TOPICS IN EXACT PRECISION MATHEMATICAL PROGRAMMING

A Thesis
Presented to
The Academic Faculty

by

Daniel E. Steffy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Algorithms, Combinatorics and Optimization

Georgia Institute of Technology
May 2011

TOPICS IN EXACT PRECISION MATHEMATICAL PROGRAMMING

Approved by:

Professor William J. Cook, Advisor
School of Industrial and Systems Engineering
Georgia Institute of Technology

Professor George Nemhauser
School of Industrial and Systems Engineering
Georgia Institute of Technology

Professor Shabbir Ahmed
School of Industrial and Systems Engineering
Georgia Institute of Technology

Professor Zonghao Gu
School of Industrial and Systems Engineering
Georgia Institute of Technology

Professor Robin Thomas
School of Mathematics
Georgia Institute of Technology

Professor Santanu Dey, Reader
School of Industrial and Systems Engineering
Georgia Institute of Technology

Date Approved: December 6, 2010

Dedicated in memory of William J. Steffy

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor, Professor William J. Cook, for his guidance during my years at Georgia Tech. He has been exceptionally generous with his time and support and I have learned a great deal from him. I would also like to thank Huizhu Wang and my family for their support through my time at Georgia Tech.

Over these years I have had the chance to collaborate with many outstanding researchers. I have enjoyed working together with Kati Wolter on several aspects of exact mixed-integer programming. Kati's work in developing an exact version of SCIP has enabled us to extensively test some of the ideas developed in this dissertation. Daniel Espinoza has also been helpful with many things; he has provided assistance in working with the QSopt_ex software and implemented an interface for QSopt_ex so it could be used within SCIP. Sanjeeb Dash has helped in using components of QSopt_ex and QSopt and provided a rectangular LU factorization code. I also had the chance to work together with Ken Chen and Ricardo Fukasawa on the exact separation of two-row cuts. David Applegate helped with several things, including locating and fixing an elusive bug in QSopt_ex. Tobias Achterberg has provided assistance and suggestions related to SCIP. Thorsten Koch and Martin Grötschel supported my visit to Zuse Institute Berlin in the summer of 2009. I would also like to thank Professor Eddie Cheng of Oakland University with whom I have collaborated on many projects, starting from my undergraduate years. Taking courses and working together with Eddie inspired me to choose Optimization as my research focus.

During my time at Georgia Tech I have been surrounded by many incredible people who have been my study partners and friends. To name a few: Doug Altner, Ashlea Bennett, Yao-Hsuan Chen, So Yeon Chun, Daniel Dadush, Ricardo Fukasawa, Marcos Goycoolea, Sam Greenberg, Chris Healey, Helder Inacio, Fatma Kılınç-Karzan, Pete Petersen, Feng Qiu, Norbert Remenyi, Chris Rockett, Dylan Shepardson, Andrew Smith, Kael Stilp, Alejandro Toriello, Juan Pablo Vielma and many others. Thank you for making my time in Atlanta

so enjoyable.

Additionally, I would like to thank all of the professors I have had at Georgia Tech; especially the members of my committee from whom I have taken many excellent courses and had many discussions. I also appreciate the helpful comments provided by my committee members that improved this thesis.

Finally, I would like to recognize several sources of financial support: NSF Grant CMMI-0726370, ONR Grant N00014-08-1-1104, the Georgia Tech President's Fellowship and the ARCS Foundation Fellowship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The field of Mathematical Programming offers a range of tools and algorithms to solve optimization problems. Software based on these ideas is used in many application areas to solve real world decision problems. However, few available software packages provide any guarantee of correct answers or certification of results, despite their widespread use. Computations are typically performed entirely in floating-point arithmetic where attempts are made to satisfy feasibility and optimality conditions approximately instead of exactly.

The focus of this dissertation is the advancement of theory and computation related to exact precision mathematical programming. We will specifically consider linear programming (LP) and mixed-integer programming (MIP) problems. Implementing software which relies entirely on exact arithmetic could give prohibitive slowdown compared to inexact methods so we make use of *hybrid symbolic-numeric* computation. In this paradigm algorithms are designed to use fast inexact arithmetic for many operations and then correct or verify the results using safe or exact computation.

In Chapter 1 we will further motivate the topic of this dissertation by describing different types of errors that can result from the use of floating-point arithmetic. We will also describe a range of applications where numerical errors are unacceptable.

In Chapter 2 we present new results in symbolic linear algebra. We study output-sensitive algorithms to solve rational linear systems of equations. These algorithms have the same worst case performance as conventional methods but are guaranteed to terminate early when the exact rational solution has a small representation. These techniques were motivated by experiments performed on linear systems arising when solving exact LPs.

In Chapter 3 we investigate solving very sparse rational linear systems of equations which arise as a bottleneck in solving exact LPs. A computational study is performed comparing four different techniques for exact rational system solving on a wide range of

instances arising from applied problems.

Chapter 4 describes a new algorithm to compute valid LP bounds by correcting approximate solutions. This algorithm is designed to be used in the MIP setting and accelerates bound computations by reusing structural information throughout a branch-and-bound tree. We show this method to be more general than some algorithms previously described for this purpose. Computational experiments are performed to demonstrate its effectiveness.

In Chapter 5 we discuss future directions for research in exact mathematical programming including challenges, opportunities and new application areas.

CHAPTER I

INTRODUCTION

A Linear Programming (LP) problem is an optimization problem of the form

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \geq b \\ & x \geq 0 \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and x is a vector of decision variables. A Mixed-Integer Programming (MIP) problem has the same form with the added integrality constraints $x_i \in \mathbb{Z}\ \forall i \in I$ where $I \subseteq \{1, \ldots, n\}$. A problem with all decision variables required to be integral is an Integer Programming (IP) problem.

The frameworks of LP, IP and MIP are powerful and flexible tools for modeling and solving decision problems. The most widely used algorithm for solving LP problems is the *simplex* method, which was developed by George Dantzig [44]. Although it performs well in practice, no version of the simplex method has been shown to run in polynomial time. For detailed coverage of the simplex algorithm see [36, 135]. Polynomial-time algorithms for LP including the ellipsoid method and interior-point methods have also been developed, details and history for these algorithms can be found in [130, 143].

The simplex method solves LPs by pivoting between adjacent vertices of the polyhedron describing the feasible region of the LP until reaching an optimal vertex. At each iteration of the simplex method, its position is represented as an *LP basis*. A feasible basis gives a structural description of a vertex of the polyhedron as a system of linear equations. An optimal LP basis also provides a description of a dual solution which gives a certificate of optimality. The simplex method is also an effective tool for re-optimizing LPs. After making certain types of changes to an LP, additional pivots can be performed to identify the new optimal solution. This characteristic is useful when solving IP and MIP problems

where solution methods often require the result of many closely related LPs.

The most commonly used framework for solving IP and MIP problems is the LP based branch-and-bound algorithm in combination with cutting planes. Classic references for these algorithms include [116, 130, 142]. We will give a brief description of these techniques for IP, although many of the ideas apply, with some differences, for MIP as well. For an IP problem, its *LP relaxation* is the LP that results after relaxing the integrality constraints. The first step of the branch-and-bound algorithm is solving the LP relaxation of the IP, which can be done using the simplex method. If the optimal solution is integral, then it satisfies the relaxed integrality constraints and is an optimal solution for the original IP. Otherwise *branching* is performed; the problem is split into two new problems which exclude the fractional solution but whose union contain all of the integer solutions. This procedure is applied recursively, possibly generating many subproblems. At any stage of the algorithm, subproblems with objective value exceeding the objective value of the best known integer solution are discarded. After termination, this algorithm produces an optimal solution and a tree describing the subproblems and LP results, which give a certificate of optimality. In the worst case the branch-and-bound algorithm may enumerate all feasible solutions, but in practice it often performs much better than enumeration.

Cutting planes are another tool used to solve IPs. A cutting plane is an additional linear constraint that can be added to the problem description without changing the feasible region, i.e. it is satisfied by all integer solutions to the problem. The pure cutting plane algorithm for IP is an iterative algorithm that starts by solving the LP relaxation. If a fractional solution is found then one or more valid cutting planes are identified and added to the problem, cutting off the fractional solution. This process is repeated iteratively. Some versions of the pure cutting plane algorithm will theoretically identify the optimal solution after a finite number of iterations, but other versions of the algorithm will not always converge.

Figure 1 gives a small example of a fractional solution being excluded by branching or adding a cutting plane; each method eliminates the fractional solution without eliminating

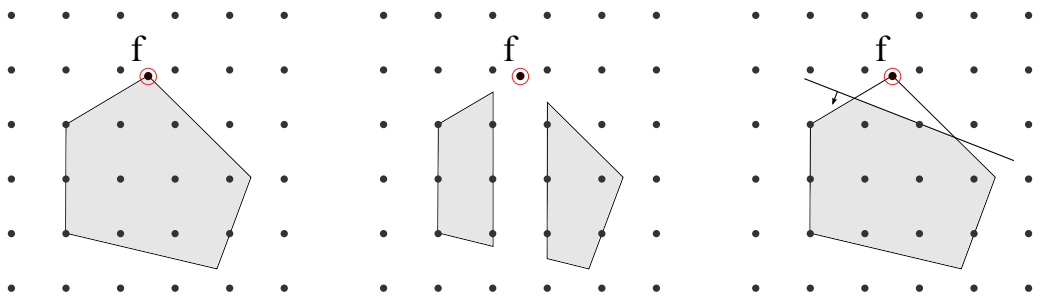


Figure 1: Branching and Cutting Planes for Integer Programming

any integer solutions from the updated search space. In practice, solvers often use a combination of branch-and-bound and cutting planes to solve IPs and MIPs. A common strategy is to iteratively apply cutting planes at the LP relaxation of the problem until some criterion is satisfied, then the branch-and-bound algorithm is applied. It can also be useful to apply *branch-and-cut* where cutting planes are added at other nodes of the branch-and-bound tree.

1.1 Motivation

1.1.1 Dangers of Numerical Computation

Many software packages have been developed to solve mathematical-programming problems. Some of the most widely used software packages today for MIP include commercial solvers IBM ILOG CLPEX, FICO XPress, Gurobi and non-commercial solvers SCIP and COIN-OR. By default, all of these software packages use an LP based branch-and-bound algorithm along with cutting planes to solve MIPs. Each of these software packages currently use floating-point numbers for their computation.

Floating-point numbers are the widely used standard for numerical computation. The most recent version of the IEEE standard for floating-point numbers is the IEEE 754-2008, released in 2008 [86]. A double precision number is represented in the form

$$(-1)^{sign} \times \left(1 + \sum_{1}^{52} 2^{-i} b_{-i}\right) \times 2^{e}$$

where *sign* is a single bit indicating the sign, b_i are 52 binary digits used to represent

the significant figures of the number and e defines the exponent, taking integer values in $e \in [-1022, 1023]$. The IEEE standard defines the representation of base 2 and base 10 floating-point numbers and dictates what results should be returned by the basic arithmetic operations in order to maintain consistent behavior across different platforms.

Floating-point computations can be performed quickly on computers but the limited size of this representation, and the limited choice of base, has its disadvantages. Many numbers, such as 1/3, cannot be represented exactly as a floating-point number and small errors can occur during arithmetic operations. The relative error incurred by a single operation is usually small but algorithms requiring many operations can accumulate and propagate these small errors, leading to errors of significant magnitude.

Another convention of numerical computation, necessitated by the possible errors, is an inexact handling of comparison relations. Software using floating-point numbers will typically consider two numbers "equal" when their difference is lower than a predefined numerical tolerance. This notion of numerical equality can be problematic for many reasons and is not an equivalence relation. If two numbers are considered $\approx$ when their difference is less than ϵ then we may have numbers a, b and c satisfying $a \approx b$, $b \approx c$, but $a \not\approx c$. Similar measures are taken when handling other relations such as $\leq$ and $\geq$. In [73] Goldberg gives a survey of issues related to floating-point computation including the problems described here.

There are several documented tragic errors involving floating-point computation. During the first US gulf war patriot missiles were used to intercept SCUD missiles, the software controlling missiles was based on floating-point computations. Repeated use of the number 1/10 in the code, which is not representable exactly as a base-2 floating point number, led to miscalculations that accumulated to form significant errors. On Feb 25, 1991, as a direct result of this miscalculation, a patriot missile failed to intercept an incoming Iraqi SCUD missile; it was off target by more than 0.6 kilometers and resulted in the death of 28 US soldiers [70]. In a later incident, the 1996 launch of the European Ariane 5 Rocket ended in failure when it went out of control and exploded 37 seconds into its flight path. The explosion was due to a software error caused by improper handling of a floating-point

calculation; the software converted a 64-bit floating-point number to a 16-bit signed integer causing an overflow and system crash. The rocket and its cargo were worth an estimated 360 million USD [59, 119].

Optimization software relying on floating-point computation can return incorrect results. Mistakes may arise from errors in floating-point computation and from the use of numerical tolerances for handling $=, \leq, \geq$ or measuring integrality of a number. By default CPLEX 12.1 uses a tolerance of `1e-6` to measure feasibility of constraints and a tolerance of `1e-5` to measure integrality of variables [85]. The manual states *"CPLEX uses numerical methods of finite-precision arithmetic. Consequently, the feasibility of a solution depends on the value given to tolerances... in the presence of numeric difficulties, CPLEX may create solutions that are slightly infeasible or integer infeasible"*. Other commercial software packages come with similar warnings. In some cases software packages may even relax these tolerances if a solution satisfying them cannot be found. The assumption is made that many users would prefer a solution that violates the tolerances than to receive no solution at all.

Errors in floating-point operations and the use of tolerances can result in the following types of mistakes when solving LP, IP or MIP problems:

1. A suboptimal solution is returned as optimal.
2. A feasible problem is declared infeasible.
3. An infeasible problem is declared feasible.

Another possible source of errors for MIP and IP problems is the use of cutting planes. Numerical problems can cause cuts to be invalid and cut off feasible solutions.

Among these possible errors some may be more likely in practice than others. As solutions are returned in a floating-point representation, any solution involving fractional values is likely to violate the constraints by a small amount, even if the returned solution is a floating-point approximation of a correct optimal solution. We would conjecture that the solver returning a suboptimal solution is a relatively likely type of error for LP, IP and MIP, although in such cases solutions may be very close to optimal. It is also possible for feasible problems to be declared infeasible for IP and MIP problems due to invalid cutting

Table 1: Objective Values Returned for `sgpf5y6`

LP Solver	**Objective Value**
Cplex 7.1 Primal	-6398.71
Cplex 7.1 Dual	-6484.44
Cplex 9.0 Primal	-6406.78
Cplex 9.0 Dual	-6484.47
Cplex 11.0 Primal	-6425.87
Cplex 11.0 Dual	-6484.46
Cplex 12.1 Primal	-6425.87
Cplex 12.1 Dual	-6484.47
Gurobi 2.0 Primal	-6484.47
Gurobi 2.0 Dual	-6484.47
XPress-15 Primal	-6380.45
XPress-15 Dual	-6344.30
XPress-20 Primal	-6349.93
XPress-20 Dual	-6408.02
QSopt Primal	-6419.94
QSopt Dual	-6480.33
CLP-1.02.01	-6480.95
CLP-1.12.0	-6481.26
GLPK-4.37	-6463.66
GLPK-4.44	-6484.47
MOSEK 6.0	-6292.06
Soplex 1.2.2	-6473.33
Exact Value	$-\frac{1621116398840608}{250000000000} \approx$ -6484.47

planes. Incorrect conclusions regarding feasibility of an LP or a solver incorrectly returning a solution for an infeasible IP/MIP may be less likely to occur frequently, but are still possible.

Table 1 gives an example of the results returned for the optimal solution value by several different LP solvers for the LP `sgpf5y6` from the Mittelmann LP test set [110]. Several of these runs were performed by William Cook and Sanjeeb Dash (personal communication), some of the other values are taken from log files on Mittelmann's optimization benchmark webpage [111]. The relative difference between the largest and smallest values is nearly 3%. This demonstrates that serious deviations can occur even when solving LPs arising from real-world problems.

Neumaier and Shcherbina [118] constructed a small IP for which multiple commercial

IP solvers incorrectly reported the problem to be infeasible; after adding an additional constraint to the model the IP solvers found a feasible solution. Their example has only 20 variables, 20 constraints and modestly-sized integer coefficients. The author of this thesis performed a study where a model was developed to determine playoff qualification of teams in the USA National Hockey League by solving an IP feasibility problem [34]. During this study there were some instances that were known to be feasible that were identified as infeasible by a commercial IP solver. The solver correctly found a feasible solution after fixing variables.

In a recent computational study [104], Margot observed many cut generators to produce invalid cuts; cutting off feasible solutions on a range of problems. He says that the question of how often a cut generator produces invalid cuts *"seems to have been completely ignored in the literature."* He argues that in order to give a fair comparison of cutting-plane generators it is necessary to measure their accuracy as well as measures of their effectiveness. If accuracy is not considered then a cut generator that produces many invalid cuts may have a faster solution time and appear to quickly close the optimality gap of a problem. Without considering its accuracy it would not be possible to recognize that the speed, and apparent effectiveness, of such a cut generator could be due to its incorrect actions. Moreover, in applications where exact or correct solutions are required, *any number of invalid cuts is unacceptable.*

Articles discussing the implementation of pure cutting plane methods [145, 146] give some explanation of why numerical problems arise as a significant problem in pure cutting plane methods. In [145] they write *"Pure cutting plane algorithms have been found not to work in practice because of numerical problems due to the cuts becoming increasingly parallel (a phenomenon accompanied by dual degeneracy), increasing determinant size and condition number, etc. For these reasons, cutting planes are in practice used in cut-and-branch or branch-and-cut mode."*

Despite the variety of ways in which floating-point computation can lead to failure, software susceptible to such error is heavily used across many industries for important decision making. This leads to the question, *why are errors tolerated by users?* We discuss

three reasons why this may be the case; exact/symbolic solutions are not required for many applications, inexact codes often produce results that are nearly correct, fast exact tools have not been readily available.

For many industrial applications an exact representation of the true optimal solution may not be more useful than a floating-point solution that is very close to feasible/optimal. To this effect Steve Wright is quoted in [27] as saying *"In many optimization problems, simple, approximate solutions are more useful than complex exact solutions."* The exact representation of a solution may have a complicated representation providing more digits of accuracy than a practitioner could act on in any meaningful way. Also, small numerical violations of some constraints may not pose any real problem because the data defining problems often comes from inexact sources.

Another important point is that, despite the possibility of errors, well designed software based on floating-point computation often produces meaningful and useful results. In most cases the solutions found may be *nearly* optimal and *nearly* feasible. In a study by Koch [95] a piece of software called `perPlex` was developed to exactly compute basic LP solutions, given an LP and a basis. By exactly computing a primal and dual solution it could also verify if a given basis was optimal. He found that for the NETLIB LP test set [117], CPLEX was able to find a truly optimal basis on all but 3 problems using the default settings; after manually tuning various settings it could find the optimal basis on all problems. The SOPLEX LP solver was also able to find most of the optimal bases using its default settings and found all optimal bases after increasing the precision to 128-bit arithmetic. This demonstrates that high performance LP solvers often find the optimal basis. It is worth noting that in many applications a near optimal solution is often acceptable. In cases when computing power is a limiting factor and users only solve problems to 1% or 5% of optimality in the first place, the user may see the extra effort required to find exact results unnecessary and impractical.

Finally, a lack of available tools to find certified or exact solutions is a reason users make due with inexact solutions. If fast exact tools were available, and did not involve a prohibitive slowdown, they could provide exact solutions to users who wanted or needed

them. The availability of such tools could also open up new application areas. In the next section we will survey several applications where correct or exact solutions are necessary.

1.1.2 Applications of Exact Mathematical Programming

There are many applications of mathematical programming where exact solutions are desirable or necessary. While this list is far from complete, it serves as a motivation for further developing exact methodologies.

1.1.2.1 Computer-Assisted Mathematics

Bailey and Borwein [14, 15] outline several possible roles of experimental mathematics: *"Gaining insight and intuition; Visualizing math principles; Discovering new relationships; Testing and especially falsifying conjectures; Exploring a possible result to see if it merits formal proof; Suggesting approaches for formal proof; Computing replacing lengthy hand derivations; Confirming analytically derived results."* Just as the frameworks of LP and MIP can be used to model many real-world problems, they can also be used to model a wide variety of mathematical structures and problems. When used to explore, evaluate and prove mathematical ideas, correctness and exactness of solutions is clearly very important.

Only a few years after the development of the exact LP solver QSopt_ex [13] it has already been used to generate output that was used to assist in proving theoretical results. Hicks and McMurry [83] use an early version of QSopt_ex to generate Farkas multipliers that are used in a proof. They prove that the branchwidth of a graph and the branchwidth of that graph's cyclic matroid are equal if the graph has a cycle of length at least 2. They used the code to assist in finding an exact solution to an LP; the correctness of the multipliers they computed can be verified by hand. In [84], the authors utilized the QSopt LP solver to provide a vector they used to prove a result related to the Caccetta-Häggkvist Conjecture. In [49], the authors use linear programming to help in the proof of new upper bounds for the densities of measurable sets in $\mathbb{R}^n$ that avoid a finite set of prescribed distances; they correct the solution returned by a floating-point LP solver to ensure the correctness of their result.

Approaches based on integer programming and column generation have been applied

to other problems including Graph Coloring [108], and computation of Crossing Numbers [29, 35]; in both cases inexact floating-point computation could lead to incorrect conclusions. The recent work of Held et al. [82] specifically addresses the problem of generating safe lower bounds for graph coloring problems.

The most high profile use of mathematical programming in proofs in recent years is Thomas Hales' proof of the Kepler Conjecture. The proof was originally announced in 1998; it has involved a series of publications and is still in the process of completion [78, 80]. One computational aspect of the proof involves solving thousands of LPs. Thomas Hales and others have established the flyspeck project [79] to fully develop a formal proof of the conjecture. The authors estimated in 2010 that it may take an additional 20 working years of effort to complete the proof. The PhD thesis of S. Obua has focused on one component of the proof involving LPs. He used interval arithmetic in combination with a floating-point LP solver to verify the result of thousands of LPs showing that many of the possible counterexamples are not counterexamples [120, 121].

In order to use solutions of LP or MIP problems in a mathematical proof it is necessary for the software to generate easily checkable certificates of correctness. Even if software is thought to be correct, the implementation of an LP or MIP solver is simply too complicated for anyone to be entirely sure there are no programming errors. In the case of linear programming a primal-dual solution is a certificate of its own optimality. The relationship between the primal and dual LP problems ensures that if a pair of primal and dual solutions is found for a problem, simply verifying that they are both feasible and have the same objective value is sufficient to show that they are both optimal solutions. Generating optimality certificates for MIP/IP problems is not as easy, but is still possible. Since integer programming is $\mathcal{NP}$-complete it is unlikely that finding optimality certificates with a small representation will be possible in general.

1.1.2.2 Feasibility Problems

Some LP or MIP problems are given with no objective function at all; where the goal is to identify a feasible solution, or determine that no solution exists. Feasibility problems may

have a small number of feasible solutions making them particularly sensitive to numerical mistakes; a single incorrect cutting plane could lead to a false result. An exact solver could decide feasibility results without the possibility of numerical mistakes. We focus special attention to feasibility problems here, as opposed to general optimization problems, because a false result could be more significant. For some optimization problems a mistake may only lead to a slightly suboptimal solution, but the false result for a feasibility problem completely reverses the conclusion.

1.1.2.3 Numerically Difficult Problems

Some models contain inherent numerical difficulties. This can occur when models contain large and small numbers together, or where the matrix is ill conditioned. It is observed in [122] that the majority of the problems in the NETLIB LP library [117] are ill conditioned. This does not imply that these LPs are unsolvable by numerical methods, but it does suggest that care must be taken to recognize and avoid numerical problems. Reformulation can sometimes help to correct numerical issues, but this may not always be possible. Bad numerical properties of models may be easily recognized by users when solvers return unexpected, infeasible or conflicting results. Many solvers are also capable of reporting numerical properties of a solution such as the condition number of the LP basis matrix, which can indicate numerical instability. Availability of exact solvers could be especially useful for solving numerically difficult problems, which could otherwise be unsolvable to any degree of confidence.

1.1.2.4 Cut generation

Many techniques have been developed to generate cutting planes for integer and mixed integer programming problems. Some classes of cuts are generated directly from a simplex tableau, others are generated using special problem structures. In some cases the separation problem is formulated as an LP [17, 18] or MIP [19, 46, 68]. Exactly solving the separation subproblems would ensure that mistakes do not lead to invalid cuts. In some cases it may even make sense to use exact methods in a cut generator that is used within a larger framework of floating-point computation. In such a case, even if the problem is not being

solved exactly, safe cut generation could eliminate the possibility of feasible solutions being cut off by incorrect cuts.

1.1.2.5 Combinatorial Auctions

A combinatorial auction involves the sale of several items where bidders are permitted to bid on combinations or packages of items in addition to bidding on individual items. This system is also referred to as package bidding. The problem of choosing winners in a combinatorial auction is $\mathcal{NP}$-complete, but can naturally be modeled as an integer programming problem. This type of auction has been proposed or implemented in many settings including logistics, airwave allocation, airport time slot allocation and financial trading. To date, billions of dollars in transactions have been decided by combinatorial auctions [50]. Exactly determining the optimal solution for these problems is of critical importance. If a false result was obtained, items could be sold according to a suboptimal solution. Discovery of a better solution could have serious legal and financial consequences.

1.1.2.6 Health Care Systems

Tools from optimization are often used to improve medical decision making. Recent work has successfully improved treatment plans for cancer [99]. Software designed for medical decision making is subjected to higher quality standards than other industries. Use of exact precision techniques could improve performance guarantees and reduce liability. Errors in medicine can occur from many causes including human error or computer errors unrelated to numerical issues. There are documented cases of medical software errors causing tragic results; in the 1980s a software error in the Therac-25 medical accelerator delivered fatal radiation overdoses to several patients before the bug was discovered [102].

1.1.2.7 Compiler Optimization

Integer programming models have been successfully used in several aspects of compiler optimization. Wilken et al. [141] used IP for instruction scheduling. Other studies have applied integer programming to the register allocation problem [75]. Incorrect decisions in these settings could cause compiler mistakes.

1.1.2.8 Chip Design Verification

Verification is a significant component of the computer chip design process. In Part III of [3] Achterberg motivates the problem, and addresses techniques to solve chip design verification problems using *constraint integer programming* (CIP). CIP is a generalization of integer programming that allows additional types of constraints, solution techniques involve a combination of methods from integer programming and constraint programming. The possibility of numerical errors is a significant concern in this application because incorrect results could result in faulty designs.

1.2 Background

1.2.1 Exact Solutions

Theoretical discussions of optimization problems use the terms solution, or optimal solution, when referring to a solution of the problem over the real or rational numbers. However, in articles reporting computational work, or software packages, the same terminology is used to describe inexact and possibly incorrect numerically obtained solutions. Therefore we make the distinction of calling solutions an *exact solution* or an *exact optimal solution* when referring to an exact symbolic description of the correct optimal solution. In general, the exact representation of the solution may be complicated. An exact solution should satisfy all of the problem constraints exactly, with no numerical error. We will also call a solution which satisfies all constraints with no error *exactly feasible* and we will call a correct lower (or upper) bound on the objective value an *exact bound* or a *valid bound*.

For both the theoretical and computational components of this dissertation we will focus attention on solving LP and MIP problems over $\mathbb{Q}$, the set of rational numbers. Although there are cases where irrational numbers are of interest, there are many reasons to focus on rational numbers, at least as a first step. Some applications involving irrational data and discussion of how safe or exact computations may be performed in the domain of real numbers are given in Chapter 5. The restriction to rational numbers is often used in theoretical settings as well, see [130]. One problem with irrational numbers is that they do not admit finite decimal representations, which is a significant obstacle for both computation

and complexity analysis. There are also theoretical differences between solving problems over the real and rational numbers. A commonly given example is the integer programming problem $\max\{x - \sqrt{2}y | x - \sqrt{2}y \leq 0; x, y \geq 1; x, y \text{ integer}\}$, note that x, y can be chosen so that $x - \sqrt{2}y$ is arbitrarily close to 0, but this value can never be attained because $\sqrt{2}$ is irrational. However, any integer programming problem with rational input data and bounded objective value attains an optimal solution. Related discussion also appears in section 1.2.4 of Espinoza [66].

1.2.2 Efforts for Correct Results in Inexact Codes

We have discussed many types of mistakes that can result from floating-point computation in optimization software. However, it is important to recognize that developers are typically very aware of the limitations of floating-point computation and implement many safeguards to minimize the occurrence of such errors. Success in solving large industrial problems is a testament to these efforts; a naive implementation of the simplex algorithm in floating-point arithmetic is likely to fail very easily, even on small problems. Software manuals generally caution users about the dangers of numerical computation and sometimes include suggestions of how to tune parameters or reformulate models to reduce numerical problems.

A historical account of many major advances for computational linear programming is given in [23]. A significant computational challenge in implementing the simplex algorithm is building and updating an LU factorization of the basis matrix. Suhl and Suhl [133] describe details of how to compute LU factorizations for sparse LP basis matrices. Their methods improve speed and numerical stability and are used by many commercial LP solvers today.

Commercial MIP codes also try to avoid generating cutting planes that are incorrect or possess bad numerical behavior. For example, in the Gurobi MIP solver, any time cuts are generated with a large deviation in the size of their coefficients they are thrown away. There are also limits on the rank of cuts that can be generated because high rank cuts can be especially susceptible to numerical problems. Even with such safeguards in place there is no guarantee that the results will be correct, and Zonghao Gu notes that programming bugs

or other problems such as compiler or hardware errors can also cause incorrect results [77]. The Gurobi LP solver also is able to switch into a higher quad-precision mode if numerical troubles are detected. Many of the safeguards listed here, and others, are implemented in most high performance optimization software systems.

1.2.3 Exact Linear Programming

Significant progress has already been made toward computationally solving LPs exactly over the rational numbers using hybrid symbolic-numeric methods [13, 55, 66, 95, 97]. We describe the basic idea of the algorithm given in [13] which is implemented as the software QSopt_ex [11]. First, a floating-point LP solver is called to solve an approximation of the problem. After performing the simplex algorithm using double-precision floating-point arithmetic this solver will return a basis, giving a structural description of the solution, alongside the numerical solution. The primal and dual solutions associated with this basis are computed in exact rational arithmetic and checked for optimality. If the solution is certified as optimal, it is returned. Otherwise, the precision of the floating-point LP solver is increased and more simplex pivots are carried out to find another solution, then the process is repeated. As a last resort pivots are done in rational arithmetic, guaranteeing that the correct solution will be identified. A similar procedure is followed to show unboundedness or infeasibility. Algorithm 1 gives an outline of this strategy.

Algorithm 1 Rational LP Algorithm (feasible, bounded case)

Input: $\max\{cx : Ax \leq b\}$ in rational precision
for precision = double, 128, 256, ..., rational **do**
  Get $\bar{A}, \bar{b}, \bar{c} \approx A, b, c$ in current precision
  Solve $\max\{\bar{c}x : \bar{A}x \leq \bar{b}\}$ by simplex algorithm
  Let $\mathcal{B}$ = optimal basis
  Compute x, y, exact rational primal/dual solutions for basis $\mathcal{B}$
  Verify optimality of solution
  if Verification successful **then**
    Return rational solution
  else
    Continue, starting next simplex solve at $\mathcal{B}$
  end if
end for

This strategy is considerably faster than using exact rational arithmetic throughout all

computations. However, the exact computation of basic solutions still remains a necessary and computationally expensive component of this algorithm. For this reason a major focus of this dissertation is the advancement of methods to solve very sparse linear systems of equations over the rational numbers.

1.2.4 Exact Mixed Integer Programming

Exact IP and MIP have seen less computational progress than exact LP, but significant first steps have been taken. An article by Neumaier and Shcherbina [118] describes methods for safe computation in MIP. The authors give strategies for generating safe LP bounds and infeasibility certificates and generating safe cutting planes. The methods they describe involve directed rounding and interval arithmetic with floating-point numbers to avoid incorrect results.

A number of other studies have considered algorithms to compute safe bounds on LP objective values relying on interval arithmetic. Jansson [87] describes algorithms for computing rigorous bounds on LP objective values, Keil et al. [93] describe a computational study of these methods on the NETLIB LP library. A more recent study by Althaus and Dumitriu [6] describe a more general method for computing safe LP bounds. Interval computations are performed by storing upper and lower bounds of a true value using floating-point numbers and performing computations in a way that preserve the bound correctness using directed rounding. This strategy will not compute exact solutions but it can compute intervals containing a solution or give rigorous objective bounds. Valid bounds on LP objectives are useful for an exact MIP solver because they can be used to prune nodes in a branch-and-bound tree without computing exact LP solutions. Interval computations are significantly faster than exact rational arithmetic and avoid some of the problems associated with floating-point numbers. However, in the presence of numerical troubles, interval computations can also break down when the gap between the upper and lower interval bounds become large.

Another central question for implementing an exact MIP solver is how to handle cutting planes. One possibility is to compute all cuts in exact arithmetic from the original problem

data, although this exact strategy could be computationally expensive. A recent study [40] describes a safe way of deriving Gomory mixed-integer cuts (GMI cuts) with floating-point numbers by using safe directed rounding. They perform computations and demonstrate this approach to be practical.

Applegate et al. [9] present a proof of optimality for the optimal tour of a traveling salesman problem with 85,900 cities. An optimal tour for the problem is given along with a certificate proving the lower bound. It is not difficult to verify the cost of the optimal tour, but building a system to represent and validate the lower bound – proving its optimality – was a significant challenge. The lower bound certificate is given in terms of the IP model of the problem. The cutting planes generated for this model are stored in a structural description and, depending on the class of cuts, their validity can be verified in different ways. Valid LP bounds in the branch-and-bound tree are obtained by correcting dual solutions from a floating-point LP solver to be feasible. The compressed proof of correctness is approximately 8MB and is available for download along with a program that verifies its correctness. This work provides a model for the type of system that could be built to verify the correctness of results for general IP or MIP problems.

A recent series of studies [47, 48, 105] have considered new approaches to proving combinatorial infeasibility via Hilbert's nullstellensatz. The authors gave algorithms to prove infeasibility for a number of combinatorial problems including non 3-colorability of graphs by solving systems of polynomial equations. Their methods are demonstrated to be computationally effective. Additionally, much of their computation uses finite field computation instead of floating-point computation, so it is not subject to the same numerical errors. Although it is not clear how their work could be extended to general IP feasibility problems it is an interesting new direction. It is conceivable that new classes of heuristics to generate certificates of infeasibility for general IP problems could be computationally effective.

1.3 Overview and Contributions of Dissertation

In Chapter 2 we describe and analyze new output-sensitive methods for computing exact solutions to linear systems of equations over the rational numbers. A common approach for

solving rational systems is to use p-adic lifting or iterative refinement to build a modular or approximate solution, then apply rational number reconstruction [57, 137]. An upper bound can be computed on the number of iterations these algorithms must perform before rational reconstruction is guaranteed to return the correct solution. However, in practice such bounds can be conservative. Output sensitive lifting is the technique of performing rational reconstruction at intermediate steps of the algorithm and verifying correctness, which allows the possibility of early termination when the solution size is small. Output-sensitive algorithms for rational systems of equations have already appeared in the literature and have been used in practice. The contribution of this chapter is describing output sensitive algorithms which are asymptotically faster than conventional methods when the final solution size is small, but maintain the same worst case complexity when solution size is large. We also introduce a variant of the iterative-refinement method that incorporates warm starts into the rational reconstruction procedure. Computational tests are performed on several classes of dense matrices and support the conclusions of the theoretical analysis.

Chapter 3 further studies methods for solving rational linear systems of equations and focuses specific attention on very sparse systems. A test set of very sparse rational systems is assembled by taking LP bases from a range of real-world LPs. We compare a direct exact solver based on LU factorization, Wiedemann's method for black-box linear algebra [140], Dixon's p-adic lifting algorithm [57], and the use of iterative numerical methods and rational reconstruction as developed by Wan [137]. Practical improvements to these algorithms are developed to exploit the extreme sparsity occurring in our test set. Extensive computational tests give a side-by-side comparison of these four methods and measure the effectiveness of a number of heuristic improvements.

Chapter 4 introduces the project-and-shift method for generating valid LP bounds for exact mixed-integer programming. The method repairs approximate LP dual solutions to be exactly feasible by performing a projection and shift to ensure all constraints are satisfied without numerical error. Bound computations are accelerated by reusing structural information about the problem through the branch-and-bound tree. We show this new method to be more generally applicable than a fast bounding method presented by Neumaier

and Shcherbina [118]. We also show that it can perform faster than solving exact LPs at every node of the branch-and-bound tree. Several variations of this algorithm are described and tested computationally in an exact branch-and-bound code implemented within the mixed-integer programming framework SCIP [3, 4].

In Chapter 5 we discuss future directions for research in exact mathematical programming. We outline some possible next steps to further develop exact methods for IP and MIP. Obstacles related to solving optimization over the real numbers are discussed. Finally, we discuss some mathematical problems where exact integer programming could be used.

CHAPTER II

OUTPUT SENSITIVE LIFTING

2.1 Introduction

Solving rational or integer linear systems of equations is a well developed area of symbolic computation. Dixon [57] gave an effective procedure for solving systems exactly by computing $\hat{x} = A^{-1}b \mod p^k$ through p-adic lifting and applying rational reconstruction to recover the exact rational solution. Wiedemann's black-box method for solving systems of equations over a finite field can also be used along with p-adic lifting or the Chinese Remainder Algorithm to solve systems in the sparse setting [92, 140]. Alternate techniques include calling a fixed precision numerical solver within an iterative refinement routine to find an extended precision solution $\hat{x} \approx A^{-1}b$, sufficient for rational reconstruction to be applied [134, 137]. Others have further developed and analyzed these methods [32, 33, 60, 64, 114, 115].

A core component of these techniques is rational number reconstruction, which allows an exact rational solution to be recovered from either a modular or approximate solution. We define the bitsize of a nonzero rational number p/q to be $bitsize(p/q) = \lceil \log(|pq|) \rceil$. For a rational vector v we will define $size(v) = \max_i bitsize(n_i/d)$ where $n/d = v$ is a representation of v using an integer vector n and a common denominator d. We remark that there are alternate definitions one could use for the size of a vector. In Chapter 3 we will use the notation of $bitsize()$ defined by looking at the maximum bitsize of a component of a vector without using a common denominator, but the notion of $size()$ defined here is relevant for the results in this chapter. In order to reconstruct a rational solution x from a modular solution, the system of equations must be solved modulo a number M, the size of which depends on the final solution size. Similarly, if a rational solution is to be reconstructed from an approximate solution, the level of approximation depends on the size of the final solution. The solution vector is unknown before solving the system so an upper bound on its size is computed to guide the rational reconstruction procedure. For an

integer system of equations $Ax = b$, Cramer's rule and the Hadamard determinant bound imply that a solution vector has size bounded by $\log(\|A\|_2^{2n-1}\|b\|_2)$. This bound can often be excessive, leading to unnecessary computation, both in the number of lifting loops that must be performed, and in the cost of performing rational reconstruction on large integers.

Output sensitive lifting is the technique of attempting rational reconstruction at intermediate steps of the algorithm with the possibility of identifying the solution early and avoiding unnecessary lifting/refinement loops. The term output sensitive lifting is used by Chen and Storjohann [32, 33] and is incorporated into their algorithms. The idea of output sensitive lifting has also been used in several other settings, such as the computation of determinants by Kaltofen [90] where it is referred to as *early termination*. Output sensitive lifting was also studied in [31] for solving systems of equations over cyclotomic fields. Use of output sensitive lifting can provide both theoretical and practical improvements when solving systems of equations exactly. It is applicable in both the dense and sparse settings.

The commonly used bounds can be weak for several reasons. Cramer's rule tells us that the denominator of a solution to an integer system $Ax = b$ will divide $det(A)$ and the Hadamard bound gives $det(A) \leq \|A\|_2^n$. While tight in some cases, the Hadamard bound is often weak; this is experimentally and probabilistically studied in [2]. Even if the determinant is well approximated by the Hadamard bound, or calculated exactly, it only provides an upper bound on the solution denominator size and there are many situations in which solutions will not meet this bound. Systems of equations may have special structure leading to small solution size, or integral solutions. In Chapter 3 of this Thesis it is observed that in systems of equations arising from linear programming applications, the solution bitsize was often much lower than this bound. In such cases, application of output sensitive lifting has a huge impact on solution times.

The size of the solution to a system of equations also depends on the right hand side. A matrix which has very complex solutions for particular right hand side vectors will have trivial or uncomplicated solutions for others. This is one way in which exact precision linear algebra differs significantly from numerical linear algebra. If a matrix can be successfully factored or inverted numerically, then solving the system for different right hand sides,

represented in machine precision, will require almost identical amounts of computation. When solving a system exactly over the rational numbers, varying the right hand side can have a drastic effect on the size of the solution and solve time.

This Chapter studies output sensitive techniques applied to two related classes of algorithms for solving linear systems. The first class of algorithm we consider is the p-adic lifting based strategy of Dixon [57] and the second algorithm is the iterative-refinement method developed by Wan [137]. Both algorithms have an iterative structure and are later defined as Algorithms 3 and 5. We will use the terminology of *p-adic lifting* and *lifting* when describing and referencing the Dixon algorithm because it constructs a modular p-adic solution from the *bottom up* in order to determine a rational solution. We will use the terminology of *iterative refinement* or *refinement* to describe Wan's Algorithm because it is based on iteratively refining an approximate solution, constructing an approximate solution in a *top down* manner. The similar structure of the Algorithms of Dixon and Wan allows output sensitive lifting to be applied in a similar way in both cases.

In Section 2 we present background material in rational reconstruction and give some related results. In Section 3 we review Dixon's method and show how it is impacted by applying output sensitive lifting. In Section 4 we describe two output sensitive versions of the iterative-refinement method, one of which incorporates warm starts for rational reconstruction. Section 5 presents computational results and Section 6 contains our conclusions.

2.2 Rational Reconstruction

2.2.1 Background

Rational reconstruction is a necessary component of all the algorithms described in this Chapter. We briefly describe rational reconstruction and some related background material. The following well known result appears in [130] as Corollary 6.3a.

Theorem 2.2.1. *There exists a polynomial algorithm which, for a given rational number α and natural number B_d tests if there exists a rational number p/q with $1 \leq q \leq B_d$ and $|\alpha - p/q| < 1/(2B_d^2)$, and if so, finds this (unique) rational number.*

If an upper bound B_d is computed for the denominators of the components of x and a

vector $\hat{x}$ satisfying $|\hat{x}-x|_\infty < 1/(2B_d^2)$ is computed, this theorem can be applied component-wise to $\hat{x}$ to compute the exact solution x. Theorem 2.2.1 is used for this purpose in the iterative-refinement method later described as Algorithm 5.

The following result is given, in more generality, as Theorem 5.26 in [136] and is analogous to Theorem 2.2.1. Also see [91] for more details and discussion.

Theorem 2.2.2. *There exists a polynomial algorithm which, for given natural numbers n, M, B_n, B_d, with $2B_nB_d \leq M$ tests if there exists a rational number p/q with $gcd(p,q) = 1$, $|p| < B_n$ and $1 \leq q < B_d$ such that $p = nq \bmod M$, and if so, finds this (unique) rational number.*

Using this result a rational system of equations can be solved by scaling it to be integral, computing a solution to the system modulo an appropriate integer M and reconstructing the exact rational solution component-wise. Theorem 2.2.2 is used for this purpose in Dixon's Algorithm which is later described as Algorithm 3.

In both of the preceding theorems, the algorithms to reconstruct rational numbers rely on the Extended Euclidean Algorithm (EEA) to compute continued fraction convergents. The standard Euclidean Algorithm computes the greatest common divisor of integers m, n by repeatedly calculating the remainder of integer divisions. The EEA records additional information along the way, including the continued fraction expansion of m/n which is computed as a byproduct of the integer divisions. The continued fraction convergents provide a sequence of increasingly accurate rational approximations. They are best approximations in the sense that each convergent is closer to m/n than any number with smaller denominator. We use $[a_0; a_1, \ldots, a_k]$ to denote the continued fraction representation of a rational number m/n, and we will call the rational number $\frac{p_i}{q_i}$ representing $[a_0; a_1, \ldots, a_i]$ the i^{th} convergent of m/n.

Algorithm 2 gives a description of the Euclidean Algorithm. The EEA will perform the same operations as the Euclidean Algorithm and its output will include the remainder sequence $r_0, \ldots, r_l$ in addition to the quotient sequence $a_0, \ldots, a_{l-1}$, where $a_i := \left\lfloor \frac{r_i}{r_{i+1}} \right\rfloor$ and

Algorithm 2 Euclidean Algorithm

Input: integers m, n
$r_0 := n, \quad r_1 := m, \quad i := 1$
while $r_i \neq 0$ **do**
  $r_{i+1} := r_{i-1} \mod r_i$
  $i := i + 1$
end while
$l := i - 1$
Return: r_l=gcd(m, n)

the matrix sequence defined by:

$$Q_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad Q_i = Q_{i-1} \begin{pmatrix} a_{i-1} & 1 \\ 1 & 0 \end{pmatrix} \quad \forall i \geq 1.$$

There are several equivalent ways to define the matrix sequence and this notation is the most convenient for our purposes. We now state some basic results concerning continued fractions; these appear (with varying notation) in either Section 3.2 of [136] or Section 12.2 of [127].

Remark 2.2.3. *Consider the rational number m/n for integers m and $n \geq 1$, let r_i be the output of Algorithm 2 and a_i, Q_i be as defined above. Also let $\frac{p_i}{q_i}$ be the i^{th} convergent of r, and define $p_{i-2} = 0$, $q_{i-2} = 1$, $p_{i-1} = 1$, $q_{i-1} = 0$. Then the following relations hold:*

1. *For $k \geq 0$, $p_k = a_k p_{k-1} + p_{k-2}$ and $q_k = a_k q_{k-1} + q_{k-2}$.*
2. $\left| \frac{p_i}{q_i} - \frac{p_{i+1}}{q_{i+1}} \right| = \frac{1}{q_i q_{i+1}}$.
3. *If $\frac{m}{n} > 0$ then $\frac{p_1}{q_1} < \frac{p_3}{q_3} < \cdots < \frac{m}{n} < \cdots < \frac{p_4}{q_4} < \frac{p_2}{q_2}$.*
4. *If $\frac{m}{n} < 0$ then $\frac{p_2}{q_2} < \frac{p_4}{q_4} < \cdots < \frac{m}{n} < \cdots < \frac{p_3}{q_3} < \frac{p_1}{q_1}$.*
5. $Q_i = \begin{pmatrix} p_{i-1} & p_{i-2} \\ q_{i-1} & q_{i-2} \end{pmatrix} \quad \forall i \geq 0$
6. $\begin{pmatrix} m \\ n \end{pmatrix} = Q_i \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$.

$$\gamma. \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = Q_i^{-1} \begin{pmatrix} m \\ n \end{pmatrix} = (-1)^i \begin{pmatrix} q_{i-2} & -p_{i-2} \\ -q_{i-1} & p_{i-1} \end{pmatrix} \begin{pmatrix} m \\ n \end{pmatrix}.$$

A straightforward implementation of rational reconstruction will require $O(d^2)$ bit operations where d is the number of bits used to represent the input. Recent articles including [103, 123, 124] describe rational reconstruction algorithms that use $O(M(d)\log(d))$ bit operations where $M(d)$ is the cost of multiplication of integers with size bounded by 2^d. Using fast multiplication of [129] we have $M(d) = O(d\log(d)\log\log(d))$. This speedup of rational reconstruction is achieved using similar ideas to the fast Extended Euclidean Algorithm.

2.2.2 Numerical Examples of Rational Reconstruction

In this Section we will present some illustrative numerical examples of rational number reconstruction, demonstrating how number are reconstructed using Theorems 2.2.1 and 2.2.2. Suppose the rational number

$$p/q = \frac{1234}{56789}$$

is an unknown solution value to a problem.

In order to apply Theorem 2.2.1 to reconstruct this number we would need an upper bound B_d on the denominator q, and an approximation α of p/q which satisfies $|\alpha - p/q| < 1/(2B_d^2)$. In this case $B_d = 10^7$ and $\alpha = 0.021729560302171$ satisfy these conditions.

Now, we list the first 9 continued fraction convergents of α and observe that taking the last convergent with denominator less than B gives p/q.

Example 2.2.4 (Continued fraction convergents of α)**.**

$$\frac{0}{1}, \frac{1}{46}, \frac{49}{2255}, \frac{99}{4556}, \frac{148}{6811}, \frac{543}{24989}, \frac{691}{31806}, \frac{\mathbf{1234}}{\mathbf{56789}}, \frac{1960973537}{90244510691}, \ldots$$

Now we show an example involving Theorem 2.2.2. To meet the conditions of this theorem we need numbers n, M, B_n, B_d, satisfying $2B_nB_d \leq M$, $|p| < B_n$, $1 \leq q < B_d$ and $p = nq \bmod M$. Taking $n = 91951526394851$, $M = 101^7$ and $B_n = B_d = 10^7$ we satisfy these conditions. In order to reconstruct p and q, the algorithm will perform the EEA on numbers n and M. The sequence of steps of the EEA generate a sequence of multipliers and

remainders s_i, t_i and r_i satisfying $n \times s_i - M \times t_i = r_i$, where at the final step $r_i = gcd(n, M)$. At each step of the algorithm the equation $r_i = ns_i \bmod M$ holds and it is from such a step that p, q are extracted. Below we show the intermediate steps of the EEA applied to n and M and observe that p, q appear at the ninth iteration, the first and only step satisfying both $s_i \leq B_d$ and $r_i \leq B_n$.

Example 2.2.5 (Steps of the Euclidean Algorithm applied to n, M)**.**

$$
\begin{array}{rcrrcrcr}
n & \times & 1 & -M & \times & 0 & = & 91951526394851 \\
n & \times & 1 & -M & \times & 1 & = & -15262008815850 \\
n & \times & 7 & -M & \times & 6 & = & 379473499751 \\
n & \times & 281 & -M & \times & 241 & = & -83068825810 \\
n & \times & 1131 & -M & \times & 970 & = & 47198196511 \\
n & \times & 1412 & -M & \times & 1211 & = & -35870629299 \\
n & \times & 2543 & -M & \times & 2181 & = & 11327567212 \\
n & \times & 9041 & -M & \times & 7754 & = & -1887927663 \\
n & \times & \mathbf{56789} & -M & \times & 48705 & = & \mathbf{1234} \\
n & \times & 86882919866 & -M & \times & 74515004879 & = & -213 \\
n & \times & 434414656119 & -M & \times & 372575073100 & = & 169 \\
n & \times & 521297575985 & -M & \times & 447090077979 & = & -44 \\
n & \times & 1998307384074 & -M & \times & 1713845307037 & = & 37 \\
n & \times & 2519604960059 & -M & \times & 2160935385016 & = & -7 \\
n & \times & 14596332184369 & -M & \times & 12518522232117 & = & 2 \\
n & \times & 46308601513166 & -M & \times & 39716502081367 & = & -1
\end{array}
$$

2.2.3 Certifying Solution Vectors

We now consider some sufficient conditions that can be efficiently checked to certify correctness of a rational system of equations. A lemma similar to the following was given by [30] and was used in [32, 33]. It can be used to certify correctness of reconstructed solutions while requiring little computation. Throughout the rest of the Chapter we will use $\|A\|_{\max} = \max |a_{ij}|$.

Lemma 2.2.6. *Suppose A is a square integer matrix, y, b are integer vectors, and $d \geq 0$ is an integer. If for some integer M*

$$Ay = bd \mod M \qquad \textit{and}$$

$$\max(d\|b\|_\infty, n\|A\|_{\max}\|y\|_\infty) < M/2$$

then $Ay = bd$.

Proof. Suppose the conditions hold but $Ay - bd \neq 0$. Since $Ay - bd$ must be integral and $Ay = bd \mod M$ we have $\|Ay - bd\|_\infty \geq M$. But we also have

$$\|Ay - bd\|_\infty \leq \|Ay\|_\infty + \|bd\|_\infty \leq 2\max(d\|b\|_\infty, n\|A\|_{\max}\|y\|_\infty) < M$$

which gives a contradiction. □

We also make the observation that the statement of this lemma can be adjusted by replacing $n\|A\|_{\max}\|y\|_\infty$ with $\|A^T\|_2\|y\|_2$, and the proof will carry through identically by the Cauchy-Schwartz inequality.

Corollary 2.2.7. *Suppose A, b, y, d satisfy the conditions of Lemma 2.2.6. If $Ay = bd$, then $d \neq 0$ implies $x = y/d$ solves $Ax = b$, and $d = 0$ implies singularity of A.*

Suppose a solution to a system of equations is computed modulo p^k for some integer k and rational reconstruction is attempted without knowledge of valid bounds, by using guessed bounds such as $B_n = B_d = \lceil \sqrt{p^k/2} \rceil$ as in Theorem 2.2.2. In such a case, since B_n, B_d are not known to be valid, the reconstructed solution may be incorrect. Lemma 2.2.6 gives a very easily checked condition to certify correctness of the solution. If the solution is known to satisfy the modular system of equations, then checking the remaining conditions of the theorem requires only a few multiplications, in contrast to a high precision matrix-vector multiplication required to evaluate the linear equations exactly.

We now provide an analogue for the case when rational numbers are reconstructed from approximate solutions.

Lemma 2.2.8. *Suppose A is a square integer matrix, b is an integer vector and x is a rational vector that is known to satisfy $\|x - A^{-1}b\|_\infty < \epsilon$. If $x = y/d$, where y is an integer vector, and d is an integer satisfying $d < 1/(n\|A\|_{\max}\epsilon)$, then $Ax = b$.*

Proof. Suppose $\hat{x} = A^{-1}b$ and $Ax \neq b$. Since $Ay - bd \neq 0$ is integral, $\|Ay - bd\|_\infty \geq 1$. Next $d < 1/(n\|A\|_{\max}\epsilon)$ and $\|x - \hat{x}\|_\infty \leq \epsilon$ implies $\|x - \hat{x}\|_\infty < 1/(nd\|A\|_{\max})$. So we have

$$\|Ay - bd\|_\infty = d\|Ax - b\|_\infty = d\|A(x - \hat{x})\|_\infty \leq dn\|A\|_{\max}\|x - \hat{x}\|_\infty < 1$$

which gives a contradiction. □

An alternative version of this system check criterion can be stated in terms of the amount by which the candidate solution violates the equations.

Lemma 2.2.9. *Suppose A is a square integer matrix, b is an integer vector and x is a rational vector that is known to satisfy $\|Ax - b\|_\infty < \epsilon$. If $x = y/d$, where y is an integer vector, and d is an integer satisfying $d < 1/\epsilon$, then $Ax = b$.*

Proof. Suppose $Ax \neq b$ then since $Ay - bd \neq 0$ is integral, $\|Ay - bd\|_\infty \geq 1$. This gives

$$1 \leq \|Ay - bd\|_\infty = d\|Ax - b\|_\infty = d\epsilon < 1$$

which is a contradiction. □

Lemma 2.2.8 implies that if a rational solution x is reconstructed from an approximate solution, where the common denominator of the vector x is small enough, and its accuracy is known to satisfy a required bound, then its correctness can be certified without evaluating the equations.

While Lemmas 2.2.6 and 2.2.8 provide conditions to quickly certify correctness of solutions that have been reconstructed, their conditions are not necessary, and a correct rational solution may fail to satisfy them. The following example illustrates that this gap can depend on both the dimension and size of the data entries.

Example 2.2.10. *Suppose $A = aI_n$ for an integer a and I_n is the n dimensional identity matrix. For an n dimensional vector $b = [a, a, \ldots, a]^T$, $x = y/d = [1, 1, \ldots, 1]^T/1$ is a*

solution to $Ax = b$ for all positive integers a, n. After solving this system modulo $M \geq 2$ for a number M not dividing a, the correct solution will be reconstructed successfully. However, the conditions in Lemma 2.2.6 will not be met unless a solution is computed modulo $M \geq 2na$.

This example also can be applied to Lemma 2.2.8, where we see that any value of $\epsilon \leq 1/2$ is sufficient for the correct solution to be determined using rational reconstruction, however the conditions are not satisfied unless the system is solved to within an error $\epsilon \leq 1/(2na)$.

Therefore, to design an algorithm that will compute and certify the correct rational solution as soon as possible, these techniques have both practical and theoretical drawbacks. We also mention that if an incorrect solution vector is checked for correctness by evaluating the linear equations of the system, its incorrectness can likely be discovered after evaluating only a small number of the equations. Therefore, although evaluating all of the linear equations could be computationally expensive, we expect identifying incorrectness of solutions to be considerably faster.

We now provide necessary and sufficient conditions that can be used to verify correctness of a reconstructed solution. While these conditions are not as easily checked as those in the previously discussed results they can be easier to verify than evaluating the equations using full precision.

Lemma 2.2.11. *Suppose A is a square integer matrix, y, b are integer vectors, d is a positive integer and $x = y/d$. Then $Ax = b$ if and only if there exists an integer $M \geq 1$ such both of the following conditions hold.*

$$Ay = bd \mod M \tag{1}$$

$$\|Ax - b\|_\infty < M/d \tag{2}$$

Proof. If $Ax = b$ then for any integer $M \geq 1$ the modular equation must hold and $\|Ax - b\|_\infty = 0 < M/d$. For the reverse direction suppose $Ax \neq b$, then for any positive integer M, $Ay = bd \mod M$ implies $\|Ay - bd\|_\infty \geq M$, which means $\|Ax - b\|_\infty \geq M/d$ so both conditions cannot hold at once. □

Thus, if solution y/d is known to satisfy $Ay = bd \mod M$, to check $Ax = b$ it is necessary and sufficient that $\|Ax-b\|_\infty < M/d$, which can be verified using approximations or interval arithmetic. Similarly, if we have computed a rational solution $x = y/d$ satisfying $\|Ax - b\|_\infty < \epsilon$, this result tells us that instead of explicitly checking $Ax = b$, it is sufficient to select any positive integer $M \geq d/\epsilon$ and verify that $Ay = bd \mod M$. Evaluating this modular system will require less computation than verifying the full precision equations, especially when d/ϵ is reasonably small.

Related results also appear in [65] where it is shown, under some assumptions, that if the solutions at two (or more) consecutive lifting steps are the same there is a high probability that they give the correct answer. As our focus is on deterministic algorithms we refer the reader to this reference for more information.

2.2.4 Warm Starting Rational Reconstruction

For the algorithm presented in Section 4 it is of interest to understand how the output of the Extended Euclidean Algorithm, and rational reconstruction, can change when its input is slightly perturbed. Understanding this will help us to perform warm starts for the rational reconstruction algorithm corresponding to Theorem 2.2.1. The following appears as Theorem A in [100]; related results appear in [103, 123].

Theorem 2.2.12. *Let $\frac{p_{k-1}}{q_{k-1}}, \frac{p_k}{q_k}$ be two consecutive convergents to a number β. Then these fractions are consecutive convergents to α if and only if*

$$\left|\alpha - \frac{p_k}{q_k}\right| < \frac{1}{q_k(q_k + q_{k-1})}.$$

This theorem gives conditions which can be used to verify that a sequence of continued fraction approximations is correct up to a certain point. The following result applies this theorem to the framework of rational reconstruction.

Theorem 2.2.13. *Let x, α be a rational numbers satisfying $|x - \alpha| < 1/(2B^2)$ for some integer B. Suppose $\frac{p_k}{q_k}$ is any continued fraction convergent of x such that $q_k < B$. If $k \geq 3$ then either $\frac{p_{k-2}}{q_{k-2}}, \frac{p_{k-1}}{q_{k-1}}$ or $\frac{p_{k-1}}{q_{k-1}}, \frac{p_k}{q_k}$ are two consecutive convergents of α.*

Proof. Without loss of generality we may assume $\frac{p_{k-1}}{q_{k-1}} \leq x \leq \frac{p_k}{q_k}$. First suppose $|\alpha - \frac{p_{k-1}}{q_{k-1}}| < \frac{1}{q_k q_{k-1}}$. Then by Remark 2.2.3 if $k \geq 1$, $q_k \geq q_{k-1} + q_{k-2}$, so we have

$$\left|\alpha - \frac{p_{k-1}}{q_{k-1}}\right| < \frac{1}{q_k q_{k-1}} \leq \frac{1}{q_{k-1}(q_{k-1} + q_{k-2})}$$

and by Theorem 2.2.12, $\frac{p_{k-2}}{q_{k-2}}, \frac{p_{k-1}}{q_{k-1}}$ are two consecutive convergents of α. So we may assume that $|\alpha - \frac{p_{k-1}}{q_{k-1}}| \geq \frac{1}{q_k q_{k-1}}$. From $|\frac{p_k}{q_k} - \frac{p_{k-1}}{q_{k-1}}| = \frac{1}{q_k q_{k-1}}$ and $\frac{p_{k-1}}{q_{k-1}} \leq x$ it follows that $\frac{p_k}{q_k} \leq \alpha$. Finally $|x - \alpha| < \frac{1}{2B^2}$ and $x \leq \frac{p_k}{q_k}$ gives

$$\left|\alpha - \frac{p_k}{q_k}\right| < \frac{1}{2B^2} \leq \frac{1}{2q_k^2} \leq \frac{1}{q_k(q_k + q_{k-1})}.$$

By Theorem 2.2.12 we have $\frac{p_{k-1}}{q_{k-1}}, \frac{p_k}{q_k}$ as two consecutive convergents of α, which establishes our desired result. □

Thus, if rational reconstruction is performed using an approximate input $x \approx \alpha$, the intermediate steps of the EEA will be correct in all but possibly the last step. If x is later refined to a more accurate approximation of α then in order to apply rational reconstruction again, we can start the algorithm where it left off, with the need for at most one step backward.

2.3 Output Sensitive Lifting for Dixon's Method

Algorithm 3 describes Dixon's well known algorithm for solving an integer system of equations $Ax = b$ [57]. This algorithm is sometimes referred to as the *p-adic lifting algorithm* for solving linear systems and related ideas are considered by other authors [96, 112, 138].

His algorithm has three steps; first an inverse of $A \mod p$ is computed, second p-adic lifting is used to construct a solution mod p^k, then the rational solution is reconstructed. Dixon showed the following bound regarding the complexity of his algorithm assuming $Ax = b$ is an n dimensional square system of equations and $\|A\|_{\max}, \|b\|_\infty$ are bounded by a constant. In his analysis he also assumed that a prime p not dividing $det(A)$ and bounded by a constant not depending on A was found (such a prime might not exist).

Theorem 2.3.1. *Let $Ax = b$ be an n dimensional square nonsingular integer system of equations. If the entries of A, b are bounded by a constant and p is bounded by a constant, then Algorithm 3 will find the rational solution using $O(n^3 \log^2(n))$ bit operations.*

Algorithm 3 Standard Dixon Algorithm

Input: A, b, p $\{Ax = b$ is system to be solved, p is a prime not dividing $det(A)\}$
Compute $A^{-1} \bmod p$
$\hat{x} := 0, i := 0, d := b, B := 2\|A\|_2^{2n-1}\|b\|_2$
while $p^i < B$ **do**
  $y := A^{-1}d \mod p$
  $\hat{x} := \hat{x} + yp^i$ $\{$This will set $\hat{x} = A^{-1}b \mod p^{i+1}\}$
  $d := \left(\frac{d-Ay}{p}\right)$
  $i := i + 1$
end while
x:=Reconstruct$(\hat{x}, p^i)$
Return: x $\{$Solution to system$\}$

We will review how this bound was obtained. The inversion of $A \bmod p$ can be done with $O(n^3)$ operations. There will be $O(\log(B))$ lifting steps. At the i^{th} iteration of the algorithm, entries of $\hat{x}$ will be in the range $[0, p^{i+1}-1]$ and d is updated to equal $(b-A\hat{x})/p^i$. Therefore d will have integral entries with bitsize $O(\log n)$. Updating $y, \hat{x}$ and d in each lifting step is accomplished with $O(n^2 \log(n))$ bit operations. This gives a total cost of $O(n^2 \log(n) \log(B))$ over all lifting steps. The rational number reconstruction, using the Extended Euclidean Algorithm component-wise, has a cost of $O(n \log^2(B))$ operations. We also have $\log(B) = \log(2\|A\|_2^{2n-1}\|b\|_2) = O(n \log(n))$, so the bit complexity is

$$O(n^3 + n^2 \log(n) \log(B) + n \log^2(B)) = O(n^3 \log^2 n).$$

In practice a word sized prime that does not divide $det(A)$ can often be identified by randomly selecting a prime. In the case that a prime is selected that does divide $det(A)$ this can be recognized in the first step of the algorithm when computing $A^{-1} \bmod p$. In general the size of p will depend on the dimension and size of entries in A. In [132] an algorithm to determine a prime p with size $O(\log n + \log\log \|A\|_{\max})$ is analyzed in Corollary 36.

More general complexity analysis of Dixon's method is given by Mulders and Storjohann as Theorem 20 in [114], which does not assume constant bounds on A, b and p. We restate a version of their result as Theorem 2.3.2.

Theorem 2.3.2. *The p-adic lifting algorithm for solving a system of integer equations*

exactly is correct and given input A, b, p it will terminate after

$$O(n^3(\log n + \log \|A\|_{\max} + \log p)^2 + n \log^2 \|b\|_\infty)$$

bit operations.

Their statement of the algorithm differs slightly from Algorithm 3 but follows the same basic structure. Mulders and Storjohann prove this theorem using standard arithmetic and in a later paper [115] they also give a detailed complexity analysis of Dixon's method using fast arithmetic as Proposition 31.

Algorithm 4 describes the output sensitive version of Dixon's Algorithm. In this variant of the algorithm, instead of waiting until the modulus of the intermediate solution $\hat{x}$ exceeds the bound B, rational reconstruction is attempted at intermediate steps. Success of the rational reconstruction is not theoretically guaranteed at these steps, so the reconstructed solution must be verified. The bit complexity of the output sensitive Dixon algorithm will depend on which steps are specified as reconstruction steps. Here we will make the choice that reconstruction is attempted at a geometric frequency, namely at steps i where $i = 2^k$ for some integer k. This choice of frequency is important. For example, if reconstruction is attempted at predetermined constant length intervals the bit complexity of the algorithm would change. We will use $\log(S) = size(A^{-1}b)$ to represent the size of the solution. Recall that we have defined $size()$ of a vector to be the maximum bitsize over all of its entries after all entries are represented with a common denominator. We also know that the numerator and denominator bounds from Cramer's rule and the Hadamard bound gave us $2S \leq B = 2\|A\|_2^{2n-1}\|b\|_2$. We will now give an analysis of the complexity of Algorithm 4 in terms of the system dimension n and the solution size $\log(S)$. For simplicity of presentation we assume entries of A, b, p are bounded by a constant.

We first give a simplified analysis of Algorithm 4 where we are assuming all of the entries in the problem input, and the prime p are of size bounded by a constant.

Theorem 2.3.3. *Let $Ax = b$ be an n dimensional square nonsingular integer system of equations and suppose the entries of A, b are bounded by a constant. Also suppose that p is a prime bounded by a constant which does not divide $det(A)$ and $\log(S) = size(A^{-1}b)$.*

Algorithm 4 Output Sensitive Dixon Algorithm

Input: A, b, p {$Ax = b$ is system to be solved, p is a prime not dividing $det(A)$}
Compute $A^{-1} \bmod p$
$\hat{x} := 0, i := 0, d := b$
while solution not found **do**
  $y := A^{-1}d \mod p$
  $\hat{x} := \hat{x} + yp^i$
  if reconstruction step **then**
    $x :=$Reconstruct$(\hat{x}, p^{i+1})$ {Using Theorem 2.2.2 component-wise with $B_n = B_d = \sqrt{p^{i+1}/2}$}
    Check $Ax = b$
  end if
  $d := \left(\frac{d - Ay}{p}\right)$
  $i := i + 1$
end while
Return: x {Solution to system}

Then the Output Sensitive Dixon Algorithm terminates after $O(n^3 + n^2 \log(n) \log(S))$ bit operations.

Proof. Reducing A mod p and computing A^{-1} mod p will require $O(n^3)$ operations.

Next we will show that the number of loops is $O(\log(S))$. In the i^{th} loop of the algorithm a solution to the system modulo p^{i+1} will be constructed. By Theorem 2.2.2 the reconstruction routine is guaranteed to succeed when both B_n and B_d exceed the (unknown) quantity S. Therefore if reconstruction is attempted at a loop where $B_n = B_d = \sqrt{p^{i+1}/2} \geq S$, or $i \geq 2(\log(S)/\log(p))$, the correct solution is ensured to be correctly reconstructed. The geometric choice of reconstruction frequency ensures we will perform at most two times the necessary number of loops beyond the earliest loop where i is large enough to correctly reconstruct the solution.

The number of operations performed in each loop, excluding the cost of the rational reconstruction attempts and the solution check, is $O(n^2 \log(n))$ as in the standard Dixon Algorithm. The computational cost of performing rational reconstruction while in loop i is $O(ni^2)$ because each component will have bitsize $O(i)$. In order to check the reconstructed candidate solution x we will first transform to a representation having a common denominator d to get $x = z/d$. Then if z or d exceed the numerator and denominator bounds B_n, B_d the check is aborted, otherwise the solution is checked by computing Az and bd. The

cost of computing Az and bd will be $O(n^2 \log(n) i)$ since it requires performing an integer matrix-vector multiplication where the entries of the matrix are bounded by a constant, and the entries of the vector are bounded by $2^{O(i)}$. Thus, since reconstruction will be attempted and verified at steps $i = 2^k$ for $k = 1, 2, \ldots, O(\log\log(S))$ we have the following bound on the combined cost of rational reconstruction and solution checking:

$$\sum_{k=1}^{O(\log\log(S))} \left(O(n(2^k)^2) + O(n^2 \log(n) 2^k) \right) = O(n \log^2(S) + n^2 \log(n) \log(S)).$$

Using $\log(S) = O(n \log(n))$ we have the following bound on the total number of bit operations $O(n^3 + n^2 \log(n) \log(S) + n \log^2(S) + n^2 \log(n) \log(S)) = O(n^3 + n^2 \log(n) \log(S))$.

□

We see that this algorithm gives an improved runtime if the solution size is small is small. Moreover, under the assumption that sizes of A, b, p are bounded by a constant, $\log(S) = O(n \log n)$ so this matches the worst case bound of $O(n^3 \log^2 n)$ given in Theorem 2.3.1.

We will now analyze Algorithm 4 without assuming constant size bounds on entries of A, b and p. This is similar to the proof of Theorem 20 in [114].

Theorem 2.3.4. *Let $Ax = b$ be a square nonsingular integer system of equations and a prime p is known not dividing $det(A)$ and $\log(S) = size(A^{-1}b)$ then the Output Sensitive Dixon Algorithm terminates after*

$$O\left(n^3 \log^2(p) + n^2 \log(\|A\|_{\max}) \log(p) + n \log(S) \log(\|b\|_\infty) + n^2 \log(S)(\log n + \log \|A\|_{\max} + \log p)\right)$$

bit operations.

Proof. We will bound the complexity of this algorithm in three steps. First we look at the initial cost of reducing A mod p and computing its mod p inverse. Second we consider the cost of all of the lifting steps. Third, we consider the combined cost of performing rational reconstruction including checking the intermediate solutions. We also note that

the Hadamard bound tells us that the solution size, $\log(S) = O(n\log(n) + n\log\|A\|_{\max} + \log\|b\|_\infty)$

Inversion: Reduction of $A \mod p$ and computation of $A^{-1} \mod p$ can be done using

$$O(n^2\log(\|A\|_{\max})\log(p) + n^3\log^2(p))$$

bit operations.

Lifting: The number of loops the algorithm will perform is $O(\log(S)/\log(p))$. We will now count the cost of each lifting loop.

At the i^{th} step of the algorithm

$$d = \left(\frac{b - A(A^{-1}b \bmod p^i)}{p^i}\right)$$

and since p^i must divide $b - A(A^{-1}b \bmod p^i)$ this implies d always has entries with absolute value at most $\|b\|_\infty + n\|A\|_{\max}$. To compute $y := A^{-1}d \mod p$ we reduce $d \bmod p$ which will cost $O(n\log(p)(\log\|b\|_\infty + \log n + \log\|A\|_{\max}))$ and then doing a mod p matrix-vector multiplication will cost $O(n^2\log^2(p))$.

Now we bound the cost of computing $\hat{x} := \hat{x} + yp^i$ in each loop. Observe that $\hat{x}$ will require at most $O(\log S)$ bits to represent it at any stage of the algorithm. p^i can be updated and stored from step to step and will always have size $O(\log(S))$. y will have size $O(\log p)$. The dominating cost will be the multiplication of yp^i which will cost $O(n\log(S)\log(p))$.

Finally we consider the cost of updating $d := (d - Ay)/p$. Since y has entries with absolute value at most p, and entries of Ay are at most $n\|A\|_{\max}p$, the cost of the multiplications and additions required to compute Ay is bounded by $O(n^2(\log(\|A\|_{\max})\log(p) + \log n))$. Subtracting Ay from d will cost $O(n(\log n + \log p + \log\|A\|_{\max} + \log\|b\|_\infty))$ and then dividing by p will have cost $O(n\log(p)(\log\|b\|_\infty + \log n + \log\|A\|_{\max} + \log p))$.

Combining terms we have the following bound on the computation required in each loop:

$$O(n\log(p)\log(S) + n\log(p)(\log n + \log\|b\|_\infty)$$
$$+n^2(\log(\|A\|_{\max})\log(p) + \log n + \log^2(p))$$

Multiplying the total loop cost by the number of loops $O(\log(S)/\log(p))$ can be bounded by:

$$O(n\log^2(S) + n\log(S)\log(\|b\|_\infty) \\ +n^2\log(S)(\log(\|A\|_{\max}) + \log n + \log p))$$

as a bound on all computation in the lifting steps.

Reconstruction: Finally we consider the combined cost of the rational reconstruction and solution checks performed. At the i^{th} loop the cost of rational reconstruction is bounded by $O(n(i\log(p))^2)$ since each of the n components will have bitsize $O(i\log(p))$. Once a candidate solution x is reconstructed it is checked for correctness. This can be done by first transforming it to be represented with a common denominator as $z/d = x$. If any entries of z or the common denominator d exceed the numerator and denominator bounds $B_n = B_d = \sqrt{p^{i+1}/2}$ then the check is aborted. Now, if the bitsize of d and the entries of z are $O(i\log(p))$ then the cost of computing Az will be $O(n^2(\log(\|A\|_{\max})(i)\log(p)+\log(n)))$ since it requires performing an integer matrix-vector multiplication where the entries of the matrix are bounded by $\|A\|_{\max}$, and the entries of the vector have bitsize $O(i\log p)$ and the largest entries of the resulting values of Az have bitsize bounded by $O(\log\|A\|_{\max}+\log n+(i)\log(p))$. The cost of computing bd is $O(n\log(\|b\|_\infty)(i)\log(p))$. Therefore the total cost of rational reconstruction and solution checking over all loops $k = 1, 2, \ldots, O(\log(\log(S)/\log(p)))$ where it is applied is:

$$\sum_{k=1}^{O(\log(\log(S)/\log(p)))} O\left(n\log^2(p)(2^k)^2 \right. \\ \left. +n^2(\log(\|A\|_{\max})\log(p)2^k + \log n) + n\log(\|b\|_\infty)\log(p)2^k\right).$$

This summation is bounded above by:

$$O(n\log^2(S) + n^2\log(S)(\log\|A\|_{\max} + \log n) \\ +n\log(S)\log(\|b\|_\infty))$$

Total cost: Finally, considering the cost of the matrix inversion, lifting and reconstruction

attempts can all be bounded by:

$$O\left(n^3 \log^2(p) + n^2 \log(\|A\|_{\max}) \log(p) + n \log^2(S) \right.$$
$$\left. + n^2 \log(S)(\log n + \log \|A\|_{\max} + \log p) + n \log(S) \log(\|b\|_\infty)\right)$$

And using the fact that $\log(S) = O(n \log(n) + n \log(\|A\|_{\max}) + \log \|b\|_\infty)$ we can simplify this to:

$$O\left(n^3 \log^2(p) + n^2 \log(\|A\|_{\max}) \log(p) + n \log(S) \log(\|b\|_\infty)\right.$$
$$\left. + n^2 \log(S)(\log n + \log \|A\|_{\max} + \log p)\right)$$

which gives the desired bound. □

This result agrees with the result of Theorem 2.3.1 when the sizes of the input numbers are all treated as constants. Comparing this to Theorem 2.3.2 we see that the algorithm can perform asymptotically faster if the final solution size is small. We will also see that it performs no worse even when the solution size is as large as possible.

Corollary 2.3.5. *The bit complexity of the Output Sensitive Dixon Algorithm as given by Theorem 2.3.4 is*

$$O(n^3(\log n + \log \|A\|_{\max} + \log p)^2 + n \log^2 \|b\|_\infty)$$

Proof. The result of Theorem 2.3.4 gives the following bound:

$$O\left(n^3 \log^2(p) + n^2 \log(\|A\|_{\max}) \log(p) + n \log(S) \log(\|b\|_\infty)\right.$$
$$\left. + n^2 \log(S)(\log n + \log \|A\|_{\max} + \log p)\right)$$

By the Hadamard bound we know that $\log(S) = O(n \log(n) + n \log(\|A\|_{\max}) + \log \|b\|_\infty)$, so plugging in this bound and removing all terms that already satisfy the desired bound we are left with:

$$O(n^2 \log(\|b\|_\infty)(\log n + \log \|A\|_{\max} + \log p))$$

To observe that these terms also meet our desired bound consider the two cases where either $\log(\|b\|_\infty) \geq n(\log n + \log \|A\|_{\max} + \log p)$ or $\log(\|b\|_\infty) \leq n(\log n + \log \|A\|_{\max} + \log p)$ and the result follows. □

2.4 Output Sensitive Iterative Refinement

The iterative-refinement method solves linear systems of equations over the rational numbers by calculating a highly accurate approximate solution and applying Theorem 2.2.1 to reconstruct the rational solution. The approximate solution is calculated and iteratively refined using numerical methods.

Algorithm 5 Iterative Refinement Method

Input: A, b {$Ax = b$ is system to be solved}
Compute numerical LU factorization of A
$N := 0$ {Numerator of the approximation}
$D := 1$ {Common denominator of approximation}
$B := 2\|A\|_2^{2n}$
$\Delta := b$ {Error measure of solution at each step}
while $D < B$ **do**
  Compute $\hat{x} :\approx A^{-1}\Delta$ {Using numerical LU factorization}
  Choose an integer $\alpha < \frac{||\Delta||_\infty}{||\Delta - A\hat{x}||_\infty}$ where $\alpha\hat{x}$ is within floating point range
  Set $\bar{x} \approx \alpha\hat{x}$ {Round to the nearest integer}
  $\Delta := \alpha\Delta - A\bar{x}$ {Update the residual}
  $D := D \times \alpha$ {Update the denominator}
  $N := N \times \alpha + \bar{x}$
end while
Reconstruct x using N/D
Return: x {Solution to system}

This general idea was used by [134], and was later improved upon by Wan [137] who showed how to more efficiently keep track of the error by rounding and adjusting the approximate solution. In order to guarantee correctness of the reconstructed solution, Cramer's rule and the Hadamard bound are used, as in Dixon's method, to bound the size of the rational solutions. Algorithm 5 gives a description of the iterative-refinement method similar to the algorithm of Wan [137]. All versions of the iterative-refinement method described in this section require the assumption that the matrix can be successfully numerically factored or inverted. The iterative structure of this algorithm is similar to Dixon's method and rational reconstruction can also be attempted at intermediate steps to make it output sensitive. This strategy is described as Algorithm 6.

If rational reconstruction is attempted component-wise at an intermediate step of the

Algorithm 6 Output Sensitive I. R. Method

Input: A, b {$Ax = b$ is system to be solved}
Compute numerical LU factorization of A
$N := 0$ {Numerator of the approximation}
$D := 1$ {Common denominator of approximation}
$\Delta := b$ {Error measure of solution at each step}
while solution not found **do**
  Compute $\hat{x} :\approx A^{-1}\Delta$ {Using numerical LU factorization}
  Choose an integer $\alpha < \frac{||\Delta||_\infty}{||\Delta - A\hat{x}||_\infty}$ where $\alpha\hat{x}$ is within floating point range
  Set $\bar{x} \approx \alpha\hat{x}$ {Round to the nearest integer}
  $\Delta := \alpha\Delta - A\bar{x}$ {Update the residual}
  $D := D \times \alpha$ {Update the denominator}
  $N := N \times \alpha + \bar{x}$
  if reconstruction step **then**
    $x :=$Reconstruct(N, D) {Using Theorem 2.2.1 and $B_d = \sqrt{D/2}$}
    Check $Ax = b$
  end if
end while
Return: x {Solution to system}

iterative-refinement algorithm then by Theorem 2.2.13 some steps of the EEA will be correct, even if the reconstructed solution is not correct. Therefore the strategy of Algorithm 6 may recompute the same leading sequence of convergents each time rational reconstruction is attempted. Algorithm 7 describes a procedure to warm start rational reconstruction within the output-sensitive iterative-refinement method in order to avoid this recomputation. It differs from the previous algorithms discussed because it interweaves the rational reconstruction routine with the refinement steps instead of calling rational reconstruction as a separable routine.

After performing each step of the while loop we obtain a refinement of the approximation of the solution to the system of equations. We will use N_i/D to represent the approximation of the i^{th} component, and note that this notation does not appear in the algorithm description because N/D is stored in terms of its EEA matrix and remainder sequence instead of explicitly. For the discussion here we assume that the numerical solver is providing enough correct solution bits that at any step of the algorithm $|N_i/D - (A^{-1}b)_i| < 1/D$ holds.

For the i^{th} component of the solution Q_k^i, r_k^i, r_{k+1}^i stores the matrix and remainder sequence representation of N_i/D after k steps of the EEA. Recall that in Remark 2.2.3 we

Algorithm 7 O.S. I.R. Method with Warm Starts

Input: A, b {$Ax = b$ is system to be solved}
Compute numerical LU factorization of A
$D := 1$ {Common denominator of approximation}
$(Q_0^i, r_0^i, r_1^i) = (I_2, 0, 1) \quad \forall i \in 1, \ldots, \dim(A)$ {Here Q_k^i, r_k^i, r_{k+1}^i represents the elements of the matrix and remainder sequence of the EEA of the i^{th} solution component after k iterations of the EEA.}
$\Delta := b$ {Error measure of solution at each step}
while solution not found **do**
  Compute $\hat{x} :\approx A^{-1}\Delta$ {Using numerical LU factorization}
  Choose an integer $\alpha < \frac{||\Delta||_\infty}{||\Delta - A\hat{x}||_\infty}$ where $\alpha\hat{x}$ is within floating point range
  Set $\bar{x} \approx \alpha\hat{x}$ {Round to the nearest integer}
  $\Delta := \alpha\Delta - A\bar{x}$ {Update the residual}
  $D := D \times \alpha$ {Update the common denominator of approximation}
  Update $Q_k^i, r_k^i, r_{k+1}^i \forall i$ using $\bar{x}_i, \alpha$ and Lemma 2.4.1
  Perform additional steps of EEA on $(Q_k^i, r_k^i, r_{k+1}^i)$ maintaining $q_{k-1}^i < \sqrt{D/2}$
  {The intermediate reconstructed solution x is defined by $x_i = p_{k-1}^i/q_{k-1}^i$.}
  Check $Ax = b$ {Use full precision check if step in loop is a power of 2, otherwise use the quick check of Lemma 2.2.8}
end while
Return: x {Solution to system}

saw that the matrix sequence stores the continued fraction convergents of a number. These values are initialized as $(Q_0^i, r_0^i, r_1^i) = (I_2, 0, 1)$. By Theorem 2.2.13 if $|N_i/D - (A^{-1}b)_i| < 1/D$ and if we update the matrix sequence maintaining $q_{k-1}^i < \sqrt{D/2}$ then either Q_k^i or Q_{k-1}^i will be a correct element of the EEA matrix sequence for the true value of the i^{th} solution component $(A^{-1}b)_i$. Thus, if $k \geq 1$, then we can be safe and backtrack to Q_{k-1}^i which will be in the matrix sequence when the EEA is applied to the numerator and denominator of $(A^{-1}b)_i$.

After performing each refinement step, two steps must be done to update the continued fraction approximation of the final solution. First Q_k^i, r_k^i, r_{k+1}^i must be updated to reflect the updated representation of the approximation N_i/D. Assuming Q_k^i is found in the matrix sequence for N_i/D we only need to update the remainders r_k^i, r_{k+1}^i. A formula to make this update is given in Lemma 2.4.1. Secondly, once r_k^i, r_{k+1}^i are updated, additional iterations of the EEA are performed to refine the matrix sequence. This is done component-wise so the progress of the EEA on each component will be different. Steps of the EEA will be performed, starting with Q_k^i, r_k^i, r_{k+1}^i, maintaining $q_{k-1}^i < \sqrt{D/2}$. After performing these

operations $x_i = p^i_{k-1}/q^i_{k-1}$ will give the continued fraction approximation of the approximate solution N_i/D with denominator not exceeding $\sqrt{D/2}$.

The following lemma gives an explicit formula for updating the remainders r^i_k, r^i_{k+1} when the approximate solution is refined.

Lemma 2.4.1. *Let $\hat{N}_i/\hat{D}$ be a rational number and suppose at the k^{th} step of the EEA the following have been computed*

$$\hat{Q}_k = \begin{pmatrix} \hat{p}_{k-1} & \hat{p}_{k-2} \\ \hat{q}_{k-1} & \hat{q}_{k-2} \end{pmatrix}, \quad \hat{r}_k, \quad \hat{r}_{k+1}.$$

Let N_i/D be a rational number for which $Q_k = \hat{Q}_k$ is known to be a matrix encountered in the application of the EEA. Then if $\frac{N_i}{D} = \frac{\hat{N}_i\alpha+\bar{x}_i}{\hat{D}\alpha}$, the following relation gives the values of r_i, r_{i+1}, the remainders encountered at the k^{th} step of the EEA when applied to N_i, D:

$$\begin{pmatrix} r_k \\ r_{k+1} \end{pmatrix} = \alpha \begin{pmatrix} \hat{r}_k \\ \hat{r}_{k+1} \end{pmatrix} + (-1)^k \bar{x}_i \begin{pmatrix} \hat{q}_{k-2} \\ -\hat{q}_{k-1} \end{pmatrix}.$$

Proof. By Remark 2.2.3, we have the following

$$\begin{aligned} \begin{pmatrix} r_k \\ r_{k+1} \end{pmatrix} = Q_k^{-1} \begin{pmatrix} N_i \\ D \end{pmatrix} &= \alpha Q_k^{-1} \begin{pmatrix} \hat{N}_i \\ \hat{D} \end{pmatrix} + \bar{x}_i Q_k^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \alpha \begin{pmatrix} \hat{r}_k \\ \hat{r}_{k+1} \end{pmatrix} + (-1)^k \bar{x}_i \begin{pmatrix} \hat{q}_{k-2} \\ -\hat{q}_{k-1} \end{pmatrix} \end{aligned}$$

which gives our proposed formula. □

This gives a way to update the remainder sequence, r_i, r_{i+1}, from $\hat{r}_i, \hat{r}_{i+1}$ without requiring to access N_i or D. The only required information is a scale factor α and difference $\bar{x}_i$, which might have a much smaller representation than N_i or D. After applying Lemma 2.4.1 to update the remainders more steps of the EEA can be performed advancing the matrix sequence and further refining the continued fraction approximation of each component N_i/D

As noted earlier, the numerator of the approximate solution, which was stored as N in Algorithms 5 and 6, is no longer explicitly stored by Algorithm 7. Instead, it is represented

by the matrix sequence and remainders which are updated at each step. From Remark 2.2.3 we see the approximation of the i^{th} solution component, represented by N_i/D, is stored as

$$\begin{pmatrix} N_i \\ D \end{pmatrix} = Q_k^i \begin{pmatrix} r_k^i \\ r_{k+1}^i \end{pmatrix}.$$

In order to check the correctness of the candidate solutions computed in Algorithm 7 the quick check given by Lemma 2.2.8 can be used at every step, this check is fast to compute but may fail to recognize a correct solution. A more expensive but always correct exact check can be done at loops that are done in a geometric progression ensuring the algorithm terminates after $O(log(S))$ loops. Within this framework there are other choices that could be made regarding how to check the solutions. A suggestion of one referee was to entirely skip the quick checks by Lemma 2.2.8 and only update Q_k^i, r_k^i, r_{k+1}^i at loops that are a power of two, or some other geometric frequency, to take advantage of asymptotically faster steps of the EEA and performing the full precision checks at these steps.

Remark 2.4.2. *If A is an $n \times n$ matrix which can successfully be numerically factored, the entries of A, b are bounded by a constant and $\log(S) = size(A^{-1}b)$, then the output-sensitive versions of the iterative-refinement methods described as Algorithm 6 and 7 will both terminate with the correct solution to $Ax = b$ after performing $O(n^3 + n^2 \log(n) \log(S))$ bit operations.*

The structure of the algorithm here mirrors the Output Sensitive Dixon's method which was analyzed in the previous section in Theorem 2.3.3, so we only note some differences here. In [137] Wan gave a proof of correctness and an analysis of his algorithm which is similar to Algorithm 5. We have stated this result as an informal remark and refer the reader to Wan's paper [137] to see how one can make a more rigorous statement of this type involving a numerical solver. The only significant difference between Algorithm 5 and its output-sensitive counterparts is how and when the rational reconstruction is performed. Moreover, by using Theorem 2.2.13 to warm start the rational reconstruction at each step Algorithm 7 will perform asymptotically the same amount of computation for rational reconstruction as Algorithm 6. In Algorithm 7 the reconstructed solution is available at

every step of the while loop and thus the quick check to certify $Ax = b$ given by Lemma 2.2.8 can be done at every loop of the refinement procedure. The more computationally expensive check will only be performed at a geometric frequency.

We note that for p-adic lifting, the idea of using warm starts for the EEA cannot be applied in the same way. Iterative refinement computes an approximate solution in a top down fashion, with each refinement making smaller and smaller adjustments leaving the leading digits unchanged. For p-adic lifting, the solution is computed from the bottom up, and the leading digits of the modular solution change at every iteration.

2.5 Computational Results

In this section we present computational results to compare the performance of the methods described in this Chapter. Source codes for the methods tested here and scripts to generate the test problems are freely available at

`www.isye.gatech.edu/~dsteffy/rational/`

for any research purposes.

2.5.1 Implementation

Output sensitive lifting can be applied in both the sparse and dense case. It can be applied in both the modular (i.e. Dixon) or numerical (iterative refinement) settings. For our computations we have chosen to evaluate it in the dense setting using both a Dixon based solver and an iterative-refinement based solver. The reason we have chosen to consider both these methods is that the Dixon based solver can be tested on some well known problems which are too ill-conditioned for a numerical solver to handle. Testing the iterative-refinement solver allows us to evaluate the warm starts within the iterative-refinement method as described in Algorithm 7 in comparison with the standard and output sensitive methods (Algorithms 5 and 6). Moreover, in Chapter 3 output sensitive lifting is tested within many methods in the sparse setting and is found to be highly successful for a large class of applied problems.

Our implementation of Dixon's algorithm is written in C/C++ and uses the FFLAS and

FFPACK packages [61], which provide fast BLAS and LAPACK routines for finite fields in C++. We implemented both the standard Dixon algorithm as described in Algorithm 3 along with the output sensitive Dixon algorithm as given in Algorithm 4. For Dixon's Algorithm any non-integral inputs are scaled to be integral before solving.

The implementation of the iterative-refinement methods are written in C and uses the BLAS [58, 98] and LAPACK [8] routines for the dense numerical linear algebra. We have used the ATLAS package [139] which provides automatically tuned BLAS and a subset of LAPACK routines. We implemented three strategies for rational reconstruction for the iterative-refinement method. First, we use the Hadamard bound as in Algorithm 5; second, we attempt reconstruction at loops which are a power of two using the framework of Algorithm 6; third, we implement a version of Algorithm 7 where the partially reconstructed solutions are stored from step to step and reused.

We use a straightforward implementation of rational reconstruction, employing a technique referred to as the DLCM method in Section 3.3.2.2 of this dissertation. This technique is also used by Chen and Storjohann [32, 33] and others [92, 131]. The DLCM method amounts to storing the LCM of the denominators of the reconstructed solution vector and using this information to accelerate component-wise reconstruction by fixing factors of the component denominators or terminating early if this common denominator grows too large. For Dixon's method we apply the DLCM method as it is described in [32, 33] and Section 3.3.2.2. For the iterative-refinement methods we use a modified strategy because the warm starts in Algorithm 7 store the work of the EEA from step to step making it incompatible with the DLCM method. The modified strategy we use is to reconstruct the candidate exact solution component-wise and keep track of the common denominator of the re-constructed components during each reconstruction step. If this common denominator of the reconstructed components exceeds the denominator bound ($\sqrt{D/2}$ where D is the denominator of the approximate solution), then no further steps of the EEA are performed and more refinement steps are performed. This gives a minor slowdown to Algorithms 5 and 6 but allows a side by side comparison between them and Algorithm 7. An recent study of vector rational number reconstruction can be found in [28].

We also comment that the purpose of our implementations were to accurately compare ideas described in this article in a straightforward implementation. The implementations are not expected to be competitive with state of the art solvers such as LinBox [60] or IML [32, 33].

2.5.2 Test Problems

The goal of our computational experiments is two-fold. First, we seek to evaluate how output sensitive lifting can accelerate linear system solving on problems for which the bitsize of the final solution is small, problems where it should have a distinct advantage. Secondly, we seek to compare the speed of the standard and output sensitive algorithms on problems whose output is very large, to verify that in the worst case there is no significant drawback to using output sensitive lifting.

In order to meet these goals and adequately compare the algorithms we chose a variety of problems in our test set. Table 2 provides descriptions of the classes of dense matrices which we use to test our methods.

Table 2: Description of Test Matrices

Matrix Type	Construction
Hadamard D_n for $n = 2^k$	$D_1 = (1)$ and $D_n = \begin{pmatrix} D_{n/2} & D_{n/2} \\ D_{n/2} & -D_{n/2} \end{pmatrix}$
Random R_n	$\{R_n\}_{ij} \in [-100, 100]$ if $i \neq j$, and $\{R_n\}_{ii} = 10,000$
Hilbert H_n	$\{H_n\}_{ij} = 1/(i + j - 1)$
Vandermonde V_n	$\{V_n\}_{ij} = i^{j-1}$
Lehmer L_n	$\{L_n\}_{ij} = \min(i, j)/\max(i, j)$

There was some difficulty in choosing which problems to consider. It is difficult to find an explicit linear system of equations for which the size of the solution meets the Hadamard bound exactly. The Hadamard matrices have determinants which meet the Hadamard determinant bound tightly $det(D_n) = 2^{n-1}$. However, when using these matrices the solution size will not be as large because the inverse matrix $D_n^{-1} = \frac{1}{n} D_n$ has small entries.

We also consider randomly generated dense matrices. For these matrices we choose the entries uniformly at random from integers with absolute value at most 100, and assign the diagonal entries all to 10,000 to ensure numerical stability. The Hilbert matrix is

a frequently cited example of an ill-conditioned matrix, and it is impossibly difficult for numerical solvers to tackle, even at low dimension. We use a type of Vandermonde matrix with the rows generated by increasing integers as described in the table. The Lehmer matrices are also a well known class of ill-conditioned matrices.

Choosing right hand sides for the systems of equations is also an important consideration. Some computational linear algebra studies use arbitrary right hand sides, such as setting b equal to the sum of the columns in A, giving a solution of all ones. While in the numerical setting, this is a perfectly reasonable right hand side to consider, it is not appropriate in our case because the algorithms studied here have run times depending on the size of the solutions. For our evaluations we will use the unit vector e_1 as the right hand side for each system. This is a reasonable choice because it corresponds to computing the first row of the inverse matrix, which should be adequately representative of the typical solution complexity.

2.5.3 Computations

Computations were performed on a Linux machine with a 2.4 GHz AMD Opteron 250 processor and 4GB of RAM. Table 3 compares the standard Dixon algorithm and the output sensitive Dixon algorithm on the entire problem set; the solve times are given in seconds. In addition to the total solution time for each method we include a profile of how time was spent in different stages of the algorithm. The solution time is divided between the following three tasks: the finite field matrix factorization, the p-adic lifting steps, and the rational reconstruction (including solution verification). Solution verification is only performed when rational reconstruction is attempted by the output sensitive methods at loops where the correctness is not guaranteed. Whenever p^i surpasses the bound B solution checks are not necessary. Due to system load and other factors solution times vary with each run, therefore these timings should be considered as approximate values. The reason times are shown to 1/100 of a second is to allow for a comparison between the subroutines. The table also includes the log of the Hadamard bound on the solution size $\log(B) = \log(2\|A\|_2^{2n-1}\|b\|_2)$, along with the actual size $\log(S) = size(A^{-1}b) \leq \log(B) - 1$.

Table 3: Solve Times for Dixon Algorithms in Seconds

Problem Details			Standard Dixon (Alg. 3)				Output Sensitive Dixon (Alg. 4)			
Matrix	$\log(B)$	$\log(S)$	**Total**	Factor	Lift	R.R.	**Total**	Factor	Lifting	R.R.& S.V.
D_{1024}	12284	10	**28.34**	0.42	27.91	0.01	**1.12**	0.43	0.07	0.62
D_{2048}	24572	11	**242.65**	2.74	239.89	0.01	**5.51**	2.71	0.29	2.51
D_{4096}	57339	12	**2273.67**	20.01	2253.61	0.05	**32.68**	21.20	1.28	10.19
R_{500}	13988	13271	**9.98**	0.19	9.63	0.17	**9.96**	0.17	9.60	0.18
R_{1000}	27988	26557	**75.51**	0.86	73.68	0.97	**75.26**	0.90	73.33	1.02
R_{2000}	55988	53137	**582.20**	4.34	571.98	5.88	**582.87**	4.43	572.35	6.09
H_{500}	1432292	1269	**5916.43**	0.17	5916.02	0.24	**6.40**	0.18	5.89	0.33
H_{1000}	5742567	2540	**204945.46**	0.84	204941.63	2.99	**84.96**	0.84	82.15	1.98
H_{2000}	22997129	5084	**-**	-	-	-	**1218.51**	4.37	1196.43	17.71
V_{100}	130944	1046	**13.72**	0.01	13.68	0.03	**0.17**	0.01	0.07	0.09
V_{300}	1474141	4079	**4049.52**	0.06	4046.52	2.93	**8.74**	0.06	4.62	4.05
V_{500}	4469528	7530	**67972.23**	0.18	67947.28	24.77	**61.02**	0.18	36.56	24.28
L_{500}	727997	3	**218.41**	0.19	218.20	0.02	**0.33**	0.19	0.01	0.13
L_{1000}	2885996	3	**3043.20**	0.88	3042.15	0.17	**1.48**	0.91	0.03	0.54
L_{2000}	11531996	3	**52873.03**	4.36	52867.53	1.14	**6.87**	4.58	0.14	2.15

The first observation we make from Table 3 is that the Hadamard bound was a very weak upper bound on the solution size on all problem classes except the randomly generated matrices. On the set of randomly generated matrices, the Hadamard bound did provide a fairly tight bound on the final solution bitsize. In these cases the output sensitive algorithm performs the same number of loops as the standard Dixon algorithm and also performs additional reconstruction attempts at intermediate steps. Even on these problems, the output sensitive Dixon algorithm has approximately the same solution times as the standard Dixon algorithm. This demonstrates that even if the Hadamard bound is nearly tight the output sensitive lifting only performs a small amount of additional computation. This occurs because at steps where incorrect solutions are generated, this incorrectness can be recognized very quickly. The first possibility is that the rational reconstruction routine aborts early if the common denominator of the reconstructed components grows too large. The second possibility is that the algorithm reconstructs an incorrect solution vector and checks its correctness by evaluating the system of equations. In this second case it will likely only evaluate a very small number of equations to recognize its incorrectness, which could be much less costly than evaluating the full system as was accounted for in the worst case complexity analysis of Theorem 2.3.3. For the remainder of the problem set the output sensitive method has an advantage of several orders of magnitude. The lifting steps that were avoided gave a significant reduction in the computational costs. We also notice on some problems with smaller solution size the cost of rational reconstruction and solution verification is larger for the output sensitive method than the standard methods due to the final solution verification.

Results for the iterative-refinement based solvers using Algorithms 5, 6 and 7 on the Hadamard and random matrices are given in Table 4. The Hilbert, Vandermonde and Lehmer matrices are were too numerically difficult for the LAPACK routines to handle so they are not included in the results.

We observe from Table 4 that the performance ratio between the standard and output sensitive lifting strategies of Algorithms 5 and 6 is similar to their related versions of the Dixon method compared in Table 3. We also observe that Algorithm 7 did not give any

Table 4: Solve Times for Iterative-Refinement in Seconds

	Lifting Strategy (Algorithm Number)		
Matrix	Std. (5)	O.S. (6)	O.S. & W.S. (7)
D_{1024}	8.57	0.91	0.92
D_{2048}	71.10	4.49	4.57
D_{4096}	965.87	27.39	27.17
R_{500}	12.12	12.07	12.46
R_{1000}	94.52	94.60	98.25
R_{2000}	763.50	756.89	798.66

improvement over the basic output sensitive lifting strategy given in Algorithm 6. The extra bookkeeping required in Algorithm 7 may be more costly than any benefits gained by saving the information for these problems. In comparison to the breakdown for Dixon's method in Table 3 for the factorization, refinement and reconstruction there was extra cost was incurred by the reconstruction steps in these algorithms due to the less efficient handling of the DLCM method that was done to allow a side by side comparison between Algorithms 5, 6 and 7.

The main purpose of these computations was to study the effectiveness of output sensitive methods, and not to compare Dixon's algorithm vs. the iterative-refinement method. However, we can make some observations about their relative speed. Before making the direct comparison we note one difference in the implementation: as described in Section 2.5.1 the implementations handle the DLCM method differently because the version of this algorithm used in Dixon's method is not compatible with the warm starts in Algorithm 7. If Algorithm 6 is adjusted to use the same version of the DLCM method as Algorithm 4 we observed that the solution times of Algorithm 6 were often faster on our test set, but not by orders of magnitude. Therefore we conclude that the iterative-refinement method can be faster than Dixon's Algorithm when both are applicable, but not by a huge margin. We also remark that tuning parameters such as how large of a prime p is used for p-adic lifting or how many digits of accuracy the numerical solver has in iterative refinement can effect the solution times. Other recent studies including [137] and Chapter 3 of this dissertation observed the iterative-refinement method to be slightly faster but not by a huge order of

magnitude.

2.6 Conclusions

Our study reinforces a conclusion that has already been observed in practice: output sensitive lifting can improve algorithms for symbolically solving systems of linear equations. We show that output sensitive algorithms can allow for systems of rational linear equations to be solved very quickly when the final solutions are small in size, while maintaining the same worst case bit complexity even when solutions are large in size. Tests were performed on several types of dense systems where output sensitive lifting was observed to give significant improvements on problems with small solution size, without noticeable slowdown even when the solution size was large.

We introduced a strategy to warm start the rational reconstruction portion of the iterative-refinement method. While this did not further improve on the other output sensitive version of iterative refinement there may be other settings in which warm starting the EEA or rational reconstruction could prove helpful.

We have primarily focused on output sensitive lifting applied to dense systems of equations; this technique is also fully applicable in the sparse setting. Our results suggest that any exact precision linear system solver relying on iterative methods should employ output sensitive lifting.

CHAPTER III

EXACT COMPUTATION OF BASIC SOLUTIONS FOR LINEAR PROGRAMMING

3.1 Introduction

The goal of this Chapter is to determine which exact methods are best suited for solving linear systems of equations arising in the solution of real-world linear-programming problems. These problems tend to be very sparse and they have been the focus of much research, due to the wide-ranging application of linear and integer programming. Until recently, software developed to solve LP problems has provided approximate floating-point solutions; commercial LP packages, such as CPLEX, attempt to find solutions within fixed error tolerances. As discussed in Section 1.2.3, an effective approach for solving LP problems exactly is to perform the simplex algorithm using inexact floating-point precision, then use symbolic computation to construct, check, and correct the final solution [13, 55, 95, 97]. This strategy is more efficient than carrying out all computations in rational precision throughout the entire simplex method. The exact solution of linear systems is a bottleneck in this procedure; solving these systems quickly can have a large influence on the solve times. Finding a fast and robust method for this setting is the objective of this study.

We will give a comparison of four solution procedures for rational linear systems. Our starting point is the LU-factorization routine developed in the QSopt [10] linear-programming code. This routine is engineered specifically for the type of sparse matrices that arise in LP applications and is based on the methods of Suhl and Suhl [133]. We adopt the QSopt routine in a direct LU-based solver, as well as an implementations of Dixon's p-adic-lifting algorithm [57] and Wan's iterative-refinement method [137]. We also consider a rational solver based on the black-box algorithm of [140]. All four methods are tested on a large collection of instances arising in the exact solution of LP problems.

The Chapter is structured as follows. The testbed of problem instances is described in

Section 3.2. The four solution methods we consider are described in Section 3.3. Results from our computational study are presented in Section 3.4, and conclusions are summarized in Section 3.5. The testbed of rational linear systems and the computer codes for the rational solvers are freely available at

`www.isye.gatech.edu/~dsteffy/rational/`

for any research purposes.

3.2 Test Instances from LP Applications

The linear-programming research community is fortunate to have several publicly-available libraries of test instances. In our study we collected these instances together into a single testbed. The set includes the instances from NETLIB [71, 117], MIPLIB 3.0 [24], MIPLIB 2003 [5], the miscellaneous, problematic, and stochastic collections of [109], the collection of [110], and a collection of traveling-salesman relaxations [12] from the TSPLIB [126]. These collections are comprised of instances gathered from business and industrial applications, and from academic studies. The problems range in size from several variables up to over one million variables.

The 695 instances in our testbed were given to QSopt_ex. For each instance that was solved by the code, the optimal basis matrix was recorded. In several cases an optimal solution was not found within 24 hours of computing time. For these examples, the basis from the last exact rational solve employed by QSopt_ex was recorded.

When examining the resulting linear systems, we found groups of instances with very similar characteristics. In these cases, we chose a representative system and deleted the other similar instances. For example, the 37 instances delf000 up to delf036 in the miscellaneous collection of [109] were replaced by the single instance delf000.

We also ran a pre-processing algorithm to repeatedly remove rows and columns having a single nonzero component. Many such examples existed in our systems, due to the inclusion of slack variables in the LP models. In the resulting collection of reduced problems, we deleted all instances having dimension less than 100.

The final problem set contains 276 instances, with dimensions ranging from 100 to over 50,000. For each instance we have both the square basis matrix and the corresponding right-hand-side vector. Within the computational results section, Table 10 includes information on the problem-set characteristics and Table 12 includes details for selected instances.

3.3 Solution Methods

3.3.1 Direct Methods

The QSopt_ex code is based on the floating-point LP solver QSopt [10], which adopts the LU-factorization methods described in [133]. We refer to the QSopt double-precision floating-point equation solver as *QSLU_double*. This solver was adapted by [66] to solve over alternate data types, including rational numbers, using the GNU Multiple Precision Library [72], and it is included in the QSopt_ex code. We refer to this rational solver as *QSLU_rational*. We created a version of the code to solve over word-sized prime-order fields using a data type with optimized operations; we call this finite-field solver *QSLU_ffield*. Figure 2 gives a performance profile comparing the speed of solving all instances in our test set using these three solvers. A performance profile plots the number of instances solved within a factor x of the fastest method time. The vertical axis represents the number of instances. The horizontal axis gives the solve-time ratios. Table 5 gives the geometric mean of the solve-time ratios, normalized by the solve time of QSLU_double.

Table 5: Relative Speed of QSLU Solvers

Solver	Time Ratio
QSLU_double	1.00
QSLU_ffield	1.05
QSLU_rational	89.72

While the double-precision and finite-field solvers are close in time, solving over the rational numbers is considerably slower. This comes as no surprise, since storing and performing operations on full-precision rational numbers is computationally expensive. This supports the idea that techniques for solving rational systems of equations that rely on fixed-precision solvers as subroutines could have advantages over direct exact methods. We

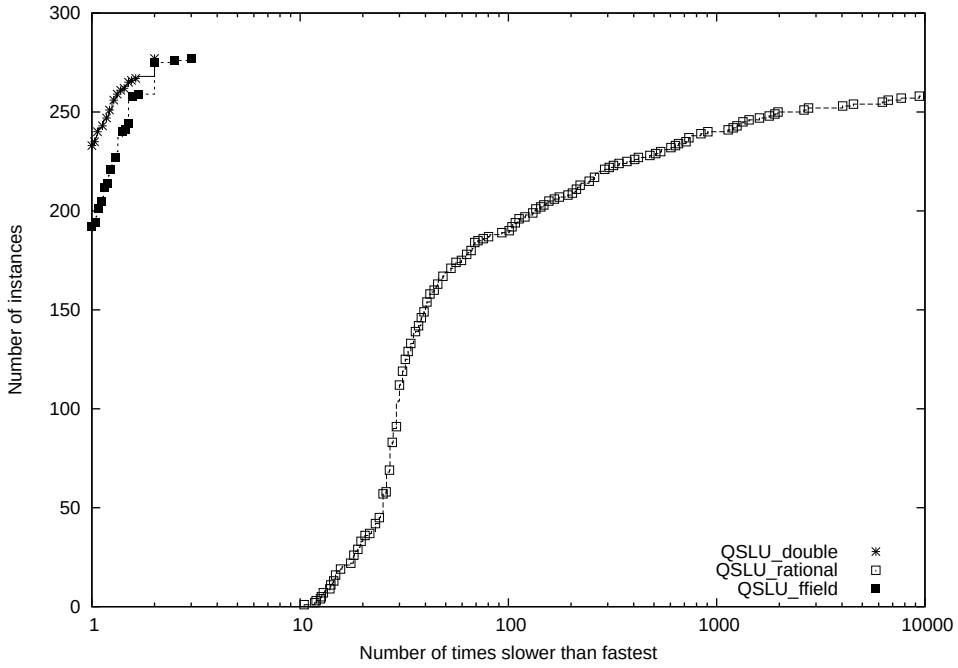


Figure 2: Comparison of QSLU Solvers

also tested a direct rational solver using the conjugate gradient method and the GMP library, programmed by Sanjeeb Dash of IBM, and found it much slower than the rational LU-factorization method.

There were many characteristics of individual problems which influenced the deficit in speed between the double-precision solver and the exact rational solver, including the dimension and the complexity of the solutions. The ratio of 89.72 presented in the table can be seen as a characteristic of our problem set and a bound on how much we could possibly hope to improve our speed over the rational solver by using the double-precision and finite-field solvers as subroutines. From the performance profile we also observe that the finite-field solver was the fastest solver on some instances. This may seem surprising as finite-field operations are generally slower than double-precision operations; one possible explanation is the simplification of some operations in the code afforded by exact finite-field computation over numerical computation, such as checking if an element is zero.

Direct exact-precision methods are not usually thought of as being among the fastest methods for solving systems of linear equations exactly. However, we experienced a reasonable level of success in the sparse setting with QSLU_rational, in some cases outperforming the other methods presented in this Chapter. A considerable amount of the computational effort of a sparse LU solver is spent finding permutations of the system to reduce the fill-in and to reduce the number of arithmetic operations that must be performed. Such computation depends only on the nonzero structure of the matrix, and this helps QSLU_rational avoid many full-precision arithmetic operations. In the dense setting, we expect a larger performance gap between a direct rational solver and fixed-precision solvers. For dense systems, BLAS routines [58, 98] can be used for fast floating-point linear algebra, and [61] have introduced a comparably fast system for dense linear algebra over finite fields. These dense routines were employed in the dense solvers described in Chapter 2.

3.3.2 Rational Reconstruction

Section 2.2.1 of Chapter 2 gives a description of techniques for rational number reconstruction and the two main results, Theorem 2.2.2 and Theorem 2.2.1, will be used in this Chapter. The polynomial-time algorithms associated with these theorems are based on finding selected continued fraction convergents using the Extended Euclidean Algorithm (EEA). Section 5.10 of [136] gives a description of rational reconstruction, including numerical examples, another description appears in [91]. The EEA appears as Algorithm 3.6 in [136] and is also discussed in Section 2.2.1 of this dissertation. Some methods have been studied to accelerate rational reconstruction, including [94, 103, 113, 123, 124]. In our implementation, we use a technique, sometimes referred to as Lehmer's GCD algorithm, to accelerate our computations (see Algorithm 1.3.7 in [37]). The EEA involves successively performing integer divisions. Lehmer's algorithm accelerates the EEA computationally by performing the integer divisions on approximations of the numbers instead of on large integers. Specifically, in our routines we replace extended-precision integer division with floating-point number inversion when possible, carrying out several steps of the EEA based on truncated data and then synchronizing and updating the full-precision data. We found this to speed up the

rational reconstruction by at least a factor of two, and more with large inputs. [38] also used Lehmer's algorithm to speed up rational reconstruction and experienced comparable levels of success.

3.3.2.1 Reconstruction Bounds

The *bitsize* of a nonzero rational number p/q is $log(|pq|)$, and the bitsize of a rational vector is defined to be the maximum bitsize of its components. In Chapter 2 we gave a related definition of *size* of a vector as being the maximum bitsize over all components after representing the vector with a common denominator. For a given instance, the exact bitsize of the solution is not known without solving the system, but it can be bounded using Cramer's rule. Cramer's rule states that for a square nonsingular system $Ax = b$, the i^{th} component of the solution vector is determined by $x_i = \frac{\det(A_i)}{\det(A)}$ where A_i is constructed by replacing the i^{th} column of A with b. Computing determinants exactly is computationally expensive, so the Hadamard bound is typically used to provide an upper bound. The Hadamard determinant bound states that $det(A) \leq \|A\|_2^n \leq n^{\frac{n}{2}} \|A\|_{\max}^n$. This gives $log(\|A\|_2^{2n-1}\|b\|_2)$ as a bound on the bitsize of x. Specifically, $B_n = \|A\|_2^{n-1}\|b\|_2$ and $B_d = \|A\|_2^n$ give upper bounds on the numerator and denominator of the solution of $Ax = b$ that are valid for Theorems 2.2.1 and 2.2.2. Table 6 shows the bound on solution bitsize generated by the Hadamard bound, along with the actual bitsize of the solution, for several of the larger instances in our test set.

Table 6: Actual Solution Size vs. Hadamard Bound

Problem	Solution Bitsize	Hadamard Bound
cont11_l	1263	4570016
gen1	439931	1046612
momentum3	159597	11521199

The examples in Table 6 illustrate that the bitsize of the solution can be much lower than the Hadamard-based bound. This suggests that computation of modular and floating-point solutions based on the Hadamard bound can lead to unnecessary computation and memory use. For the problem cont11_l, computing an approximate solution with 4,570,016

bits for each component would require over 31 gigabytes of memory for storage alone, when a solution with 1,000 times fewer digits gives sufficient information to successfully reconstruct the exact solution. In Chapter 2 we saw that using the Hadamard bound could result in very slow solution times. For many of the problems in our test set of LP problems performing computations using this bound would not even be a possibility as it would cause the algorithms to exhaust the computer memory. For a more in depth investigation into the tightness of the Hadamard bound, see [1, 2].

As an alternative to the Hadamard bound, we can use smaller but possibly incorrect bounds, then verify the results that are obtained. In this scheme, we attempt rational reconstruction on an approximate solution corresponding to the guessed bound, and check the resulting exact solution for correctness. If it is correct, we can terminate, and otherwise we increase the bound and repeat. Correctness of a candidate solution can be easily certified by evaluating the linear equations. In [32, 33], this technique is referred to as *output-sensitive lifting*. This technique is used in a different context by [62] where it is referred to as early termination. This method is made especially practical because while computing high-precision solutions by iterative methods, less-precise solutions are encountered at intermediate steps without any extra computation, giving an opportunity to try rational reconstruction. Chen and Storjohann also provide a simple formula to certify solutions obtained via modular rational reconstruction without evaluating all equations. A detailed treatment of output sensitive lifting can be found in Chapter 2.

3.3.2.2 Vector Reconstruction

Reconstructing the solution vector of a system of equations can be achieved by applying Theorem 2.2.1 or Theorem 2.2.2 component-wise to approximate or modular solution vectors. Considering information from the entire system of equations can lead to faster methods for reconstructing a solution vector. We discuss two such techniques, one using the relationship of the denominators of the solution components to accelerate rational reconstruction, the second using the equations to deduce some values without reconstruction.

For many systems of equations, the denominators of the components of the solution

vector share common factors. The first method we look at exploits this situation. This method is discussed in [33, 92] for use in modular rational reconstruction and we call it the *DLCM*. The DLCM technique for modular reconstruction is well known and currently used in other software such as LinBox and NTL [60, 131]. Let Δ be the least common multiple of the denominators of the components that have been reconstructed so far. Suppose the next component of the solution we reconstruct is p/q, from n, M, B_n, B_d, as in Theorem 2.2.2. Compute $n' = \Delta n \bmod M$, then reconstruct p'/q' from n', M using bounds $B_n, \lfloor B_d/\Delta \rfloor$, and assign

$$\frac{p}{q} := \frac{p'}{q'\Delta}.$$

Fixing Δ as a factor of the denominator, and then reconstructing the remaining factors of the denominator and the numerator, accelerates the routine because rational reconstruction terminates faster when the denominator bound B_d is lower. In fact, if q divides Δ then p/q can be immediately identified by the rational-reconstruction routine without any steps of the EEA. It is possible to reduce M to a value $M' \geq M/\Delta$, as described in [92], to further accelerate this procedure.

The DLCM technique can also be applied to accelerate floating-point rational reconstruction. Suppose a component of the solution p/q is to be reconstructed from an approximation α, and a common denominator Δ of other components is known. Rational reconstruction is applied to find a rational number p'/q' that best approximates $\Delta\alpha$ with denominator less than B_d/Δ, and then the assignment

$$\frac{p}{q} := \frac{p'}{q'\Delta}$$

can be made. Again, by assigning some factors of the denominator ahead of time, we reduce the calculations in the rational-reconstruction routine.

We mention one possible drawback of this technique. For DLCM to work correctly, the bound B_d must be an upper bound on the size of the common denominator of the entire solution vector, while component-wise rational reconstruction only requires B_d to be a bound on the individual denominators of the solution vector's components. The Hadamard bound given in the previous section will always bound the common denominator. However,

a smaller bound that is valid for each component individually, but less than the common denominator of the components, can cause this technique to fail. From the statements of the theorems, we see that increasing the bound B_d necessitates the computation of approximate solutions with more digits of accuracy, or solutions modulo a larger number. The following example illustrates this possible drawback.

Example 3.3.1. *If the solution to $Ax = b$ is $x = (1/2, 1/3, \ldots, 1/p_n)$ where p_n is the n^{th} prime number, then a bound of $B_d = p_n$ will suffice for component-wise rational reconstruction. However, the DLCM method requires the bound B_d to be at least $2 \times 3 \times \cdots \times p_n$ to terminate properly.*

Despite the possible drawback highlighted in Example 3.3.1, this characteristic could also prove advantageous in an output-sensitive lifting algorithm. Suppose output-sensitive lifting is applied with some insufficiently large bounds B_n, B_d. The DLCM method would recognize failure and terminate early after the common denominator Δ grows larger than B_d. Therefore, that rational-reconstruction attempt with a bound too small to determine the actual solution would terminate before reconstructing every component of the solution vector. This can avoid a significant amount of computation that would otherwise be spent in failed vector reconstruction attempts. This strategy is used by [33] to reduce the overall time spent on rational reconstruction in their rational solver. When reconstructing a vector component-wise a similar early stopping criterion can be set by maintaining a common denominator of the reconstructed components and terminating if it becomes larger than B_d. This strategy is also used in the implementation in Chapter 2.

The second technique we explored is the use of the equations from the system to deduce some components of the solution vector. Once part of the solution vector is reconstructed, it is possible that the known components, along with the equations, will directly imply the values of unknown components. If the equations are sparse, evaluating an equation to determine the exact rational value of an unknown solution component could be faster than performing rational reconstruction to determine that component. We call this method of reconstructing some components and then deducing all implied components the *ELIM*

technique. To apply this technique, the primary challenge is to determine an order in which to consider components for reconstruction, and to determine which equations to use for deducing values.

A matrix A is said to have *lower bandwidth* of p if $a_{ij} = 0$ whenever $i > j + p$ [74]. The lower-bandwidth minimization problem is the problem of performing row and column permutations on a matrix to minimize its lower bandwidth. If the $n \times n$ matrix A defining a system of equations has lower bandwidth of p, and the final p components of the solution vector are known exactly, then all remaining components can be determined by solving a $n - p$ lower-triangular system of equations by backwards substitution. Determining the minimum number of components of the solution vector that must be reconstructed, in order to deduce the remaining portion of the solution vector from the equations, is equivalent to the lower-bandwidth minimization problem.

We use a greedy heuristic, Algorithm 8, to determine the variable ordering. The algorithm partitions the columns of A into a set R and an ordered list E. Variables corresponding to columns in R will be reconstructed and variables corresponding to columns in E will be deduced using equations from the system. A list $L(i)$ for $i \in E$ is constructed such that $L(i)$ gives an index to a row of A that has a nonzero element in its i^{th} column and has zeros in every column j appearing after i in the list E. Thus, $R \cup E$ gives an ordering to reconstruct the variables, where every variable in R is obtained by rational reconstruction and each variable i in E can be deduced using constraint $L(i)$ and the preceding variables. We use A_j to denote the j^{th} column of A, and we use a_i to denote the i^{th} row of A. The algorithm is described in terms of deleting rows and columns of the matrix; after such deletions, we maintain the original labeling of the remaining rows and columns.

We found this heuristic effective in reducing the number of variables to be reconstructed. Table 7 shows the number of variables that could be eliminated by the routine, that is, the number of variables in the list E.

To illustrate the overall effectiveness of the DLCM and ELIM methods, Table 8 compares the solve times and loop count of an exact solver based on Dixon's method (introduced later) on our entire problem set. Details of the output-sensitive lifting used can be found in

Algorithm 8 Variable Ordering Algorithm

Input: Matrix A {From $Ax = b$}
Initialize: $R = \emptyset$, $E = \emptyset$
while $A \neq \emptyset$ **do**
  Remove any all zero rows a_i from A
  if $\exists i$ such that a_i has a unique nonzero element a_{ij} **then**
    $E := E \cup \{j\}$ {Variable j can be eliminated}
    $L(j) := i$ {Implied by constraint i}
    Remove A_j,a_i from A
  else
    Choose A_j with the maximum number of nonzeros
    $R := R \cup \{j\}$ {Mark column with most nonzeros for rational reconstruction}
    Remove A_j from A
  end if
end while
Return: R, E, L

Table 7: Percent of Variables Eliminated

Elimination %	Instances (out of 276)
70%+	219
80%+	145
90%+	66
95%+	35

Section 3.4.1. The solve times are expressed as geometric means of the ratios with the time needed for Dixon's method using component-wise rational reconstruction. It also shows the geometric mean of the ratios of how many loops each method performed to achieve the final solution. The ratios presented here compare the total solve times, of which the reconstruction is just a part. This measure is used to consider the variation in solve times because the ELIM method and the component-wise reconstruction are in some cases able to construct a solution with less information than the DLCM method. We also tested a combination of the two techniques, listed as "DLCM and ELIM" in the table. In this case we applied the ELIM routine as it is described but applied the DLCM method to reconstruct the first p components of the solution vector.

From this table we observe that both methods reduce the overall solve time by approximately 60%. The loop ratio here indicates how many loops of p-adic lifting were required

to identify the correct solution by using output-sensitive lifting. As we can see, the DLCM method required a geometric average of 16 % more loops than the component-wise reconstruction, indicating that the phenomenon shown in Example 3.3.1 does occur to some degree, although much less dramatic than the worst case. Despite these extra loops, the huge speedup and possibility of early termination gained at each reconstruction by the DLCM method, over the component-wise computation, still allows the DLCM to finish much faster. We also see that the ELIM method is able to finish in fewer loops than the component-wise reconstruction in some cases; this would occur when some of the components that are deduced by the elimination routine have representation too large to have been reconstructed component-wise. When combining DLCM and ELIM we see that the number of loops again increases as the DLCM computation terminates the procedure early, resulting in a slight slowdown over the pure ELIM routine.

Table 8: Improvement Using Vector Reconstruction

Vector RR Method	Solve Time Ratio	Loops Count Ratio
Component-wise	1.00	1.00
DLCM	0.39	1.16
ELIM	0.40	0.97
DLCM and ELIM	0.41	1.15

From these tests see that the DLCM method is fastest for our set of instances. We therefore use DLCM throughout our modular and floating-point rational-reconstruction routines for the remainder of the Chapter. Note, however, that the ELIM method is nearly as fast, and on certain classes of sparse problems it may be faster. Dixon's method uses modular rational reconstruction, but in our tests we noticed similar acceleration of the floating-point reconstruction routine using these techniques.

We also performed tests comparing the speed of the component-wise and DCLM methods for reconstructing solution vectors, excluding the time required to solve the linear system. To make this comparison we considered the solution vector from each problem. We determined a bound for the component-wise reconstruction to terminate correctly, which was

$B_1 = 2B^2$ where B is the largest numerator or denominator in any component of the solution. We also computed a bound valid to reconstruct the vector using the DLCM method, which is $B_2 = 2B^2$ where B is the largest of the numerator in any component of the solution or the least common denominator for the solution vector, whichever is larger. We found that on 129 out of 276 problems the bounds were the same, and that the geometric average of $\log(B_2)/\log(B_1)$ was 1.21, which corresponds to the loop ratio in Table 8. One would expect this number to match the 1.16 in Table 8. The difference occurs because the table is generated by timings on our Dixon solver which uses an output-sensitive lifting scheme that will not attempt rational reconstruction at every loop. When the bounds B_1, B_2 are close to each other the solvers will often finish at the same time, even if the bounds are not equal. As we are taking the geometric mean in both cases this explains the gap; the arithmetic mean would be expected to be the same.

As we saw in Table 8, DLCM was still a faster overall strategy despite these extra loops, but there were some instances where this ratio was quite large and effected speed. On some problems this ratio was as high as 66, and the Dixon solver using output-sensitive lifting and component-wise reconstruction was over 100 times faster than the DLCM-based Dixon solver. We also considered the time required to reconstruct the solution by component-wise reconstruction at B_1, divided by the time to use DLCM vector reconstruction at bound B_2; we found the geometric mean of these ratios to be 7.80. To have a pure direct comparison of these we also compared component-wise reconstruction and DLCM vector reconstruction both at B_2 and found their ratio to be 8.92, and found DLCM to be as much as 75 times faster on individual problems. These final numbers indicate that, although component-wise reconstruction can be faster on some problems, the DLCM method can provide a huge speedup overall.

3.3.3 Iterative Refinement

Applying Theorem 2.2.1, a rational system of equations can be solved exactly given a bound on the size of the denominators of the solution and an approximate solution within a required degree of accuracy. When the number of digits of accuracy required of the approximate

solution is large, solving the system in extended-precision floating-point arithmetic can be as slow as solving the system directly in rational precision, or slower. The iterative refinement procedure allows us to use repeated approximate floating-point solves to construct an extended-precision solution, taking advantage of the speed of a floating-point LU solver.

Iterative refinement is the process of finding and refining an approximate solution. Once a system is solved approximately, the exact error of the approximate solution can be determined, and further approximate solves can be used to help correct the error. Repeating this process gives solutions that are increasingly accurate. State of the art floating-point solvers typically perform iterative refinement to refine a double-precision solution so that the backwards error is close to machine epsilon.

Ursic and Patarra [134] adopted iterative refinement to obtain high-accuracy approximate solutions, and combined this with rational reconstruction to solve linear systems of equations exactly. Wan [137] introduced an improved version of this algorithm, reducing the number of extended-precision operations that are required. Wan's method works with systems of equations that are integer; rational systems are handled by scaling the entries to be integral. We have given a detailed description of Wan's Algorithm as Algorithm 5 in Chapter 2.

For this Chapter, we implemented a version of iterative refinement using Wan's strategy and the solver QSLU_double. Incorrect choices of α can quickly cause the algorithm to fail, so computing an error measure of the approximate solution $\hat{x}$ at each step to guide the selection of α is necessary. Note that scaling a rational problem to be integral can create difficulties for numerical LU-factorization solvers, since some entries can become very large. This problem is avoided by performing the numerical LU factorization on the original unscaled form of the problem, then using the scaled integral matrix only in the refinement steps of the algorithm.

We will later show that iterative refinement and rational reconstruction is a very effective method for solving systems of equations exactly, often performing the fastest on our test set. The drawback to this method is its vulnerability to numerical difficulties with floating-point computations. We did experience some trouble on a small subset of examples that were

numerically unstable; other methods considered in this Chapter do not share this problem.

3.3.4 Dixon's Method

Dixon's method [57, 138] for solving exact rational systems of equations relies on Theorem 2.2.2. This algorithm is stated in terms of integer systems of equations, so we first scale a rational system to be integer. In order to determine a solution modulo a large number M, [57] uses the p-adic-lifting procedure, which constructs a solution modulo p^k by successively solving systems of equations modulo p. Algorithm 3 gives a detailed description of this algorithm.

In our implementation, we use QSLU_ffield for the finite-field solves. For a nonsingular integer matrix A, A mod p is nonsingular for a prime p if and only if p does not divide $det(A)$. Instead of computing $det(A)$ to guide the choice of p, we can guess different primes until an LU factorization is successful. The prime p is chosen small enough so that all numbers can be stored as machine-precision integers.

The p-adic-lifting procedure can be thought of as an analogue to the iterative-refinement method, since they both use fixed-precision solve routines iteratively to build extended-precision solutions. One advantage Dixon's method has over iterative refinement is that the finite-field elements are stored exactly, leaving no chance for numerical problems when performing calculations. For further detail and analysis of the complexity of Dixon's method see Chapter 2 or [32, 33, 57, 114].

3.3.5 Wiedemann's Method

Wiedemann's method for solving systems of equations over finite fields was introduced in [140]. We say that a sequence $\{a_i\}$ is *linearly generated* if there exists $c_0, c_1, \ldots c_m$ such that $\forall k \geq 0$, $c_0 a_k + c_1 a_{k+1} + \ldots c_m a_{k+m} = 0$. The polynomial $f(x) = c_0 + c_1 x + \ldots c_m x^m$ with c_m normalized to 1 is called the *minimum polynomial* of the sequence. Wiedemann's method uses the fact that, over a finite field, the sequence $I, A, A^2, A^3, \ldots$ is linearly generated. His method calculates the minimum polynomial of this sequence (or of the sequence $\{A^i b\}_{i=0}^{\infty}$) using a randomized algorithm based on the Berlekamp-Massey algorithm [22, 106]. This

gives an explicit formula for solving $Ax = b$:

$$c_0 I + c_1 A + c_2 A^2 + \ldots c_m A^m = 0,$$

$$c_0 A^{-1} b + c_1 b + c_2 Ab + \ldots c_m A^{m-1} b = 0,$$

$$A^{-1} b = -c_0^{-1}(c_1 b + c_2 Ab + \ldots c_m A^{m-1} b).$$

In both the computation and evaluation of the minimum polynomial, access to the matrix is only needed as a matrix-vector multiplication oracle. Therefore, Wiedemann's method is referred to as a black-box algorithm, and it is particularly suited for working with sparse matrices. A presentation of this technique is given in Section 12.4 of [136].

Dixon's method for solving rational systems of equations, given in Algorithm 3, relies on a finite-field solve routine for each p-adic-lifting step. Replacing QSLU_ffield with Wiedemann's method gives an alternative approach to solving systems of equations exactly. A more detailed description of Wiedemann's method applied to solving rational systems of equations, including complexity analysis, is given in [92].

3.4 Computational Results

3.4.1 Implementation

We tested four methods in the C programming language to solve rational linear systems of equations: QSLU_rational, iterative refinement, Dixon, and Wiedemann. Figure 3 gives a flowchart showing the relationship of these four methods. The rational-reconstruction routines used in the methods share a common structure, using the techniques detailed in Section 3.3.2. We implemented fast finite-field operations, storing the elements as integers, pre-computing inverse tables, using delayed modulus computation, and using floating-point operations to accelerate multiplications. These techniques are standard and they are employed by other software packages such as [60, 131].

When used, we attempted rational reconstruction with a frequency relative to the number of loops in the iterative refinement/p-adic-lifting procedure. In [33], rational reconstruction is attempted every 10 loops. After some experimentation, we found it effective to attempt rational reconstruction with a geometric frequency; we choose specifically to

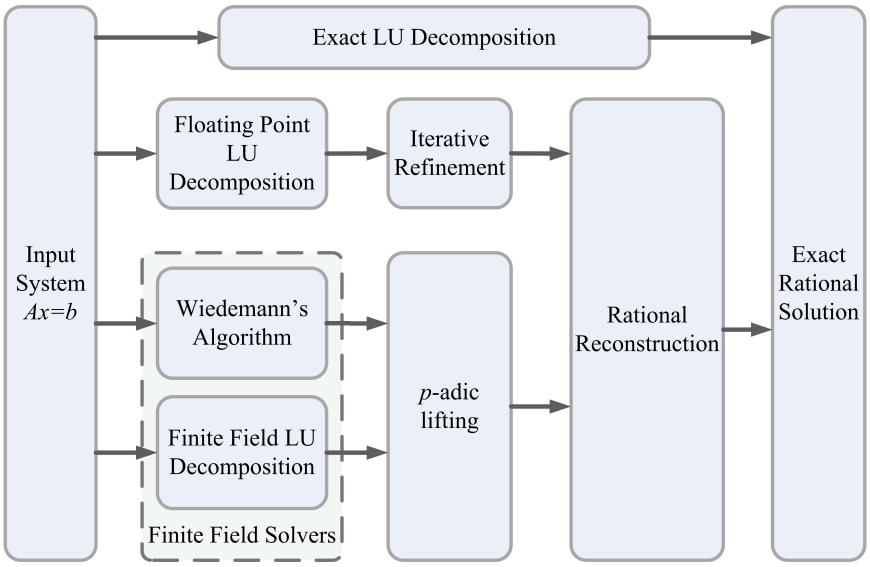


Figure 3: Relationship Between Algorithms

attempt reconstruction when the loop number is a power of two. Using this strategy, if the approximate solution after k loops is sufficient to reconstruct the rational solution, then at most $log(k)$ unsuccessful attempts are made. We remark that if rational reconstruction is attempted before enough loops are completed to accurately reconstruct the solution, this can be recognized without reconstructing the entire vector, as mentioned in Section 3.3.2.2. Therefore, failed attempts at reconstruction will stop before reconstructing each component and thus have a considerably lower computational cost than the final reconstruction step in which each component is reconstructed. Despite this fact we still found it computationally faster to attempt reconstruction less frequently. A more detailed analysis of our strategy for output-sensitive lifting can be found in Chapter 2. We also modified the geometric strategy slightly by imposing a maximum number of loops between reconstruction attempts, giving improved solve times for our problem set. A similar strategy is also employed and analyzed by [31] for solving linear systems over cyclotomic fields. A description and analysis of output-sensitive lifting applied to determinant computation can be found in [90]. A probabilistic output-sensitive lifting strategy is used in [65] where it is applied to the computation of integer hulls.

To verify the competitiveness of QSLU_double, which was used as a subroutine in iterative refinement, we compared it with the well-known solvers Pardiso [128] and SuperLU [51]. In Table 9, it can be seen that the QSLU_double code was faster on average over our testbed of LP instances, although Pardiso and SuperLU were faster on some individual problems. We may not expect QSLU_double to outperform Pardiso and SuperLU on other classes of instances, as it was developed using a method engineered specifically to solve very sparse bases arising in the solution of LP problems. In our use of QSLU_double, we perform two refinement steps in double precision to improve the double-precision solution. SuperLU contains a similar refinement scheme. We measured the backward relative error of the final solutions and found SuperLU to produce more accurate solutions, with relative backward error average of 1.27e-16, compared to 1.58e-15 for QSLU_double. The backward relative error of a solution x is defined to be $\max_i \frac{|(Ax-b)_i|}{\sum_j |A_{ij}x_j|+|b_i|}$. We found Pardiso to achieve comparable errors on many instances, but we experienced numerical difficulties on some examples, leading to unsatisfied constraints. The performance of the Pardiso and SuperLU codes are compared with other numerical solvers in a nice computational study by [76], covering symmetric systems.

Table 9: Numerical Sparse LU Solvers

Solver	Time Ratio
QSLU_double	1.00
SuperLU	2.36
Pardiso	2.50

We also compared our Wiedemann-based solver with the Wiedemann solver found in LinBox 1.1.6 [60]. On the instances both codes completed, we found our new code to be 6.91 times faster by geometric mean, with ratios ranging from 0.13 to 17.30. We also compared our Wiedemann finite-field solver against the LinBox Wiedemann finite-field solver, which are used as subroutines in the rational solvers, and found our solver to be 1.84 times faster, with ratios ranging from 0.013 to 3.50. A referee performed similar experiments on a smaller subset of our problems and observed our Wiedemann rational solver to be 2.72 times faster

and our Wiedemann finite-field solver to be on average 62% slower on a 3.4Ghz Intel P4 computer. These results suggest that the speedup in our rational solver comes from the rational-reconstruction techniques employed in our implementation. The purpose of this comparison is simply to demonstrate that our code is reasonably fast; LinBox is a much larger and more general software package written in C++, and our software was tuned to be as fast as possible on our specific problem set.

3.4.2 Rational System Results

Dixon's method, QSLU_rational, and Wiedemann's method were all able to solve all instances in our test set to completion. Iterative refinement was able to solve all but 5 of the problems, which failed for numerical reasons. The problems where iterative refinement failed were: `cont11_l`, `pilot`, `rat7a`, `de063155`, and `de063157`. Solve times varied greatly, ranging from fractions of a second to days. In Table 10 we present a comparison of the solve times over all instances. Computations were performed on linux-based machines with 2.4GHz AMD Opteron 250 processors and 4 gigabytes of RAM. To avoid the slower instances outweighing all others, we normalized all solve times by dividing by the time for Dixon's method. We then computed the geometric means over the entire set of instances and also over selected subsets. Using the geometric mean instead of the arithmetic mean helps to prevent the results from being skewed by outliers. The five instances where iterative refinement failed are omitted from the averages in that column. The partitions of the problem set are based on the dimension of the instances, the bitsize of the final solutions, and the density of the instances, taken as the average number of nonzeros per row.

An immediate observation is that on the entire problem set, iterative refinement is the fastest method, followed by Dixon's method, which is nearly as fast. QSLU_rational is more than 3 times slower on average, and Wiedemann's method is on average nearly 40 times behind. By considering the various subsets of instances, we can identify patterns concerning the effect of problem characteristics on the solve times. One observation is that the dimension of the problem has the most significant relative effect on the Wiedemann method. The larger instances can have a minimum polynomial of a higher degree, requiring

Table 10: Geometric Means of Relative Solve Times

Problems			Solver Time Ratios			
Subset of Probs.		Size of Subset	Dixon	Iter. Refine	QSLU_rational	Wied.
All Problems		276	1.0	**0.861**	3.247	38.370
Dim.	100-300	79	1.0	**0.859**	5.228	12.853
	300-1,000	98	1.0	**0.835**	2.787	24.664
	1,000-10,000	84	1.0	**0.889**	2.654	106.924
	10,000+	15	1.0	**0.892**	2.219	703.087
Sol bitsize	0-100	92	1.0	**0.993**	10.744	36.892
	100-1,000	79	1.0	**0.911**	4.069	57.008
	1,000-10,000	55	1.0	**0.708**	0.971	39.443
	10,000+	50	1.0	**0.751**	0.949	21.405
Nz./row	2-3	84	1.0	**0.966**	2.314	61.571
	3-5	99	1.0	**0.814**	2.352	34.924
	5-10	68	1.0	**0.825**	4.982	32.789
	10+	25	1.0	**0.811**	11.342	17.433

huge numbers of matrix-vector multiplications to perform the finite-field solves.

We note that Dixon and iterative refinement have the best advantage over the QSLU_rational code on the instances with smaller solution bitsize; problems with a small solution bitsize can be computed with only a few steps of refinement/p-adic lifting. Relative to Dixon and iterative refinement, QSLU_rational becomes slower as the density increases, presumably because the LU factorization has more fill-in and computation, which is relatively more expensive using rational arithmetic. Wiedemann's method becomes relatively faster as the density increases; this is likely because increased density gives more work to the LU factorizations used by the other methods.

In Figure 4 we give a performance profile comparing the four methods on the full problem set. In this profile we can observe the close performance of Dixon and iterative refinement, the lag in speed of the QSLU_rational method, and the significantly slower speed of Wiedemann's method. We note the sharp edge in the curves for Dixon and iterative refinement, near the top where they cut far to the right quickly. This is caused by a small group of instances on which QSLU_rational is faster by a significant amount. Some the instances where QSLU_rational has a speed advantage are those having large solution bitsize; unfortunately

the bitsize of a solution is not available before solving to aide in choosing a method.

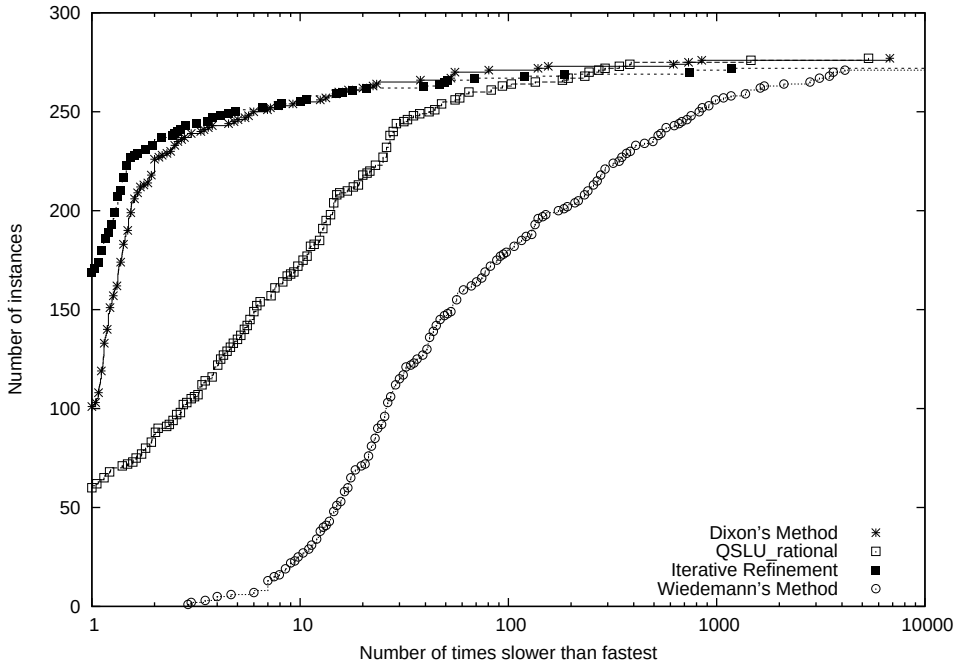


Figure 4: Comparison of Rational Solvers

Another important observation we can make from our experiments is how each of the methods balances time between their internal subroutines. Table 11 provides profiling data showing how the time was spent by each method, excluding QSLU_rational. This table was generated by considering the percent breakdown of the various stages of the algorithm by each method, for each instance, then the (arithmetic) average was taken over all instances. By *residual computation* we mean the time spent within each loop calculating the new right hand side for which the fixed precision solve will be computed and the radix conversion of the intermediate solution. Dixon and iterative refinement spent similar portions of their time on the LU factorizations and rational reconstruction (including the time for the solution verification), with a variation of time in their inner loops. The backsolves were faster for iterative refinement, but the computation of the residual for lifting was more expensive due to the additional work to compute the scaling factor α.

We note that for Wiedemann's method, the largest portion of time was spent doing the first solve and then successive backsolves over finite fields; the large number of matrix-vector multiplications that must be done in these stages is relatively much slower than the LU factorization and solves. This table suggests that if Dixon's method and iterative refinement are to be made faster computationally, deeper investigation into accelerating rational reconstruction could be helpful, as it occupied nearly half of the solve time for these methods on average. The large portion of time spent on rational reconstruction may come as some surprise when one considers the worst-case complexity analysis of Dixon's method and iterative refinement, which do not have rational reconstruction as the dominating factor. We believe this is explained by the extreme sparsity of our matrices, which allows for faster than predicted computation in other portions of the algorithm, while not reducing the required computation for rational reconstruction. When comparing this to the results in Chapter 2, we can observe that the computations performed on the dense systems used a larger portion of their computation on the lifting steps and less on the rational reconstruction and solution verification. We can also note that due to the extreme sparsity of our test matrices the time required to verify a solution by verifying that it satisfies all of the equations is also relatively much less expensive than the corresponding solution verification performed on dense systems.

Table 11: Profile of Time Spent

Solve Component	Dixon	Iter. Refine	Wiedemann
Factorization/First Solve	11.9 %	10.1 %	23.7 %
Backsolves	15.5 %	4.5 %	61.3 %
Residual Computation	25.6 %	39.8 %	7.7 %
Rational Reconstruction	47.0 %	45.6 %	7.3 %

Finally, Table 12 provides solve times and detailed information for select instances.

3.5 Conclusions

The results of this computational study provide a picture of how rational solver methods perform on a test set of very sparse real-world instances arising in linear-programming

applications. By using variations of the QSopt factorization code and using common rational reconstruction strategies, we give a side by side comparison of these methods.

There are several conclusions we can make from our computations. The two methods we found to be the fastest were Dixon's method and iterative refinement. These two methods perform repeated fixed-precision solves to obtain high-accuracy solutions and apply rational reconstruction. Iterative refinement is approximately 15% faster overall, but Dixon's method has an advantage in numerical stability. This agrees with the conclusion of Wan [137] that his method is faster than Dixon's method on well-conditioned matrices and with the general knowledge that Dixon's method should be faster than a direct elimination method using rational arithmetic. Our conclusions also agree with a computational result in [63], who found that Wiedemann's method could be slower than direct elimination techniques on structured sparse matrices.

For such a speed difference we find Dixon's method to be the most attractive method for our application of exact-precision linear programming. An exact LP solver can call the exact linear system solver many times, making robustness very important. In other application areas, iterative refinement might be more attractive, especially if the systems are known to be numerically stable. We note also that in some exact LP solution schemes, a double-precision LU factorization of the basis matrix may be available at the end of a call to the simplex method. In such cases, the factorization can be used in the steps of the iterative-refinement method, resulting in a substantial savings in time. In our tests, the QSLU_rational code is faster than the other methods on a small subset of the instances. If multiple processors are available, a reasonable strategy is to run Dixon's method on one processor, and QSLU_rational on another.

We found Wiedemann's method to not be attractive for our LP test instances. Its black-box nature apparently does not make it competitive in this setting, as the LU factorizations for these very sparse problems can be computed very quickly. We believe that the QSLU codes benefited not only from the extreme sparsity of the matrices but from their structure as well. For other classes of sparse matrices for which LU factorizations are not possible without significant fill-in, we would expect Wiedemann's method to perform more competitively. In

such cases dense solvers might also be competitive, especially on the problems with smaller dimension. In a study by [64] tests were performed on randomly generated sparse systems and it was found that an efficient dense solver was often able to beat their sparse solver unless the dimension was very large.

We have tried to use uniform standards as much as possible between our codes in order to give a fair comparison of the methods we are evaluating. However, there is always room for improvement in any implementation. We will make some comments on several improvements that could be made and how they could influence the results of our study. We thank the referees for some excellent suggestions in this light. The iterative solvers could benefit from better strategies for radix conversion, such as those used by [33] in their IML software; this strategy would take advantage of the asymptotically fast multiplication algorithms in GMP. Other gains for the iterative solves could come from implementing asymptotically faster strategies for rational number reconstruction such as the HalfGCD strategy outlined in [103]. For QSLU_rational a speedup could possibly be achieved by delaying canonicalization of the rational numbers, which might reduce the overall time spent on the frequent GCD computations that are made in association with arithmetic operations. Additional speedup in QSLU_rational might be achieved by applying additional effort in the LU factorization to maintain sparsity; more aggressive strategies than those currently used might pay off due to the high cost of rational arithmetic. If all of these suggestions were implemented it would not effect the relative performance of the iterative solvers. We also conjecture that the iterative solvers could be improved more than the direct rational solver, which would not change our conclusions.

Table 12: Details for Specific Instances

Problem Characteristics				Solve Times in Seconds			
Problem Name	Dim	Nz/Row	Sol. Bitsize	Dixon	Iter. Refine	QSLU_rational	Wiedemann
brd14051	16360	11.05	1604	4.34	3.80	29.08	3245.80
cont11_l	58936	3.04	1263	7811.22	[failed]	1.15	1254009.51
fome13	24884	2.84	149	0.40	0.39	1.48	541.25
gen1	329	33.48	439931	1511.03	1395.16	77387.00	118338.65
gen2	328	27.11	20095	6.30	4.39	1706.02	41.39
gen4	375	23.78	22468	10.64	9.18	2673.08	150.09
jendrec1	1779	19.22	23452	8850.06	16908.90	14.27	1134895.15
maros-r7	1350	23.64	46915	42.45	30.17	21.80	1112.93
mod2	4435	2.92	42404	86.93	61.16	1.57	4894.21
momentum3	3254	4.65	159597	652.54	407.21	2978.85	17784.33
nemswrld	2205	6.04	16327	9.73	6.54	16.05	487.04
nug30	14681	3.10	1453	2.66	1.85	191.67	1397.15
pilot	1132	14.68	34247	24.29	[failed]	272.44	459.32
pilot4	289	9.70	79932	25.18	19.52	13.34	582.12
pilot87	1625	19.32	118607	395.61	316.16	13154.93	7235.02
pla33810	18940	6.51	270	0.51	0.48	2.63	822.83
pla85900.nov21	40304	5.72	2182	8.97	7.92	171.19	22234.55
progas	1167	5.56	102225	92.04	63.90	15.37	1798.70
rat5	902	13.33	28969	17.71	12.27	3337.09	211.51
self	924	170.35	46997	266.45	253.80	20970.46	3854.43
slptsk	2315	14.87	77394	213.14	215.33	421.01	7965.67
stat96v3	13485	3.70	3037	4.71	3.77	18.05	2569.22
stat96v4	3139	7.56	150292	618.96	449.77	2763.37	12924.12
stormg2_1000	14075	2.31	34	0.06	0.09	0.18	94.33
watson_1	5729	2.53	744	83.04	74.44	0.09	7165.69

CHAPTER IV

SAFE LP BOUNDS FOR EXACT MIXED INTEGER PROGRAMMING

4.1 Introduction

Chapter 1 discusses many applications where finding exact solutions for MIPs is desirable or necessary. In this Chapter we present a new method for computing valid LP bounds within a MIP framework. We demonstrate its effectiveness using an exact branch-and-bound MIP solver implemented within the SCIP [3, 4] framework.

Hybrid approaches have proven very successful for quickly finding exact rational solutions to Linear Programming (LP) problems. Algorithms using a mix of floating-point and symbolic computation are described in [13, 55, 66, 95, 97] and efficiently implemented and studied by Applegate et al. [13] as QSopt_ex [11]. These hybrid methods exploit the fact that floating-point LP solvers are often able to find an optimal, or near optimal LP basis, even when the solutions have some numerical error. Algorithm 1 in Section 1.2.3 outlines the method used by Applegate et al. [13]. Their strategy can be summarized as follows: the basis returned by a double-precision LP solver is tested for optimality by symbolically computing the basic solution, if it is suboptimal then additional simplex pivots are performed with increased precision, this process is repeated until the optimal basis is identified. This method is considerably faster than an implementation using entirely rational arithmetic.

Following in the success of hybrid symbolic-numeric methods for linear programming, hybrid methods are a clear choice for exact mixed-integer programming. The hybrid approach we have adopted for the branch-and-bound algorithm maintains a floating-point representation of the MIP problem that is either an approximation or relaxation of the original problem. Computations are performed using floating-point arithmetic as much as possible. The decisions that could lead to an incorrect result if done incorrectly, namely computing LP bounds and identifying a new best primal solution, are always performed

in a safe or exact manner. Computing a primal solution exactly will necessarily involve solving an exact LP. Therefore the question of how to quickly compute valid LP bounds is a critical question for building a fast branch-and-bound based exact MIP solver. A more detailed description of the hybrid method for mixed-integer programming can be found in [42].

In Section 4.2 we describe some known methods for generating valid bounds for LP problems. In Section 4.3 we describe the project-and-shift method for generating dual bounds. A full description of our computational experiments presented in Section 4.4.

4.2 Background

The most straightforward way of computing valid LP bounds at nodes of a branch-and-bound tree is to solve the LP relaxation exactly. Within a branch-and-bound framework LP computations can be made faster by warm starting the dual simplex method at each node with the optimal basis from the parent node. Reoptimization can often be accomplished using a small number of pivots. This type of warm start could be used when solving node LPs exactly. However, computing exact LP solutions in this way may still be much slower than the floating-point LP solver. Even in the case when the exact LP solver quickly determines the optimal basis by performing additional pivots in floating-point arithmetic, it would still compute an exact solution and verify its optimality at that node. We have seen that symbolically computing basic LP solutions is an expensive component of solving exact LPs, and therefore performing this computation at every node is undesirable.

Applegate et al. [13] describe an exact rational MIP solver based on their exact LP solver where each LP encountered was solved exactly. While their exact LP solver was only moderately slower than the floating-point LP solvers, the exact MIP code was slower than commercial solvers by two or three orders of magnitude. The cost associated with computing numerous node LP solutions exactly is an explanation for this relative slowdown. Despite the possible disadvantages of solving an exact LP at every node, it is important to recognize that an exact LP solver will be a necessary component of an exact MIP solver and is necessary to compute exact primal solutions. An exact LP solver also has the advantage

that it will provide the tightest valid LP bound at any node of the branch-and-bound tree. It may be necessary to compute this tight bound if all integer variables have been fixed through branching and a bound strong enough to cut off the node is not obtained by other methods.

4.2.1 Exploiting Primal Bound Constraints

When using the hybrid approach, access to approximate LP results will be available at each node, giving either an optimal solution or proof of infeasibility returned by a floating-point LP solver. The LP result may or may not be correct, but it is reasonable to assume that it is often nearly correct and can provide useful information that can be used by alternate dual bounding methods.

A special case occurs when all primal variables have finite upper and lower bounds. This structure allows computation of valid dual bounds by using the dual variables corresponding to the primal variable bounds to correct an approximate dual solution. This technique was employed by Applegate et al. in the Concorde software package [39] to give valid bounds when solving Traveling Salesman Problem (TSP) instances by branch-and-cut. For the TSP all variables are bounded by zero and one and the use of this bounding technique is described in section 5.4 of [12]. Neumaier and Shcherbina [118] described this procedure more generally for MIPs having available upper and lower bounds on all primal variables. Consider the following primal dual pair of LPs:

Primal:	**Dual:**
$\max \quad c^T x$	$\min \quad b^T y - l^T z_l + u^T z_u$
$\text{s.t.} \quad Ax \leq b$	$\text{s.t.} \quad A^T y \; - I z_l \; + I z_u \quad = c$
$l \leq x \leq u$	$y, z_l, z_u \geq 0$

Any approximate dual solution $\tilde{y}, \tilde{z}_l, \tilde{z}_u \geq 0$ can be corrected to be exactly dual feasible by increasing z_l, z_u. If $r = c - A^T\tilde{y} + I\tilde{z}_l - I\tilde{z}_u$ is the error of the approximate solution, then a feasible solution is given by:

$$(\mathbf{y}, \mathbf{z_l}, \mathbf{z_u}) = (\tilde{y}, \tilde{z}_l + r^+, \tilde{z}_u + r^-).$$

Where $r_i^+ = \max(r_i, 0)$ and $r_i^- = \max(-r_i, 0)$. This gives $b^T\mathbf{y} - l^T\mathbf{z_l} + u^T\mathbf{z_u}$ as a valid upper bound on the primal objective. Since this dual bounding method corrects approximate dual solutions using the dual variables coming from primal bound constraints we will call it the *primal-bound-shift method*. The difference between the bound and the objective value of the approximate dual solution will be small if the approximate dual solution does not violate the constraints by a large amount and the bounds on the primal variables l, u are not large.

Proposition 4.2.1. *Let $\tilde{y}, \tilde{z}_l, \tilde{z}_u \geq 0$ be an approximate dual solution, with cost $b^T\tilde{y} - l^T\tilde{z}_l + u^T\tilde{z}_u$, then the bound computed by the primal bound correction method described above will be at most $-l^T r^+ + u^T r^-$ larger than the cost of the approximate dual solution.*

Proof. Subtracting the objective value of the approximate solution from the objective value of the corrected solution gives: $(b^T\mathbf{y} - l^T\mathbf{z_l} + u^T\mathbf{z_u}) - (b^T\tilde{y} - l^T\tilde{z}_l + u^T\tilde{z}_u) \leq -l^T r^+ + u^T r^-$. □

Neumaier and Shcherbina observed that exact precision arithmetic can be entirely avoided when computing r and computing the bound by using floating-point computation and interval arithmetic (or directed rounding if the problem is described in floating-point representable numbers). The strength and simplicity of computing this bound suggests that it will be an excellent choice when tight primal variable bounds are available. The drawback to this method is that if some variable bounds are very large or missing then it could produce weak or infinite bounds. We found that in our test set, 87 out of 155 problems were missing at least some variable bounds, which could lead to failure of this method.

4.2.2 Interval Methods

Some recent studies [6, 87, 93] have looked at solving or detecting feasibility of LPs using interval methods. The methods presented in these articles are more general and sophisticated than the primal-bound-shift method described in the previous section. Althaus and Dumitriu [6] describe an algorithm to certify feasibility and produce bounds for LPs. Their algorithm identifies the implied equalities of the LP and then, using interval arithmetic, corrects the interval solution to satisfy all of the constraints by shifting it toward the relative interior of the polyhedron. They implemented a version of the algorithm to certify

feasibility of problems and had a high rate of success. They also describe how a variant of their method could be used to compute valid LP bounds. Their method almost entirely avoids exact precision arithmetic and does not require any special assumptions on the problem structure. However, it requires the solution of an auxiliary problem each time a bound is computed and can potentially fail due to numerical problems.

4.3 Project and Shift

The methods described in the previous section determine a valid dual solution, or an interval containing a valid dual solution by correcting an approximate dual solution. Similarly, the method presented in this section will generate valid bounds by repairing approximate dual solutions to be exactly feasible. An approximate dual solution is projected to satisfy all of the equality constraints and then shifted toward the feasible region in a way to satisfy all of the remaining inequalities. We will not require upper and lower bounds on primal variables but by imposing some more general conditions on the problem structure we are able to effectively reuse information throughout the branch-and-bound tree.

4.3.1 Basic Idea

The bounding method is defined in terms of the dual problem. Throughout the rest of the Chapter we assume we are working with a MIP with a root and node LP relaxation given below, where we have $A \in \mathbb{Q}^{m \times n}$, $c, x \in \mathbb{Q}^n$, $b, b', y \in \mathbb{Q}^m$, $\bar{A} \in \mathbb{Q}^{\bar{m} \times n}$ and $\bar{b}, z \in \mathbb{Q}^{\bar{m}}$.

Root Primal:	**Root Dual:**	**Node Primal:**	**Node Dual:**
$\max \quad c^T x$	$\min \quad b^T y$	$\max \quad c^T x$	$\min \quad b'^T y + \bar{b}^T z$
$\text{s.t.} \quad Ax \leq b$	$\text{s.t.} \quad A^T y = c$	$\text{s.t.} \quad Ax \leq b'$	$\text{s.t.} \quad A^T y + \bar{A}^T z = c$
	$y \geq 0$	$\bar{A}x \leq \bar{b}$	$y, z \geq 0$

The most frequent operation performed in a branch-and-bound tree is the modification of primal variable bounds. In our notation this would correspond to lowering components of b, or to introducing new inequalities if the bound was previously infinite. We can assume that the LP relaxation at any node has $b' \leq b$. Adding a cutting plane corresponds to adding

a new inequality to the primal problem and thus a new column to the dual problem. A solution feasible for the root node dual LP will be feasible for all of the node dual problems in the branch-and-bound tree by setting the additional components z to zero.

Algorithm 9 LP Bound by Dual Correction (Single LP Version)

Input: Dual constraints $A^T y = c, y \geq 0$, approximate solution $\tilde{y}$
Determine implied equalities of dual polyhedron
Compute relative interior point y^* of dual polyhedron
Fix implied zero components of $\tilde{y}$ to zero
Project $\tilde{y}$ to satisfy $A^T \tilde{y} = c$
Take convex combination y of $\tilde{y}$ and interior point y^* to ensure $y \geq 0$
Return: Dual bound $b^T y$

Algorithm 9 is a simplified description of the project-and-shift algorithm as it would be applied to a single LP. It is described using the notation of the root node as given above. As long as the dual LP is feasible this algorithm will always produce a valid bound. It makes use of an approximate dual solution and corrects it to be exactly feasible. Here we describe bounds in the case where the dual objective is bounded; we discuss the case of primal infeasibility/dual unboundedness in Section 4.3.5. The main idea of the algorithm given by Althaus and Dumitriu [6] is similar to this idea and uses interval arithmetic to avoid the exact computation.

In principle, Algorithm 9 could be applied to generate a valid LP bound at each node of the branch-and-bound tree, but the operations required could be quite expensive. The operations of determining the implied equality constraints of the polyhedron and finding a relative interior solution could be more difficult than solving the exact LP in the first place. We now adopt this procedure to work efficiently in a MIP setting by performing the most expensive computations only once at the root node of the branch-and-bound tree and reusing the structural information in order to decrease the computation performed at each bound computation.

The algorithm we describe will have two components. A setup phase that must be solved once at the root node, and a bound computation operation that can be called at nodes of the branch-and-bound tree. We make the assumption that the matrix A^T has full row rank and that there are no implied equalities. We will later demonstrate that this assumption

about the problem structure is more general than the assumption of primal bounds and is often satisfied on a large test set of real-world problems.

In the setup phase, given as Algorithm 10, we choose a submatrix A_S^T of A^T by selecting a subset S of the columns that span $\mathbb{R}^n$. This subset S could be chosen to be all of the columns. Then an LU factorization of A_S^T is computed. Finally a corrector point y^* is computed such that it is dual feasible and that it has strictly positive values in all components corresponding to columns in S. Throughout this Chapter we refer to such a point as an *S-interior point*.

Algorithm 10 Project and Shift: Setup Phase

Input: Root dual constraints $A^T y = c, y \geq 0$
Choose subset S of columns of A^T spanning $\mathbb{R}^n$
Compute exact LU factorization of submatrix A_S^T induced by S
Compute exact S-interior dual solution y^*
Return: S, LU, y^*

The node bound computation is given as Algorithm 11. An approximate dual solution, $\tilde{y}, \tilde{z} \geq 0$, is corrected to be exactly feasible. First, the violation of the equality constraints r is computed and an adjustment correcting this violation $w \in \mathbb{R}^m$ is computed using the LU factorization found in the setup phase. After adding this correction to $(\tilde{y}, \tilde{z})$ it is possible that some components of $(\tilde{y} + w, \tilde{z})$ are negative, this possible negativity is corrected by taking a convex combination with y^* to ensure that all components satisfy non-negativity. Note that the approximate dual solution is preconditioned to be non-negative. In the case that some components are slightly negative due to numerical errors, those components can be set to zero.

Algorithm 11 Project and Shift: Node Bound Computation

Input: Node dual constraints $A^T y + \bar{A}^T z = c, \quad y, z \geq 0$, approximate dual sol. $\tilde{y}, \tilde{z} \geq 0$
Compute error in equality constraints $r = c - A^T \tilde{y} - \bar{A}^T \tilde{z}$
Solve $A^T w = r$ using precomputed LU factorization of A_S^T
Choose smallest $\lambda \in [0, 1]$ such that $(y, z) = (1 - \lambda)(\tilde{y} + w, \tilde{z}) + \lambda(y^*, 0)$ is non-negative
Return: Dual bound $b'^T y + \bar{b}^T z$

Choosing a value of λ such that $(y, z) = (1 - \lambda)(\tilde{y} + w, \tilde{z}) + \lambda(y^*, 0) \geq 0$ is always possible. This is because all components of $\tilde{y}, \tilde{z} \geq 0$, with the only negativity in $(\tilde{y} + w, \tilde{z})$

coming from w whose support is a subset of the columns in S. By definition y^* is strictly positive in all components of S. Feasibility of (y, z) is guaranteed because it is a convex combination of two solutions to the equality constraints of the system. This explains the use of our assumptions: the assumption that the columns of S span $\mathbb{R}^n$ implies that the LU factorization of A_S^T can be used to correct any violation r; the assumption that there are no implied equalities ensures the existence of an S-interior point y^* that can be used to correct any negativity appearing in w.

Throughout the remainder of the Chapter we will refer to the use of Algorithms 10 and 11 together within a MIP branch-and-bound tree as the *project-and-shift method*. Algorithm 9 gave a simplified description of this method as it would be applied to a single LP.

The setup phase will require solving one or two exact LPs that will be described in the later sections and computing an LU factorization. The node bound computations will require considerably less computation, the most expensive part being the back-solve of a system of equations that is done with a precomputed LU factorization. This method may be relatively slow if used to compute a single LP bound but in a branch-and-bound tree it could prove reasonable, especially when many nodes are processed. We would expect the node bound computation step of this algorithm to be considerably faster than solving an exact LP at a given node. Even if the optimal basis is passed to the exact LP solver as a warm start it would compute the final basic solution exactly, which may require solving a system of equations exactly. Computing a basic solution exactly is likely to be slower than using a precomputed LU factorization to make the correction in Algorithm 11.

We now show that the assumption that the dual problem had no implied equalities and a full row rank constraint matrix is more general than the assumption that all primal variables have finite upper and lower bounds. For a problem described as equality constraints and non-negativity constraints on the variables the term *implied equality* refers to any variable bounds that are implied to be tight.

Proposition 4.3.1. *Suppose that an LP of the form:*

$$\begin{aligned} \min \quad & b^T y - l^T z_l + u^T z_u \\ s.t. \quad & A^T y \ - I z_l \ + I z_u = c \\ & y, z_l, z_u \geq 0 \end{aligned}$$

is feasible, then it has no implied equalities and the constraint matrix has full row rank.

Proof. The constraint matrix has full row rank because it has an identity submatrix. We now show that there are no implied equalities. Let (y, z_l, z_u) be a feasible solution and let $\alpha = \sum_{i=1}^{m} (A^T)_i$ (the sum of all the columns of A^T). Consider the solution $(y + \mathbb{1}, z_l + \mathbb{1} + \alpha^+, z_u + \mathbb{1} + \alpha^-)$. We can see that each component is at least one and $A^T(y + \mathbb{1}) - I(z_l + \mathbb{1} + \alpha^+) + I(z_u + \mathbb{1} + \alpha^-) = A^T y + \alpha - I z_l - \alpha^+ + I z_u + \alpha^- = A^T y - I z_l \ + I z_u = c$. Therefore this gives a feasible solution that is strictly positive in each component verifying that no variables are implied to be zero. □

Moreover, as we will see in Section 4.4 when considering our test set of 155 problems, the conditions that the dual problem has no implied equalities and that the dual constraint matrix is full rank holds on 150 of them, where only 68 of the problems had bounds on all primal variables.

The project-and-shift method relies on the existence of a full row rank submatrix A_S^T and also on the existence of a corrector point y^* that is S-interior. We now show that the existence of the S-interior point is equivalent to the condition that there are no implied equalities.

Proposition 4.3.2. *Suppose the LP* $\min\{b^T y | A^T y = c, \ y \geq 0\}$ *is feasible and* A^T *has full row rank. Then there exists an* S*-interior point for a full row rank subset of columns* S *of* A^T *if and only if there are no implied equalities.*

Proof. First, suppose there is an S-interior point of a full rank subset of the columns S. We may assume that $A^T = [A_S^T | A_N^T]$ where N is the set of columns not in S and there is a solution (y_S, y_N) with $y_S > 0$. Let $i \in N$ and suppose $y_i = 0$, then we can construct a solution in the following way. Since A_S^T has full row rank there exists a solution w

to the equations $A_S^T w = A_i^T$. We can choose $\epsilon > 0$ such that $(y_S - \epsilon w) > 0$ and then $(y_S - \epsilon w,\ y_N + \epsilon e_i)$, where e_i is the ith unit vector, is a feasible S-interior point that is strictly positive in component i. This can be repeated for any column in N and therefore there are no implied equalities. The converse of the statement holds trivially by taking S equal to all the columns. □

Now we consider the quality of the bounds that are produced by the project-and-shift algorithm. Proposition 4.3.3 gives a bound on the strength of the LP bound in terms of approximate dual solution and the information computed in Algorithm 10.

Proposition 4.3.3. *Assume that when Algorithm 10 is applied to the root node problem it identifies an S-interior point y^* with objective value z_R, and that $\forall i \in S$, $y_i^* \geq d > 0$. Next, suppose Algorithm 11 is applied at the node to compute a bound, given an approximate solution $\tilde{y}, \tilde{z} \geq 0$ with objective value $\tilde{z}_N = b'^T\tilde{y} + \bar{b}^T\tilde{z}$. Also assume that $(z_R - \tilde{z}_N) \geq 0$ (otherwise z_R can be taken as a safe dual bound). Let $r = c - A^T\tilde{y} - \bar{A}^T\tilde{z}$ be the error of the approximate solution and let w be the correction used, $A^T w = r$. Then the bound returned will be at most*

$$\tilde{z}_N + (1/d)(\max_{i \in S} w_i^-)(z_R - \tilde{z}_N) + (b'^T w)^+.$$

Proof. First note that $(1/d)(\max_{i \in S} w_i^-)$ gives an upper bound on the value of λ computed in Algorithm 11. Now we overestimate the dual bound produced by Algorithm 11.

$$\begin{aligned} b'^T y + \bar{b}^T z &= (1-\lambda)(b'^T(\tilde{y} + w) + \bar{b}^T\tilde{z}) + \lambda(b'^T y^*) \\ &\leq (1-\lambda)\tilde{z}_N + (1-\lambda)b'^T w + \lambda z_R \\ &\leq \tilde{z}_N + \lambda(z_R - \tilde{z}_N) + (b'^T w)^+ \\ &\leq \tilde{z}_N + (1/d)(\max_{i \in S} w_i^-)(z_R - \tilde{z}_N) + (b'^T w)^+ \end{aligned}$$ □

Unlike Proposition 4.2.1 and the primal-bound-shift dual bound method, the project-and-shift method does not necessarily depend on the values or existence of primal variable bounds. We can also identify characteristics of the problem, and of the information computed in Algorithm 10 that can lead to stronger bound computations. If the S-interior point y^* has a good objective value, and if its components in S have large values, this

can improve the bound quality. Another desirable feature is that the projection vector w should hopefully be of small magnitude; this may be harder to control as w will depend on the solution to the system of equations $A^T w = r$. We also note that if the approximate dual solution only violates the constraints by a small amount, this will generally lead to a smaller difference between the objective value of the approximate dual solution and the bound value.

4.3.2 Generating Projections

A key component of the project-and-shift method is the projection step. A projection of the approximate dual solution into the affine hull of the dual polyhedron ensures that all equality constraints are satisfied. Projection of a vector into an affine space is a basic operation of linear algebra, in this section we explain our strategy for computing projections.

As described in Algorithms 10 and 11, the projection is done by computing an LU factorization of a rectangular matrix A_S^T, which is used to compute a corrector w by solving $A^T w = r$. Using an LU factorization can take advantage of sparsity of the matrix, which is often very sparse in real world MIP problems. Computing LU factorizations on problems arising from LP applications is a well studied area so we can take advantage of these highly developed techniques. Also, the matrix factorization, which is the most difficult operation is only performed once during Algorithm 10 in the setup stage. When Algorithm 11 is called to compute node bounds, the projection is accomplished by performing a back-solve using the already computed factorization.

Alternative strategies for computing projections could use other methods, such as orthogonal projections. An advantage of computing orthogonal projections is that the approximate solution would be mapped to the closest point in the affine hull. The drawback of using orthogonal projections is that they may be significantly more computationally expensive. To symbolically compute an orthogonal projection at each node may be even more difficult than calling the exact LP solver with warm starts.

In the project-and-shift method we choose a subset of the columns S to define a submatrix A_S^T to control which submatrix has its LU factorization computed, and which components can be adjusted during the projection. Choosing S as all the columns is a valid choice, but there are reasons to choose a smaller subset. Choosing a smaller subset S will reduce the dimension of A_S^T making the computation of the rectangular LU factorization faster. Ideally we would also choose the columns of S as dual variables which can be adjusted without having a large effect on the objective value. One alternative choice for S is to consider the optimal primal solution at the root node and then choose S to be the set of all dual columns corresponding to active primal constraints. If we correctly compute the optimal primal solution at the root node this choice of S would give a full row rank submatrix A_S^T. Relative to other dual columns, those corresponding to active constraints at the optimal root node solution may have better objective cost, leading to an improvement in the bound quality as it is shown in Proposition 4.3.3

4.3.3 Identifying an S-interior Point

Freund, Roundy and Todd [69] described a method to simultaneously compute an interior point and identify the affine hull of a polyhedron by solving a single LP. Using a similar idea we write an Auxiliary problem that given a set S will identify an S-interior point if one exists. Expressing the problem in the dual form, implied equality constraints correspond to components of the problem that must be zero in any feasible solution. Here the added variables are $\delta \in \mathbb{R}^{|S|}$ and λ is a single variable.

Dual LP Problem:

$$\begin{aligned} \min \ & b^T y \\ \text{s.t.} \ & A^T y = c \\ & y \geq 0 \end{aligned}$$

Aux. LP Problem:

$$\begin{aligned} \max \ & \sum_{i \in S} \delta_i \\ \text{s.t.} \ & A^T y - \lambda c = 0 \\ & y_i \geq \delta_i \ \ \forall i \in S \\ & y \geq 0, \lambda \geq 1, \ 0 \leq \delta_i \leq 1 \end{aligned}$$

If we set S equal to all the columns of A^T and suppose $P = \{y | A^T y = c, y \geq 0\}$ and

y, δ, λ is an optimal solution to the Aux. problem then $\frac{1}{\lambda}y$ is contained in the relative interior of P, and $\delta_i = 0$ implies $y_i = 0$ for every $y \in P$. Geometrically this problem adds an extra dimension λ which scales the right hand side of the constraints converting the polyhedron to a conic form. The variables δ are indicators of each inequality being satisfied strictly, and if any δ_i can take a nonzero value, it can attain its maximum value of 1 by increasing y, λ and moving further into the cone. Choosing S to be any strict subset of the dual columns, an optimal solution would produce an S-interior point $\frac{1}{\lambda}y$ if one exists; otherwise some variables δ_i for $i \in S$ would be zero.

In Algorithm 9 it was necessary to identify the affine hull and an interior point of the dual polyhedron. Solving this auxiliary problem exactly would accomplish these goals. However, it would not be practical to solve for each bound computation because that would require solution of an exact LP at each node.

We can use this Aux. LP Problem within Algorithm 10 to correctly identify y^* as needed. It would also recognize if no S-interior point exists if δ_i was equal to zero for any $i \in S$ in the optimal solution. By Proposition 4.3.2 if this Aux. Problem fails to find an S-interior point then the problem has implied equalities.

The clear disadvantage of using this method to identify the S-interior point is that the point chosen in an arbitrary way. It could have a very bad objective value and could also have some components with strictly positive but have very tiny values that could result in poor bound values after application of Algorithm 11; both of these situations were observed on some problems in our test set. Next we consider ways of choosing an S-interior point while considering both its objective value and the value of its positive components.

Computing a bound using the project-and-shift method requires identification of an S-interior point. In Algorithm 11 we see that the value of the bound computed will depend on the objective value of this point, and the magnitude of the shift will depend on how large the values of the point are. Therefore the ideal point y^* would have a good objective value and also have large values in the components in S. Since we have assumed that S induces a full row rank submatrix of A^T and there exists an S-interior point we can use the following auxiliary problem to identify one, where α is weights given to balance the two components

of the objective function. We just introduce one additional variable δ that is a lower bound entries of S.

Optimized Aux. LP Problem:

$$\begin{aligned}
\max \quad & (1-\alpha)(\max\{1,|z_{LP}|\})\delta + \alpha(b^T y) \\
\text{s.t.} \quad & A^T y = c \\
& y_i \geq \delta \quad \forall i \in S \\
& y \geq 0,\ 0 \leq \delta \leq M
\end{aligned}$$

The first part of the objective function, $(1-\alpha)(\max\{1,|z_{LP}|\})\delta$, corresponds to a lower bound on the minimum value over components in S. Maximizing δ maximizes the minimum over all components of y in S. The second part $\alpha(b^T y)$ corresponds to the objective value of the original dual problem. One option for solving this problem is to choose the weight α and solve the problem using an exact LP solver. We would typically have access to the optimal solution for the root node LP so we normalize the first term by including a factor of $\max\{1,|z_{LP}|\}$ where z_{LP} is the optimal objective value at the root node. For practical purposes, instead of computing z_{LP} exactly at the root node, we could also use an approximation of this value provided by the floating-point LP solver.

In our experiments we found this strategy to be successful for some problems, but in other cases the optimal solution had a value of $\delta = 0$, even when setting the value of α to be very small. When $\delta = 0$ this indicates that that the solution does not give an S-interior point, leading to failure of the project-and-shift method. In Table 13 we list some of the failure rates for different values of α. (We make a note that this table actually gives an underestimate of the failure rate, for each method a small number of instances timed out or halted for system reasons, and those instances are not included as failures here.)

After observing this behavior we employed a second strategy where the problem is solved in two stages. First, the problem was solved by setting $\alpha = 0$. Then knowing a feasible value of δ, the lower bound on δ is adjusted to reflect this known feasible value. Second, with this lower bound on δ, α is set equal to 1 and the problem is re-solved. Solving the second

Table 13: Success Rate of Single Stage Optimized Aux. LP Problem

Value of α	Failure Rate ($\delta = 0$)
$\alpha = 0.01$	26/155
$\alpha = 0.0001$	10/155
$\alpha = 0.000001$	5/155

problem can also be done as a reoptimization since the only modification to the problem changed the variable bounds and objective. After solving the first step of this problem and adjusting the lower bound on δ, the optimal basis is still primal feasible (but not dual feasible) and therefore re-optimization can be done using the primal simplex algorithm.

4.3.4 Shifting by Interior Ray

Thus far we have described the shift step of the project-and-shift method as using a projected approximate solution and then shifting it using a convex combination of it along with a corrector point y^*. An alternative is to correct the projected solution by adding some multiple of a ray from the dual recession cone to correct it. We could adjust Algorithm 10 so that instead of finding an S-interior point y^* it would compute a ray r^* in the dual recession cone, $R = \{r : A^T r = 0, r \geq 0\}$, satisfying $r_i^* > 0 \ \forall i \in S$. Following our previous notation, we would call this an *S-interior ray*. Then, in Algorithm 11 instead of computing a bound from the corrected solution as the convex combination $(y, z) = (1 - \delta)(\tilde{y} + w, \tilde{z}) + \delta(y^*, 0)$ we would use $(y, z) = (\tilde{y} + w, \tilde{z}) + \gamma(r^*, 0)$, where γ is chosen large enough that $(y, z) \geq 0$.

The drawback of this strategy is that it is less general than the previous assumptions. For example, if the root node dual LP has a bounded feasible region, a ray r^* satisfying these conditions does not exist.

Proposition 4.3.4. *Existence of an S-interior ray in the dual is implied by presence of all primal bounds. Existence of S-interior ray implies existence of S-interior point.*

Proof. If the primal problem has upper and lower bounds on all variables the dual will be in the following form, the same as in Proposition 4.3.1, given by $\min\{b^T y - l^T z_l + u^T z_u | A^T y - I z_l + I z_u = c, \ \ y, z_l, z_u \geq 0\}$. Then the ray given by $(\mathbb{1}, \mathbb{1} + \alpha^+, \mathbb{1} + \alpha^-)$ where

$\alpha = \sum_{i=1}^{m} (A^T)_i$ is in the recession cone and is strictly positive in each component.

Now to see that existence of an S-interior ray implies existence of S-interior point we observe that taking any feasible solution and adding some multiple of an S-interior ray would give an S-interior point. □

Examples can easily be constructed to demonstrate that none of the reverse implications hold. Table 14 lists how often each of these conditions held at the root node of the problem on our test set.

Table 14: Occurrence of Conditions

Condition	Occurrence
All primal bounds present	68/155
Existence of S-interior ray	128/155
Existence of S-interior solution	150/155

4.3.5 Proving LP Infeasibility

In addition to bounding LP objective values at nodes of the branch-and-bound tree another frequent operation is to certify primal infeasibility of nodes. Primal infeasibility can be certified by proving that the dual problem is unbounded. Proving dual infeasibility / primal unboundedness is not as relevant a problem because if the root node primal LP has a bounded objective value, then it will remain bounded through the branch-and-bound tree. In this section we describe two different approaches of how the project-and-shift algorithm can be adapted to certify primal infeasibility.

One possible method for proving the dual objective value is unbounded is to is to have the floating-point LP solver return a cost-improving dual ray and then correct this approximate ray to be exactly feasible. We could use the idea of projecting and shifting the dual ray to be exactly feasible and if this operation produced an exactly feasible dual ray that was cost improving then the primal infeasibility would be certified. The projection could be done the same way as we have done in Algorithms 10 and 11, except that we would correct the ray to satisfy $A^T y = 0$, and the shift could be accomplished by using an S-interior ray of the

dual. This approach has two drawbacks, first it would require an alternate Aux. problem to be solved at the root node to identify an S-interior ray. Secondly, it would require the less general condition that there is an S-interior ray which was discussed in the previous section.

An alternate approach is to compute a valid dual bound strong enough to prune the node without explicitly finding a cost improving dual ray. Whenever a valid dual bound for a node is identified that surpasses the primal bound (the best known primal objective value) that node can be pruned. In fact, many software packages for branch-and-bound will terminate the simplex algorithm early at a node after a dual solution good enough to prune it is identified.

Therefore at a node which is thought to be dual unbounded we can simply attain a dual solution that surpasses the best primal bound by some amount and apply the project-and-shift algorithm to that approximate solution with the goal of pruning the node. An approximate dual solution surpassing the primal bound value should be readily available at a primal infeasible node. Such a dual solution could be returned directly by the floating-point LP solver, or it could be constructed by adding a multiple of an approximate cost improving dual ray to a dual feasible solution.

4.4 Computational Results

In this section we describe an implementation of the project-and-shift algorithm, our test sets and the computational results. The first goal of our computational tests is to determine which of the several variations of the algorithm is the most effective in practice. The second goal is to demonstrate that this method provides an advantage over other dual bounding methods on some classes of problems.

4.4.1 Implementation and Test Set

The project-and-shift method for generating valid LP bounds is implemented within a hybrid exact branch-and-bound version of the MIP software package SCIP [3, 4]. The hybrid branch-and-bound code stores an exact representation of the problem, but applies the branch-and-bound algorithm on a floating-point approximation or relaxation of the

problem. In order to find exact precision solutions and to avoid any numerical errors, symbolic computation is used to verify or recompute any computations that would be susceptible to numerical error. Whenever an incumbent primal solution is identified using the floating-point code, it is recomputed exactly by solving an exact LP. Finally, a valid dual bounding method is used to compute the LP bound at each node. The implementation allows the choice of several dual bounding methods, including the variants of the project-and-shift algorithm described in this Chapter. Much of this exact framework was developed by Kati Wolter and an in depth coverage of the hybrid branch-and-bound implementation within SCIP will be presented in her forthcoming PhD thesis. An extended description also appears in [42]. We also remark that the project-and-shift method is capable of computing bounds in a branch-and-cut framework, but as the current exact version of SCIP does not include cuts this functionality is not included.

The exact code is based on SCIP version 1.2.0.8. QSopt_ex 2.5.5 [11] is used as the exact LP solver and CPLEX 12.10 [85] is used as the floating-point LP solver. Daniel Espinoza implemented a SCIP interface and additional features for QSopt_ex to allow for this integration. The auxiliary LPs in the setup phase of the project-and-shift algorithm are solved using the QSopt_ex interface. The rectangular LU factorizations are computed using a code developed by combining the exact LU factorization code from QSopt_ex [13] and a sparse numerical rectangular LU factorization code from [45] which was provided by Sanjeeb Dash, both of these were based directly on the methods of Suhl and Suhl [133].

Computations are performed on a test set of 155 MIP problems selected from MIPLIB 3.0 [24], MIPLIB 2003 [5], and the collection of Mittelmann [110]. Following the convention used throughout this dissertation we solve the problems as they are given, we do not round the input data to nearby rational numbers with small denominators or modify the input data in any other way. The problems are all converted to ZIMPL format and read in as rational numbers. Computational tests were run on the Zuse Institute Berlin (ZIB) computing cluster using Intel E5420 processors with a one hour time limit.

Before presenting results comparing the different variants of the project-and-shift algorithm we briefly describe some other aspects of the implementation which increased the

overall speed of the methods. The version of QSopt_ex used in our implementation uses QSopt as its floating-point LP solver. We are therefore able to increase the speed of the code by solving the floating-point approximation of the exact LPs with CPLEX and then using the final basis from CPLEX to warm start QSopt_ex. Another practical step that is taken is a simple postprocessing of the exactly feasible dual solution obtained by the project-and-shift method when some constraints of the problem have right and left hand sides. If the dual multipliers for both sides of a specific constraint are nonzero then they can be lowered by equal amounts so that one becomes zero, this will reduce the cost of the dual solution and improve the bound quality.

4.4.2 Computations

We have described several variants of the project-and-shift algorithm, the significant decisions to be made are how to choose the set S and how to choose the S-interior point. We want a method that is as general as possible and is fast for the MIP application. In general the speed of the dual bound method for MIP will be influenced by two things; how fast the bounds can be computed and how strong the bounds are, because weak bounds may lead to an increased node count.

As a first consideration we eliminate some of the variants due to their lack of generality. The use of an S-interior ray instead of an S-interior point described in Section 4.3.4 has conditions that satisfied less often than the other versions. Also, from Proposition 4.3.4 we know that any of the times it fails are when the conditions of the primal-bound-shift method do not hold.

Next we consider the optimized interior point described in Section 4.3.3. We described an auxiliary problem with a two part objective function. When using nonzero values of α we often experienced problems where the solution to the optimization problem was not S-interior, the failure rates are listed in Table 13. Based on these failure rates we will not consider these variants as viable alternatives, however we do consider the setting of $\alpha = 0$ and the two stage problem. We also remark that when using different nonzero values of alpha did work, their performance on the overall branch-and-bound tree was similar to the

performance of $\alpha = 0$.

After eliminating these possibilities we are left with three choices of how to compute the interior point. First, we could choose an arbitrary interior point using the Aux. problem listed in the beginning of Section 4.3.3. Second, we could solve the Optimized interior point problem given in Section 4.3.3 with $\alpha = 0$; maximizing the minimum size of components in S. Third, we could solve the two stage problem, where we maximize the minimum over components in S, and then do further optimization with a modified objective function to improve the point's objective value. In the tables, we will denote these three settings as P:Arbitrary, P:Opt and P:TwoStage, respectively. The second parameter we have to choose is how to select the set S. We consider two possibilities, first we can let S be equal to all the dual columns. The second possibility is to set S equal to all dual columns corresponding to primal constraints that are active at the optimal root node solution, motivation for such a choice is discussed in Section 4.3.3. We denote the parameter choices for the set S by S:All and S:Active. These settings give us six combinations to compare.

First we compare the behavior at the root node, evaluating the quality of the bound produced and the time necessary to compute it. The dual bound time required at the root node is dominated by the solution of the auxiliary problem, which in each case involves solving an exact LP. Table 15 compares the relative quality of the dual bounds at the root node by comparing the bound value with the exact LP solution value. Namely, we define the bound quality to be $q = (z_{LP} - DB)/max(1, z_{LP})$, where z_{LP} is the optimal LP value, and DB is the value of the dual bound obtained. For each setting we list how many problems had bounds with relative quality in five different ranges, the infinite bound arising on problems where the conditions required by the algorithm were not satisfied. The bounds are said to have value less than ϵ_m when q is below machine epsilon; it is mapped to zero when converted to double precision. Any problems which fail due to memory overflow or other system errors for any of the compared parameter settings are excluded from the comparison in the tables.

The relative quality of the bounds produced by these settings is not surprising. Choosing S equal to the active columns leads to increased bound quality. The selection of the interior

point also impacts the quality, with the arbitrary interior point giving the worst bound, and the two stage problem producing better bounds on average.

Table 15: Relative Bound Quality at Root Node

Setting	$[0, \epsilon_m)$	$[\epsilon_m, 10^{-9})$	$[10^{-9}, 10^{-3})$	$[10^{-3}, \infty)$	∞
S:Active;P:Opt	50	83	9	2	5
S:Active;P:Arbitrary	50	77	15	2	5
S:Active;P:TwoStage	49	88	5	2	5
S:All;P:Opt	32	89	17	6	5
S:All;P:Arbitrary	32	80	26	6	5
S:All;P:TwoStage	34	91	13	6	5

Table 16 compares the computation time required at the root node by each method. For each method three measures of time are considered; the geometric mean of the solve times over the problem set, the total solution time over the problem set and the shifted geometric mean over the problem set. The shifted geometric mean of numbers $t_1, \ldots, t_n$ is defined to be $(\Pi_{i=1}^{n}(t_i + s))^{1/n} - s$ and is an intermediate measure between arithmetic and geometric mean, we use a shift factor of $s = 10$. Each of these three measures are computed and the table shows the relative percent increase compared with the reference setting S:Active;P:Opt.

Table 16: Relative Bound Computation Time at Root Node

	Instances Solved by All at Root (149)		
Setting	Geometric Mean	Shifted Geometric Mean	Total Time
S:Active;P:Opt	0	0	0
S:Active;P:Arbitrary	+2	+14	+14
S:Active;P:TwoStage	+68	+96	+444
S:All;P:Opt	+56	+69	+214
S:All;P:Arbitrary	+11	+21	+37
S:All;P:TwoStage	+143	+187	+805

Figure 5 gives a performance profile comparing the solution times at the root node. From this table and performance profile we also observe a predictable outcome. Selecting the interior point using the two stage problem leads to a significant increase in the solution times. Selecting the interior point in the arbitrary way is sometimes slower than the setting P:Opt,

we would attribute this to the arbitrary point selection problem having more variables than the optimized problem. Also, choosing S equal to the active columns instead of all columns reduces the solution time.

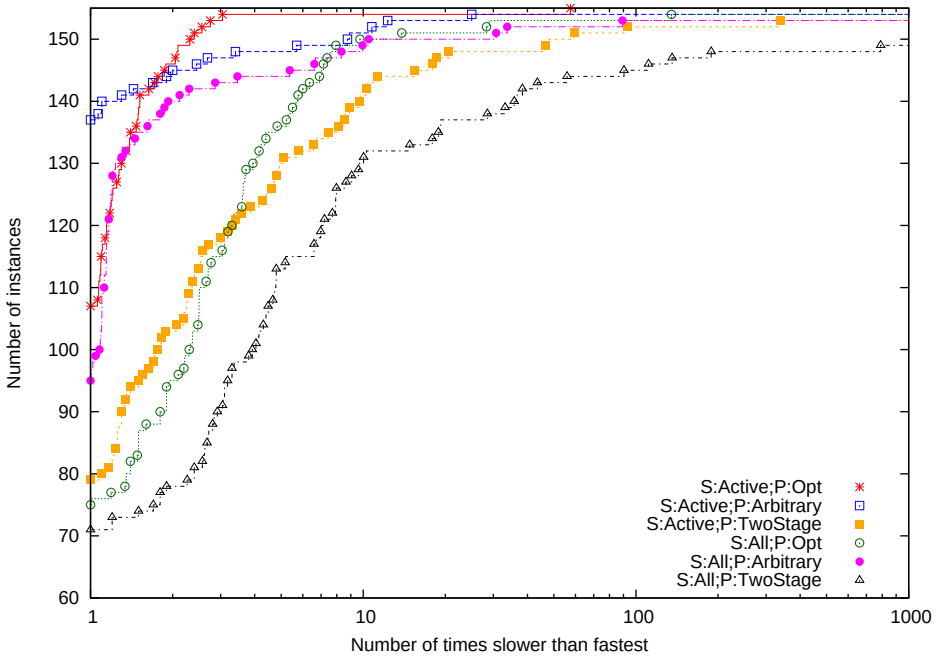


Figure 5: Bound Computation Time at Root Node

Next, Table 17 compares the total time solution times of these methods when used to solve MIP problems using branch-and-bound. A considerable number of the problems remain unsolved after the one hour time limit, so the table compares the solution time on the problems which all methods finished.

From Table 17 we identify the settings S:Active;P:Opt, S:Active;P:Arbitrary and S:All;P:Opt as the three most promising methods to compare in more detail. We will compare these three methods more closely in Table 18 where we include details for specific problems. For each algorithm and problem instance we list the number of branch-and-bound nodes processed, the optimality gap remaining after one hour, or zero if the problem was solved within one hour. We also include the total solution time, and the total amount of time

Table 17: Relative Overall Computation Time

	Instances Solved by All (34)		
Setting	Geometric Mean	Shifted Geometric Mean	Total Time
S:Active;P:Opt	0	0	0
S:Active;P:Arbitrary	+6	+6	+3
S:Active;P:TwoStage	+14	+12	+5
S:All;P:Opt	+4	+2	+2
S:All;P:Arbitrary	+11	+10	+5
S:All;P:TwoStage	+23	+20	+10

spent on computing dual bounds. This table also includes details showing the performance of using the Exact LP solver to compute the dual bounds. Additional statistics are compiled in the final rows of the table, comparing the overall solution times on all problems and on the subset of problems that were solved by all methods. The average optimality gap is also included for problems on which all methods timed out. Figure 6 gives a performance profile comparing the solution times of these methods for solving these MIPs.

Considering the results presented in Table 18 and Figure 6 it is not entirely clear which of these three variants of the project-and-shift algorithm is best. None of the methods dominates the others by a large margin and there is some crossover in the performance profile. The settings S:Active;P:Opt does have the fastest average solution times on the problems solved by all methods, so we will consider it as the best choice. One additional observation that can be made is that on problems solved to optimality, the node counts were often similar or the same between the different methods. This indicates that although the project-and-shift method is producing LP bounds that are not as tight as the exact LP solutions, they are often tight enough that they do not lead to a significant increase in the number of nodes required to solve the problem.

Table 18: Detailed Results on Entire Problem Set

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
10teams	>67 713	>0.8	>3600.0	>2386.2	>68 586	>0.8	>3600.0	>2373.5	>71 500	>0.8	>3600.0	>2329.4	>68 708	>0.8	>3600.0	>2217.8
30:70:4_5:0_5:100	>7 942	—	>3600.0	>3028.5	>10 209	—	>3600.0	>2955.6	>9 981	—	>3600.0	>2944.0	>2 048	—	>3600.0	>3487.8
30:70:4_5:0_95:98	>7 997	—	>3600.0	>3253.5	>10 220	—	>3600.0	>3146.3	>10 008	—	>3600.0	>3247.1	>11 702	>4.3	>3600.0	>3423.5
30:70:4_5:0_95:100	>10 585	>0.0	>3600.0	>2635.9	>11 537	>0.0	>3600.0	>3222.4	>10 744	—	>3600.0	>3371.5	170	0.0	80.3	62.3
a1c1s1	>99 604	—	>3600.0	>3371.4	>98 320	—	>3600.0	>3375.1	>81 199	—	>3600.0	>3413.5	>75 948	—	>3600.0	>3398.8
acc-0	52	0.0	3.2	1.0	52	0.0	3.1	1.0	52	0.0	4.5	2.0	47	0.0	4.2	1.8
acc-1	3 224	0.0	338.4	67.8	3 224	0.0	340.1	69.2	3 224	0.0	338.3	68.9	58	0.0	21.2	4.6
acc-2	241	0.0	45.7	6.0	241	0.0	45.8	6.0	241	0.0	47.6	7.9	49	0.0	20.5	4.1
acc-3	>6 780	—	>3600.0	>192.7	>6 772	—	>3600.0	>201.5	>4 849	—	>3600.0	>1171.1	1 117	0.0	861.7	409.3
acc-4	>6 478	—	>3600.0	>182.5	>6 438	—	>3600.0	>191.4	>5 478	—	>3600.0	>686.5	927	0.0	3001.2	2547.3
acc-5	>9 375	—	>3600.0	>250.8	>9 301	—	>3600.0	>274.8	>8 704	—	>3600.0	>506.4	>2 249	—	>3600.0	>2541.3
acc-6	>9 029	—	>3600.0	>222.3	>9 034	—	>3600.0	>223.5	>7 476	—	>3600.0	>824.1	>1 865	—	>3600.0	>2897.6
aflow30a	>357 811	>17.9	>3600.0	>3318.7	>401 982	>17.7	>3600.0	>3288.2	>310 351	>18.1	>3600.0	>3357.8	>187 142	>33.2	>3600.0	>3383.2
aflow40b	>101 541	>85.2	>3600.0	>3194.7	>113 452	>85.2	>3600.0	>3144.1	>89 634	>85.3	>3600.0	>3242.2	>72 132	>85.5	>3600.0	>3307.3
air03	21	0.0	27.8	26.8	21	0.0	24.6	23.5	21	0.0	55.5	54.5	21	0.0	3.5	3.0
air04	>13 670	>0.4	>3600.0	>2245.5	>13 846	>0.4	>3600.0	>2229.2	>14 329	>0.4	>3600.0	>2178.4	>9 836	>0.5	>3600.0	>2641.9
air05	>21 646	>1.0	>3600.0	>2722.2	>21 911	>0.9	>3600.0	>2711.8	>23 683	>0.9	>3600.0	>2644.6	>17 095	>1.0	>3600.0	>2907.8
arki001	>20 847	—	>3600.0	>3565.4	>21 030	—	>3600.0	>3562.0	>54 354	—	>3600.0	>3508.5	>4 107	—	>3600.0	>3592.2
atlanta-ip	>1 265	—	>3600.0	>3294.4	—	—	—	—	>551	—	>3600.0	>3415.3	>1	—	>3600.0	>3587.8
bc1	>9 882	>92.7	>3600.0	>3429.7	>7 290	>93.9	>3600.0	>3474.7	>3 479	—	>3600.0	>1013.1	>2 419	>98.0	>3600.0	>3561.7

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
bell3a	362 609	0.0	558.1	465.1	362 608	0.0	526.3	432.8	362 620	0.0	680.1	585.7	>325 382	>0.0	>3600.0	>3406.1
bell5	408 929	0.0	495.9	444.3	409 007	0.0	457.0	405.3	408 938	0.0	571.5	520.2	>406 853	>0.0	>3600.0	>3470.1
bienst1	42 018	0.0	295.9	197.3	42 017	0.0	331.2	233.6	41 892	0.0	297.6	200.3	40 867	0.0	806.2	702.0
bienst2	>433 943	>0.5	>3600.0	>2144.5	>384 254	>3.0	>3600.0	>2312.1	>430 247	>0.6	>3600.0	>2147.0	>187 530	>15.8	>3600.0	>2950.8
binkar10_1	>123 485	—	>3600.0	>3413.2	>123 394	—	>3600.0	>3417.1	>125 010	—	>3600.0	>3414.9	>92 925	—	>3600.0	>3444.7
blend2	44 988	0.0	210.4	200.9	44 990	0.0	242.4	232.6	45 006	0.0	255.9	246.3	44 992	0.0	577.3	554.4
cap6000	>40 793	—	>3600.0	>3454.4	>33 967	—	>3600.0	>3480.6	>34 210	—	>3600.0	>3479.6	>26 952	—	>3600.0	>3491.3
dano3_3	40	0.0	121.5	72.3	40	0.0	394.8	345.6	40	0.0	95.0	45.9	40	0.0	406.0	372.4
dano3_4	193	0.0	418.4	247.3	193	0.0	756.6	585.6	193	0.0	254.4	83.0	193	0.0	1835.1	1713.7
dano3_5	>2 050	>0.1	>3600.0	>2390.3	>1 844	>0.1	>3600.0	>2503.1	>4 607	>0.0	>3600.0	>952.8	>274	>0.5	>3600.0	>3461.1
dano3mip	>1 792	—	>3600.0	>2056.9	>1 532	—	>3600.0	>2315.0	>1 874	—	>3600.0	>2260.1	>379	—	>3600.0	>3357.8
danoint	>93 852	>6.9	>3600.0	>2652.8	>77 898	>7.0	>3600.0	>2820.1	>156 110	>6.7	>3600.0	>2045.5	>41 684	>7.2	>3600.0	>3157.6
dcmulti	20 133	0.0	117.0	108.1	20 133	0.0	115.1	106.1	20 133	0.0	114.2	105.2	20 133	0.0	204.5	192.9
disctom	>15 982	—	>3600.0	>3441.2	>15 806	—	>3600.0	>3442.5	>17 755	—	>3600.0	>3413.0	>14 649	—	>3600.0	>3498.7
egout	60 871	0.0	121.6	112.2	60 871	0.0	116.8	107.3	60 871	0.0	129.4	120.2	60 871	0.0	350.8	335.6
eilD76	>58 995	>14.8	>3600.0	>3144.3	>60 693	>14.7	>3600.0	>3133.9	>68 896	>14.0	>3600.0	>3055.6	>49 543	>15.9	>3600.0	>3213.4
enigma	128 058	0.0	162.3	142.6	128 058	0.0	153.6	132.6	128 058	0.0	145.8	125.2	128 058	0.0	333.1	307.4
fast0507	>1 654	—	>3600.0	>2828.6	>1 973	—	>3600.0	>2738.2	>1 550	—	>3600.0	>2867.9	>1 090	—	>3600.0	>3107.7
fiber	>408 765	>246.5	>3600.0	>3373.4	>417 500	>246.3	>3600.0	>3372.6	>359 389	>247.7	>3600.0	>3405.3	>156 918	>259.2	>3600.0	>3421.4
fixnet6	>419 759	>407.1	>3600.0	>3492.0	>466 530	>404.9	>3600.0	>3476.7	>360 793	>410.6	>3600.0	>3505.6	>551 901	>401.6	>3600.0	>3403.9

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
flugpl	3 519	0.0	1.0	1.0	3 519	0.0	1.0	1.0	3 519	0.0	1.0	1.0	3 519	0.0	1.6	1.4
gen	34 100	0.0	395.9	369.8	34 100	0.0	396.4	370.3	34 100	0.0	428.5	401.8	34 100	0.0	619.0	585.8
gesa2-o	>247 639	>1.4	>3600.0	>3415.3	>246 754	>1.4	>3600.0	>3413.9	>204 990	>1.4	>3600.0	>3450.6	>110 485	>1.5	>3600.0	>3464.1
gesa2	>229 127	>1.3	>3600.0	>3419.0	>229 289	>1.3	>3600.0	>3412.7	>149 686	—	>3600.0	>3474.8	>99 994	—	>3600.0	>3473.8
gesa3	128 210	0.0	2584.3	2417.3	128 210	0.0	2610.3	2426.8	128 210	0.0	2757.1	2563.3	>87 201	>0.2	>3600.0	>3445.2
gesa3_o	178 437	0.0	3296.1	3076.8	178 437	0.0	3320.5	3107.0	178 437	0.0	2829.2	2618.0	>111 188	>0.2	>3600.0	>3441.7
glass4	>384 328	>362.5	>3600.0	>1771.0	>1 171 185	>237.5	>3600.0	>3235.1	>991 324	>467.5	>3600.0	>3277.4	>366 408	>467.5	>3600.0	>3413.4
gt2	>2 050 442	>202.3	>3600.0	>3363.0	>2 166 833	>202.3	>3600.0	>3349.7	>1 994 278	>202.4	>3600.0	>3367.6	>2 475 073	>202.1	>3600.0	>3278.4
harp2	>89 613	—	>3600.0	>3473.3	>75 872	—	>3600.0	>3487.9	>64 974	—	>3600.0	>3509.1	>87 192	—	>3600.0	>3344.8
irp	>12 497	>0.2	>3600.0	>3374.2	>12 330	>0.2	>3600.0	>3366.5	>9 574	>0.2	>3600.0	>3427.8	>9 747	>0.2	>3600.0	>3408.1
khb05250	6 606	0.0	29.2	27.2	6 606	0.0	28.1	26.0	6 606	0.0	28.1	26.1	6 606	0.0	42.8	39.7
l152lav	11 933	0.0	341.0	310.5	11 929	0.0	331.1	304.2	11 933	0.0	342.9	314.8	11 934	0.0	277.2	247.9
liu	>371 923	—	>3600.0	>3270.4	>292 146	—	>3600.0	>3307.5	>288 363	—	>3600.0	>3293.3	>155 958	—	>3600.0	>3339.0
lrn	>2 705	—	>3600.0	>2978.3	>2 898	—	>3600.0	>3194.9	>2 841	—	>3600.0	>2610.7	>2 591	—	>3600.0	>3418.7
lseu	795 963	0.0	735.5	642.7	795 963	0.0	696.4	603.5	795 955	0.0	857.6	766.4	795 963	0.0	866.5	773.5
manna81	>116 663	—	>3600.0	>2796.4	>148 729	—	>3600.0	>3084.5	>142 993	—	>3600.0	>3106.2	>58 587	—	>3600.0	>3364.9
markshare1	>5 110 920	—	>3600.0	>3190.7	>5 193 545	—	>3600.0	>3185.6	>6 421 663	—	>3600.0	>3089.2	>413 114	—	>3600.0	>3492.8
markshare1_1	>5 730 348	—	>3600.0	>3136.4	>5 707 276	—	>3600.0	>3145.2	>6 958 002	—	>3600.0	>3043.8	>367 517	—	>3600.0	>3512.4
markshare2	>3 341 270	—	>3600.0	>3310.6	>3 356 203	—	>3600.0	>3307.5	>4 390 772	—	>3600.0	>3220.9	>425 509	—	>3600.0	>3486.6
markshare2_1	>4 858 039	—	>3600.0	>3182.4	>4 935 983	—	>3600.0	>3180.2	>6 046 692	—	>3600.0	>3082.8	>424 657	—	>3600.0	>3482.2

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
markshare4_0	3 826 128	0.0	1556.4	1264.2	3 826 128	0.0	1592.2	1300.2	3 826 128	0.0	1264.5	975.8	>588 269	—	>3600.0	>3475.5
mas74	>238 507	>44.2	>3600.0	>3559.7	>222 882	>33.7	>3600.0	>3562.1	>942 942	>35.8	>3600.0	>3443.7	>141 089	>38.9	>3600.0	>3544.5
mas76	>328 188	>3.9	>3600.0	>3546.1	>304 821	>1.8	>3600.0	>3546.7	>889 555	>2.9	>3600.0	>3455.2	>172 382	>2.0	>3600.0	>3534.4
mas284	>92 668	>3.4	>3600.0	>3571.1	>79 559	>3.4	>3600.0	>3573.8	>448 212	>2.3	>3600.0	>3455.4	>113 862	>3.2	>3600.0	>3532.2
mik.250-20-75.1	>302 618	—	>3600.0	>3547.9	>298 488	—	>3600.0	>3550.0	>278 163	—	>3600.0	>3552.8	>308 826	—	>3600.0	>3518.8
mik.250-20-75.2	>287 060	—	>3600.0	>3551.8	>276 540	—	>3600.0	>3554.2	>273 599	—	>3600.0	>3554.5	>309 497	—	>3600.0	>3515.7
mik.250-20-75.3	>317 175	—	>3600.0	>3545.4	>301 644	—	>3600.0	>3547.8	>279 477	—	>3600.0	>3552.3	>315 537	—	>3600.0	>3518.8
mik.250-20-75.4	>326 176	—	>3600.0	>3544.2	>304 791	—	>3600.0	>3548.1	>296 038	—	>3600.0	>3550.7	>316 271	—	>3600.0	>3516.4
mik.250-20-75.5	>304 133	—	>3600.0	>3549.7	>292 111	—	>3600.0	>3552.4	>280 423	—	>3600.0	>3551.6	>310 380	—	>3600.0	>3517.8
misc03	1 561	0.0	4.0	3.5	1 561	0.0	4.0	3.3	1 561	0.0	4.8	4.1	1 559	0.0	4.6	4.1
misc07	368 179	0.0	2598.3	2292.7	368 179	0.0	2495.4	2187.4	368 182	0.0	2879.0	2573.5	365 354	0.0	3451.2	3125.1
mitre	>29 285	—	>3600.0	>3441.6	>30 525	—	>3600.0	>3439.1	>29 082	—	>3600.0	>3448.4	>30 430	—	>3600.0	>3453.5
mkc	>52 897	—	>3600.0	>3421.2	>55 348	—	>3600.0	>3413.6	>54 740	—	>3600.0	>3419.9	>46 055	—	>3600.0	>3399.9
mkc1	>68 462	>5.1	>3600.0	>3336.9	>60 222	>5.1	>3600.0	>3367.9	>56 798	>5.1	>3600.0	>3376.1	>81 500	>2.8	>3600.0	>3301.3
mod008	59 211	0.0	280.8	272.6	59 211	0.0	248.0	239.3	59 211	0.0	253.7	245.0	59 211	0.0	626.7	604.6
mod010	93 732	0.0	2915.5	2704.5	93 748	0.0	2993.7	2699.2	93 731	0.0	3427.5	3212.4	93 730	0.0	2653.2	2416.5
mod011	>22 293	>16.8	>3600.0	>3214.9	>22 989	>16.7	>3600.0	>3201.2	>21 147	>16.9	>3600.0	>3234.6	>18 787	>17.0	>3600.0	>3262.4
modglob	>579 175	>0.7	>3600.0	>3433.6	>560 373	>0.7	>3600.0	>3440.8	>648 464	>0.7	>3600.0	>3414.0	>250 647	>0.8	>3600.0	>3445.5
momentum1	>13	—	>3600.0	>46.0	>50	—	>3600.0	>95.9	>50	—	>3600.0	>2993.1	>23	—	>3600.0	>67.6
momentum2	>272	—	>3600.0	>1411.1	>274	—	>3600.0	>1836.3	>270	—	>3600.0	>2244.5	>4	—	>3600.0	>3593.4

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
msc98-ip	>4 666	—	>3600.0	>1651.5	>4 749	—	>3600.0	>1635.7	>1 966	—	>3600.0	>3202.1	>3	—	>3600.0	>3592.7
mzzv11	>10 769	—	>3600.0	>2335.9	>10 684	—	>3600.0	>2346.8	>9 097	—	>3600.0	>2486.4	>6 565	—	>3600.0	>2922.3
mzzv42z	>13 139	—	>3600.0	>3021.0	>12 981	—	>3600.0	>3025.9	>12 663	—	>3600.0	>2982.2	>6 156	—	>3600.0	>3223.9
neos1	>95 449	>187.9	>3600.0	>3313.5	>99 373	>187.9	>3600.0	>3301.9	>59 178	>193.8	>3600.0	>3419.9	>78 833	>190.8	>3600.0	>3358.3
neos2	>107 487	—	>3600.0	>3243.0	>112 005	—	>3600.0	>3233.7	>96 262	—	>3600.0	>3244.8	>55 430	—	>3600.0	>3422.7
neos3	>90 277	—	>3600.0	>3202.3	>90 748	—	>3600.0	>3198.5	>79 475	—	>3600.0	>3243.5	>73 245	—	>3600.0	>3289.0
neos5	>1 015 645	>4.3	>3600.0	>3350.2	>1 059 648	>4.3	>3600.0	>3339.2	>1 503 908	>3.9	>3600.0	>3242.8	>1 435 468	>3.4	>3600.0	>3282.3
neos6	>10 942	>2.4	>3600.0	>3073.9	>10 144	>1.2	>3600.0	>3144.4	>12 348	>1.2	>3600.0	>3011.5	>6 348	>2.4	>3600.0	>3307.9
neos7	>134 628	>204.5	>3600.0	>3084.5	>130 056	>208.7	>3600.0	>3111.1	>138 787	>200.5	>3600.0	>3068.5	>61 611	>201.3	>3600.0	>3314.9
neos8	>10 061	—	>3600.0	>2843.6	>9 952	—	>3600.0	>2837.7	>7 650	—	>3600.0	>3010.0	>937	—	>3600.0	>3511.8
neos9	>3 116	>3.3	>3600.0	>3081.0	>3 463	>3.3	>3600.0	>3027.4	>643	>3.3	>3600.0	>3482.9	>2 415	>3.3	>3600.0	>2983.4
neos10	>8 431	—	>3600.0	>2711.4	>9 663	—	>3600.0	>2743.2	>5 648	—	>3600.0	>2937.0	>1 318	—	>3600.0	>3478.5
neos11	32 006	0.0	2713.9	705.3	32 006	0.0	2709.1	704.8	32 004	0.0	2737.1	732.2	30 590	0.0	3082.0	1209.0
neos12	>30 361	—	>3600.0	>3774.1	>28 461	—	>3600.0	>3783.4	>30 031	—	>3600.0	>3776.2	>5 500	>48.7	>3600.0	>861.9
neos13	>17 503	>72.3	>3600.0	>2853.9	>18 372	>72.3	>3600.0	>2838.1	>15 971	>72.3	>3600.0	>2909.2	>263	—	>3600.0	>3587.4
neos16	>702 007	—	>3600.0	>2939.6	>693 855	—	>3600.0	>2951.7	>629 149	—	>3600.0	>3017.2	>433 120	—	>3600.0	>3159.8
neos20	>192 238	—	>3600.0	>2654.3	>205 920	—	>3600.0	>2747.0	>164 808	—	>3600.0	>2834.7	>9 891	—	>3600.0	>3554.0
neos21	>189 938	>20.2	>3600.0	>2283.4	>188 660	>20.3	>3600.0	>2285.6	>169 045	>21.4	>3600.0	>2422.1	>117 274	>25.8	>3600.0	>2744.1
neos22	>92 327	—	>3600.0	>3218.6	>104 029	—	>3600.0	>3295.3	>87 545	—	>3600.0	>3335.8	>69 204	—	>3600.0	>3374.3
neos23	>520 062	>161.6	>3600.0	>2992.9	>400 090	>161.6	>3600.0	>3128.7	>395 352	>151.2	>3600.0	>3127.8	>170 464	>175.2	>3600.0	>3341.6

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
neos616206	>407 862	—	>3600.0	>2964.4	>421 743	—	>3600.0	>2955.5	>356 452	—	>3600.0	>3043.1	>209 257	—	>3600.0	>3249.6
neos632659	>914 056	>20.3	>3600.0	>3307.4	>928 216	>20.3	>3600.0	>3301.8	>935 475	>20.3	>3600.0	>3297.6	>1 096 299	>20.3	>3600.0	>3260.1
neos648910	>548 131	—	>3600.0	>2989.9	>549 610	—	>3600.0	>3004.5	>518 611	—	>3600.0	>3045.1	>314 804	—	>3600.0	>3265.9
neos808444	>20 740	—	>3600.0	>3145.9	>17 844	—	>3600.0	>3060.6	>20 092	—	>3600.0	>3150.5	>1 154	—	>3600.0	>3576.9
neos818918	>129 227	>1.2	>3600.0	>2543.9	>127 167	>1.2	>3600.0	>2561.8	>137 003	>1.2	>3600.0	>2474.4	>45 979	>1.2	>3600.0	>3262.5
neos897005	86	0.0	682.3	340.2	86	0.0	761.1	373.2	86	0.0	1208.5	855.8	49	0.0	188.9	32.8
net12	>8 700	—	>3600.0	>1436.0	>8 562	—	>3600.0	>1436.2	>8 181	—	>3600.0	>1571.5	>5 843	—	>3600.0	>2194.1
noswot	>1 652 721	—	>3600.0	>3287.7	>1 131 678	—	>3600.0	>3388.1	>1 811 037	—	>3600.0	>3265.1	>289 668	—	>3600.0	>3472.3
ns1648184	>68 989	>4.1	>3600.0	>1632.0	>66 253	>4.3	>3600.0	>1629.1	>80 766	>4.0	>3600.0	>1295.6	>51 161	>4.1	>3600.0	>2278.4
ns1688347	>65 239	—	>3600.0	>3129.0	>63 235	—	>3600.0	>3142.3	>28 005	—	>3600.0	>3318.7	>23 413	—	>3600.0	>3441.6
ns1671066	>48 620	>107.4	>3600.0	>3447.2	>42 796	>107.4	>3600.0	>3426.1	>59 370	>107.4	>3600.0	>3413.1	>59 217	>107.1	>3600.0	>3409.8
ns1692855	>49 851	—	>3600.0	>2933.0	>46 872	—	>3600.0	>2964.6	>28 482	—	>3600.0	>3344.9	>5 903	—	>3600.0	>3551.8
nsrand-ipx	>23 885	>380.8	>3600.0	>3449.4	>23 966	>380.8	>3600.0	>3451.0	>24 406	>380.8	>3600.0	>3449.5	>10 340	>381.1	>3600.0	>3534.6
nug08	143	0.0	18.9	4.3	143	0.0	18.6	4.0	143	0.0	20.6	6.0	143	0.0	42.7	28.7
nw04	>1 500	>3.1	>3600.0	>3451.6	>1 588	>3.1	>3600.0	>3445.2	—	—	—	—	>3 230	>2.6	>3600.0	>3269.6
opt1217	>379 735	—	>3600.0	>3452.4	>378 586	—	>3600.0	>3453.7	>491 135	—	>3600.0	>3411.3	>895 535	—	>3600.0	>3212.0
p0033	2 670	0.0	1.0	1.0	2 670	0.0	1.0	1.0	2 670	0.0	1.1	1.0	2 670	0.0	1.3	1.1
p0201	5 788	0.0	19.5	17.0	5 780	0.0	17.6	15.2	5 780	0.0	19.6	17.1	5 780	0.0	30.7	28.2
p0282	>1 111 012	—	>3600.0	>3405.9	>1 128 368	—	>3600.0	>3399.1	>938 282	—	>3600.0	>3435.8	>241 671	—	>3600.0	>3487.1
p0548	>642 649	—	>3600.0	>3467.4	>688 984	—	>3600.0	>3456.7	>607 990	—	>3600.0	>3473.9	>322 071	—	>3600.0	>3453.5

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
p2756	>189 601	—	>3600.0	>3394.3	>200 158	—	>3600.0	>3383.3	>135 982	—	>3600.0	>3376.3	>177 300	—	>3600.0	>3347.6
pk1	>981 800	>35.6	>3600.0	>3408.3	>830 196	>221.0	>3600.0	>3442.3	>1 768 596	>1.1	>3600.0	>3245.1	>271 480	>325.3	>3600.0	>3494.5
pp08a	>1 876 334	>104.0	>3600.0	>3305.9	>1 907 653	>103.9	>3600.0	>3297.1	>1 767 142	>104.4	>3600.0	>3323.2	>1 881 898	>104.0	>3600.0	>3298.6
pp08aCUTS	>919 969	>51.4	>3600.0	>3264.4	>946 902	>51.4	>3600.0	>3254.8	>1 091 940	>51.1	>3600.0	>3203.7	>276 096	>53.8	>3600.0	>3434.9
prod1	>1 275 042	—	>3600.0	>3371.8	>1 269 327	—	>3600.0	>3376.4	>1 277 284	—	>3600.0	>3366.9	>202 926	—	>3600.0	>3502.0
prod2	>744 461	—	>3600.0	>3433.5	>736 806	—	>3600.0	>3433.9	>744 942	—	>3600.0	>3435.2	>139 155	—	>3600.0	>3520.4
protfold	>3 450	—	>3600.0	>227.2	>3 407	—	>3600.0	>255.6	>7 542	>84.2	>3600.0	>2209.2	>623	—	>3600.0	>2907.6
qap10	246	0.0	573.3	25.6	246	0.0	572.1	23.9	246	0.0	576.6	27.9	244	0.0	2639.2	2123.9
qiu	>191 218	>90.1	>3600.0	>2369.0	>193 732	>90.0	>3600.0	>2361.1	>205 625	>89.9	>3600.0	>2282.9	>1 700	—	>3600.0	>3586.8
qnet1	>183 960	>86.0	>3600.0	>3358.6	>173 613	>86.1	>3600.0	>3356.4	>184 951	>86.0	>3600.0	>3345.3	>122 323	>86.3	>3600.0	>3369.9
qnet1_o	>213 999	>3.9	>3600.0	>3308.6	>201 506	>4.1	>3600.0	>3339.5	>190 964	>5.2	>3600.0	>3341.1	>140 854	>14.2	>3600.0	>3346.6
ran8x32	>698 265	>5.4	>3600.0	>3427.8	>800 186	>5.4	>3600.0	>3401.5	>659 102	>5.5	>3600.0	>3438.6	>310 209	>6.1	>3600.0	>3435.9
ran10x26	>681 290	>6.3	>3600.0	>3418.3	>789 329	>6.2	>3600.0	>3385.5	>670 434	>6.3	>3600.0	>3418.2	>234 001	>8.4	>3600.0	>3365.5
ran12x21	>719 219	>6.9	>3600.0	>3432.1	>811 923	>6.8	>3600.0	>3411.0	>687 446	>6.9	>3600.0	>3441.2	>268 505	>11.2	>3600.0	>3449.9
ran13x13	>1 102 614	>9.1	>3600.0	>3392.4	>1 252 546	>8.3	>3600.0	>3362.6	>1 055 961	>9.1	>3600.0	>3400.6	>1 113 801	>9.1	>3600.0	>3357.2
rentacar	165	0.0	70.8	63.6	179	0.0	161.1	153.8	165	0.0	66.6	59.5	156	0.0	44.0	36.9
rgn	10 249	0.0	30.0	20.2	10 249	0.0	27.0	18.6	10 249	0.0	33.8	25.3	10 219	0.0	99.5	95.5
roll3000	>85 153	—	>3600.0	>3037.6	>86 620	—	>3600.0	>3026.1	>86 450	—	>3600.0	>3025.7	>38 841	—	>3600.0	>3330.6
rout	>384 739	>33.3	>3600.0	>3248.6	>367 363	>33.3	>3600.0	>3262.3	>422 680	>33.3	>3600.0	>3213.5	>248 562	>33.3	>3600.0	>3320.9
set1ch	>452 323	—	>3600.0	>3480.9	>477 229	—	>3600.0	>3475.1	>472 421	—	>3600.0	>3473.6	>181 095	—	>3600.0	>3472.9

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
seymour	>41 029	—	>3600.0	>1941.1	>44 774	—	>3600.0	>1849.1	>36 002	—	>3600.0	>2182.0	>12 647	—	>3600.0	>3075.2
seymour1	>33 269	>2.1	>3600.0	>1468.1	>34 850	>2.1	>3600.0	>1367.8	>29 228	>2.1	>3600.0	>1775.5	>13 252	>2.2	>3600.0	>2972.0
sp97ar	>451	—	>3600.0	>1262.7	>578	—	>3600.0	>1484.6	>544	—	>3600.0	>1594.2	>479	—	>3600.0	>1873.9
stein27	4 031	0.0	2.8	2.0	4 031	0.0	2.7	1.8	4 031	0.0	3.1	2.2	4 031	0.0	4.7	4.0
stein45	58 329	0.0	102.0	72.3	58 329	0.0	97.8	68.2	58 333	0.0	107.5	79.1	58 312	0.0	166.3	141.7
swath	>201 478	—	>3600.0	>4909.2	>181 969	—	>3600.0	>4898.0	>199 396	—	>3600.0	>4907.1	>32 398	>106.4	>3600.0	>3441.6
swath1	>72 068	—	>3600.0	>3717.9	>71 897	—	>3600.0	>3718.2	>70 862	—	>3600.0	>3717.7	>32 762	>9.7	>3600.0	>3453.4
swath2	>67 899	—	>3600.0	>3672.6	>63 808	—	>3600.0	>3672.7	>68 483	—	>3600.0	>3676.9	>33 197	>12.2	>3600.0	>3447.3
swath3	>73 206	—	>3600.0	>3714.8	>73 049	—	>3600.0	>3711.1	>72 357	—	>3600.0	>3711.8	>33 363	>15.6	>3600.0	>3447.5
t1717	>722	—	>3600.0	>2206.0	>709	—	>3600.0	>2200.4	>511	—	>3600.0	>2592.4	>633	—	>3600.0	>3314.7
timtab1	>1 499 402	—	>3600.0	>3330.7	>1 524 486	—	>3600.0	>3331.6	>1 125 343	—	>3600.0	>3401.3	>323 837	—	>3600.0	>3447.8
timtab2	>890 315	—	>3600.0	>3378.1	>896 678	—	>3600.0	>3377.5	>662 825	—	>3600.0	>3433.6	>251 485	—	>3600.0	>3440.0
tr12-30	>333 727	—	>3600.0	>3482.7	>354 363	—	>3600.0	>3480.5	>343 214	—	>3600.0	>3492.0	>142 773	—	>3600.0	>3479.7
vpm1	>1 150 067	>11.6	>3600.0	>3386.4	>1 302 083	>11.3	>3600.0	>3355.0	>1 041 362	>11.8	>3600.0	>3405.5	>1 394 444	>11.1	>3600.0	>3323.8
vpm2	>931 541	>28.2	>3600.0	>3386.0	>973 731	>28.1	>3600.0	>3372.5	>929 556	>28.3	>3600.0	>3384.1	>749 524	>29.1	>3600.0	>3374.8
all (153)																
geom. mean	52 069	—	1692.8	1187.7	52 866	—	1715.5	1227.8	52 210	—	1715.5	1308.8	23 913	—	1776.4	1496.8
sh. geom. mean	55 691	—	1822.1	1311.1	56 126	—	1847	1355.8	55 512	—	1838	1430.1	27 296	—	1906.6	1642.1
arithm. mean	391 229	—	2919.4	2375.7	397 555	—	2924.1	2404.3	435 280	—	2925.7	2447.2	176 655	—	2976.1	2690.3

continued on next page

Table 18: Detailed Results on Entire Problem Set (Continued)

	S:Active;P:Opt				S:Active;P:Arbitrary				S:All;P:Opt				Exact LP			
	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT	Nodes	Gap	Time	DBT
all optimal (30)																
geom. mean	4 447	—	91	54.6	4 459	—	97.7	59.6	4 446	—	97.2	59.5	3 590	—	117.5	85.4
sh. geom. mean	5 556	—	124.5	81.3	5 565	—	134.5	90	5 555	—	130.3	85.3	4 808	—	160.5	128.1
arithm. mean	59 611	—	445.9	301.9	59 612	—	469.5	321.4	59 607	—	491.2	347	59 316	—	646.8	523.1
all timeout-sol (46)																
geom. mean	166 127	11.6	—	3006.5	169 182	12	—	3051.9	182 187	0	—	2966	104 921	14.3	—	3243.1
sh. geom. mean	166 531	19.7	—	3006.8	169 607	20.2	—	3052.1	182 984	18.5	—	2966.4	105 840	22.7	—	3243.1
arithm. mean	413 816	63.1	—	3065.5	442 351	64.2	—	3103	480 705	64.4	—	3052.3	331 371	73.2	—	3258.1

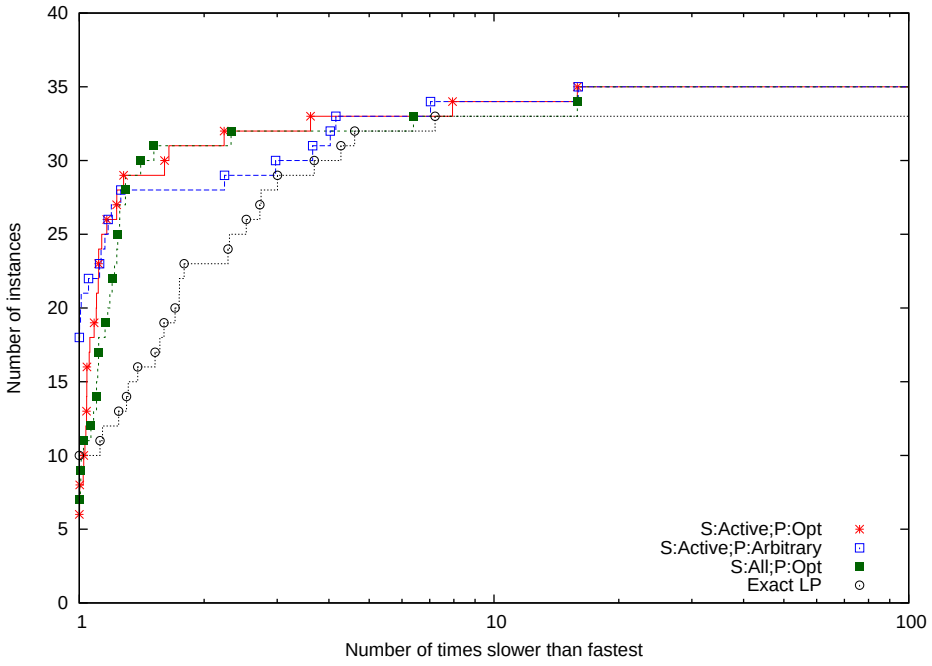


Figure 6: Overall Computation Time

All of the tests given so far have compared the methods on the entire problem set. However, for problems where all of the primal variables have reasonable upper and lower bounds the primal-bound-shift method described in Section 4.2.1 is expected to be the best dual bounding method. Therefore, we now focus on the subset of our problem instances that are missing some primal bounds. Figure 7 give a performance profile comparing the project-and-shift method with settings S:Active;P:Opt against the exact LP solver on this set of 87 problems. We see that there is an even more clear differentiation of the speed here. It is considerably faster on most of the instances and is able to solve several more problems within the one hour time limit.

The five problems where the conditions are not met for project and shift are: `swath`, `swath1`, `swath2`, `swath3` and `neos12`. We note that four out of five of these problem instances are coming from the same type of structure. This indicates that the conditions that the dual constraint matrix has full row rank and that there are no implied equalities

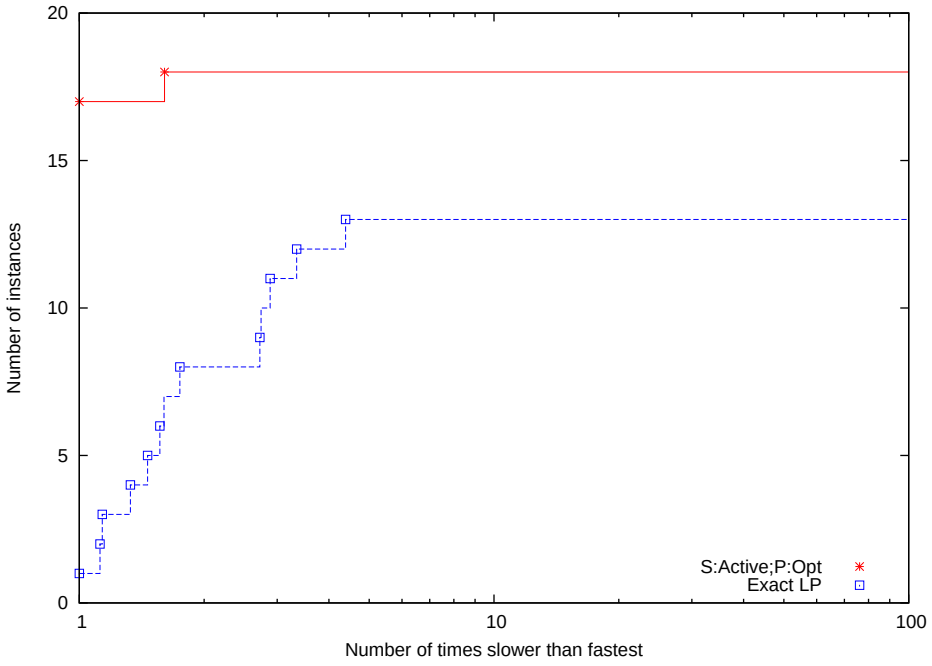


Figure 7: Overall Computation Time on Problems with Missing Bounds

is often satisfied on real-world problems.

4.5 Conclusions

We have described a new method for computing dual bounds. This method does not rely on primal variable bounds to compute the bound. It requires the exact solution to an LP at the root node and an exact LU factorization, but once this information is computed the LP bounds at each node of the branch-and-bound tree can be computed quickly.

One possible future direction would be looking at some combinations of the project-and-shift method and the methods of Althaus and Dumitriu [6]. First, the auxiliary LP to identify the polyhedral structure given in section 4.3.3 could be used in place of the iterative algorithm used in [6] to determine implied equalities of the system; this observation may speed up their method. It is also possible that the use of interval methods, as Althaus and Dumitriu have done, could be applied to the project-and-shift method to eliminate some of the exact computation and further increase the speed.

As a final remark, we note that building a code that dynamically chooses between project-and-shift, primal-bound-shift and other dual bounding strategies is faster than any of the single methods on their own. A computational study describing this type of combination can be found in [42].

CHAPTER V

FUTURE CHALLENGES

This chapter outlines future research directions and open questions related to exact mathematical programming.

5.1 Cutting Planes for Exact MIP

An important next step of developing a fast exact rational MIP solver is determining how cutting planes should be derived and managed. Cutting planes are an extremely useful tool for solving MIP problems. Commercial solvers typically use cutting planes together with branch-and-bound; disabling cuts can significantly slow down solution times.

Within a hybrid symbolic-numeric MIP solver it is not clear how cuts should be handled, other than the necessary requirement that all cuts must be valid. In the hybrid approach for exact MIP, much of the computational work is performed on a floating-point relaxation or approximation of the problem with the results verified or corrected using safe or symbolic computation. One possible approach for using cutting planes in this model is to generate cuts using exact rational arithmetic, adding them directly to the exact representation of the problem. Much of the computation associated with branch-and-bound would still be performed on the numerical approximations or relaxations of the MIP, including the added cuts. The disadvantage of this strategy is that deriving cuts using exact arithmetic may be very slow. For example, it may involve computing rows of a simplex tableau in rational arithmetic.

Some authors have studied ways to safely compute valid cuts using floating-point arithmetic. This approach is discussed by Neumaier and Shcherbina [118], and Cook et al. [40] perform a computational study of safe GMI cuts. The safe floating-point approach allows for cuts to be computed very quickly and avoids the possibility of incorrect rounding causing invalid cuts. However, within the hybrid framework it is not clear how these cuts should be added back to the exact description of the problem. Should they be added to the exact

description of the problem exactly as they were generated, or should the cut coefficients be transformed to other rational numbers?

A safe rounding procedure may produce cuts with numerically complicated coefficients (involving rational numbers with large bitsize). Cuts with complicated coefficients could cause difficultly whenever attempts are made to solve an exact LP. It is possible that cuts derived exactly using rational arithmetic might have special structure (i.e. integrality, half integrality or small rational representation) that could be destroyed in a safe rounding procedure. As we have seen in Chapters 2 and 3, the difficulty of computing basic LP solutions exactly is directly related to the bitsize of the final solution, so avoiding LPs with unnecessarily complicated coefficients is important.

It may be helpful to post-process cuts computed using a safe floating-point rounding procedure to have a simpler representation. This could be accomplished by scaling the inequalities and safely rounding coefficients up or down to give integer coefficients. The coefficients could also be safely outer approximated by continued fraction convergents with small denominators. This type of operation would weaken a cut by a small amount, but give it a less complicated rational representation.

We also remark that an exact separation routine may be able to derive the exact representation for facets of a problem. A safe floating-point rounding procedure could slightly perturb the inequality to no longer exactly represent a facet. It is not clear what effect this could have on the convergence behavior of the cutting-plane method.

There are several other questions related to which cuts should be separated and added that are not specific to the case of exact computation. For example, in recent studies such as those of Zanette et al. [145, 146], other desirable properties of cuts are discussed. These authors improve the behavior of a pure-cutting plane method by using the lexicographical dual simplex method which helps avoid generating many nearly parallel cuts. They also try to generate cuts with relatively small integer coefficients. These choices maintain LP basis matrices with smaller condition number and determinant, which helps avoid numerical problems.

Within an exact MIP solver it may be necessary to find a balance between many conflicting desirable characteristics. We would predict that the safe floating-point approach, such as that of Cook et al. [40], will be a successful tool for exact MIP, but questions regarding how these cuts should be mapped into the exact representation of the problem should be addressed. Implementation and experimentation are necessary to answer these questions, and could also lead to more general conclusions about the computational behavior of cutting planes.

We would predict that computing all cuts using exact arithmetic will likely lead to slower overall performance, but there are some situations where exact separation of cuts would be useful even if it is slower. Several recent computational studies [19, 46, 68] have had the goal of optimizing over the closure of various classes of cuts. These studies focused specifically on measuring the strength of specific cuts, not on increasing the speed of the MIP solver. The use of exact methods, or safe floating-point methods, in this type of study would allow stronger conclusions to be drawn. We would not make any strong prediction that the results of these studies were significantly affected by numerical mistakes. However, with the uncertainty of numerical computations used to derive cutting planes there is always some possibility of error.

5.1.1 Exact Separation of Two-Row Lattice-Free Cuts

The Gomory mixed-integer cut is one of the most successfully used cutting planes for MIP. Gomory developed a procedure to derive a violated cutting plane from a single row of a simplex tableau of a fractional solution. Recently, attention has focused on deriving cutting planes from two-row relaxations. There is hope that by using two-row relaxations new cuts may be derived that can build on the success of the single-row GMI cut.

Andersen et al. [7] characterize the facet-defining inequalities of the two row mixed-integer system and show that they can be described as split cuts [41] or intersection cuts [16] based on lattice-free triangles and quadrilaterals in $\mathbb{R}^2$. A *lattice-free* body contains no integer points in its interior. Detailed polyhedral studies by Basu et al. [21], Borozan and Cornuéjols [25], Cornuéjols and Margot [43], Dey and Wolsey [53, 54], and Zambelli [144]

have further developed the theory related to two-row mixed-integer systems.

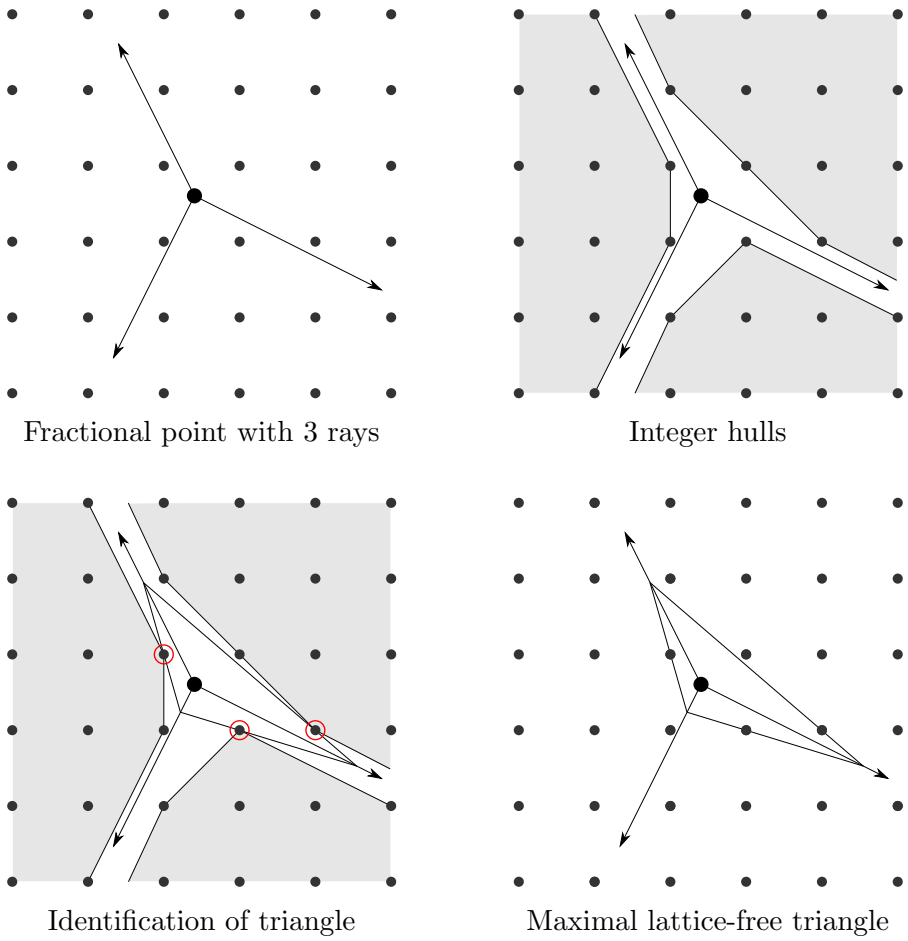


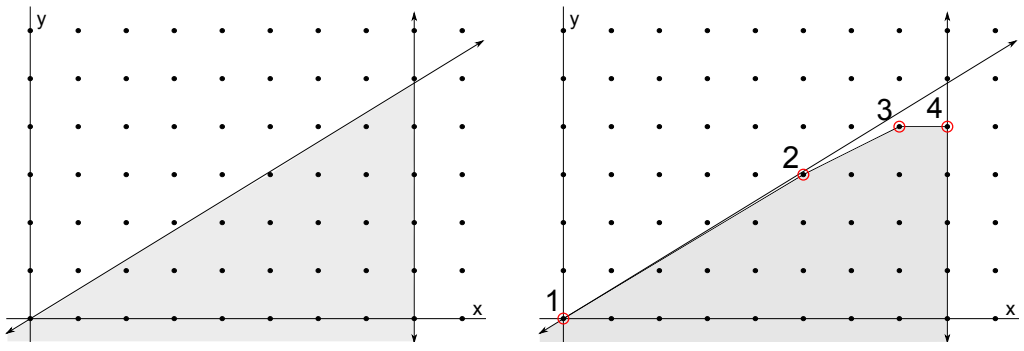
Figure 8: Generating a Maximal Lattice-Free Triangle

The separation problem for computing lattice-free triangle cuts involves identifying a maximal lattice-free triangle containing a specific fractional point f in its interior. The vertices of the triangle should lie on three of several open rays in $\mathbb{R}^2$ originating at f. A related separation problem is associated with separating quadrilateral cuts. Dey et al.[52] observed that all cuts coming from maximal lattice-free triangles can be enumerated by generating the integer hulls of the three convex cones formed by triples of rays r_1, r_2, r_3. Once the integer hulls of these cones are computed for a specific triple of rays, the unique maximal triangle corresponding to those rays can be computed geometrically, Figure 8 illustrates this procedure.

One component of this separation routine is to compute integer hulls of cones in $\mathbb{R}^2$ defined by two inequalities. A polynomial-time algorithm for this problem was developed by Harvey [81] as part of a procedure for computing integer hulls of general two-dimensional polyhedra. Harvey's algorithm begins by performing a unimodular transformation so that one inequality becomes vertical. The inequalities are then shifted so that the first integer point on the non-vertical inequality is the origin. The transformed region is of the form:

$$\begin{aligned} My - x &\leq 0 \\ x &\leq D \end{aligned}$$

After this transformation, the origin is an extreme point of the integer hull. To discover the next extreme point of the integer hull closest to the origin it is necessary to determine the integer point (x, y) where y/x is the best under approximation of M with $1 \leq x \leq D$. This problem is equivalent to finding the odd principal convergent of the rational number M with denominator closest to, but no larger than, D. The odd principal convergents of M can be efficiently computed by applying the Extended Euclidean Algorithm to the numerator and denominator of M. After the next extreme point is identified, the inequalities are shifted to position this extreme point at the origin. The process is repeated until D is shifted to zero, identifying the final extreme point of the integer hull. The output of the EEA can be reused at each iteration, so it is only applied once.



Two-dimensional cone Integer hull

Figure 9: Integer Hull of the Transformed Cone

Figure 9 illustrates the result of the algorithm on a problem whose integer hull has four

extreme points. The extreme points are labeled in the order which the algorithm would identify them. We have implemented Harvey's algorithm for cones to produce the exact rational description of the integer hulls, using the GMP library [72] for multiple-precision arithmetic. Since this algorithm is based on the Euclidean Algorithm, the core computation is similar to the methods used for the rational reconstruction algorithms associated with Theorems 2.2.1 and 2.2.2.

A number of recent computational studies of Basu et al. [20] and Dey et al. [52] have focused on evaluating the effectiveness of these types of cuts computationally. In both cases the authors separate cuts in a heuristic manner, possibly weakening their conclusions. Initial exact computational tests have been performed in collaboration with K. Chen, W. Cook and R. Fukasawa. Detailed algorithmic and computational results are expected to appear in the PhD Thesis of K. Chen.

5.2 Irrational and Nonlinear Problems

LP, IP and MIP problems with rational input data are guaranteed to have optimal solutions with a rational description. The techniques for computing exact solutions that have been developed and used in this thesis have heavily exploited the properties of the rational numbers. Other classes of problems, involving irrational input data or nonlinear constraints, may require computation over the real numbers in order to find exact solutions. Some of the LPs arising in the proof of the Kepler Conjecture involve irrational numbers [120]. Computation over the real numbers is significantly more difficult than over the rational numbers; even storing the symbolic representations of irrationals poses a challenge. Because of these challenges we would imagine that interval methods, such as those discussed in Section 1.2.4, may be a good way to handle these types of problems.

Methods for computing rigorous bounds on LP/MIP objective values have been developed by Jansson [87], Keil et al. [93] and Althaus and Dumitriu [6]. Techniques for computing rigorous bounds by using interval methods are applicable to real numbers without any extra challenge, because numbers are not stored exactly. Several studies have also considered techniques for generating safe bounds for convex optimization problems.

Borradaile and Hentenryck [26] study safe methods for computing linear under and over estimators of functions for applications in global optimization. Jansson [88] describes fast methods for computing rigorous objective bounds for convex optimization problems, some of which can be computed quickly by post-processing a solution returned by a numerical solver. Jansson et al. [89] consider methods for computing rigorous error bounds on semidefinite programming problems and LPs.

5.3 Applications

In this section we discuss two mathematical applications for exact integer programming. We describe how integer programming models could be used as a tool for proving bounds for Ramsey numbers or for factoring integers.

The Ramsey Number $R(n, m)$ is defined to be the smallest number r for which every graph on r nodes contains either a clique of size n or a stable set of size m [56]. The question of computing low Ramsey numbers is a difficult and actively studied problem. It is currently known that $R(3, 3) = 6, R(4, 4) = 18$ and $43 \leq R(5, 5) \leq 49$ [67, 107]. Table 19 gives some currently known bounds on some small Ramsey numbers. Discovery of many of the known upper bounds for specific Ramsey numbers have involved massive computations, an up to date survey of this problem is maintained by Radziszowski [125].

Table 19: Some Known Bounds for $R(n, m)$

$R(n,m)$	3	4	5	6
3	6	9	14	18
4	9	18	25	[35,41]
5	14	25	[43,49]	[58,87]
6	18	[35,41]	[58,87]	[102,165]

For integers m, n, k there is a natural representation of all counterexamples to the statement $R(n, m) \leq k$ as an integer programming feasibility problem. In this case, a feasible solution to the following set of constraints is a certificate that $R(n, m) > k$ and infeasibility indicates that $R(n, m) \leq k$. The binary variables x_{ij} are decision variables determining if

the edge (i, j) is in graph with vertex set $K = \{1, \ldots, k\}$

$$\begin{aligned}
\sum_{i,j \in S,\ i<j} x_{ij} \leq & \quad \binom{n}{2} - 1 \quad \forall S \subseteq K, |S| = n \\
\sum_{i,j \in S,\ i<j} x_{ij} \geq & \quad 1 \quad \forall S \subseteq K, |S| = m \\
x_{ij} & \quad binary \quad i, j \in K.
\end{aligned}$$

Solving this feasibility problem directly would not be practical for many reasons; the number of constraints is exponential in k and the problem structure is highly symmetric. In order to prove infeasibility of an instance, exact computation is also necessary. In order to use this type of model some possible approaches would be to derive valid cutting planes that could be added to the problem, develop symmetry breaking inequalities and consider problem specific branching rules.

Another possible application for exact integer programming is to compute integer factorizations. Factoring integers is a fundamental mathematical problem. The difficulty of this problem is the basis for modern crypto-systems, but it is not known to be $\mathcal{NP}$-hard [101]. We will describe a formulation of the integer factoring problem developed by Cook, Kannan and Lovász. If N is the number to be factored, they consider the variables X, Y and formulate $N = XY$ as a MIP. In order to represent this nonlinear constraint with linear constraints they use binary variables $x_0, x_1, \ldots, x_{t-1}$ representing the bitwise description of X, where $t = \lfloor \log_2 N \rfloor$. Introducing additional variables $v_0, \ldots, v_{t-1}$ we can use the set of constraints:

$$\begin{aligned}
v_{t-1} &= x_{t-1}Y \\
v_{t-2} &= 2v_{t-1} + x_{t-2}Y \\
&\ldots \\
v_0 &= 2v_1 + x_0 Y
\end{aligned}$$

Each one of these nonlinear constraints can be expressed linearly, $v_0 = 2v_1 + x_0 Y$ can be expressed as

$$\begin{aligned}
2v_1 \leq & \quad v_0 \quad \leq 2v_1 + Y \\
2v_1 + Y - (1 - x_0)N \leq & \quad v_0 \quad \leq 2v_1 + Nx_0
\end{aligned}$$

This is referred to as the Horner multiplication system, and any feasible solution will satisfy $v_0 = XY$. In order to represent this problem without requiring large coefficients Cook, Kannan and Lovász adjusted the formulation using the Chinese Remainder Theorem. The adjusted formulation would express several constraints $N_i = X_i Y_i \bmod p_i$ for several primes p_i where $N_i = N \bmod p_i$ and $p_1 \cdots p_n \geq N$. Using this formulation avoids large coefficients and does not have a large number of constraints, but creates problems that are difficult to solve and are prone to numerical problems. There are other ways to formulate this problem as a MIP, instead of creating binary variables for the bit representation of Y and using a single integer variable for Y, one could represent both X and Y using binary variables describing their bitwise representation. There is no guarantee that mixed-integer-programming techniques will lead to competitive factoring algorithms. However, development of a fast exact MIP solver will allow further computational development of these ideas without numerical problems causing incorrect results. Some next steps could include studying the polyhedral structure of this model and the development and testing of specialized cutting planes.

5.4 Final Remarks

This chapter has outlined some future directions for research involving exact precision mathematical programming. There are still many open questions regarding the best computational strategies for finding exact solutions to optimization problems. It remains to be seen how cutting planes should be handled within an exact IP/MIP solver and how the speed of such a solver will compare with inexact numerical solvers using cutting planes. There are many other challenges which include solving problems involving nonlinear constraints or irrational input data. There are also exciting new application areas where more mature tools for exact precision optimization could prove useful.

REFERENCES

[1] Abbott, J., Bronstein, M., and Mulders, T., "Fast deterministic computation of determinants of dense matrices," in *ISSAC '99: Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 197–204, ACM, 1999.

[2] Abbott, J. and Mulders, T., "How tight is Hadamard's bound?," *Experiment. Math.*, vol. 10, no. 3, pp. 331–336, 2001.

[3] Achterberg, T., *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[4] Achterberg, T., "SCIP: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, pp. 1–41, 2009.

[5] Achterberg, T., Koch, T., and Martin, A., "MIPLIB 2003," *Operations Research Letters*, vol. 34, pp. 361–372, 2006.

[6] Althaus, E. and Dumitriu, D., "Fast and accurate bounds on linear programs," *Proceedings of 8th International Symposium on Experimental Algorithms (SEA 2009)*, vol. 5526, pp. 40–50, 2009.

[7] Andersen, K., Louveaux, Q., Weismantel, R., and Wolsey, L., "Inequalities from two rows of a simplex tableau," in *Integer Programming and Combinatorial Optimization*, Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 1–15, Springer-Verlag, 2007.

[8] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D., *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[9] Applegate, D., Bixby, R., Chvátal, V., Cook, W., Espinoza, D., Goycoolea, M., and Helsgaun, K., "Certification of an optimal TSP tour through 85,900 cities," *Operations Research Letters*, vol. 37, no. 1, pp. 11 – 15, 2009.

[10] Applegate, D., Cook, W., and Dash, S., "QSopt." `http://www.isye.gatech.edu/~wcook/qsopt/`, 2005.

[11] Applegate, D., Cook, W., Dash, S., and Espinoza, D., "QSopt_ex." `http://www.dii.uchile.cl/~daespino/ESolver_doc/main.html`, 2007.

[12] Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J., *The Traveling Salesman Problem: A Computational Study*. Princeton, New Jersey, USA: Princeton University Press, 2006.

[13] Applegate, D. L., Cook, W., Dash, S., and Espinoza, D. G., "Exact solutions to linear programming problems," *Operations Research Letters*, vol. 35, pp. 693–699, 2007.

[14] Bailey, D. and Borwein, J., *Mathematics by Experiment: Plausible Reasoning in the 21st Century*. Natick, MA, USA: AK Peters, 2008.

[15] Bailey, D. and Borwein, J., "Exploratory experimentation and computation," *Preprint*, 2010.

[16] Balas, E., "Intersection cuts–a new type of cutting planes for integer programming," *Operations Research*, vol. 19, no. 1, pp. 19–39, 1971.

[17] Balas, E. and Bonami, P., "Generating lift-and-project cuts from the LP simplex tableau: open source implementation and testing of new variants," *Mathematical Programming Computation*, vol. 1, pp. 165–199, 2009.

[18] Balas, E., Ceria, S., and Cornuéjols, G., "A lift-and-project cutting plane algorithm for mixed 0–1 programs," *Mathematical Programming*, vol. 58, pp. 295–324, 1993.

[19] Balas, E. and Saxena, A., "Optimizing over the split closure," *Mathematical Programming*, vol. 113, pp. 219–240, 2008.

[20] Basu, A., Bonami, P., Cornuéjols, G., and Margot, F., "Experiments with two-row cuts from degenerate tableaux," *INFORMS Journal on Computing*, To Appear.

[21] Basu, A., Conforti, M., Cornuéjols, G., and Zambelli, G., "Maximal lattice-free convex sets in linear subspaces," *Mathematics of Operations Research*, vol. 35, pp. 704 – 720, 2010.

[22] Berlekamp, E. R., "Factoring polynomials over finite fields," *Bell System Technical Journal*, vol. 46, pp. 1853–1859, 1967.

[23] Bixby, R., "Solving real-world linear programs: A decade and more of progress," *Operations Research*, vol. 50, no. 1, pp. 3–15, 2002.

[24] Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P., "An updated mixed integer programming library: MIPLIB 3.0," *Optima*, vol. 58, pp. 12–15, 1998.

[25] Borozan, V. and Cornuéjols, G., "Minimal valid inequalities for integer constraints," *Mathematics of Operations Research*, vol. 34, pp. 538–546, 2009.

[26] Borradaile, G. and Hentenryck, P., "Safe and tight linear estimators for global optimization," *Mathematical Programming*, vol. 102, pp. 495–517, 2005.

[27] Borwein, J. and Crandall, R., "Closed forms: What they are and why they matter," *Preprint*, 2010.

[28] Bright, C., "Vector rational number reconstruction," Master's thesis, School of Mathematics, University of Waterloo, 2010.

[29] Buchheim, C., Chimani, M., Ebner, D., Gutwenger, C., Jünger, M., Klau, G., Mutzel, P., and Weiskircher, R., "A branch-and-cut approach to the crossing number problem," *Discrete Optimization*, vol. 5, no. 2, pp. 373 – 388, 2008.

[30] Cabay, S., "Exact solution of linear equations," in *SYMSAC '71: Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, (New York, NY, USA), pp. 392–398, ACM, 1971.

[31] Chen, L. and Monagan, M., "Algorithms for solving linear systems over cyclotomic fields," *Submitted*, 2008.

[32] Chen, Z., "A BLAS based C library for exact linear algebra over integer matrices," Master's thesis, School of Computer Science, University of Waterloo, 2005.

[33] Chen, Z. and Storjohann, A., "A BLAS based C library for exact linear algebra on integer matrices," in *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 92–99, ACM, 2005.

[34] Cheng, E. and Steffy, D., "Clinching and elimination of playoff berth in the NHL," *International Journal of Operations Research*, vol. 5, no. 3, pp. 187–192, 2008.

[35] Chimani, M., Mutzel, P., and Bomze, I., "A new approach to exact crossing minimization," in *Algorithms - ESA 2008* (Halperin, D. and Mehlhorn, K., eds.), vol. 5193 of *Lecture Notes in Computer Science*, pp. 284–296, Springer Berlin / Heidelberg, 2008.

[36] Chvátal, V., *Linear Programming*. New York, NY, USA: W. H. Freeman and Company, 1983.

[37] Cohen, H., *A Course in Computational Algebraic Number Theory*. New York, NY, USA: Springer, 2000.

[38] Collins, G. E. and Encarnación, M. J., "Efficient rational number reconstruction," *Journal of Symbolic Computation*, vol. 20, pp. 287–297, 1995.

[39] Concorde. `http://www.tsp.gatech.edu`, 2010.

[40] Cook, W., Dash, S., Fukasawa, R., and Goycoolea, M., "Numerically safe Gomory mixed-integer cuts," *INFORMS Journal on Computing*, To Appear.

[41] Cook, W., Kannan, R., and Schrijver, A., "Chvátal closures for mixed integer programming problems," *Mathematical Programming*, vol. 47, pp. 155–174, 1990.

[42] Cook, W., Koch, T., Steffy, D., and Wolter, K., "An exact rational mixed-integer programming solver," Submitted.

[43] Cornuéjols, G. and Margot, F., "On the facets of mixed integer programs with two integer variables and two constraints," *Mathematical Programming*, vol. 120, pp. 429–456, 2009.

[44] Dantzig, G., "Maximization of a linear function of variables subject to linear inequalities," *Activity Analysis of Production and Allocation*, pp. 339–347, 1951.

[45] Dash, S. and Goycoolea, M., "A heuristic to generate rank-1 GMI cuts," *Technical Report, IBM*, 2010.

[46] Dash, S., Günlük, O., and Lodi, A., "On the MIR closure of polyhedra," in *Integer Programming and Combinatorial Optimization* (Fischetti, M. and Williamson, D., eds.), vol. 4513 of *Lecture Notes in Computer Science*, pp. 337–351, Springer Berlin / Heidelberg, 2007.

[47] de Loera, J., Lee, J., Malkin, P., and Margulies, S., "Hilbert's nullstellensatz and an algorithm for proving combinatorial infeasibility," in *ISSAC '08: Proceedings of the Twenty-First International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 197–206, ACM, 2008.

[48] de Loera, J., Lee, J., Margulies, S., and Onn, S., "Expressing combinatorial optimization problems by systems of polynomial equations and Hilbert's nullstellensatz," *Combinatorics, Probability and Computing*, vol. 18, pp. 551–582, 2009.

[49] de Oliveira Filho, F. M. and Vallentin, F., "Fourier analysis, linear programming, and densities of distance avoiding sets in $\mathbb{R}^n$," *J. Eur. Math. Soc.*, vol. 12, pp. 1417–1428, 2010.

[50] de Vries, S. and Vohra, R., "Combinatorial Auctions: A Survey," *INFORMS Journal on Computing*, vol. 15, no. 3, pp. 284–309, 2003.

[51] Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H., "A supernodal approach to sparse partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[52] Dey, S., Lodi, A., Tramontani, A., and Wolsey, L., "Experiments with two row tableau cuts," in *Integer Programming and Combinatorial Optimization* (Eisenbrand, F. and Shepherd, F., eds.), vol. 6080 of *Lecture Notes in Computer Science*, pp. 424–437, Springer Berlin / Heidelberg, 2010.

[53] Dey, S. and Wolsey, L., "Lifting integer variables in minimal inequalities corresponding to lattice-free triangles," in *IPCO'08: Proceedings of the 13th international conference on Integer programming and combinatorial optimization*, (Berlin, Heidelberg), pp. 463–475, Springer-Verlag, 2008.

[54] Dey, S. and Wolsey, L., "Constrained infinite group relaxations of MIPs," *Technical Report, available at Optimization Online*, 2009.

[55] Dhiflaoui, M., Funke, S., Kwappik, C., Mehlhorn, K., Seel, M., Schömer, E., Schulte, R., and Weber, D., "Certifying and repairing solutions to large LPs: How good are LP-solvers?," in *SODA '03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, (Philadelphia, PA, USA), pp. 255–256, SIAM, 2003.

[56] Diestel, R., *Graph Theory*. Berlin: Springer, 2006.

[57] Dixon, J. D., "Exact solution of linear equations using p-adic expansion," *Numerische Mathematik*, vol. 40, pp. 137–141, 1982.

[58] Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. S., "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[59] Dowson, M., "The Ariane 5 software failure," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, p. 84, 1997.

[60] Dumas, J.-G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B. D., Turner, W. J., and Villard, G., "LinBox: A generic library for exact linear algebra," in *Mathematical Software: ICMS 2002* (Cohen, A. M., Gao, X.-S., and Takayama, N., eds.), (Singapore), pp. 40–50, World Scientific, 2002.

[61] Dumas, J.-G., Giorgi, P., and Pernet, C., "Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages," *ACM Transactions on Mathematical Software*, vol. 35, no. 3, p. Article 19, 2008.

[62] Dumas, J.-G., Saunders, B. D., and Villard, G., "On efficient sparse integer matrix smith normal form computations," *Jounal of Symbolic Computation*, vol. 32, pp. 71–99, 2001.

[63] Dumas, J.-G. and Villard, G., "Computing the rank of large sparse matrices over finite fields," in *CASC'2002 : Computer Algebra in Scientific Computing*, pp. 63–74, 2002.

[64] Eberly, W., Giesbrecht, M., Giorgi, P., Storjohann, A., and Villard, G., "Solving sparse rational linear systems," in *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, (New York, NY, USA), pp. 63–70, ACM, 2006.

[65] Emiris, I. Z., "A complete implementation for computing general dimensional convex hulls," *International Journal of Computational Geometry and Applications*, vol. 8, pp. 223–253, 1998.

[66] Espinoza, D. G., *On Linear Programming, Integer Programming and Cutting Planes*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 2006.

[67] Exoo, G., "A lower bound for R(5,5)," *Journal of Graph Theory*, vol. 13, pp. 97–98, 1989.

[68] Fischetti, M. and Lodi, A., "Optimizing over the first Chvátal closure," *Mathematical Programming*, vol. 110, pp. 3–20, 2007.

[69] Freund, R. M., Roundy, R., and Todd, M. J., "Identifying the set of always-active constraints in a system of linear inequalities by a single linear program," *Technical Report, Sloan School of Management, MIT*, 1985.

[70] GAO: The United States General Accounting Office, "GAO/IMTEC 92-26 patriot missile defense software problem led to system failure at Dhahran, Saudi Arabia." Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives, 1992.

[71] Gay, D. M., "Electronic mail distribution of linear programming test problems," *COAL Newsletter*, vol. 13, pp. 10–12, 1985.

[72] GMP, "GNU multiple precision arithmetic library, version 4.2." `http://gmplib.org`, 2008.

[73] Goldberg, D., "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[74] Golub, G. and van Loan, C., *Matrix Computations*. Baltimore, Maryland, USA: Johns Hopkins University Press, 1983.

[75] Goodwin, D. and Wilken, K., "Optimal and near-optimal global register allocation using 0-1 integer programming," *Software: Practice and Experience*, vol. 26, no. 8, 1996.

[76] Gould, N. I. M., Scott, J. A., and Hu, Y., "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, p. Article 10, 2007.

[77] Gu, Z. personal communication, 2010.

[78] Hales, T. C., "A proof of the Kepler conjecture," *Annals of Mathematics*, vol. 162, pp. 1065–1185, 2005.

[79] Hales, T. C., "The flyspeck project," 2010.

[80] Hales, T., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., and Zumkeller, R., "A revision of the proof of the Kepler conjecture," *Discrete and Computational Geometry*, vol. 44, pp. 1–34, 2010.

[81] Harvey, W., "Computing two-dimensional integer hulls," *SIAM Journal on Computing*, vol. 28, pp. 2285–2299, 1999.

[82] Held, S., Sewell, E., and Cook, W., "Safe lower bounds for graph coloring," *Preprint*, 2010.

[83] Hicks, I. and McMurray, N., "The branchwidth of graphs and their cycle matroids," *Journal of Combinatorial Theory Series B*, vol. 97, pp. 681–692, 2007.

[84] Hladky, J., Král, D., and Norine, S., "Counting flags in triangle-free digraphs," *Preprint*, 2010.

[85] IBM, *CPLEX V12.1, User's Manual for CPLEX*. IBM ILOG, 2009.

[86] IEEE, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, aug. 2008.

[87] Jansson, C., "Rigorous lower and upper bounds in linear programming," *SIAM Journal on Optimization*, vol. 14, no. 3, pp. 914–935, 2004.

[88] Jansson, C., "A rigorous lower bound for the optimal value of convex optimization problems," *Journal of Global Optimization*, vol. 28, no. 1, pp. 121–137, 2004.

[89] Jansson, C., "Rigorous error bounds for the optimal value in semidefinite programming," *SIAM Journal on Numerical Analysis*, vol. 46, no. 1, pp. 180–200, 2007.

[90] Kaltofen, E., "An output-sensitive variant of the baby steps/giant steps determinant algorithm," in *ISSAC '02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 138–144, ACM, 2002.

[91] Kaltofen, E. and Rolletschek, H., "Computing greatest common divisors and factorizations in quadratic number fields," *Mathematics of Computation*, vol. 53, no. 188, pp. 697–720, 1989.

[92] Kaltofen, E. and Saunders, B. D., "On Wiedemann's method of solving sparse linear systems," in *Proceedings of the Ninth International Symposium on Applied, Algebraic Algorithms, Error-Correcting Codes, Lecture Notes in Computer Science 539*, (Heidelberg, Germany), pp. 29–38, Springer, 1991.

[93] Keil, C. and Jansson, C., "Computational experience with rigorous error bounds for the netlib linear programming library," *Reliable Computing*, vol. 12, pp. 303–321, 2006.

[94] Khodadad, S. and Monagan, M., "Fast rational function reconstruction," in *ISSAC '06: Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 184–190, ACM, 2006.

[95] Koch, T., "The final NETLIB-LP results," *Operations Research Letters*, vol. 32, pp. 138–142, 2004.

[96] Krishnamurthy, E., Rao, T., and Subramanian, K., "p-adic arithmetic procedures for exact matrix computations," *Proc. of the Indian Academy of Sciences, Series A*, vol. 82, pp. 165–175, 1975.

[97] Kwappik, C., "Exact Linear Programming," Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1998.

[98] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T., "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[99] Lee, E. and Zaider, M., "Operations Research Advances Cancer Therapeutics," *INTERFACES*, vol. 38, no. 1, pp. 5–25, 2008.

[100] Lehmer, D. H., "Euclid's algorithm for large numbers," *The American Mathematical Monthly*, vol. 45, no. 4, pp. 227–233, 1938.

[101] Lenstra, A. K., "Integer factoring," *Designs, Codes and Cryptography*, vol. 19, pp. 101–128, 2000.

[102] Leveson, N. and Turner, C., "An investigation of the Therac-25 accidents," *Computer (IEEE)*, vol. 26, pp. 18 –41, July 1993.

[103] Lichtblau, D., "Half-GCD and fast rational recovery," in *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 231–236, ACM, 2005.

[104] Margot, F., "Testing cut generators for mixed-integer linear programming," *Mathematical Programming Computation*, vol. 1, pp. 69–95, 2009.

[105] Margulies, S., *Computer Algebra, Combinatorics, and Complexity: Hilbert's Nullstellensatz and NP-complete Problems*. PhD thesis, Department of Computer Science, University of California, Davis, 2008.

[106] Massey, J. L., "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, pp. 122–127, 1969.

[107] McKay, B. and Radziszowski, S., "R(4,5)=25," *Journal of Graph Theory*, vol. 19, pp. 309–322, 1995.

[108] Mehrotra, A. and Trick, M., "A column generation approach for graph coloring," *INFORMS Journal on Computing*, vol. 8, no. 4, pp. 344–354, 1996.

[109] Mészáros, C., "LPtestset." `http://www.sztaki.hu/~meszaros/public_ftp/lptestset/`, 2006.

[110] Mittelmann, H. D., "LPtestset." `http://plato.asu.edu/ftp/lptestset/`, 2006.

[111] Mittelmann, H. D., "Benchmarks for Optimization Software." `http://plato.asu.edu/bench.html`, 2010.

[112] Moenck, R. and Carter, J., "Approximate algorithms to derive exact solutions to systems of linear equations," in *Symbolic and Algebraic Computation* (Ng, E., ed.), vol. 72 of *Lecture Notes in Computer Science*, pp. 65–73, Springer Berlin / Heidelberg, 1979.

[113] Monagan, M., "Maximal quotient rational reconstruction: an almost optimal algorithm for rational reconstruction," in *ISSAC 2004: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 243–249, ACM, 2004.

[114] Mulders, T. and Storjohann, A., "Diophantine linear system solving," in *ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, (New York, NY, USA), pp. 181–188, ACM, 1999.

[115] Mulders, T. and Storjohann, A., "Certified dense linear system solving," *Journal of Symbolic Computation*, vol. 37, pp. 485–510, 2004.

[116] Nemhauser, G. and Wolsey, L., *Integer and Combinatorial Optimization*. New York, NY, USA: Wiley-Interscience, 1988.

[117] NETLIB. Linear programming library. `http://www.netlib.org/lp/`.

[118] Neumaier, A. and Shcherbina, O., "Safe bounds in linear and mixed-integer linear programming," *Mathematical Programming*, vol. 99, no. 2, pp. 283–296, 2004.

[119] Nuseibeh, B., "Ariane 5: Who dunnit?," *IEEE Software*, vol. 14, pp. 15–16, 1997.

[120] Obua, S., *Flyspeck II: the basic linear programs*. PhD thesis, Technische Universität München, 2008.

[121] Obua, S. and Nipkow, T., "Flyspeck II: the basic linear programs," *Annals of Mathematics and Artificial Intelligence*, vol. 56, no. 3-4, pp. 245–272, 2009.

[122] Ordóñez, F. and Freund, R., "Computational experience and the explanatory value of condition measures for linear optimization," *SIAM J. on Optimization*, vol. 14, no. 2, pp. 307–333, 2003.

[123] Pan, V. Y. and Wang, X., "Acceleration of Euclidean algorithm and rational number reconstruction," *SIAM Journal of Computing*, vol. 32, pp. 548–556, 2003.

[124] Pan, V. Y. and Wang, X., "On rational number reconstruction and approximation," *SIAM Journal of Computing*, vol. 33, pp. 502–503, 2004.

[125] Radziszowski, S., "Small ramsey numbers (revision 12)," *The Electronic Journal of Combinatorics*, 2009.

[126] Reinelt, G., "TSPLIB–a traveling salesman library," *ORSA Journal on Computing*, vol. 3, pp. 376–384, 1991.

[127] Rosen, K., *Elementary Number Theory*. New York: Addison Wesley Longman, 2000.

[128] Schenk, O. and Gärtner, K., "Solving unsymmetric sparse systems of linear equations with PARDISO," *Journal of Future Generation Computer Systems*, vol. 3, no. 20, pp. 475–487, 2004.

[129] Schonhage, A., "Schnelle berechnung von kettenbruchentwicklungen," *Acta Inform.*, vol. 1, pp. 139–144, 1971.

[130] Schrijver, A., *Theory of Linear and Integer Programming*. Chichester, UK: Wiley, 1986.

[131] Shoup, V., "NTL: A library for doing number theory." `http://www.shoup.net/ntl/`, 2008.

[132] Storjohann, A., "The shifted number system for fast linear algebra on integer matrices," *Journal of Complexity*, vol. 21, pp. 609–650, 2005.

[133] Suhl, U. H. and Suhl, L. M., "Computing sparse LU factorizations for large-scale linear programming bases," *ORSA Journal on Computing*, vol. 2, no. 4, pp. 325–335, 1990.

[134] Ursic, S. and Patarra, C., "Exact solution of systems of linear equations with iterative methods," *SIAM Journal on Matrix Analysis and Applications*, vol. 4, no. 1, pp. 111–115, 1983.

[135] Vanderbei, R. J., *Linear Programming: Foundations and Extensions*. Boston, Massachusetts, USA: Kluwer, 2001.

[136] von zur Gathen, J. and Gerhard, J., *Modern Computer Algebra*. Cambridge, UK: Cambridge University Press, 2003.

[137] Wan, Z., "An algorithm to solve integer linear systems exactly using numerical methods," *Journal of Symbolic Computation*, vol. 41, pp. 621–632, 2006.

[138] Wang, P. S., "A p-adic algorithm for univariate partial fractions," in *SYMSAC '81: Proceedings of the Fourth ACM Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 212–217, ACM, 1981.

[139] Whaley, R. C. and Petitet, A., "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005.

[140] Wiedemann, D. H., "Solving sparse linear equations over finite fields," *IEEE Trans. on Inf. Theory*, vol. 32, pp. 54–62, 1986.

[141] Wilken, K., Liu, J., and Heffernan, M., "Optimal instruction scheduling using integer programming," *SIGPLAN Notices*, vol. 35, no. 5, pp. 121–133, 2000.

[142] Wolsey, L., *Integer Programming*. New York: Wiley-Interscience, 1998.

[143] Wright, S., *Primal-Dual Interior-Point Methods*. Philidelphia: SIAM, 1987.

[144] Zambelli, G., "On degenerate multi-row gomory cuts," *Operations Research Letters*, vol. 37, no. 1, pp. 21 – 22, 2009.

[145] Zanette, A., Fischetti, M., and Balas, E., "Can pure cutting plane algorithms work?," in *IPCO'08: Proceedings of the 13th International Conference on Integer Programming and Combinatorial Optimization*, (Berlin, Heidelberg), pp. 416–434, Springer-Verlag, 2008.

[146] Zanette, A., Fischetti, M., and Balas, E., "Lexicography and degeneracy: can a pure cutting plane algorithm work?," *Mathematical Programming*, pp. 1–24, 2009.

VITA

Dan Steffy was born on the 21st of July 1981 in Royal Oak Michigan, a suburb of Detroit. His father William was a chemist and his mother Ann is a social worker. He has one younger brother, Pete. After attending Kimball High School, Dan attended Oakland University in Rochester, MI, earning a Bachelor's degree in Mathematics in 2004 and a Master's degree in Mathematics in 2005 under the supervision of Professor Eddie Cheng. In August of 2005 he joined the School of Industrial and Systems Engineering at Georgia Tech to pursue a Ph.D. in Algorithms, Combinatorics and Optimization under the supervision of Professor William Cook. Dan's hobbies include sports, music, cooking and strategy games.