FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Dávid Kubek

Experimental Analysis of the Simplex Method for the Multicommodity Flow Problem

Computer Science Institute of Charles University

Supervisor of the bachelor thesis:	Mgr. Martin Koutecký, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2023

To my family, parents, and sisters Laura and Linda, I dedicate this thesis with heartfelt gratitude for their unwavering support, love, and encouragement throughout my journey. My special thanks go to my supervisor, Mgr. Martin Koutecký, Ph.D. for his invaluable guidance, patience, and expertise. I am also deeply grateful to my friends, whose support provided the motivation and strength necessary to complete this endeavor.

Title: Experimental Analysis of the Simplex Method for the Multicommodity Flow Problem

Author: Dávid Kubek

Institute: Computer Science Institute of Charles University

Supervisor: Mgr. Martin Koutecký, Ph.D., Computer Science Institute of Charles University

Abstract: This thesis investigates the multicommodity min-cost flow (MMCF) problem. We aim to contribute to the search for a combinatorial algorithm for MMCF. The simplex method is employed to examine the vertices of the polyhedron of feasible solutions using experimental analysis on a set of publicly available MMCF instances. To achieve this, we develop a solver capable of tracing solutions in each iteration of the algorithm in exact arithmetic, which was not available in existing solvers. Our investigation focuses on the fractionality of MMCF problems and the impact of different pivoting rules, particularly whether the fractionality is exponential or polynomial with respect to increasing dimension. Our findings suggest that fractionality exhibits exponential behavior.

Keywords: simplex, linear programming, experiment, flow problem

Název práce: Experimentální analýza simplexové metody na problému multikomoditního toku

Autor: Dávid Kubek

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Martin Koutecký, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Tato práce se zabývá problémem multikomoditního toku minimální ceny (MMCF). Naším cílem je přispět k hledání kombinatorického algoritmu pro MMCF. Použili jsme simplexovou metodu k experimentálnímu prozkoumání vrcholů polyedru přípustných řešení na sadě veřejně dostupných instancí MMCF. K dosažení tohoto cíle jsme vyvinuli řešič, který je schopen sledovat řešení v každé iteraci algoritmu v přesné aritmetice; tato funkcionalita nebyla k dispozici v existujících řešičích. Zaměřujeme se na zlomkovost MMCF instancí a vliv volby pivotovacího pravidla, zejména zda je zlomkovost exponenciální nebo polynomiální s ohledem na rostoucí dimenzi problému. Naše zjištění naznačují, že zlomkovost vykazuje exponenciální chování.

Klíčová slova: simplex, lineární programování, experiment, tokový problém

Contents

Introduction

Motivation

Notwork flows are an important part of combinatorial optimization [1], and after a short reflection they can be found everywhere you look. Electrical and power networks bring lighting into our homes. National highway systems, rail networks, and airline service networks facilitate long-distance travel for work, and exploration of new places. Manufacturing and distribution networks provide access to essential goods and consumer products. Furthermore, computer and social networks have revolutionized the way we share information and conduct our business and personal lives.

In all these problem domains, and many others, the primary objective is to move an entity (be it electricity, a product, a person, a vehicle, or a message) from one point to another within the underlying network as efficiently as possible. This efficiency is essential not only from the point of providing a good service to its users, but also to optimize the utilization of the underlying and typically expensive transmission facilities. Network flow problems, provide valuable insight into developing effective solutions for these real-world challenges, making them a vital aspect of combinatorial optimization.

The simplest variant of a network flow problem, the maximum flow (MF) problem (see Section 1.3.1), aims to transport the maximum possible amount of a commodity through a given network. This problem has been extensively studied, and numerous algorithms, such as the algorithms of Ford and Fulkerson, Dinitz, and Goldberg (see [2]), have been developed to find optimal solutions.

A natural extension of the maximum flow problem involves incorporating costs for individual arcs within the network. Instead of maximizing the transported amount, the focus shifts to minimizing transportation costs for fixed commodity demands. This leads to the minimum cost flow (MCF) problem (see Section 1.3.2), for which an efficient combinatorial algorithm, the network simplex algorithm exists (see [1]).

Further generalization of MCF brings us to the multicommodity min-cost flow (MMCF) problem (see Section 1.3.3), where multiple commodities must be transported within a single network. Although the MMCF problem can be solved using linear programming techniques, a combinatorial algorithm remains elusive.

The MF and MCF problems share the same constraint matrix, which is totally unimodular, implying that the corresponding polyhedron has all vertices integral vertices. It is not difficult to construct a linear programming problem with a vertex containing a large denominator. The largest denominator present in a coordinate among all vertices of a polyhedron is referred to as its *fractionality* (see Definition 24). It is of significant interest to understand the fractionality of the MMCF problem, both in general terms and for specific benchmark instances. This investigation provides an opportunity to analyze various problem instances and their fractionality, emphasizing the importance of this notion in the context of the MMCF problem.

Our Contribution

In this thesis, we aim to make a contribution to the pursuit of a combinatorial algorithm for the MMCF problem. According to the work of Hladík [3], MMCF matrices exhibit large circuits of order $2^{\Omega(n)}$. This finding implies the existence of linear programming (LP) problems with these matrices, characterized by integral inputs and high fractionality of the same order (refer to Lemma 2). However, no MMCF problem with such high fractionality has been constructed.

In this study, we use the simplex method (refer to Section 1.4) to examine the vertices of the polyhedron of feasible solutions for a collection of publicly available MMCF instances [4]. To focus on the investigation of problem fractionality, we first require a solver capable of solving the problem using exact arithmetic while simultaneously being able to trace the solutions in each iteration of the algorithm. As no existing freely available solver provides this functionality, one of our primary contributions is the development of such a tool.

Using this tool, we investigate the distribution of fractionality in linear optimization problems, focusing on the impact of different pivoting rules on MMCF and benchmark problems from the Netlib library [5]. We observe that the Dantzig rule (refer to Definition 20) exhibits a considerably higher mean fractionality compared to other rules for MMCF instances, while for Netlib instances, all rules produce similar fractionality levels.

Moreover, we investigate the relationship between problem complexity and fractionality, considering the number of iterations and problem dimension as complexity measures. The fractionality of a problem tends to increase with the number of iterations, particularly for the Dantzig rule in MMCF instances. We also observe a positive correlation between problem dimension and fractionality, with the Dantzig rule having the most significant impact on this relationship in MMCF instances. In contrast, the Netlib dataset on a whole exhibits greater variation and weaker relationships between fractionality and complexity measures, with no significant difference between the effects of different pivoting rules.

We also focus on the question whether the fractionality is exponential or polynomial with respect to increasing dimension. Our findings suggest that fractionality exhibits exponential behavior.

Organization of the Thesis

Chapter 1 introduces the notation, terminology, and fundamental concepts of linear programming, the simplex method, and network flows. In Chapter 2, we detail the efforts made to implement a simplex method with exact arithmetic and the functionality to trace transient solutions, as well as our final approach of addressing the problem. Chapter 3 contain the analysis, interpretation, and summarization of the results obtained from the problem instances we have solved, and the identification of potential avenues for future research. While the exposition of the simplex method in Chapter 1 provides a general understanding of the method, Appendix A offers a more comprehensive overview of the simplex algorithm necessary for making modifications to modern implementations. Lastly, in Appendix B, we finish with a brief manual on using the software we have developed.

Chapter 1

Preliminaries

1.1 Notation

In this section, we introduce the notation that will be employed throughout the following text.

For $n \in \mathbb{Z}$ we denote $[n]$ to be the set $\{1, 2, \ldots, n\}$.

Vectors are denoted in bold font (e.g., $\mathbf{x}$, $\mathbf{y}$), while their elements and all other scalars are denoted in standard font (e.g., $x_1, x_2, \ldots$; t). We denote $\mathbf{x} \in \mathbb{F}^n$ to be an n-element vector from the field $\mathbb{F}$, and similarly, we denote $A \in \mathbb{F}^{m \times n}$ to be a matrix with m rows and n columns, with entries from the field $\mathbb{F}$.

The coordinates of a vector may be indexed by a set, for example $\mathbf{c} \in \mathbb{R}^E$ has an entry c_e for each $e \in E$. Such vector may also be thought of as a function $c : E \to \mathbb{R}$ and the value may be interchangeably expressed as $c_e = c(e)$.

It is assumed that the vector $\mathbf{x}$ is always a column vector, making its transpose $\mathbf{x}^T$ a row vector. We use superscripts for indexing; for instance, a collection of vectors $\mathbf{x}^{\mathbf{1}}, \mathbf{x}^{\mathbf{2}}, \ldots, \mathbf{x}^{\kappa}$ implies that x^i_j denotes the j-th entry of vector $\mathbf{x}^i$. The i-th row of matrix $A \in \mathbb{F}^{m \times n}$ is given by the vector $A_{i,*} \in \mathbb{F}^m$, and the j-th column corresponds to the vector $A_{*,j} \in \mathbb{F}^n$. The *identity matrix* $I_n \in \mathbb{F}^{n \times n}$ is defined as a matrix with $I_{i,i} = 1$ for all $i = 1, 2, \ldots, n$ and $I_{i,j} = 0$ for $i \neq j$.

In certain cases, it is useful to refer to a subvector or submatrix induced by a set of indices. Let $\mathbf{x} \in \mathbb{F}^n$, $A \in \mathbb{F}^{m \times n}$, and $\mathcal{I} = [n]$, with $B \subseteq \mathcal{I}$. Then, A_B represents a new matrix formed by selecting the i-th column of A for all $i \in B$. For instance, for $B = \{2, 4, 5\}$, we have:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \to A_B = \begin{pmatrix} a_{12} & a_{14} & a_{15} \\ a_{22} & a_{24} & a_{25} \\ a_{32} & a_{34} & a_{35} \\ a_{42} & a_{44} & a_{45} \\ a_{52} & a_{54} & a_{55} \end{pmatrix}$$

The vector $\mathbf{x}_B$ is defined in a similar manner. While the order of elements in columns is not crucial, it is necessary to maintain consistency in the index set usage. For instance, indices can be assumed to always be ordered by any chosen linear ordering.

Furthermore, we adopt the standard notation $\mathbf{x} \leq \mathbf{y}$ to indicate that $x_i \leq y_i$ for all i. Analogously, we use $\mathbf{x} = \mathbf{y}$, $\mathbf{x} < \mathbf{y}$, etc.

1.2 Linear Programming

The scope of this work is limited in its ability to fully explain all the details of linear programming (LP). While we provide an overview of some of the key concepts and techniques involved, there is much more we had to omit. For readers interested in delving deeper into linear programming, we highly recommend consulting the works of Matoušek [6] and Chvátal [7]. These authors have written extensively on the subject, and their texts provide a comprehensive treatment of the theory and applications of linear programming.

We note that while both books by Matoušek and Chvátal deal with the theory of linear programming, they have slightly different emphases. Matoušek's book is more oriented towards the theoretical aspects of the subject, providing a rigorous treatment of the mathematical underpinnings of linear programming. Chvátal's book, on the other hand, is invaluable for its insights into the practical implementation of algorithms and real-world usage of linear programming. Therefore, we also source most of the theoretical results from the book of Matoušek and practical results from Chvátal's book.

Linear programming is a well-researched area of optimization problems due to its versatility in modeling a wide range of real-world problems. In fact, linear programming has found extensive application in economics and corporate management, where it is employed in tasks such as planning, production, transportation, and decision-making [8]. These applications often involve maximizing revenue or minimizing costs in production schemes.

Linear programming models have proven to be useful in approximating nonlinear real-world problems (for example [9]). In fact, many combinatorial problems, such as the shortest path, maximum flow, maximum matching and others can be formulated as linear programs (for these and many other applications refer to [10]).

Beyond direct applications, linear programming is often utilized as a subroutine in more complex algorithms. For instance, it is employed in approximation algorithms such as LP SAT, Scheduling (see [11]) or optimization algorithms (Frank-Wolfe [12]) to solve complex optimization problems.

The formal definition of a problem of linear programming follows.

Definition 1 (Linear Programming Problem). *A* Linear Programming problem *is a problem of mathematical optimization, such that the objective and constraint functions are linear:*

$$\begin{aligned} \textit{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & \mathbf{a}_i^T\mathbf{x} \leq b_i, \qquad i = 1, \dots, m \end{aligned}$$

or more concisely:

$$\textit{maximize} \quad \mathbf{c}^T\mathbf{x} \qquad \textit{subject to} \quad A\mathbf{x} \leq \mathbf{b} \tag{LP}$$

where $\mathbf{c} \in \mathbb{R}^n$ *is called the* cost vector*,* $A \in \mathbb{R}^{m\times n}$ *is the* constraint matrix *(*$\mathbf{a}_i$ *being the* i*-th row) and* $\mathbf{b} = (b_1, \dots, b_m) \in \mathbb{R}^m$ *is the* right hand side*, and* $\mathbf{x} \in \mathbb{R}^n$ *is the vector of* decision variables.

Definition 2 (Feasible solution). *Given an* (LP)*, the vector* $\mathbf{x} \in \mathbb{R}^n$ *is said to be* feasible *if it satisfies* all *constraints, i.e., if* $A\mathbf{x} \leq \mathbf{b}$.

Definition 3 (Optimal solution). *Given an* (LP) *the feasible solution* $\mathbf{x}^* \in \mathbb{R}^n$ *is said to be* optimal *if it yields the maximum objective value among all feasible solutions, i.e.*

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{c}^T\mathbf{x}^*$$

for all $\mathbf{x} \in \mathbb{R}$ *feasible.*

Linear programming (LP) problems can be expressed in many forms. For sake of simplicity of description, we will be assuming that an LP problem is given to us in a so called *standard form.* This is without loss of generality as the problems can be easily converted between different forms (see [6]).

Definition 4 (Standard Form). *A LP problem is in* standard form *if it has the form*

$$\begin{aligned} \textit{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{LP-SF}$$

For the use with the simplex algorithm (discussed in Section 1.4) a slightly different formulation of the problem is required

Definition 5 (Computational Standard Form)**.** *An LP problem is in* computational standard form *if it has the form*

$$\begin{aligned} \textit{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{LP-CSF}$$

and additionally

- *the system* $A\mathbf{x} = \mathbf{b}$ *has at least one solution;*
- A *has full row rank.*

1.2.1 Geometric Interpretation

One particularly appealing aspect of linear programming problems is their relatively simple geometric interpretation. The geometric perspective enables us to comprehend problems from a different angle, often yielding new insights. Consequently, we can consider the problem in more tangible terms rather than relying solely on abstract vector descriptions.

Definition 6 (Feasible Region)**.** *For an* (LP) *we define the* feasible region *of the problem as*

$$M = \{x \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{b}\}$$

Definition 7 (Hyperplane, Half-Space)**.** *We define* hyperplane

$$h = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^T\mathbf{x} = b\}$$

and half-space *as*

$$\begin{aligned} h^- &= \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^T\mathbf{x} \leq b\} \\ h^+ &= \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^T\mathbf{x} \geq b\} \end{aligned}$$

for $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$ *and* $b \in \mathbb{R}$.

In other words, half-space is a region of space that includes all points on one side of a hyperplane.

Each constraint of the problem defines a half-space. The feasible region, also referred to as solution space, of a linear program can be understood geometrically

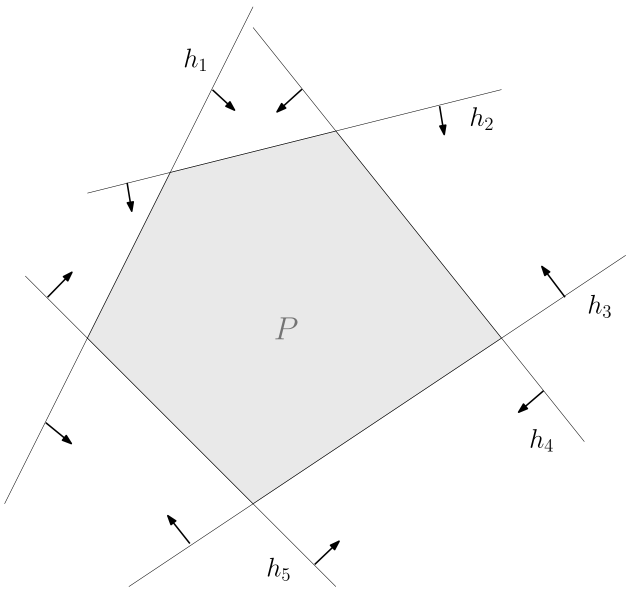


Figure 1.1 This figure presents a polyhedron P, which is formed by the intersection of five half-spaces bounded by hyperplanes $h_1, h_2, \ldots, h_5$. The arrows indicate the respective regions of the space spanned by each half-space.

as the intersection of half-spaces defined by the constraints of the problem. Simply put, it is the region of space that satisfies all of the constraints simultaneously.

The feasible region can be represented as a polyhedron (see Figure 1.1), which is the intersection of a finite number of half-spaces. The faces of a polyhedron correspond to the half-spaces that define it. For more details see [6].

Of particular importance for us will be the definition of a *vertex*. As we will see shortly, am optimal solution of a LP will be found by moving from one vertex of the feasible region to another.

Definition 8 (Vertex). *A vector* $\mathbf{v}$ *is called a* vertex *of a polyhedron* $P \subseteq \mathbb{R}^n$ *if* $\mathbf{v} \in P$ *and there exists a nonzero vector* $\mathbf{c} \in \mathbb{R}^n$ *such that* $\mathbf{c}^T\mathbf{v} > \mathbf{c}^T\mathbf{y}$ *for all* $\mathbf{y} \in P - \{\mathbf{v}\}$. *Geometrically it means that the hyperplane* $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{c}^T\mathbf{x} = \mathbf{c}^T\mathbf{v}\}$ *touches the polyhedron* P *exactly at* $\mathbf{v}$.

This definition is illustrated by Figure 1.2.

By representing the feasible region as a polyhedron, we can easily visualize the constraints of the problem and the optimal solutions that lie at its vertices. This visualization can help us identify the most promising regions of the solution space and guide our search for the optimal solution.

Definition 9 (Infeasibility). *We say that* (LP) *is* infeasible *when there is no feasible solution that satisfies all the constraints of the problem.*

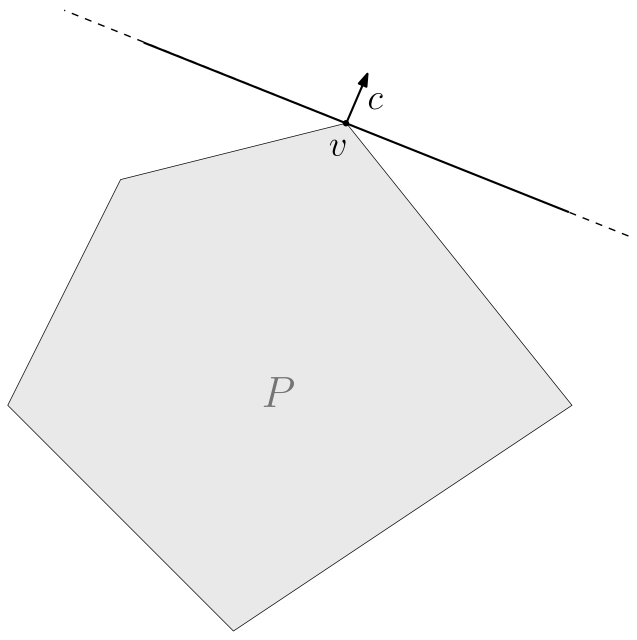


Figure 1.2 In accordance with Definition 8, this figure illustrates the relationship between the polyhedron P and the vector **c**, which defines the vertex v. Specifically, the definition says that the hyperplane, determined by the vector **c**, intersects P only at the point v.

Definition 10 (Unboundedness)**.** *We say that* (LP) *is* unbounded *if for every* $k \in \mathbb{R}$ *there exists a feasible solution* $\mathbf{x}$*, such that* $\mathbf{c}^T\mathbf{x} \geq k$*.*

Linear programming unboundedness occurs when the objective function of a linear program can be increased without limit, while satisfying all the constraints. In other words, the feasible region of the linear program is unbounded in the direction of the objective function.

If a linear programming problem is infeasible or unbounded, it may be necessary to relax some of the constraints or to reformulate the problem in a different way to make it solvable. Alternatively, the problem may be abandoned if it is determined that a feasible solution does not exist or if the cost of finding one is too high.

1.3 Network Flows

In this section we introduce the basic notion of network flows together with some standard definitions and formulations of different kinds of problems.

1.3.1 Maximum Flow

The most simple variant of a network flow problem is the maximum flow problem. In general, we have some network represented by a graph with two designated nodes. Each edge of the network has a capacity on how much it can transport. More formally:

Definition 11 (Network)**.** Network *is the 5-tuple* $(V, E, s, t, \mathbf{c})$ *where*

- (V, E) *is an oriented graph,*
- $\mathbf{c} \in \mathbb{R}_+^E$ *is an assignment of* capacities *to edges,*
- $s, t \in V$ *are distinct vertices of the graph. The vertex* s *is called* source *or* origin *and* t *is called* target *or* destination.

The notation and definitions are adopted from [13] and [2].

The objective is to identify the maximum amount of some commodity that can be transported between the origin and the destination within the network. We do this by finding an assignment $\mathbf{f} \in \mathbb{R}^E$ with each entry representing the amount of the commodity that flows through the corresponding edge.

The following definitions will be useful in formulating the problem.

Definition 12 (Inflow, Outflow, Excess)**.** *Given a network* $(V, E, s, t, \mathbf{c})$ *and a vector* $\mathbf{f} \in \mathbb{R}^E$ *we define for all* $v \in V$ *the:*

- Inflow
$$f_v^+ = \sum_{e=(\cdot,v)} f_e$$
- Outflow
$$f_v^- = \sum_{e=(v,\cdot)} f_e$$
- Excess
$$f_v^\Delta = f_v^+ - f_v^-$$

The inflow, outflow and excess also implicitly define the vectors $\mathbf{f}_-, \mathbf{f}_+, \mathbf{f}_\Delta \in \mathbb{R}^V$. Note that f_v^+, f_v^- and f_v^Δ are all linear functions of $\mathbf{f}$.

Definition 13 (Flow). *For a given network $(V, E, s, t, \mathbf{c})$ a vector* $\mathbf{f} \in \mathbb{R}^E$ *is a* flow *if the following constraints hold:*

1. *capacity constraints*
$$\mathbf{0} \leq \mathbf{f} \leq \mathbf{c}$$
2. *Kirchhoff's law*
$$(\forall v \in V - \{s, t\}) \quad f_v^\Delta = 0$$

The value *of the flow* $|\mathbf{f}|$ *is defined as* f_t^Δ*. It can be shown that* $f_t^\Delta = -f_s^\Delta$*.*

This problem can be then formulated as a linear program.

Definition 14 (Maximum Flow Problem). *For a network* $N = (V, E, s, t, \mathbf{c})$*, the* Maximum Flow (MF) problem *is to find a flow* $\mathbf{f} \in \mathbb{R}^E$ *such that the* $|\mathbf{f}|$ *is maximum possible.*

$$\begin{aligned} \textit{maximize} \quad & |\mathbf{f}| \\ \textit{subject to} \quad & f_v^\Delta = 0 \quad (\forall v \in V - \{s, t\}) \\ & \mathbf{0} \leq \mathbf{f} \leq \mathbf{c} \end{aligned} \tag{MF}$$

1.3.2 Minimum-Cost Flow

A generalization of the maximum flow problem is the minimum-cost flow problem. This variant differs from the maximum flow problem in that each edge now has an associated transportation cost along with its capacity. Additionally, each vertex is associated with a demand, indicating how much commodity needs to be transported or how much of the given commodity it supplies to the network. The objective is to find the flow of minimum cost that satisfies the demand constraints.

Definition 15 (Minimum-Cost Flow Network). *Let*

- $G = (V, E)$ *be a directed (multi-)graph,*
- $\mathbf{c} \in \mathbb{R}^E$ *be a vector of* capacities*,*
- $\mathbf{w} \in \mathbb{R}^E$ *be a vector of edge* weights *or* costs*,*
- $\mathbf{d} \in \mathbb{R}^V$ *be a vector of vertex* demands *such that*

$$\sum_{v \in V} d_v = 0$$

then $(V, E, \mathbf{c}, \mathbf{w}, \mathbf{d})$ *is called* minimum-cost flow network *for graph* G*. If for* $v \in V$

- $d_v < 0$*, then we refer to* v *as* supply *node,*
- $d_v > 0$*, then we refer to* v *as* demand *node,*
- $d_v = 0$*, then we refer to* v *as* transit *node.*

Definition 16 (Minimum-Cost Flow Problem). *For a given minimum-cost flow network* $(V, E, \mathbf{c}, \mathbf{w}, \mathbf{d})$ *we define the* mimimum-cost flow (MCF) problem *as*

$$\begin{array}{ll} \textit{minimize} & \mathbf{w}^T\mathbf{f} \\ \textit{subject to} & \mathbf{f}_\triangle = \mathbf{d} \\ & \mathbf{0} \leq \mathbf{f} \leq \mathbf{c} \end{array} \tag{MCF}$$

1.3.3 Multicommodity Min-Cost Flow

The multicommodity min-cost flow (MMCF) problem is a generalization of the ordinary (MCF) problem, in which flows of a different nature (commodities) must be routed at minimal cost on a network, competing for the resources represented by the arc capacities.

The number of commodities in real-life MMCFs ranges from just a few, as in most distribution problems, to very many, such as the number of all the possible origin/destination pairs in some telecommunication models.

In a very general form, the problem can be defined as follows:

Definition 17 (Multicommodity Min-Cost Flow (MMCF) Problem). *Let*

- $G = (V, E)$ *be a directed (multi-)graph;*
- $\mathcal{K}$ *be set of commodities;*

- *$N^\kappa = (V, E, \mathbf{c}^\kappa, \mathbf{w}^\kappa, \mathbf{d}^\kappa)$ be a minimum-cost flow network for commodity $\kappa \in \mathcal{K}$;*
- *$\mathbf{u} \in \mathbb{R}^E$ is the upper bound on arc over all the commodities.*

then the Multicommodity Min-Cost Flow Problem *is defined as*

$$\begin{aligned} \textit{minimize} \quad & \sum_{\kappa \in \mathcal{K}} (\mathbf{w}^\kappa)^T \mathbf{f}^\kappa \\ \textit{subject to} \quad & \mathbf{f}^\kappa_\Delta = \mathbf{d}^\kappa \quad \forall\, \kappa \in \mathcal{K} \\ & \sum_\kappa \mathbf{f}^\kappa \le \mathbf{u} \\ & \mathbf{0} \le \mathbf{f}^\kappa \le \mathbf{c}^\kappa \qquad \forall\, \kappa \in \mathcal{K} \end{aligned} \tag{MMCF}$$

Although MMCF is a structured linear program, standard LP techniques often fail to be efficient enough in practice, and several specialized algorithms have been proposed for its solution [14].

1.4 The Simplex Algorithm

In 1947 G. B. Dantzig designed the *simplex method* for solving linear programming formulations of U.S. Air Force planning problems. It has been added to the list as one of the top ten algorithms with the greatest influence on the development and practice of science and engineering in the 20th century [15].

Many efforts have been made since Dantzig's initial proposed algorithm in order to enhance the performance.

Why use the simplex method?

The choice of a specific algorithm for solving linear programming problems depends on various factors, such as the size and structure of the problem, the desired accuracy, and the available computing resources. While both the simplex method and interior point methods (see Appendix A.1.2) are effective approaches for solving linear programming problems, there are some situations where the simplex method may be preferred over interior point methods.

One advantage of the simplex method is that it is conceptually simpler and easier to implement than interior point methods, especially for small to medium-sized problems. The simplex method works by iteratively moving along the edges of the feasible region until an optimal solution is reached, and its geometric interpretation makes it easy to understand and visualize.

Unfortunately, the simplex method has the disadvantage that its running time is not guaranteed to be polynomial. In the worst case, the number of iterations needed to solve a linear program can grow exponentially with the size of the problem as shown by [16]. On the other hand, on real world problems the simplex algorithm behaves surprisingly well and the number of iterations scales linearly with the number of constraints.

1.4.1 Simplex Algorithm Idea

In this section, we introduce two variants of the simplex algorithm: the standard simplex and the revised simplex algorithm. While both algorithms share the same underlying principle and conceptually execute identical steps to find the optimal solution, they diverge significantly in their computational approaches. Notably, the standard simplex often becomes computationally infeasible for large, sparse problems commonly encountered in real-world applications. The revised simplex method is able to address this limitation.

The following sections will offer a fairly standard description of these algorithms and our objective is to recapitulate the core ideas for the reader. However, to fully comprehend the challenges associated with implementing these methods in practice, and to be able to modify the source code of current simplex algorithm implementations, a more in-depth understanding is necessary. To this end, we have provided a more comprehensive explanation of the simplex method in Appendix A, where readers can acquire a deeper insight of the algorithm's intricacies.

The idea of the simplex method is one of successive improvements. We aim to traverse through the vertices 8 of the polyhedron that correspond to basic feasible solutions. Given optimal conditions, this process will converge to the optimal solution in a finite number of steps, except under certain circumstances which we will elaborate upon later.

Definition 18 (Basic feasible solution)**.** *A feasible solution* $\mathbf{x}$ *of* (LP-CSF) *is called* basic *if there exists an m-element set $B \subseteq [n]$ such that*

- *the submatrix A_B is nonsingular.*
- $x_j = 0$ *for all* $j \notin B$

The set B is called a basis *of the LP. We will also sometimes denote the* basis matrix *A_B by B. The set of nonbasic column indices is denoted by $N = [n] - B$. The basis might also be presented as a pair (B, N). A variable x_j is called* basic *if $j \in B$ and* nonbasic *if $j \in N$.*

In the context of geometric interpretation, basic feasible solutions hold particular significance as they precisely correspond to the vertices of the linear programming problem. This relationship is encapsulated in the established theorem given below as presented in [6]:

Theorem 1 (Basic Feasible Solution and Vertices). *Let P be the polyhedron of all feasible solutions of* (LP-CSF). *Then the following two conditions for a point* $\mathbf{v} \in P$ *are equivalent:*

1. $\mathbf{v}$ *is a vertex of the polyhedron* P*,*
2. $\mathbf{v}$ *is a basic feasible solution of the linear program.*

The simplex algorithm begins with a feasible basis and uses a series of operations on a description of a problem until an optimal solution is computed. The algorithm searches for an optimal solution by moving from one adjacent feasible solution to another, along the edges of the feasible region. It also guarantees monotonicity of the objective value.

1.4.2 Standard Simplex Algorithm

In this section, we discuss the standard simplex algorithm, the original method for solving linear programming problems. While the algorithm can be expressed in two primary forms – tableau and dictionary – we will concentrate on the dictionary form, as both forms are equivalent and differ solely in data representation.

Definition 19 (Dictionary). *Given a* (LP-SF)*, we define a* dictionary *as a system of equations*

$$\begin{aligned} x_{n+i} &= b_i - \sum_{j=1}^{n} a_{ij} x_j \quad i = 1, \dots, m \\ z &= \sum_{j=1}^{n} c_j x_j \end{aligned}$$

This conversion is done by first introducing slack variables $x_{n+1}, x_{n+2}, \dots, x_{n+m}$*, putting them on the left hand side of the equations and setting* z *as the value objective function.*

The variables on left hand side are called basic *and on the right hand side* nonbasic.

The dictionary presents a system of equations with solutions that correspond to feasible solutions for the given problem. However, the dictionary cannot be entirely arbitrary, as it must be associated with a basic solution. For further details, please refer to Appendix A.3.1.

Iteration of Standard Simplex

Each iteration of the simplex method will consist of the following steps

1. choosing an *entering* nonbasic variable
2. choosing a *leaving* basic variable
3. constructing next dictionary

The computational process of constructing a new dictionary is called *pivoting*.

Pivoting constitutes the most computationally demanding step. Although it is relatively straightforward, the process involves selecting the row of a dictionary corresponding to the leaving variable and expressing the entering variable using this row and other variables in this row. Subsequently, this expression of the entering variable is substituted into all the remaining occurrences of that variable. For large values of m and n, updating the entire table can be very time-consuming. Memory issues may also arise, as even a single iteration for sparse problems can generate a dense matrix.

The following set of equations gives a conceptual idea how even a sparse set of equations can lead to a dense one. We start with the dictionary on the left and perform one pivoting step by choosing x_1 to be the entering variable and x_8 to be the leaving variable.

$$\begin{aligned} x_5 &= x_1 & x_5 &= x_2 + x_3 + x_4 - x_8 \\ x_6 &= x_1 & x_6 &= x_2 + x_3 + x_4 - x_8 \\ x_7 &= x_1 & x_7 &= x_2 + x_3 + x_4 - x_8 \\ x_8 &= -x_1 + x_2 + x_3 + x_4 & x_1 &= x_2 + x_3 + x_4 - x_8 \\ z &= x_1 & z &= x_2 + x_3 + x_4 - x_8 \end{aligned}$$

1.4.3 Revised Simplex Algorithm

Next we move to the description of the revised simplex algorithm.

Conceptually the revised simplex method follows the exact same pattern as the standard simplex method. Most notably, we will not be working with dictionaries, which we used to guide our choice of entering and leaving variables and which we updated after each iteration. Instead the only thing we vary between iterations will be the set of indices of the current basis.

We start with the following observation. For a given basis (B, N) we can write the system of equations as

$$A_B\mathbf{x}_B + A_N\mathbf{x}_N = \mathbf{b}$$

express the basic variables as

$$\mathbf{x}_B = A_B^{-1}\mathbf{b} - A_B^{-1}A_N\mathbf{x}_N$$

and the objective function as

$$z = \mathbf{c}_B^T A_B^{-1}\mathbf{b} + (\mathbf{c}_N^T - \mathbf{c}_B^T A_B^{-1} A_N)\mathbf{x}_N$$

Again, for details how to obtain this representation refer to Appendix A.4.

We shall see that each iteration of the revised simplex method requires solving two systems of linear equations.

1. Solve the system $\mathbf{y}^T A_B = \mathbf{c}_B$
2. Choose an entering column $\mathbf{a}$. This may be any column of A_N s.t. $\mathbf{y}^T\mathbf{a}$ is less than the corresponding component of $\mathbf{c}_N$. If there is no such column, then the current solution is optimal.
3. Solve $A_B\mathbf{d} = \mathbf{a}$
4. Find the largest t s.t. $\mathbf{x}_B^* - t\mathbf{d} \geq 0$. If there is no such t, then the problem is unbounded; otherwise, at least one component of $\mathbf{x}_B^* - t\mathbf{d}$ equals 0 and the corresponding variable is leaving the basis.
5. Set the value of the entering variable at t and replace the values of $\mathbf{x}_B^*$ of the basic variables by $\mathbf{x}_B^* - t\mathbf{d}$. Replace the leaving column of A_B by the entering column and in the basis heading, replace the leaving variable by the entering variable.

The main speedup comes from the end of the iteration. Where standard simplex has to do a laborious update of the whole dictionary, no such computation is needed in the revised simplex method. We simply add the entering variable and remove the leaving variable from the basis.

On large sparse problems encountered in practice, an iteration of the revised simplex method takes less time than an iteration of the standard simplex method. The revised simplex method beats the standard simplex method as soon as the number of rows exceeds about 100 [7].

1.4.4 Pivoting Rules

In this section, the fundamental principles guiding the selection of an entering and leaving variable, are described. When multiple potential options exist, a *pivoting rule* is employed to determine the entering variable.

The performance of the simplex algorithm is highly dependent on the chosen pivoting rule. The rule influences not only the number of iterations but also the path the algorithm follows while navigating the vertices and edges of the polyhedron of feasible solutions 6 for the given problem. In this section, we introduce several well-known rules used in the simplex algorithm.

Definition 20 (Dantzig Rule)**.** *For the entering variable choose a non-basic variable x_j with the largest coefficient.*

The Dantzig rule is the oldest and perhaps most often used, as it builds upon the basic idea of seeking the most substantial increase in the objective function at each iteration. However, it does not guarantee the algorithm's termination [7].

Definition 21 (Bland Rule)**.** *Select the non-basic entering variable x_j with the smallest possible index j, and the leaving variable x_k with the smallest possible index k.*

The Bland rule aims to address the shortcomings of the Dantzig rule. It can be shown that using this rule, the algorithm does not cycle [7].

Definition 22 (Largest Increase Rule)**.** *For the entering variable choose a non-basic variable x_j such that pivoting yields the largest absolute improvement in objective function.*

Although this rule guarantees the best local increase in the objective function at each iteration, it is not often used due to its computational complexity. While it often results in fewer iterations, the computational effort required may outweigh the benefits, leading to simpler rules solving the problem more quickly despite a higher iteration count.

Definition 23 (Random Rule)**.** *For the entering variable choose uniformly at random from the available candidates.*

This rule is of interest due to its unbiased property when traversing basic solutions, showing no preference for any particular path.

1.4.5 Factorization of the Basis

We briefly touch upon the principle through which the main speedup of the revised simplex method is achieved. The efficiency of the revised simplex algorithm hinges upon the implementation of its first and third steps during each iteration (see Section 1.4.3). As a general practice, the systems of equations involved are not recomputed entirely from scratch. Rather, some device is employed to facilitate the solution process, which is updated at the end of each iteration. The revised

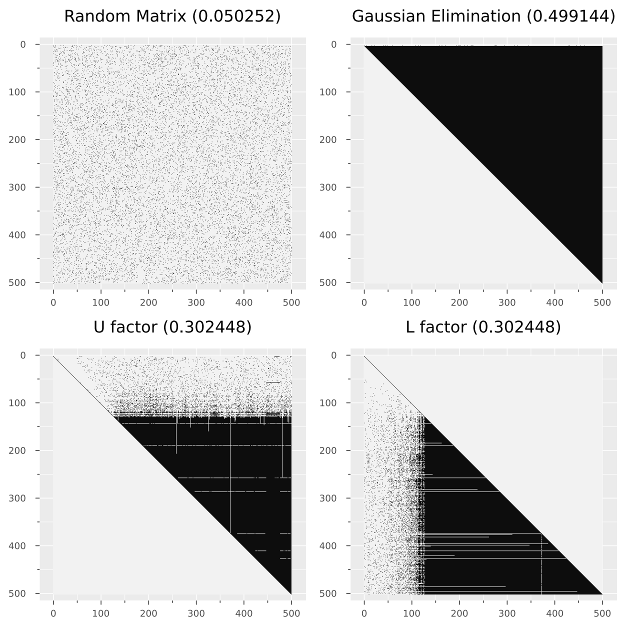


Figure 1.3 This visualization illustrates the change in the number of non-zero values in a random sparse matrix after applying Gaussian elimination and LU factorization methods. Each field in the matrix filled black represents a nonzero element of a matrix. It is important to observe that even when the initial matrix has a small proportion of non-zero values (indicated in parentheses), the quantity of non-zero values increases rapidly upon applying different matrix operations. The example of the naive Gaussian elimination method is particularly striking, as the matrix essentially becomes dense after only a few iterations.

simplex algorithm encompasses a whole class of implementations, each of which is dependant upon the selection of the device that aids in solving the two systems.

To naively solve a system of equations under general circumstances we inadvertently have to use some form of the Gaussian elimination. In large sparse problems, this is can become very slow and computationally intractable as the number of non zero entries increases during the elimination. This phenomenon is called *fill-in* and is visualized in Figure 1.3.

General idea

Let B_k denote the basis matrix obtained after k iterations of the simplex method. We wish to exploit the fact that B_k differs from the preceding B_{k-1} in only one column. Consider a fixed k and say that p-th column is the one in which the basis matrices differ. Note the p-th column is the entering column $\mathbf{a}$ of the k-th iteration appearing on the right hand side of the system $B_{k-1}\mathbf{d} = \mathbf{a}$.

By comparing the columns on both sides, it can be shown that

$$B_k = B_{k-1}E_k$$

with E_k standing for the identity matrix whose p-th column is replaced by $\mathbf{d}$. We call the matrix E_k an *eta-matrix*.

This equation is essential for the revised simplex method as no matter what device we use, its update relies on this fact and that the eta-matrix E_k is always readily available.

Let B_0 be the initial basis matrix. Successive application of this observation for k iterations yield $B_1 = B_0E_1$, $B_2 = B_0E_1E_2, \ldots$ or

$$B_k = B_0E_1E_2 \cdots E_k$$

this *eta factorization* enables us to see the two systems of equations we are required to solve as

$$(((\mathbf{y}^T B_0)E_1) \cdots)E_k = \mathbf{c}_B$$

and

$$B_0(E_1(\cdots (E_k\mathbf{d}))) = \mathbf{a}$$

Note that systems $\mathbf{v}^T E_i = \mathbf{u}$ and $E_i\mathbf{v} = \mathbf{u}$ are extremely easy to solve. What remains, is to be able to quickly solve the systems $\mathbf{v}^T B_0 = \mathbf{u}$ and $B_0\mathbf{v} = \mathbf{u}$.

The two most basic methods are to factorize this basis matrix are

- Product Form of Inverse (PFI) [17]

  Historically the PFI has been introduced shortly after the simplex method. The idea is to represent the basis inverse as a product of eta-matrices.

- LU factorization [18]

  The idea is to represent the basis matrix B as a product of an upper triangular matrix U and a lower triangular matrix L.

  Both numerical stability and the number of non-zero elements, which is created in the eta-matrices (see Figure 1.3), heavily depend on the choice of the pivot element in the elimination process.

With these techniques, we are able to reuse the representation of the basis matrix B_0 in all successive iterations to quickly solve the system of equations.

However, with the number of eta-matrices numerical inaccuracies, storage requirements and computational effort increase in every simplex iteration. Therefore, a *reinversion* or *refactorization* of the current basis matrix is triggered either when a certain number of iterations since the last inversion has passed or numerical or storage problems occur. For more details refer to [7, 19].

Other more advanced techniques for basis factorization the exploit the sparsity of the basis matrix, include:

- Bertels and Golub [20]
- Forrest and Tomlin [21]
- Suhl and Suhl [22]

1.5 Circuits and Fractionality

Definition 24 (Fractionality). Fractionality *of a number* $p/q \in \mathbb{Q}$ *is* q *if* p, q *are co-prime. The* fractionality *of a vector* $\mathbf{x} \in \mathbb{Q}^n$ *is the maximum over the fractionality over all of its entries* x_i.

Definition 25 (Circuit). *Let* $A \in \mathbb{Z}^{m \times n}$. *A vector* $\mathbf{c} \in \mathbb{Z}^n$ *is a* circuit of A *if its entries are co-prime, it satisfies* $A\mathbf{c} = \mathbf{0}$, *and it is support-minimal, that is, there does not exist any* $\mathbf{c}' \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$ *with* $A\mathbf{c}' = \mathbf{0}$ *such that supp*$(\mathbf{c}') \subset$ *supp*$(\mathbf{c})$.

Lemma 2 (Large Circuits Mean Large Fractionality[1]). *Let* $A \in \mathbb{Z}^{m \times n}$ *and* $\mathbf{c} \in \mathbb{Z}^n$ *be a circuit of* A. *Then there exist* $\mathbf{b} \in \mathbb{Z}^m, \mathbf{l}, \mathbf{u} \in \mathbb{Z}^n$ *such that the polytope* $P = \{\mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$ *defined by the constraint matrix* A *contains a vertex* $\mathbf{v}$ *with fractionality* $\|\mathbf{c}\|_\infty$.

Proof. The idea is to enforce that P only contains two vertices: the origin $\mathbf{0}$ and a scaled version of $\mathbf{c}$, namely $\mathbf{c}/\|\mathbf{c}\|_\infty$. Specifically, let $\mathbf{b} = \mathbf{0}$, and for every $i \in [n]$, if $c_i \geq 0$, let $l_i = 0$ and $u_i = 1$, otherwise let $l_i = -1$ and $u_i = 0$. Clearly $\mathbf{0} \in P$. Next observe that our setting of the bounds means that P only contains points whose support is contained in supp$(\mathbf{c})$. However, by the fact that $\mathbf{c}$ is a circuit, there are no such non-zero points in P with support strictly contained in supp$(\mathbf{c})$. Thus, the only points contained in P are of the form $\lambda\mathbf{c}$ for some $\lambda \geq 0$, and the largest λ for which $\lambda\mathbf{c} \in P$ is $\lambda = 1/\|\mathbf{c}\|_\infty$. Hence $\mathbf{c}/\|\mathbf{c}\|_\infty$ is an extreme point of P, that is, a vertex. It remains to note that at least some entries of $\mathbf{c}/\|\mathbf{c}\|_\infty$ have $\|\mathbf{c}\|_\infty$ as their denominator because the entries of $\mathbf{c}$ are co-prime, thus, $\mathbf{c}/\|\mathbf{c}\|_\infty$ has fractionality $\|\mathbf{c}\|_\infty$. □

[1]The proof of this lemma has been provided by Mgr. Martin Koutecký, Ph.D.

Chapter 2

Implementation

Our goal is to analyze the fractionality of basic feasible solutions for various types of LP problems, with a particular focus on (MMCF). To accomplish this, we need a solver that operates with exact rational arithmetic and can access the basic solution in each iteration.

This chapter outlines the steps taken to address the problem, detailing the different technologies explored and the approaches initially considered as potential solutions but ultimately discarded, along with the reasoning behind their dismissal.

2.1 Initial Implementation and Problems

After an initial review of available linear programming solvers, several observations were made. Firstly, most solvers operate primarily with floating point numbers, which is unsurprising given that the majority of practical applications require working with real numbers. This is due to the fact that input data often represents measurements from real-world scenarios, which inherently carry a margin of error, rendering exact arithmetic impractical. Modern CPUs are also optimized for floating point operations, offering performance comparable to integer arithmetic.

Moreover, existing solvers do not readily grant access to the internal state of the simplex algorithm, which is reasonable since most applications solely require only the final basic solution and objective function value. Exposing the internal workings of an algorithm may hinder flexibility in improving, extending, or modifying it, and may also impose considerable maintenance requirements on the library's authors, making it potentially impractical if there is insufficient demand for such functionality.

Consequently, the preliminary approach involved implementing the simplex algorithm from scratch. At the time, the authors were familiar with the standard simplex algorithm 1.4.2 and opted to proceed with it. The programming language of choice was Python due to its maturity, readability, rapid development capabilities, and available support. Given that the primary goal was to solve instances and trace parameters with ample time and computing power at hand, runtime performance was not of primary concern.

Initially, the Python `fractions` standard library was employed for rational arithmetic, as it is built on Python's arbitrary large integers, which was convenient. Data representation was handled using NumPy [23] arrays, which offer a robust and comprehensive interface for working with data ranges and slices.

However, numerous challenges soon emerged. For instance, the standard simplex algorithm is only efficient for small problems. Even for relatively small and sparse problems (with row counts in the hundreds), updating the dictionary becomes progressively slow and memory-intensive due to the fill-in issues (see Figure 1.3). This issue is worsened by the fact that Python's fraction library prioritizes ease of use over performance optimization. Additionally, since `Fractions` are not a native NumPy type, calculations must be carried out in Python, degrading performance due to frequent transitions between Python and C code and inherent Python inefficiency.

There is no reason to use the standard simplex algorithm if the problem description is not dense already. At this stage, memory consumption still remained the primary concern, leading to a naive implementation of the revised simplex method (see Section 1.4.3). Despite the change of the algorithm used, performance and memory issues persisted (as we were not utilizing the factorization of the basis at the time, which is so useful in the revised simplex method).

We tried to address this problem by the development of a more memory efficient approach to rational arithmetic and storage of the problem description than the `fractions` library. We have come up with a solution we named `FractionArray`, which was essentially implemented as a pair of NumPy arrays containing the numerators and denominators of the input matrix problem description. Nevertheless, this was not enough to address the memory problems, which persisted still as the input matrices become large very quickly. We have also realized that even the storage of sufficiently large problems in memory presents an issue.

One potential solution involved using the sparse arrays from the SciPy library [24], which are a wrapper around NumPy arrays and offer multiple sparse array types. However, incorporating our custom `FractionArray` proved challenging, leading to the development of a custom implementation `SparseFractionArray` of sparse arrays inspired by the SciPy library. A factorization of the basis using the LU update has also been implemented.

Unfortunately, arbitrary precision arithmetic was still required, and, at the time of writing, no native NumPy support for arbitrary length integers was available. Resorting to the standard Python `int` type introduced performance issues, a realization that only occurred after much of the interface and functionality had been implemented.

An attempt to rectify this involved using the C++ GNU MP library and binding it to Python, then employing Cython to transpile the code to C. Although Cython offered the appealing combination of C's speed and Python's readability, the authors' inexperience with the tool led to numerous problems, ultimately necessitating the abandonment of this approach.

Alternative options, such as implementing the project from scratch in another language like C++ or Julia, were considered. Julia, a modern dynamically compiled language with a rich ecosystem for scientific computing, combines the performance of C with the readability of Python. However, due to our previous experience and newly acquired knowledge of the many implementation details and pitfalls, it has been deemed easier to take an existing simplex implementation and base our work on an already established project.

At first, we tried to look at the SageMath [25] system. SageMath is a comprehensive, open-source mathematics software system that relies on several well-known open-source packages, such as NumPy, SciPy, Sympy [26], and others. It offers a unified, Python-based language and direct access to these packages through interfaces or wrappers.

SageMath's Python-based implementation, maturity, and reputable development team made it an attractive option. Within SageMath, the *Interactive Simplex Method* module was identified as a potential solution for the project. However, it became apparent that the module was intended for educational purposes only, resulting in significantly slower performance compared to standard LP solvers. A closer look at the source code revealed that the module did not even utilize sparse representation of arrays and used standard simplex textbook method, leading to similar issues previously faced in the project.

Recognizing that Python alone would not suffice for the project's requirements, alternative solutions were explored. The focus shifted to established solvers or solvers in languages the authors were already familiar with.

During our investigation, several options were considered for obtaining exact rational solutions of simplex. The following software packages and implementations were explored:

- `Algorithm:Simplex` [27] — A Perl module available that currently supports phase II of the simplex method, requiring a feasible solution to begin with.

- `qsopt-ex` [28, 29] — A library written in C that provides floating-point solutions that can be continuously refined until the exact final basis is found and the exact solution can be computed.

- GNU Linear Programming Toolkit (GLPK) [30] — A library for linear and mixed-integer programming that includes an exact simplex algorithm and is available under the GNU General Public License.
- `cl-rational_simplex` [31] — A Common Lisp implementation of the simplex method that uses rational arithmetic.

The GLPK library was chosen for further consideration for several reasons, including:

1. It is a mature and time-tested open-source library.
2. The library is written in the C programming language, which is known for its performance benefits.
3. It has good documentation and a comprehensive reference manual, making it easy to learn and understand.
4. The codebase is quite readable, which makes debugging and modification easier.
5. Previous experience with the library, ensuring a smoother implementation process.

Unfortunately, the GLPK library did not provide access to the basic solutions during iterations, only offering the basic solution from the last iteration. To work around this limitation, a somewhat improvised method was employed: the library allowed setting an iteration limit for the algorithm, and if the number of iterations exceeded this limit, the algorithm would stop. By setting the iteration limit to 1, accessing the last basic solution (which, in this case, would be the first one), incrementing the number of iterations by one, and running the algorithm again, the full trace of basic solutions of the algorithm could be reconstructed.

This workaround, however, was highly inefficient for two main reasons:

1. The initialization step of the algorithm had to be performed for every iteration instead of just once per run, introducing significant overhead.
2. If the number of iterations to solve the problem was k, this approach essentially required running k^2 iterations of the algorithm. This is infeasible for problems that need millions of iterations to reach an optimal solution.

Therefore, pausing the iteration of the algorithm and then continuing was not a viable option.

However, upon examining the library's source code, the authors determined that it was possible to implement the required functionality to trace the state of the simplex algorithm directly. Consequently, we proceeded to modify the GLPK library to include this tracing functionality, addressing the inefficiencies and limitations of the original workaround.

2.2 Implementation of GLPK Tracing Extension

In this section, we discuss the implementation of the extension for parameter tracing for the GLPK library. We begin with a brief overview of the internal methods used by GLPK, focusing on those associated with the exact simplex implementation. This information is not readily available in the manual or other documentation.

For guidance on working with GLPK methods, refer to [30], and for details on using our extended version, refer to Appendix B.

The standard entry point for the exact simplex algorithm in GLPK is the `glp_exact` function (see Figure 2.1). This function takes a `glp_prob lp` object, which contains the problem description (stored internally using floating-point numbers). After performing some initial checks (e.g., verifying control parameters and variable bounds), the function creates a `SSX ssx` object, which serves as the rational arithmetic counterpart of the `lp` object. GLPK uses the GNU MP [32] library for rational arithmetic. The problem is then solved by passing it to the `ssx_driver` function. After that, the solution is converted back to floating-point numbers and saved in the `lp` object before being returned after some final cleanup.

As we can see, the standard API does not provide a way for the user to access even the final basic solution of the exact simplex algorithm. Therefore, we extended this function to save the exact final solution to a user specified file. Additionally, for the purposes of this thesis, we included an option to scale the problems to make them integral by scaling them with the least common multiple (LCM) of all denominators.

Next, we examine the `ssx_driver` function (refer to Figure 2.2). This function sets up the simplex algorithm by computing the initial factorization of the basis (GLPK uses the LU decomposition with the Markowitz pivoting strategy), calculating the initial values of basic variables, and checking for feasibility. It also manages the phases of the simplex algorithm (refer to Appendix A.3.3). Apart from adjusting the parameters for tracing, no significant changes were made to this function.

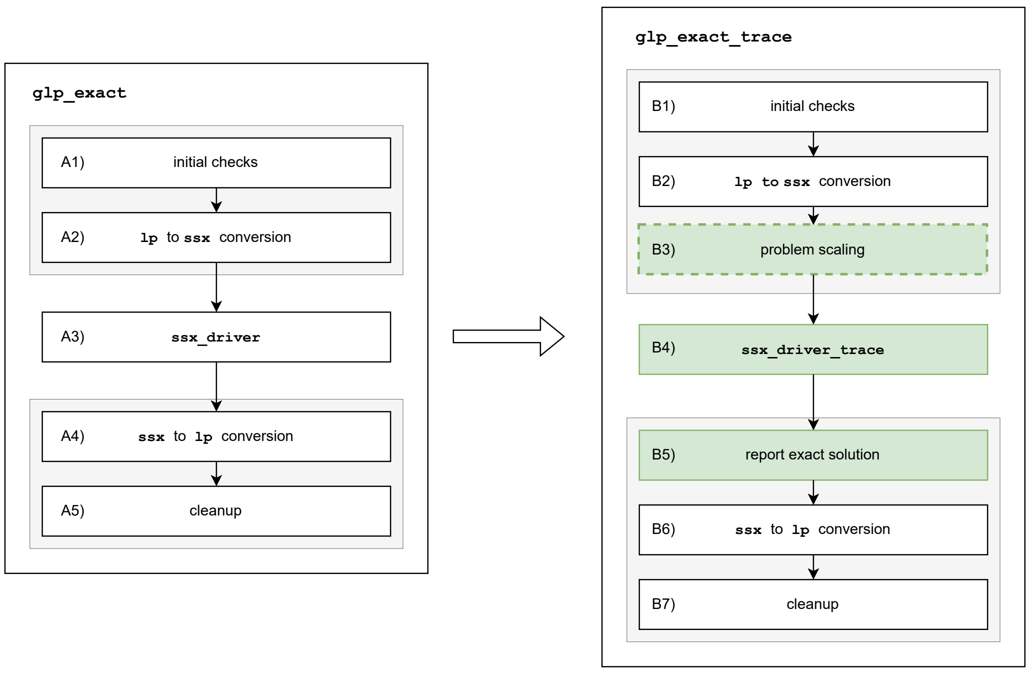


Figure 2.1 This figure illustrates the flow of the `glp_exact` function, which serves as the standard entry point for the exact simplex algorithm. The modifications we have implemented are highlighted in green. Our extensions include the optional scaling procedure and the capability to save the exact rational solution prior to its conversion to floating-point numbers.

Lastly, we investigate the `ssx_phase_I` and `ssx_phase_II` function (see Figure 2.3). These functions implement the main loop of the simplex algorithm as described in Appendix A.4.1. The majority of the modifications were implemented in these functions. We have extended the function to trace the status of the variables (whether the variable is basic or at one of its bounds, etc.), the values of all variables, and the values of the objective function in each iteration of the algorithm. At the time of writing this thesis, the exact implementation of GLPK supported only the Dantzig rule 20. We have added the Bland 21, best 22, and random 23 rules.

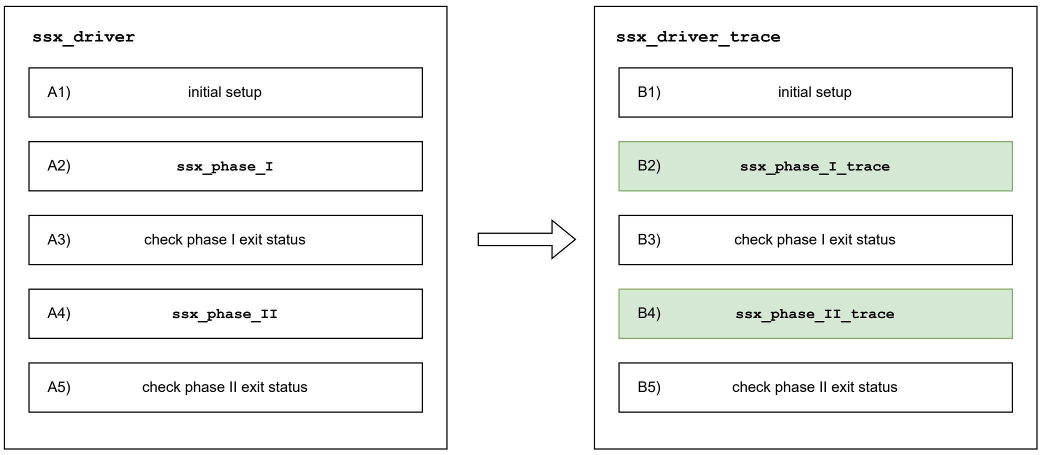
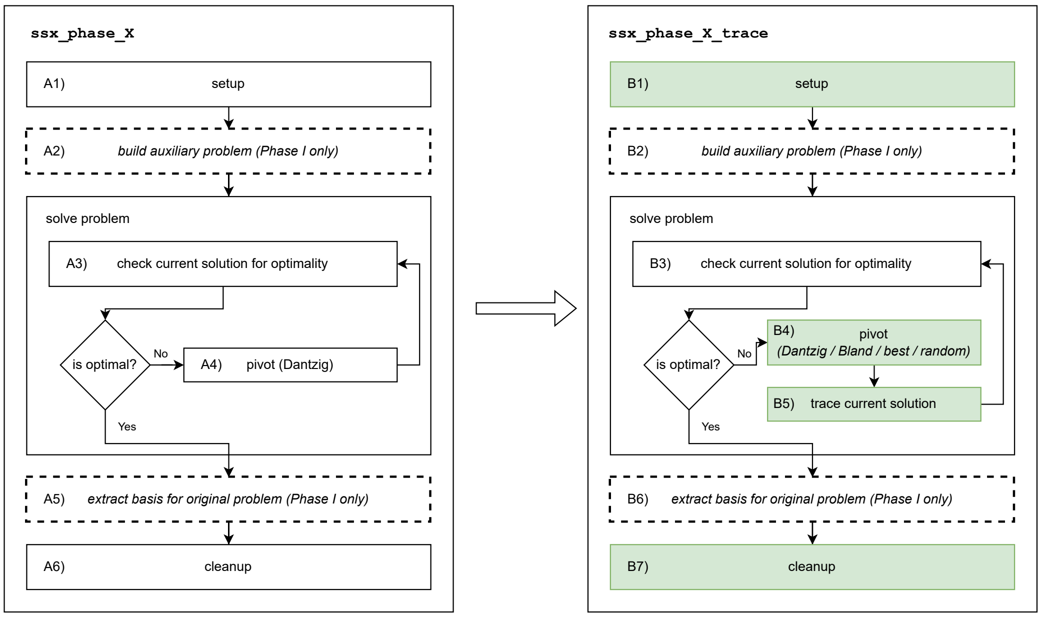


Figure 2.2 This figure illustrates the flow of the `ssx_driver` function, which serves as the primary driver of the primal simplex algorithm. The crucial part of this function was extended to incorporate necessary adjustments to support tracing (highlighted in green), while leaving the core functionality mostly unaltered.



ssx_phase_X
A1) setup
A2) build auxiliary problem (Phase I only)
solve problem
A3) check current solution for optimality
is optimal?
No
A4) pivot (Dantzig)
Yes
A5) extract basis for original problem (Phase I only)
A6) cleanup
ssx_phase_X_trace
B1) setup
B2) build auxiliary problem (Phase I only)
solve problem
B3) check current solution for optimality
is optimal?
No
B4) pivot (Dantzig / Bland / best / random)
B5) trace current solution
Yes
B6) extract basis for original problem (Phase I only)
B7) cleanup

Figure 2.3 This figure illustrates the flow of the `ssx_phase_X` functions (for Phase I and Phase II) and the modifications made to them (highlighted in green). In addition to tracing the current basic feasible solution, we have implemented additional pivoting rules. Consequently, the setup and cleanup procedures of these functions have been adjusted to accommodate these enhancements.

Chapter 3

Results and discussion

Before we commence our analysis, we provide a short overview of the problem instances used to in our analysis. For MMCF instances, we also include the format parameter used to parse them into a LP problem using the Graph library (found at [4]). This format is also used by the `convert_problem` tool we provide (see Appendix B).

MMCF Instances Description

1. **Mnetgen problems:** Mnetgen is a famous random generator of Multicommodity Min Cost Flow instances.
   Graph library option: 'm'

2. **PDS problems:** PDS instances are from a military conflict transportation model and are considered "hard".
   Graph library option: 'm'

3. **JLF problems:** JLF instances were collected for testing computational approaches to MMCF. They contain six small groups of instances.
   Graph library option: 'p'

4. **Vance problems:** Randomly generated instances on undirected graphs. Contains three different groups of instances: small and easy, considerably harder, and with fractional data.
   Graph library option: 'o'

5. **AerTranspo problems:** These instances simulate real-world airline transportation problems and have a relatively large number of commodities.
   Graph library option: 'o'

6. **Planar and Grid problems:** Two sets of test problems, planar networks and grid networks with randomly chosen origin and destination nodes. *Graph library option:* 'm'

Netlib Instances Description

The Netlib LP problems library [5] is a collection of LP problems in standardized format, consisting of objective function and constraints, along with known optimal solutions or a statement of infeasibility. It was compiled in the early 1990s to provide a set of benchmark problems for testing LP algorithms. The library contains a wide range of problems from various application areas, such as production planning, transportation, and finance. The problems vary in size, ranging from small to very large, and are available in both sparse and dense formats. The Netlib library has been widely used to evaluate and compare the performance of linear programming solvers.

As Netlib data encompasses a broader range of problem types, we use it as a baseline for comparing MMCF instance solutions. The Netlib instances serve as a benchmark for assessing the algorithm's behavior on MMCF problems.

3.1 Data Description

Throughout the execution of the simplex algorithm, we monitored the maximum fractionality of variables in the basic solution for each iteration. The denominators we work with can get a length of hundreds of digits and the number of iterations can get very large (we experienced traces for single problems in hundreds of gigabytes). To save hard drive space, we traced only the bit length of the denominators, and while this does not give the exact value for the fractionality, it provides us with a good upper bound. In other words given a basic solution $\mathbf{x}_B$ for an iteration, we tracked the value $g = \lceil \log_2 q \rceil$, where q represents the maximum fractionality over all basic solutions.

Subsequently, we computed summary statistics, such as the average and maximum fractionality of the basic solutions across all iterations. Using these summary statistics inevitably leads to the loss of some data and information concerning the fractionality during the problem-solving process. Nevertheless, these statistics still provide valuable insights into the behavior of the problems.

Our primary focus is on integral problems, or (LP) problems which have an integral right-hand side vector $\mathbf{b}$. However, the problems we work with do not adhere to this formulation. As our best effort to generate problems that do, we scaled the problems by multiplying vector $\mathbf{b}$ by the least common multiple

(LCM) of all the denominators. More specifically, for

$$\mathbf{b} = (p_1/q_1, p_2/q_2, \ldots, p_n/q_n)$$

we considered the problem

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq k \cdot \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

where $k = \mathrm{LCM}(q_1, q_2, \ldots, q_n)$. This approach effectively scales the polyhedron of feasible solutions by k.

We sourced the data from the MMCF and Netlib problems, running each problem with four pivoting rules (Dantzig 20, Bland 21, random 23, and best 22). In total, we solved approximately 1600 problems. Due to computational constraints, we excluded problems that required more than 4 GB of memory to solve.

We executed each problem on a single CPU system (since GLPK does not support parallel processing) with 4 GB of memory and a maximum runtime of one week. Consequently, the solutions of some problems did not reach completion. Notably, the "best" pivoting rule, which is the most computationally intensive, is underrepresented in the dataset.

We will further divide every analysis into two sections: Phase I and Phase II solutions. This division is meaningful as the algorithm essentially solves different problems in each phase due to differences in the objective function and additional variables (see the two-phase method A.3.3). Sometimes, it may happen that a problem may be solved to optimality in Phase I (for example, when the goal of the problem is to find a feasible solution).

There is a also notable difference in the number of instances provided by different instance types, with Mnetgen contributing the most instances. As a result, Mnetgen instances will have the greatest impact on our analysis. Additionally, we may examine specific data subsets if they appear unusual or noteworthy.

Table 3.1presents the number of problems solved for each dataset for each pivoting rule used.

Most of the plots presented will be on a log-log scale. To better understand the result presented, note that relationships of the form $y = ax^k$ appear as straight lines in a log-log graph, with the exponent k corresponding to the slope, and the coefficient a corresponding to the intercept.

First, we investigate the general trend of fractionality over the course of the algorithm's execution. In Figure 3.1, we depict the relationship between the bits of fractionality and the increasing number of iterations for a small sample of similarly sized MMCF problems. Note that, in most cases, the Dantzig rule results in significantly higher fractionality.

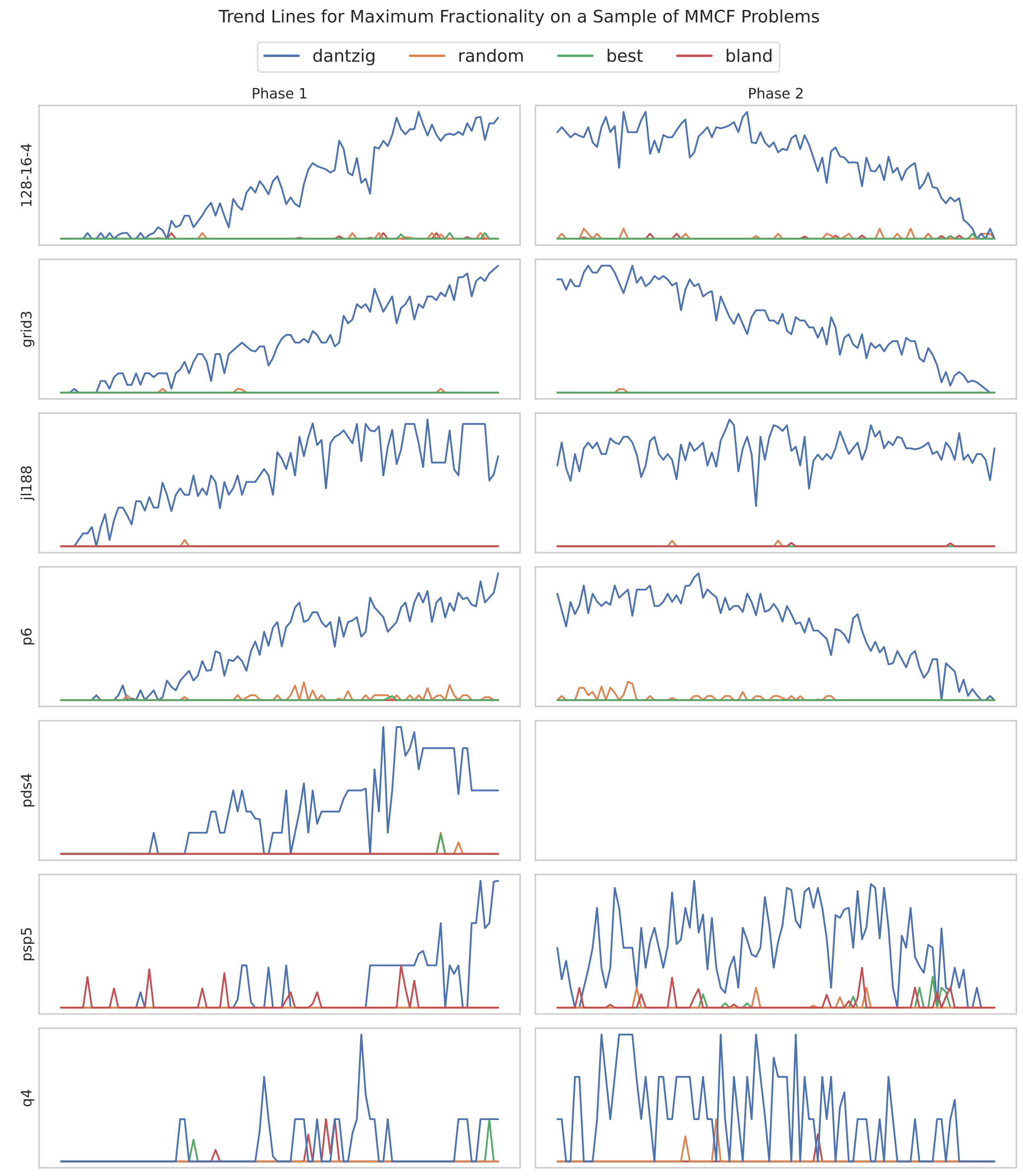


Figure 3.1 This figure illustrates the trend lines depicting the changes in fractionality of basic solutions for a selected sample of MMCF instances. The objective is to illustrate the general pattern of evolution throughout the solution process for the problem. Different colors represent various pivoting rules. It is important to note that the number of iterations is not identical for all pivoting rules even though it is depicted as such. Instead, the pivoting rules have been stretched to span the entire length of the plot for easier comparison between them. The absence of Phase II indicates that the problem was resolved entirely during Phase I.

Dataset	Pivot Rule	Instance Count	Out of Resources
mmcf	best	254	178
	bland	269	163
	dantzig	331	101
	random	382	50
netlib	best	76	17
	bland	70	23
	dantzig	88	5
	random	88	5

Table 3.1 Summary of the number of instances solved from each dataset, categorized by pivoting rule.

3.2 Fractionality Distribution Analysis

In Figure 3.2, we display the distribution of fractionality, split by pivoting rules. We include data from both MMCF and Netlib datasets in the comparison. Observe that the Dantzig rule appears to have a considerably higher mean fractionality compared to other rules for MMCF instances. Conversely, for Netlib instances, all rules seem to produce similar fractionality levels.

Next, we investigate whether some pivoting rule exhibits higher maximum fractionality in a statistically significant manner. First, we employ the Kruskal-Wallis nonparametric test to determine if all pivoting rules have the same effect on the fractionality.

Based on the results presented in Table 3.2 and considering a significance level of $\alpha = 0.05$, we can confidently reject the null hypothesis that the population medians of all pivoting rules are equal for MMCF instances in both Phase I and Phase II. This indicates that there is a significant difference between one or more pivoting rules. On the other hand, for the Netlib instances, we cannot reject the null hypothesis in neither Phase I nor Phase II, which implies that the effect of all pivoting rules is relatively similar in this case.

Subsequently, we perform a post-hoc pairwise test for multiple comparisons of mean rank sums (Dunn's test) with Bonferroni correction to identify significant differences between pairs of pivoting rules. The results in form of resulting p-values are listed in Table 3.3. The results indicate that we can distinguish the effects of the Dantzig rule from all other rules, as well as the best rule from the Bland rule in Phase I. In Phase II, we are additionally able to differentiate the effects of Bland's rule from the best rule.

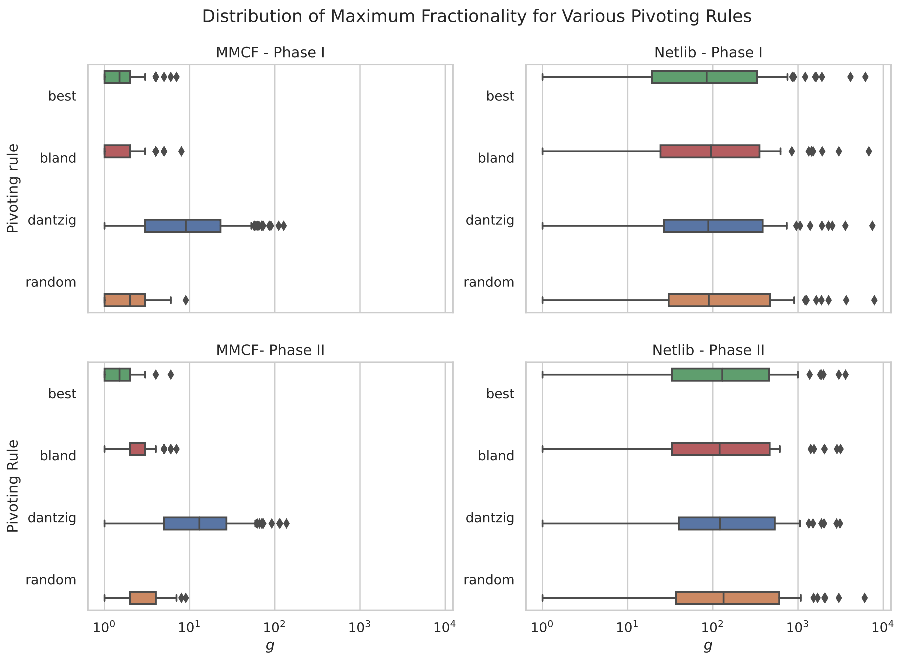


Figure 3.2 The distribution of the maximum fractionality with respect to the pivoting rule used to solve the instance. Data from Phase I and Phase II, as well as both the MMCF and Netlib datasets, are shown. We observe that for MMCF problems the Dantzig rule yields considerably higher maximum fractionality then other rules. On the other hand in Netlib problems the rules behave more similarly. Note that the x-axis is scaled logarithmically.

		p-value	Reject H_0 ($\alpha = 0.05$)
Dataset	Phase		
mmcf	1	0.000	Yes
	2	0.000	Yes
netlib	1	0.857	No
	2	0.969	No

Table 3.2 The results of the Kruskal-Wallis test, testing for null hypothesis of median of maximum fractionality among all pivoting rules being the same. We provide the resulting p-values rounded to 3 decimal places.

			best	bland	dantzig	random
Dataset	Phase	Pivot Rule				
mmcf	1	best	1.000	0.569	0.000	0.001
		bland	0.569	1.000	0.000	0.166
		dantzig	0.000	0.000	1.000	0.000
		random	0.001	0.166	0.000	1.000
	2	best	1.000	0.000	0.000	0.000
		bland	0.000	1.000	0.000	1.000
		dantzig	0.000	0.000	1.000	0.000
		random	0.000	1.000	0.000	1.000

Table 3.3 The results of a post-hoc analysis using the Dunn test. We list the resulting p-values rounded to 3 decimal places. We compare the effect on maximum fractionality for all pairs of pivoting rules.

3.3 Relationship of Iterations and Fractionality

One potential measure of a problem's complexity is the number of iterations the simplex algorithm requires to find an optimal solution. We can view the number of iterations as a kind of practical, real-world indicator of problem complexity.

We anticipate that the fractionality of a problem will increase as the number of iterations grows. This intuition is based on the assumption that, during the simplex method's execution, the problem has time to accumulate increasingly more fractional solutions.

In Figure 3.3, we illustrate the relationship between fractionality and the number of iterations performed in each phase. Additionally, we fit a regression line to indicate the general trend of change for both datasets. Observe that the fractionality of Netlib problems appears to be generally higher than that of MMCF problems. Additionally, the regression lines appear to be parallel, suggesting a similar asymptotic behavior.

3.3.1 MMCF

As visualized in Figure 3.4, different pivoting rules appear to have varying effects on the rate at which fractionality changes as the number of iterations increases. Notably, the Dantzig rule seems to exhibit the most significant effect. In Table 3.4, we present the parameters of the fitted regression lines. This information is also visualized in Figure 3.5 where we also include 95% confidence intervals. We observe that, in both Phase I and Phase II, the Dantzig rule yields the largest slope among all the rules. This finding is consistent with our previous results.

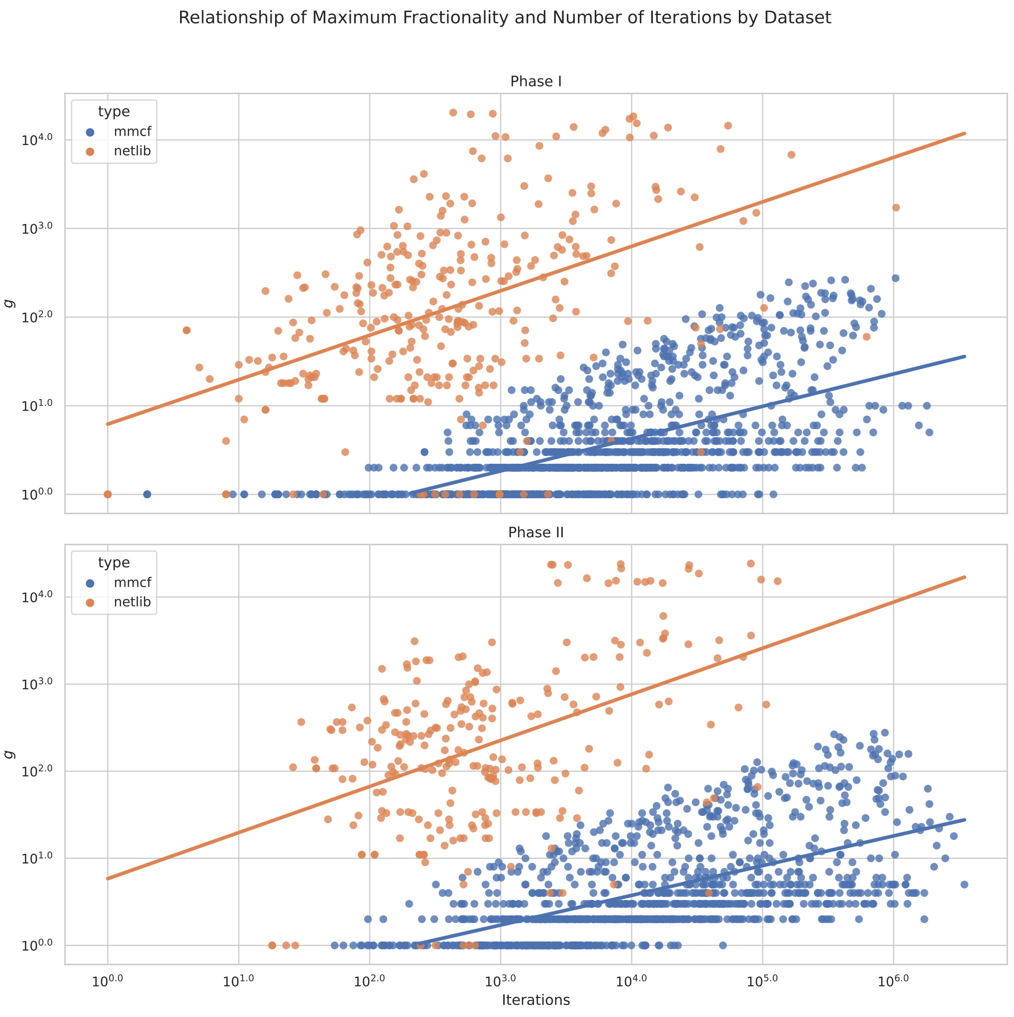


Figure 3.3 The relationship of maximum fractionality with respect to the number of iterations for different datasets. The results from all pivoting rules have been merged together. Linear functions are fitted to show the general trends of the change. Observe that the trend lines do appear to be parallel, suggesting similar asymptotic behavior across the datasets. Note that the plots are on a log-log scale.

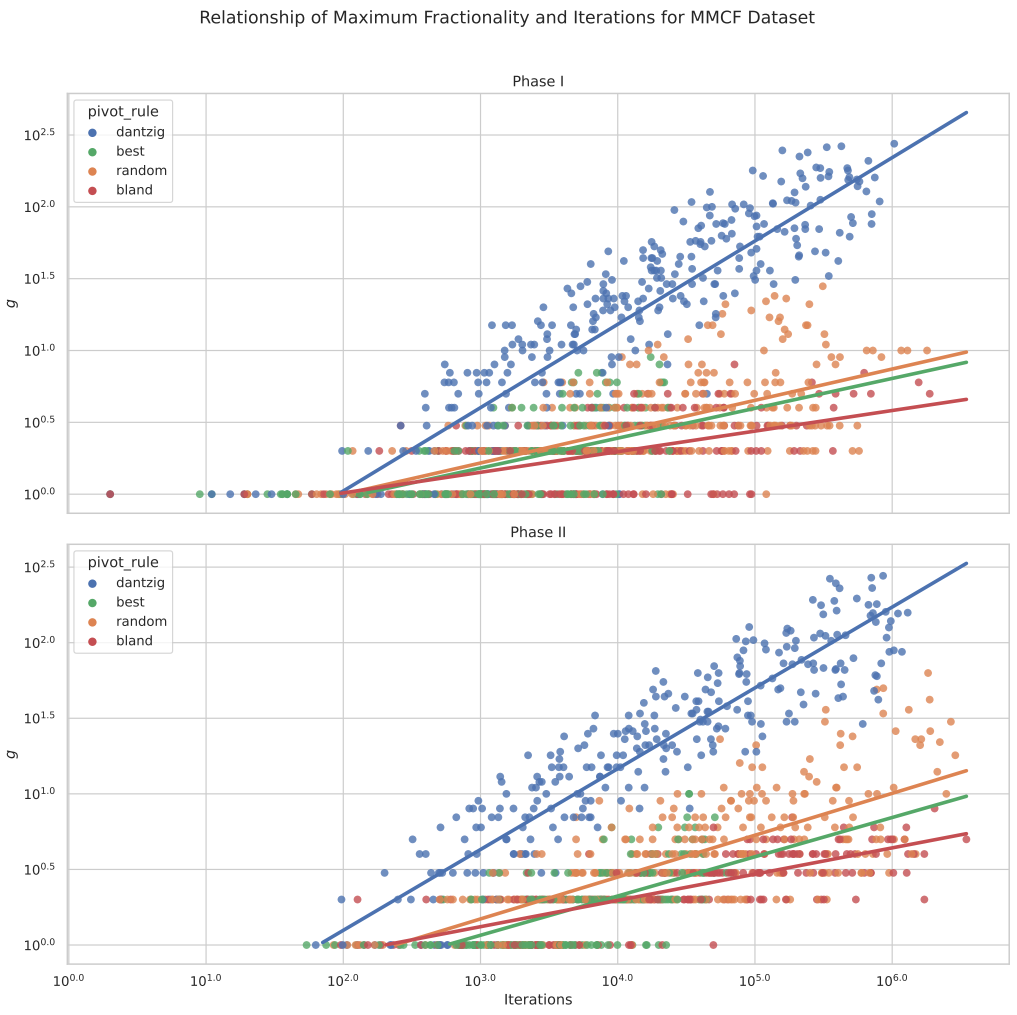


Figure 3.4 The relationship between maximum fractionality and the number of iterations for the MMCF dataset. The pivoting rules used to solve the instance are differentiated by color. Observe the general increasing trend and note that the Dantzig rule trend line has a significantly steeper slope. Note that the axes are on a log-log scale.

Phase	Pivot Rule	Intercept	Slope
1	bland	-0.278	0.143
	best	-0.441	0.208
	random	-0.437	0.218
	dantzig	-1.138	0.580
2	bland	-0.400	0.174
	best	-0.714	0.260
	random	-0.658	0.277
	dantzig	-0.972	0.535

Table 3.4 A summary of the regression results for the log-log relationship between maximum fractionality and the number of iterations as illustrated in Figure 3.4.

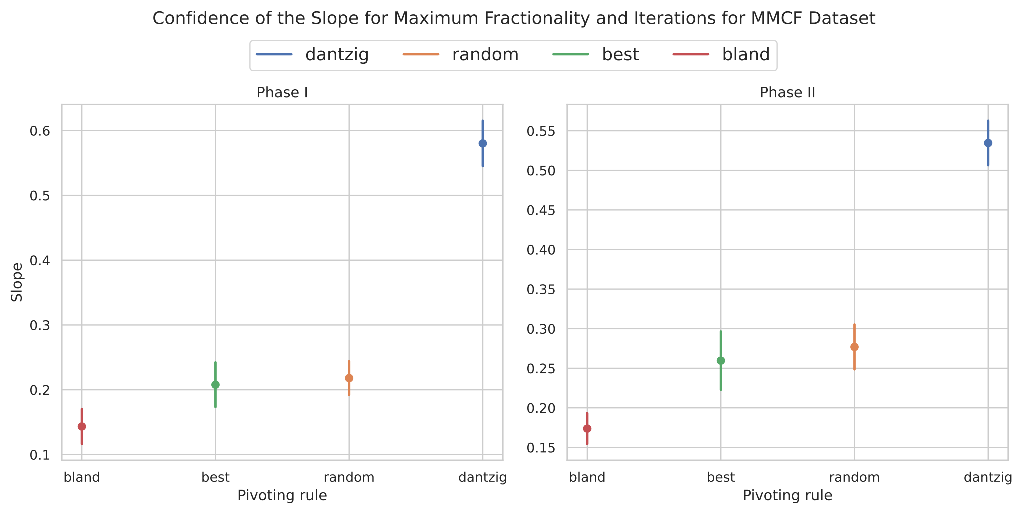


Figure 3.5 This figure visualizes the 95% confidence intervals for the slopes of the regression lines depicted in Figure 3.4.

Phase	Pivot Rule	Intercept	Slope
1	bland	0.901	0.400
	best	0.722	0.491
	random	0.682	0.542
	dantzig	0.692	0.630
2	best	1.051	0.364
	random	0.998	0.458
	bland	0.554	0.554
	dantzig	0.241	0.818

Table 3.5 A summary of the regression for the log-log relationship between maximum fractionality and the number of iterations for the Netlib instances depicted in Figure 3.6.

3.3.2 Netlib

As depicted in Figure 3.6, similar to the MMCF dataset, there appears to be a visible positive correlation between the number of iterations and fractionality in the Netlib dataset. However, the variation is now much greater, and the fit is not as strong. Additionally, there does not seem to be any significant difference between the different pivoting rules. This conclusion can also be drawn from Table 3.5. Although we can order the coefficients by magnitude, the 95% confidence intervals are much larger, as seen in Figure 3.7. Consequently, we cannot claim any pronounced difference between the rules, which is consistent with our previous results.

3.4 Relationship of Dimension and Fractionality

Next, we consider a more theoretical measure of problem complexity, namely the problem dimension. Given the way the simplex algorithm works, the problem becomes more complex to solve as the dimension increases. We will analyze the effect of increasing dimension on the maximum fractionality of the instances.

The basic solutions are found in the vertices of the polyhedron of feasible solutions. The vertices of the polyhedron are the points where at least the number of rows (dimension) of the hyperplanes bounding the polyhedron meet. One expectation to have is that the more hyperplanes we have, the more "complicated" the vertices created will be. Consequently, we would expect the fractionality to increase as the dimension of the problem increases.

Firstly, we assess how the dependence on dimension behaves between the datasets. In Figure 3.8, we again observe a positive trend in both datasets.

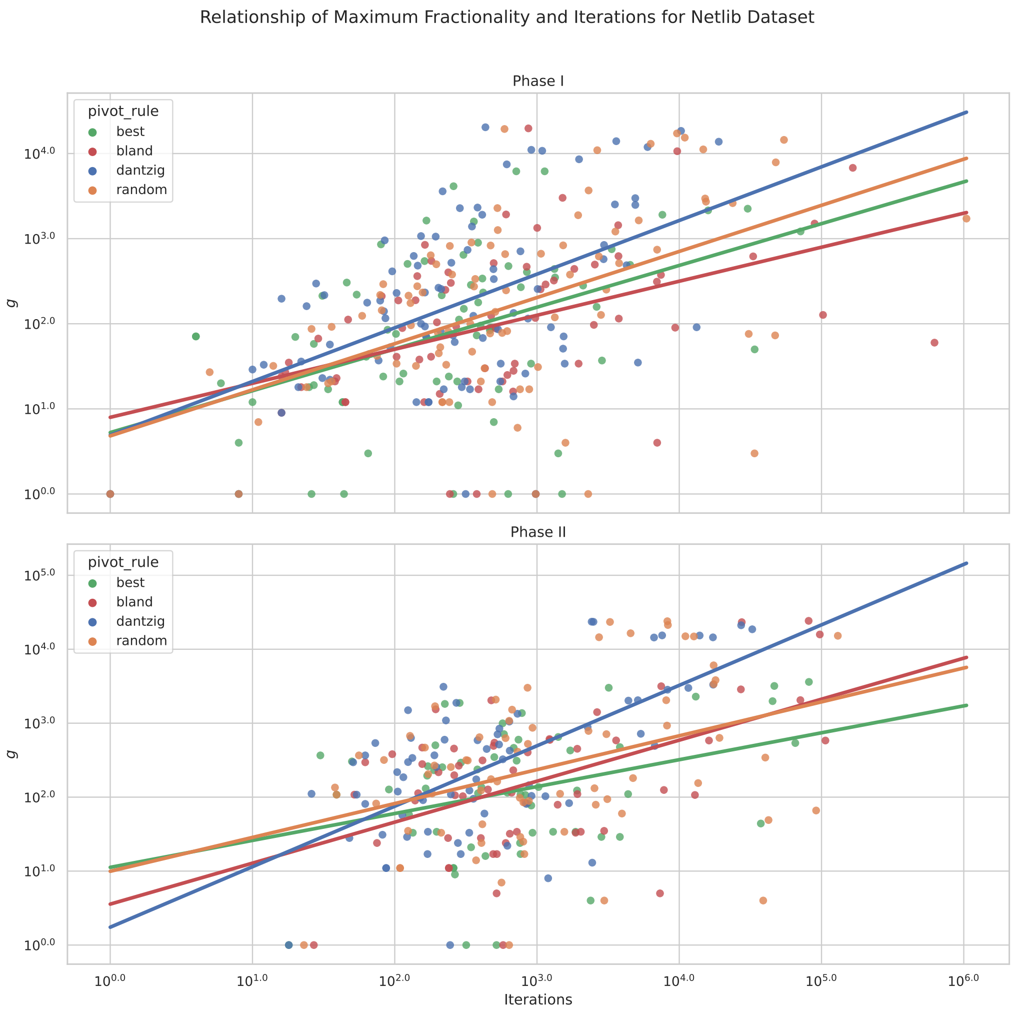


Figure 3.6 The relationship between maximum fractionality and the number of iterations for the Netlib dataset, differentiating the pivoting rules used to solve the instances by color. Note that the axes are on a log-log scale.

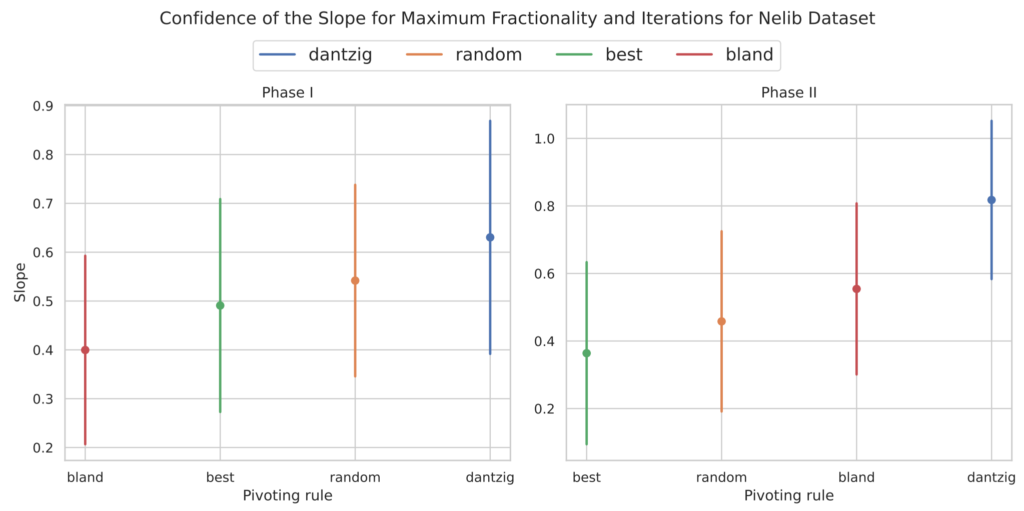


Figure 3.7 This figure visualizes the 95% confidence interval for the slope of the regression lines for Netlib instances depicted in Figure 3.6.

3.4.1 MMCF

In the MMCF dataset, there appears to be a strong positive correlation between the dimension of the problem and the fractionality as demonstrated in Figure 3.9. A similar increasing trend can be observed for all pivoting rules. The pivoting rules, once again, affect the rate of change, with the Dantzig rule having the most significant impact. This trend seems to be consistent between both Phase I and Phase II. The strength of these effects is summarized again in Table 3.6 and Figure 3.10.

Additionally, the reader might observe certain groups of instances exhibiting similar behavior and question whether these instances belong to the same type. To address this issue, in Figure 3.11 we provide a visualization of the same relationship, but this time differentiating the instance types. This visualization aids in understanding the relationship between instance types and their corresponding behaviors.

3.4.2 Netlib

In the Netlib dataset, a positive correlation can also be observed between the dimension of the problem and the fractionality, as depicted in Figure 3.12. Unlike the MMCF dataset, the slope appears to be consistent across all pivoting rules, suggesting that the relationship between the problem dimension and fractionality is not influenced by the choice of the pivoting rule in the case of the Netlib instances. These effects are again summarized in Table 3.7 and Figure 3.13.

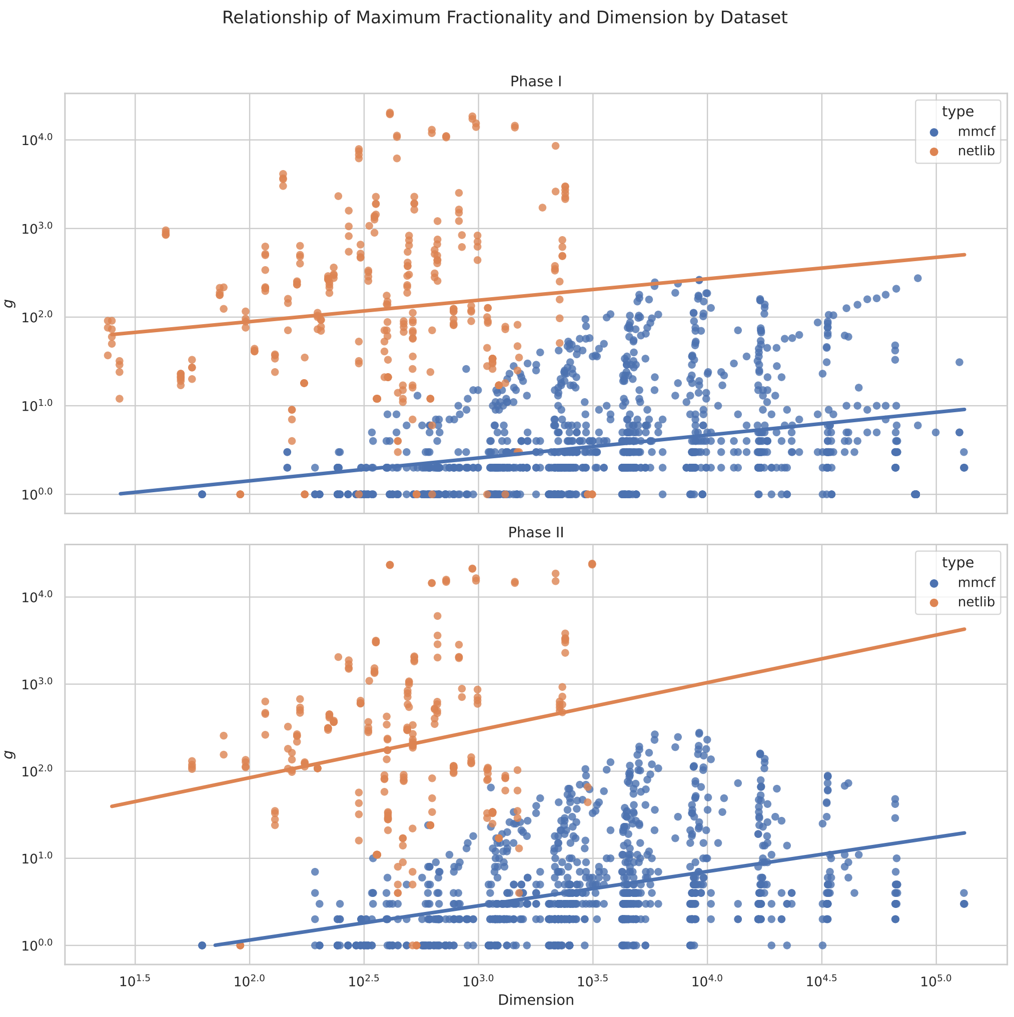


Figure 3.8 In this figure, we compare the different datasets by examining the relationship between maximum fractionality and the problem's dimension. The results from all pivoting rules have been merged together. A linear function is also fitted to demonstrate the general trend of the change. We observe a positive trend in both cases. Note that the plots are on a log-log scale.

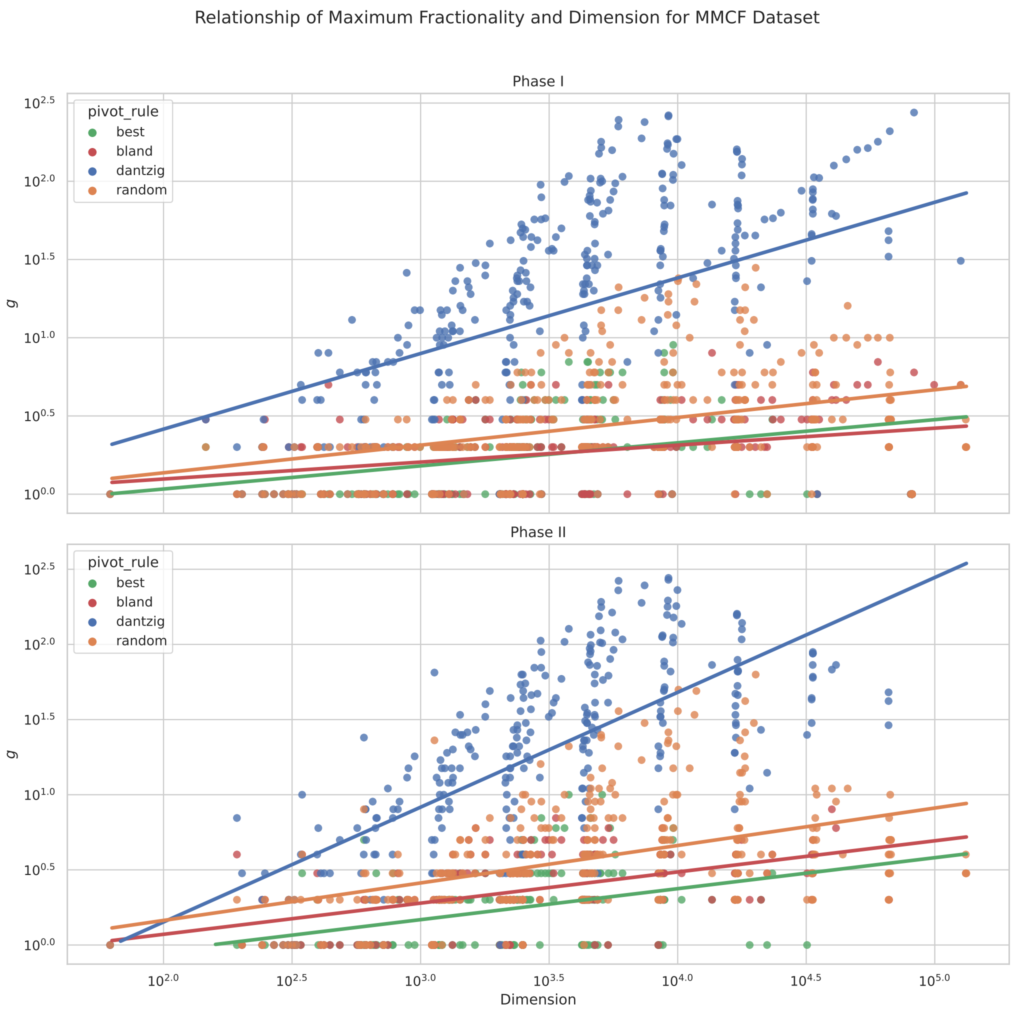


Figure 3.9 This figure illustrates the relationship between maximum fractionality and dimension for the MMCF dataset. The pivoting rule used to solve the instance is differentiated by color. For each pivoting rule we fit a trend line and observe a positive trend. Note that the axes are on a log-log scale.

Phase	Pivot Rule	Intercept	Slope
1	bland	-0.120	0.108
	best	-0.263	0.148
	random	-0.217	0.177
	dantzig	-0.551	0.483
2	best	-0.450	0.206
	bland	-0.343	0.207
	random	-0.335	0.249
	dantzig	-1.375	0.764

Table 3.6 A summary of the regression for the log-log relationship between maximum fractionality and dimension in the MMCF instances, as illustrated in Figure 3.9.

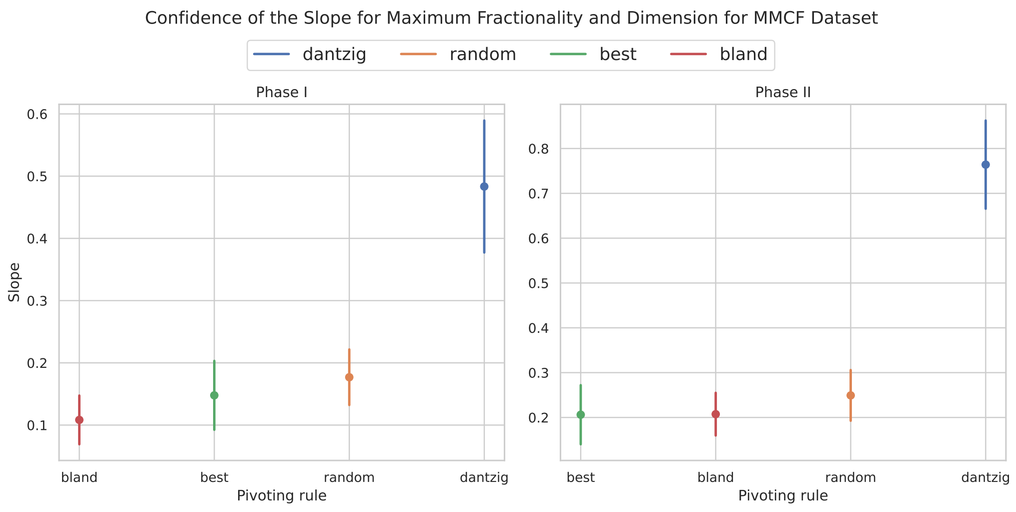


Figure 3.10 This figure visualizes the 95% confidence intervals for the slopes of the regression lines from Figure 3.9.

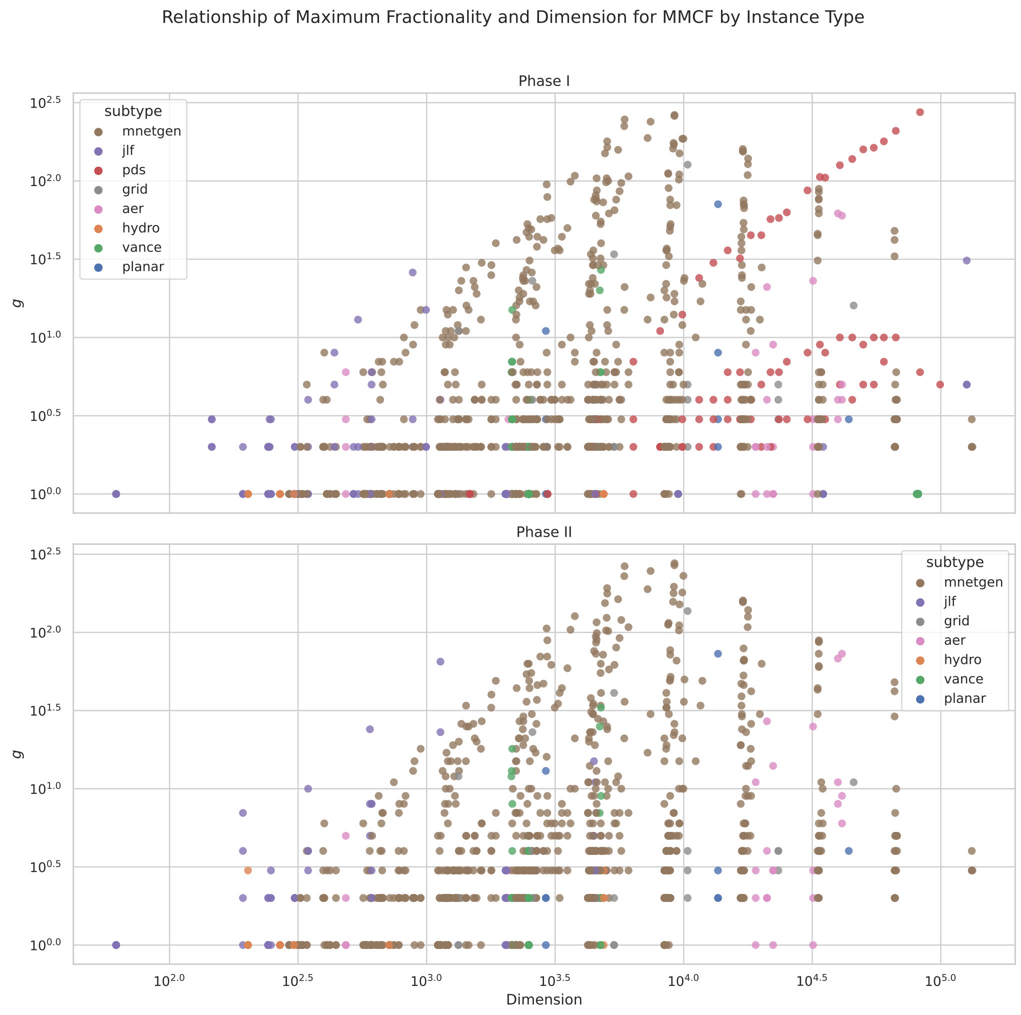


Figure 3.11 The relationship between maximum fractionality and dimension for the MMCF dataset. Different instance types are differentiated by color. Note that the axes are on a log-log scale.

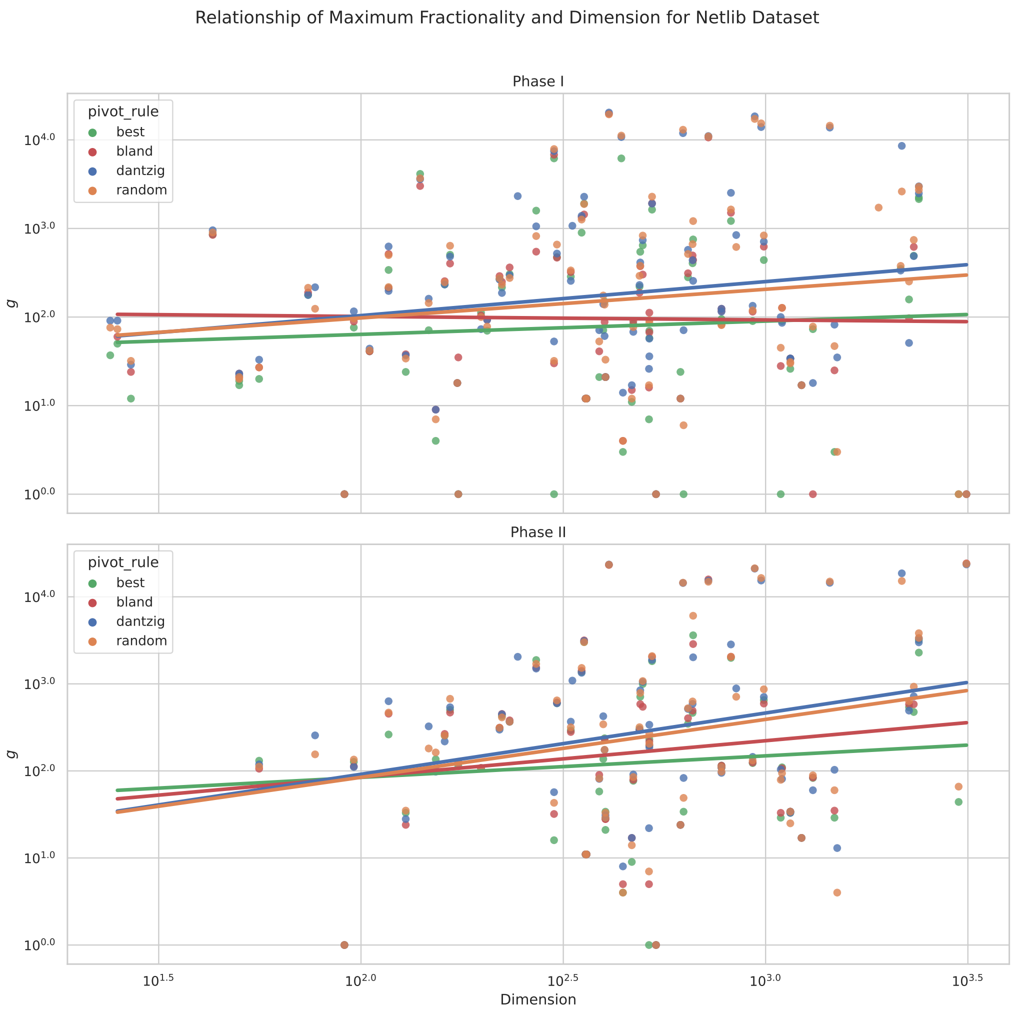


Figure 3.12 The relationship between maximum fractionality and dimension for the Netlib dataset. We differentiate pivoting rules used to solve the instance by color. We can see a positive trend for all rules used but not much of a significant difference between the rules themselves. Note that the axes are on a log-log scale.

Phase	Pivot Rule	Intercept	Slope
1	bland	2.086	-0.039
	best	1.505	0.150
	random	1.343	0.324
	dantzig	1.250	0.383
2	best	1.432	0.247
	bland	1.097	0.417
	random	0.596	0.665
	dantzig	0.555	0.704

Table 3.7 A summary of the regression for the log-log relationship between maximum fractionality and dimension for the Netlib instances, as depicted in Figure 3.9.

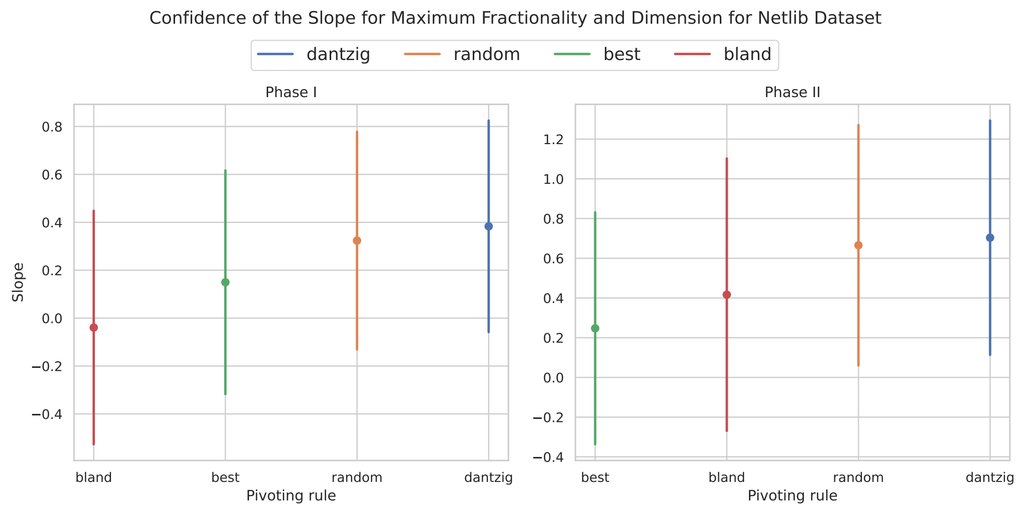


Figure 3.13 This figure visualizes the 95% confidence intervals for the slopes of the regression lines for the Netlib instances, as depicted in Figure 3.9.

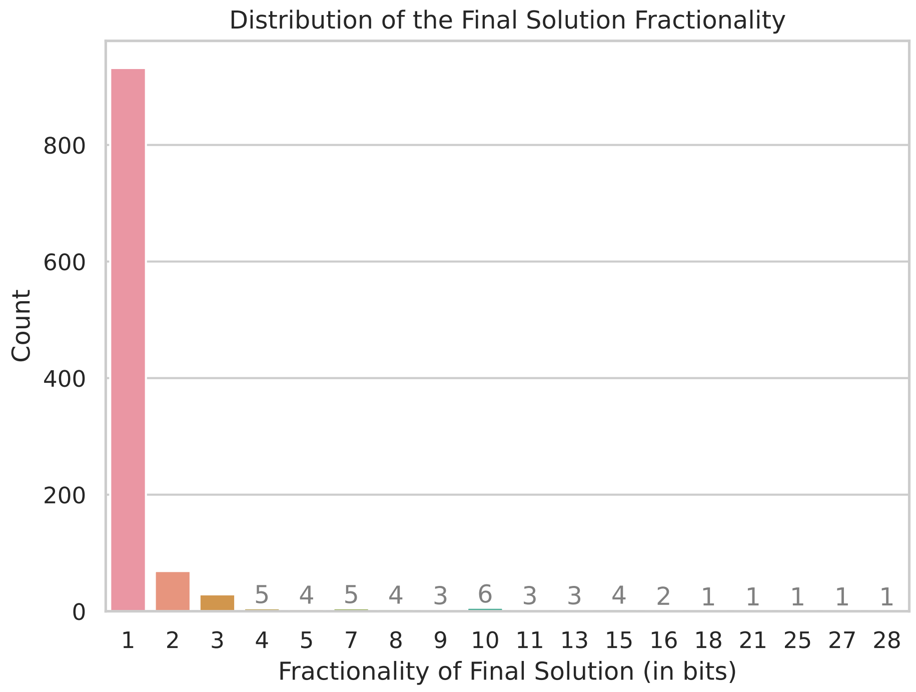


Figure 3.14 The quantity of occurrences corresponding to each achieved fractionality value of the final basic solution. For counts that are hardly discernible, a gray-colored label is provided, indicating the exact number of instances.

3.5 Fractionality of the Final Basic Solution

In Figure 3.14, we examine the distribution of fractionality for the final basic solutions. The majority of the instances analyzed exhibit low fractionality, with only a few instances demonstrating fractionality higher than a few bits. Upon further investigation, we have not identified any specific connection between these instances and a particular dataset or problem type. It is worth mentioning that in only 6 out of 336 cases, the fractionality of the final solution varied between the different pivoting rules used. In these instances, it was consistently the Dantzig rule that yielded higher fractionality compared to the other pivoting rules.

3.6 Discussion

In our analysis of the results, we observe that the maximum fractionality of the problems increases as expected with both the number of iterations and increasing dimension. Interestingly, unlike the results from the Netlib dataset (in which we were unable to statistically determine any difference in the effect of pivoting rules on fractionality), the MMCF data demonstrates a significant difference in how pivoting rules influence fractionality. More specifically, solutions produced using the Dantzig rule result in significantly more fractional solutions. This phenomenon does not seem to be present in the Netlib problems, and we do not offer any specific explanation for this observation.

To further interpret the results, we examine the linear regression analysis for the dependence on dimension in the MMCF dataset (refer to Table 3.6). Let n represent the dimension of a problem. In a log-log plot we interpret the line with intercept a and slope k as

$$\log_{10} g = k \log_{10} n + a$$

which translates to a power relationship

$$g = 10^a \cdot n^k$$

By substituting g, we obtain the upper bound on maximum fractionality

$$q \leq 1024^a \cdot 2^{(n^k)}$$

Specifically, for the Dantzig rule, we have $a = -1.375$ and $k = 0.764$, which yields

$$q \approx 7.26 \cdot 10^{-5} \cdot 2^{(n^{0.764})} = \Omega\left(2^{(n^{0.764})}\right)$$

Thus, in this case, fractionality grows exponentially with increasing dimension. Other results can be interpreted similarly.

Additionally, due to the nature of the MMCF dataset, the problems we have available often appear in groups with similar dimensions. When plotted on a graph as in Figure 3.9, such problems can be identified by data points with similar x-coordinates. If we assume that the dimension of a problem dictates its complexity in terms of maximum fractionality, one might expect the maximum fractionality to remain consistent for problems with similar dimensions. However, Figure 3.9 reveals an intriguing phenomenon where data points are grouped vertically, indicating that problems with similar dimensions exhibit varying maximum fractionality.

We hypothesize that the reason for this observation is as follows. When tracing a path through the polyhedron of basic solutions for a problem, we essentially explore only one specific path on the polyhedron and compute the maximum from that path. Nonetheless, this does not necessarily imply that we have discovered the most fractional path overall, as many paths can be taken to obtain the optimal solution. Consequently, it is expected that some problems will yield higher measured values while others may produce lower values, even if they have similar dimensions.

Finally, we did not observe any discernible trend in the fractionality of the final basic solution. As illustrated in Figure 3.14, our analysis revealed that, for the majority of the examined data, the fractionality of the final solution was quite low, with only a few exceptions. We were unable to attribute this observation to any specific pivoting rule or subset of the data.

Conclusion

In this thesis, we have presented an efficient approach for tracing basic solutions during the execution of the simplex algorithm. We developed software that can be used as a standalone application or through a library interface (see Appendix B).

In [3], Hladík discovered that MMCF matrices have large circuits ($2^{\Omega(n)}$), which implies the existence of LP problems with these matrices that possess integral inputs and high fractionality ($2^{\Omega(n)}$). However, he was unable to directly construct an MMCF problem with a highly fractional vertex. Consequently, we investigated the fractionality of MMCF problems, specifically whether the maximum fractionality is exponential or polynomial with respect to dimension. Our analysis suggests that the fractionality behaves on the order of $2^{(n^{0.764})}$ (for the Dantzig rule), which unambiguously indicates an exponential relationship.

Future work

We now highlight some possible avenues for further research that exceeded the scope of this work. These include:

- We observed that the Dantzig rule yields significantly greater fractionality in MMCF problems, a fact that does not hold for general problems from the Netlib library. It may be worthwhile to investigate why this is the case and what the exact reason is for this discrepancy.

- We noted that during the solution construction of a problem, we explore only one of many possible paths through the polyhedron of feasible solutions. Therefore, the maximum fractionality we obtain is only an estimate. To obtain a better estimate for a given problem, it might be useful to examine many more possible paths. This could be accomplished through the tools we provide by executing the problems multiple times using the random rule or by devising new pivoting rules that employ randomness.

- Our results indicate that efforts should be directed toward constructing a family of instances with fractionality exhibiting $2^{\Omega(n)}$ behavior, rather than attempting to prove that fractionality is indeed polynomial.

- It would be worthwhile to manually analyze a selected instance with high fractionality (ideally in the final solution) relative to its dimension. This analysis aims to identify and understand the underlying factors contributing to the observed fractionality in the instance. Such a targeted examination could shed light on the characteristics of MMCF problems that give rise to high fractionalities and inform the development of future algorithms and approaches.

- It may be beneficial to explore the optimal solution space once the optimal value z^* is known. By adding a constraint $\mathbf{c}^T\mathbf{x} = z^*$, the optimal solutions form a polyhedron, which could be investigated using the simplex method with a random pivoting rule. This approach would enable the examination of whether there are highly fractional optimal solutions (vertices) or provide insights into other phenomena within the optimal solution space.

Bibliography

[1] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. “Network flows”. In: (1988).

[2] Martin Mares and Tomas Valla. *Pruvodce labyrintem algoritmu*. CZ. NIC, 2017.

[3] Richard Hladík. “Combinatorial Algorithms for Flow Problems”. In: (2021).

[4] Università di Pisa. *Multicommodity (MMCF) Problems*. Accessed: 2023-05-02. 2013. URL: `http://groups.di.unipi.it/optimize/Data/MMCF.html`.

[5] Jack J. Dongarra, Eric Grosse, et al. *Netlib Repository*. Accessed: 2023-05-02. 2013. URL: `http://www.netlib.org/lp`.

[6] Jiří Matoušek and Bernd Gärtner. *Understanding and using linear programming*. Vol. 33. Springer, 2007.

[7] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming*. Macmillan, 1983.

[8] Robert Dorfman, Paul Anthony Samuelson, and Robert M Solow. *Linear programming and economic analysis*. Courier Corporation, 1987.

[9] William F Sharpe. “A linear programming approximation for the general portfolio analysis problem”. In: *Journal of Financial and Quantitative Analysis* 6.5 (1971), pp. 1263–1275.

[10] Hamdy A Taha. *Operations research: an introduction*. Pearson Education India, 2013.

[11] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[12] Marguerite Frank and Philip Wolfe. “An algorithm for quadratic programming”. In: *Naval research logistics quarterly* 3.1-2 (1956), pp. 95–110.

[13] Martin Mares. “Krajinou grafovych algoritmu”. In: ITI. 2007.

[14] Paola Cappanera and Antonio Frangioni. “Symmetric and asymmetric parallelization of a cost-decomposition algorithm for multicommodity flow problems”. In: *INFORMS Journal on Computing* 15.4 (2003), pp. 369–384.

[15] Jack Dongarra and Francis Sullivan. “Guest Editors Introduction to the top 10 algorithms”. In: *Computing in Science & Engineering* 2.01 (2000), pp. 22–23.

[16] Victor Klee and George J Minty. “How good is the simplex algorithm”. In: *Inequalities* 3.3 (1972), pp. 159–175.

[17] George B Dantzig and Wm Orchard-Hays. “The product form for the inverse in the simplex method”. In: *Mathematical Tables and Other Aids to Computation* (1954), pp. 64–67.

[18] Harry M Markowitz. “The elimination form of the inverse and its application to linear programming”. In: *Management Science* 3.3 (1957), pp. 255–269.

[19] Achim Koberstein. “The dual simplex method, techniques for a fast and stable implementation”. In: *Unpublished doctoral thesis, Universität Paderborn, Paderborn, Germany* (2005).

[20] Richard H Bartels and Gene H Golub. “The simplex method of linear programming using LU decomposition”. In: *Communications of the ACM* 12.5 (1969), pp. 266–268.

[21] John JH Forrest and John A Tomlin. “Updated triangular factors of the basis to maintain sparsity in the product form simplex method”. In: *Mathematical programming* 2.1 (1972), pp. 263–278.

[22] Uwe H Suhl and Leena M Suhl. “Computing sparse LU factorizations for large-scale linear programming bases”. In: *ORSA Journal on Computing* 2.4 (1990), pp. 325–335.

[23] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[24] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[25] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.7)*. https://www.sagemath.org. 2023.

[26] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[27] Mateu Hunter. *Algorithm::Simplex*. Accessed: 2023-05-02. 2017. URL: https://github.com/mateu/Algorithm-Simplex.

[28] Jon Lund Steffensen. *QSopt Exact*. Accessed: 2023-05-02. 2017. URL: `https://github.com/jonls/qsopt-ex`.

[29] Daniel G Espinoza. *On linear programming, integer programming and cutting planes*. Georgia Institute of Technology, 2006.

[30] Andrew Makhorin. “GNU linear programming kit-reference manual for GLPK version 4.64”. In: *Moscow Aviation Inst., Moscow, Russia, Tech. Rep* (2017).

[31] Marius Posta. *cl-rational-simplex*. Accessed: 2023-05-02. 2009. URL: `https://github.com/postamar/cl-rational-simplex`.

[32] Torbjörn Granlund and The GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Version 6.2.1. Free Software Foundation. 2020. URL: `https://gmplib.org/`.

[33] Leonid G Khachiyan. “Polynomial algorithms in linear programming”. In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (1980), pp. 53–72.

[34] Narendra Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 302–311.

[35] Margaret Wright. “The interior-point revolution in optimization: history, recent developments, and lasting consequences”. In: *Bulletin of the American mathematical society* 42.1 (2005), pp. 39–56.

Appendix A

The Simplex Algorithm

The main idea of the simplex algorithm is to begin with a certain feasible solution and use a specific sequence of operations until an optimal solution is computed. It searches for an optimal solution by moving from one adjacent feasible solution to another, along the edges of the feasible region. It also guarantees monotonicity of the objective value.

Unfortunately, the worst case complexity of the simplex method is exponential as shown in [16]. However, on real world problems the simplex behaves surprisingly well and the number of iterations scales linearly with the number of constraints.

Many efforts have been made since Dantzig's initial proposed algorithm in order to enhance the performance. A small overview of other algorithms for solving linear programs follows.

A.1 Algorithms for Solving LP

A.1.1 Ellipsoid Method

In 1978 Khachiyan proposed the first exact polynomial algorithm for LP [33], the so called *ellipsoid method.* The ellipsoid method has had a significant influence on the theory of linear programming; however, in practice, its high execution time per iteration renders it less competitive when compared to the simplex algorithm.

A.1.2 Interior Point Methods

Alternative approaches, such as interior point algorithms, were proposed to traverse the feasible region's interior. However, due to their costly execution time per iteration and potential numerical instability, these methods initially struggled to compete with the simplex algorithm in practice.

In 1984, Karmarkar proposed the first interior point method that outperformed the simplex algorithm [34]. Since then, numerous advancements have been made in both the theory and practice of interior point methods. Karmarkar's polynomial-time LP method reported solution times consistently 50 times faster than the simplex method.

This breakthrough revitalized the study of interior-point methods and barrier problems, demonstrating the possibility of creating a linear programming algorithm with polynomial complexity that could compete with the simplex method.

For a more comprehensive account of the history of interior point methods, see [35].

A.2 Simplex Method

A.2.1 Computational Form of LP

In Section 1.2, we introduced the standard form and computational standard form of the simplex algorithm. While the original simplex algorithm was developed for problems in standard form, more recent modifications utilize a different form known as the *computational form*. The computational form again fulfills further requirements, i.e., the constraint matrix has to have full row rank and only equality constraints are allowed.

To put (LP-SF) into a computational form, inequality constraints are transformed into equality constraints by introducing *slack-variables*. In LP-systems this is usually done in a standardized way by adding a complete identity matrix to the constraint matrix [29].

Definition 26 (Computational Form)**.** *A problem is in* computational form *if it has the form*

$$\begin{aligned} \textit{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u} \end{aligned} \tag{LP-CF}$$

with $A \in \mathbb{R}^{m\times n}$ having full row rank, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ being respectively the lower and upper bounds on decision variables.

One advantage of the computational form is the separation of variable bounds from constraints. The simplex algorithm is then adapted to handle these bounds separately, incurring minimal additional overhead. The reduced size of the constraint matrix allows for faster computation. Although we will not delve into the precise details of this modification, the necessary steps are described in [7].

We will continue to use the standard form (LP-SF) and its computational variant (LP-CSF) (with individual bounds assumed to be incorporated in the constraint matrix) for explanations of the algorithms. However, it is worth noting that these forms are seldom employed in practice, as they hold limited significance for practical LP systems.

A.3 Standard Simplex Algorithm

In this section, we will present a description of the standard simplex algorithm. The standard simplex method can be presented in two main flavours, namely the tableau and dictionary forms. Although both forms are equivalent and differ only in their data representation, we will focus on the dictionary form in our explanation.

The idea of the simplex method (in general) is one of successive improvements. We aim to traverse through the vertices of the polyhedron that correspond to basic feasible solutions. Given optimal conditions, this process will converge to the optimal solution in a finite number of steps, except under certain circumstances which we will elaborate upon later.

A.3.1 Initialization

Suppose we are given a problem in standard form. An initial step of the method involves introducing slack variables for converting each inequality into an equality. In other words, we cast the problem from the standard form to the computational form. However, we will approach this representation from a slightly different perspective by introducing the concept of a dictionary.

Definition 27 (Dictionary)**.** *Given a* (LP-SF)*, we define a* dictionary *as a system of equations*

$$\begin{aligned} x_{n+i} &= b_i - \sum_{j=1}^{n} a_{ij}x_j \qquad i = 1, \dots, m \\ z &= \sum_{j=1}^{n} c_j x_j \end{aligned}$$

Where $x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are newly introduced slack variables *and z being a variable denoting as the objective function.*

The variables on left hand side are called basic *and on the right hand side* nonbasic.

Additionally, we require that

1. *every solution of the set of equations comprising a dictionary must also be a solution of the original problem and vice versa,*
2. *the equations of every dictionary must express m of the variables and the objective function z in terms of the remaining n variables.*

We will also require the property that this system of linear equations is associated with one particular feasible solution, which we get by requiring one additional property

3. *Setting the nonbasic variables at zero and evaluating the left-hand side variables we arrive at a feasible solution.*

A dictionary with property 3. is called a basic dictionary*.*

This dictionary as we described it is a *basic feasible dictionary* if and only if all the coefficients b_i are non-negative. We call such problems as problems with *feasible origin*. For the moment we will consider only problems with feasible origin and we will handle initialization for other kinds of problems later.

We do not care for the values of the slack variables provided they remain non-negative. It can be observed that the set of feasible solutions for the original decision variables does not change by introducing the slack variables.

When provided with a dictionary, obtaining the next feasible solution is relatively straightforward. The general approach involves selecting a non-basic variable and adjusting its value while still satisfying all constraints to produce another feasible solution. The dictionary provides guidance in selecting the variable x_j and determining the appropriate change in its value.

After this step however, the challenge lies in discovering a feasible solution that improves the objective value even further. The previous iteration's ease was rooted in the presence of not only a feasible solution but also the dictionary. Our aim will be to construct a new feasible solution with an accompanying basic feasible dictionary.

A.3.2 Iteration

Each iteration of the simplex method will consist of the following steps

1. choosing an *entering* nonbasic variable x_j
2. choosing a *leaving* basic variable x_k
3. constructing the next dictionary

The computational process of constructing a new dictionary is called *pivoting*.

Choosing an entering variable

The choice of the entering variable will be motivated by our desire to improve the value of the objective function z.

Definition 28 (Entering variable)**.** *The* entering variable *is a nonbasic variable x_j with a positive coefficient c_j in the last row of the dictionary.*

The idea is that the increase of x_j will necessarily bring about the increase of the objective function.

It is important to note that, unless there is only one variable with a positive coefficient, this rule is ambiguous. If multiple candidates are available, any of them may potentially serve as the entering variable. The rules used used in such cases are discussed in Section 1.4.4.

Now only the case when no candidate is available for the entering variable remains. Actually, in this situation, we have reached the optimal solution (and the feasible solution is provided by the current basic feasible dictionary) [7].

Choosing a leaving variable

The way we can increase the value our chosen entering variable is not arbitrary. The non-negativity of the basic variables imposes an upper bound on the increment of the entering variable.

Definition 29 (Leaving variable)**.** *The* leaving variable *is a basic variable whose non-negativity imposes the most stringent upper bound on the increase of the entering variable.*

Let t be the current value of the entering variable x_j. To determine the leaving variable we increase the value t from zero to some positive value maintaining the values of the remaining nonbasic variables at their zero levels and adjusting the values of the basic variables so as to preserve the non-negativity of the basic variables as presented in the current dictionary. As t increases the value of the basic variable changes until one hits its zero bound and this one will be the leaving variable. Therefore we have to know the largest possible value of t.

We perform this choice by using the so called *minimum ratio test.* We choose k such that $-\frac{b_k}{a_{kl}}$ is minimal from all $-\frac{b_i}{a_{il}}$ for $i \in B$, $a_{it} < 0$. The basic variable x_k will then be the leaving variable.

Note that the choice of k above is again ambiguous and the choice is again further determined through the use of a pivoting rule (see Section 1.4.4).

Similarly, the issue of not having a suitable choice for the variable arises again. In this instance, however, it signifies that there is no bound on the increase of the entering variable. Consequently, there are no bounds on the value of the objective function, rendering the problem unbounded (see 10).

Pivoting

The final step of an iteration is the step of pivoting or creating a new dictionary. We will update our basis by adding in the entering variable and removing the leaving variable.

This step is conceptually simple. We choose the row of the dictionary corresponding to the leaving variable. Using this row we now express the entering variable using the other variables. We then substitute it to all remaining occurrences. Or more formally we add $\frac{b_k}{a_{kl}} \cdot a_{il}$ multiple of the k-th row to i-th row.

By our definition of the dictionary the leaving variable must have the value of zero, otherwise the resulting dictionary will not be basic feasible. This requirement is however guaranteed by the way we chose it the leaving variable sing the minumum ratio test.

A.3.3 Problems that can arise

Degeneracy

During the course of the algorithm it may happen that we encounter a point where we will have a basic solution with one or more basic variables at zero. Such solutions and the corresponding linear programs are called *degenerate*. A situation that forces a degenerate pivot step may occur only for a linear program in which several feasible bases correspond to a single basic feasible solution. This has aside effect that an increase of objective function can be zero. Simplex iterations that do not change the basic solution are also called degenerate. Degeneracy is a rule rather than an exception in LP problems, it has been said that nearly all problems from practical applications yield degenerate steps at some stage of the simplex method [7].

Termination

Next, we present the most troublesome possible case of degenerate iterations. It is possible for the simplex algorithm to go through an endless sequence of iterations without finding an optimal solution. The good news is that is is possible to avoid cycling by using an appropriate pivoting rule (see 21). Moreover cycling is a rare phenomenon. It is also the only way for the algorithm to fail to terminate (for details see [7]).

Two-phase simplex

So far we have discussed only problems with a feasible origin. However, we often encounter problems that do not have this form. The question then becomes how to find an initial basic feasible solution. Fortunately we will not have to devise a specialized algorithm for this and we will be able to use the simplex method itself to fix this. The idea is to apply the simplex method to an *auxiliary problem*, whose solution will give us a feasible solution for our original problem.

Definition 30 (Auxiliary Problem)**.** *For a problem in computational standard form 5*

$$\begin{aligned} \textit{minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \textit{subject to} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^n$*,* $A \in \mathbb{R}^{m \times n}$*,* $\mathbf{b} \in \mathbb{R}^m$*. Assume* $\mathbf{b} \geq \mathbf{0}$*. We define the* auxiliary problem *as*

$$\begin{aligned} \textit{minimize} \quad & \sum_{i=1}^{m} x_{n+i} \\ \textit{subject to} \quad & \overline{A}\overline{\mathbf{x}} = \mathbf{b} \\ & \overline{\mathbf{x}} \geq \mathbf{0} \end{aligned} \tag{LP-AUX}$$

where $\overline{A} = (A \mid I_m)$ *and* $\overline{\mathbf{x}} = (\mathbf{x} \mid x_{n+1}, \dots, x_{n+m})^T$*.*

The auxiliary problem has an apparent basic feasible solution

$$\overline{\mathbf{x}} = (0, \dots, 0, b_1, \dots, b_m) \in \mathbb{R}^{n+m}$$

We also observe that if

$$\overline{\mathbf{x}} = (x_1, \dots, x_n, 0, \dots, 0) \in \mathbb{R}^{n+m}$$

is a basic solution obtained for (LP-AUX), it is also a basic solution for the original problem. Consequently, when solving the auxiliary problem results in an objective value of zero, we have found a basic solution for the original problem. This outcome is valid only if none of the auxiliary variables are part of the basis. However, if such variables are present in the basis, they can be removed through a process called "driving out," which is elaborated upon in [7].

To summarize, when solving a general LP problem, we will proceed in two phases:

- **Phase I**

  We apply the simplex algorithm to an auxiliary LP problem in order to find an initial feasible solution for the original problem.

- **Phase II**

  We solve the original problem using the basis found in Phase I.

This approached is called the *two-phase simplex method.*

A.4 Revised Simplex Method

Next we move to the description of the revised simplex method. Conceptually the revised simplex method follows the exact same pattern as the standard simplex method. In each iteration we first choose the entering variable, then the leaving variable and finally update the current basic feasible solution. However, as will become apparent there are some fundamental differences that we are able to exploit when solving real world problems.

Most notably, we will not be working with dictionaries, which we used to guide our choice of entering and leaving variables and which we updated after each iteration. Instead, the only thing we will have available will be the information about which variables are currently in the basis.

We start with the following observation that will be essential in our description of the algorithm

For given basis (B, N) we write the system

$$A\mathbf{x} = \mathbf{b}$$

as

$$A_B\mathbf{x}_B + A_N\mathbf{x}_N = \mathbf{b}$$

explicitly differentiating between basic and nonbasic variables. Since the matrix A_B is non-singular (see [7]), we are able to express the basic variables as

$$\mathbf{x}_B = A_B^{-1}\mathbf{b} - A_B^{-1}A_N\mathbf{x}_N \tag{A.1}$$

We do the same with the objective function $z = \mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N$. Substituting $\mathbf{x}_B$ we obtain

$$z = \mathbf{c}_B^T A_B^{-1}\mathbf{b} + (\mathbf{c}_N^T - \mathbf{c}_B^T A_B^{-1}A_N)\mathbf{x}_N \tag{A.2}$$

This, essentially allows us to compute the dictionary we have described in the standard simplex method for an arbitrary given basis (B, N).

Note that as $\mathbf{x}_N = 0$, we can express the value of the basic variables for the current basis as

$$\mathbf{x}_B = A_B^{-1}\mathbf{b} = B^{-1}\mathbf{b}$$

We shall see that each iteration of the revised simplex method requires solving two systems of linear equations. Typically, these systems are not solved form scratch, instead, some device facilitating their solution is used and updated in each iteration (refer to Section 1.4.5).

Each iteration of the revised simplex method may or may not take less than that the corresponding iteration of the standard simplex method which depends on the particular implementation of the revised simplex method and the nature of the data [7].

A.4.1 Iteration

Choosing the entering variable

In the standard simplex method, the objective function was readily available in the last row of the dictionary. All we had to do was chose a variable with a positive coefficient (according to some pivoting rule). However, it is not so in the revised simplex method.

As in standard simplex method, an entering variable may be any nonbasic variable with a positive coefficient. As seen in (A.2), the coefficients of nonbasic variables can be computed as

$$\mathbf{c}_N - \mathbf{c}_B^T B^{-1} A_N$$

We compute this vector by substituting

$$\mathbf{y} = \mathbf{c}_B^T B^{-1}$$

and firstly solving the system

$$\mathbf{y}^T B = \mathbf{c}_B$$

and finally we calculate

$$\mathbf{c}_N - \mathbf{y}^T A_N$$

We then proceed to choose the entering variable just as in the standard simplex method.

Note that it is possible that individual components of $\mathbf{c}_N - \mathbf{y}^T A_N$ may be calculated individually. For nonbasic variable x_j corresponding to some coefficient c_j of $\mathbf{c}_N$ and column $\mathbf{a}$ of A_N the relevant component equals $c_j - \mathbf{y}^T\mathbf{a}$.

The entering variable may be any nonbasic variable x_j for which $\mathbf{y}^T\mathbf{a} < c_j$. This fact is used in many efficient implementations.

The corresponding column $\mathbf{a}$ of A is called the *entering column.*

Choosing the leaving variable

In the standard simplex method we did the choosing of the leaving variable using the minimal ratio test with information readily available in the dictionary.

Using the representation from Equation (A.1) we have

$$\mathbf{x}_B = \mathbf{x}'_B - B^{-1}A_N\mathbf{x}_N$$

and so $\mathbf{x}_B$ changes from $\mathbf{x}'_B$ to $\mathbf{x}'_B - t\mathbf{d}$ with $\mathbf{d}$ standing for the column of $B^{-1}A$ that corresponds to the entering variable. Note $\mathbf{d} = B^{-1}\mathbf{a}$ where $\mathbf{a}$ is the entering column.

We find the largest t such that $\mathbf{x}^*_B - t\mathbf{d} \geq 0$ (using the minumum ratio test), if there is no such t, the problem is unbounded. Let k be the row index that attains the bound of the minumum ratio test.

Basis update

So far it might seem that this method required too much unnecessary laborious computation. We might ask whether this was worth it or not. The main speedup comes from the end of the iteration. Where standard simplex has to do a laborious update of the whole dictionary, no such computation is needed in the revised simplex method. We simply add the entering variable and remove the leaving variable from the basis:

$$B := (B - \{x_k\}) \cup \{x_j\}$$

Appendix B

Software Usage

B.1 Library Interface

This chapter presents the usage of the `glpk-simplex-trace` library, a fork of the GLPK (GNU Linear Programming Kit) Version 4.64 library. This library has been extended to enable the tracing of basic solutions of the exact simplex algorithm provided with the original distribution. The library builds on the standard GLPK API; for more details, please refer to the GLPK Manual [30].

The source code complete with installation instructions can be found on *Github* at dkubek/glpk-simplex-trace.

The main method for running the solver with tracing is the `glp_exact_trace` method. This method takes as arguments a `glp_prob` object containing the problem, a `glp_smcp` object of standard control parameters for GLPK, and a `glp_ssxtrace` tracer object. The tracer is then initialized using the `glp_create_ssxtrace` method, by supplying it with a `glp_stmcp` object containing tracer control parameters.

The tracer control parameters are as follows:

- **`store_mem`**: Controls whether to store the solution in memory (Phase II *only*).

  Possible values are:

  - `GLP_STORE_TRACE_MEM_ON` – save solution in memory (default);
  - `GLP_STORE_TRACE_MEM_OFF` – do not save solution in memory.

- **`objective_trace`**: Controls whether to store the objective values.

  Possible values are:

  - `GLP_OBJECTIVE_TRACE_ON` – turn tracing of objective values ON (default);
  - `GLP_OBJECTIVE_TRACE_OFF` – turn tracing of objective values OFF.

- **`basis_trace`**: Controls whether to store the values of basic variables.

  Possible values are:

  - `GLP_BASIS_TRACE_ON` – turn tracing of basic variable values ON (default);
  - `GLP_BASIS_TRACE_OFF` – turn tracing of basic variable values OFF.

- **`status_trace`**: Controls whether to store the status of variables. Possible values are:

  - `GLP_STATUS_TRACE_ON` – turn tracing of basic variable values ON (default);
  - `GLP_STATUS_TRACE_OFF` – turn tracing of basic variable values OFF.

- **`pivot_rule`**: Pivoting rule to use.

  Possible values are:

  - `GLP_TRACE_PIVOT_BEST` – use the best (largest increase) rule 22;
  - `GLP_TRACE_PIVOT_BLAND` – use Bland's rule 21;
  - `GLP_TRACE_PIVOT_DANTZIG` – use Dantzig's (textbook) rule 20 (default);
  - `GLP_TRACE_PIVOT_RANDOM` – use random rule 23.

- *`fractionality_bits_trace`*: Controls whether to only store fractionality bits (intended exclusively for purposes of this thesis)

  Possible values are:

  - `GLP_TRACE_BITS_ONLY_ON` – turn tracing of bits ON;
  - `GLP_TRACE_BITS_ONLY_OFF` – turn tracing of bits OFF (default)

- *`scale`*: Scale the problem by multiplying the right-hand side by LCM of denominators (intended primarily for purposes of this thesis).

  Possible values are:

  - `GLP_TRACE_SCALE_ON` – turn scaling ON;

  - GLP_TRACE_SCALE_OFF – turn scaling OFF (default).

- **X_file_basename** (where X is one of objective, status, variable, info) specifies the file where values are to be stored. If left empty, the values will not be stored. All values are empty by default.

Note that *do not recommend* using the store_memory option, as the results can get very large very quickly even for relatively small problems. It only supports the phase II and storing the solution to a file is better in almost all cases.

For a small example of how to use the library refer to Listing 1. For a more complete example refer to dkubek/glpsol_trace.

Listing 1 Example of using the library.

```c
void solve(char* filename) {
    glp_prob *P;
    P = glp_create_prob();

    // Use default Simplex Method Control Parameters (SMCP)
    glp_smcp params = NULL;

    // Read the~problem from a~file in~an~LP format
    glp_read_lp(P, NULL, filename);

    // Construct initial basis
    glp_adv_basis(P, 0);

    // Use Simplex Trace Method Control Parameters to~default
    glp_stmcp trace_params = GLP_DEFAULT_STMCP;

    // Create a~trace object for~storing information
    glp_ssxtrace* trace = glp_create_ssxtrace(&trace_params);

    // Solve using trace
    glp_exact_trace(P, &params, trace);

    // Cleanup
    glp_ssxtrace_free(trace);
    glp_delete_prob(P);
}
```

B.2 CLI Tool Usage

glpsol_trace

```
glpsol_trace <MODEL_FILENAME> [OPTION...]
```

The available options are:

- `-lp`: Specifies the input type of `MODEL_FILENAME` to be the CPLEX LP format.
- `-mps`: Specifies the input type of `MODEL_FILENAME` to be (FREE) MPS format.
- `-pivot <RULE>`: Specifies the pivoting rule to use. Available options for `RULE` are `dantzig`, `bland`, `best`, and `random`.
- `-info-file <FILENAME>`: Specifies the file where to store problem information and final solution.
- `-obj-file <FILENAME>`: Specifies the file where to store the trace of objective values.
- `-status-file <FILENAME>`: Specifies the file where to store the trace of variable status.
- `-var-file <FILENAME>`: Specifies the file where to store the trace of variable values.
- `-h`, `-help`: Print usage.

Additional options used mostly for internal purposes are:

- `-info`: Print problem info and exit.
- `-bits-only`: Print the number of fractionality bits only.
- `-scale`: Scale the problem to have integral RHS.

`convert_problem`

```
convert_problem [OPTIONS]
```

The available options are:

- `-i`, `-input <file>`: Specify input filename.
- `-f`, `-format <type>`: Specify format type (s/c/p/o/d/u/m) for the Graph library.
- `-o`, `-output <file>`: Specify output filename.
- `-t`, `-out-format <fmt>`: Specify output format `mps/json` (default: mps).
- `-h`, `-help`: Display this help message and exit.

One additional option used for internal purposes is:

- `-c`, `-check-cost`: Enable checking unified cost. This checks whether every arc has the same cost for all commodities.

The source code complete with installation instructions and example usage can be found on *Github* at dkubek/glpsol_trace.

B.3 Output Format

In this section, we describe the format of the output files. The output format of a rational number is either an integer or two integers separated by a slash "/". The status of a variable can be one of five possible values:

- `1` – basic variable
- `2` – non-basic variable on lower bound
- `3` – non-basic variable on upper bound
- `4` – non-basic free (unbounded) variable
- `5` – non-basic fixed variable

A variable can be either *logical* or *structural* (for more information on structural and logical variables, refer to the GLPK Manual [30]).

B.3.1 Info File Format

The info file output is split into two sections. The first section is right at the beginning of the file, with each line containing a “key : value” pair. The possible values listed are:

- **rows**: The number of rows of the problem matrix. Also represents the number of logical variables.
- **cols**: The number of columns of the problem matrix. Also represents the number of structural variables.
- **nonzeros**: The number of nonzero values in the problem matrix.
- **iterations**: The number of iterations needed to solve the problem.
- **status**: The exit status of the solution.
- **objective**: The final objective value as a rational.
- **scale** (optional): Integer by which the problem has been scaled (if the “–scale” option has been used). Dividing the objective value by this number yields the objective value of the original problem.

The second part is delimited by:

```
--- BEGIN VARIABLES (TYPE-NAME-STATUS-VALUE) ---
...
--- END VARIABLES ---
```

Each line in this section contains three values separated by space in the order type, name, status, value. The meaning of the field is as follows:

- **type** denotes whether the variable is logical (`l`) or structural (`s`);
- **name** denotes the name of the variable;
- **status** denotes the final status of the variable;
- **value** denotes the final value of the variable.

B.3.2 Var File Format

The i-th line of the variable file contains a list of rational numbers separated by space, representing the values of the variables in the i-th iteration. The order of variables is as defined in the info file.

B.3.3 Obj File Format

The i-th line contains a single rational number representing the objective value in the i-th iteration.

B.3.4 Status File Format

The i-th line of the variable file contains a list of integers separated by space, representing the status of the variables in the i-th iteration. The order of variables is as defined in the info file.