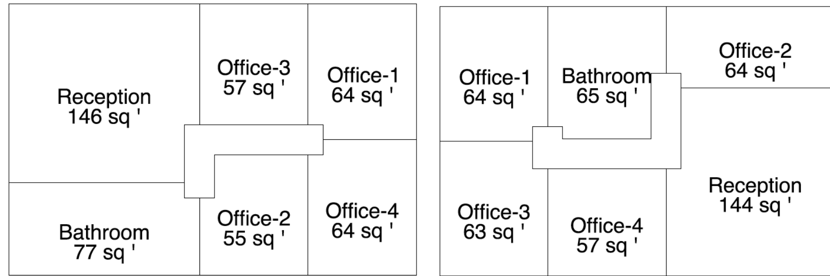
$$A_j(A) = -\sum_{a \in A} weight(a) \tag{2.3}$$

The effects of both of these terms in the cost function is highlighted by Figure 2.6. The user specification for the shapes of the rooms in this layout was a single 8' by 6' bathroom, three 7' by 5' offices and one 10' by 12' reception. An adjacency between the reception and bathroom was given a score of 1 and all others a score of 0. The boundary was 27' by 18'.

When the shape term is excluded from the optimization in Figure 2.6a, the resultant layout does not contain rooms with appealing shapes based on the input specification as the bathroom's final size is much larger than the input. Note that the rooms will all be slightly larger than the inputs because the specified sizes fill 63% of the boundary. When the adjacency term is excluded from the optimization in Figure 2.6b, the reception and the bathroom are not adjacent. When both terms are included in Figure 2.6c, there is a compromise in the final shapes and locations of the room that more closely matches the input.

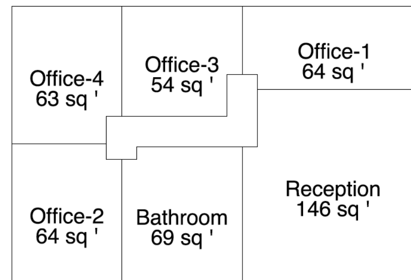
2.5 System Design and Implementation

Evolutionary Treemap was implemented within QCAD [16], an open-source CAD software, to test its performance on real use cases. The design choices for the extension to QCAD were



(a) Optimization with the shape cost removed

(b) Optimization with the adjacency cost removed



(c) Optimization with full cost function

Figure 2.6: Effect of cost function parameters on the generated layout

based on the following set of high-level requirements:

- The software must be fast due to the nature of the stochastic optimization algorithms that are used to compute the location of the rooms in the layout.
- High performance computers must not be required on the client side such that no specialty hardware is required to run the software.
- The software must be designed to function with existing CAD software and is able to plot the resultant layouts on the canvas.
- The software should not be specific to one CAD software allowing the possibility to function with any program through extension interfaces.
- Users should be able to specify the layout with words and numbers in a spreadsheet-like document rather than drawing the shapes for the rooms themselves.

This section explains the design and implementation of the layout planning software including the representation of the user specification, the software architecture of the project,

the object-oriented design of the algorithms library, and the application programming interface (API) for the web server.

2.5.1 Representation of the User Specification

The first stage of designing this software is to specify the types of inputs that will be required to describe the desired layout. These were presented to the user as a spreadsheet and represented in software in JSON format. The first input required by the software is the identity and number of the rooms that are desired in the layout. For example, a user must be able to specify a three bedroom house with one living room and kitchen. The meaning of a room's identity must also be specified by a user. For example, the living room should be associated with a desired shape and size. Finally, the relationship between rooms must be enumerated. For example, the living room and kitchen should be adjacent.

The first item that must be entered by a user is the list of room identities and target shapes. In this representation, the rooms are assumed to be rectangles and the dimensions given for each are the length and width. The length and width of the room both quantify the area and the aspect ratio of the room. This identifies the desired size and shape of the room in the layout.

The second item that must be entered by the user is the desired adjacencies between rooms. This relationship can be represented by a graph, called a connectivity graph, where each room r in R is represented by a vertex and any r_i is connected by an edge to r_j with some w if the rooms are desired to be adjacent. Each weight, w, represents a heuristic measure of the desirability of the connection. They could also be used as identifiers for the type of desired connection, for example doorways or open walls.

If all inputs are considered as hard constraints in the final layout the problem can be over constrained. To reduce the possibility that a layout cannot be found for a given user input, the inputs other than the number of rooms and their identities are treated as soft constraints. The algorithm attempts to match as many as possible but provides no guarantee that they will all be met. Due to the heuristic nature of stochastic optimization, a user can run the program multiple times to yield different results. This is a desirable feature when working with architectural design. A solution that is close to the user's optimum is acceptable because of the editing

facilities provided by the CAD software. A user can edit a layout that is close to their desired specifications to create a design that meets all of their requirements and still spend less time and resources than required by manual efforts.

2.5.2 Software Library Architecture

This component of the software contains the implementation of the *Evolutionary Treemap* algorithm and contains facilities for extending the work to additional layout generation methods. The core algorithms library is implemented in Scala, a Java Virtual Machine (JVM) based language [17]. This language enables rapid development of algorithms and project components due to its concise functional programming syntax. The language also is operable with any library written in Java and benefits from run-time performance enhancement from the JVMs Just-In-Time (JIT) compiler.

The layout generation system was implemented with the architecture shown in Figure 2.7. This system contains the *Generation Strategy* class to manage the generation procedure and the *Blueprint* class to encapsulate the parameters requested by a user including desired connectivity, area and aspect ratio. It also contains the *Generator* interface to encapsulate algorithms such as *Evolutionary Treemap* and the *Tuner* interface to provide a future facility for algorithms that add features to the layout such as furniture, doors, and windows. To enable higher levels of the library to enact classes that implement the *Generator* or *Tuner* interfaces without details of the actual implementation, the class hierarchies follow the Strategy pattern [18]. This allows storage of classes that implement these interfaces in a single container that can be iterated over in a polymorphic manner.

Algorithm 4 shows the *run* method of the *Generation Strategy* class. This method accomplishes the generation of the layouts with two stages. The first transforms the abstract requirements of the *Blueprint* into a concrete set of *Layout* instances in vector form with the *Generator* instances. The second stage of the method passes each vector layout through multiple stages of fine tuning. Each of these stages updates the individual layout with features that more closely matches the user specifications. The final result is an array of vector layouts that are returned to the calling code.

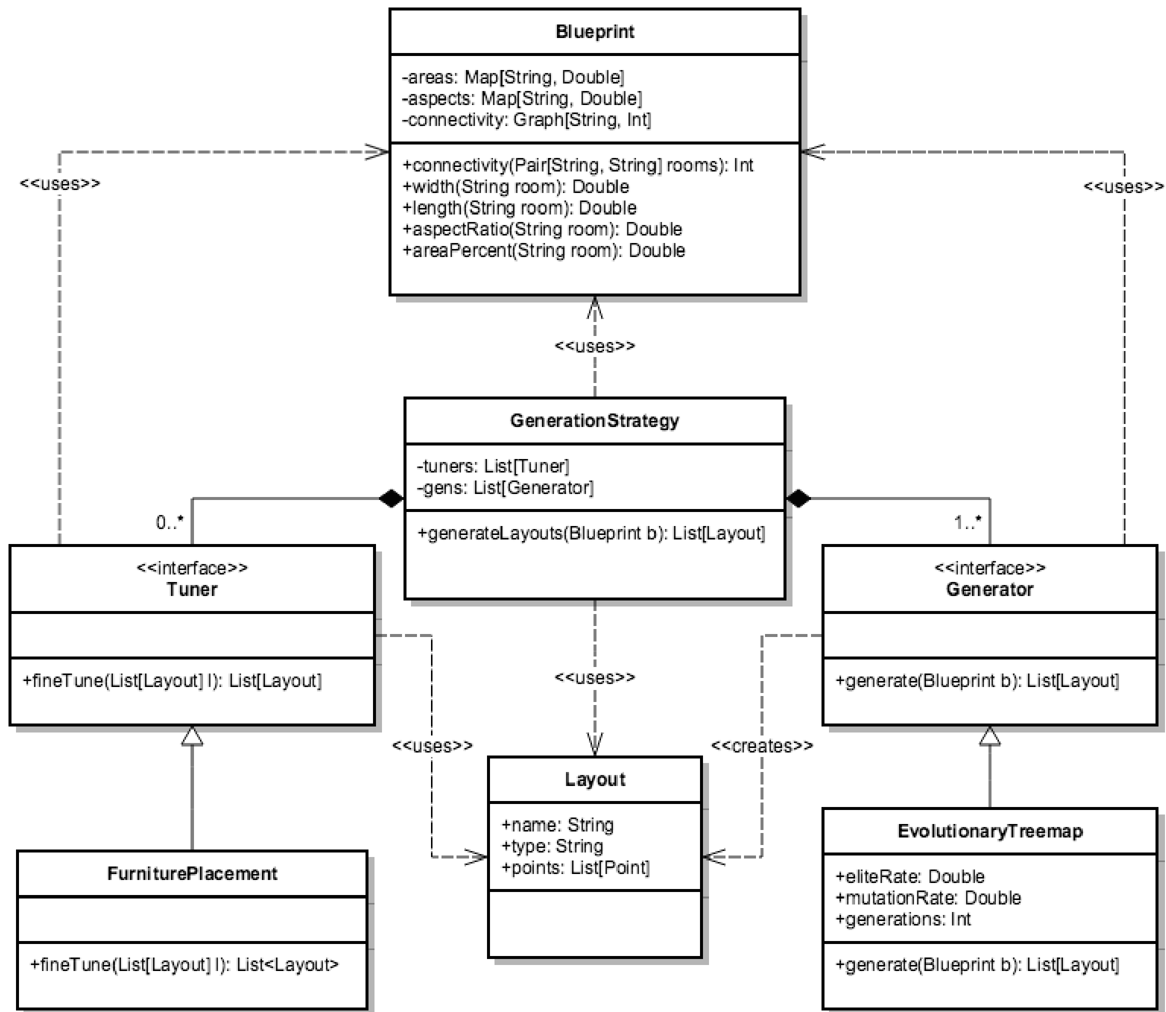


Figure 2.7: UML Diagram of the Layout Generation Software

2.5.3 API Design

A web service was developed for the core algorithms library to facilitate communication with remote clients. This artifact enables client applications to generate layouts by issuing HTTP commands. The web service was implemented in Scala due to the availability of multiple modern web frameworks. This simplified the build configuration and compatibility with the core library. The web service was developed with the Play Framework [19].

The web service is a multi-threaded RESTful application, meaning that no persistent connections are maintained between the service and its clients. Each session is run in a thread pool using the Future programming construct available in the Scala concurrent library [8]. When

Algorithm 4 GenerationStrategy

```
1: procedure GENERATELAYOUTS(blueprint)
2:     layouts ← List[Layout]()
3:     for gen ∈ gens do
4:         layout ← gen.generate(blueprint)
5:         layouts.append(layout)
6:     end for
7:     for layoutinlayouts do
8:         for t ∈ tuners do
9:             layout = t.fineTune(layout)
10:        end for
11:    end for
12:    return layouts
13: end procedure
```

a new session is requested, the service launches a new Future and immediately replies to the client with a unique hash code. Status messages generated by the core library are queued and sent to the client when requested through a specific route. The client can detect when a solution is finished by periodically checking the status messages. A completed solution can be retrieved by an HTTP GET request. This protocol is summarized in Table 2.2.

Table 2.2: Web Service API

HTTP	Route	Action
POST	/session/new	Returns a JSON object with an ID for the new session
DELETE	/session/:id/delete	Deletes a session with ID = id
GET	/session/:id/status	Retrieves the status of the session with ID = id
GET	/session/:id/solution	Retrieves the layouts for session ID = id
POST	/backwards/new/:kernel	Backwards compatibility call for new sessions
GET	/assets/*file	Retrieve static files (such as parameters for the GUI)

The assets route contains a set of static JSON files that contain information about the core library's algorithms including the parameters required for each generator and optimizer. Currently this route only contains files that define a set of configured Generation Strategies that can be chosen by clients. This simplifies the user interface required to work with the library during the development phases.

Table 2.3: Shape Specifications for the Rooms in the Medium Sized Office Layout

Room	Quantity	Length (ft)	Width (ft)	Area (sq. ft)
Server Room	1	6	6	36
Reception	1	12	10	120
Office	2	10	7	70
Lunch Room	1	13	10	130
Large Office	2	10	10	100
Kitchen	1	10	7	70
Conference Room	2	14	10	140
Bathroom	1	10	9	90

2.6 Discussion

Tests of the *Evolutionary Treemap* algorithm as implemented with the architecture described in the prior section show that it generates open-concept layouts in a multitude of styles. Testing the algorithm was done with a custom architectural layout generation system programmed in Scala using the Apache Commons Math Library for genetic optimization [20]. This library provides a random key genetic algorithm that was tailored for use as the optimization meta-heuristic in the *Evolutionary Treemap* implementation. This section will present results from the algorithm for both residential and office layouts.

2.6.1 Office Buildings

The first style of layout is office buildings. The input specification used to generate the office buildings is shown in Table 2.3. This example is representative of a floor in a medium sized office building. The algorithm was set to give a weight of 1 for adjacencies from the kitchen to the lunch room, reception to the bathroom, and conference room to the bathroom. The *Evolutionary Treemap* algorithm was set to run for 10 generations with a population of 500 layouts. The algorithm requires two other parameters; the mutation rate and the elite rate. The mutation rate is the degree at which new chromosomes are randomly modified and the elite rate is the fraction of the chromosomes that are kept from the higher ranks. The mutation rate of the algorithm was set to 0.08 and the elite rate of the algorithm was set to 0.01. Samples of the output that is generated by the algorithm is shown in Figure 2.8.

In both examples in Figure 2.8, the majority of connectivity relationships are satisfied.

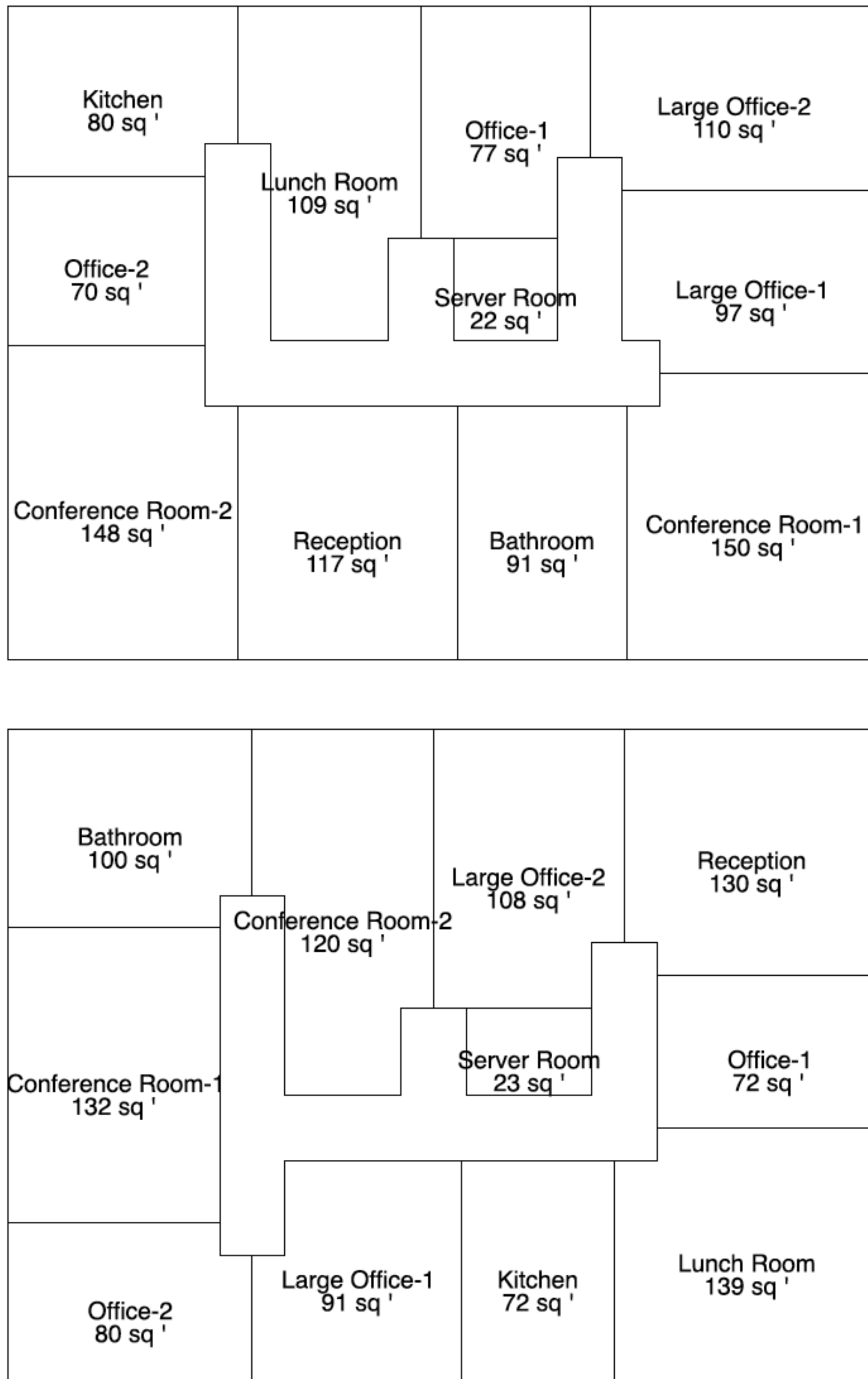


Figure 2.8: Two example office layouts generated by the *Evolutionary Treemap* algorithm based on the specification in Table 2.3

For example, in the first layout the kitchen and lunchroom are adjacent and at the end of the corridor on the top left. The bathroom is also adjacent to one of the conference rooms and the reception. Though this room is not adjacent to the second conference room, the layout is a good compromise between the desired features requested by the user. The second layout is a different style, where the kitchen and lunch room are located in the bottom right corner. The bathroom is now adjacent to both conference rooms and accessible from the hall. In both examples, the rooms are close to the target sizes as specified.

2.6.2 Open-Concept Residential Buildings

The second style of layouts that shown is residential homes. The input specification used to generate these layouts is shown in Table 2.4. A weight of 1 was assigned for adjacencies from the living room to kitchen, kitchen to dining room, living room to dining room, bedroom to bedroom and bedroom to bathroom. The *Evolutionary Treemap* algorithm was again set to run for 10 generations with a population of 500 layouts. The mutation rate of the algorithm was set to 0.08 and the elite rate of the algorithm was set to 0.01. Samples of the output generated by the algorithm for this specification is shown in Figure 2.9.

Table 2.4: Shape Specifications for the Rooms in the Residential Home Layout

Identity	Quantity	Length	Width	Area	Percentage Area
Living Room	1	25	20	500	43.86%
Kitchen	1	15	10	150	13.16%
Dining Room	1	12	10	120	10.53%
Bathroom	1	8	6	48	4.21%
Bedroom	3	10	10	100	26.32%

The examples in Figure 2.9 show that the algorithm is suitable for both office and residential styles. As requested, in the first example, the kitchen and living room are adjacent and the bedroom and bathrooms are accessible through the corridor. The second example has the kitchen, dining and living room all adjacent and meets more of the connectivity requirements. This shows that a user can generate multiple examples from the same specification and select one that suits the style most appropriate for their usage.

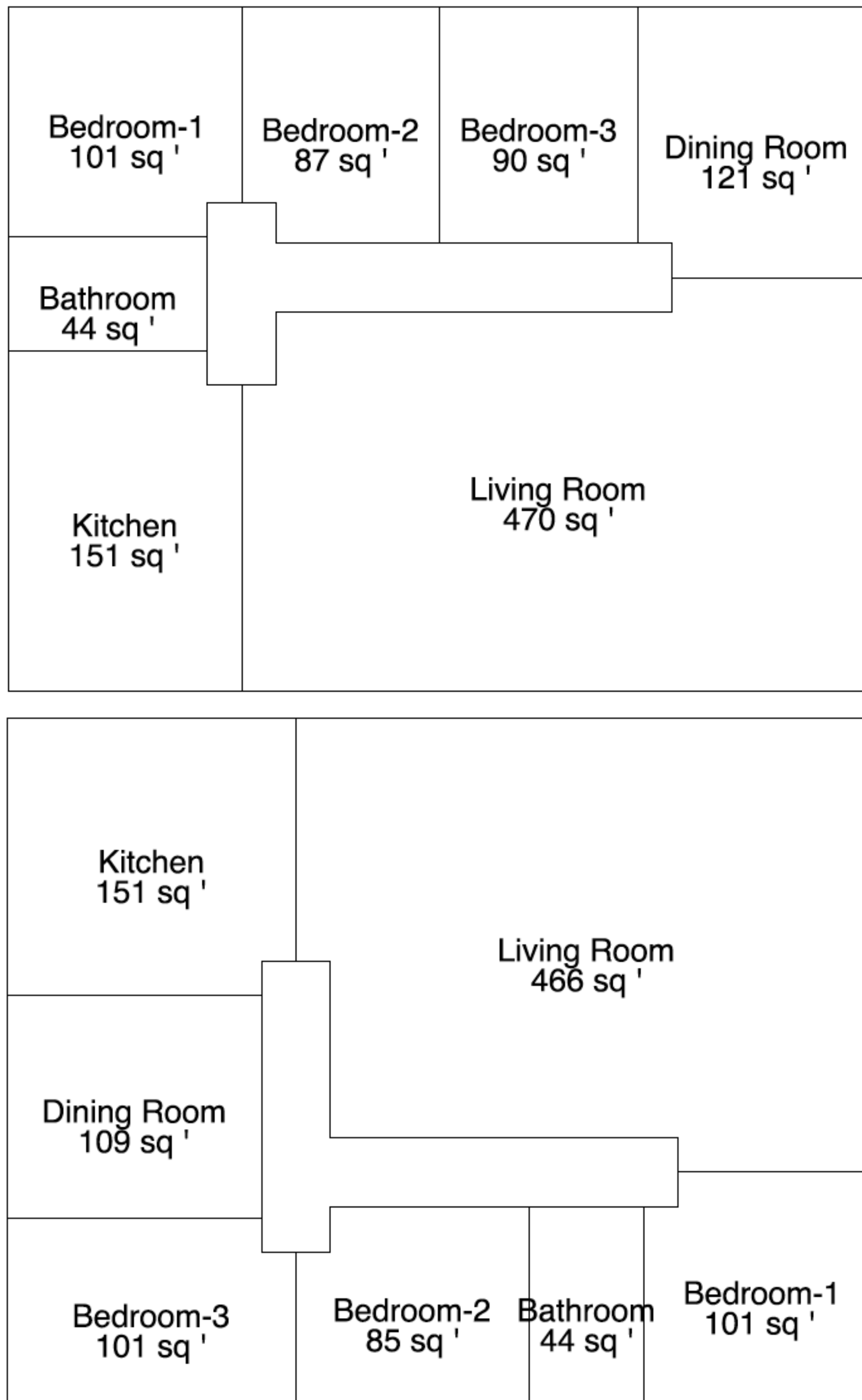


Figure 2.9: Residential layouts generated by the *Evolutionary Treemap* algorithm based on the specification in Table 2.4

2.6.3 Convergence Behavior

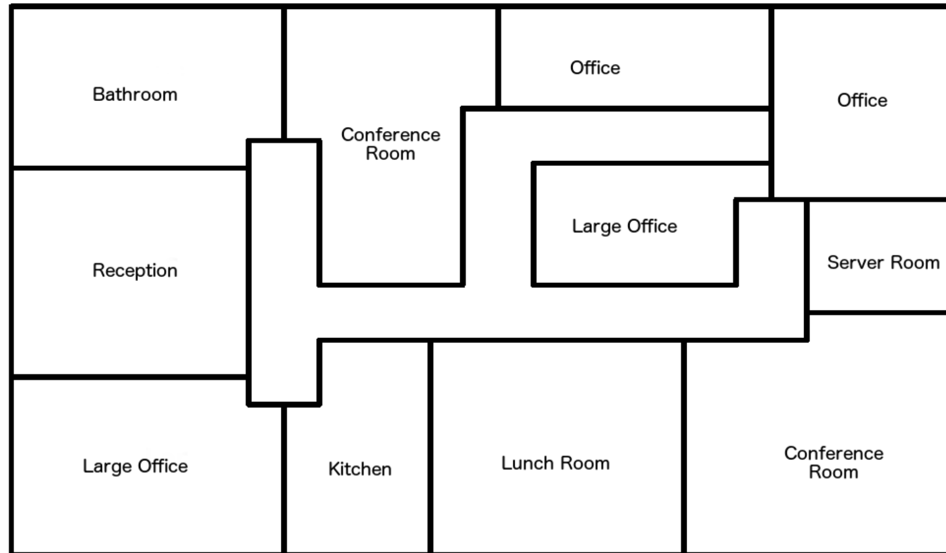
The convergence behavior of a genetic algorithm should be examined in order to identify the speed at which the algorithm reaches an optimum. To identify the convergence behavior of the prior examples, the average cost function value at each generation as well as the lowest cost value at each generation were tracked. Figure 2.10 shows that the crossover operations quickly identify an elite solution to the layout generation problem and that the cost value of the population converges to a smaller value as generations of the algorithm pass. The quick convergence of this algorithm allows a user to iteratively modify their input parameters.

2.6.4 Future Work

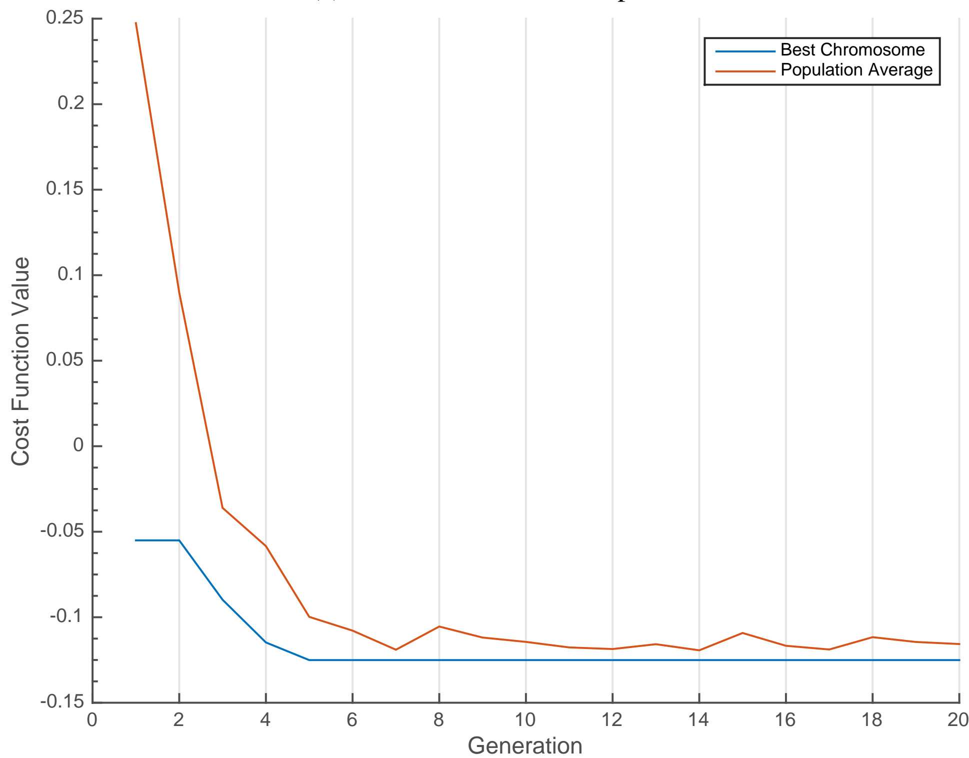
This section outlines the future directions of research into functional layout generation. The first item that will be addressed is two main features of the *Evolutionary Treemap* algorithm that should be introduced increase its applicability to real use cases; fixed structures and non-rectangular boundaries. A simple heuristic is proposed to solve these challenges that works well in some special cases and an open problem is presented that can be a subject of future work. The second item that will be addressed is the corridor optimizer function. Though it works well in smaller buildings, in larger buildings there are often artifacts present that would not be seen in real architecture. The final item that will be presented is commentary on the generation of multi-story buildings which require fixed structures such as stairs and elevators. Though the current literature presents several generation algorithms that are capable of producing single floor layouts, only [12] presented an algorithm that was applicable to multi-store constructions. This challenge will be addressed in terms of the general formulation of the layout generation algorithm.

Fixed Structures and Non-Rectangular Boundaries

Non-rectangular outer boundaries and layouts that contain fixed structures are a more challenging topic in layout generation. These two challenges are related because non-rectangular outer boundaries can be modeled as rectangular boundaries with fixed structures located on the edges and fixed structures can be modeled as boundary polygons that contain holes. As the



(a) Generated Office Floorplan



(b) Algorithm Convergence Behavior

Figure 2.10: Convergence of *Evolutionary Treemap* over twenty generations for an office layout with a mutation rate of 0.08 and an elite rate of 0.01

two problems are interchangeable, this section will address the fixed structure problem. The results and commentary are applicable to the non-rectangular boundaries problem without loss of generalization.

To begin, consider the solution space $\mathcal{F}_f \mid \mathcal{F}_f \subset \mathcal{F}$ where $\mathcal{F}_f$ contains all of the possible layouts that contain a set of fixed structures F_S. These constitute the solutions to a more broad range of applicable user input parameters that contain the specification of fixed structures where $F_S = \{p, c\} \mid p \in P \wedge c \in C$ where p is a polygon with coordinates $x \mid x \in \mathbb{R}^2$ and c is a classifier. The fixed structures may also be involved in connectivity relationships with the other rooms in the layout and thus may be present as a node in the desired adjacency graph. The solution space $\mathcal{F}_f$ is infinite in size and must be reduced in order to be searched for candidate solutions. It is assumed that subtracting each F_S from the boundary polygon to result in a non-rectangular shape is acceptable if and only if each member of F_S is included in the algorithm that detects adjacencies in the layout.

Reducing $\mathcal{F}_f$ requires a generator function that is capable of producing an arrangement of rooms that does not modify the fixed structures and therefore must select the location of each room that has been specified in the input. This generator must function within a non-rectangular space. This excludes the possibility of using a function like treemap for finding the polygons for each room. A function that is capable of partitioning a general non-rectangular shape into a set of N polygons is thus required where the polygons resemble common shapes seen in layouts for each room and are close to the area and aspect ratio constraints provided by the user. The problem is as follows, given a two-dimensional polygon $B \mid B \in \mathbb{R}^2$ find a set of polygons $P_r \mid P_1 \cap P_2 \cap ... \cap P_N = B$, where $\cap$ is the polygon union operator, and $\forall p \in P$ with parameters Q (such as area, width, length, aspect ratio) and user specified target parameters Q_U, that produces the global minimum of $|Q - Q_U|$. Any optimization heuristic that is formulated to solve this problem would likely be highly dependent on the actual location of the fixed structures and the shape boundary and thus the challenge is finding one that is successful for a subset that represents a large percentage of architect designed shapes.

One simple heuristic that could be considered is one that assumes the fixed structures are located on the edge of the boundary and are proportionally small. In this case, the boundary can simply be assumed to be rectangular and the problem can be solved by using an augmented

Evolutionary Treemap algorithm. The augmentation is an added optimizer function that subtracts the fixed structures from the polygons in which it is contained after the generation of the layout. An example of a layout generated with this algorithm is shown in Figure 2.11.

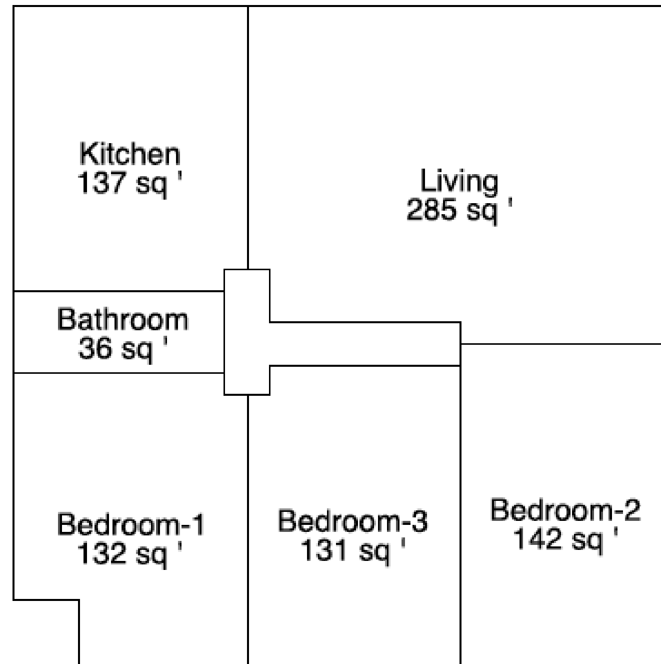


Figure 2.11: An example layout with a non-rectangular boundary generated with the augmented *Evolutionary Treemap* algorithm

This heuristic however does not account for structures that are not on the edge, and thus is not applicable for test cases of this form. It also would perform poorly for fixed structures that are large as entire rooms could be removed from the layout during the operation of the optimizer function. The heuristic's dependence on the output shape could be resolved in a brute force manner, for example with a set of optimizer functions that augment *Evolutionary Treemap* or one of the other generation algorithms proposed in literature that handle mutually exclusive output shapes. Such a system could be driven by a computer vision algorithm that is trained on features of the boundary polygons and directs the selection of a heuristic. Though this is not an elegant solution, it could increase the space of possible input problems that are solved when increased generality is desired.

Corridor Optimizer Function

The corridor placement algorithm presented in Section 2.4.2 was shown to perform well in smaller layouts that did not contain large sets of loops in the interior edges of the layout. As the size of the layout increases the corridor algorithm can occasionally cause artifacts in the final layout that may be unnecessary. For example, the corridor place by the algorithm in Figure 2.12 has a loop highlighted in red that may or may not be deemed necessary by a designer and would add to the material cost of a building. Applying the spanning tree algorithm would eliminate this loop, however this may not be desired. Building codes for fire exits and accessibility may require loops in certain regions. This concern could be addressed with a second stochastic optimization algorithm for the shape of the corridor. This algorithm would be run at each iteration of the *Evolutionary Treemap* algorithm and, though this would increase the overall complexity of the algorithm with direct consequences to the runtime, it would potentially provide less artifacts that would not be found in a real building.

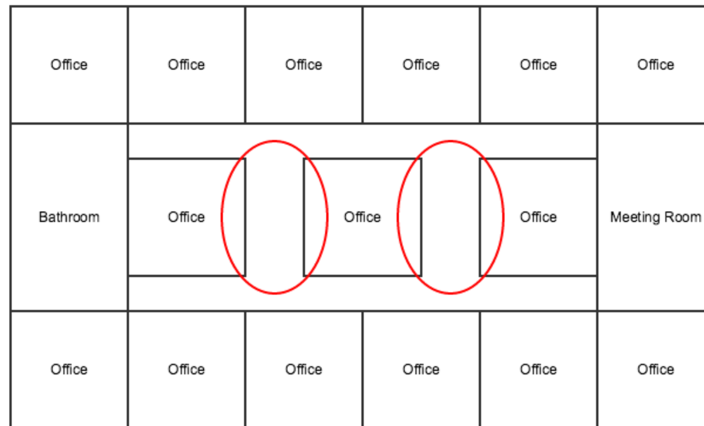


Figure 2.12: An example of a corridor that contains loops which may or may not be necessary depending on the designer preferences

An example of such a stochastic optimization algorithm for the corridor could be based upon a boolean vector **b** where $||\mathbf{b}|| = k$ and k is the number of interior edges in the layout. In this representation, an interior edge k_i would be included in the shape that forms the corridor if and only if the corresponding b_i is true. The possible values of this vector thus form a solution space C of potential corridors that can be placed in the layout where $||C| \; = 2^k$. If the desired features of the corridor can be expressed in a cost function that is subject to some user input

then a generic stochastic optimization can be used to search C for the optimum.

Multi-Story Buildings

Generating a multi-story building is a challenge due to the fact that each floor of the layout must contain elements whose location is fixed for the entire building. For example, elevator shafts and stairways must be located on the same place on each floor. Other elements, such as washrooms and kitchens, may also require similar locations to minimize the cost of pipes associated with the plumbing of the building. The challenge is furthered by the fact that a generation algorithm that attempts to allocate space within an existing building will treat these rooms as fixed structures whereas an algorithm that generates the whole building may be able to treat the constraints on the shapes of these objects as fuzzy. Though the algorithms described in Section 2.2 are all able to produce quality layouts for single floor buildings, only the simulated annealing algorithm proposed in [12] was able to produce multi-story houses. No author has yet claimed an algorithm that can generate an entire building with many levels within one optimization. This section addresses possible heuristic methodologies that could increase the number of layouts that can be generated in tandem for a single building.

Solving a building that contains n levels requires the generation of a set of layouts, L, of size n where each floor shares a set of fixed structures, F_S, each of which has a location defined by a polygon, p, that is common for each level. It is assumed that p is provided for all F_S. With this assumptions, the algorithm can consider each fixed objects individually for each layout. It is also assumed that the user has specified the rooms that are desired on each level of the building and their classifiers, but that the polygons are unknown. The problem is thus to find the location for each room in each layout. If these assumptions are made, the multi-story problems becomes recursive sub-problems of layout generation with fixed structures. The heuristic presented for managing fixed structures can then be applied if the conditions are met. This algorithm would have a theoretical complexity of $O(n \times f)$ where f is the complexity of the layout generation algorithm.

2.7 Conclusion

Architectural layout generation accelerates the time required to produce CAD drawings for a construction or virtual reality project. Algorithms that generate layouts can be represented by a general form that simplifies comparisons of the results produced. *Evolutionary Treemap* is an algorithm that generates layouts in multiple styles and is based on a genetic algorithm that shows good convergence behavior. The results produced by *Evolutionary Treemap* differ from others in literature due to the generality of the results. Extending this work to allow generation of more realistic layouts requires facilities for generating multi-floor layouts, layouts with fixed structures and improving the corridor generation algorithm. Thus the *Evolutionary Treemap* algorithm is a useful algorithm that can be implemented within CAD software to provide users with an accelerated methodology to produce drawings for their projects.

Bibliography

[1] Maysam Mirahmadi and Abdallah Shami. A novel algorithm for real-time procedural generation of building floor plans. abs/1211.5842, 2012.

[2] The Royal Architectural Institute of Canada. Determining appropriate fees for the services of an architect, 2009. http://www.mbarchitects.org/docs/guide_architectservicefees(e).pdf.

[3] Autocad: Design every detail, 2015. http://www.autodesk.com/products/autocad/overview.

[4] Revit, 2015. http://www.autodesk.com/products/revit-family/overview.

[5] Solidworks, 2015. http://www.solidworks.com/.

[6] University of Colorado Boulder. Design development phase (dd), 2015. http://www.colorado.edu/fm/design-development-phase-dd.

[7] CD Projekt Red. The witcher 3 - wild hunt, 2015. http://thewitcher.com/witcher3/.

[8] Valve Software. Left 4 dead 2, 2008. http://www.valvesoftware.com/games/l4d2.html.

[9] Ubisoft. Assassin's creed syndicate, 2015. http://assassinscreed.ubi.com/en-ca/home/.

[10] M. Bruls, K. Huizing, and J.J. Van Wijk. Squarified treemaps. In *Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization*, pages 33 – 42, 2000.

[11] F. Marson and S.R. Musse. Automatic real-time generation of floor plans based on squarified treemaps algorithm. pages 624817 (10 pp.) –, 2010.

[12] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. In *ACM Transactions on Graphics*, volume 29, 2010.

[13] MacHi Zawidzki, Kazuyoshi Tateyama, and Ikuko Nishikawa. The constraints satisfaction problem approach in the design of an architectural functional layout. 43(9):943 – 966, 2011.

[14] Darcy Chia and Lyndon While. Automated design of architectural layouts using a multi-objective evolutionary algorithm. In Grant Dick, WillN. Browne, Peter Whigham,

Mengjie Zhang, LamThu Bui, Hisao Ishibuchi, Yaochu Jin, Xiaodong Li, Yuhui Shi, Pramod Singh, KayChen Tan, and Ke Tang, editors, *Simulated Evolution and Learning*, volume 8886 of *Lecture Notes in Computer Science*, pages 760–772. Springer International Publishing, 2014.

[15] Lawrence V. Snyder and Mark S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, 174(1):38 – 53, 2006.

[16] QCAD - 2D CAD for windows, linux and mac, 2015. http://www.qcad.org/en/.

[17] The scala programming language, 2015. http://www.scala-lang.org/.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[19] Play: The high velocity web framework for java and scala, 2015. https://www.playframework.com/.

[20] Apache commons math: Genetic algorithms, 2015. http://commons.apache.org/proper/commons-math/userguide/genetics.html.

Chapter 3

Parallel Linear Programming with Dense Data Structures

3.1 Introduction

The *Simplex Algorithm* is an iterative optimization algorithm that solves linear programming problems. Though industrial implementations of *Simplex* with sparse data structures exploit properties of practical problems to improve performance, an efficient parallel sparse algorithm has yet to be discovered [1]. Dense forms of the algorithm exhibit data-level parallelism that can be implemented for parallel processors to obtain speed up. This work proposes an energy-efficient hardware accelerated dense Linear Programming (LP) solver based on the *Simplex Algorithm* for solving LP problems. The system is operable on Field Programmable Gate Arrays (FPGAs), Graphic Processing Units (GPUs), and multi-core computer processing units (CPUs). The system is targeted towards dense linear programming problems found in radiotherapy treatment planning applications as they represent a challenge to modern solvers. The implementation differs from others in literature as it is based on the *Dictionary Simplex Algorithm* rather than the traditional Tableau algorithm, contributing to a reduced memory consumption.

Performance benchmarking for the linear programming system on randomly generated problems with dense matrices reveals speed ups relative to a sequential implementation that approach two and ten times faster on a CPU and GPU respectively. The FPGA exhibited no

speed gain but proved to be the most efficient with respect to Simplex iterations processed per unit energy with an efficiency 5 times greater than a CPU implementation. This is a notable speed improvement and power saving in comparison with current technology for solving dense problems because the GPU code can solve problems with speeds up to 50 times faster than an open-source sparse solver. This shows that the dense implementation performs well for problems that are not well suited for a sparse solver.

This chapter is organized as follows: The first section summarizes the literature available on parallel implementations of the *Simplex Algorithm*, the second introduces the prerequisite background information on the mathematical notation and algorithms of linear programming, the third presents details of the OpenCL implementation of the algorithm, and the fourth discusses benchmarking results.

3.2 Literature Review

The *Simplex Algorithm* is used to solve a class of optimizations known as linear programming problems. Linear programming problems are maximization or minimization problems for a function composed of a sum of weighted decision variables bounded by a set of linear constraints. These problems can be solved by many different versions of the *Simplex Algorithm* that have been proposed and refined in order to handle issues that occur during the solve process such as numerical instability and degeneracy. Though the refinements discovered in the past century of linear programming research increased the size of tractable problems, many practical linear programs cannot be solved within a practical time limit - their computational complexity surpasses software capability. This section summarizes the efforts in literature to eliminate the performance barrier imposed by software through implementation of the algorithm at a lower level.

The majority of literature on accelerating the *Simplex Algorithm* with parallel computing focuses on the dense variant of the algorithm. As the algorithm requires a set of matrix calculations that operate in series, the algorithm can be implemented based on sparse linear algebra or dense linear algebra depending on the properties of the problems from the desired application. The algorithm consists of three subroutines, pricing, ratio test and pivot, which are

each matrix operations on the problem data. Though the sparse form of the algorithm is the traditional highest performer due to the structure of practical problems, the micro-parallelism of dense Simplex coupled with advances in parallel computing resulted in renewed interest in the dense form. Literature that proposes hardware acceleration of the dense algorithm reports performance that surpasses sparse software for small test sets [2, 3].

Dense linear programming problems, though rare in practice, are found in the field of radiotherapy treatment [4, 5]. The radiotherapy LP problem requires computing configuration parameters for a set of beams that are aimed at a patient's treatment area. Since the radiation delivered by many beams will overlap to cover the full treatment area, the dosage matrix of the LP model can be dense. This type of problem forms the motivation for a hardware accelerated dense linear programming algorithm.

At present, the powerful hardware components in heterogeneous computing systems such as GPUs are left idle while linear programming software performs simplex iterations on the computer processor. Recent literature shows that these resources can be utilized to accelerate the algorithm. An FPGA implementation developed in [3] was used to solve small scale linear programming problems and reported acceleration when compared industry leading simplex software. An implementation utilizing a GPU reported comparable acceleration for large, randomly generated problem sets [2]. This performance increase warrants further study of the dense algorithm with modern parallel computing systems.

Not only would a parallel *Simplex Algorithm* advance the field of linear programming, it would also find direct application within integer linear programming solvers [6]. Integer linear programming is a related form of optimization in which the problems contain integer variables. These problems are solved through many iterations of the *Simplex Algorithm* and thus require high performance computing resources. The high resources required for this computation renders problems of modest size intractable. Access to a faster *Simplex Algorithm* would increase the size of solvable ILP problems in industries from network design [7] to power plant maintenance [8].

OpenCL [9] is a programming language that can accelerate the *Simplex Algorithm* for multiple platforms. An OpenCL implementation of Simplex optimized for GPUs in literature demonstrated acceleration of over 20 times the sequential version [2]. This chapter compares

the performance of the dictionary *Simplex Algorithm* on a GPU and an FPGA. This is possible with the recently introduced FPGA OpenCL SDK [10]. Literature providing similar comparisons for video compression and information filtering highlight the differences in performance and power of the platforms [11, 12]. A technology that provides a green solution could reduce energy expenses and cooling requirements of large scale server farms used to solve integer linear programming problems in financial, traffic and general data analysis. The application dependent analysis of performance versus power provided for the *Simplex Algorithm* gives insight into hardware linear optimization capabilities.

3.3 Background

Linear programming algorithms find the extreme value of a linear objective function subject to constraints. The mathematical form of a linear programming problem is (3.1), where $\mathbf{c}$ is the objective function coefficients, $\mathbf{x}$ is the decision variables, $\mathbf{A}$ is the constraint system and $\mathbf{b}$ is the value of the constraints. The problem is assumed to have m constraints and n decision variables. In this chapter a minimization problem is considered without loss of generality.

$$\begin{aligned} \min \ & \mathbf{cx} \\ \text{s.t.} \ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x}, \mathbf{b} \geq 0 \end{aligned} \tag{3.1}$$

Solving a linear programming problem requires identifying the extreme value of the cost function and the value of the decision variables at that solution. The following definitions are required to understand the solution of a linear programming problems:

Definition 3.3.1. *A Feasible Solution is a set of variable values,* $\mathbf{x}$*, that satisfies the constraints.*

Definition 3.3.2. *The Feasible Region is the convex polyhedron in* $n+m$ *dimensions formed by the constraint equations containing all feasible problem solutions.*

Definition 3.3.3. *A Basic Feasible Solution (BFS) is a feasible solution at an intersection of*

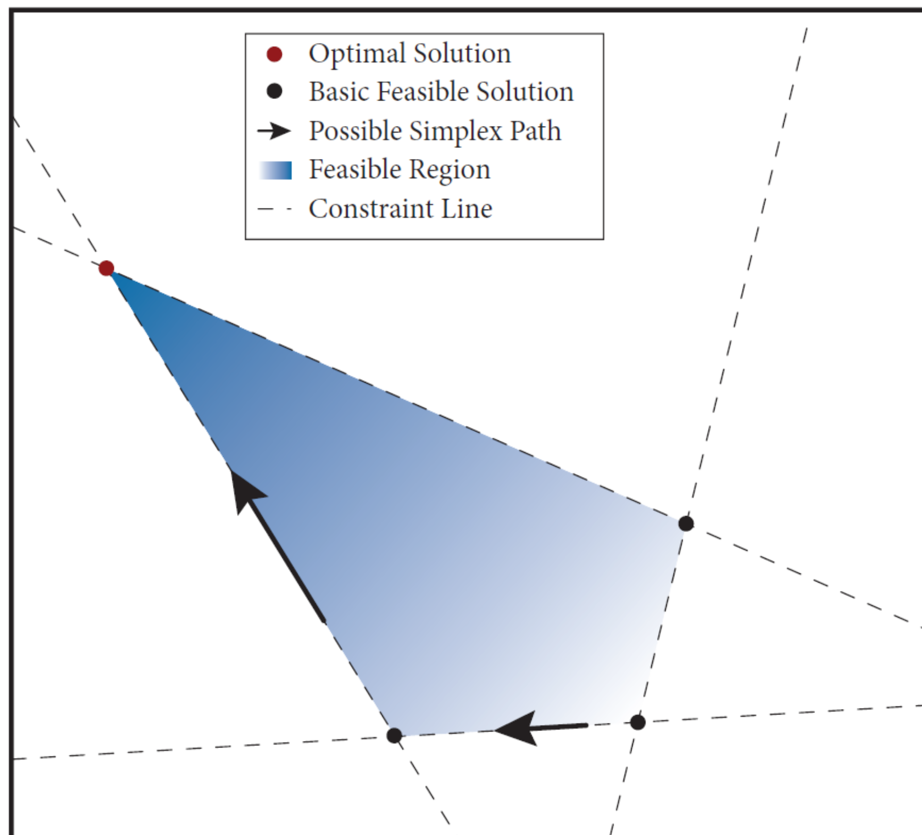


Figure 3.1: Visualization of the feasible region in a simple linear programming problem with two decision variables and four constraints. Arrows indicate a possible Simplex path.

constraints. The optimal value of any linear programming problem is always basic feasible solution [13].

Algorithms that solve linear programming problems iteratively check feasible solutions for optimality. The *Simplex Algorithm* restricts this search to BFS and traces a path along the edges of the feasible region towards the optimal solution. At each iteration, a new BFS is chosen to improve the objective by replacing a variable with poor objective contribution with a better candidate. These variable changes propel the algorithm along a geometric path as shown in Figure 3.1. Adjacent vertices of the convex set are tested until the optimum is found.

3.3.1 The Dictionary Simplex Algorithm

This section summarizes the *Dictionary Simplex Algorithm*, a dense matrix algorithm for solving linear programming problems to introduce terminology for later sections. Since the intent was to evaluate the performance of the *Simplex Algorithm* on hardware devices and OpenCL design methodology for application in high performance computing environments, problems of type (3.1) were used to develop and test the algorithm. Though practical problems contain additional types of constraints and variables, linear programming literature often analyses this form of problem to motivate concepts [14]. Studying this form provides initial insight into the dictionary algorithm performance and OpenCL's capability as a cross-platform solver. The work reported here is intended as an initial step towards a full design. The intent was to measure the actual speed-up from a GPU and FPGA with simplified problems so that an upper bound on performance was available. The contribution of this chapter is profiling that reveals this upper bound.

The Augmented Linear Programming Problem

Problem (3.1) requires modification prior to processing with Simplex. A modified form, (3.2), where $\hat{\mathbf{A}}$ represents an augmented constraint system with m additional variables, $\mathbf{z}$, that change the inequality constraints to equality constraints is required. These variables are referred to as slack variables because their values represents the distance between the constraint system and its bounds.

$$\begin{aligned} \min\ & \mathbf{cx} && && (3.2)\\ \text{s.t.}\ & \hat{\mathbf{A}}\hat{\mathbf{x}} = \mathbf{b} && \hat{\mathbf{A}} = [\mathbf{A}\ \ \mathbf{I}] \\ & \hat{\mathbf{x}} \geq 0 && \hat{\mathbf{x}} = [\mathbf{x}\ \ \mathbf{z}] \end{aligned}$$

There is a second reason for augmenting the constraint system with $\mathbf{z}$. The coefficients of $\mathbf{z}$ in $\hat{\mathbf{A}}$ form an invertible matrix to initiate the *Simplex Algorithm*. This is the initial BFS used to begin traversal of the feasible region.

The Optimal BFS

Consider a partition of $\hat{\mathbf{x}}$ into sets $\mathbf{x_B}$ of size m and $\mathbf{x_N}$ of size n called the basic and the non-basic variables. The columns of the problem data corresponding to these variables are differentiated using subscripts B and N. The partition of $\hat{\mathbf{A}}$ is represented by $\mathbf{B}$ and $\mathbf{N}$ for notational simplicity.

For the considered linear programming problem, the following theorems apply (refer to [13] for proofs):

Theorem 3.3.4. (Basic Feasibility) *A partition of* $\hat{\mathbf{x}}$, $[\mathbf{x_B}, \mathbf{x_N}]$ *where* $\mathbf{x_N} = 0$, $\mathbf{x_B} = \mathbf{B}^{-1}\mathbf{b}$ *and* $\hat{\mathbf{x}} \geq 0$ *is a BFS.*

Theorem 3.3.5. (Optimality) *The optimal BFS to the linear programming problem satisfies* $\mathbf{c_N} - \mathbf{c_B}\mathbf{B}^{-1}\mathbf{N} \geq 0$.

Three mechanisms based on Theorems 3.3.4 and 3.3.5 compose the *Simplex Algorithm*. The first two, Pricing and The Ratio Test, choose two variables to exchange positions and form a partition of $\hat{\mathbf{x}}$ with a lower value of $\mathbf{cx}$. The variable that leaves the basic set is called the *leaving variable* and is replaced by the *entering variable*. The third, Pivot, updates the problem data for the new solution. Figure 3.2 shows an overview of the iterative procedure formed by applying these three algorithms in sequence to generate improving solutions.

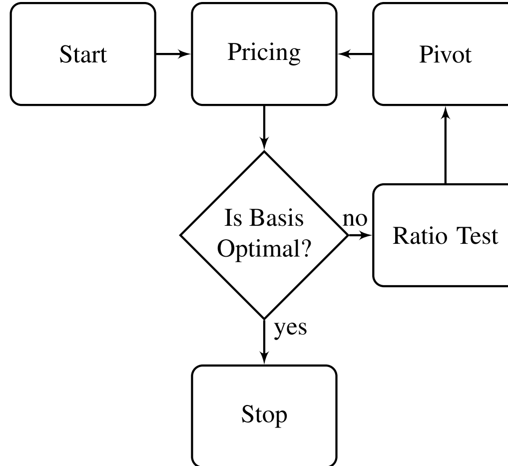


Figure 3.2: The Stages of the *Simplex Algorithm*

The Dictionary Data Structure

The dictionary algorithm combines Theorems 3.3.4 and 3.3.5's critical data in structure (3.3), where the top row is the result of the optimality check, or *reduced costs*, the right-most column is the basic variable values and the top right corner is the current objective value. The three interior algorithms improve the current solution using values of $\mathbf{B}^{-1}\mathbf{N}$ to choose the entering and leaving variables. This structure is referred to as the dictionary.

$$\mathbf{D} = \begin{pmatrix} \mathbf{c_N} - \mathbf{c_B}\mathbf{B}^{-1}\mathbf{N} & \mathbf{c_B}\mathbf{x_B} \\ & \\ \mathbf{B}^{-1}\mathbf{N} & \mathbf{x_B} \end{pmatrix} \tag{3.3}$$

The Initial Partition

Simplex begins an iterative search for the optimum with a starting BFS. Advanced linear programming solvers calculate this with a Phase 1 algorithm [15]. However, for the problem type under consideration, the partition $[\mathbf{x_B}, \mathbf{x_N}] = [\mathbf{z}, \mathbf{x}] = [\mathbf{b}, \mathbf{0}]$ satisfies Theorem 3.3.4 and can act as an initial suboptimal solution.

With all variables equal to zero, $[\mathbf{c_B}, \mathbf{c_N}] = [\mathbf{0}, \mathbf{c}]$, $[\mathbf{B}, \mathbf{N}] = [\mathbf{I}, \mathbf{A}]$ and the initial dictionary is (3.4). It is important to note that the interior of the dictionary contains $\mathbf{A}$ at this step, the initial constraint system of order $m \times n$. This feature of the dictionary algorithm will be analyzed

Section 3.3.1.

$$\mathbf{D} = \begin{pmatrix} \mathbf{c} & \mathbf{0} \\ & \\ \mathbf{A} & \mathbf{b} \end{pmatrix} \tag{3.4}$$

The Pricing Algorithm

Pricing refers to the procedure used to select the entering variable. Negative reduced costs in the first row of (3.3) correspond to variables that violate the condition in Theorem 3.3.5 and will improve the objective if chosen to enter. Any violating variable can be chosen. The employed selection strategy dictates the length of the path traced by Simplex. Advanced pricing algorithms in literature attempt to choose wise candidates and shorten this path [15] .

Access to many computational cores adds an additional design consideration to the selection of the pricing algorithm. Most practical linear programming solvers use the idea of partial pricing to reduce iteration processing time. This technique examines a subset of the reduced costs to choose an entering candidate. There are opportunities for parallel pricing algorithms that use multiple cores to examine several sets of reduced costs or to accelerate the search through a single subset of values. The only way to justify such algorithms is through empirical evidence. The best algorithm for pricing in each problem is still an open question.

This design uses the pricing algorithm called steepest descent that chooses the variable with the minimum of all reduced costs to enter the basis. Although the algorithm is susceptible to slow improvement in some problems [13], its efficient parallel form led to inclusion in the dictionary algorithm.

The Ratio Test

The leaving variable is calculated by the ratio test. The non-basic variable at the index of the minimum positive value in $\mathbf{x_B}$ divided by the entering column is selected. The objective function is decreased by the product of the minimum ratio and the entering reduced cost.

The ratio test detects if a problem is unbounded. This occurs when the feasible region of the problem is not a closed set. If there is no leaving candidate, the objective has no lower limit

[15].

The Pivot Algorithm

Algorithm 5 calculates the dictionary structure for the new solution, where the index of the entering and leaving variables are represented by *enter* and *leave*. Combining the strategies for updating the reduced costs, objective value, basis variable values, and $\mathbf{B^{-1}}$ presented in [15] creates an efficient parallel form of the pivot algorithm.

The first stage of the algorithm stores the leaving row, *pRow*, entering column, *pCol*, and the value of the dictionary at the intersection of *pCol* and *pRow*, *pElem*. The subsequent nested loop requires these values and they could be overwritten if not stored before parallel execution.

The values of the dictionary do not require calculation through the equations in (3.3) for each iteration, although this is a strategy that could be used to increase numerical stability of a practical solver.

The Size of the Dictionary

Memory usage is a crucial design consideration for linear programming solvers because practical problems can contain a large number of decision variables and constraints. Furthermore, future ILP solvers based on the implementation could store and process more linear relaxations of a problem in parallel as part of a branch and bound framework.

The *Tableau Simplex Algorithm* is the common algorithm implemented in dense form [14]. It relies on structure (3.5) for the internal computations. Note the presence of $\hat{\mathbf{A}}$ in place of $\mathbf{N}$. Compared to (3.3), this structure has m additional columns that lead to $O(m^2)$ additional calculations. (3.6) and (3.7) show that this extra memory is redundant.

$$\mathbf{T} = \begin{pmatrix} \mathbf{c} - \mathbf{c_B}\mathbf{B}^{-1}\hat{\mathbf{A}} & \mathbf{c_B}\mathbf{x_B} \\ & \\ \mathbf{B}^{-1}\hat{\mathbf{A}} & \mathbf{x_B} \end{pmatrix} \tag{3.5}$$

The variables already in the basis cannot be entering candidates, and as such, their reduced costs are zero. (3.6) shows that these values are explicitly stored in $\mathbf{T}$ and can be removed.

Algorithm 5 Pivot

```
1: procedure PIVOT(enter, leave, D, pCol, pRow)
2:     pElem = D[leave, enter]
3:     parallel for i = 1 to m + 1 do
4:         if i = leave then
5:             pCol[i] = 0
6:         else
7:             pCol[i] = −D[i, enter]/pElem
8:         end if
9:     end parfor
10:    parallel for i = 1 to n + 1 do
11:        pRow[i] = D[leave, i]
12:    end parfor
13:    parallel for i = 1 to m + 1 do
14:        parallel for j = 1 to n + 1 do
15:            if i = leave and j = enter then
16:                D[i, j] = 1/pElem
17:            else if j = enter then
18:                D[i, j] = pCol[i]
19:            else if i = pRow then
20:                D[i, j]/ = pElem
21:            else
22:                D[i, j]+ = pRow[j] ∗ pCol[i]
23:            end if
24:        end parfor
25:    end parfor
26: end procedure
```

$$\begin{aligned} \mathbf{c} - \mathbf{c_B}\mathbf{B}^{-1}\hat{\mathbf{A}} &= [\mathbf{c_B} \quad \mathbf{c_N}] - \mathbf{c_B} \times [\mathbf{B}^{-1}\mathbf{B} \quad \mathbf{B}^{-1}\mathbf{N}] \\ &= [\mathbf{0} \quad \mathbf{c_N} - \mathbf{c_B}\mathbf{B}^{-1}\mathbf{N}] \end{aligned} \tag{3.6}$$

The columns of the tableau below these reduced costs is an identity matrix that can be removed. (3.7) shows this result.

$$\begin{aligned} \mathbf{B}^{-1}\hat{\mathbf{A}} &= [\mathbf{B}^{-1}\mathbf{B} \quad \mathbf{B}^{-1}\mathbf{N}] \\ &= [\mathbf{I} \quad \mathbf{B}^{-1}\mathbf{N}] \end{aligned} \tag{3.7}$$

The dictionary algorithm does not store this unnecessary data, and as a result, requires less computations per Simplex iteration and less overall memory.

The *Tableau Simplex Algorithm* has a storage requirement given by the denominator of (3.8). The proposed algorithm has a storage requirement given by the numerator of (3.8).

$$M_R = \frac{(m+1) \times (n+1)}{(m+1) \times (m+n+1)} \tag{3.8}$$

If it is assumed that the number of constraints and variables in the problem are approximately equal, the memory required to store the problem is reduced to a half of its former value. M_R represents the ratio of memory required in the proposed algorithm to the tableau method and asymptotically approaches 50% as the size of the problem grows.

In [2], read and write copies of the tableau were stored in device memory to facilitate hardware implementation. Processing the data prior to pivoting removes the storage required for a second copy.

3.4 Parallel Implementation of the Dictionary Algorithm

A prototype linear programming solver was developed in OpenCL to benchmark parallel performance. An OpenCL program is divided between a *host* that manages program flow and

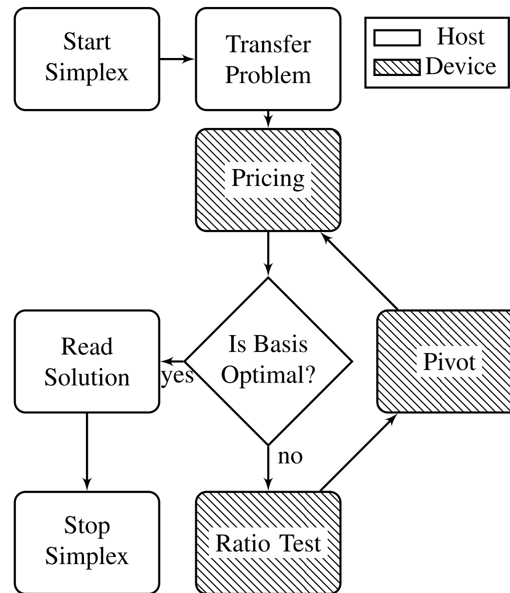


Figure 3.3: Dictionary Algorithm Design Architecture

a *device* that provides parallel resources [16]. A traditional heterogeneous computing system uses a CPU host to drive an acceleration device such as a GPU or FPGA. OpenCL functions, referred to as kernels, operate in parallel on the device. Device memory is divided into high, medium and low latency layers called global, local, and private respectively.

3.4.1 OpenCL Software Design Architecture

The dictionary algorithm implementation applies device kernels in sequence to solve a linear programming problem. The architecture is presented in Figure 3.3, where checking for unboundedness after the ratio test is omitted for clarity. The host initiates the algorithm by queuing the pricing kernel and performs simplex iterations until the detection of the optimal basis.

This design has a main processing loop with minimal transfers between device and host to reduce the memory latency of the application. The important data is sent to the device at the start of the operation. Reducing the number of transfers between the host memory and device memory increases the overall efficiency of the design. Reading the minimum ratio, to check unboundedness, and the entering reduced cost, to detect the optimal basis, are the only device to host memory transfers required.

Table 3.1: Hardware Device Specifications

Device	Power (W)	Memory Bandwidth (GB/s)
Intel Core i7 4930k	130	59.7
Nvidia GeForce GTX-780	250	288.4
Altera Nallatech PCIe-385N	25	23.99

The three interior algorithms were implemented as follows. Entering candidates are priced using a vector reduction on the device. This is a parallel processing technique used to resolve a vector to a single value with commutative operations [17]. For the ratio test, calculating individual ratios are independent operations that are done in parallel The resultant set is reduced by the device to obtain the minimum. The pivot is performed on the device with a block processing implementation of Algorithm 3.3. This requires a two-dimensional kernel in which each realization accesses and updates one element of the dictionary. The two dimensional algorithm is required because the **D** is a two dimensional matrix.

3.5 Benchmarking Results

This section benchmarks the OpenCL dictionary algorithm implementation on hardware accelerators and compares results with serial performance. The design was tested with an Intel Core i7 4930k CPU, a Nvidia GeForce GTX-780 GPU, and a Nallatech PCIe-385N equipped with an Altera Stratix V FPGA. The relevant specifications of these three devices are presented in Table 3.1.

Performance curves were generated by solving multiple sets of linear programming problems with sizes ranging from 256×256 to 8192×8192 variables and constraints. Multiple variations in OpenCL workgroup sizes were tested and the analysis was performed with the best configuration for each device.

3.5.1 Speed Up

The OpenCL implementation grew faster than the sequential code on all three devices with increasing problem size. For applications without stringent energy requirements, Figure 3.4

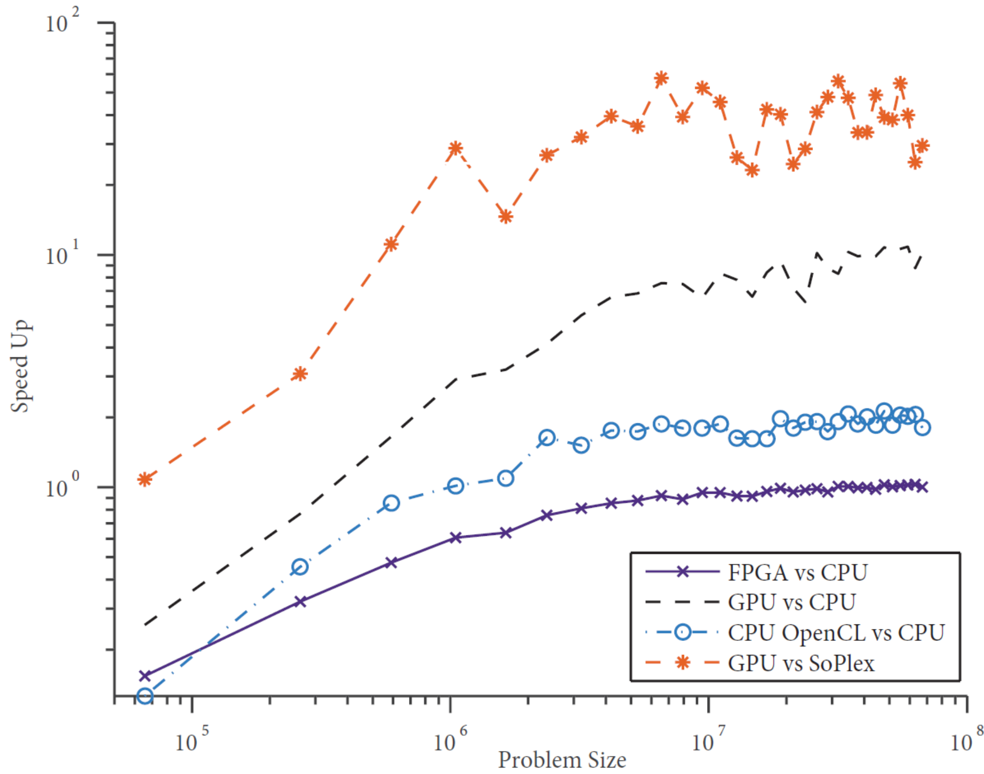


Figure 3.4: Speed up of the OpenCL dictionary algorithm over the sequential C++ algorithm

shows the speed up achieved by the OpenCL solver over the sequential, dense solver. The GPU was the fastest device of the three tested. This advantage became pronounced when the problem size increased beyond 756 by 756 variables and constraints due to higher utilization of the device memory bandwidth and streaming multiprocessors. The speed up became constant when the problem size increased beyond 4096 by 4096 due to saturation of the available memory bandwidth. Though the FPGA was the slowest solution, with close to unity speed up, its low power consumption is an asset that was not considered in this particular test.

The performance of the proposed OpenCL solver was compared to SoPlex [18], an advanced sparse LP solver, over the same range of random, dense problems. SoPlex represents the state of the art in linear programming software with benchmarks that are competitive with other open-source solvers such as Coin-OR [19] and GLPK [20]. The comparison showed that the GPU code was up to 50 times faster than the sparse solver for these dense problems. This shows a niche application for the dense solver where the sparse solver is at an inherent disadvantage.

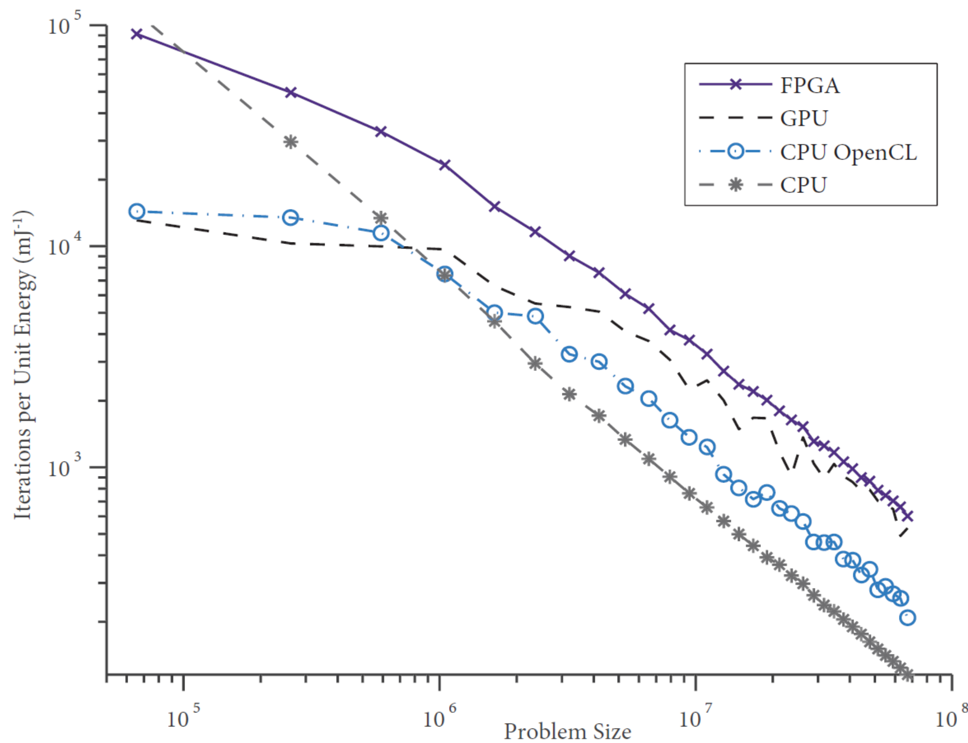


Figure 3.5: Device performance versus power analysis

3.5.2 Performance versus Power

Normalizing the speed of each device with its estimated power consumption resulted in further insight. The average number of iterations possible per unit energy was calculated to gain greater understanding of device performance. This represents the speed at which a solver can process linear programming problems. Higher values represent faster processing for each Joule consumed. Figure 3.5 compares this metric to identify the energy savings accomplished through hardware acceleration.

This result indicates that the FPGA outperforms the CPU and GPU for the entire test spectrum. Although it had the lowest speed, it was the most efficient platform. Large-scale integer linear programming servers with energy consumption concerns would be best suited with the specific FPGA that was tested.

3.5.3 Memory Bandwidth

In an efficient design, the kernel should be always saturated with processing. The memory is not a performance factor alone. A good design saturates both memory and processing time. An efficient OpenCL design for pivot should use the full memory bandwidth available on a device.

The apparent memory bandwidth usage for each device was calculated as an empirical measure of efficiency by multiplying number of accesses to global memory (reading the dictionary value, the problem size, the pivot element, the pivot column and row values, and writing to the dictionary) in the pivot kernel by the size of the problem and the runtime of a Simplex iteration. Since the pivot calculation overpowers the runtime of the other algorithms, they were assumed to have negligible contribution. The bandwidth was normalized for each device with the specifications in Table 3.1.

Figure 3.6 shows that all devices are operating near their nominal memory bandwidth. This means that the design effectively targets the architecture. Reliance on memory bandwidth furthers the importance of the dictionary algorithm's optimized memory. Dividing the tableau size by a factor of two reduces the overall amount of data that must be provided to the kernels from the global memory.

3.5.4 Future Directions

The future direction of this work should focus on three different areas: measuring more accurate power consumption figures for each of the devices, testing more OpenCL compute devices, and comparing performance to a full hardware implementation. The first area is required because the power consumption figures in this work are estimates based on the manufacturer reported specifications. Drawing an accurate comparison based on actual measured power would illuminate further the differences in performance for each of the processors. The second area could focus the testing of the design on more OpenCL enabled devices. Devices such as AMD GPUs and other Intel many-core processors could be used to draw a comparison across a wider range of devices. The third area could focus on implementing the *Dictionary Simplex Algorithm* with Verilog for an FPGA and measuring the performance of a full hardware implementation. This would also enable a comparison between the performance of the OpenCL

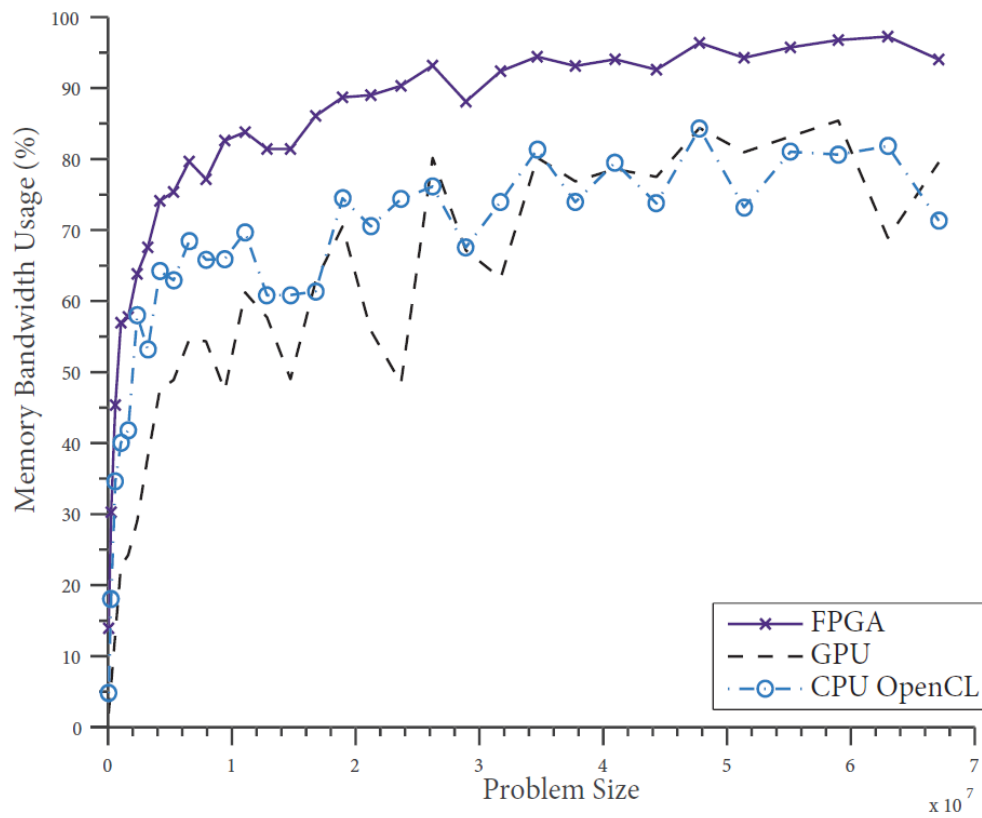


Figure 3.6: The estimated percentage of available memory bandwidth used by each device

SDK for FPGAs and the traditional lower level implementations.

3.6 Conclusions

This chapter presented an OpenCL implementation of the *Simplex Algorithm* and benchmarked performance of parallel hardware computing devices. The proposed hardware implementation of the *Dictionary Simplex Method* reduced the memory requirements of the dense algorithm and solved, to our knowledge, the largest problems with FPGAs to date. OpenCL provided an efficient design methodology that enabled performance tests on a variety of devices.

Test results indicate that the GPU provides the fastest solutions to linear programming problems but is less power efficient than the FPGA. They also show that the memory bandwidth of a device is a critical specification. Analysis of the internal proportions of the dictionary algorithm reveals that an efficient implementation of the pivot algorithm is a critical requirement in a practical solver. The central contribution of this work is the benchmarking to derive an upper-bound for speed-up of the software.Although the tested problems are academic in nature, benchmarking provided further proof that a practical hardware accelerated linear solver would be a great contribution to the field.

Bibliography

[1] J.A.J. Hall. Towards a practical parallelisation of the simplex method. *Computational Managment Science*, pages 139–170, 2003.

[2] A. Hamzic, A. Huseinovic, and N. Nosovic. Implementation and performance analysis of the simplex algorithm adapted to run on commodity OpenCL enabled graphics processors. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies (ICAT)*, pages 1–7, 2011.

[3] S. Bayliss, C.-S. Bouganis, G.A. Constantinides, and W. Luk. An FPGA implementation of the simplex algorithm. In *IEEE International Conference on Field Programmable Technology*, pages 49–56, 2006.

[4] A. Olafsson and S.J. Wright. Linear programming formulations and algorithms for radiotherapy treatment planning. *Optimization Methods and Software*, 21(2):201–231, April 2006.

[5] H.E. Romeijn, R.K. Ahuja, J.F. Dempsey, and A. Kumar. A new linear programming approach to radiation therapy treatment planning problems. *Operations Research*, 54(2):201–216, April 2006.

[6] M.E. Lalami and D. El-Baz. Gpu implementation of the branch and bound method for knapsack problems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 1769–1777, May 2012.

[7] Xiaoqiu Wang and S. Konishi. Optimization formulation of packet scheduling problem in lte uplink. In *2010 IEEE 71st Vehicular Technology Conference (VTC 2010-Spring)*, pages 1–5, May 2010.

[8] J.F. Dopazo and H.M. Merrill. Optimal generator maintenance scheduling using integer programming. *Power Apparatus and Systems, IEEE Transactions on*, 94(5):1537–1545, Sept 1975.

[9] OpenCL - the open standard for parallel programming of heterogenous systems, 2013. http://www.khronos.org/opencl/.

[10] Achieve power-efficient acceleration with OpenCL on Altera FPGAs, 2014. http://www.altera.com/products/software/opencl/opencl-index.html.

[11] D. Chen and D. Singh. Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 5–12, Aug 2012.

[12] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–304, Jan 2013.

[13] D. Solow. *Linear Programming: An Introduction to Finite Improvement Algorithms*. Elsevier Science Publishing Co., Inc., Amsterdam, The Netherlands, 1984.

[14] W. L. Winston. *Operations Research: Applications and Algorithms*. Wadsworth, Inc., Indiana, 1994.

[15] Istvan Maros. *Computational Techniques of the Simplex Method*. Kluwer Academic Publishers, Norwell, MA, 2003.

[16] B.R. Gaster, Howes L., Kaeli R. K., Mistry P., and Schaa D. *Heterogenous Computing with OpenCL*. Thomson Brooks/Cole, The University of California, 2004.

[17] T.H. Cormen, C. Stein, R.L. Rivest, and C.E.Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education 2nd Edition, 2001.

[18] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996. http://www.zib.de/Publications/abstracts/TR-96-09/.

[19] Computational infrastructure for operations research, 2012. http://www.coin-or.org/.

[20] GLPK - GNU linear programming kit, 2012. http://www.gnu.org/software/glpk/.

Chapter 4

Multi-Path Parallelism in the Simplex Algorithm

4.1 Introduction

Though parallel processors are a resource that can improve the performance of software, exploiting them for inherently sequential algorithms, such as linear programming codes based on the *Simplex Algorithm*, is difficult. The complex nature of this problem has spawned numerous efforts within operations research literature to design a practical linear programming code that operates in parallel. The literature shows two distinct approaches for solving this problem; reducing the time per iteration of the algorithm with *internal parallelism* or decreasing the number of iterations required by the algorithm with *external parallelism*. Both topologies achieve a reduced software run-time without compromising the accuracy of final solutions. The first accomplishes this through implementation of sparse linear algebra algorithms in a data-parallel manner and the second spawns multiple threads that race on different paths through the solution space in a task-parallel manner.

This chapter presents a method to parallelize the *Simplex Algorithm* that does not rely on data-level parallelism of the internal matrix calculations, instead relying on the heuristic nature of the path-finding algorithms that trace the direction to the optimum value of the linear programming problem. This form of task-level parallelism uses multiple pricing algorithms to reduce the overall number of iterations required to solve the problem. The parallel algorithm

is shown to improve the performance of an open-source linear programming library with best-case performance with four threads of up to 14.73 times, average-case performance of 1.56 worst-case performance of 0.3 times on a large set of standardized programming problems. This chapter shows that the algorithms has statistically significant performance gains at a 95% confidence level. The algorithm extends the work in [1], another form of externally parallel *Simplex Algorithm*, by removing the need to synchronize at each iteration.

The first section of this chapter presents a literature review on the different methods for parallelizing the *Simplex Algorithm*, classifying them into externally or internally parallel. The second presents an upper bound for performance from internally parallel strategies based on profiling data from SoPlex and Amdahl's Law [2]. The third proposes the *Multi-Path Simplex Algorithm* and explains the detailed operation of the code. The fourth presents a novel test methodology for linear programming software and reveals the performance improvements attained through the parallel algorithm on large, sparse linear programming problems.

4.2 Literature Review

The *Simplex Algorithm* solves optimization models with linear constraints and objective functions known as linear programming problems. This inherently sequential algorithm forms the basis of most linear programming software. The major open source linear programming libraries, such as SoPlex [3], Coin-OR [4] and GLPK [5] solve linear programming problems with a single thread.

This algorithm is sequential because it requires a set of iterative matrix operations such as triangular solve and matrix-vector multiplication on large sparse data which do not traditionally operate well in parallel [6]. Linear programming software exploits sparsity to decrease the consumed memory and overall runtime. Sparse matrix operations reduce the number of computations needed to solve larger linear programming problems. The sparse algorithms significantly increase the complexity of the software and require careful design of data structures and algorithms to ensure practical problems can be solved.

Despite the apparently sequential nature of the algorithm, multiple authors proposed novel parallel topologies. A review of the major literature on parallelizing the *Simplex Algorithm*

[6] concluded that though many promising strategies exist, a practical improvement has yet to be gained from parallel computing. The techniques presented in [6] can be classified into two distinct groups. The first exploits parallelism within each iteration's basic linear algebra subroutines. This will be defined as *internal parallelism*. The second attempts to reduce the number of total iterations taken by the algorithm to find the optimal solution. This will be defined as *external parallelism*. As the number of iterations required to solve a linear programming problem with the *Simplex Algorithm* cannot be predicted, accelerating the time per iteration with faster implementations and reducing the global number of iterations with better entering and leaving variable selections are both effective strategies to improve the software.

Internally parallel algorithms improve the performance of the algorithm by modifying subroutines that take place during each iteration, such as pricing and updating the basis factorization, to operate on parallel cores. This improves the performance of the algorithm by reducing the overall time spent per iteration. This method's success occurs in problems with randomly generated dense matrices that have exploitable structure on FPGAs [7] and GPUs [8]. This approach has yet to yield an implementation within a popular open source linear programming library as these parallel formulations rely on dense linear algebra. The literature does not report results for the larger sparse problems of practical interest that are available in the Netlib or Mittleman test sets [9, 10].

Externally parallel algorithms reduce the global number of iterations taken by the *Simplex Algorithm*. References in the literature to externally parallel algorithms are less frequent than their internal counterparts. The externally parallel algorithm that is most referenced in literature involves following multiple paths around the convex constraint polyhedron of the linear programming problem. Maros and Mitra [1] presented an algorithm of this type that resulted in speed ups over a sequential software. The modest speed ups reported in [1] are an important contribution that shows external parallelism is a viable technique for accelerating *Simplex*.

4.3 Internal Parallelism Performance Bounds

Linear programming codes are large and complex with many modular components applied sequentially. Improving a sequential code-base of this type with parallel resources requires

detailed profiling in order to develop an effective strategy. The algorithms that dominate the overall runtime of the solver should be targeted in order to maximize the effects of parallelism. This section presents the methodology used to identify the bounds on internal parallelism by profiling a popular open source linear programming code, SoPlex. SoPlex is an open-source linear programming code that is part of a large operations research code suite [3]. The code is written in C++ and follows an object-oriented design approach. The software is capable of solving large sparse problems.

4.3.1 Profiling Methodology

SoPlex was subjected to profiling to illuminate the detailed components of the solution process. The process for each problem was profiled with the perf profiling tool on a Linux system that contained an Intel i7-4930k Ivy-Bridge Processor and 32GB of RAM. The solver was compiled with its default settings for maximum performance as provided by the documentation, terminated if it had not found a solution after one thousand seconds, and run with the default command line arguments for algorithm parameters. The time limit is used as the metric of failure for the solver, which is enough time to solve any of the test cases provided. The profiler measured the obtained objective function, number of iterations, and runtime.

Sample linear programming problems compiled from the Netlib, Mittleman and other linear programming and mixed integer programming databases [9, 10, 11] formed the set of profiled problems. The linear relaxations of the mixed integer problems were solved rather than the full problem. For each problem the time spent in each function was measured and the total time spent in that function for solving the entire test set was calculated. Table B.1 contains the full directory of test cases.

4.3.2 Computing Performance Limits

Figure 4.1 presents the cumulative profiling results from SoPlex on the test cases. The largest contributors to the runtime of the solver are the sparse matrix vector multiplication, *setupPUpdate* and triangular solve, *solveUleftNoNZ* functions. The names of the functions are drawn directly from the SoPlex source code. Combined the two algorithms contribute approximately

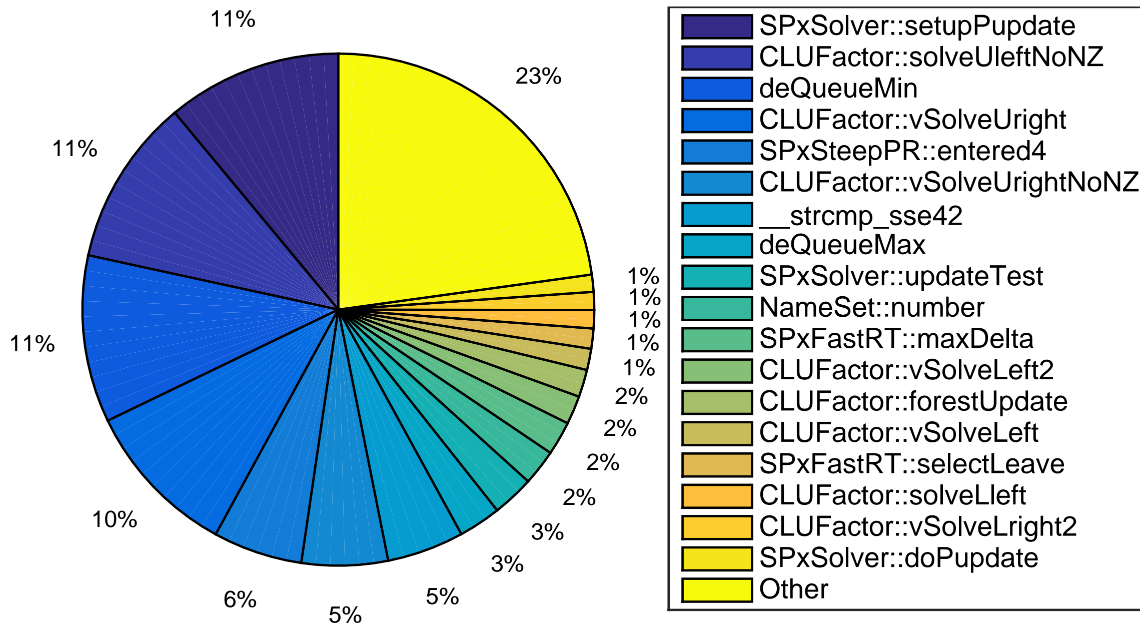


Figure 4.1: Distribution of algorithm runtime in SoPlex for the selected test problems

22% of the runtime of the algorithm. Other algorithms that contribute large time consumption are several other triangular solve algorithms, the pricing calculation *entered4* and dequeuing elements from a heap with *deQueueMin*. Other algorithms, which each individually account for less than one percent of the runtime, contribute a combined twenty-three percent.

The profiling results for SoPlex reveal upper bounds on the performance of internal parallelism through extrapolation based on Amdahl's law. Amdahl's Law is important when assessing the potential performance improvement to software by exploiting parallelism. This law states that if a percentage, P of a system can be conducted in parallel, the maximum performance improvement by conducting the process on infinite processors is given by the inverse of $1 - P$ [2]. Thus a parallel processor can only make a positive impact on the performance of linear programming software if the run-time nature of the code is heavily skewed to a small subset of algorithms that have efficient parallel forms. Though many of the algorithms contained within a sparse solver may effectively target a parallel processor due to high degrees of coarse grained parallelism, it is only possible to impact the software in a meaningful way if these algorithms dominate a high percentage of the runtime. This performance analysis identified that these algorithms do not exist.

Figure 4.2 shows the maximum theoretical speed-up of SoPlex extrapolated from Figure 4.1

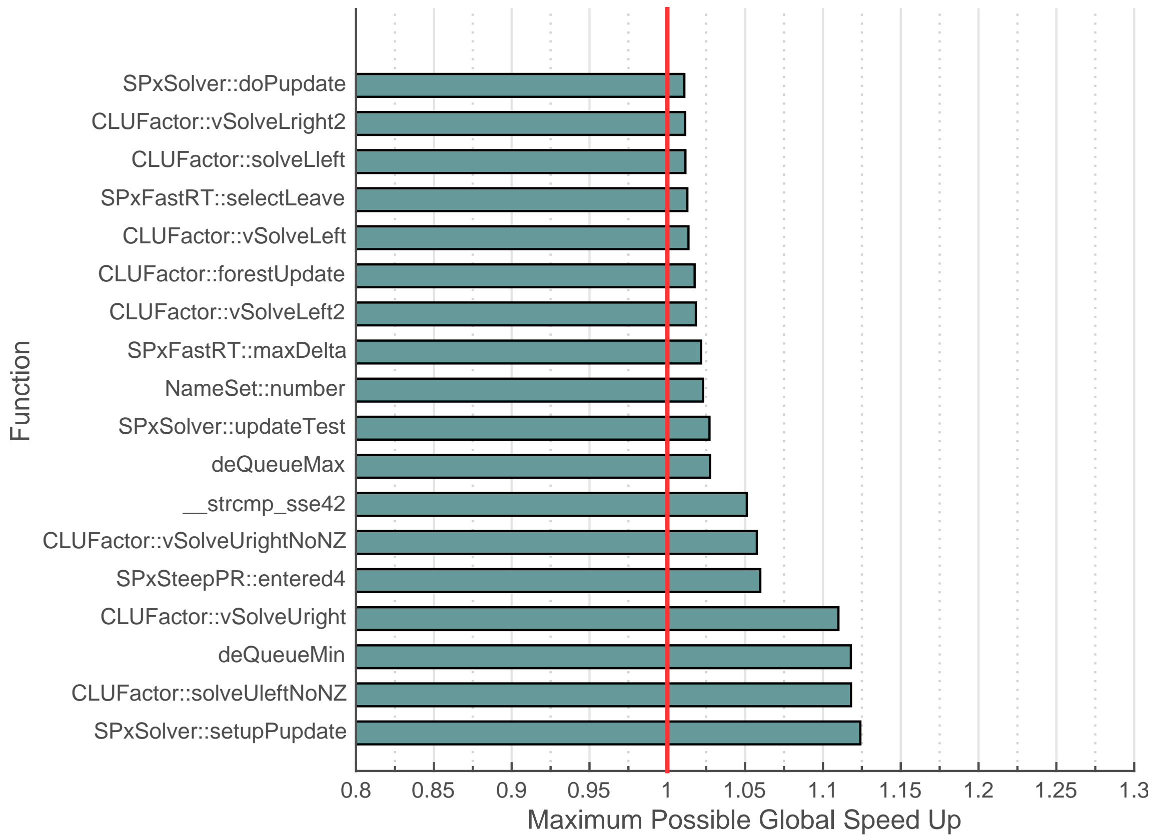


Figure 4.2: Maximum possible speedup predicted by Amdahl's Law

using Amdahl's Law. The maximum possible speed-up to the software is approximately 1.1 times given a parallel sparse matrix vector multiplication kernel that can complete the same calculation as its sequential counterpart in zero time. The second greatest contributor is sparse triangular solve. A parallel form of this algorithm that takes close to negligible amounts of time would also improve the speed of the software by approximately 1.1 times.

Practical implementations of these algorithms will not take zero time and will be subject to overheads. These are from conversions between the special data structures beyond Compressed Sparse Row (CSR) that are required to allow exploitation of parallelism and the transfer of data between a CPU and a massively parallel processor such as a Graphics Processing Unit (GPU). Therefore Amdahl's law overestimates the actual possible impact from parallelization of these individual algorithms. It is possible that parallel versions of these inherently sequential algorithms could be outperformed by the sequential versions.

This analysis shows that the achievable benefits from exploiting internal parallelism in a

linear programming solver based on the *Simplex Algorithm* are minor due to the nature of the software. Performance improvements from internal parallelism cannot affect the solver because of the large number of sequential algorithms.

4.4 The Multi-Path Simplex Algorithm

This section describes the *Multi-Path Simplex Algorithm*, an externally parallel algorithm that uses parallel threads to solve a linear programming problem with different configurations to solve a linear programming problem. The parallel algorithm runs several instances of the algorithm in individual threads with different pricing algorithms. The configuration parameters could also include some combination of different ratio test algorithms or others. When a user considers the number of threads that are available to the solver as well as the available memory, the number of parallel combinations can be tailored to a specific machine.

Algorithm 6 presents the *Multi-Path Simplex Algorithm*. This shows the details for parallel execution including the usage of a mutex and atomic variables. The mutex in this algorithm ensures there are no race conditions and the atomic booleans allow cancellation of a solver from multiple threads. The inputs to the algorithm are a list of configured solver instances that are parametrized based on command line options for the executable.

The first portion of the algorithm allocates resources by instantiating a mutex and forming a dictionary that links each of the solvers to an atomic boolean variable. This variable is initially set to false to indicate that a solution has yet to be found. The solver checks whether or not it has been set to true at each iteration of the algorithm. When the cancel switch is set to true, the solver is terminated. The atomics do not cause delays as there is little competition for access between threads.

Next, the solve method is called for each solver within individual threads and the first to find a solution obtains the mutex. The solution from this solver is stored, the *solved* flag is set to true, and the other solvers are canceled. As each solver exits the main loop with a partial solution to the problem, it obtains the mutex, but infers that a solution has already been obtained. They release the mutex and the parent thread returns the winning solution.

Algorithm 6 Multi-Path Simplex Algorithm

```
1: procedure SIMPLEX(solvers, problem)
2:     solved ← false
3:     mtx ← Mutex()
4:     cancel ← Dictionary()
5:     for each solver ∈ solvers do
6:         cancel[solver] = solver.getAtomicCancelBoolean()
7:     end for
8:     parallel for solver ∈ solvers do
9:         solver.solve(problem)
10:        mtx.lock()
11:        if not solved then
12:            solution ← solver.solution()
13:            solved ← true
14:            for each other ∈ solvers do
15:                if other ≠ solver then
16:                    cancel[other].store(true)
17:                end if
18:            end for
19:        end if
20:        mtx.unlock()
21:    end parfor
22:
23:    return solution
24: end procedure
```

4.4.1 Algorithm Configurations

The performance of the *Simplex Algorithm* depends on the input configuration parameters. The option to run multiple pricing algorithms in parallel creates a software with a greater number of configuration options than the sequential software. This section presents an analysis of the maximum possible benefits that can be achieved with multi-path parallelism based on the configuration parameters that are chosen and presents rationale for selecting a parallel solver over a traditional solver.

In the ideal scenario, each parallel configuration would show performance that surpasses any of the possible sequential configurations. This ideal performance is not achievable due to the nature of the *Simplex Algorithm* as shown by Lemma 4.4.1.

Lemma 4.4.1. *Any implementation of the Simplex Algorithm has at least one pricing algorithm that will result in the global minimum runtime to solve a specific linear programming problem. This minimum runtime can only be found if the problem is solved with every available pricing algorithm.*

Proof. Assume a solver has a set of P unique pricing algorithms available for solving a linear programming problem. Each $p \in P$ corresponds to a specific path that is followed along the edges of the constraint polyhedron from the initial basis to the final basis. Each path is likely to have a different length, N_p, and visit a different set of variables. The total time required to solve the linear programming problem for a pricer p is given by (4.1) where t_{pi} is the time taken for iteration, i, and t_p is the total time.

$$t_p = \sum_{i=1}^{N_p} t_{pi} \tag{4.1}$$

For any pricer, each t_{pi} is likely to vary as the sparsity of the data structures changes throughout iterations. However, the overall behavior can be predicted with complexity analysis. The total number of steps N_p, however, cannot be predicted and is an open problem [12]. The set of run times for each pricer is thus unknown prior to actually solving the problem. Within this set there will be one minimum, which could be repeated in the event of a tie between two of the algorithms. Therefore there is one global minimum runtime for a given linear

programming problem and this value can only be quantified by solving the problem with each pricer. □

Given that user is likely to solve their model once, and that the optimal pricing algorithm is impossible to predict, the parallel algorithm increases the probability that a solution is found. For example, SoPlex contains five pricing algorithms. If multiple of these are run simultaneously, the probability that the optimum algorithm is chosen scales linearly with the number of threads. Each additional thread adds twenty percent to the probability that the optimum configuration is chosen in the first attempt to solve the problem. The algorithm reduces the probability that a solution is never found due to a high level of degeneracy, where the algorithm cycles between two variables for an indeterminate amount of time.

4.5 Profiling The Multi-Path Algorithm

Testing the *Multi-Path Simplex Algorithm* reveals a performance improvement attainable from parallel cores for an average user of the software. This section presents the performance results for the algorithm, discussing the effect of problem size on the behavior of the parallel solver.

Measurements of the sequential and parallel algorithm performance were derived from a fair test methodology to ensure accuracy. The nature of the algorithm presented in Lemma 4.4.1 makes this comparison a challenge. To summarize, of the set of pricing algorithms available in a linear programming solver, there is always a single algorithm that will surpass all others in performance. One paths chosen over the constraint polyhedron from the initial basis to the optimal basis will always yield a shorter run-time. As this configuration, with current techniques, cannot be predicted without manual analysis of the sparsity pattern and advanced expertise from a user of the software, a model of the user must be used to select runtime parameters. This study applied random parameter selection to represent a novice user. Each trial selected a random pricing algorithm from those available in SoPlex. The Dantzig pricer was excluded due to its inability to solve large problems.

The results from solving each test case 100 times with both solvers with randomly selected pricing algorithms to generate an average solution time for each problem are presented in Table B.1. The results were computed on a system containing an Intel i7 4930k processor and

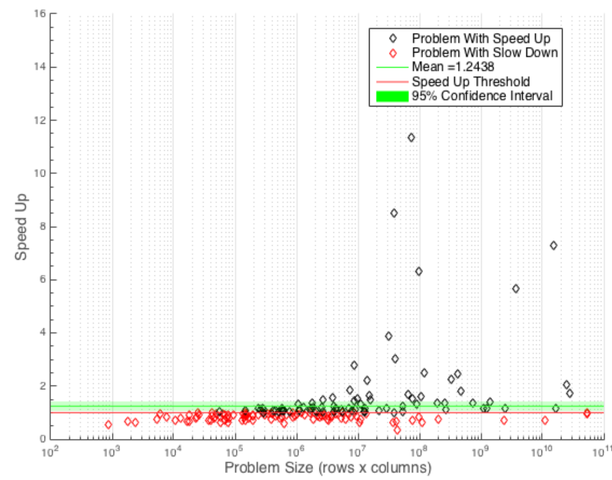
32 GB of RAM. The algorithm was implemented in C++ and the thread construct of the boost library [13] was used for multi-threading the solver.

4.6 Discussion

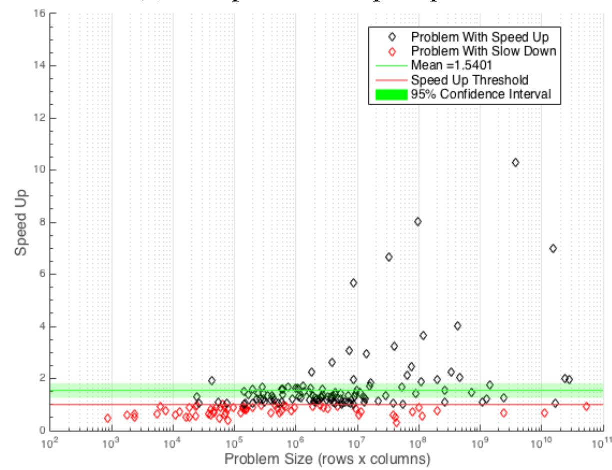
Profiling the parallel algorithm revealed trends in performance. The speed up, given as the ratio of the time taken by the sequential algorithm to solve the linear programming problem to the time taken by the parallel algorithm, signifies performance improvement if it is greater than 1. To prove that the parallel algorithm has greater performance, the statistical significance of the mean was tested. The significance test was based on a null hypothesis claiming the speed up obtained is less than or equal to 1. This section presents an analysis that shows that this null hypothesis can be rejected based on the performance data in Table B.1.

Figure 4.3 summarizes the profiling results by comparing the speed up with multiple threads to the product of the number of rows and columns in the problem. The average speed up with two, three and four threads were 1.24, 1.54, and 1.56 respectively. The green regions in Figure 4.3 show the 95% confidence intervals for these averages and are always above the red line at a speedup of one. The gap between the lower end of the 95% confidence interval and this line shows that the acceleration achieved by running multiple cores is statistically significant. This allows rejection of the null hypothesis that the parallel algorithm has no effect on the speed of the solver. The alternate hypothesis, that the parallel algorithm improves the solver, is acceptable.

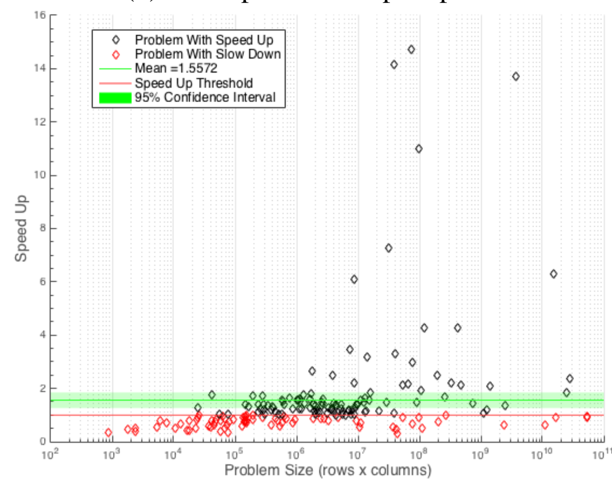
This speed up measurement includes the smaller problems that do not see significant gains. For small problems, the solve time for any pricing algorithm is essentially trivial and the overheads from managing multiple threads dominates the runtime. If problems with a size of less than a million elements are excluded, the average speed up for two, three and four threads increase to 1.51, 1.97 and 2.04 respectively. This result is shown in Figure 4.4. The statistical evidence for speed up is stronger when the smaller problems are eliminated from the analysis, and the gap between the speed up threshold and lower end of the confidence intervals increased. This is a significant improvement in the speed of the solver.



(a) Two parallel Simplex paths

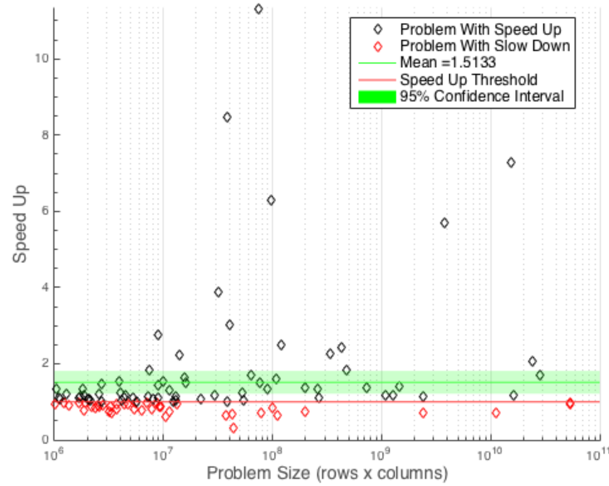


(b) Three parallel Simplex paths

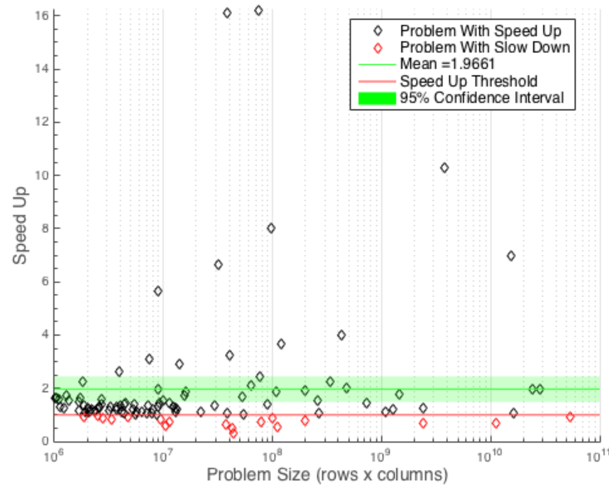


(c) Four parallel Simplex paths

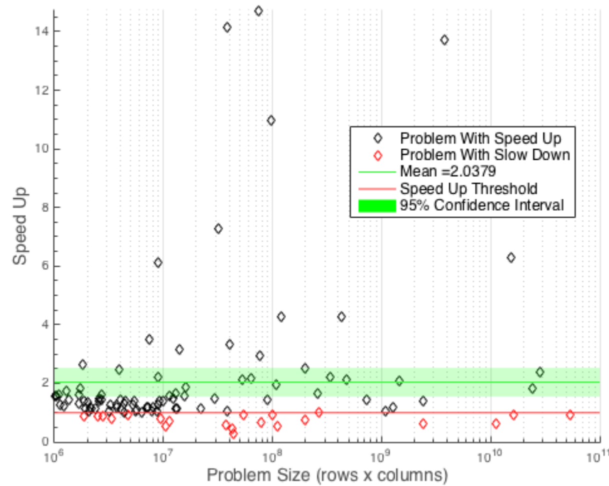
Figure 4.3: Ratio of the time taken by SoPlex to the time taken by the *Multi-Path Simplex Algorithm* for the full data set



(a) Two parallel Simplex paths



(b) Three parallel Simplex paths



(c) Four parallel Simplex paths

Figure 4.4: Ratio of the time taken by SoPlex to the time taken by the *Multi-Path Simplex Algorithm* for problems with over 10^6 elements

4.6.1 Future Directions

The *Multi-Path Simplex Algorithm* enables several new topologies for linear programming solvers. Though this study focuses on pricing algorithms, there are other configuration parameters that can alter the runtime behavior of a solver. Parallel versions of these could be tested to identify the most significant statistical result. The other area in which this research can be furthered is by experimenting with the solver on massively parallel processors. As the number of possible configurations of a solver is large, more parallel processors would enable a larger variety of simultaneous configurations. Large scale simulations could be run to measure how the algorithm scales across larger numbers of cores.

4.7 Conclusion

The *Multi-Path Simplex Algorithm* is a variation on the classical *Simplex Algorithm* that runs different configuration parameters in parallel. It improves the performance of the open source library SoPlex by an average of 2 times for very large, sparse linear programming problems. The overall speed ups are, to our knowledge, the largest reported speed up for an externally parallel algorithm.

Bibliography

[1] I. Maros and G. Mitra. Investigating the sparse simplex algorithm on a distributed memory multiprocessor. *Parallel Computing*, 26(1):151 – 170, 2000.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996. http://www.zib.de/Publications/abstracts/TR-96-09/.

[4] Computational infrastructure for operations research, 2012. http://www.coin-or.org/.

[5] GLPK - GNU linear programming kit, 2012. http://www.gnu.org/software/glpk/.

[6] J.A.J. Hall. Towards a practical parallelisation of the simplex method. *Computational Managment Science*, pages 139–170, 2003.

[7] S. Bayliss, C.-S. Bouganis, G.A. Constantinides, and W. Luk. An FPGA implementation of the simplex algorithm. In *IEEE International Conference on Field Programmable Technology*, pages 49–56, 2006.

[8] A. Hamzic, A. Huseinovic, and N. Nosovic. Implementation and performance analysis of the simplex algorithm adapted to run on commodity OpenCL enabled graphics processors. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies (ICAT)*, pages 1–7, 2011.

[9] Netlib linear programming sample problems, 2015. http://www.netlib.org/lp/.

[10] Computational optimization research at lehigh - mip instances, 2012. http://coral.ise.lehigh.edu/data-sets/mixed-integer-instances/.

[11] Hans Mittleman. Benchmarks of simplex LP solvers, 2015. http://plato.asu.edu/ftp/lpsimp.html.

[12] Istvan Maros. *Computational Techniques of the Simplex Method*. Kluwer Academic Publishers, Norwell, MA, 2003.

[13] The boost C++ libraries, 2015. http://www.boost.org/.

Chapter 5

Conclusion

This thesis examined two optimization problems, architectural layout generation and linear programming, and presented high performance implementations of several proposed algorithms to enhance efficiency and design. The first study presented a detailed general form of the layout generation problem, deriving properties of the optimization problem's solution space and proposing *Evolutionary Treemap*. The study showed an implementation of the proposed algorithms in a highly scalable cloud platform that can be extended to include new works in the field. It was followed by an implementation of the *Simplex algorithm*, a classical optimization algorithm for solving linear programming problems, for GPUs and FPGAs with OpenCL that improved upon the memory structure and energy efficiency of other solutions in literature. The final study of the thesis proposed and implemented the *Multi-Path Simplex Algorithm*, realizing improvements to the performance and reliability of an open source linear programming library measured by several novel profiling methodologies. This final chapter will summarize the main conclusions from each of the studies and present commentary on the possible future directions that can be examined in each field.

5.1 Intelligent Architectural Design

The architectural design chapter of this thesis presented the general form of the optimization problem and solved it with the *Evolutionary Treemap* algorithm. The chapter also presented a detailed summary of the design of a scalable cloud platform for architectural layout generation.

The results of the research on layout generations opens several research topics in the field. The major topics are non-rectangular boundaries, fixed rooms and multi-level buildings. Though the proposed heuristics enable the solution of some layouts with these requirements, a large portion of the solution space is still unattainable. The primary focus of future research in this field is finding an algorithm that can capably handle these situations without resorting to heuristic methods. If these problems are solved, a new frontier of possible solutions and designs can be created autonomously. Benefits in both the fields of architectural and video game design are possible with these types of solutions.

The second direction approachable in this field is finding new optimizer and generator functions for producing functional layouts. A literature review of greater detail of the methodologies in computer graphics could bring forth new ideas for layout generation. Algorithms such as delaunay triangulation, for example, could be operated as generator functions if used in creative ways. Some of the algorithms from computer graphics may require high performance compute devices such as GPUs to prove effective as it is critical that the runtime of a generator function is small so that it can be run many times by an optimization meta-heuristic. Other algorithms, such as simulated annealing and particle swarm optimization, could also be explored as the drivers for *Evolutionary Treemap* to see the convergence properties that they provide.

5.2 Parallel Linear Programming

The two linear programming chapters of this thesis followed different approaches to derive speed ups for the *Simplex Algorithm* with parallel computing. The first chapter followed a data-level parallel approach for the dense variant of the algorithm, implemented it with OpenCL for high performance processors and then benchmarked the algorithm on random problems. The work in this chapter targeted a relatively rare form of linear programming problem with dense matrices. This chapter identified a challenge inherent to parallel linear programming design; the majority of problems are sparse and cannot be effectively solved with a GPU or FPGA. The second linear programming chapter performed benchmarking on open-source sparse linear programming software based on the simplex algorithm to identify the limits of data-level

parallelism in the algorithm. This chapter then proposed the *Multi-Path Simplex Algorithm* to circumvent the limit of data-level computing as imposed by Amdahl's Law on the system with a task-parallel algorithm. The algorithm performed well on practical problems, with an average speed-up of over two times on large problems. Together the chapters show methodologies for parallelizing the algorithm in two ways with different benefits and drawbacks based on the application. Higher speedups are attainable if the problem is dense and the software is created to exploit that on GPUs and moderate speedups are possible if the problem is more suitably modeled with sparse matrices.

The future directions in linear programming with parallel computing concern testing the *Multi-Path Simplex Algorithm* on a wider variety of test cases and implementing it within other open-source softwares. Adding the multi-path algorithm to Coin-OR [1], GLPK [2] and Lp-Solve [3] constitutes a future research project that could further illuminate the performance benefits of the algorithm. The algorithm should also be tested on a processor that can support a larger number of threads to measure the scalability of the algorithm across multiple cores as well as the parameters of the algorithm that should be used to give optimal performance.

5.3 Concluding Remarks

This thesis approached two optimization problems in software, generating floor plans and implementing the *Simplex Algorithm* in parallel. The first section presented the general form of the layout generation optimization problem and implemented several proposed algorithms as a web service. The software for this project allows architects and video game designers to generate optimal layouts from a spreadsheet based specification. The second section presented two methodologies for accelerating the *Simplex Algorithm* for different input data topologies. An OpenCL implementation of the algorithm was proposed for solving problems with dense matrices and showed a speedup of over ten times a sequential code with a GPU. An implementation of the *Multi-Path Simplex Algorithm* was proposed for solving problems with large sparse matrices and yielded a speedup of over twice that of an open-source software.

Bibliography

[1] Computational infrastructure for operations research, 2012. http://www.coin-or.org/.

[2] GLPK - GNU linear programming kit, 2012. http://www.gnu.org/software/glpk/.

[3] Lpsolve linear programming solver, 2015. http://lpsolve.sourceforge.net/.

Appendix A

SoPlex Performance Data

Table A.1: SoPlex Runtime for Selected Linear Programming Test Cases

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
neos631710.mps	1.88E+02	26422	265.020	266.605
ex9.mps	8.10E+01	31639	106.920	107.496
neos1429212.mps	3.00E+01	27553	87.290	89.616
mspp16.mps	3.41E+02	474	9.670	67.154
mzzv11.mps	-2.29E+04	49432	26.780	26.918
netdiversion.mps	2.31E+02	28711	24.540	25.925
neos1367061.mps	3.13E+07	19364	22.810	23.389
neos-934278.mps	2.60E+02	23671	19.200	19.349
map18.mps	-9.33E+02	21570	9.450	13.187
core2536-691.mps	6.88E+02	47858	12.760	12.920
ns1758913.mps	-1.50E+03	2163	5.490	12.248
map20.mps	-9.99E+02	20402	8.140	11.883
dfl001.mps	1.13E+07	25519	10.320	10.389
satellites1-25.mps	-2.00E+01	16253	10.080	10.155
n3seq24.mps	5.20E+04	3813	7.240	9.708
vpphard.mps	0.00E+00	11241	8.050	8.818
rail507.mps	1.72E+02	13241	7.810	8.126
pilot87.mps	3.02E+02	17925	7.610	7.685

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
opm2-z7-s2.mps	-1.29E+04	15896	6.710	6.839
app1-2.mps	-2.65E+02	14641	6.150	6.387
bab5.mps	-1.25E+05	25618	6.050	6.164
unitcal_7.mps	1.94E+07	24544	5.170	5.396
greenbea.mps	-7.26E+07	19038	3.460	3.517
neos-476283.mps	4.06E+02	5858	1.570	3.472
neos-1601936.mps	1.00E+00	14407	3.290	3.337
truss.mps	4.59E+05	22542	3.080	3.108
biella1.mps	3.06E+06	16717	3.020	3.089
pilot.mps	-5.57E+02	10951	2.730	2.771
net12.mps	1.72E+01	14207	2.580	2.696
msc98-ip.mps	1.95E+07	14662	2.560	2.659
fit2p.mps	6.85E+04	15947	2.480	2.538
rmatr100-p5.mps	7.62E+02	10735	2.260	2.304
air04.mps	5.55E+04	11892	2.030	2.063
d2q06c.mps	1.23E+05	12305	2.000	2.035
stocfor3.mps	-4.00E+04	13971	1.640	1.748
greenbeb.mps	-4.30E+06	11491	1.510	1.537

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
ns1208400.mps	0.00E+00	11064	1.480	1.527
maros-r7.mps	1.50E+06	5515	1.260	1.335
fit2d.mps	-6.85E+04	10729	0.930	0.988
pilot-ja.mps	-6.11E+03	11607	0.890	0.909
rmatr100-p10.mps	3.61E+02	5500	0.780	0.819
neos-849702.mps	0.00E+00	7153	0.760	0.779
neos10.mps	-1.25E+03	838	0.330	0.605
triptim1.mps	1.00E+100	1334	0.220	0.589
n3div36.mps	1.14E+05	317	0.240	0.585
neos1597104.mps	-3.00E+01	194	0.160	0.582
tanglegram1.mps	0.00E+00	426	0.120	0.572
80bau3b.mps	9.87E+05	7968	0.530	0.566
neos13.mps	-1.26E+02	2060	0.370	0.545
pilotnov.mps	-4.50E+03	7113	0.500	0.526
sp98ic.mps	4.44E+08	1553	0.370	0.516
25fv47.mps	5.50E+03	7393	0.490	0.504
acc-tight5.mps	0.00E+00	3527	0.430	0.450
pilot-we.mps	-2.72E+06	5978	0.400	0.418

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
degen3.mps	-9.87E+02	6047	0.390	0.412
rocII-4-11.mps	-1.19E+01	450	0.100	0.321
perold.mps	-9.38E+03	4933	0.290	0.305
neos-1337307.mps	-2.03E+05	4892	0.270	0.296
nesm.mps	1.41E+07	7277	0.260	0.279
ns1830653.mps	6.15E+03	1442	0.200	0.260
pw-myciel4.mps	0.00E+00	2239	0.220	0.254
neos-1109824.mps	2.78E+02	139	0.140	0.244
iis-pima-cov.mps	2.66E+01	1276	0.190	0.237
ash608gpia-3col.mps	2.00E+00	2984	0.120	0.233
mine-90-10.mps	-8.87E+08	2242	0.200	0.227
sp98ir.mps	2.17E+08	2495	0.200	0.226
csched010.mps	3.32E+02	5866	0.200	0.210
bnl2.mps	1.81E+03	2921	0.170	0.196
iis-bupa-cov.mps	2.65E+01	1179	0.160	0.193
neos-1396125.mps	3.89E+02	2482	0.160	0.169
grow22.mps	-1.61E+08	4558	0.160	0.165
d6cube.mps	3.15E+02	1179	0.140	0.162

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
30n20b8.mps	1.57E+00	272	0.040	0.159
woodw.mps	1.30E+00	1760	0.130	0.158
neos-916792.mps	2.62E+01	1165	0.100	0.157
mine-166-5.mps	-8.22E+08	1587	0.120	0.155
rococoC10-001000.mps	7.52E+03	2543	0.130	0.141
lectsched-4-obj.mps	0.00E+00	196	0.050	0.135
eilB101.mps	1.08E+03	1459	0.110	0.123
reblock67.mps	-3.93E+07	1758	0.100	0.117
roll3000.mps	1.11E+04	1880	0.090	0.110
rmine6.mps	-4.62E+02	1522	0.080	0.105
fit1p.mps	9.15E+03	3110	0.090	0.104
zib54-UUE.mps	3.88E+06	1956	0.050	0.102
danoint.mps	6.26E+01	1897	0.080	0.096
pilot4.mps	-2.58E+03	1675	0.080	0.087
scsd8.mps	9.05E+02	2063	0.070	0.083
iis-100-0-cov.mps	1.67E+01	516	0.050	0.074
eil33-2.mps	8.11E+02	387	0.050	0.072
mcsched.mps	1.94E+05	1994	0.060	0.071

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
bnl1.mps	1.98E+03	1464	0.060	0.071
n4-3.mps	4.08E+03	1341	0.060	0.069
aflow40b.mps	1.01E+03	1607	0.050	0.068
czprob.mps	2.19E+06	1569	0.050	0.065
grow15.mps	-1.07E+08	1837	0.050	0.064
scfxm2.mps	3.67E+04	1164	0.050	0.062
stocfor2.mps	-3.90E+04	1869	0.040	0.058
wood1p.mps	1.44E+00	145	0.030	0.057
scfxm3.mps	5.49E+04	1706	0.050	0.055
neos18.mps	7.00E+00	432	0.010	0.055
cov1075.mps	1.71E+01	455	0.040	0.052
degen2.mps	-1.44E+03	1325	0.040	0.052
tanglegram2.mps	0.00E+00	185	0.010	0.050
maros.mps	-5.81E+04	1328	0.040	0.049
cycle.mps	-5.23E+00	994	0.030	0.049
beasleyC3.mps	4.04E+01	1140	0.040	0.047
bnatt350.mps	0.00E+00	657	0.020	0.047
ns1688347.mps	2.00E+00	230	0.020	0.047

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
neos-686190.mps	5.13E+03	322	0.020	0.043
pg5_34.mps	-1.66E+04	3400	0.030	0.039
binkar10_1.mps	6.64E+03	1300	0.020	0.038
stair.mps	-2.51E+02	655	0.030	0.036
ship12l.mps	1.47E+06	1085	0.020	0.035
ship08l.mps	1.91E+06	809	0.020	0.030
fffff800.mps	5.56E+05	928	0.020	0.030
ganges.mps	-1.10E+05	1304	0.020	0.029
sctap3.mps	1.42E+03	657	0.020	0.029
sctap2.mps	1.72E+03	505	0.020	0.028
qiu.mps	-9.32E+02	1288	0.020	0.028
grow7.mps	-4.78E+07	1012	0.020	0.027
fit1d.mps	-9.15E+03	985	0.020	0.027
scagr25.mps	-1.48E+07	784	0.020	0.025
modszk1.mps	3.21E+02	651	0.010	0.023
macrophage.mps	0.00E+00	678	0.000	0.023
sierra.mps	1.54E+07	638	0.010	0.022
ship12s.mps	1.49E+06	649	0.010	0.021

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
bandm.mps	-1.59E+02	543	0.010	0.021
etamacro.mps	-7.56E+02	717	0.010	0.020
boeing1.mps	-3.35E+02	557	0.010	0.020
newdano.mps	1.17E+01	469	0.010	0.019
ship08s.mps	1.92E+06	514	0.010	0.018
shell.mps	1.21E+09	595	0.010	0.018
gfrd-pnc.mps	6.90E+06	664	0.010	0.018
scsd6.mps	5.05E+01	422	0.010	0.017
scfxm1.mps	1.84E+04	462	0.000	0.016
brandy.mps	1.52E+03	486	0.010	0.016
bienst2.mps	1.17E+01	469	0.010	0.016
gmu-35-40.mps	-2.41E+06	341	0.010	0.015
finnis.mps	1.73E+05	505	0.010	0.015
scrs8.mps	9.04E+02	608	0.010	0.015
forplan.mps	-6.64E+02	280	0.000	0.014
e226.mps	-1.88E+01	377	0.010	0.014
pigeon-10.mps	-1.00E+04	292	0.010	0.013
ship04l.mps	1.79E+06	473	0.000	0.012

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
dfn-gwin-UUM.mps	2.75E+04	375	0.000	0.012
sctap1.mps	1.41E+03	262	0.000	0.012
standmps.mps	1.41E+03	189	0.000	0.012
ran16x16.mps	3.12E+03	379	0.000	0.011
tuff.mps	2.92E+00	220	0.000	0.011
agg3.mps	1.03E+07	235	0.000	0.011
seba.mps	1.57E+04	2	0.000	0.011
israel.mps	-8.97E+05	184	0.000	0.010
capri.mps	2.69E+03	327	0.000	0.010
agg2.mps	-2.02E+07	204	0.000	0.010
ship04s.mps	1.80E+06	383	0.000	0.010
lotfi.mps	-2.53E+01	227	0.000	0.009
agg.mps	-3.60E+07	95	0.000	0.009
sc205.mps	-5.22E+01	209	0.000	0.009
standata.mps	1.26E+03	50	0.000	0.009
scsd1.mps	8.67E+00	95	0.000	0.009
scorpion.mps	1.88E+03	245	0.000	0.008
beaconfd.mps	3.36E+04	88	0.000	0.008

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
bore3d.mps	1.37E+03	100	0.000	0.008
m100n500k4r1.mps	-2.50E+01	174	0.000	0.008
share1b.mps	-7.66E+04	217	0.000	0.008
share2b.mps	-4.16E+02	115	0.000	0.008
timtab1.mps	2.87E+04	13	0.000	0.008
glass4.mps	8.00E+08	74	0.000	0.007
adlittle.mps	2.25E+05	87	0.000	0.007
boeing2.mps	-3.15E+02	154	0.000	0.007
afiro.mps	-4.65E+02	16	0.000	0.007
ns1766074.mps	5.83E+03	27	0.000	0.007
enlight13.mps	0.00E+00	0	0.000	0.006
recipe.mps	-2.67E+02	40	0.000	0.006
mik-250-1-100-1.mps	-7.98E+04	100	0.000	0.006
noswot.mps	-4.30E+01	104	0.000	0.006
blend.mps	-3.08E+01	97	0.000	0.006
enlight14.mps	0.00E+00	0	0.000	0.006
kb2.mps	-1.75E+03	56	0.000	0.006
stocfor1.mps	-4.11E+04	111	0.000	0.005

Table A.1: SoPlex Runtime for Selected Test Cases: Continued from Previous Page

Problem Name	Objective	Iterations	Reported Time (s)	Measured Time (s)
vtp-base.mps	1.30E+05	78	0.000	0.005
sc50b.mps	-7.00E+01	50	0.000	0.005
scagr7.mps	-2.33E+06	178	0.000	0.005
sc105.mps	-5.22E+01	94	0.000	0.005
sc50a.mps	-6.46E+01	46	0.000	0.004

Table A.2: Average Runtime of SoPlex Functions over Selected Linear Programming Test Cases from Perf Profiling Reports

Function	Average Time (ms)
soplex::SPxSolver::setupPupdate	484.77
soplex::CLUFactor::solveUleftNoNZ	463.72
soplex::deQueueMin	463.36
soplex::CLUFactor::vSolveUright	435.14
soplex::SPxSteepPR::entered4	247.67
soplex::CLUFactor::vSolveUrightNoNZ	238.91
__strcmp_sse42	213.43
soplex::deQueueMax	117.52
soplex::SPxSolver::updateTest	115.91
soplex::NameSet::number	98.92
soplex::SPxFastRT::maxDelta	93.60
soplex::CLUFactor::vSolveLeft2	79.11
soplex::CLUFactor::forestUpdate	75.79
soplex::CLUFactor::vSolveLeft	58.93
soplex::SPxFastRT::selectLeave	55.89
soplex::CLUFactor::solveLleft	50.54
soplex::CLUFactor::vSolveLright2	49.93
soplex::SPxSolver::doPupdate	47.56

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SPxBasis::dualStatus	42.50
soplex::SPxSolver::clearUpdateVecs	41.61
soplex::CLUFactor::updateRow	40.78
soplex::SPxFastRT::minDelta	39.02
soplex::CLUFactor::solveUleft	34.70
soplex::SPxSteepPR::selectEnterHyperCoDim	34.43
soplex::CLUFactor::solveLleftForest	33.93
__memcpy_ssse3_back	32.78
soplex::SPxSolver::updateFtest	31.47
strtok	29.63
__memset_sse2	29.61
soplex::CLUFactor::vSolveLright	27.53
soplex::NameSet::add	23.82
soplex::SPxSteepPR::selectEnter	23.04
soplex::SPxSolver::enter	22.80
soplex::CLUFactor::initFactorMatrix	22.28
soplex::MPSInput::readLine	21.93
soplex::SPxSolver::doRemoveCol	20.52

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SPxSteepPR::selectEnterDenseDim	20.06
soplex::SPxSteepPR::selectLeave	19.17
soplex::IdxSet::operator=	18.78
soplex::SPxSolver::doRemoveRow	18.03
soplex::NameSetNameHashFunction	16.86
soplex::CLUFactor::colSingletons	16.73
soplex::SPxMainSM::duplicateRows	15.64
soplex::SPxMainSM::duplicateCols	14.61
soplex::SPxDevexPR::selectEnterHyperCoDim	13.01
soplex::SPxDevexPR::entered4	12.46
soplex::SPxDevexPR::selectEnterDenseDim	10.84
soplex::SPxSolver::perturbMaxEnter	10.70
soplex::SPxLPBase<double >::readMPS	10.27
soplex::SPxSteepPR::left4	10.23
soplex::SPxMainSM::simplifyRows	10.20
soplex::SPxLPBase<double >::added2Set	9.55
soplex::SPxSolver::updateCoTest	9.14
soplex::CLUFactor::remaxRow	9.14

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
__memmove_ssse3_back	8.97
soplex::SPxSteepPR::selectLeaveHyper	8.67
__GI_____strtod_l_internal	7.47
soplex::SPxSolver::leave	7.35
soplex::CLUFactor::rowSingletons	7.19
soplex::CLUFactor::vSolveRight4update2	6.60
soplex::CLUFactor::eliminatePivot	5.51
soplex::CLUFactor::selectPivots	4.95
soplex::CLUFactor::setupColVals	4.92
soplex::CLUFactor::packRows	4.31
soplex::SPxDevexPR::selectLeaveHyper	4.08
soplex::SPxDevexPR::left4	3.96
clear_page_c_e	3.83
soplex::ClassArray<soplex::Nonzero<double >>::reMax	3.45
soplex::SPxDevexPR::buildBestPriceVectorEnterCoDim	3.25
__memcpy_sse2	3.13
soplex::CLUFactor::vSolveRight4update	3.07
soplex::SPxScaler::applyScaling	3.01

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
std::istream::getline	2.99
_int_malloc	2.89
soplex::SPxSolver::doRemoveRows	2.54
soplex::SPxMainSM::simplifyCols	2.36
memchr	2.35
__frexp	2.29
soplex::SPxSolver::doRemoveCols	2.16
_int_free	2.05
soplex::SPxSolver::nonbasicValue	2.00
soplex::CLUFactor::setPivot	1.93
soplex::SPxDevexPR::selectLeave	1.92
soplex::SPxSolver::computeTest	1.79
__ldexp	1.74
soplex::SPxMainSM::simplifyDual	1.71
soplex::SPxSolver::perturbMinEnter	1.65
soplex::SPxShellsort<soplex::Nonzero<double >, soplex::SPxMainSM::ElementCompare>	1.64
soplex::SPxSolver::computePvec	1.58
soplex::CLUFactor::setupRowVals	1.57

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SVSetBase<double >::operator=	1.44
round_and_return	1.39
soplex::SVSetBase<double >::SVSetBase	1.35
soplex::SPxScaler::computeScalingVecs	1.32
page_fault	1.28
copy_user_enhanced_fast_string	1.26
soplex::CLUFactor::remaxCol	1.25
free	1.17
str_to_mpn.isra.0	1.09
soplex::NameSet::reMax	1.07
soplex::CLUFactor::solveUright	1.07
soplex::SPxQuicksortPart<soplex::SPxPricer::IdxElement, soplex::SPxPricer::IdxCompare>	1.05
strcmp@plt	1.02
soplex::SPxSolver::qualConstraintViolation	1.00
soplex::SPxColId::SPxColId	0.93
soplex::NameSet::memPack	0.90
soplex::SPxMainSM::removeEmpty	0.89
soplex::SVSetBase<double >::ensurePSVec	0.89

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SPxSolver::computeFrhsXtra	0.88
strtok@plt	0.88
__strlen_sse2	0.85
__strlen_sse2_pminub	0.84
realloc	0.81
soplex::NameSet::memRemax	0.81
soplex::CLUFactor::factor	0.81
soplex::LPColSetBase<double >::add	0.78
malloc	0.78
apic_timer_interrupt	0.77
strtod	0.76
soplex::CLUFactor::initFactorRings	0.73
soplex::SPxQuicksort<soplex::Nonzero<double >, soplex::SPxMainSM::ElementCompare>	0.69
soplex::SPxSolver::computeCoTest	0.69
__mpn_construct_double	0.69
soplex::SPxMainSM::handleExtremes	0.68
soplex::DSVectorBase<double >::add	0.65
__scalbn	0.64

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
thread_group_cputime	0.62
soplex::CLUFactor::forestReMaxCol	0.59
soplex::Array<soplex::DSVectorBase<double >>:: Array	0.59
soplex::SPxSolver::computeFtest	0.56
soplex::SPxMainSM::RowSingletonPS::execute	0.53
soplex::SLUFactor::solveLeft	0.52
_int_realloc	0.51
soplex::SPxDevexPR::selectEnterDenseCoDim	0.49
soplex::SLUFactor::solveRight4update	0.49
std::istream::sentry::sentry	0.49
soplex::DSVectorBase<double >::makeMem	0.49
soplex::CLUFactor::solveLeft	0.48
native_write_msr_safe	0.48
run_timer_softirq	0.47
trigger_load_balance	0.46
__audit_syscall_entry	0.46
soplex::SPxMainSM::removeRowSingleton	0.46
soplex::Array<soplex::DSVectorBase<double >>::Array	0.46

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
rcu_check_callbacks	0.46
_raw_spin_lock	0.45
cputime_adjust	0.44
malloc_consolidate	0.43
__mpn_lshift	0.43
task_tick_fair	0.43
update_wall_time	0.43
soplex::SPxSolver::value	0.42
soplex::SPxScaler::maxRowRatio	0.42
soplex::CLUFactor::packColumns	0.42
soplex::UserTimer::stop	0.41
gzstream::gzstreambuf::underflow	0.41
gzread	0.40
soplex::SPxSolver::getEnterVals2	0.39
soplex::SPxSteepPR::selectLeaveSparse	0.38
system_call_after_swapgs	0.37
get_page_from_freelist	0.37
soplex::SPxDevexPR::selectEnterSparseCoDim	0.37

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SPxSolver::solve	0.37
soplex::SPxSolver::getEnterVals	0.36
soplex::SPxSolver::getLeaveVals	0.35
_raw_spin_lock_irqsave	0.34
cpuacct_charge	0.34
__list_del_entry	0.34
soplex::SPxSolver::computeFrhs	0.34
soplex::SPxSolver::shiftPvec	0.34
hrtimer_interrupt	0.32
perf_event_task_tick	0.32
__acct_update_integrals	0.30
timerqueue_add	0.30
mem_cgroup_charge_common	0.30
soplex::SPxSolver::perturbMin	0.29
soplex::SPxBasis::change	0.28
soplex::SVSetBase<double >::ensureMem	0.28
soplex::SPxSolver::terminate	0.28
__audit_syscall_exit	0.27

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
soplex::SPxScaler::maxColRatio	0.26
__mem_cgroup_commit_charge	0.26
memset@plt	0.25
soplex::SPxBasis::loadDesc	0.24
soplex::SPxSolver::unShift	0.24
unmap_page_range	0.23
sched_clock_cpu	0.23
soplex::CLUFactor::eliminateRowSingletons	0.22
soplex::SPxFastRT::maxReEnter	0.22
release_pages	0.21
get_pageblock_flags_group	0.21
notifier_call_chain	0.21
__times	0.21
__strcpy_sse2_unaligned	0.21
soplex::SPxFastRT::selectEnter	0.20
soplex::SLUFactor::solve2right4update	0.19
__mem_cgroup_uncharge_common	0.19
soplex::SVSetBase<double >::create	0.19

Table A.2: SoPlex Perf Data: Continued from Previous Page

Function	Average Time (ms)
system_call	0.19
soplex::SPxSolver::getLeaveVals2	0.19

Appendix B

Multi-Path Simplex Algorithm

Performance Data

Table B.1: Averaged Performance Results for Multi-Path Simplex versus SoPlex with Randomly Selected Pricing Algorithms for 100 Trials

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
map20.mps	5.50E+05	5.41E+10	8.1241	8.5817	8.6580	8.9328	0.95	0.94	0.91
map18.mps	5.50E+05	5.41E+10	8.9168	9.1431	9.6144	9.5691	0.98	0.93	0.93
neos631710.mps	8.34E+05	2.83E+10	283.1175	165.3114	145.1529	118.7310	1.71	1.95	2.38
neos1429212.mps	1.86E+06	2.44E+10	221.8736	107.3840	111.7682	121.4320	2.07	1.99	1.83
mspp16.mps	2.77E+07	1.64E+10	13.2943	11.3971	12.5915	14.4184	1.17	1.06	0.92
netdiversion.mps	6.15E+05	1.54E+10	186.3394	25.5333	26.6599	29.5934	7.30	6.99	6.30
ns1758913.mps	1.28E+06	1.12E+10	11.9836	17.1247	17.5836	18.8245	0.70	0.68	0.64
neos1367061.mps	2.60E+05	3.76E+09	180.5735	31.7684	17.5245	13.1604	5.68	10.30	13.72
vpphard.mps	3.72E+05	2.43E+09	5.8911	5.1539	4.7298	4.2777	1.14	1.25	1.38
tanglegram1.mps	2.05E+05	2.38E+09	0.1371	0.1889	0.2050	0.2211	0.73	0.67	0.62
app1-2.mps	1.99E+05	1.44E+09	7.4355	5.2403	4.2473	3.5685	1.42	1.75	2.08
unitcal_7.mps	1.28E+05	1.26E+09	6.2065	5.3392	5.1441	5.2559	1.16	1.21	1.18
neos10.mps	2.51E+05	1.10E+09	0.4546	0.3864	0.4124	0.4314	1.18	1.10	1.05
n3seq24.mps	3.23E+06	7.24E+08	4.7606	3.4742	3.2592	3.3004	1.37	1.46	1.44
triptim1.mps	5.15E+05	4.72E+08	0.7356	0.4029	0.3623	0.3475	1.83	2.03	2.12
ex9.mps	5.17E+05	4.26E+08	173.1979	70.9207	43.1179	40.5704	2.44	4.02	4.27

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
msc98-ip.mps	9.29E+04	3.35E+08	5.3558	2.3840	2.3763	2.4334	2.25	2.25	2.20
neos-934278.mps	1.26E+05	2.66E+08	20.9351	19.0565	19.2842	20.9731	1.10	1.09	1.00
stocfor3.mps	6.49E+04	2.62E+08	1.7276	1.2828	1.1289	1.0344	1.35	1.53	1.67
rocII-4-11.mps	2.43E+05	2.01E+08	0.1237	0.1687	0.1623	0.1661	0.73	0.76	0.74
net12.mps	8.04E+04	1.98E+08	1.9159	1.4122	0.9884	0.7669	1.36	1.94	2.50
neos-476283.mps	3.95E+06	1.19E+08	8.3704	3.3624	2.2970	1.9590	2.49	3.64	4.27
lectsched-4-obj.mps	8.24E+04	1.12E+08	0.0502	0.0790	0.0913	0.0958	0.64	0.55	0.52
bab5.mps	1.56E+05	1.07E+08	8.0805	5.0052	4.3084	4.1752	1.61	1.88	1.94
n3div36.mps	3.41E+05	9.92E+07	0.1629	0.1912	0.1836	0.1799	0.85	0.89	0.91
mzzv11.mps	1.35E+05	9.73E+07	321.3787	50.9608	40.0313	29.2708	6.31	8.03	10.98
ash608gpia-3col.mps	7.42E+04	9.04E+07	0.2679	0.2023	0.1910	0.1837	1.32	1.40	1.46
neos1597104.mps	3.31E+05	7.84E+07	0.2048	0.2827	0.2855	0.2993	0.72	0.72	0.68
rmatr100-p5.mps	2.62E+04	7.63E+07	1.6432	1.0877	0.6721	0.5554	1.51	2.44	2.96
dfl001.mps	3.56E+04	7.42E+07	177.3327	15.6517	10.9300	12.0416	11.33	16.22	14.73
opm2-z7-s2.mps	7.98E+04	6.43E+07	4.3274	2.5407	2.0390	1.9852	1.70	2.12	2.18
satellites1-25.mps	5.90E+04	5.40E+07	8.0872	7.6965	8.0243	8.8235	1.05	1.01	0.92
rmatr100-p10.mps	2.19E+04	5.34E+07	0.5527	0.4423	0.3323	0.2614	1.25	1.66	2.11

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
neos-1109824.mps	8.95E+04	4.40E+07	0.1223	0.3726	0.3973	0.4084	0.33	0.31	0.30
tanglegram2.mps	2.69E+04	4.23E+07	0.0190	0.0282	0.0375	0.0417	0.67	0.51	0.46
fit2p.mps	5.03E+04	4.06E+07	4.5195	1.5027	1.3981	1.3659	3.01	3.23	3.31
core2536-691.mps	1.78E+05	3.88E+07	204.0668	24.0405	12.6853	14.4193	8.49	16.09	14.15
neos13.mps	2.54E+05	3.81E+07	0.4568	0.4544	0.4275	0.4288	1.01	1.07	1.07
neos18.mps	2.46E+04	3.78E+07	0.0258	0.0400	0.0419	0.0457	0.64	0.62	0.57
rail507.mps	4.69E+05	3.21E+07	30.7102	7.9038	4.6132	4.2183	3.89	6.66	7.28
maros-r7.mps	1.45E+05	2.95E+07	1.0075	0.8566	0.7476	0.6745	1.18	1.35	1.49
80bau3b.mps	2.10E+04	2.22E+07	0.3350	0.3160	0.2982	0.2946	1.06	1.12	1.14
neos-1337307.mps	3.08E+04	1.62E+07	0.5593	0.3734	0.3018	0.3009	1.50	1.85	1.86
bnatt350.mps	1.91E+04	1.55E+07	0.0623	0.0383	0.0362	0.0396	1.63	1.72	1.58
neos-1601936.mps	7.25E+04	1.39E+07	5.6853	2.5564	1.9401	1.7989	2.22	2.93	3.16
neos-686190.mps	1.81E+04	1.34E+07	0.0520	0.0546	0.0432	0.0455	0.95	1.20	1.14
greenbea.mps	3.09E+04	1.29E+07	2.4592	2.1553	1.9523	1.4999	1.14	1.26	1.64
greenbeb.mps	3.09E+04	1.29E+07	1.1977	1.1633	1.0997	1.0675	1.03	1.09	1.12
ns1208400.mps	8.17E+04	1.24E+07	1.1844	1.1618	0.9175	0.8031	1.02	1.29	1.47
ns1688347.mps	6.69E+04	1.13E+07	0.0264	0.0348	0.0371	0.0376	0.76	0.71	0.70

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex Time (s)	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
				2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
d2q06c.mps	3.24E+04	1.12E+07	2.1241	1.6241	1.4598	1.3606	1.31	1.46	1.56
30n20b8.mps	1.10E+05	1.06E+07	0.0490	0.0801	0.0827	0.0897	0.61	0.59	0.55
pilot87.mps	7.32E+04	9.91E+06	11.0520	7.2626	7.1583	7.9324	1.52	1.54	1.39
zib54-UUE.mps	1.53E+04	9.32E+06	0.0326	0.0367	0.0388	0.0417	0.89	0.84	0.78
woodw.mps	3.75E+04	9.23E+06	0.1214	0.1367	0.0855	0.0858	0.89	1.42	1.41
sp98ic.mps	3.16E+05	8.99E+06	0.3207	0.2893	0.2498	0.2496	1.11	1.28	1.28
biella1.mps	7.15E+04	8.82E+06	9.8832	3.5623	1.7441	1.6200	2.77	5.67	6.10
truss.mps	2.78E+04	8.81E+06	2.5776	1.8123	1.3195	1.1578	1.42	1.95	2.23
pw-myciel4.mps	1.78E+04	8.65E+06	0.1598	0.1635	0.1553	0.1536	0.98	1.03	1.04
bnl2.mps	1.40E+04	8.11E+06	0.1872	0.1729	0.1579	0.1585	1.08	1.19	1.18
rmine6.mps	1.81E+04	7.76E+06	0.1055	0.1288	0.1006	0.1003	0.82	1.05	1.05
air04.mps	7.30E+04	7.33E+06	2.4130	1.3105	0.7839	0.6930	1.84	3.08	3.48
macrophage.mps	9.49E+03	7.15E+06	0.0175	0.0152	0.0131	0.0149	1.15	1.34	1.18
mine-166-5.mps	1.94E+04	7.00E+06	0.1159	0.1195	0.1078	0.0997	0.97	1.08	1.16
ship12l.mps	1.62E+04	6.25E+06	0.0254	0.0331	0.0233	0.0253	0.77	1.09	1.00
mine-90-10.mps	1.54E+04	5.64E+06	0.1717	0.1678	0.1555	0.1546	1.02	1.10	1.11
iis-pima-cov.mps	7.19E+04	5.53E+06	0.1890	0.1916	0.1843	0.1805	0.99	1.03	1.05

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex Time (s)	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
				2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
cycle.mps	2.07E+04	5.44E+06	0.0488	0.0613	0.0346	0.0353	0.80	1.41	1.38
pilot.mps	4.32E+04	5.26E+06	2.1334	1.9020	1.7517	1.7092	1.12	1.22	1.25
ns1830653.mps	1.01E+05	4.78E+06	0.1980	0.2071	0.2138	0.2138	0.96	0.93	0.93
n4-3.mps	1.40E+04	4.44E+06	0.0700	0.0604	0.0483	0.0501	1.16	1.45	1.40
stocfor2.mps	8.34E+03	4.38E+06	0.0637	0.0579	0.0460	0.0479	1.10	1.38	1.33
beasleyC3.mps	5.00E+03	4.38E+06	0.0442	0.0467	0.0415	0.0437	0.95	1.07	1.01
acc-tight5.mps	1.61E+04	4.09E+06	0.5078	0.4936	0.4603	0.4560	1.03	1.10	1.11
rococoC10-001000.mps	1.18E+04	4.03E+06	0.1007	0.0823	0.0754	0.0697	1.22	1.34	1.44
aflow40b.mps	6.78E+03	3.93E+06	0.1374	0.0886	0.0524	0.0554	1.55	2.62	2.48
mcsched.mps	8.09E+03	3.68E+06	0.0703	0.0759	0.0537	0.0561	0.93	1.31	1.25
sctap3.mps	8.87E+03	3.67E+06	0.0162	0.0196	0.0132	0.0137	0.82	1.23	1.18
ship08l.mps	1.28E+04	3.33E+06	0.0183	0.0258	0.0220	0.0236	0.71	0.83	0.77
czprob.mps	1.07E+04	3.27E+06	0.0575	0.0657	0.0444	0.0452	0.87	1.30	1.27
ship12s.mps	8.18E+03	3.18E+06	0.0119	0.0158	0.0102	0.0112	0.75	1.17	1.07
neos-916792.mps	1.34E+05	2.81E+06	0.1280	0.1395	0.1430	0.1461	0.92	0.89	0.88
degen3.mps	2.46E+04	2.73E+06	0.6765	0.4612	0.4315	0.4246	1.47	1.57	1.59
roll3000.mps	2.94E+04	2.68E+06	0.1262	0.1238	0.0911	0.0870	1.02	1.39	1.45

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
sp98ir.mps	7.17E+04	2.57E+06	0.1674	0.1393	0.1309	0.1141	1.20	1.28	1.47
d6cube.mps	3.77E+04	2.57E+06	0.1147	0.1317	0.0902	0.0833	0.87	1.27	1.38
sierra.mps	7.30E+03	2.50E+06	0.0115	0.0127	0.0119	0.0131	0.90	0.96	0.87
binkar10_1.mps	4.50E+03	2.36E+06	0.0249	0.0297	0.0212	0.0222	0.84	1.18	1.12
ganges.mps	6.91E+03	2.20E+06	0.0223	0.0259	0.0184	0.0199	0.86	1.21	1.12
pilotnov.mps	1.31E+04	2.12E+06	0.3665	0.3488	0.3119	0.3064	1.05	1.18	1.20
sctap2.mps	6.71E+03	2.05E+06	0.0122	0.0112	0.0110	0.0118	1.09	1.10	1.03
pilot-we.mps	9.13E+03	2.01E+06	0.3467	0.3253	0.2726	0.2580	1.07	1.27	1.34
nesm.mps	1.33E+04	1.94E+06	0.2054	0.2097	0.1822	0.1809	0.98	1.13	1.14
pilot-ja.mps	1.47E+04	1.87E+06	0.7622	0.6436	0.5606	0.5408	1.18	1.36	1.41
ship08s.mps	7.11E+03	1.86E+06	0.0086	0.0111	0.0091	0.0097	0.77	0.94	0.88
neos-849702.mps	1.93E+04	1.81E+06	0.3338	0.2467	0.1495	0.1265	1.35	2.23	2.64
neos-1396125.mps	5.51E+03	1.73E+06	0.1488	0.1288	0.0925	0.0822	1.16	1.61	1.81
reblock67.mps	7.50E+03	1.69E+06	0.0963	0.0863	0.0653	0.0605	1.12	1.48	1.59
iis-bupa-cov.mps	3.84E+04	1.66E+06	0.1468	0.1482	0.1269	0.1134	0.99	1.16	1.29
scfxm3.mps	7.78E+03	1.36E+06	0.0411	0.0448	0.0270	0.0289	0.92	1.52	1.42
25fv47.mps	1.04E+04	1.29E+06	0.2857	0.2391	0.1668	0.1631	1.20	1.71	1.75

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
maros.mps	9.61E+03	1.22E+06	0.0572	0.0578	0.0453	0.0464	0.99	1.26	1.23
modszk1.mps	3.17E+03	1.11E+06	0.0211	0.0199	0.0160	0.0169	1.06	1.32	1.25
scsd8.mps	8.58E+03	1.09E+06	0.0745	0.0672	0.0469	0.0462	1.11	1.59	1.61
fit1p.mps	9.87E+03	1.05E+06	0.0656	0.0493	0.0403	0.0421	1.33	1.63	1.56
qiu.mps	3.43E+03	1.00E+06	0.0401	0.0419	0.0246	0.0258	0.96	1.63	1.55
shell.mps	3.56E+03	9.51E+05	0.0095	0.0111	0.0110	0.0116	0.86	0.87	0.82
perold.mps	6.02E+03	8.60E+05	0.2310	0.2357	0.1931	0.1884	0.98	1.20	1.23
ship04l.mps	6.33E+03	8.51E+05	0.0048	0.0056	0.0061	0.0067	0.84	0.78	0.71
bnl1.mps	5.12E+03	7.56E+05	0.0667	0.0653	0.0403	0.0410	1.02	1.66	1.63
gfrd-pnc.mps	2.38E+03	6.73E+05	0.0095	0.0104	0.0102	0.0109	0.92	0.94	0.87
wood1p.mps	7.02E+04	6.33E+05	0.0364	0.0615	0.0377	0.0377	0.59	0.96	0.96
csched010.mps	6.38E+03	6.17E+05	0.1686	0.1548	0.1236	0.1207	1.09	1.36	1.40
scfxm2.mps	5.18E+03	6.03E+05	0.0253	0.0236	0.0159	0.0167	1.07	1.59	1.52
ship04s.mps	4.35E+03	5.86E+05	0.0037	0.0045	0.0049	0.0053	0.81	0.76	0.69
pg5_34.mps	7.70E+03	5.85E+05	0.0573	0.0496	0.0357	0.0372	1.16	1.61	1.54
scrs8.mps	3.18E+03	5.73E+05	0.0127	0.0118	0.0117	0.0124	1.07	1.09	1.02
seba.mps	4.35E+03	5.29E+05	0.0018	0.0023	0.0027	0.0031	0.79	0.68	0.60

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
gmu-35-40.mps	4.84E+03	5.11E+05	0.0078	0.0076	0.0074	0.0077	1.04	1.06	1.01
standmps.mps	3.68E+03	5.02E+05	0.0036	0.0039	0.0041	0.0045	0.92	0.87	0.79
pigeon-10.mps	8.15E+03	4.56E+05	0.0058	0.0067	0.0061	0.0066	0.87	0.96	0.88
fffff800.mps	6.23E+03	4.47E+05	0.0217	0.0199	0.0186	0.0195	1.09	1.17	1.11
grow22.mps	8.25E+03	4.16E+05	0.1852	0.1723	0.1455	0.1431	1.07	1.27	1.29
pilot4.mps	5.14E+03	4.10E+05	0.0894	0.0920	0.0675	0.0689	0.97	1.33	1.30
standata.mps	3.03E+03	3.86E+05	0.0018	0.0024	0.0027	0.0029	0.74	0.66	0.62
iis-100-0-cov.mps	2.30E+04	3.83E+05	0.0495	0.0499	0.0420	0.0419	0.99	1.18	1.18
danoint.mps	3.23E+03	3.46E+05	0.0895	0.0935	0.0661	0.0667	0.96	1.35	1.34
finnis.mps	2.31E+03	3.05E+05	0.0100	0.0108	0.0092	0.0097	0.93	1.09	1.04
newdano.mps	2.18E+03	2.91E+05	0.0140	0.0126	0.0114	0.0118	1.11	1.22	1.18
bienst2.mps	2.18E+03	2.91E+05	0.0164	0.0162	0.0117	0.0119	1.01	1.40	1.38
eilB101.mps	2.41E+04	2.82E+05	0.0773	0.0671	0.0459	0.0449	1.15	1.68	1.72
etamacro.mps	2.41E+03	2.75E+05	0.0111	0.0114	0.0117	0.0122	0.98	0.95	0.91
fit2d.mps	1.29E+05	2.63E+05	1.0199	0.9233	0.8400	0.8262	1.10	1.21	1.23
degen2.mps	3.98E+03	2.37E+05	0.0561	0.0611	0.0390	0.0398	0.92	1.44	1.41
scagr25.mps	1.55E+03	2.36E+05	0.0194	0.0166	0.0162	0.0171	1.17	1.20	1.13

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
scsd6.mps	4.32E+03	1.98E+05	0.0084	0.0117	0.0079	0.0084	0.72	1.06	1.00
tuff.mps	4.52E+03	1.95E+05	0.0044	0.0049	0.0051	0.0054	0.90	0.87	0.82
grow15.mps	5.62E+03	1.94E+05	0.0662	0.0774	0.0415	0.0389	0.86	1.59	1.70
stair.mps	3.86E+03	1.66E+05	0.0429	0.0529	0.0312	0.0326	0.81	1.37	1.31
agg3.mps	4.30E+03	1.56E+05	0.0037	0.0040	0.0043	0.0047	0.92	0.86	0.79
agg2.mps	4.28E+03	1.56E+05	0.0030	0.0035	0.0038	0.0041	0.87	0.78	0.73
scfxm1.mps	2.59E+03	1.51E+05	0.0086	0.0090	0.0082	0.0087	0.95	1.04	0.99
dfn-gwin-UUM.mps	2.63E+03	1.48E+05	0.0042	0.0043	0.0041	0.0045	0.98	1.04	0.94
ran16x16.mps	1.02E+03	1.47E+05	0.0045	0.0048	0.0056	0.0060	0.93	0.80	0.75
eil33-2.mps	4.42E+04	1.45E+05	0.0359	0.0334	0.0357	0.0387	1.07	1.01	0.93
sctap1.mps	1.69E+03	1.44E+05	0.0045	0.0050	0.0053	0.0055	0.91	0.86	0.82
bandm.mps	2.49E+03	1.44E+05	0.0173	0.0236	0.0115	0.0122	0.73	1.49	1.42
scorpion.mps	1.43E+03	1.39E+05	0.0033	0.0040	0.0039	0.0043	0.82	0.84	0.77
boeing1.mps	3.49E+03	1.35E+05	0.0108	0.0112	0.0115	0.0121	0.97	0.94	0.89
glass4.mps	1.82E+03	1.28E+05	0.0014	0.0019	0.0021	0.0024	0.73	0.67	0.60
capri.mps	1.77E+03	9.57E+04	0.0036	0.0039	0.0041	0.0044	0.92	0.87	0.82
agg.mps	2.41E+03	7.95E+04	0.0020	0.0026	0.0024	0.0026	0.79	0.83	0.77

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
enlight14.mps	1.12E+03	7.68E+04	0.0005	0.0008	0.0012	0.0015	0.62	0.41	0.34
cov1075.mps	1.43E+04	7.64E+04	0.0398	0.0444	0.0376	0.0391	0.90	1.06	1.02
bore3d.mps	1.43E+03	7.34E+04	0.0016	0.0020	0.0023	0.0025	0.79	0.69	0.62
timtab1.mps	8.29E+02	6.79E+04	0.0008	0.0011	0.0014	0.0016	0.72	0.59	0.52
forplan.mps	4.56E+03	6.78E+04	0.0056	0.0062	0.0060	0.0063	0.91	0.93	0.89
e226.mps	2.58E+03	6.29E+04	0.0072	0.0075	0.0073	0.0075	0.97	1.00	0.97
scsd1.mps	2.39E+03	5.85E+04	0.0021	0.0026	0.0024	0.0027	0.82	0.87	0.79
enlight13.mps	9.62E+02	5.71E+04	0.0006	0.0009	0.0012	0.0015	0.61	0.46	0.38
brandy.mps	2.15E+03	5.48E+04	0.0057	0.0055	0.0052	0.0055	1.03	1.10	1.03
m100n500k4r1.mps	2.00E+03	5.00E+04	0.0042	0.0044	0.0047	0.0051	0.94	0.89	0.82
lotfi.mps	1.08E+03	4.71E+04	0.0028	0.0033	0.0036	0.0039	0.85	0.78	0.73
beaconfd.mps	3.38E+03	4.53E+04	0.0014	0.0018	0.0020	0.0023	0.79	0.69	0.61
grow7.mps	2.61E+03	4.21E+04	0.0129	0.0132	0.0067	0.0073	0.98	1.93	1.78
sc205.mps	5.51E+02	4.16E+04	0.0026	0.0029	0.0031	0.0033	0.88	0.82	0.77
vtp-base.mps	9.08E+02	4.02E+04	0.0009	0.0013	0.0015	0.0018	0.73	0.61	0.53
mik-250-1-100-1.mps	5.35E+03	3.79E+04	0.0012	0.0017	0.0018	0.0020	0.69	0.64	0.58
share1b.mps	1.15E+03	2.63E+04	0.0035	0.0038	0.0034	0.0036	0.92	1.04	0.98

Table B.1: Averaged Performance Results for Multi-Path Simplex: Continued from Previous Page

Problem	Non Zeros	Size	SoPlex	Multi-Path Simplex Time (s)			Multi-Path Simplex Speed Up		
			Time (s)	2 Threads	3 Threads	4 Threads	2 Threads	3 Threads	4 Threads
israel.mps	2.27E+03	2.47E+04	0.0033	0.0034	0.0034	0.0036	0.99	0.99	0.92
fit1d.mps	1.34E+04	2.46E+04	0.0184	0.0185	0.0141	0.0145	0.99	1.30	1.27
boeing2.mps	1.20E+03	2.37E+04	0.0022	0.0026	0.0024	0.0026	0.85	0.92	0.84
noswot.mps	7.35E+02	2.33E+04	0.0011	0.0014	0.0020	0.0021	0.78	0.57	0.52
ns1766074.mps	6.66E+02	1.82E+04	0.0006	0.0009	0.0011	0.0013	0.66	0.53	0.43
scagr7.mps	4.20E+02	1.81E+04	0.0020	0.0021	0.0022	0.0025	0.93	0.89	0.79
recipe.mps	6.63E+02	1.64E+04	0.0006	0.0009	0.0011	0.0013	0.65	0.53	0.44
stocfor1.mps	4.47E+02	1.30E+04	0.0013	0.0016	0.0018	0.0020	0.81	0.73	0.66
sc105.mps	2.80E+02	1.08E+04	0.0009	0.0012	0.0015	0.0017	0.72	0.59	0.50
share2b.mps	6.94E+02	7.58E+03	0.0014	0.0016	0.0018	0.0020	0.85	0.76	0.69
blend.mps	4.91E+02	6.14E+03	0.0014	0.0015	0.0015	0.0017	0.95	0.92	0.80
adlittle.mps	3.83E+02	5.43E+03	0.0009	0.0012	0.0014	0.0017	0.77	0.64	0.54
sc50a.mps	1.30E+02	2.40E+03	0.0006	0.0009	0.0011	0.0014	0.61	0.50	0.40
sc50b.mps	1.18E+02	2.40E+03	0.0006	0.0010	0.0009	0.0012	0.62	0.63	0.49
kb2.mps	2.86E+02	1.76E+03	0.0006	0.0009	0.0011	0.0013	0.69	0.58	0.48
afiro.mps	8.30E+01	8.64E+02	0.0004	0.0007	0.0008	0.0011	0.56	0.46	0.35

Curriculum Vitae

Name:	Bradley de Vlugt
Post-Secondary Education and Degrees:	The University of Western Ontario London, ON 2009 - 2013 BESc
	University of Western Ontario London, ON 2013 - 2015 MESc
Honours and Awards:	Ontario Graduate Scholarship 2014-2015 Vale INCO Reserved Scholarship 2009-2013 PSAC Groulx Scholarship 2009
Related Work Experience:	Research Assistant The University of Western Ontario 2013 - 2015 Teaching Assistant The University of Western Ontario 2013 - 2015

Publications:

B.de Vlugt, M. Mirahmadi, A. Shami and S. Primak, Hardware Accelerated Linear Programming: Parallelizing the Simplex Algorithm with OpenCL, IEEE Super Computing, New Orleans, LA, 2015.