LINEAR PROGRAMMING ALGORITHMS USING LEAST-SQUARES METHOD

A Thesis
Presented to
The Academic Faculty

by

Seunghyun Kong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
H. Milton Stewart School of Industrial and Systems Engineering

Georgia Institute of Technology
May 2007

LINEAR PROGRAMMING ALGORITHMS USING LEAST-SQUARES METHOD

Approved by:

Ellis L. Johnson , Committee Chair
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Earl Barnes
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Joel Sokol
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Martin Savelsbergh,
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Prasad Tetali
School of Mathematics
Georgia Institute of Technology

Date Approved: March 13, 2007

To my beloved wife, Yuijn

ACKNOWLEDGEMENTS

First, I want to thank my wife Yujin for her enduring patience and loving support during my study. She has been always there for me from the very beginning to this very end. She has been at the core of my motivation even when I was doubting if could finish this program. I also have to mention my beloved children, Lynn and Olivia, who have been an encouraging reminder that I have a job to complete.

I appreciate my parents for the prayer they have done for me morning and night. Not only that, my father managed to continue his financial support even after he had to suffer all the financial loss from his business. It was more than a sacrificial love that the father could give to his son.

Special thanks goes to my advisor, Dr. Ellis Johnson. No wonder he amazed me with his creative advice with the depth of his knowledge whenever I had to ask his help in my research. What inspired me most was his personality. His patience has sustained my research no matter how slow my research would seem to progress. He is the best teacher that I could have in my life.

I would like to thank Dr. Earl Barnes and Dr. Sokol for reviewing the papers line by line for months, and giving me ideas and helpful advices. I would like to thank Dr. Tetali, who taught me two important math courses, which helped me very much. Dr. Savelsbergh's helpful suggestions made good improvement in the research.

I want to express my appreciation to my office mate Kapil, who taught me many programming skills, and spent so much time for discussing my research. I would like to thank Aang, Ethan and Wut, who started Ph.D program with me, and shared and helped in many ways. I also want to express thanks to other ISYE friends for their help.

I am indebted to many of the Korean friends in ISYE. Myunsuk and Hyunsuk helped me in many ways in my research. And Jisoo, Misook, Joongsup, Sungil, Sungwon, and Chuljong helped me when my family had a hard time

Finally and most importantly, I would like to give thanks to God, who was there with me in my difficult time, is with me, and will be with me.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

SUMMARY x

I INTRODUCTION 1

1.1 Survey of the Literature 2

II NON-NEGATIVE LEAST-SQUARES ALGORITHM 4

2.1 Algorithm 5

2.2 Solving Least-Squares Problems 9

2.2.1 Updating a QR-factorization 12

2.3 Re-factorization 17

2.4 Data Structure 23

III LEAST-SQUARES PRIMAL-DUAL ALGORITHM 27

3.1 Algorithm 27

3.2 Improving Convergence Rates 29

3.3 Computational Results 30

3.4 Concluding Remarks 36

IV A LEAST-SQUARES NETWORK FLOW ALGORITHM 37

4.1 Algorithm 37

4.1.1 Least-Squares Properties 38

4.2 Variations 48

4.2.1 Least-Squares Network Flow Algorithm with Upper Bounds . . . 48

4.2.2 Tree-Wise Pricing Strategy 49

4.3 Computational Results 50

4.4 Concluding Remarks 57

V COMBINED OBJECTIVE LEAST-SQUARES ALGORITHM 58
5.1 Algorithm 59
5.2 Computational Results 65
5.3 Concluding Remarks 71
VI ALGORITHMS FOR A LARGE SCALE LINEAR PROGRAMMING 73
6.1 Primal-Dual Subproblem Method 73
6.2 Least-Square Subproblem Method 75
6.2.1 Computational Results 76
REFERENCES 79
VITA 82

LIST OF TABLES

1 Density 31

2 Instances: Set-Partition 32

3 CPU Time: LSPD - small 34

4 CPU Time: LSPD - medium and large 34

5 Execution Details: LSPD Variation 35

6 Instances: Network Flow 51

7 CPU Time: LSNF 53

8 Execution Details: LSNF 55

9 CPU Time: COLS - quick 66

10 CPU Time: COLS - small, and medium 67

11 Execution Details: COLS($M = 1.0 \times 10^6$) 69

12 Profiling Results: COLS - sppnw14 70

13 CPU Time: COLS($M = 1.0 \times 10^6$) - large 71

14 Instances: RJmod and RJmax 77

15 Execution Details: Larger-Scale 77

16 Execution Comparison: LSSUB and CPLEX Sifting Algorithm 77

LIST OF FIGURES

1 Representation of $||Q^T b - Rx_b||$: $\bar{R}$, $\bar{b}$, and $\underline{b}$ 11

2 Geometric Interpretation of Givens Rotation 13

3 Addition of a Column 14

4 Deletion of a Column 15

5 Fill-in : Pivoting on the Element in the $(1, 1)$ Position 21

6 Data Structure: Q^T 24

7 Example : Data Structure Matrix 25

8 Data Structure: R 25

9 A tree T with Supplies and Demands on its Nodes 40

10 Matrix Notation for Tree T 42

11 Forward Arcs: (a, b) and (e, f) and Reverse Arcs: (c, d) and (g, h) 44

12 Subtrees $\mathbf{T}_d$ and $\mathbf{T}_g$ Separated From the Main Tree 47

13 Change of the Density at Each Re-Factorization 69

SUMMARY

The simplex method is the most widely used algorithm for solving linear programming problems. It works very efficiently in practice and is incorporated in most commercial software. However, degeneracy causes slow convergence in certain classes of problems. The airline crew-scheduling set-partitioning formulation is one the problems where degeneracy causes substantial difficulties.

This thesis is a computational study of recently developed algorithms which aim to overcome the above pitfall in the simplex method. To benchmark solution techniques, we use the airline crew-scheduling set-partitioning formulation for comparing the performance of the algorithms treated here with CPLEX. We study the following algorithms: the non-negative least squares algorithm, the least-squares primal-dual algorithm, the least-squares network flow algorithm, and the combined-objective least-squares algorithm.

All of the above four algorithms use least-squares measures to solve their subproblems, so they do not exhibit degeneracy. But, their properties are not well established yet in literature. Moreover, they have never been efficiently implemented and thus their performance have also not been proved. In this research we implement these algorithms in an efficient manner and improve their performance compared to on their preliminary results.

The non-negative least-squares algorithm is a least-squares algorithm with additional non-negativity constraints. The algorithm works by solving a series of unconstrained least-squares problems and using convex combination process to maintain non-negativity. We show strict improvement during minor iterations and develop basis update techniques and data structures that fit our purpose. In addition, we also develop a measure to help find a good ordering of columns and rows so that we have a sparse and concise representation of QR-factors.

The least-squares primal-dual algorithm uses the non-negative least-squares problem as

its subproblem, which minimizes infeasibility while satisfying dual feasibility and complementary slackness. We develop a special technique of relaxing and restoring complementary slackness to improve the convergence rate.

The least-squares network flow algorithm is the least-squares primal-dual algorithm applied to min-cost network flow instances. This is similar to the application of the simplex algorithm to solve min-cost network flow instances efficiently (Network simplex algorithm). The least-squares network flow algorithm can efficiently solve much bigger instances than the least-squares primal-dual algorithm. We implement an upper bounded version of the algorithm. After preliminary tests, we devise a specialized pricing scheme whose performance is similar to the CPLEX network solver.

The combined-objective least-squares algorithm is the primal version of the least-squares primal-dual algorithm. Each subproblem tries to minimize true objective and infeasibility simultaneously so that optimality and primal feasibility can be obtained together. It uses a big-M to minimize the infeasibility. We use dynamic M values to improve the convergence rate. We test the least-squares subproblem method, which uses the least-squares primal-dual algorithm and the combined-objective least-squares algorithm to solve large-scale crew pairing problems.

Our computational results show that the least-squares primal-dual algorithm and the combined-objective least-squares algorithm perform better than the CPLEX Primal solver, but are slower than the CPLEX Dual solver. The least-squares network flow algorithm performs as fast as the CPLEX Network solver.

CHAPTER I

INTRODUCTION

The simplex algorithm for linear programming is one of the most successful algorithms developed in the 20th century. The main drawback of the algorithm is that the solution often does not improve for many iterations, a problem which is called degeneracy. During degenerate iterations, one of the basic columns is replaced by one of the non-basic columns (like general simplex non-degenerate iterations), but the solution remains the same. The degenerate iteration is not in vain since the basis changes toward a non-degenerate basis even in degenerate iterations. But sometimes the same basis is repeated after some number of iterations, which is called cycling. Even though there are some remedies for preventing degeneracy such as lexicographic rules, and perturbation techniques, they create more vertices or iterations and increase the computational burden.

Airline crew-pairing set-partition integer programming formulations are large-scale optimization problems, and are our primary interest since they exhibit large amounts of degeneracy when the linear programming relaxations are solved by the simplex algorithm. The algorithms described in the thesis do not have degeneracy. Hence we use tested our algorithms with airline crew-pairing set-partition integer programming formulations.

The primal-dual simplex method is one of the simplex-like algorithms that were developed by Dantzig [24] as a generalization of Kuhn's algorithm [23], which solves a sequence of small-size phase-I linear programming problems to improve the dual solution. Its intermediate dual solutions move in the interior of the dual polyhedron. If we look at this algorithm from the dual point of view, then it is interpreted as a dual active set algorithm since this algorithm takes care of the dual constraints that are tight.

As a variation of the primal-dual simplex method, the least-squares primal-dual algorithm that uses non-negative least-squares sub-problem will be presented.

The non-negative least-squares algorithm is a special case of the least-squares algorithm

and can be solved with a sequence of least-squares algorithms. And the sequence of least squares algorithms can be solved using QR-factorization to make efficient uodates. When a column is added to a matrix or removed from a matrix, there is a very easy way to update a QR-factorization. This useful technique can be used with column generation.

Like LU factorization, there are some ways of making QR factorization sparse. This is really important since QR factorization is used for solving sets of linear systems and sparse systems can be easily solved.

This research is an extension of Gopalakrishnan [19]. This includes three new algorithms: the least-squares primal-dual, which is a variation is the primal-dual simplex algorithm; the combined objective least-squares algorithm; and the least-squares network flow algorithm.

This thesis is focused on practical computation. We develop some sparse matrix schemes for the upper triangular matrix R, a compact way of storing the product form of inverse of the Q^T matrix in QR factorization, and a way of getting a sparser upper triangular matrix in QR-factorization. In the combined-objective least-squares algorithm, the convergence of the algorithm is really dependent on the value of big-M. We develop a way to change big-M to get better convergence. The least-squares network flow algorithm has good properties to lessen its computational efforts.

For large scale linear programming, we tested the least-squares primal-dual and the combined-objective least-squares algorithms in conjunction with the least-squares sub-problem method.

1.1 Survey of the Literature

Gopalakrishnan [19] is a primary source for this research since it includes the least-squares primal-dual, combined-objective least-squares, and least-squares network flow algorithms. The least-squares primal-dual algorithm and combined-objective least-squares algorithm are based on the non-negative least-squares algorithm, which was developed by Lawson and Hanson [25] and includes techniques that can be used with least-squares algorithms. The way of updating the R matrix of QR-factorization after adding a column and deleting a

column is also treated in the book. Björck [5] has comprehensive numerical approaches for least-squares problems.

Since the simplex algorithm was developed, it has evolved into a state of the art algorithm in many senses. So the least-squares primal-dual and the combined-least-squares algorithms need to imitate its good features, such as numerical stability, sparsity, a simple product form of the inverse, and easy updates of the basis. Chvátal [6] treats the basic ideas of techniques such as the product form of the inverse, eta-file, and network simplex and its data structures. Forrest and Tomlin [12] and Suhl and Suhl [33] explain advanced techniques for the simplex algorithm. We also used [29] for better understanding of the computational aspects of the simplex algorithm.

For sparse least-squares problems, Matstoms [30] and George [14] have good introduction to sparsity-related techniques. In the early literature, Duff [9] experimented with row ordering and pivot element selection in sparse QR-factorization. In the simplex algorithm, Markowitz [28] developed a simple and practical rule that is still used for selecting pivot element. Golub [17] proposed a way to get a numerically stable QR-factorization. George [27], Thomas [7], George and Liu [15] and many others explain ways of getting sparser QR-factorizations using graph algorithms. A multifrontal algorithm [10] is the most recent advance in finding sparse factorizations. Introductory error analysis is found in [13].

CHAPTER II

NON-NEGATIVE LEAST-SQUARES ALGORITHM

A non-negative least-squares problem (NNLS) is a least-squares problem with non-negativity constraints. The problem can be written as

$$\begin{aligned} \text{Minimize} \quad & \rho^T \rho \\ \text{s.t.} \quad & Ex_E + \rho = b \\ & x_E \geq 0 \end{aligned} \qquad (\textbf{NNLS})$$

or, equivalently

$$\begin{aligned} \text{Minimize} \quad & ||b - Ex_E||^2 \\ \text{s.t.} \quad & x_E \geq 0, \end{aligned}$$

where E is m by n matrix, x_E and ρ (residual) are n and m dimensional column decision variables, and b is m dimensional column vector. The sbscript E is used for easier understanding in the next chapter (the LSPD algorithm).

The NNLS algorithm has a strong resemblance to the simplex algorithm (Phase I) in many aspects.

$$\begin{aligned} \text{Minimize} \quad & \sum |\rho_i| \\ \text{s.t.} \quad & Ex_E + \rho = b \\ & x_E \geq 0 \end{aligned} \qquad (\textbf{Phase I})$$

(**Phase I**) and (**NNLS**) are minimizing the L_2 and L_1 norm of the residual ρ, respectively.

The NNLS algorithm was introduced by Lawson and Hanson [25] and was used to solve the Phase I problem in linear programming in [26]. We use the NNLS algorithm to solve the primal feasibility step of a primal-dual algorithm.

2.1 Algorithm

Like the simplex algorithm, the NNLS algorithm has a basis B consisting of a set of linearly independent columns of E used for solving a system of equations. x^*_E, the solution of (**NNLS**), is $[x^*_B, x_{E\setminus B} = 0]$, where x^*_B is the solution of

$$\min \quad \|b - Bx_B\|^2, \; x_B \geq 0.$$

This is similar to the situation in the simplex algorithm, but the details differ considerably. In particular, a basis is not invertible and $x^*_B > 0$ in (**NNLS**). A feasible basis B for (**NNLS**) is a set of linearly independent columns of E that have all positive components in the solution of

$$\min \quad \|b - Bx_B\|^2.$$

The solution of the unconstrained minimization problem $\min ||b - Bx_B||^2$ is given by

$$x^*_B = (B^T B)^{-1} B^T b. \tag{1}$$

When E is a node-arc incidence matrix from network-flow problem there is a very simple way to get the solution x^*_B without solving the system of equations; it is described in Chapter 4.

Each major iteration of NNLS solves

$$\begin{aligned} \min \quad & ||b - Bx_B - A_s x_s||^2 && (2) \\ \text{s.t.} \quad & x_B\,,\, x_s \geq 0, \end{aligned}$$

with feasible basis B. It requires at least one minor iteration. The minor iteration is a repeated process (inside a major iteration) of solving unconstrained minimization problems

$$\min ||b - Bx_B - A_s x_s||^2, \tag{3}$$

or

$$\min ||b - \bar{B}x_{\bar{B}} - A_s x_s||^2, \tag{4}$$

where $\bar{B}$ is the subset of B that remains after dropping some columns out of B. Let $x^*_{[B,A_s]}$ and $x^*_{[\bar{B},A_s]}$ be solutions of (3) and (4), respectively. The residuals $\rho^*_B = b - Bx^*_{[B,A_s]}$ and

$\rho^*_{\bar{B}} = b - \bar{B}x^*_{[\bar{B},A_s]}$ are the residuals of the solutions $x^*_{[B,A_s]}$ and $x^*_{[\bar{B},A_s]}$, respectively. These ρ^*_B and $\rho^*_{\bar{B}}$ are computed after the least-squares problem is solved as a byproduct.

The NNLS algorithm starts with empty basis $B = [\,]$. B is a feasible basis since x^*_B has no non-positive component, $\rho^*_B = b$, and $x_{E\setminus B} = 0$.

In each major iteration, we choose a column A_s with ${\rho^*_B}^T A_s > 0$, since it violates the KKT conditions of **NNLS**

$$E^T(b - Ex^*_E) \leq 0, \tag{5a}$$

$$x^*_E E^T(b - Ex^*_E) = 0, \tag{5b}$$

$$x_E \geq 0. \tag{5c}$$

If there is no column satisfying ${\rho^*_B}^T A_s > 0$, then $x^* = [x^*_B, x_{E\setminus B} = 0]$ is the optimal solution of (**NNLS**).

Suppose there is a column A_s with ${\rho^*_B}^T A_s > 0$. Then we need to solve $\min ||b - Bx_B - A_s x_s||^2$. Let $P = I - B(B^T B)^{-1} B^T$ be the projection matrix onto the orthogonal space of the column space of B. $\rho^*_B = b - Bx^*_B = Pb$, which implies that ρ^*_B is in the orthogonal space of the column space of B. Then we can clearly see that ${\rho^*_B}^T P = {\rho^*_B}^T$. With this property we can see that the new column A_s can reduce the norm of the residual of the system and the solution value x^*_s of the new column A_s is positive, where x^*_s is the last component of $x^*_{[B,A_s]}$ and corresponds to the column A_s.

Proposition 2.1.1 *The strict inequality min* $||b - Bx_B - A_s x_s||^2 <$ *min* $||b - Bx_B||^2$ *holds.*

Proof:

$$
\begin{aligned}
&\min\, ||b - Bx_B - A_s x_s||^2 \\
=&\min\, ||{\rho_B}^* - PA_s x_s||^2 \\
\leq&{\rho_B}^{*T}{\rho_B}^* - 2{\rho_B}^{*T} PA_s x_s + ||PA_s||^2 x_s^2 \\
=&{\rho_B}^{*T}{\rho_B}^* - 2{\rho_B}^{*T} A_s x_s + ||PA_s||^2 x_s^2 \\
<&{\rho_B}^{*T}{\rho_B}^* = \min\, ||b - Bx_B||^2 \text{ with sufficiently small positive } x_s
\end{aligned}
$$

■

Corollary 2.1.2 *The problem* $min||b - Bx_B - A_s x_s||^2$ *has solution* $x_s > 0$.

Proof: From Proposition 2.1.1, $||b - Bx_B - A_s x_s||^2$ is minimized at

$$x_s^* = \theta^* = \frac{{\rho_B^*}^T A_s}{||PA_s||^2} > 0, \text{and } x_B = (B^T B)^{-1} B^T (b - A_s \theta^*). \tag{6}$$

■

If $x^*_{[B,A_s]}$ is strictly positive, then it is also an optimal solution of $\min ||b - Bx_B - A_s x_s||^2, x \geq 0$. After updating $B = [B, A_s]$, this completes one major iteration of the NNLS algorithm with improved norm of the residual (by Proposition 2.1.1). If $x^*_{[B,A_s]}$ is non-negative but includes at least one zero component, then zero components are removed from $x^*_{[B,A_s]}$ and the corresponding columns in B are removed. Then we need to update $B = [B, A_s]$ and this completes one major iteration of the NNLS algorithm with smaller norm of the residual (by Proposition 2.1.1).

Suppose $x^*_{[B,A_s]}$ includes negative components. Then we take the convex combination of $[x^*_B, 0]$ and $x^*_{[B,A_s]}$,

$$x_{conv}(\lambda) = (1 - \lambda)[x^*_B, 0] + \lambda x^*_{[B,A_s]} \geq 0, \quad \text{with } 0 \leq \lambda \leq 1. \tag{7}$$

We want λ as big as possible while satisfying $x_{conv}(\lambda) \geq 0$, and denote this value $\bar{\lambda}$. $\bar{\lambda} > 0$ since x^*_B and x^*_s are positive and $\bar{\lambda} < 1$ since $x^*_{[B,A_s]}$ has at least one negative component. Let $\bar{\theta}$ denote the last component of $x_{conv}(\bar{\lambda})$. Observe that $\bar{\theta} = \theta^* \bar{\lambda} < \theta^*$ since $\bar{\lambda} < 1$. By the choice of $\bar{\lambda}$ there is at least one zero component in $x_{conv}(\bar{\lambda})$. We define $\bar{B}$ from B by dropping columns corresponding to the zero components in $x_{conv}(\bar{\lambda})$ and x^*_B is reconstructed using the positive components of $x_{conv}(\bar{\lambda})$. Observe that $||\rho^*_B||^2 > ||b - Bx_{conv}(\bar{\lambda})||^2$ because of the convexity of $||\cdot||^2$. But we cannot guarantee that $\bar{B}$ is a a feasible basis.

The second minor iteration starts with solving $\min ||b - \bar{B}x_{\bar{B}} - A_s x_s||^2$. If the solution $x^*_{[\bar{B},A_s]}$ is strictly positive then we update B as $[\bar{B}, A_s]$ and finish the major iteration. If $x^*_{[\bar{B},A_s]}$ is non-negative with a zero component, then we drop zero components from $x^*_{[\bar{B},A_s]}$ and the corresponding columns from $\bar{B}$. We update B with $[\bar{B}, A_s]$ and the next iteration starts.

Suppose $x^*_{[\bar{B},A_s]}$ has a negative component. Then we take the convex combination

$$x_{conv}(\lambda) = (1-\lambda)x^*_B + \lambda x^*_{[\bar{B},A_s]} \geq 0, \quad \text{with } 0 \leq \lambda \leq 1. \tag{8}$$

Again we want λ as big as possible satisfying $x_{conv}(\lambda) \geq 0$, and denote this value $\bar{\lambda}$. $\bar{\lambda} > 0$ since x^*_B is strictly positive and $\bar{\lambda} < 1$ since $x^*_{[\bar{B},A_s]}$ has at least one negative component. By the choice of $\bar{\lambda}$ there is at least one zero component in $x_{conv}(\bar{\lambda})$. We redefine $\bar{B}$ by dropping columns corresponding to the zero components in $x_{conv}(\bar{\lambda})$ and x^*_B is constructed by the positive components of $x_{conv}(\bar{\lambda})$. Solving $\min ||b - \bar{B}x_{\bar{B}} - A_s x_s||^2$, taking the convex combination (8), and dropping columns are repeated until $x^*_{[\bar{B},A_s]}$ is strictly positive. We then update B with $[\bar{B}, A_s]$ and then the next major iteration starts. Let $f_B(\theta) = \min ||b - Bx_B - A_s\theta||^2$ and $f_{\bar{B}}(\theta) = \min ||b - \bar{B}x_{\bar{B}} - A_s\theta||^2$. Note that B and $\bar{B}$ are strictly convex since they have linearly independent columns.

We are going to show that the last component of $x^*_{[\bar{B},A_s]}$ is bigger than the last component of $x^*_{[B,A_s]}$ so that A_s stays positive and never drops out of basis during convex combination procedure (7) and $||\rho^*_B||^2 > ||\rho^*_{\bar{B}}||^2$. This explanation requires some notation to be defined. $x_B(\theta)$ is the components corresponding to the columns of B in the solution of $\min||b - Bx_B - A_s\theta||^2$ when θ is fixed. Observe that $x_B(\theta) = ((B^TB)^{-1}B^T)(b - A_s\theta)$. Let $\rho_B(\theta) = b - Bx_B(\theta) - A_s\theta = \rho^*_B - PA_s\theta$. We can define $x_{\bar{B}}(\theta)$ and $\rho_{\bar{B}}(\theta)$ in the same way.

Theorem 2.1.3 *$\min ||b - \bar{B}x_{\bar{B}} - A_s x_s||^2$ has optimal solution $x^*_{[\bar{B},A_s]}$ whose last component is greater than $\bar{\theta}$.*

Proof: Since $[\bar{B}, A_s] \subset [B, A_s]$, we have $f_{\bar{B}}(\theta) \geq f_B(\theta)$ and $f_{\bar{B}}(0) > f_B(0)$. At $\theta = \bar{\theta}$ we have $f_{\bar{B}}(\bar{\theta}) = f_B(\bar{\theta})$. Since $x_{\bar{B}}(\bar{\theta})$ is strictly positive by construction, $x_{\bar{B}}(\bar{\theta} + \delta_1)$ is also strictly positive with sufficiently small positive δ_1.

Suppose $f_{\bar{B}}(\bar{\theta})' \geq 0$. Then there is δ_2 satisfying $f_{\bar{B}}(\bar{\theta} - \delta_2) < f_B(\bar{\theta} - \delta_2)$ with sufficiently small positive δ_2, since $f_B(\bar{\theta})' < 0$ and $f_{\bar{B}}(\bar{\theta})' \geq 0$. But it contradicts $f_{\bar{B}}(\theta) \geq f_B(\theta)$. Therefore we can increase θ after dropping some columns while satisfying non-negativity constraints. By the convexity of $|| \cdot ||^2$, the last component of $x^*_{[\bar{B},A_s]} > 0$, which is larger than $\bar{\theta}$. ∎

Observe that $||\rho^*_B||^2 > ||b - \bar{B}x_{conv}(\bar{\lambda})||^2 > ||\rho^*_{\bar{B}}||^2$. The same results hold for the later minor iterations, so that the last component of $x^*_{[\bar{B},A_s]}$ monotonically increases and $||b - \bar{B}x_{conv}(\bar{\lambda})||^2$ monotonically decreases at each minor iteration, until $x_{[\bar{B},A_s]^*} > 0$.

Since there is no zero-component solution of the least-squares problem, the solution from each major iteration improves strictly so that there is no degeneracy. Columns corresponding to zero components in the solution are removed even though they are linearly independent of the other columns in B. Computational results show that the basic matrix is rectangular. LU-factorization for the simplex algorithm does not work for a rectangular matrix. So, another technique is used to treat the rectangular matrix. This is explained in the next sub-chapter.

The algorithm described above is similar to a number of existing algorithms, including one to solve the bounded least-squares problem by Bjorck [5], and another to solve the NNLS problem by Lawson and Hanson [25], and Leichner, Dantzig and Davis [26].

The authors of [26] show that such a NNLS algorithm performs better than the simplex algorithm (in solving the linear programming Phase I problem) on a wide range of linear programming problems.

As a summary, the NNLS algorithm is described in **Algorithm 1**;

2.2 Solving Least-Squares Problems

In the previous chapter, we showed that the solution of $\min ||b - Bx_B||^2$ is given by $x^*_B = (B^TB)^{-1}B^Tb$. Recall that the columns of B are linearly independent and B is not a square matrix, so $(B^TB)^{-1}$ exists. However, computing x^*_B by this formula is very expensive since it involves inverting the matrix B^TB and computing several matrix products. In this chapter we explain how least-squares problems are solved in practice. We focus on the unconstrained minimization problem, because in the previous chapter we explained how the constrained minimization problem $\min ||b - Bx_B||^2, x_B \geq 0$ can be solved as a sequence of unconstrained minimization problems.

In solving linear programming problems by the simplex algorithm one needs to solve linear system of the form $Bx_B = b$. This system can be solved efficiently if the matrix B is

Algorithm 1 NNLS : Minimize $||b - Ex_E||^2$, $x_E \geq 0$

1: $\rho^*_B = b$, $B = \emptyset$, $x^*_B = \emptyset$, and $x_{E\setminus B} = 0$.
2: **while** $I(:= \{\rho^{*T} A_j \ : \rho^{*T} A_j > 0\}) \neq \emptyset$ **do**
3: $s \in I$
4: **if** $x^*_{[B,A_s]}(:= argmin||b - Bx_B - A_s x_s||) \ngeq 0$ **then**
5: Let $\bar{\lambda}$ be the largest λ such that $x_{conv}(\lambda) = (1-\lambda)[x^*_B, 0] + \lambda x^*_{[B,A_s]} \geq 0$
6: $x^*_B = x_{conv}(\bar{\lambda})$ and eliminate zero components in x^*_B.
7: Construct $\bar{B}$ with the corresponding columns of x^*_B.
8: **while** $x^*_{[\bar{B},A_s]}(:= argmin||b - \bar{B}x_{\bar{B}} - A_s x_s||) \ngeq 0$ **do**
9: Let $\bar{\lambda}$ be the largest λ such that $x_{conv}(\lambda) = (1-\lambda)x^*_B + \lambda x^*_{[B,A_s]} \geq 0$
10: $x^*_B = x_{conv}(\bar{\lambda})$ and eliminate zero components in x^*_B.
11: Construct $\bar{B}$ with the columns corresponding to x^*_B.
12: **end while**
13: $\rho^*_B = b - Bx^*_B - A_s x^*_s$, $x \leftarrow [x^*_B, x_{E\setminus B} = 0]$
14: $B = \bar{B} \setminus A_s$
15: **end if**
16: $B = [B, A_s]$
17: **if** $\rho^*_B = 0$ **then**
18: STOP : x is optimal solution for NNLS with min $||b - Ex_E||^2$=0
19: **end if**
20: **end while**
21: x is optimal solution for NNLS with min $||b - Ex_E||^2 > 0$

sparse. In this case, B^{-1} is represented as the product of matrices for which solving linear equations is in a simple matter. For example, a sparse factorization of B^{-1} is often used. In our case we solve problems of the form $\min ||b - Bx_B||^2$, where B is sparse but not a square matrix.

There are several approaches to solving this least-squares problem. x^*_B can be computed by solving the normal equation $B^T(Bx_B - b) = 0$ using a Cholesky factorization of B. In each major iteration, B is updated by adding a column, and sometimes dropping several columns. The QR-factorization can be updated efficiently for such changes in B. There are several ways to compute a QR-factorization of B. The Gram-Schmidt procedure, singular value decomposition, and elementary orthogonalization (Givens or Householder transformation) can be used to orthogonalize the columns of B. See [5] for more detail. In our case, we use Givens transformations because they are convenient for managing sparsity.

Let B be a rectangular basis matrix for the NNLS problem. Let Q be an orthogonal matrix such that $B = QR$ where R is a $m \times k$ matrix of the type shown in Figure 1. The

non-zero portion of R appears in the nonsingular upper triangular matrix $\bar{R}$ in Figure 1. Consider the problem $\min ||b - Bx_B||^2$. Observe that

$$||b - Bx_B||^2 = ||Q^T(b - Bx_B)||^2 = ||Q^T b - Rx_B||^2.$$

Let $\bar{b}$ and $\underline{b}$ be the upper k and lower $m-k$ subvectors of $Q^T b$. We then have

$$||b - Bx_B||^2 = ||Q^T b - Rx_B||^2 = ||\bar{b} - \bar{R}x_B||^2 + ||\underline{\mathrm{b}}||^2 \geq ||\underline{\mathrm{b}}||^2,$$

so we can solve $\min ||b - Bx_B||^2$ by solving $\bar{R}x_B = \bar{b}$ for x_B^*. This is an easy calculation since $\bar{R}$ is a triangular and nonsingular matrix.

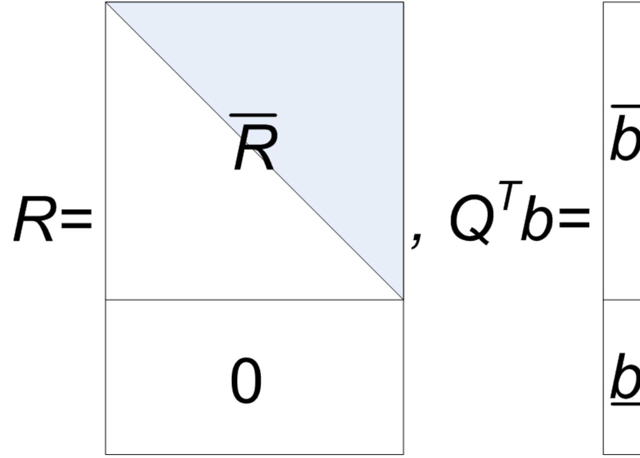

Figure 1: Representation of $||Q^T b - Rx_b||$: $\bar{R}$, $\bar{b}$, and $\underline{b}$

Given x_B^*, we compute the residual $\rho^* = (b - Bx_B^*)^T$ and determine an entering column A_s as explained in the previous chapter. We must then solve $\min ||b - Bx_B - A_s x_s||^2$. If the solution $x^*_{[B,A_s]}$ of this problem is strictly positive we take $[B, A_s]$ as our update of B. Suppose that the solution $x^*_{[B,A_s]}$ has some negative or zero components. We must then use the convex combination procedure to drop some columns from B to form $\bar{B}$. We then solve $\min ||b - \bar{B}x_{\bar{B}} - A_s x_s||^2$ and repeat the convex combination procedure and dropping of columns of $\bar{B}$ until we have a feasible basis $[\bar{B}, A_s]$. Thus before updating the basis B we must solve at least one least-squares minimization problem obtained by adding a column to B, and possibly deleting certain columns. Thus we must compute a QR-factorization for a matrix $[B, A_s]$ or $[\bar{B}, A_s]$ obtained by adding and possibly dropping columns to a matrix

whose QR-factorization was for the previous B. We explain how this is done in the next sub-chapter.

2.2.1 Updating a QR-factorization

For an elementary orthogonal transformation, two methods can be used, the Householder transformation and the Givens transformation. The Householder transformation and the Givens transformation are mathematically equivalent. But it was reported by [9] that the Givens transformation has an advantage over the Householder transformation when sparsity is concerned. Hence only the Givens transformation is used for the orthogonalization in this research. Björck [5] has excellent survey of QR-factorization and least-squares algorithm.

Givens transformation is a sequence of Givens rotations which are two dimensional (flat) rotations to one of the axes. Suppose there is a column A_s which has non-zero elements a in the ith and b in the jth position. Assume that the ith position is the diagonal position in the matrix and the jth position is below diagonal position. The Givens rotation matrix G of A_s is identity matrix except four entries (i,i), (i,j), (j,i),

$$\mathbf{G} = \begin{bmatrix} \ddots & & & & \\ & cos & & sin & \\ & & \ddots & & \\ & -sin & & cos & \\ & & & & \ddots \end{bmatrix}$$

, where $cos = \frac{a}{\sqrt{a^2+b^2}}$ and $sin = \frac{b}{\sqrt{a^2+b^2}}$ and all the other diagonal elements are 1.

The multiplication $G \cdot A_s$ makes the ith position in A_s $\sqrt{a^2+b^2}$ and the jth position 0. The length of (a,b) is preserved and (a,b) are rotated to $(c = \sqrt{a^2+b^2}, 0)$, which is on the first axis, as shown in Figure 2. One application of this G to B changes the ith row and the jth row. And the other rows except the ith and the $j^{th}$are not changed by G. A Givens rotation with $a = 0$ can be considered as row permutation for the column and it is used to make a zero pivot position nonzero by multiplying the ith row by -1 and swapping the ith row and the jth row since the cos value is zero and the sin value is 1.

The orthogonalization can be done by removing non-zeros in rows or columns. Since

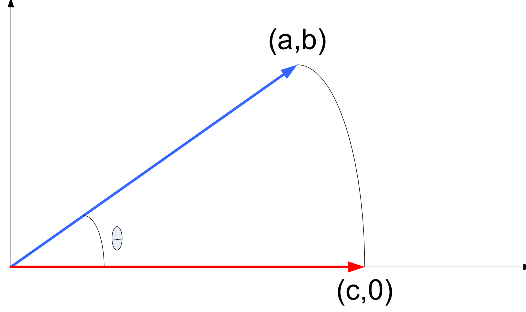

Figure 2: Geometric Interpretation of Givens Rotation

one column is appended to the basis in every iteration of the NNLS algorithm, column-wise orthogonalization is the more convenient way.

There are two situations that we need to update Q and R matrices, appending one column to the basis and removing a middle or left most column from the basis.

At the beginning of each major iteration of NNLS, we have a basis B and its QR-factorization Q and R. We need to solve min $||b - Bx_B - A_s x||^2$. min $||b - Bx_B - A_s x_s||^2$ is equivalent problem of min $||Q^T b - Rx_B - Q^T A_s x_s||^2 = ||Q^T b - [R, Q^T A_s][x_B^T, x_s]^T||^2$. When we combine R and $Q^T A_s$ as $[R, Q^T A_s]$ then it is similar to the first figure in Figure 3, which is upper triangular except the last column. Orthogonalization of this matrix $[R, Q^T A_s]$ starts with finding pivot element in the last column since the other columns are already orthogonalized. The pivot element is moved to a diagonal position by row permutation first, and is used for eliminating other elements below the diagonal position. Eventually any nonzero element can be a diagonal element but usually it is chosen for numerical stability and sparsity management. After the pivot element is chosen and moved to the diagonal position by row permutation (in fact, a Givens rotation of 90°) then a Givens rotation is applied to eliminate non-zero elements below the diagonal. Since one Givens rotation removes only one below diagonal element, many Givens rotations are necessary if there are many below-diagonal elements. The example in Figure 3 is used for this explanation. Suppose b is chosen as a pivot element. We need to move this element to the diagonal position by row permutation. After row permutation (G_0 - Givens rotation of 90°) the column becomes $[\times \times$

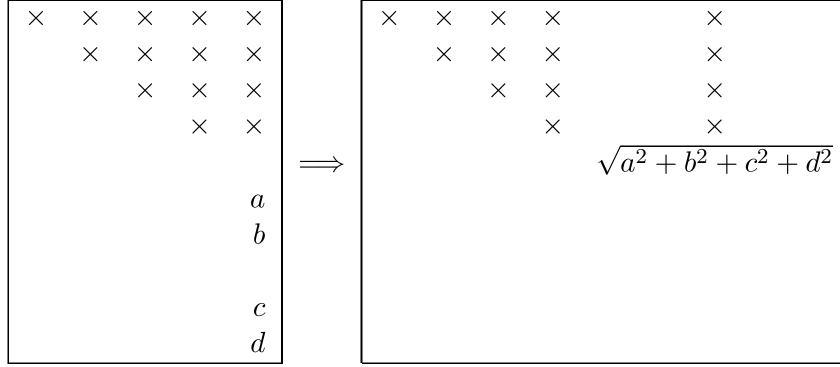

Figure 3: Addition of a Column

$\times \times b\ a\ \cdot\ \cdot\ c\ d]^T$. (An empty position is represented by $\cdot$.) The element in the sixth position becomes zero after one Givens rotation (G_1) and then the result is $[\times\times\times\times\sqrt{a^2+b^2}\ \cdot\cdot\cdot\ c\ d]^T$. Observe that all rows except the fifth and the sixth are left unchanged. One more Givens rotation (G_2) makes our column $[\times\times\times\times\sqrt{a^2+b^2+c^2}\ \cdot\ \cdot\ \cdot\ \cdot\ d]^T$. The next Givens rotation (G_3) makes it $[\times\times\times\times\sqrt{a^2+b^2+c^2+d^2}\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot]^T$. The overall Givens transformation for this column can be represented as $G_3 \cdot G_2 \cdot G_1 \cdot G_0$, and the new upper triangular matrices R and Q^T are updated as $G_3 \cdot G_2 \cdot G_1 \cdot G_0 \cdot Q^T$. If there are k elements below diagonal and non-zero diagonal elements in $Q^T A_s$, then one Givens transformation requires k Givens rotations, $G_1, \ldots, G_k$, and may require one row permutation G_0 depending on the original diagonal element of $Q^T A_s$. With this Givens transformation we can transform $[R, Q^T A_s]$ to upper triangular form $G_k \cdots G_1 G_0 [R, Q^T A_s] = [R, G_k \cdots G_1 G_0 Q^T A_s]$ and orthogonal matrix $Q G_0^T \cdots G_k^T$.

There is one important property of about QR-factorization after a new column is appended. Observe that the diagonal element after QR-factorization becomes the two-norm of the diagonal element and the elements below the diagonal. Therefore we don't need to worry about numerical stability when solving the system $\bar{R} x_B = \bar{b}$ since the diagonal entries of $\bar{R}$ after QR-factorization are relatively large.

Suppose there is a negative component in the solution of the problem $\min ||b - Bx_B - A_s x_s||^2$. After performing the convex combination procedure we drop some columns of B to obtain $\bar{B}$. We must find a QR-factorization of $[\bar{B}, A_s]$. Recall that $Q^T [B, A_s] = R$, where

R has the form of the matrix in Figure 1. For the moment assume that only one column of B is dropped to obtain $\bar{B}$. Then $Q^T[\bar{B}, A_s]$ is an upper Hessenburg matrix (a matrix that has zero entries below the first sub-diagonal). For example, if $Q^T[B, A_s]$ is the matrix in Figure 4 (a), and column 3 is dropped we obtain the upper Hessenburg matrix $Q^T[\bar{B}, A_s]$ in Figure 4 (b). We will bring this matrix to upper triangular form by a using Givens transformation. The first Givens rotation (G_1) transforms Figure 4 (b) to Figure 4 (c). Notice that G_1 changes row 3 and 4 in positions on or above the diagonal. Let G_2 be the Givens rotation of the vector (d', e) to $(\sqrt{d'^2 + e^2}, 0)$. Applying G_2 to Figure 4 (c) produces the upper triangular matrix in Figure 4 (d). If more than one column drops from B during the convex combination procedure we drop one at a time, updating the QR-factorization each time a column is dropped.

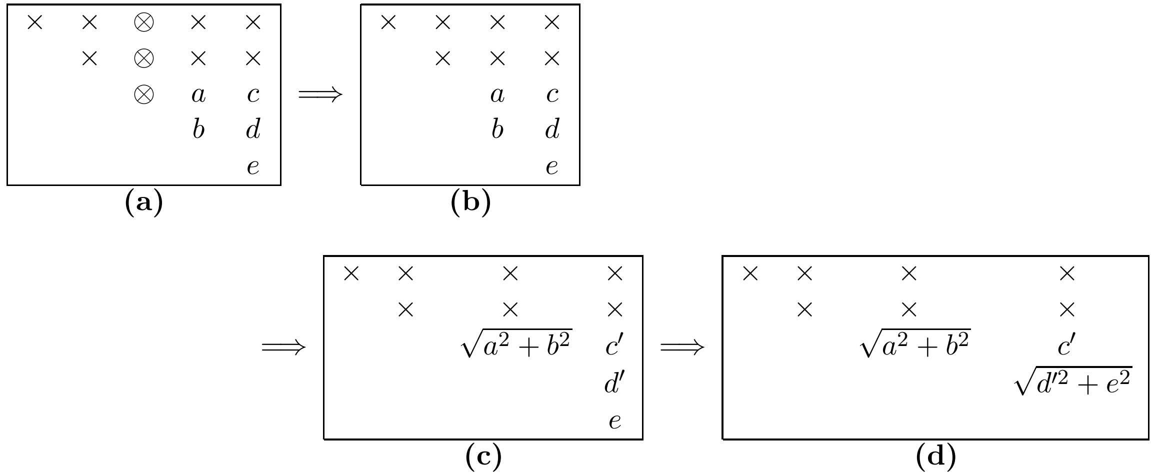

Figure 4: Deletion of a Column

Suppose after solving $\min ||b - \bar{B}x_{\bar{B}}||^2$ the solution $x^*_{[\bar{B}, A_s]}$ has negative or zero components. Then we can do similar procedure in the previous paragraph and repeat until we have feasible basis $[\bar{B}, A_s]$.

Some care has to be taken to preserve numerical stability while performing the Givens rotation. For example, if e in Figure 4 (b) is very small, we look ahead to see if d' in 4 (c) is also very small. If so, the matrix $\bar{R}$ has a small diagonal term $\sqrt{d'^2 + e^2}$ and this makes solving the equations $\bar{R}x_B = \bar{b}$ by back substitution unstable. In such a case we also

temporarily consider a Givens rotation of 90°, changing the pivot element in rows 3 and 4 in Figure 4 (b) before performing the Givens rotation of rows 3 and 4. If this results in a large diagonal element for $\bar{R}$, we perform this operation. If e is not very small there is no need to consider this operation.

We store Q^T as the product of Givens rotations. Thus we always have $Q^T = G_u \cdot G_{u-1} \cdots G_1$ for some integer u. The computation $Q^T A_s$ can be carried out equally fast or faster if Q^T is kept as an explicit matrix or as the product of Givens rotations since it is not efficient to keep explicit Q^T as concise form. Also, computing $Q^T A_s$ with an explicit form of Q^T is the same as computing $G_u(G_{u-1}(\cdots(G_1 A_s)\cdots))$, as far as numerical stability is concerned. However, it is very time consuming to updated an explicit form of Q^T after a new Givens rotation is performed. On the other hand, updating the product form is trivial since the data already stored does not have to be altered in the product form of Q^T. We just append a new Given rotation to Q^T.

If the number of Givens rotations, denoted u, in the product form of Q^T becomes too large, it takes too much time to compute $Q^T A_s = G_u(G_{u-1}(\cdots(G_1 A_s)\cdots))$, and computing this product generates numerical errors. Thus when the sequence of Givens rotations in the representation of Q^T becomes too large we re-factor Q^T as the product form of a small number of Givens rotations. During the new QR-factorization of the basis, we reorder the rows and the columns of B to achieve a sparse R and a short sequence of Givens rotations for Q^T. The way this is done is explained in the next sub-chapter.

There is not much we can do to control the sparsity of R during QR updates as columns are added to or dropped from B. However we have a lot of freedom to influence the sparsity of R during the re-factorization procedures.

The updating strategy we explain for the QR-factorization is similar to the Forrest-Tomlin update [11] for updating an LU factorization of a linear programming basis matrix. A Forrest-Tomlin update uses column and row interchange to produce a matrix which is upper triangular with a row spike at the last row. Row operations are needed to bring this matrix to upper triangular form. Our strategy is different in that column interchanges are never used. When a column is dropped from B we are left with a matrix in upper

Hessenburg form. If this matrix has a zero on its diagonal a Givens rotation of 90° (a row swap) is performed to remove zero diagonal element. Givens rotations of 90° are also needed to improve numerical stability. When a column is appended to B, we may also need a Givens rotation of 90° to remove a zero from the diagonal.

2.3 Re-factorization

Since Q is stored as a product $Q^T = G_u \cdots G_0 I$ of Givens rotations, computing $Q^T A_s$ by the formula $G_u(G_{u-1} \cdots (G_0 A_s) \cdots)$ is too time consuming if u is very large. Also, since each G_i involves a square root and a division calculation round-off error occurs each time a Givens rotation is applied. For small values of u this round off error is negligible but for large values of u it can be significant. Thus we always re-factor the basis using a small number of Givens rotations once u becomes too large.

Re-factorization of B can be triggered in four ways. First, we re-factor B if the operations to compute $Q^T A_s$ exceed a certain threshold. Second, we re-factor B if the round-off error in computing $Q^T A_s$ causes the columns of B to become linearly dependent. Third, we re-factor B if pre-assigned memory space for storing Q^T or R are used up. Fourth, we refactor if the density of the upper triangular matrix R becomes too high.

During the re-factorization process these things should be taken into consideration. First, the number of Givens rotation used must be small, so that $Q^T A_s$ can be computed quickly. Second, the matrix R must be sparse so that solving the system $\bar{R} x_B = \bar{b}$ does not require many arithmetic operations. Third, the re-factorization must be done quickly.

Similar considerations for re-factoring the basis in linear programming problems were studied many researchers [29]. In linear programming an LU factorization of the basis is usually computed when the basis is updated. By proper rearrangement of the rows and columns of the matrix the sparsity of the LU-factorization can be reduced. We have developed a scheme similar to one in linear programming, for rearrangement of the rows and columns of our basis matrix so that R is sparse. In linear programming there is a trade-off between pivoting to achieve numerical accuracy and pivoting to achieve sparse LU-factors. However in our experiment with set-partitioning problems from the airline industry,

numerical instability seems not to arise, so we concentrate on a pivoting to preserve sparsity.

We now give an example that demonstrates the advantage of interchanging rows and columns in QR-factorization. Consider finding a QR-factorization for the matrix

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix},$$

without permuting the rows or the columns. The upper triangular matrix R can be obtained by performing Givens rotations. The first Givens rotation matrix is

$$G_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} & & & & \frac{1}{\sqrt{2}} \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ -\frac{1}{\sqrt{2}} & & & & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

$G_1 \cdot B$ is

$$\begin{bmatrix} \sqrt{2} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

Observe that the sparse pivot row becomes fully dense after the Givens rotation. Similarly, if we pivot on the diagonal elements, without interchanging the rows and the columns, then

we have

$$G_4 \cdot G_3 \cdot G_2 \cdot G_1 \cdot B = R = \begin{bmatrix} \sqrt{2} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{\sqrt{3}}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ 0 & 0 & \frac{2}{\sqrt{3}} & \frac{1}{2\sqrt{3}} & \frac{1}{2\sqrt{3}} \\ 0 & 0 & 0 & \frac{\sqrt{5}}{2} & \frac{1}{2\sqrt{5}} \\ 0 & 0 & 0 & 0 & -\frac{1}{\sqrt{5}} \end{bmatrix}.$$

Thus R is completely dense.

Now observe that if we interchange the first and the last columns of B, then interchange the first and the last rows, we have

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Suppose we have matrix B which is a basis (linearly independent columns of E) in the NNLS algorithm. We consider $\tilde{B} = M_1 B M_2$ for QR-factorization, where M_1 and M_2 are the permutation matrices of size $m \times m$ and $n \times n$, respectively. It is a well known fact ([16]) that the sparsity of R is dependent on M_2 and the number of computations is highly dependent on M_1. From our previous example we can see that it requires five Givens rotations for QR-factorization, and the upper-triangular part is totally full without row and column permutations and no Givens rotation is required for QR-factorization with one column permutation and one row permutation.

Fill-in is an increase in the number non-zero elements during matrix operations. Symbolic factorization is factorization without considering actual values and does not consider accidental cancellation of the elements during factorization. The number of operations in symbolic factorization is a good upper-bound on the actual operations since the accidental cancellations rarely happen. Hence most of the graph-related algorithms to get minimum or small fill-ins construct their graph with symbolic factorization. In our approach, we use symbolic factorization because of the simplicity of the symbolic computation, but our

algorithm is not graph-related.

One of the most famous results about the sparsity for the simplex algorithm (LU-factorization) is from Markowitz [28]. Markowitz's rule finds the pivot element with the minimum $(r-1)(c-1)$ values out of all the candidates, where r and c are the number of non-zero elements in the row and the column where the candidate element is located. This method is also numerically very stable and can produce a very sparse matrix. The value $(r-1)(c-1)$ of a pivot element is the upper bound on the number of fill-ins if the element becomes the pivot element and this is exact fill-in if the elements in each row have only one common column index, which is the column index of the pivot element. So this number $(r-1)(c-1)$ is not the exact fill-in but works well in practice.

We are going to describe a new procedure to find a pivot element that gives a good ordering of the columns and the rows. Since finding the best column ordering of the matrix to generate a sparsest matrix is a NP-hard problem([34]), our approach is heuristic.

We illustrate our method with an example first before introducing notation. We assume that there is only one common column index in all rows, that is the pivot row, in Figure 5(a). Each nonzero element is represented as $\times$. The first Givens rotation makes our matrix Figure 5(b). Newly created fill-in elements are represented as $\boxtimes$. Notice that the second row makes a fill-in in the first row. One more Given rotation makes Figure 5(b) into Figure 5(c). Notice that both rows affect each other so that three fill-ins are created. One more Givens rotation makes our matrix Figure 5(d). Observe that fill-ins created in each row are the union of the column indices in both rows.

Suppose column c has r entries in row $1, 2, ..., r$. Let V_i denote the set of column indices in row i and let v_i denote the number of non-zero elements in row i. If we assume that for all i and j, $i \neq j$, $V_i \cap V_j = \{c\}$ ($|V_i \cap V_j| = 1$), then the fill-ins (in fact, the upper bound) for the column operation are easily obtained. Assume that the pivot element in column c is the first element in column c (row 1), and a Givens rotation is applied for the first row and the other rows, one by one, from the second to the last.

After the first Givens rotation for row 1 and 2, the set of nonzero column indices of the first row becomes $V_1 \cup V_2$ and the set of nonzero column indices of the second row becomes

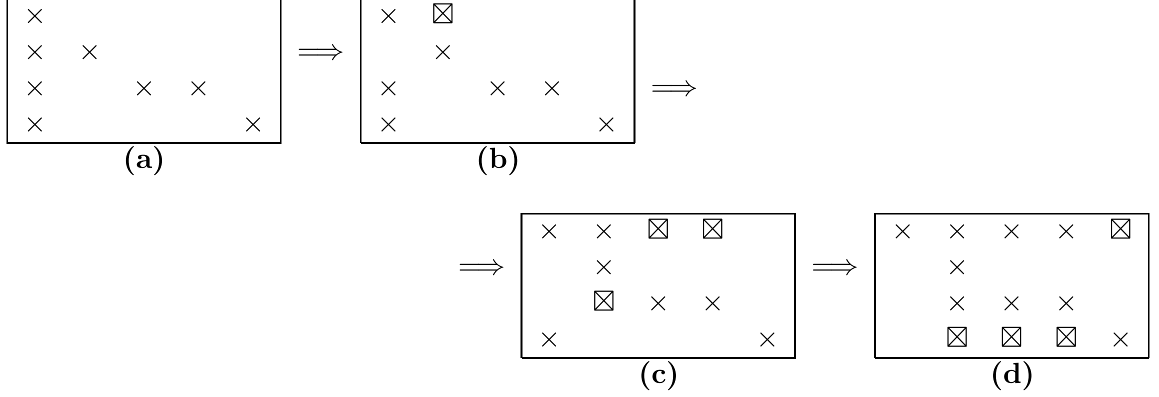

Figure 5: Fill-in : Pivoting on the Element in the $(1,1)$ Position

$(V_1 \cup V_2)\backslash c$. $|V_1 \cup V_2| = v_1 + v_2 - 1$, and $|(V_1 \cup V_2)\backslash c| = v_1 + v_2 - 2$. The number of fill-ins caused by this iteration in the second row is $v_1 - 1$. After the second Givens rotation is done, the set of nonzero column indices of the first row becomes $V_1 \cup V_2 \cup V_3$, whose size is $v_1 + v_2 + v_3 - 2$, and the set of nonzero column indices of the third row becomes $(V_1 \cup V_2 \cup V_3) \setminus c$, whose size is $v_1 + v_2 + v_3 - 3$. The number fill-ins caused by this iteration in the third row is $v_1 + v_2 - 2$. In similar fashion, the number of fill-ins in the ith row is $\sum_{k=1}^{i-1} v_k - (i-1)$. Then the total fill-ins except the first row are $(r-1)v_1 + (r-2)v_2 + ... + v_{r-1} - \frac{r(r-1)}{2} = \sum_{i=1}^{r-1}(r-i)v_i - \frac{r(r-1)}{2}$. After all iterations are done, the number of fill-ins in the first row is $\sum_{k=2}^{r} v_k - (r-1)$. The sum of all the fill-ins in this operations for one pivot column is

$$\sum_{i=1}^{r-1}(r-i) \cdot v_i + \sum_{i=2}^{r} v_i - \frac{r(r-1)}{2} - (r-1). \tag{9}$$

We choose a column which has minimum value in (9) as the pivot column out of the candidate columns. This rough upper-bound of the fill-in in QR-factorization has the same assumption as Markowitz's rule [28]. Observe that (9) is dependent on the order of the rows. Therefore each time (9) is calculated, the row which has the smallest value v_i becomes the pivot element and the Givens rotations are applied, based on the value v_i, the smaller the earlier, so that (9) becomes as small as possible. After the first pivot column is chosen Givens rotations are applied based in the same order we have calculated in (9). After the first column is orthogonalized the first column and the first row are not our concern so these are excluded from the later computation for the choosing pivot column. The later iterations

for choosing the pivot column work in the same way as described.

The best local strategy is to compare column indices so that we can predict the exact amount of fill-in. However it is too expensive to compare the column indices because sparse matrix storage is used (see the next section).

The above mentioned procedure is possible if swapping column order is not expensive since this column ordering strategy is not an in-advance ordering like the minimum degree algorithm. The minimum degree algorithm needs to be applied before the factorization starts, so there is no column ordering needed during factorization. The data structure for our re-factorization algorithm is dynamic so that it works at the same time as QR-factorization of B. So during the re-factorization process the data structure is modified a little bit differently from general updating before re-factorization starts. The difference is that the column indices of each row are not in ascending order, so it does not need to swap entries of two columns.

There is another modification for the simple strategy of keeping matrix R as sparse as possible in view of the data structure. As iterations go on, the lower right part of the matrix, which is not processed yet, becomes very dense. Then a sparse matrix doesn't have any advantage in memory space and number of operations. Hence, if the density of the lower right part of the matrix becomes bigger than some value, the data in the lower right part is moved to full matrix format and re-factorization goes on. Instead (9) becomes far from the correct number because the assumption cannot be realistic. So, instead, we choose the pivot column with minimum number of non-zero elements in the candidate column.

We need to mention the numerical stability. We already mentioned that there is no numerical concern for choosing pivot element. But it is possible that B may include linearly dependent columns at the re-factorization stage because of the numerical error accumulated during Givens rotation. When this happens, at some point during the orthogonalization, the unprocessed part of B has a column which has almost zero norm. When it is detected the column is removed immediately.

One easily available software package for finding a good ordering of the columns is COLAMD [8], which is a minimum degree algorithm. This uses a column approximate

minimum degree algorithm and is downloadable from

http://www.cise.ufl.edu/research/sparse/colamd/COLAMD-2.5.1.tar.gz.

We have compared the performance of our algorithm with COLAMD. The result from this preliminary comparison study in Chapter 3.3 shows that our simple strategy works as good as the existing software in case of set-partitioning instances.

2.4 Data Structure

To develop an efficient linear programming solver, it is important to develop data structures that are convenient for the calculations used by the algorithm. We developed such data structures for the NNLS algorithm.

Golub [18] and Björck [5] describe some of the existing data structures for QR-factorization. The structures they describe assume that the matrix to be factorized is fixed. In our case we must factor a basis matrix that changes by adding or dropping columns at each iteration. The QR-factorization must be updated each time the basis changes, and we need a structure that can be updated quickly. Lawson and Hanson [25], who developed the NNLS algorithm, proposed several data structures for this problem, but they did not consider a structure that is suitable for sparse matrices. Their first method stores the explicit Q^T matrix and uses a Householder transformation for updating Q when a column is added to the matrix to be factored. They use Givens rotations when a column is deleted. Their second method stores Q as a product of Householder transformations only. Their third method, the one implemented in **FORTAN** in Lawson and Hanson [25], when applied to the problem $\{\min ||b - Ex_E||^2, x \geq 0\}$, reduces the entire matrix E to row echelon form by multiplying on the left by Q^T. This amounts to maintaining an update of all nonbasic columns of E, as is done in the tableau form of the simplex algorithm.

The main computation in our NNLS algorithm requires solving a problem of the form $||b - Bx_B - A_s x_s||^2$. If $B = QR$ is a QR-factorization of B we must compute $Q^T A_s$, perform some Givens rotations, and solve the triangular system $\bar{R}x_B = \bar{b}$. Thus we need to store Q^T and R in a way that facilitates these computations. In particular, we must store R in a way to make the Givens rotation easier to perform. We find it convenient to store Q^T as

a product form of Givens rotations.

We can describe this matrix by three elements; cos, i, and j. sin can be computed as $\sqrt{1 - cos^2}$. However, it is better to store G as four elements; cos, sin and two row indices i, j. cos and sin are stored in an array in double precision, and i and j are stored in an integer array. Thus a product $G_k G_{k-1} \cdots G_1$ of Givens rotations is stored in two arrays that are shown in Figure 6.

cos_1 sin_1	cos_2 sin_2	$\cdots$ $\cdots$ $\cdots$ $\cdots$ $\cdots$	cos_k sin_k

i_1 j_1	i_2 j_2	$\cdots$ $\cdots$ $\cdots$ $\cdots$ $\cdots$	i_k j_k

Figure 6: Data Structure: Q^T

, (cos_r and sin_r are the cos and sin values for G_r) and i_r and j_r are the i and j indices for G_r. Updating Q^T is very easy since we can write G_{k+1} at the end of the arrays for the current Q^T.

It is very time consuming to compute $Q^T A_s$ if A_s is stored in a sparse matrix format. The reason is that in forming the products $G_k(\cdots(G_2(G_1 A_s))\cdots)$ new nonzeros are likely to form at each stage, and it is cumbersome to insert these in the sparse array. Thus we expand A_s to an explicit array before computing the product $G_k(\cdots(G_2(G_1 A_s))\cdots)$. The product $Q^T b = G_k(\cdots(G_2(G_1 b))\cdots)$ is computed a bit differently from $Q^T A_s$. Since b never changes, we compute $G_1 b$ and update this product every time a new Givens rotation is added to Q^T. For the same reason we use the explicit form of A_s in computing $Q^T A_s$, we use the explicit form of the array b for computing $Q^T b$.

The data structure of R we developed is an extension of the row major form of sparse matrix format since a Givens rotation requires many row operations, so it is convenient for row operations. Row major form of sparse matrix format is represented with three arrays. First, all the elements are enumerated from the first row to the last row; each row is based on the order of the column indices. Second, each column index is enumerated from the first row to the last row; each row is based on the order of the column indices. Third, the position of each column in the above two arrays is stored so that we can directly access each row without searching. This sparse matrix representation is widely used in many of the matrix applications. But it does not fit our purpose since it does not have the necessary

$$\begin{bmatrix} 3 & 1 & 2 & 2 \\ & 3 & 7 & 4 \\ & & 9 & 8 \\ & & & 7 \end{bmatrix}$$

Figure 7: Example : Data Structure Matrix

features we need. Hence we developed a data structure which fits our need.

We will describe a brief overview of the data structures of R since they are heavily dependent on the implementation. We developed a data structure of R based on its necessary functionality, quick addition of column $Q^T A_s$ and deletion of middle of the column and easy row access for Givens rotation. Our extension of the row major form of sparse matrix format makes Givens rotation easy. For example, suppose we have the matrix as in Figure 7. This is stored in our data structures, in Figure 8.

3	1	2	2	×	×	3	7	4	×	×	9	8	×	×	7	×	×	⋯	*entry*
2	4	3	1	×	×	4	3	1	×	×	1	3	×	×	3	×	×	⋯	*col_ind_pointer*

1	7	12	16	⋯	*rowbeg*
3	1	4	2	⋯	*col_ind*

Figure 8: Data Structure: R

The first array *entry* stores entries of each row based on their column indices. The second array *col_ind_pointer* stores the position of column index so that we need to look at the position in *col_ind* to retrieve column index. The third array *col_ind* stores actual indices of the columns. Fourth array *rowbeg* stores the position of the first entry of each row. If we want to read the first element in the 3rd row, then we need to find index $k = rowbeg[3] = 12$ first. Entry is obtained by $entry[k] = entry[12] = 9$ and column index is $col_ind[col_ind_pointer[k]] = col_ind[col_ind_pointer[12]] = col_ind[1] = 3$.

For a quick addition of a column, we made some empty space after each row, at positions with × in array *entry* and *col_ind_pointer* in Figure 8. Sometimes pre-assigned empty spaces for some row are used up and there is no more space for new entry for the row. Then we move the row to the other position, which is located at the end of the matrix, after the last × in array *entry* and *col_ind_pointer* in Figure 8, so that we can add more elements. For

quick deletion of column, we use *col_ind*. Suppose the 3rd column is removed. In this data structure we don't need to search and remove elements in array *entry* and *col_ind_pointer* corresponding to the 3rd column. To remove the 3rd column it is required to search 3 in *col_ind* and change it by -1. Then when we read any element from R we need to check that the column index is not -1. If it is, the element actually does not exist.

We explained there exists empty memory space for each row and empty space for fully used rows, but we didn't mention the size of such empty space. And we explained, the way we store the Q^T matrix is as a product form of transpose, but we didn't mention about the size of the the array we are going to use. The size of empty space for R and the size of Q^T are related. If the size of Q^T is too big then error can be accumulated. And if the size of Q^T is too small or the empty space for R is too small re-factorization is triggered too often. If the empty space of R is too large then it is a waste of the space. Hence our space allocation is based on computational experience. R is allocated a space of 40% of the full $m \times m$ matrix. Initially 70% is used for the rows and 30% for future usage for the fully used rows. Q^T is continually re-allocated in a bigger space until the first re-factorization is triggered by the high density of R and the size of Q^T is fixed until the algorithm terminates. In middle-sized instances this space can store approximately 2500 Givens rotations.

So far we have described the basic data structure. But there are necessities of modification during the re-factorization since re-factorization is important for the performance of the algorithm because it can reduce the number of Givens rotations and make R more sparse. We need all the same information we have described in the previous paragraph and some more structure to facilitate the re-factorization process. We store all the row indices of the columns so that we don't need to search the non-zero elements in the matrix. This data is stored in a similar data structure to R but only the row indices are stored; the elements are not stored. And we need to store the norm of the column which is not processed yet so that we can easily tell the dependent columns. For choosing the pivot column we have use the V_i value many times. So V_i values are stored and updated after each iteration so that we don't need to recompute every time they are required.

CHAPTER III

LEAST-SQUARES PRIMAL-DUAL ALGORITHM

3.1 Algorithm

We consider the linear programming problem in standard form

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad Ax &= b \\ x &\geq 0 \end{aligned} \tag{\textbf{P}}$$

and its dual

$$\begin{aligned} \max \quad & \pi b \\ \text{s.t.} \quad & \pi A \leq c, \end{aligned} \tag{\textbf{D}}$$

where A is an $m \times n$ matrix, and b and c are column and row vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. x and π are column and row vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively.

The LSPD algorithm for solving (**P**) begins with a dual feasible point π, that is, a point π satisfying $\pi A_j \leq c_j$, $j = 1, \ldots, n$. Let E be a matrix whose columns are those columns A_j in A satisfying $\pi A_j = c_j$. Assume that E has at least one column. Let x_E denote the components of x corresponding to the columns of E. For $A_j \notin E$, we have $\pi A_j < c_j$ and we set $x_j = 0$. If we solve $Ex_E = b,\ x_E \geq 0$, we will have solved (**P**), for we will have satisfied dual feasibility, primal feasibility, and the complementary slackness condition ($x^T(c - \pi A) = 0$). An interesting property of the primal-dual approach is that if the attempt to solve $Ex_E = b$, $x_E \geq 0$ fails, all is not lost. If this system does not have a solution, the very act of trying to solve it shows us how to update π to a dual feasible point π' satisfying $\pi' b > \pi b$. Thus we have a dual feasible point with a better objective value.

There are several ways to check if the system $Ex_E = b$, $x_E \geq 0$ is solvable. The first

approach by Dantzig et al. [24] and [31] is to solve the Phase I linear programming problem

$$\min \sum |\rho_i| \;\; s.t. \;\; Ex_E + \rho = b, \;\; x_E \geq 0.$$

In our approach we solve the nonnegative least-squares problem

$$\min \quad \tfrac{1}{2} \left\| b - Ex_E \right\|^2 \;, \;\; x_E \geq 0, \tag{10}$$

or, equivalently,

$$\begin{aligned} \min \quad & \tfrac{1}{2}\rho^T \rho \\ s.t. \quad & Ex_E + \rho &= b, \\ & x_E &\geq 0. \end{aligned}$$

Let x_E^* denote the solution of (10). If the minimum is zero, we have achieved primal feasibility for (**P**) at x^*, obtained by appending zeros to x_E^* in positions corresponding to the columns of $A \setminus E$. So assume the minimum in (10) is positive. By the Kuhn-Tucker optimality conditions for (10) we have,

$$E^T(b - Ex_E^*) \leq 0 \quad \text{and} \quad (x_E^*)^T E^T (Ex_E^* - b) = 0. \tag{11}$$

Let $\rho^* = b - Ex_E^*$. The first condition in (11) can be written as $\rho^{*T} E \leq 0$. It follows readily that $\pi' = \pi + t\rho^{*T}$ is dual feasible for (**D**) for sufficiently small positive values of t. In fact, $\pi' A_j = (\pi + t^* \rho^{*T}) A_j \leq c_j$ if

$$t^* = \min_{\rho^{*T} A_j > 0} \; \frac{c_j - \pi A_j}{\rho^{*T} A_j}.$$

The second condition in (11) can be written as $-(x_E^*)^T E^T \rho^* = 0$. It follows that

$$\pi' b = \pi' b - t^* (x_E^*)^T E^T \rho^* = \pi b + t^* \rho^{*T} (b - Ex_E^*) = \pi b + t^* \rho^{*T} \rho^* > \pi b$$

for $t^* > 0$. We will therefore take $t = t^*$ since we want π' to be feasible and $\pi' b$ as large as possible. Note that the matrix E corresponding to π' contains at least one new column A_j with $\rho^{*T} A_j > 0$.

At this point we have replaced the dual feasible point π with a dual feasible point π' satisfying $\pi b < \pi' b$. This completes one step of the LSPD algorithm. This step is repeated

until we find a matrix E such that the minimum in (10) is zero. The corresponding solution of $Ex_E = b$, $x_E \geq 0$ gives the solution of (**P**).

Thus we have described a primal-dual method for solving (**P**). Solving the NNLS problem is already described in Chapter 2.

Algorithm 2 LSPD : min cx : $Ax = b$, $x \geq 0$

1: Initialization : dual feasible point π
   (assuming that there exists at least a column whose reduced cost is zero.)
2: **loop**
3: $E = \{A_j \in A | \pi A_j = c_j\}$,
4: $x_E^* = argmin\{||b - Ex_E||^2 : x_E \geq 0\}$
5: $\rho^* = b - Ex_E^*$
6: **if** $\rho^* = 0$ **then**
7: STOP π is dual and $x = [x_E^*, x_{A \setminus E} = 0]$ is primal optimal solution
8: **end if**
9: $S = \{A_j \in A | \rho^* A_j > 0\}$
10: **if** $S = \emptyset$ **then**
11: STOP : dual is unbounded and primal is infeasible
12: **else**
13: $t^* = min\{\frac{c_j - \pi A_j}{\rho^* A_j} : j \in S\}$
14: **end if**
15: $\pi \leftarrow \pi + t^* \rho^*$
16: **end loop**

3.2 Improving Convergence Rates

We have experimented with a variation on the basic LSPD algorithm to enhance its performance. For medium and large size problems, the convergence rate can be improved by relaxing the complementary slackness condition. In the later stage of the variation, the complementary slackness condition is restored to get an optimal solution. The variation has two assumptions. First, a dual improvement direction ρ becomes better if we have larger size E regardless of the quality of E. Second, columns with large $c_j - \pi A_j$ values in E have less probability to have $c_j - \pi A_j = 0$ in later iterations.

The variation has three phases. The first phase is for obtaining a feasible primal solution without complementary slackness conditions. The second phase is removing some columns which have large $c_j - \pi A_j$ from B and E so that we are close to exact complementary slackness. The third phase is restoring the complementary slackness conditions.

The first phase starts with the same situation as the basic algorithm, a dual feasible point π. We construct a dual approximate equality set E with $c_j - \pi A_j < \epsilon$, where ϵ is a certain positive value, so that we have a bigger E. Then we may find better direction while reducing the norm of the residual. If ϵ is too small, the convergence rate to primal feasibility is too slow but we have a better dual solution. If ϵ is too big, the convergence rate is fast but we cannot get a good dual point. The second phase is removing columns with large $c_j - \pi A_j$ from B and E so that we have better complementary slackness condition. In the third phase we will solve the exact NNLS algorithm with a new E which has the columns satisfying $c_j = \pi A_j$. All the columns with positive $c_j - \pi A_j$ are removed from its B and E at the beginning of the third phase. If too many columns are removed, then it takes too much time to restore primal feasibility.

In the next sub-chapter, we will show computational results for the algorithm and the variation which yields better convergence rates.

3.3 Computational Results

We compared the LSPD algorithm to the primal and the dual simplex algorithms in ILOG CPLEX 9.020. We turned off the CPLEX preprocessor to ensure a fair comparison between algorithms. The LSPD was implemented in the C++ programming language but without using objective-oriented features. All tests were performed using one processor on a Sun 900MHz 4-processor Ultrasparc-IIICu with 16GB RAM running Solaris 9.

We obtained our test instances from the standard sources. All the instances are obtained from OR-Library[4]. Set-partitioning instances solved by the simplex algorithm show a lot of degeneracy. Since LSPD and NNLS do not allow degeneracy, we have strict improvement in each iteration, so that our result is more promising in this kind of highly degenerate problem. We categorize the instances into three, small, medium and large. There are many small instances and they are trivial to solve. The medium size instances have less than 200 columns and the large instances have more than 400 columns. The sizes of the problem and the objective values are listed in Table 2. The CPU time and the iterations needed to solve the small, medium, and large instances are in Table 3 and Table 4. These

tables include CPLEX Primal and Dual solver results and LSPD algorithm results. Table 4 includes results for the variation of LSPD in the column of the name **LSPD var.** and for the basic algorithm in **LSPD basic**. Columns labeled **time** are CPU times in sec. and **Iter** means the number of major iterations.

Table 1 shows the results of the re-factorizations. # is the number of Givens rotations in Q^T. The instances are obtained from the optimal basis of the some of the instances. After optimal basis is obtained re-factorization is done with the procedure explained in Chapter 2.3 and COLAMD [8]. The results show that the superior performance of the procedure. Since COLAMD is developed for the sparse R matrix not for the small number of Givens rotations in Q^T. So most of the time spent is for obtaining Givens rotations.

Table 1: Density

Size		**Prev.**	**New procedure**			**COLAMD**		
# row	**# col**	**den.**	**den.**	**time**	**#**	**den.**	**time**	**#**
100	12	0.229	0.083	0.000	101	0.083	0.010	100
124	63	0.076	0.052	0.000	290	0.055	0.000	555
135	71	0.014	0.014	0.000	134	0.014	0.010	134
139	125	0.018	0.008	0.000	137	0.008	0.010	136
145	72	0.023	0.014	0.000	145	0.014	0.020	145
163	36	0.090	0.075	0.000	342	0.074	0.030	500
426	245	0.227	0.149	0.270	5933	0.134	0.150	13695
531	81	0.060	0.012	0.020	532	0.012	0.220	532
646	161	0.085	0.032	0.040	1345	0.034	0.760	3401
801	196	0.081	0.043	0.070	2212	0.042	0.680	5110
825	168	0.071	0.029	0.060	1528	0.031	0.710	3229

Since the results on the small instances are trivial to compare, we don't analyze these results. The results from the medium and large size instances show that in most of the cases the performance of the LSPD variation with approximation (which is always faster than the LSPD basic version) is ranked first or second compared to CPLEX Primal and Dual results, the first rank in eight instances and the second rank in six instances. For most of the instances, CPLEX Dual shows the best running time and LSPD is between CPLEX Primal and Dual. For the instances which LSPD is slower than CPLEX Primal and Dual, LSPD results are a little bit slower than CPLEX Primal results.

When we compare the number of iterations, the number of iterations in LSPD are similar to the number of iterations in CPLEX Dual and much better than CPLEX Primal. Modern simplex dual algorithms have a special feature, long step, that decreases the number of iterations significantly whenever there are upper bounds on the variables, and the set-partitioning instances from OR-library have upper bounds on each variable.

Table 5 has the detailed results for medium and large size instances. The basic LSPD computational results show that more than 90% of time is spent on pricing (computing t values for dual updates and obtaining E) in most of the cases. Likewise, the LSPD variation spends around 90% for the pricing step even though there is improvement in the computational times. The column labeled **Pricing** has the percentage in CPU time for pricing. **Phase I**, **Phase II**, and **Phase III** show the percentage of CPU time spent for each phase in the LSPD variation. For most of the instances, Phase III is the main time consuming phase, attaining a complementary slackness. The number of LSPD iterations is in column **Iter** and the number of major iterations in NNLS as column **Addition** are listed in Table 5. The number of re-factorizations for LSPD are listed as **Ref**. The addition of entering columns takes around 80% of the whole running time for the NNLS algorithm. Re-factorization is very small portion of the NNLS computation.

Table 2: Instances: Set-Partition

Instance	Row	Column	Optimal Solution
sppaa01	823	8904	55535.436388
sppaa02	531	5198	30494.000000
sppaa03	825	8627	49616.363636
sppaa04	426	7195	25877.609268
sppaa05	801	8308	53735.928571
sppaa06	646	7292	26977.187500
sppkl01	55	7479	1084.000000
sppkl02	71	36699	215.250000
sppnw01	135	51975	114852.000000
sppnw02	145	87879	105444.000000
sppnw03	59	43749	24447.000000
sppnw04	36	87482	16310.666667
sppnw05	71	288507	132878.000000
sppnw06	50	6774	7640.000000
sppnw07	36	5172	5476.000000
sppnw08	24	434	35894.000000
sppnw09	40	3103	67760.000000

Continued on next page

Table 2 (continued)

Instance	Row	Column	Optimal Solution
sppnw10	24	853	68271.000000
sppnw11	39	8820	116254.500000
sppnw12	27	626	14118.000000
sppnw13	51	16043	50132.000000
sppnw14	73	123409	61844.000000
sppnw15	31	467	67743.000000
sppnw16	139	14863	1181590.000000
sppnw17	61	118607	10875.750000
sppnw18	124	10757	338864.250000
sppnw19	40	2879	10898.000000
sppnw20	22	685	16626.000000
sppnw21	25	577	7380.000000
sppnw22	23	619	6942.000000
sppnw23	19	711	12317.000000
sppnw24	19	1366	5843.000000
sppnw25	20	1217	5852.000000
sppnw26	23	771	6743.000000
sppnw27	22	1355	9877.500000
sppnw28	18	1210	8169.000000
sppnw29	18	2540	4185.333333
sppnw30	26	2653	3726.800000
sppnw31	26	2662	7980.000000
sppnw32	19	294	14570.000000
sppnw33	23	3068	6484.000000
sppnw34	20	899	10453.500000
sppnw35	23	1709	7206.000000
sppnw36	20	1783	7260.000000
sppnw37	19	770	9961.500000
sppnw38	23	1220	5552.000000
sppnw39	25	677	9868.500000
sppnw40	19	404	10658.250000
sppnw41	17	197	10972.500000
sppnw42	23	1079	7485.000000
sppnw43	18	1072	8897.000000
sppus02	100	13635	5965.000000
sppus03	77	85552	5338.000000
sppus04	163	28016	17731.666667

Table 3: CPU Time: LSPD - small

Instance	CPLEX Primal		CPLEX Dual		LSPD	
	CPU time	**Iter.**	**CPU time**	**Iter**	**CPU time**	**Iter**
sppnw06	0.14	924	0.13	97	0.17	104
sppnw07	0.06	366	0.04	22	0.05	37
sppnw08	0.00	29	0.00	28	0.00	31
sppnw09	0.04	417	0.03	45	0.08	105
sppnw10	0.01	85	0.01	27	0.00	31
sppnw11	0.08	384	0.08	55	0.15	84
sppnw12	0.00	37	0.00	23	0.01	29
sppnw15	0.01	84	0.01	18	0.00	19
sppnw19	0.05	326	0.03	44	0.04	53
sppnw20	0.01	146	0.01	34	0.01	48
sppnw21	0.01	45	0.00	15	0.01	20
sppnw22	0.01	83	0.00	22	0.00	19
sppnw23	0.01	82	0.00	33	0.00	37
sppnw24	0.00	20	0.01	15	0.01	16
sppnw25	0.01	53	0.00	22	0.01	36
sppnw26	0.01	61	0.00	24	0.00	34
sppnw27	0.01	52	0.01	16	0.01	18
sppnw28	0.01	57	0.01	16	0.01	27
sppnw29	0.02	124	0.03	48	0.02	47
sppnw30	0.03	105	0.01	18	0.03	36
sppnw31	0.03	183	0.02	29	0.03	50
sppnw32	0.01	16	0.01	20	0.01	30
sppnw33	0.03	120	0.02	20	0.01	27
sppnw34	0.01	100	0.00	22	0.00	35
sppnw35	0.02	81	0.02	18	0.01	20
sppnw36	0.02	159	0.03	51	0.03	47
sppnw37	0.01	87	0.01	19	0.00	20
sppnw38	0.01	48	0.00	33	0.01	33
sppnw39	0.01	60	0.01	12	0.01	16
sppnw40	0.00	56	0.01	18	0.00	20
sppnw41	0.00	19	0.00	13	0.00	16
sppnw42	0.01	97	0.01	29	0.01	29
sppnw43	0.01	71	0.01	25	0.01	41

Table 4: CPU Time: LSPD - medium and large

Instance	CPLEX Primal		CPLEX Dual		LSPD basic		LSPD var.	
	time	**Iter**	**time**	**Iter**	**time**	**Iter**	**time**	**Iter**
sppkl01	0.10	505	0.18	151	0.37	201	0.37	116
sppkl02	0.99	3388	0.97	191	0.85	119	0.80	98

Continued on next page

Table 4 (continued)

Instance	CPLEX Primal		CPLEX Dual		LSPD basic		LSPD var.	
	time	Iter	time	Iter	time	Iter	time	Iter
sppnw01	1.66	3270	0.53	130	1.64	135	1.38	101
sppnw02	3.80	5339	1.37	154	4.44	144	3.11	107
sppnw03	0.91	1018	0.68	88	1.47	143	0.63	48
sppnw04	1.75	799	2.35	136	4.05	159	1.96	55
sppnw05	15.17	5077	7.11	132	34.14	256	11.31	74
sppnw13	0.19	484	0.20	85	0.36	104	0.28	64
sppnw14	3.65	1812	2.57	154	11.74	235	4.51	77
sppnw16	20.83	20067	31.25	626	32.39	584	13.23	802
sppnw17	4.82	1081	3.27	100	5.27	108	2.69	75
sppnw18	0.42	2544	0.59	393	2.00	722	0.92	139
sppus02	0.34	594	0.33	95	0.32	91	0.27	45
sppus03	5.65	1758	2.66	85	4.94	157	1.23	28
sppus04	1.24	3137	1.00	226	1.96	299	1.02	88
sppaa01	34.65	28517	8.97	4107	10.13	1327	8.31	987
sppaa02	9.59	18919	0.77	1067	6.14	943	5.78	899
sppaa03	38.75	37595	4.24	2476	34.95	2163	20.12	1729
sppaa04	4.19	10101	2.37	1578	9.10	1332	6.69	821
sppaa05	25.62	25760	4.56	2846	31.66	2241	26.45	2001
sppaa06	21.02	30577	1.77	1463	16.83	1651	14.87	1511

Table 5: Execution Details: LSPD Variation

Instance	Phase I	Phase II	Phase III	Pricing	Iter	Addition	Ref
sppkl01	34	7	59	89	116	343	3
sppkl02	22	10	68	94	98	378	2
sppnw01	28	21	51	95	101	336	3
sppnw02	24	9	67	89	107	364	3
sppnw03	36	17	47	92	48	289	3
sppnw04	18	12	70	92	55	279	2
sppnw05	28	6	66	97	97	359	4
sppnw13	38	9	53	93	64	182	2
sppnw14	36	8	56	91	77	402	3
sppnw16	59	4	37	81	207	801	14
sppnw17	22	7	71	94	75	197	2
sppnw18	74	3	23	77	139	426	7
sppus02	25	50	25	88	45	214	2
sppus03	36	14	50	90	28	142	4
sppus04	62	32	6	74	88	428	9
sppaa01	34	9	57	92	987	3245	9
sppaa02	22	13	65	94	899	2927	7
sppaa03	17	7	76	89	1729	4325	12

Continued on next page

Table 5 (continued)

Instance	Phase I	Phase II	Phase III	Pricing	Iter	Addition	Ref
sppaa04	31	13	56	85	821	2004	7
sppaa05	19	24	57	96	2001	4857	21
sppaa06	27	5	68	98	1511	3924	19

3.4 Concluding Remarks

The computational results show that the LSPD algorithm's performs is between CPLEX Primal and Dual solvers. It is not a bad result since the LSPD algorithm requires more research about updating sparse QR factorization, sparsity and numerical stability, and more. Our research and implementation are just one effort to show that the LSPD algorithm has practical implications.

The iteration results in [3] show that the number of iterations of LSPD are better than the simplex primal and dual algorithms. But the dual simplex implementation used in the test may not have dual long step feature so that the iterations of CPLEX Dual are almost 10 times better than the results listed in [3]. In the same fashion, we devise a long-step method but it includes an upper bound version of the algorithm and other complexities. Our future research includes these results.

CHAPTER IV

A LEAST-SQUARES NETWORK FLOW ALGORITHM

4.1 Algorithm

In this chapter, we are going to describe the least-squares network flow algorithm. There is only one difference between LSPD and LSNF, and that is the way the least-squares problem is solved. The LSNF algorithm is a special algorithm tailored to the min-cost network flow problem. The structure of the node-arc incidence matrix in a min-cost network flow problem makes it possible to compute primal solution x and residual ρ efficiently. We explain this in chapter 4.1.1.

The minimum cost network flow (**MCNF**) problem [1] can be stated as

$$\begin{aligned} \min cx & & \\ \text{s.t } Nx &= b \\ x &\geq 0, \end{aligned} \qquad (\textbf{MCNF})$$

where N is a p by q node-arc incidence matrix of a directed graph $N = (V, \mathbb{E})$ with $\sum_{i=1}^{p} b_i = 0$. Henceforth we shall refer to N both as a matrix and a network according to the context in which it is used. A typical column of N will be denoted by A_j. Let π be a dual feasible point for (**MCNF**). Let E denote the admissible arc set, which are the columns in N, satisfying $\pi A_j = c_j$. The primal-dual approach requires us to solve the NNLS problem

$$\begin{aligned} \min \quad & ||b - Ex_E||^2 \\ \text{s.t} \quad & x_E \geq 0. \end{aligned} \qquad (12)$$

Let B denote a basis for (12), which is a set of independent columns in E. In the network flow problem B represents a set of arcs in N that does not create any cycle in the underlying graph of N. Therefore (V, B) corresponds to a tree or a forest in the network N. Since the columns of B correspond to a forest we can write B as

$$B = \begin{pmatrix} T_1 & \mathbf{O} & . & . & . & \mathbf{O} \\ \mathbf{O} & T_2 & \mathbf{O} & . & . & \mathbf{O} \\ . & \mathbf{O} & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & \mathbf{O} \\ \mathbf{O} & . & . & . & \mathbf{O} & T_k \end{pmatrix}, \tag{13}$$

where each T_i is the node-arc incidence matrix for a tree in N and some trees with one node may have empty columns. It follows that min $||b - Bx_B||^2$ = min $\|b_{T_1} - T_1 x_{T_1}\|^2$ + min $\|b_{T_2} - T_2 x_{T_2}\|^2 + \cdots$ + min $\|b_{T_k} - T_k x_{T_k}\|^2$, where $b = [b_{T_1}, b_{T_2}, \cdots, b_{T_k}]$ and $x_B = [x_{T_1}, x_{T_2}, \cdots, x_{T_k}]$, with b_{T_i} and x_{T_i}, $i = 1, 2, \ldots, k$, being subvectors of b and x corresponding to the tree T_i in the forest (V, B). That is, solving min$||b - Bx_B||^2$ is equivalent to solving many subproblems min$||b_{T_i} - T_i x_{T_i}||^2$. Since the trees T_i's for all $i = 1, \ldots, k$ have no nodes in common, each $||b_{T_i} - T_i x_{T_i}||^2$, for each $i = 1, \ldots, k$, can be solved independently. Thus for network flow problems the non-negative least-squares problems decompose into sequence of smaller problems on trees. These problems can be solved very efficiently.

Before going into detail we want to explain some terminology. For any tree T, we define a flow x_T and a residual ρ_T to be vectors satisfying $Tx_T + \rho_T = b_T$. Each component of the residual measures the violation of flow conservation in each node in T so that x_T is a feasible tree flow if the supply/demand on the tree nodes is $b_T - \rho_T$. The flows we encounter come from either a least-squares solution, or the convex combination process. Let x_T^* be a least-squares flow, that is, flow from the problem min$||b_T - Tx_T||^2$. In the following sub-chapter we are going to use one technique many times to prove the claims: adding constraints corresponding to some tree with flow x_T and residual ρ_T after convex combination.

In the following chapter we discuss some special least-squares properties of network flow problems that can be used in conjunction with appropriate data structures for manipulating trees to solve minimum cost flow problems quickly using the LSNF algorithm.

4.1.1 Least-Squares Properties

We begin this chapter with a discussion of the unconstrained least-squares algorithm. At the end of the chapter we show how some of the special properties of the network basis

matrices, which will be explained in this chapter, can benefit the LSNF algorithm.

Each major iteration of NNLS (updating a feasible basis B) is comprised of many minor iterations (solving a sequence of least-squares problems, forming convex combinations of solutions and dropping columns from $[B, A_s]$). The advantage of the unconstrained minimization problem

$$\min ||b_T - Tx_T||^2, \tag{14}$$

in the minor iteration, is that it is small and can be solved efficiently. This is proved in the following theorem.

Theorem 4.1.1 *Let T be a directed tree with $|T \in N|$ nodes. Let x_T^* solve min $\|b_T - Tx_T\|^2$, where $b_T = [b_1, b_2, \cdots, b_{|T|}]^T$ is the vector of the components of bcorresponding to nodes of T. Let $\rho_T^* = [\rho_1^*, \rho_2^*, \ldots, \rho_{|T|}^*]^T = (b_T - Tx_T^*)$ denote the residual of b_T corresponding to x_B^*. Then*

$$\rho_1^* = \rho_2^* = \cdots = \rho_{|T|}^* = \delta_T^* = \frac{\sum_{i \in T} b_i}{|T|}$$

where δ_T^* is the common residual for the tree T.

Proof The optimality condition for (14) is

$$T^T(b_T - Tx_T^*) = 0.$$

This condition says that ${\rho_T^*}^T A_j = 0$ for all columns A_j in T. Recall that the column A_j contains only two non-zeros:+1 and -1. If these elements occur in position r and s we have ${\rho_T^*}^T A_j = \rho_r^* - \rho_s^* = 0$, which means $\rho_r^* = \rho_s^*$. It follows that all components in $\rho_T^* = b_T - Tx_T^*$ have the same value. When we add the rows of the equation $Tx_T^* + \rho_T^* = b_T$ we obtain

$$\sum_{i \in T} \rho_i^* = \sum_{i \in T} b_i.$$

Since all ρ_i^*'s in T have the same value, we have

$$\delta_T^* = \rho_1^* = \cdots = \rho_n^* = \frac{\sum_{i \in T} b_i}{|T|}.$$

■

From above theorem we can write $\rho_T^* = (\delta_T^*, \delta_T^*, \ldots, \delta_T^*)$ where $\delta_T^* = \frac{\sum_{i \in T} b_i}{|T|}$. Let B denote the basis matrix described in (13) and let $\rho^* = b - Bx_B^*$. Then $\rho^* = (\rho_{T_1}^*, \rho_{T_2}^*, \ldots, \rho_{T_k}^*)$ where $\rho_{T_i}^*$ is a vector with $|T_i|$ component, each equal to $\frac{\sum_{r \in T_i} b_r}{|T_i|}$.

This means that for a basis B (independent columns of E), we can compute the residual without solving any equations. In particular it is not necessarily to compute x_B^* in order to compute ρ^*.

The converse of the above theorem is also true, i.e., if x_T^* and ρ_T^* are vectors satisfying $Tx_T^* + \rho_T^* = b$ and each component of ρ_T^* has the same value, that is $\frac{\sum_{j\in T} b_j}{|T|}$, then x_T^* is a least-squares solution of min $||b_T - Tx_T||^2$. The proof of the assertion follows from the fact that the optimal solution of unconstrained convex optimization problem of min $\|b - Tx_T\|^2$ satisfies $T^T(b_T - Tx_T^*) = T^T\rho = 0$.

We have just seen that we can compute the residual of min $||b_T - Tx_T||^2$ without computing x_T^* . But for our purpose we will also need to know x_T^* to check non-negativity of the flow x_T^*. We will now explain how x_T^* can be computed. From the definition of node balance, it is easily seen that we can remove δ_T^* of the commodity from each of the nodes of T to make it a balanced tree, which means that $Tx_T = b - \rho_T$ has a feasible and unique solution. Consider any arc (i,j) in T whose least-squares flow we are interested in. Suppose (i,j) connects two trees T_i and T_j in tree T. The flow on the arc (i,j) in T is obtained by simply adding all the flow balance constraints of the nodes in T_i. This gives the net flow out of tree T_i through (i,j).

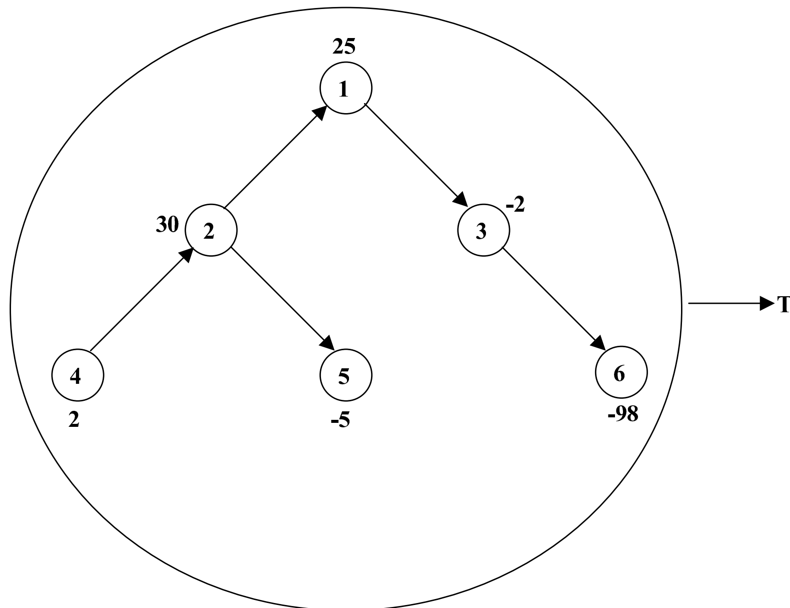

Figure 9: A tree T with Supplies and Demands on its Nodes

For example, consider computing flow x_T and the residual ρ_T of a tree in Figure

9. The residual is computed with the formula in Theorem 4.1.1 and $\delta_T^* = \frac{\sum_{i=1}^{6} b_i}{|T|} = \frac{25+30-2+2-5-98}{6} = -8$. We only compute x_{21} in this example. Observe that arc $(2,1)$ connects two subtrees in Figure 9, one is T_1 (nodes $2, 4$, and 5) and the other is T_2 (nodes $1, 3$, and 6). The node-arc incidence matrices for these trees are given below with corresponding right hand side vector b_T.

$$T = \begin{bmatrix} 0 & 0 & -1 & 1 & 0 \\ -1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix}, \quad b_T = \begin{bmatrix} 25 \\ 30 \\ -2 \\ 2 \\ -5 \\ -98 \end{bmatrix}. \tag{15}$$

Note that if we add the equations corresponding to nodes in tree T_1 we get

$$x_{21} + 3\delta_T^* = 30 + 2 - 5 = 27.$$

If we add the equations corresponding to the nodes in T_2 we get

$$-x_{21} + 3\rho_T^* = 25 - 2 - 98 = -75.$$

Since we know $\rho_T^* = 8$ we can easily calculate $x_{21} = 51$.

Figure 10 represents the block matrix structure of the node-arc incidence matrix of the tree T shown in Figure 9. The following Theorem shows it formally.

Theorem 4.1.2 *Let T be a directed tree and let x_T^*, ρ_T^*, and δ_T^* be as defined in Theorem 4.1.1. Let x_{ij} be the least-squares flow corresponding to a arc (i,j) in T. Let T_i and T_j be the subtrees of T connected by (i,j). Then*

$$x_{ij} = \sum_{k \in T_i} b_k - \delta_T^* |T_i| = \delta_T^* |T_j| - \sum_{k \in T_j} b_k.$$

Proof Adding all the equations of $Tx_T^* + \rho_T^* = b_T$ corresponding to nodes in T_i in Figure 10, we get

$$\begin{aligned} & x_{ij} + \sum_{k \in T_i} \delta_k^* = \sum_{k \in T_i} b_k \\ \Rightarrow \quad & x_{ij} = \sum_{k \in T_i} b_k - \delta_T^* |T_i|. \end{aligned} \tag{16}$$

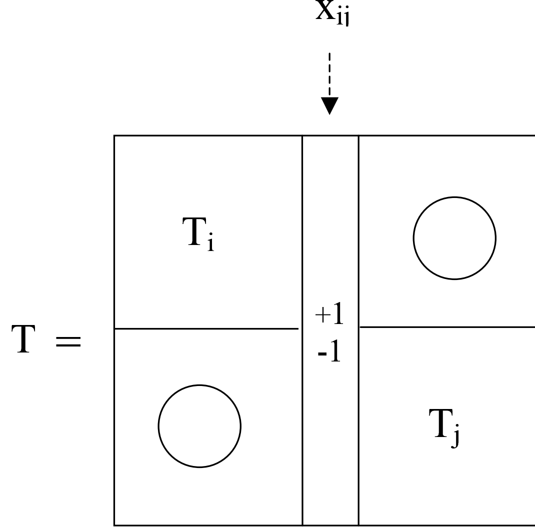

Figure 10: Matrix Notation for Tree T

Similarly, if we add all the equations corresponding to nodes in T_j we get

$$-x_{ij} + \sum_{k \in T_j} \delta_k^* = \sum_{k \in T_j} b_k$$

$$\Rightarrow \qquad x_{ij} = \delta_T^* |T_j| - \sum_{k \in T_j} b_k. \tag{17}$$

Combining (16) and (17) we get the required result. ■

Now consider finding an entering column A_s for a basis B. This operation is the same as joining two trees with an arc (i, j). We must find a column in $A_s \in E \backslash B$ with $\rho^{*T} A_s > 0$. If A_s corresponds to an arc from node i to node j, we have

$$\rho^{*T} A_s = \rho_i{}^* - \rho_j{}^* = \delta_{T_i}^* - \delta_{T_j}^* > 0 \tag{18}$$

where i is in T_i and j is in T_j. (Clearly node i and node j are not in the same tree since $\delta_{T_i}^* \neq \delta_{T_j}^*$.) The above equation says the common residual for T_i is larger than the common residual for T_j. Thus in searching for a column to enter the basis we need only consider arcs directed from higher to lower node residual. This arc (i, j) combines two trees T_i and T_j. This results in a positive flow on the entering arc. Intuitively a positive flow on the entering arc (i, j) implies that T_i has greater supply in some sense than T_j. This has been made more precise in the following theorem. Before stating the theorem we explain some notation.

During each iteration of the minor iteration of the NNLS algorithm the tree containing the entering arc (i, j) becomes smaller as some arcs are dropped from the tree during the convex combination procedure. Let T^k be the subtree of T that contains the arc (i, j) after some arcs are dropped from T^{k-1} during convex combination in the $(k-1)^{th}$ minor iteration. Let T_i^k and T_j^k denote the subtrees of T^k connected by arc (i, j) in T^k. Let $x^*_{T^k}$ be a least-squares flow in tree T^k and $x^C_{T_i^k}$ and $x^C_{T_j^k}$ be the flow in tree T_i^k and T_j^k after the convex combination operation is performed in $(k-1)^{th}$ minor iteration, e.g., $x^C_{T^k} = (1-\lambda)[{x^C_{T_i^{k-1}}}^T, x^C_{ij^{k-1}}, {x^C_{T_j^{k-1}}}^T]^T + \lambda x^*_{T^{k-1}}$ where $x^C_{ij^{k-1}}$ is a flow in (i, j) after convex combination operation at $(k-1)^{th}$ minor iteration. We define T^0=T, $T_i^0 = T_i$, $T_j^0 = T_j$, $x^C_{T^0} = x^*_T$, $x^C_{T_i^0} = x^*_{T_i}$, $x^C_{T_j^0} = x^*_{T_j}$, and $x^C_{ij^0} = 0$. Let $\delta^*_{T^k}$ be the common residual value of T^k and $\delta^C_{T_i^k}$ and $\delta^C_{T_j^k}$ be residual values corresponding to the nodes of T_i^k and T_j^k after the convex combination is performed in the $(k-1)^{th}$ minor iteration, i.e., $\delta^C_{T_i^k} = (1-\lambda)\delta^C_{T_i^{k-1}}$ $+ \lambda\, \delta^*_{T^{k-1}}$ and $\delta^C_{T_j^k} = (1-\lambda)\delta^C_{T_j^{k-1}} + \lambda\, \delta^*_{T^{k-1}}$. We define $\delta^*_{T^0}$=δ^*_T, $\delta^C_{T_i^0}$=δ_{T_i} and $\delta^C_{T_j^0}$=δ_{T_j}. We know that $\frac{\sum_{k \in T_i} b_k}{|T_i|} = \delta^*_{T_i} > \delta^*_{T_j} = \frac{\sum_{k \in T_j} b_k}{|T_j|}$. Theorem 4.1.3 also shows that $\delta^C_{T_i^k} > \delta^C_{T_j^k}$ for all k. We define *forward arcs* in the tree T^k as those that are either oriented towards node i in the subtree T_i^k or away from the node j in the subtree T_j^k of T^k. In Figure 11 the arcs (a, b) and (e, f) are forward arcs. *Reverse arcs* are the opposite of forward arcs. In Figure 11 the arcs (c, d) and (g, h) are reverse arcs.

With these notations and definitions we proceed to the next two least-squares properties of network flow problems.

Theorem 4.1.3 *Let $\delta^*_{T^k}$, $\delta^C_{T_i^k}$ and $\delta^C_{T_j^k}$ be as defined above. Then $\delta^C_{T_i^k} > \delta^*_{T^k} > \delta^C_{T_j^k}$.*

Proof Let x_{ij} be the flow on the entering arc (i, j) in T^k after the convex combination is performed. The equation $T^k x^C_{T^k} + \rho^C_{T^k} = b_{T^k}$ represents the flow and residual of tree T_i^k.

If we add all the rows corresponding to the nodes in T_i^k and T_j^k , then we have

$$x_{ij} = \sum_{r \in T_i^k} b_r - \delta^C_{T_i^k}|T_i^k| = \delta^C_{T_j^k}|T_j^k| - \sum_{r \in T_j^k} b_r$$

$$\Rightarrow \sum_{r \in T_i^k} b_r + \sum_{r \in T_j^k} b_r = \delta^C_{T_i^k}|T_i^k| + \delta^C_{T_j^k}|T_j^k|$$

$$\Rightarrow \delta^*_{T^k} = \frac{\sum_{r \in T_i^k} b_r + \sum_{r \in T_j^k} b_r}{|T_i^k| + |T_j^k|} = \frac{\delta^C_{T_i^k}|T_i^k| + \delta^C_{T_j^k}|T_j^k|}{|T_i^k| + |T_j^k|} \tag{19}$$

Since $\delta^C_{T_i^0} > \delta^C_{T_j^0}$, we have $\delta^C_{T_i^0} > \delta^*_{T^0} > \delta^C_{T_j^0}$ by (19) with. We can use induction to complete

the proof. Assume the inequality is true for $k-1$. Since $\delta^C_{T_i^{k-1}} > \delta^*_{T^{k-1}} > \delta^C_{T_j^{k-1}}$ and $\delta^C_{T_i^k}$ is a convex combination of $\delta^C_{T_i^{k-1}}$ and $\delta^*_{T^{k-1}}$, we have $\delta^C_{T_i^{k-1}} > \delta^C_{T_i^k} > \delta^*_{T^{k-1}}$ and similarly $\delta^*_{T^{k-1}} > \delta^C_{T_j^k} > \delta^C_{T_j^{k-1}}$. This implies that $\delta^C_{T_i^k} > \delta^C_{T_j^k}$. And it follows from (19) that $\delta^C_{T_i^k} > \delta^*_{T^k}$ $> \delta^C_{T_j^k}$. ■

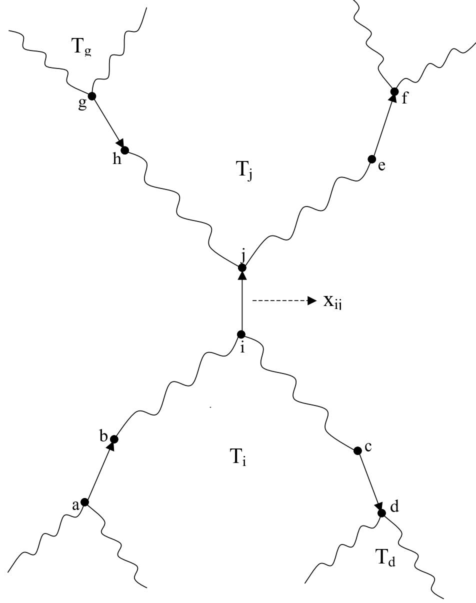

Figure 11: Forward Arcs: (a, b) and (e, f) and Reverse Arcs: (c, d) and (g, h)

We can use Theorem 4.1.3 to show that the forward arcs in the tree T^k will always have positive flow during the execution of the minor loop. We prove this in the following corollary of Theorem 4.1.3.

Corollary 4.1.4 *All forward arcs in the tree T^k have positive flow.*

Proof Let (a, b) be a forward arc in T^k. To fix the idea, assume $(a, b) \in T_i^k$. Let $T_a^k \subseteq T_i^k$ be the subtree of T^k rooted at the a and not containing the arc (a, b). Let ${x^C_{ab}}^k$ be the flow on the arc (a, b) at k^{th} minor iteration after convex combination. Let ${x^*_{ab}}^k$ be the least-squares flow at the kth minor iteration. We prove the result by induction on the minor iteration k of the NNLS algorithm.

From the feasibility assumption of T_i^0 we have ${x^C_{ij}}^0 > 0$, that is,

$$ {x^C_{ab}}^0 = \sum_{r \in T_a^0} b_r - \delta^C_{T_i^0} |T_a^0| > 0. $$

Then the least-squares solution after joining two trees T_i and T_j with arc (a, b) is

$$x_{ab}^{*\,0} = \sum_{r \in T_a^0} b_r - \delta^*_{T_i^0} |T_a^0| > 0.$$

Since $\delta^C_{T_i^0} > \delta^*_{T_i^0}$, we have $x_{ab}^{*\,0} > 0$. Hence the convex combination solution $x_{ab}^{C\,1} = (1 - \lambda) x_{ab}^{C\,0} + \lambda x_{ab}^{*\,0} > 0$.

Let us assume that $x_{ab}^{C\,p-1} > 0$ at $k = p - 1$. This implies that

$$x_{ab}^{C\,p-1} \quad = \sum_{r \in T_a^{p-1}} b_r - \delta^C_{T_i^{p-1}} |T_a^{p-1}| > 0.$$

We shall show that $x_{ab}^{C\,p} > 0$ after the least-squares computation when $k = p$.

If we add all the rows of the equation $T^{p-1} x^*_{T^{p-1}} + \rho^*_{T^{p-1}} = b_{T^{p-1}}$ corresponding to the nodes in T_a^{p-1} we obtain

$$x_{ab}^{*\,p-1} \quad = \sum_{r \in T_a^{p-1}} b_r - \delta^*_{T^{p-1}} |T_a^{p-1}|.$$

Since $\delta^C_{T_i^{p-1}} > \delta^*_{T^{p-1}}$ from Theorem 4.1.3 we have $x_{ab}^{*\,p-1} > 0$ after the least-squares computation in iteration $k = p - 1$. The convex combination solution is $x_{ab}^{C\,p} = (1 - \lambda) x_{ab}^{C\,p-1} + \lambda x_{ab}^{*\,p-1} > 0$. A similar argument shows that forward arcs in the tree T_j^k have positive flows during the kth iteration of the minor iteration in the NNLS algorithm. ∎

As a Corollary of Corollary 4.1.4 we can prove that entering arc (i, j) has positive flow.

Corollary 4.1.5 *Entering arc (i, j) in the tree T^k has positive flow.*

Proof We use the same notation in Corollary 4.1.4 with $(i, j) = (a, b)$. We prove the result by induction on the minor iteration k of the NNLS algorithm.

It is clear that $x_{ij}^{C0} = 0$. The flow from Theorem 4.1.2 at the entering arc is

$$\begin{aligned} x_{ij}^{*\,0} &= \sum_{r \in T_i^0} b_r - \delta^*_{T^0} |T_i^0| > 0 \\ &= |T_i^0| (\delta^C_{T_i^0} - \delta^*_{T^0}) > 0 \quad \text{by Theorem 4.1.3.} \end{aligned}$$

Hence the convex combination solution $x_{ij}^{C1} = (1 - \lambda) x_{ij}^{C0} + \lambda x_{ij}^{*\,0} > 0$.

Let us assume that $x_{ij}^{Cp-1} > 0$ at $k = p - 1$. This implies that

$$x_{ij}^{Cp-1} \quad = \sum_{r \in T_i^{p-1}} b_r - \delta^C_{T_i^{p-1}} |T_i^{p-1}| > 0,$$

if we add all equations $T^{p-1}x_{T^{p-1}} + \rho^C_{T_{p-1}} = b_{T_{p-1}}$ corresponding to rows in T_i^{p-1}. We shall show that $x_{ij}^{Cp} > 0$ after the least-squares computation, when $k = p$.

If we add all the rows of the equation $T^{p-1}x^*_{T^{p-1}} + \rho^*_{T^{p-1}} = b_{T^{p-1}}$ corresponding to the nodes in T_i^{p-1} we obtain

$$x^{*\,p-1}_{ij} \quad = \sum_{r \in T_i^{p-1}} b_r - \delta^*_{T^{p-1}} |T_i^{p-1}|.$$

Since $\delta^C_{T_i^{p-1}} > \delta^*_{T^{p-1}}$ from Theorem 4.1.3 we have $x^{*\,p-1}_{ij} > 0$, and the convex combination solution $x_{ij}^{Cp} = (1-\lambda)x_{ij}^{Cp} + \lambda x^{*\,p}_{ij} > 0$. ∎

Corollary 4.1.4 can be used to examine just the reverse arcs of T^k during a minor iteration. If none of the reverse arcs have negative flow we can compute the residual vector quickly using Theorem 4.1.1 and pick the next arc to enter the forest in the major iteration of the NNLS algorithm.

During a minor iteration some reverse arcs whose flow value goes to zero during the convex combination are dropped from the basis. This can lead to some subtrees being separated from T as shown in Figure 12. In the following theorem we show that the subtrees that separate from tree T during the convex combination will have a positive least-squares flow.

Theorem 4.1.6 *Let $\bar{T}$ be a subtree of T_i^k that separates from $T^K = T_i^k \cup T_j^k \cup (i,j)$ when arcs are dropped from T^k during the convex combination procedure. Let $x^C_{\bar{T}}$ denote the components of $x^C_{T_i^k}$ corresponding to arcs in $\bar{T}$. Then $x^C_{\bar{T}}$ is a solution of $min||b_{\bar{T}} - \bar{T}x_{\bar{T}}||^2$, which has all positive components.*

Proof Let (u,v) be the arc of T^k that dropped when $\bar{T}$ separated from T^k during the $(k-1)^{th}$ minor iteration. Since $\bar{T}$ is a single tree it has all positive flows in $x^C_{\bar{T}}$. If we add all the rows of the equations $T_i^k x^C_{T_i^k} + \rho^C_{T_i^k} = b_{T_i^k}$ corresponding to the nodes in $\bar{T}$ we obtain

$$-x_{uv} + \delta_{T_i^k} \cdot |\bar{T}| = \sum_{r \in \bar{T}} b_r.$$

Since $x_{uv} = 0$, this implies that

$$\rho^C_{T_i^k} = \frac{\sum_{r \in \bar{T}} b_r}{|\bar{T}|}.$$

By Theorem 4.1.1, this is the residual corresponding to the solution of the problem $\min||b_{\bar{T}} - \bar{T}x_{\bar{T}}||^2$. From the converse we know than only the least-squares solution has these properties. Hence it follows that $x^C_{\bar{T}}$ is in fact the least-squares flow. ∎

This theorem implies that when we compute $x^*_{T^k}$ we only need to solve the least-squares problem corresponding to the subtree of T^k containing the entering arc (i, j).

In Figure 12 as reverse arcs (c, d) and (g, h) drop out of the tree T^k the subtrees T^k_d and T^k_g separate from T^k. The separated trees T^k_d and T^k_g will have positive least-squares flows, $x^C_{T_g}$ $x^C_{T_d}$, given by the convex combination procedure.

Figure 12: Subtrees $\mathbf{T}_d$ and $\mathbf{T}_g$ Separated From the Main Tree

Some popular tree data structures for manipulating trees can be used for implementing the results in Theorems and Corollaries 4.1.1 - 4.1.6 to streamline and adapt the LSPD algorithm for solving min-cost flow problems quickly. For our computational purpose we used the thread data structure to update the tree when an arc enters the basis or is dropped from it. We refer to Chapter 4 of [6] for details on this tree data structure. Using Theorem 4.1.1 we can compute the residual, $\rho^*_T = b - Tx^*_T$, which is used to select an entering arc, in constant time. This is because whenever an arc (i, j) joins two trees T_i and T_j , the common residual value δ_T can be computed as $\frac{\sum_{k \in T} b_k}{|T|}$ since $\sum_{k \in T} b_k$ is stored in the data structure. Corollary 4.1.4 can be used to restrict all computations to reverse arcs during the execution of the minor iteration of the NNLS algorithm. Theorem 4.1.6 can be used to restrict all computations in an iteration of the LSNF algorithm to the tree that contains the entering arc (i, j) and no computations are required for trees that separate from the tree containing the entering arc. For large problems with many arcs Corollary 4.1.4 and Theorems 4.1.6 can

have considerable influence on the computational time of the LSNF algorithm, since in such cases there are likely to be many forward arcs and trees that separate can be quite large. Finally the block matrix structure of the basis makes the algorithm especially attractive by allowing all computations in an iteration of the LSNF algorithm to be done on joining the two blocks corresponding to the two trees connected by the entering arc.

In the next sub-chapter, some variations of the basic algorithm will be explained that make LSNF more general and more efficient.

4.2 Variations

In this chapter, two algorithmic variations from the basic algorithm described so far are presented. The first variation is a version of the algorithm that deals with upper bounds on variables. The second is about the way pricing is done.

4.2.1 Least-Squares Network Flow Algorithm with Upper Bounds

Even though all the theory for LSNF is developed here without upper bounds, it is easily extended to the upper-bounded case.

A minimum cost network flow problem with upper bounds is formulated as follows:

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Nx = b \\ & 0 \le x \le u \ . \end{array}$$

From the dual of the problem and the complementary slackness condition, implicit dual feasibility can be derived as follows:

$$\begin{cases} c_j - \pi A_j \ge 0 & \text{if } x_j = 0 \\ c_j - \pi A_j = 0 & \text{if } 0 < x_j < u_j \\ c_j - \pi A_j \le 0 & \text{if } x_j = u_j \end{cases} \quad . \tag{20}$$

In the upper-bounded case, a dual feasible point can be easily obtained. Any real vector π in $\mathbb{R}^p$ can serve as dual feasible point by choosing the corresponding primal flow x appropriately. First calculate $c_j - \pi A_j$ and set x_j based on the sign of the $c_j - \pi A_j$. That is, $x_j = 0$ when $c_j - \pi A_j > 0$, $x_j = u_j$ when $c_j - \pi A_j < 0$. In our experiments we have always chosen $c > 0$. Thus for $\pi = 0$ the equality $\pi A_j = c_j$ never holds. Thus the set E is initially empty, and we take $\rho^* = b$ as the improving direction for π.

The improvement in the dual variable $\pi = \pi + t\rho^*$ is computed based on the dual feasibility condition defined above in (20). The dual feasible point π can move along the direction of ρ^*, which is the residual of bounded least-squares problem (explained in the next paragraph), as long as the sign of $c_j - \pi' A_j$ does not change. If any of the $c_j - \pi' A_j$ become zero for a variable which has non-zero $c_j - \pi A_j$ the dual movement is stopped and t fixed. So t^* can be computed as

$$t^* = min\Big\{\frac{c_j - \pi A_j}{\rho^* A_j} : (i,j) \in \mathbb{E}, \rho^* A_j \neq 0, \text{ and } \frac{c_j - \pi A_j}{\rho^* A_j} > 0\Big\}.$$

Given the update of π the new subproblem is defined as follows:

$$\begin{aligned} \min \quad & ||b - Ex_E||^2 \\ & 0 \leq x_E \leq u. \end{aligned}$$

This is a bounded least-squares problem(BLS) and can be solved in a manner similar to the way the NNLS problem is solved. A feasible basis B for BLS is defined as a B for which the solution x_B^* of the unconstrained least-squares problem $\min ||b - Bx_B^*||^2$ satisfies $0 < x_B^* < u_j$. Each major iteration starts with a feasible basis, a collection of trees, and chooses an arc (column) to connect two trees. Any arc (column) A_j which has $\rho^{*T} A_j > 0$ with x_j at its lower bound or $\rho^{*T} A_j < 0$ with x_j at its upper bound can be selected. The entering column (arc) is appended to the basis B to form a basis $\bar{B} = [B, A_s]$. After solving the least-squares problem $\min ||b - \bar{B} x_{\bar{B}}||^2$, we perform the convex combination procedure if some components of $x_{\bar{B}}^*$ violates its bound constraints, which are

$$0 \leq x_{\bar{B}}(\lambda) = (1 - \lambda) x_B^* + \lambda x_{\bar{B}}^* \leq u_j. \tag{21}$$

In the convex combination procedure λ is chosen as large as possible according to condition (21). Because of the upper bound restriction, it is possible that entering column (arc), A_s, can be dropped from the basis at the same major iteration with the corresponding variable x_s going from one bound to the other.

4.2.2 Tree-Wise Pricing Strategy

In our earlier experiments we observed that the pricing operation took approximately 90% of the computational time. Therefore we developed an alternative pricing strategy that exploits the tree-wise decomposability of the basis matrix. With this pricing strategy we obtained 10-20 times better results than with the general pricing strategy.

In our strategy we only update the components of π corresponding to some of the new trees in the optimal basis for the equality set E. We have not been able to mathematically analyze this strategy with that of updating all components of π, but a from practical point of view, the new strategy performs very well, and can be implemented very efficiently. We will give a brief description of the new strategy.

Suppose there are r trees, $T_1, T_2, \ldots, T_r$, for which the components of ρ^* have changed from the previous iteration. Let $\mu^*_{T_i}$ be the vector with the same components as ρ^* in the positions corresponding to nodes in T_i, and zeros in the remaining positions, $i = 1, \ldots, r$. Then our update formula can be written as

$$\pi' = \pi + \sum_{i=1}^{r} t_i \mu^*_{T_i}, \tag{22}$$

and each t_i value is chosen to maintain dual feasibility. Observe that only components of π in positions corresponding to nodes in the trees $T_1, \ldots, T_r$ have been updated. There are many ways compute the update (22). In our implementation we sequentially update π one tree at a time, and compute π' iteratively by the formula

$$\pi_{i+1} = \pi_i + t_i \mu^*_{T_i}, \forall i = 1, \ldots, r$$

with $\pi_1 = \pi$ and $\pi_{r+1} = \pi'$. The scalar t_i is computed by the formula

$$t_i = \min\{\frac{c_j - \pi A_j}{{\mu^*_{T_i}}^T A_j} : \frac{c_j - \pi A_j}{{\mu^*_{T_i}}^T A_j} > 0, {\mu^*_{T_i}}^T A_j \neq 0\}. \tag{23}$$

Recall the components of ρ^* corresponding to nodes in a tree have the same value. Thus if A_j corresponds to an arc with both endpoints in T_i, then $\mu^*_{T_i} A_j = 0$. Similarly, if A_j corresponds to an arc with both endpoints outside T_i, then $\mu^*_{T_i} A_j = 0$ by definition of $\mu^*_{T_i}$. Thus in computing t_i we need only consider these arcs in the cut set for T_i. Our data structure for storing trees allows this calculation to be done very efficiently. The equality set E is updated each time a value of t_i is computed in (23). In practice we find that the integer r (the size of tree updated) is small, so π and E set can be updated quickly. It is easy to show that the update (22) has $\pi' b > \pi b$.

In the next sub-chapter we discuss our computational results.

4.3 Computational Results

We compared the LSNF algorithm with the simplex-based algorithms in ILOG CPLEX 9.020. Since the Primal Simplex solver and Barrier solver are consistently much slower than

the Network solver and Dual Simplex solver we do not include results of experiments with the Primal Simplex solver and Barrier solver. We turned off the CPLEX preprocessor to ensure a fair comparison between algorithms. LSNF was implemented in the C programming language. All tests were performed using one processor on a Sun 900MHz 4-processor Ultrasparc-IIICu with 16GB RAM running on Solaris 9.

We obtained our test instances from the standard sources. Assignment instances are from the OR-Library[4] and the minimum cost network flow instances, generated by the program NETGEN[22], can be found in the MP-Test Data Repository[32]. Table 6 describes the types of the instances used for the comparison.

Table 6: Instances: Network Flow

	description
assign	complete assignment problem (bipartite graph)
assignp	non-complete assignment problem (bipartite graph)
big	capacitated min-cost network flow
cap	capacitated min-cost network flow
stndrd	capacitated min-cost network flow
transp	transportation problem

Table 7 shows the size of the instances and the number of nodes that have supply/demand. Also in Table 7 we compare the execution times in CPU seconds for all types of instances in Table 6. The execution time does not include the time for CPLEX or our code to read the instance file and assign memory space, so the execution time means only the computational time to get the optimal solution. The best results out of three algorithms (CPLEX Network, CPLEX Dual Simplex, and LSNF algorithm) are written in bold. Since all three algorithms obtained the optimal solution, it is not necessary to report solution values.

Each type of the problem in Table 6 shows very different behavior. For the complete assignment problems LSNF runs faster than CPLEX. But for the non-complete assignment problems and transportation problems, CPLEX Network solver is faster than LSNF, which in turn is faster than CPLEX Dual solver. One possible analysis of these results is that complete assignment problems are more degenerate than non-complete assignment and transportation problems, so LSNF, which is impervious to degeneracy, solves them faster.

Min-cost network flow problem comparisons show different behavior. The LSNF results

show better or equal performance for 37 out of 59 instances compared to CPLEX.

Table 8 has detailed information about the number of iterations reported by the CPLEX Network solver, the Dual Simplex solver, and the LSNF algorithm (the number of major iteration, the number of minor iterations, and the number of least-squares applications in NNLS) and the percentage of the time spent for pricing. It also shows the average tree size during LSNF execution.

The number of iterations of LSNF in Table 8 shows the superior performance of the pricing algorithm we developed. The extremely degenerate instances, big complete assignment instances, require only two LSNF master iterations.

The percentage of time spent during pricing (an average of 58.4%) means the NNLS algorithm (tree structure) has similar importance to the pricing step. Since NNLS has combinatorial structure similar to network simplex, it is insightful to analyze NNLS's behavior. In the NNLS algorithm an arc enters the basis to form a relatively small tree by connecting two trees in the forest. In the network simplex algorithm an arc enters the basis to form a cycle. The length of the cycle formed in an iteration of the network simplex algorithm can be compared to the number of arcs in the new tree formed in the LSNF algorithm. The column **LSNF minor iter.** in Table 8 shows the total number of minor iterations and the total number of least-squares problems encountered by the LSNF algorithm in solving the test problems. On average 1.2 minor iterations (LS) were executed during the execution of the NNLS step in the LSNF algorithm. The transportation instances have a higher average of minor iterations (1.45) than the others since they have bigger supply/demand compared to capacity. The assignment problems have fewer minor iterations (1.06) since their supplies/demands are all one and there is plenty of arc capacity to carry the flow.

We were also interested in measuring the average size of the tree in the unconstrained least-squares problems that are solved during the execution of the LSNF algorithm. Table 8 shows the average tree size for each of the test instances solved by the LSNF algorithm under **Avg. Tree Size**. The average tree size can be compared to the average size of the cycle formed during an iteration of the network simplex algorithm. Hence, the product of the average tree size and the number of iterations of the minor iteration gives an estimate of the amount of work involved in the execution of the NNLS algorithm in LSNF.

We note that all of the LSNF results shown in Table 7 and Table 8 use our specialized tree pricing algorithm. Without the specialized tree pricing, LSNF would not be competitive

with CPLEX.

Table 7: CPU Time

	$\frac{Supply}{Demand}$	**row**	**column**	**CPLEX Network**	**CPLEX Dual**	**LSNF**
assign100	200	200	10000	**0.01**	0.04	0.03
assign200	400	400	40000	**0.08**	0.34	0.10
assign300	600	600	90000	0.24	0.94	**0.07**
assign400	800	800	160000	0.42	2.12	**0.09**
assign500	1000	1000	250000	1.10	4.61	**0.22**
assign600	1200	1200	360000	2.35	4.67	**0.35**
assign700	1600	1400	490000	3.58	13.70	**0.60**
assign800	1600	1600	640000	5.27	17.08	**0.79**
assignp800	1600	1600	100058	**0.36**	1.04	0.44
assignp1500	3000	3000	99845	0.79	1.26	**0.65**
assignp3000	6000	6000	99630	**1.12**	1.47	1.37
assignp5000	10000	10000	99970	**1.91**	2.19	2.03
big1	20	25000	120000	1.90	1.24	**0.60**
big2	1000	20000	140001	15.22	5.74	**4.35**
big3	1879	2000	170000	**0.94**	3.02	1.05
big4	2000	5000	100108	2.64	1.86	**1.13**
big6	1890	5000	60092	2.00	1.05	**0.89**
big7	1894	5000	40105	1.48	0.78	**0.74**
cap1	30	1000	10000	0.04	0.06	**0.02**
cap2	30	1000	30000	**0.06**	0.15	0.08
cap3	30	1000	40000	0.08	0.19	**0.07**
cap4	100	5000	30000	**0.22**	0.28	**0.22**
cap5	100	5000	40000	**0.30**	0.45	0.38
cap6	100	5000	50000	0.32	0.37	**0.29**
cap7	100	5000	60000	0.43	0.63	**0.27**
cap8	200	10000	40000	0.57	0.70	**0.50**
cap9	200	10000	50000	**0.52**	0.93	0.72
cap10	200	10000	70000	0.98	1.25	**0.94**
cap11	200	10000	80000	0.86	1.32	**0.73**
cap12	200	10000	90000	1.03	1.08	**0.85**
cap13	30	1000	10000	**0.06**	0.09	0.11
cap14	30	1000	30000	**0.13**	0.22	0.20
cap15	30	1000	40000	**0.14**	0.27	0.27
cap16	100	5000	30000	0.52	0.39	**0.38**
cap17	100	5000	40000	0.62	0.61	**0.49**
cap18	100	5000	50000	0.73	**0.55**	0.59
cap19	100	5000	60000	0.78	0.70	**0.60**
cap20	200	10000	40000	1.39	1.04	**1.01**
cap21	200	10000	50000	1.44	**0.96**	1.05

Continued on next page

Table 7 (continued)

	$\frac{Supply}{Demand}$	row	column	**CPLEX Network**	**CPLEX Dual**	**LSNF**
cap22	200	10000	60000	1.53	1.27	**1.02**
cap23	200	10000	70000	1.78	1.15	**1.07**
cap24	200	10000	80000	2.03	1.46	**1.36**
cap25	200	10000	90000	2.19	1.50	**1.43**
cap26	30	1000	10000	0.05	0.06	**0.04**
cap27	30	1000	30000	**0.11**	0.18	0.17
cap28	30	1000	40000	**0.12**	0.25	0.22
cap29	100	5000	30000	**0.26**	0.40	0.31
cap30	100	5000	40000	**0.34**	0.41	0.41
cap31	100	5000	50000	0.48	0.55	**0.38**
cap32	100	5000	60000	0.50	0.54	**0.37**
cap33	200	10000	40000	**0.68**	0.84	1.00
cap34	200	10000	50000	1.01	1.09	**0.91**
cap35	200	10000	60000	1.06	1.28	**0.87**
cap36	200	10000	70000	1.26	1.22	**0.89**
cap37	200	10000	80000	1.28	1.25	**0.97**
cap38	200	10000	90000	1.28	1.36	**0.85**
cap39	400	10000	100000	2.06	1.57	**1.30**
cap40	400	10000	120000	2.51	2.18	**1.55**
cap41	400	10000	140000	2.28	1.87	**1.40**
stndrd36	1200	8000	15000	**0.68**	1.00	1.40
stndrd37	950	5000	23000	0.64	**0.57**	0.63
stndrd38	625	3000	35000	0.43	0.53	**0.41**
stndrd39	880	5000	15000	0.50	**0.43**	0.59
stndrd40	400	3000	23000	**0.27**	0.29	**0.27**
stndrd41	2000	2000	10000	**0.09**	0.13	0.21
stndrd42	10000	10000	30000	**0.50**	1.33	1.37
stndrd43	2	4000	20000	0.07	0.07	**0.05**
stndrd45	2	4000	20000	0.05	0.06	**0.01**
stndrd46	7437	8000	35561	**1.23**	1.84	2.11
stndrd47	7460	8000	35539	1.23	1.76	**1.18**
stndrd48	2560	3000	15441	**0.20**	0.39	0.27
transp1	800	800	10028	**0.04**	0.20	0.10
transp2	800	800	20000	**0.06**	0.37	0.16
transp3	800	800	30000	**0.10**	0.51	0.30
transp4	800	800	40002	**0.12**	0.41	0.24
transp5	1000	1000	20049	**0.08**	0.51	0.20
transp6	1000	1000	30049	**0.11**	0.38	0.25
transp7	1000	1000	40025	**0.16**	0.43	0.33
transp8	1000	1000	50055	**0.17**	0.61	0.41
transp9	400	400	10000	**0.02**	0.07	0.07
transp10	400	400	20000	**0.04**	0.20	0.11
transp11	600	600	10020	**0.03**	0.15	0.08

Continued on next page

Table 7 (continued)

	$\frac{Supply}{Demand}$	row	column	**CPLEX Network**	**CPLEX Dual**	**LSNF**
transp12	600	600	20000	**0.05**	0.19	0.13
transp13	600	600	30000	**0.07**	0.39	0.22
transp14	600	600	40000	**0.09**	0.62	0.22

Table 8: Execution Details: LSNF

	CPLEX Network iter.	**CPLEX Dual iter.**	**LSNF major iter.**	**LSNF minor iter.**	**LS iter.**	**ave. tree size**	**percent in pricing**
assign100	1367	214	14	405	426	9.0	75.0
assign200	6134	559	19	1098	1164	24.2	88.1
assign300	17150	925	3	2203	2313	67.8	30.7
assign400	15108	1217	5	2563	2733	16.2	71.3
assign500	26351	1556	2	3827	4149	106.4	21.0
assign600	37776	1751	2	4964	5409	123.4	20.9
assign700	51031	2075	2	6136	6740	125.7	28.4
assign800	62354	2291	2	7660	8453	133.8	16.3
assignp800	16679	1897	43	4298	4481	29.1	78.9
assignp1500	50635	3427	62	7008	7313	37.2	75.2
assignp3000	67164	6448	114	13990	14437	71.6	48.3
assignp5000	92385	10939	184	21755	22636	34.2	53.1
big1	70608	1590	430	2091	2114	90.5	61.6
big2	209391	11747	405	19378	24252	64.0	43.7
big3	22437	2801	70	3651	5279	40.6	87.1
big4	54101	5202	131	7348	10060	68.6	58.1
big6	52817	5147	178	7334	10024	49.6	56.8
big7	42573	5083	190	7269	9867	48.5	45.1
cap1	3005	362	85	445	500	20.9	50.0
cap2	4677	307	124	460	505	26.4	92.8
cap3	6094	342	96	478	526	23.2	76.9
cap4	12970	1408	226	2045	2207	45.4	65.7
cap5	17967	1519	379	2295	2473	44.3	63.3
cap6	19066	1367	223	2119	2280	46.7	70.4
cap7	25121	1362	185	2041	2193	48.7	79.8
cap8	29596	2906	355	4154	4468	54.7	44.8
cap9	28083	2922	406	4592	4964	81.9	41.9
cap10	47139	2976	382	4706	5024	112.2	53.2
cap11	43200	3032	344	4512	4875	70.7	53.8
cap12	49590	2854	375	4559	4888	72.7	54.5
cap13	4898	741	204	1451	2000	28.0	61.0
cap14	10772	801	209	1510	2093	32.4	88.9

Continued on next page

Table 8 (continued)

	CPLEX Network iter.	CPLEX Dual iter.	LSNF major iter.	LSNF minor iter.	LS iter.	ave. tree size	percent in pricing
cap15	11164	776	197	1378	1949	24.0	80.1
cap16	27004	2408	299	3885	4761	44.3	41.7
cap17	30201	2425	315	3954	4702	48.2	50.5
cap18	34487	2422	336	3723	4560	44.2	70.5
cap19	39682	2390	347	4169	5096	44.1	52.0
cap20	48459	4599	412	7905	9190	80.0	33.3
cap21	53790	4172	464	6799	8021	73.7	41.2
cap22	58150	4219	461	6696	7968	49.0	54.3
cap23	68354	4204	359	7251	8504	65.3	57.5
cap24	77047	4553	445	7530	8982	72.4	56.7
cap25	82022	4336	475	7371	8671	91.8	47.1
cap26	4080	510	127	811	1021	19.6	50.0
cap27	8176	612	167	1121	1387	30.1	90.4
cap28	10183	664	178	1099	1396	30.0	73.7
cap29	16496	1737	308	2606	2931	53.6	59.5
cap30	20593	1833	334	2869	3256	50.8	56.7
cap31	27527	1883	258	2974	3339	53.1	61.0
cap32	31465	1659	243	2600	2929	61.6	73.7
cap33	33480	3555	526	5754	6338	104.3	40.5
cap34	43382	3488	404	5775	6383	104.8	45.3
cap35	49246	3416	427	5503	6134	80.3	44.2
cap36	58653	3392	382	5409	5983	74.9	51.4
cap37	57058	3641	421	5625	6258	55.5	55.0
cap38	59422	3352	348	5013	5572	61.1	59.8
cap39	59193	3899	332	6300	7064	82.0	57.5
cap40	65783	3972	366	6372	7117	84.4	62.7
cap41	65984	3676	318	5463	6128	66.2	63.2
stndrd36	19430	6292	402	10052	11986	102.6	16.4
stndrd37	22778	3814	248	6141	7560	57.6	37.8
stndrd38	19306	2369	186	3631	4646	39.6	53.0
stndrd39	18204	3729	257	5929	7360	68.8	31.5
stndrd40	14057	1931	168	3077	3710	46.3	51.2
stndrd41	6993	2145	187	4207	4365	24.8	62.8
stndrd42	31387	10860	292	20791	21542	32.5	43.7
stndrd43	5994	513	133	890	1019	75.0	33.3
stndrd45	3653	51	133	132	132	48.0	66.6
stndrd46	23866	10655	148	15129	21857	150.8	13.4
stndrd47	23360	10495	108	13547	19260	72.6	20.9
stndrd48	7951	3739	89	4952	7086	40.1	29.5
transp1	2756	1142	108	1443	2087	27.4	61.0
transp2	3929	1153	105	1475	2151	24.6	76.2
transp3	5750	1109	154	1465	2128	30.6	71.4

Continued on next page

Table 8 (continued)

	CPLEX Network iter.	CPLEX Dual iter.	LSNF major iter.	LSNF minor iter.	LS iter.	ave. tree size	percent in pricing
transp4	6945	1138	78	1435	2069	25.4	79.2
transp5	4873	1478	107	1863	2705	26.8	93.0
transp6	6398	1410	121	1833	2655	29.2	93.0
transp7	8499	1423	136	1872	2719	24.6	86.6
transp8	9566	1477	110	1942	2859	29.1	74.6
transp9	1560	518	69	688	977	26.2	93.3
transp10	2773	557	60	767	1135	19.1	67.6
transp11	2256	829	70	1065	1531	22.8	83.0
transp12	3573	819	78	1124	1648	22.6	78.6
transp13	4651	859	92	1119	1638	23.5	85.5
transp14	5723	892	73	1145	1691	19.9	71.4

4.4 Concluding Remarks

In this chapter we have discussed a least-squares approach to solving network flow problems. We proved some interesting least-squares properties of network flow problems and showed how these properties can be used to make the LSPD algorithm run quickly for solving minimum cost flow problems. Very good computational results were obtained with the LSNF algorithm. The performance was comparable to CPLEX for all the test instances.

CHAPTER V

COMBINED OBJECTIVE LEAST-SQUARES ALGORITHM

The least-squares primal-dual algorithm (LSPD) is dual ascent algorithm using least-squares subproblem. Combined-objective least-squares (COLS) is primal version of LSPD that also uses least-squares measures that are solved by QR-factorization.

We want to solve the standard linear programming problem

$$\begin{aligned} \min \quad & cx & \\ \text{s.t.} \quad & Ax & = b \\ & x & \geq 0, \end{aligned} \qquad (\mathbf{P})$$

and its dual

$$\begin{aligned} \max \quad & \pi b & \\ \text{s.t.} \quad & \pi A & \leq c^T, \end{aligned} \qquad (\mathbf{D})$$

where c, x are column and row vectors in $\mathbb{R}^n$, b and π are column and row vectors in $\mathbb{R}^m$, and A is a matrix of size $m \times n$.

The COLS algorithm we develop is to solve

$$\begin{aligned} \min \quad & cx + \tfrac{M}{2}\rho^T\rho & \\ \text{s.t.} \quad & Ax + \rho & = b \\ & x & \geq 0 \end{aligned} \qquad (\mathbf{PM})$$

or, equivalently

$$\begin{aligned} \min \quad & cx + \tfrac{M}{2}||b - Ax||^2 & \\ & x & \geq 0 \end{aligned}$$

and its dual

$$\begin{aligned} \max \quad & \pi b - \tfrac{1}{2M}\pi^T\pi & \\ \text{s.t.} \quad & \pi^T A & \leq c. \end{aligned} \qquad (\mathbf{DM})$$

Let the optimal solution of (**P**) and (**D**) be x_0^* and π_0^*, respectively, and let the optimal solution of (**PM**) and (**DM**) be x^* and π^*, respectively. $\rho^* = b - Ax^*$. From the KKT optimality conditions we can derive $\pi^* = M\rho^{*T}$.

As M increases, the objective values of (**D**) and (**DM**) become closer; equivalently the objective values of (**P**) and (**PM**) become closer. This can be proved by the following arguments:

$$\begin{aligned}
&\pi_0^* b - \frac{1}{M}\pi_0^{*T}\pi_0^* \\
\leq&\pi^* b - \frac{1}{M}\pi^{*T}\pi^*, \quad \text{by the optimality of } \pi^* \text{ to } (\mathbf{DM}) \\
\leq&\pi^* b \\
\leq&\pi_0^* b, \quad \text{by the optimality of } \pi_0^* \text{ to } (\mathbf{D}).
\end{aligned} \tag{24}$$

As M increases enough all inequalities become nearly equal. We have that the objective value of (**D**) ($\pi^* b - \frac{1}{M}\pi^{*T}\pi^*$) and (**DM**) ($\pi_0^* b$) become close enough. In this chapter we describe the combined-objective least-squares algorithm to solve (**PM**) with large enough M.

In the LSPD algorithm each subproblem solves

$$\min ||b - Bx_B||^2,\ x_B \geq 0, \tag{25}$$

where B is subset of columns of A and x_B is the components of x corresponding to B, to find a dual improving direction and primal feasibility. Its focus is on attaining primal feasibility. But in the COLS algorithm each subproblem solves

$$\min c_B x_B + \frac{M}{2}||b - Bx_B||^2,\ x_B \geq 0 \tag{26}$$

to achieve primal feasibility and optimality.

We describe the COLS algorithm in Chapter 5.1. The computational results and a variation to achieve a better convergence rate are described in Chapter 5.2. The conclusion is in Chapter 5.3.

5.1 Algorithm

The COLS algorithm solves (**PM**) by using a sequence of problems

$$\min c_B x_B + \frac{M}{2}||b - Bx_B||^2,\ x_B \geq 0 \tag{27}$$

(where B is a matrix of k columns of A and x_B and c_B are column and row vectors in $\mathbb{R}^k$ corresponding to B), to find the basis which has the best objective value. A feasible basis B is the basis whose optimal solution x_B^* of (27) is strictly positive. B might contain a different number of columns than a basis for (**P**) because we use a column-dropping procedure. The columns to be dropped can be found by solving a sequence of unconstrained optimization problems

$$\min c_B x_B + \frac{M}{2}||b - Bx_B||^2 \tag{28}$$

and using convex combination processes, which will be explained later in this section.

The optimal solution of (28) can be obtained simply, since (28) is an unconstrained convex quadratic problem, by solving equations

$$c_B^T - M \cdot B^T(b - Bx_B^*) = 0. \tag{29}$$

This can be solved easily if we have a QR-factorization of B, such that $B = QR$, where Q is an $m \times m$ orthonormal matrix and R is an $m \times k$ upper triangular matrix. We can write (29) as

$$c_B - M \cdot R^T(Q^T b - Rx_B^*) = 0, \tag{30}$$

and it can be solved in two steps: $c_B^T = R^T u$ and $Rx_B^* = Q^T b - u/M$ since R is upper triangular.

To obtain a QR-factorization of B we use Givens rotations which have an advantage in sparsity issues over other methods (see Chapter 2.2) and have good numerical behavior.

Q^T is stored in a product form since it is more difficult to update if stored in explicit form. When we store Q^T as a product form, it is very simple to append information about new Givens rotations to the previously-stored Q^T. We use the same techniques as in Chapter 2.2 for updating the R matrix to maintain its upper-triangular form.

The data structures for R and Q^T, numerical stability, and sparsity are treated in Chapter 2.2 and 2.3 since the implementation of COLS and LSPD share the same QR-factorization routine. The result related to the sparsity of the COLS algorithm will be explained in Chapter 5.2.

The optimality condition of (**PM**) can be written as

$$\pi^* A \leq c^T \qquad (\textbf{KKT-}\text{a})$$

$$\pi^* = M\rho^{*T} \qquad (\textbf{KKT-}\text{b})$$

$$Ax^* + \rho^* = b \qquad (\textbf{KKT-}\text{c})$$

$$x \geq 0. \qquad (\textbf{KKT-}\text{d})$$

These are very useful in analyzing optimality. Suppose we have a feasible basis B and corresponding solution x_B^* ($x^* = [x_B^*, x_{A\backslash B} = 0]$). It automatically satisfies (**KKT-**d) and (**KKT-**c). From (**KKT-**b) we can get a dual feasible point π^*. If π^* satisfies (**KKT-**a), then x^* and π^* are optimal solutions for (**PM**) and (**DM**). Suppose (**KKT-**a) is not satisfied because of column A_s ($c_s - \pi A_s < 0$, which means $\frac{c_s}{M} - \rho^{*T} A_s < 0$). Then we append A_s to B to construct $[B, A_s]$. Let $z_B(u) = c_B u + \frac{M}{2}||b - Bu||^2$. Then the following theorem proves that we improve the solution by appending A_s to B.

Theorem 5.1.1 *$z_{[B,A_s]}(x^*_{[B,A_s]}) < z_B(x_B^*)$ and the last component of $x_{\bar{B}}$ is positive.*

Proof:

$$\begin{aligned} z_{[B,A_s]}([x_B^*, v]) =& c_B x_B^* + c_s v + \frac{M}{2}||b - Bx_B^* - A_s v||^2 \\ =& z_B(x_B^*) + \frac{M}{2}(||A_s||^2 v^2 + 2(\frac{c_s}{M} - \rho^{*T} A_s)v). \end{aligned}$$

Since $\frac{c_j}{M} - \rho^{*T} A_j < 0$, we have that $z_{[B,A_s]}(x_B^*, v) < z_B(x_B^*)$ for sufficiently small v. Let $x^*_{[B,A_s]}$ be the optimal solution of $\min ||b - Bx_B - A_s x_s||^2$. Therefore $z_{[B,A_s]}(x^*_{[B,A_s]}) \leq z_{[B,A_s]}(x_B^*, v) < z_B(x_B^*)$. $z_{[B,A_s]}(x_B^*, v)$ is minimized at $v > 0$. Hence $x^*_{[B,A_s]}$ has a positive last component because of the convexity of $z_{[B,A_s]}$. ■

In the simplex algorithm, the entering column is always dependent on columns of the B, so that one of the dependent columns is replaced with the entering column in a basis. But in COLS, the entering column A_s may not be a dependent column or several dependent columns can be replaced with an entering column A_s. When A_s is independent of B, then $\bar{B} = B$ and it is required to solve $\min c_{\bar{B}} x_{\bar{B}} + c_s x_s + \frac{M}{2}||b - \bar{B} x_{\bar{B}} - A_s x_s||^2$. The following proposition and theorem prove that the newly updated basis $\bar{B}$ (after dropping dependent columns) leads to an improved solution.

Proposition 5.1.2 *If $A_s = By$, where y is a nonzero column vector in $\mathbb{R}^k$, then $c_s < c_B y$.*

Proof:

$$c_s/M < \rho^{*T} A_s, \quad \text{(by the condition for an entering column.)}$$
$$= \rho^{*T} By.$$

By the optimality condition (29) ($c_B^T = MB^T\rho^*$), and the above, we have $c_s < c_B y$. ■

The computation of $A_s = By$ can be done efficiently with $Q^T A_s = Ry$ since R is an upper triangular matrix. The following theorem describes how some dependent columns are selected and dropped from the basis $\bar{B}$ to obtain a better basis, and shows how unboundedness is detected by COLS.

Theorem 5.1.3 *Let $\tilde{\theta} = \min\{\frac{x_i^*}{y_i} : \frac{x_i^*}{y_i} > 0, y_i \neq 0\}$, where x_i^* is i^{th} component of x_B^* and y is a vector in $\mathbb{R}^k$ satisfying $A_s = By$. Let $S = \{A_i : \frac{x_i^*}{y_i} = \tilde{\theta}\}$. Define $\bar{B}$ as the subset of $[B, A_s]$ such that $\bar{B} = B \setminus S$. $\bar{B}$ satisfies $z_{[\bar{B},A_s]}(x^*_{[\bar{B},A_s]}) < z_B(x_B^*)$, where $x^*_{[\bar{B},A_s]}$ is an optimal solution of $\min c_{\bar{B}} x_{\bar{B}} + \frac{M}{2}||b - \bar{B}x_{\bar{B}} - A_s x_s||^2$. If $S = \emptyset$ then* (**P**) *and* (**PM**) *are unbounded.*

Proof: Consider

$$z_{\bar{B}}(x_B, x_s) = c_B x_B + c_s x_s + \frac{M}{2}||b - Bx_B - A_s x_s||^2.$$

We want to minimize $z_{[B,A_s]}(x_B, x_s)$, where x_s is constant and x_B is variable. The optimality condition of the above can be written as

$$c_B^T = MB^T((b - A_s x_s) - Bx_B) = 0, \tag{31}$$

and

$$x_B(x_s) = x_B^* - (B^T B)^{-1} B^T A_s x_s = x_B^* - y x_s. \tag{32}$$

(32) is the movement from the solution x_B^* when we increase the value of x_s.

Consider the value of $z_{[B,A_s]}$,

$$z_{[B,A_s]}([x_B(x_s), x_s]) = c_B x_B(x_s) + c_s x_s + \frac{M}{2}||b - Bx_B(x_s) - A_s x_s||^2.$$

The third term of $z_{[B,A_s]}([x_B(x_s), x_s])$,

$$||b - Bx_B(x_s) - A_s x_s||^2$$
$$= ||b - B(x_B^* - (B^T B)^{-1} B^T A_s x_s) - A_s x_s||^2$$
$$= ||b - Bx_B^*||^2, \text{ by the dependency of } A_s,$$

does not improve and the first and the second term can be transformed as

$$\begin{aligned}
&c_B x_B(x_s) + c_s x_s \\
&= c_B(x_B^* - (B^T B)^{-1} B^T A_s x_s) + c_s x_s \\
&= c_B x_B^* + (c_s - c_B (B^T B)^{-1} B^T A_s) x_s \\
&= c_B x_B^* + (c_s - c_B y) x_s.
\end{aligned}$$

Then

$$\begin{aligned}
z_{[B,A_s]}([x_B(x_s), x_s]) &= c_B x_B^* + \frac{M}{2}||b - Bx_B^*||^2 + (c_s - c_B y)x_s \\
&= z_B(x_B^*) + (c_s - c_B y)x_s.
\end{aligned}$$

From Proposition 5.1.2, we thus have $x_{[B,A_s]}(x_B(x_s), x_s) < z_B(x_B^*)$ with positive x_s.

Some components in $x_B(x_s)$ become zero when $x_s = \tilde{\theta} > 0$, and they are the columns corresponding to S. Denote this as $[x_B(\tilde{\theta}), \tilde{\theta}]$, which is non-negative. After S is removed from B we have linearly independent columns in $[\bar{B}, A_s]$. Since $\tilde{\theta}$ is positive we have $z_{[B,A_s]}(x_B(\tilde{\theta}), \tilde{\theta}) < z_B(x_B^*)$. Therefore we have $z_{[\bar{B},A_s]}(x^*_{[\bar{B},A_s]}) \leq z_{[\bar{B},A_s]}(x_B(\tilde{\theta}), \tilde{\theta}) < z_B(x_B^*)$.

If S is empty then we can decrease $z_{\bar{B}}$ infinitely, so that (**P**) and (**PM**) are unbounded.

■

After adding column A_s as in Theorem 5.1.3, $[\bar{B}, A_s]$ has linearly independent columns (either dependent columns S are dropped from B or A_s is linearly independent of the columns in B), so that $[\bar{B}, A_s]$ also has linearly independent columns. But we cannot guarantee that $[\bar{B}, A_s]$ is a feasible basis because we haven't solved

$$\min c_{\bar{B}} x_{\bar{B}} + c_s x_s \frac{M}{2} ||b - \bar{B} x_{\bar{B}} - A_s x_s||^2. \tag{33}$$

There are three situations we can face after solving (33).

First, $x^*_{[\bar{B},A_s]}$ could be strictly positive. In this case, no corrective action is necessary; after setting $B := [\bar{B}, A_s]$, the next major iteration begins. Second, $x^*_{[\bar{B},A_s]}$ could be non-negative but not strictly. Then, we drop the zero-value components of $x^*_{[\bar{B},A_s]}$ and update B with the remaining columns in $x^*_{[\bar{B},A_s]}$. $B := [\bar{B}, A_s]$ and the next major iteration starts. Third, $x^*_{[\bar{B},A_s]}$ could have one or more negative components. Then we need to repeat the following procedure until we have a feasible basis $[\bar{B}, A_s]$, which is a subset of the original $[B, A_s]$.

1. $x^*_{B_0} = [x_B(\tilde{\theta}), \tilde{\theta}]$

2. $B_0 = \bar{B}$

3. $x^*_{B_i}$ is constructed by dropping zero component in $x_{\bar{B}_i}(\lambda^*_i)$, $i = 1, 2, \cdots$.

4. $[\bar{B}_i, A_s]$ is constructed by the columns corresponding to $x^*_{B_{i-1}}$.

5. $[x^*_{\bar{B}_i}, x^*_{s_i}]$ is the solution of $c_{\bar{B}_i} x_{\bar{B}_i} + A_s x_s + \frac{M}{2}||b - \bar{B}_i x_{\bar{B}_i} - A_s x_s||^2$. (Suppose $x^*_{\bar{B}_i}$ has at least one negative component.)

6. Construct $x_{\bar{B}_i}(\lambda_i) = (1 - \lambda) x^*_{B_{i-1}} + \lambda[x^*_{\bar{B}_i}, x^*_{s_i}]$, and find λ^*_i, the largest λ satisfying $x_{\bar{B}_i}(\lambda_i) \geq 0$.

λ^*_i is always strictly between zero and one. $\lambda^*_i > 0$ since $x^*_{B_i} > 0$. $\lambda^*_i < 1$ since $x^*_{\bar{B}_i}$ has at least one negative component.

To show the finiteness of the COLS algorithm, we use the next theorem. It says that the objective value of the final feasible basis at the end of each major iteration is better than the objective value before adding the entering column to the basis. With this theorem, we have strict improvement at each major and minor iteration so that the same basis is never repeated. Within one major iteration, the last component of $x^*_{B_i}$ increases monotonically. With a finite number of columns we thus have convergence in a finite number of iterations. There are several pieces of notation to be defined to help describe each step of the proof of for the next theorem.

Theorem 5.1.4 *$z_{\bar{B}_{i+1}, A_s}([x^*_{\bar{B}_{i+1}}, x^*_{s_{i+1}}]) < z_{[\bar{B}_i, A_s]}(x^*_{B_i})$ and x^*_{s+1} is bigger than the last component of $x^*_{B_i}$.*

Proof: Let $f_{\bar{B}_i}(\theta) = \min c_{\bar{B}_i} x_{\bar{B}_i} + \frac{M}{2}||b - \bar{B}_i x_{\bar{B}_i} - A_s\theta||^2$ and $f_{\bar{B}_{i+1}}(\theta) = \min c_{\bar{B}_{i+1}} x_{\bar{B}_{i+1}} + \frac{M}{2}||b - \bar{B}_{i+1} x_{\bar{B}_{i+1}} - A_s\theta||^2$. Since $\bar{B}_i \subset \bar{B}_{i+1}$, $f_{\bar{B}_i}(\theta) \leq f_{\bar{B}_{i+1}}(\theta)$ for $0 \leq \theta \leq 1$.

Suppose the last component of $x^*_{B_i}$ is $\bar{\theta}$ and $f_{\bar{B}_{i+1}}(\theta)$ is minimized at $\theta_{i+1} = \bar{\theta} - \delta_1$, where $\delta_1 \geq 0$. We have $f'_{\bar{B}_{i+1}}(\bar{\theta}) \geq 0$ and $f'_{\bar{B}_i}(\bar{\theta}) < 0$ because of strict convexity of f_B and $f_{\bar{B}}$. Then there is a positive δ_2 satisfying $f_{\bar{B}_{i+1}}(\bar{\theta} - \delta_2) > f_{\bar{B}_i}(\bar{\theta} - \delta_2)$, where $0 < \delta_2 < \delta_1$. This contradicts $f_B(\theta) \leq f_{\bar{B}}(\theta)$. Hence $f'_{\bar{B}_{i+1}}(\bar{\theta}) < 0$ by the convexity of $f_{\bar{B}_{i+1}}$. Therefore $z_{\bar{B}_{i+1}, A_s}([x^*_{\bar{B}_{i+1}}, x^*_{s_{i+1}}]) < f_{\bar{B}_{i+1}}(\bar{\theta} + delta) < z_{[\bar{B}_i, A_s]}(x^*_{B_i})$ with sufficiently small positive δ. ■

From this theorem we have that the last component of $\min c_B x_B + c_s x_s + \frac{M}{2}||b - Bx_B - A_s x_s||^2$ never drop and each minor iteration we can improve our objective function value. Then the major iteration starts with $[B, A_s]$ and it ends with feasible basis $[\bar{B}_k, A_s]$ which is feasible basis with improved objective value.

Thus the COLS algorithm finitely converges to the optimal solution or ends with unboundedness. We summarize this algorithm:

Algorithm 3 COLS : Minimize $cx, s.t. Ax = b$, $x \geq 0$

1: $\rho^* := b$, $B := \emptyset$, $x_B^* := []$, and $x_N = 0$.
2: **while** $I := argmax\{i \;: \rho^{*T} A_i > c_i/M, \text{where } A_i \in A\} \neq \emptyset$ **do**
3:   **if** $s \in I$ is dependent of the columns of B **then**
4:     Solve for y in $By = A_s$
5:     $S = A_{argmin\{\frac{x_{Bk}}{y_k} \;|y_k>0\}}$
6:     **if** $S = \emptyset$ **then**
7:       STOP : unbounded
8:     **else**
9:       $B := B \setminus S$
10:     **end if**
11:   **end if**
12:   $B := [B, A_s]$ and $x_B^* := [x_B^*, 0]$.
13:   **while** $x_{\bar{B}}^* := argmin\{c_{\bar{B}} x_{\bar{B}} + \frac{1}{2}||b - Bx_{\bar{B}}||^2\} \leq 0$ **do**
14:     Find the largest λ^* such that $x_{\bar{B}}(\lambda) = (1 - \lambda)x_B^* + \lambda x_{\bar{B}}^* \geq 0$
15:     x_B^* is constructed with the positive components of $x_{\bar{B}}(\lambda^*)$.
16:     $\bar{B}$ is constructed with columns corresponding to x_B^*.
17:   **end while**
18:   $x_B^* := x_{\bar{B}}^*$, $\rho^* := b - Bx_B^*$,and $x^* \leftarrow [x_B^*, x_N = 0]$
19: **end while**
20: x^* is optimal solution for min $cx, s.t. Ax = b$, $x \geq 0$.

5.2 Computational Results

We compared the COLS algorithm with the primal and dual simplex algorithms in ILOG CPLEX 9.020. We turned off the CPLEX preprocessor to ensure a fair comparison between algorithms. The COLS algorithm was implemented in the C++ programming language but without object-oriented features. All tests were performed using one processor of a Sun 900MHz 4-processor Ultrasparc-IIICu with 16GB RAM running Solaris 9.

The size of the problem and the objective values are listed in Table 2 in Chapter 3.3 since we uses the same set-partitioning instances.

Preliminary results showed that over 80% of the time is spent calculating the smallest value of $\frac{c_j}{M} - \pi^T A_j$ to find an entering column for the basis. We tried several criteria to find

an entering column and found that using the first column whose $\frac{c_j}{M} - \pi^T A_j$ value is better (more negative) than the first several negative $\frac{c_j}{M} - \pi^T A_j$ works best. The comparison results are in Table 9, which are tested with $M = 1.0 \times 10^6$. The **COLS quick** column has the results of this pricing scheme. The computational CPU times are decreased drastically but the number of iterations increases a lot. Our new pricing does not take too much time to find an entering column. All the further results in this paper use this technique to find the entering column.

Table 9: CPU Time

	COLS general		**COLS quick**	
Instance	**time**	**iter.**	**time**	**iter.**
sppkl01	0.50	322	0.20	856
sppkl02	3.20	525	0.93	2750
sppnw03	3.13	288	0.62	1151
sppnw04	5.94	255	0.98	1953
sppnw05	26.97	212	3.39	1395
sppnw06	0.22	154	0.09	404
sppnw13	0.84	290	0.28	1001
sppnw14	24.08	533	1.75	2291
sppnw16	15.08	286	1.56	1317
sppnw17	22.27	508	1.94	1843
sppnw18	0.99	413	0.52	721
sppus02	0.92	302	0.29	791
sppus03	11.88	407	0.88	973
sppus04	2.94	478	1.25	865

The CPU times and iterations of the small and the medium instances are in shown in Table 10. This table includes CPLEX Primal and Dual results and the COLS algorithm with $M = 1.0 \times 10^6$ and $M = 1.0 \times 10^8$. Table 10 shows pretty good results for the medium size instances. Most of the instances (11 out of 14) show better results than CPLEX Primal and Dual solvers. In almost all instances, $M = 1.0 \times 10^6$ shows better convergence than $M = 1.0 \times 10^8$.

The COLS algorithm is designed for solving $\min cx + \frac{M}{2}||b - Ax||^2$, $x \geq 0$ not for $\min cx, Ax = b, x \geq 0$. So there is always error related to $||\cdot||^2$. As M grows big, the error decreases since $||\cdot||^2$ is emphasized more. As M becomes smaller the convergence is quicker since cx gets more attention than $||\cdot||^2$ so that we can have better columns in the basis

in the early stage of the algorithm. But $||\cdot||^2$ for small M is significant so that it can be hard to ensure a feasible solution. Large M emphasizes the $||\cdot||^2$ part so that the value of $cx + \frac{M}{2}||b - Ax||^2$ decreases quickly but we have more possibility to have bad columns in early stages. Hence selecting a good M value is critical to balancing good convergence rates and feasible solutions. First, $\frac{c_j}{M}$ should not be too small. If there are a lot of variations in c_j values then the maximum M value must be chosen carefully so that the column with small c_j are not ignored. Ideally, M should be chosen based on the objective value cx^*. When M is too small compared to cx^* (the optimal objective solution value of $\min cx, Ax = b, x \geq 0$) then it creates error in $||\cdot||^2$. When M is too big it converge too slowly. But since we don't have a prediction for cx^* before the algorithm terminates, this criterion is not realistic. In our test cases, all the instances we tried were set-partitioning problems. Most of c_j values are within the same order of magnitude so that it is easy to get a good M value. In our computation we found $1.0 \times 10^6 \leq M \leq 1.0 \times 10^8$ shows good behavior, quick convergence and small error.

Table 10: CPU Time: COLS - small, and medium

	CPLEX Primal		**CPLEX Dual**		**COLS($M = 10^6$)**		**COLS($M = 10^8$)**	
Instance	**time**	**iter.**	**time**	**iter.**	**time**	**iter.**	**time**	**iter.**
sppnw07	0.06	366	0.04	22	0.06	316	0.05	276
sppnw08	0.00	29	0.00	28	0.01	120	0.00	106
sppnw09	0.04	417	0.03	45	0.07	392	0.09	451
sppnw10	0.01	85	0.01	27	0.02	133	0.01	122
sppnw11	0.08	384	0.08	55	0.06	281	0.11	532
sppnw12	0.00	37	0.00	23	0.01	105	0.01	108
sppnw15	0.01	84	0.01	18	0.00	69	0.01	90
sppnw19	0.05	326	0.03	44	0.03	227	0.05	263
sppnw20	0.01	146	0.01	34	0.01	99	0.01	103
sppnw21	0.01	45	0.00	15	0.01	84	0.00	84
sppnw22	0.01	83	0.00	22	0.02	125	0.01	125
sppnw23	0.01	82	0.00	33	0.01	83	0.01	83
sppnw24	0.00	20	0.01	15	0.01	134	0.01	128
sppnw25	0.01	53	0.00	22	0.01	107	0.01	107
sppnw26	0.01	61	0.00	24	0.00	87	0.00	79
sppnw27	0.01	52	0.01	16	0.02	110	0.01	123
sppnw28	0.01	57	0.01	16	0.01	88	0.00	96
sppnw29	0.02	124	0.03	48	0.01	162	0.02	178
sppnw30	0.03	105	0.01	18	0.01	77	0.01	77
sppnw31	0.03	183	0.02	29	0.02	148	0.02	141
sppnw32	0.01	16	0.01	20	0.00	66	0.00	66

Continued on next page

Table 10 (continued)

	CPLEX Primal		CPLEX Dual		COLS($M = 10^6$)		COLS($M = 10^8$)	
Instance	**time**	**iter.**	**time**	**iter.**	**time**	**iter.**	**time**	**iter.**
sppnw33	0.03	120	0.02	20	0.03	217	0.03	217
sppnw34	0.01	100	0.00	22	0.01	113	0.01	117
sppnw35	0.02	81	0.02	18	0.03	166	0.02	196
sppnw36	0.02	159	0.03	51	0.01	145	0.01	145
sppnw37	0.01	87	0.01	19	0.01	66	0.01	66
sppnw38	0.01	48	0.00	33	0.02	158	0.02	167
sppnw39	0.01	60	0.01	12	0.01	81	0.01	71
sppnw40	0.00	56	0.01	18	0.00	50	0.00	50
sppnw41	0.00	19	0.00	13	0.01	41	0.00	41
sppnw42	0.01	97	0.01	29	0.01	87	0.00	65
sppnw43	0.01	71	0.01	25	0.01	145	0.01	123
sppkl01	0.10	505	0.18	151	0.20	856	0.25	975
sppkl02	0.99	3388	0.97	191	0.93	2750	0.84	2442
sppnw03	0.91	1018	0.68	88	0.62	1151	0.70	1449
sppnw04	1.75	799	2.35	136	0.98	1953	0.88	1669
sppnw05	15.17	5077	7.11	132	3.39	1395	3.50	1738
sppnw06	0.14	924	0.13	97	0.09	404	0.09	379
sppnw13	0.19	484	0.20	85	0.28	1001	0.28	990
sppnw14	3.65	1812	2.57	154	1.75	2291	2.09	3020
sppnw16	20.83	20067	31.25	626	1.56	1317	1.67	1357
sppnw17	4.82	1081	3.27	100	1.94	1843	2.24	2948
sppnw18	0.42	2544	0.59	393	0.52	721	0.92	1889
sppus02	0.34	594	0.33	95	0.29	791	0.29	850
sppus03	5.65	1758	2.66	85	0.88	973	1.13	1499
sppus04	1.24	3137	1.00	226	1.25	865	1.51	1411

After obtaining the optimal solution values we found that there were small errors (on the order of $\frac{10.0}{M}$) in ρ^*. To remove this very small error we use a post-processing step where we solve $\min cx + \frac{M}{2}|b - Ax||^2$, $x \geq 0$. This is equivalent to making M infinitely large, ensuring a feasible solution. After one iteration, we always get the optimal solution without changing the basis, showing that COLS had converged to the optimal basis.

Table 11 has detailed results for $M = 1.0 \times 10^6$. **Ratio** and **Density** are the aspect ratio of columns and rows at the final basis before post-processing is executed. Aspect ratios are larger than expected since there are components in x_B^* which have very small values and will be removed during post-processing. Density of the basis varies by the re-factorization. Figure 13 shows typical behavior of the changes as re-factorizations are executed. In early stages of the algorithm, density is relatively high since the entering columns are chosen

mostly for decreasing $||\cdot||^2$ so that the columns with many nonzero elements are generally chosen. In the later stage of the algorithm, columns with fewer nonzero elements are chosen to decrease cx and $||\cdot||^2$ together. Hence the density becomes smaller. It also shows that the re-factorization algorithm works well to achieve a sparser R matrix. The re-factorization

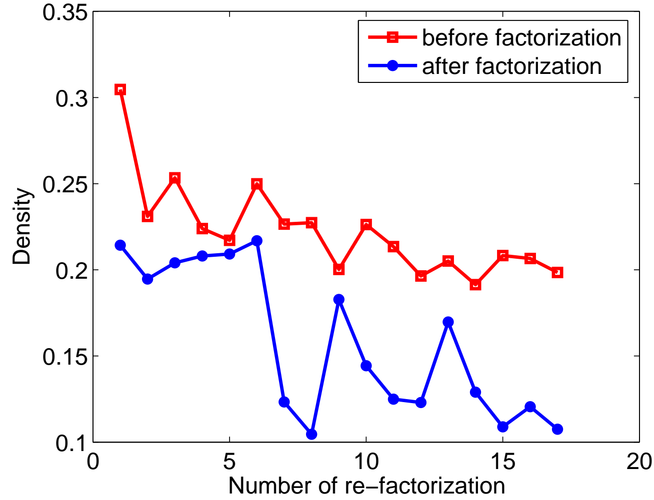

Figure 13: Change of the Density at Each Re-Factorization

times for the COLS algorithm for the medium instances are relatively small compared to the total execution time, as shown in the **ref.** (number of re-factorizations) and **ref. time** columns of Table 11.

Table 11: Execution Details: COLS($M = 1.0 \times 10^6$)

Instance	time	iter.	ratio	density	ref.	ref. time
sppnw07	0.06	316	0.56	0.37	3	0.00
sppnw08	0.01	120	0.83	0.32	0	0.00
sppnw09	0.07	392	0.78	0.30	2	0.00
sppnw10	0.02	133	0.92	0.25	0	0.00
sppnw11	0.06	281	0.90	0.17	2	0.00
sppnw12	0.01	105	0.89	0.15	0	0.00
sppnw15	0.00	69	0.52	0.34	0	0.00
sppnw19	0.03	227	0.55	0.48	1	0.01
sppnw20	0.01	99	0.77	0.46	0	0.00
sppnw21	0.01	84	0.60	0.42	0	0.00
sppnw22	0.02	125	0.65	0.43	0	0.00
sppnw23	0.01	83	0.95	0.29	0	0.00
sppnw24	0.01	134	0.74	0.30	1	0.01

Continued on next page

Table 11 (continued)

Instance	time	iter.	ratio	density	ref	ref. time
sppnw25	0.01	107	0.80	0.48	0	0.00
sppnw26	0.00	87	0.70	0.31	0	0.00
sppnw27	0.02	110	0.55	0.47	0	0.00
sppnw28	0.01	88	0.67	0.51	0	0.00
sppnw29	0.01	162	0.89	0.45	1	0.00
sppnw30	0.01	77	0.54	0.51	0	0.00
sppnw31	0.02	148	0.77	0.42	1	0.00
sppnw32	0.00	66	0.84	0.43	0	0.00
sppnw33	0.03	217	0.70	0.46	1	0.00
sppnw34	0.01	113	0.75	0.47	1	0.00
sppnw35	0.03	166	0.52	0.28	1	0.00
sppnw36	0.01	145	0.75	0.47	0	0.00
sppnw37	0.01	66	0.79	0.46	0	0.00
sppnw38	0.02	158	0.70	0.34	0	0.00
sppnw39	0.01	81	0.48	0.47	0	0.00
sppnw40	0.00	50	0.68	0.45	0	0.00
sppnw41	0.01	41	0.76	0.31	0	0.00
sppnw42	0.01	87	0.70	0.43	0	0.00
sppnw43	0.01	145	0.94	0.36	1	0.00
sppkl01	0.20	856	0.82	0.30	7	0.01
sppkl02	0.93	2750	0.73	0.29	30	0.04
sppnw03	0.62	1151	0.69	0.29	10	0.03
sppnw04	0.98	1953	0.92	0.32	12	0.01
sppnw05	3.39	1395	0.86	0.13	14	0.05
sppnw06	0.09	404	0.62	0.39	3	0.01
sppnw13	0.28	1001	0.98	0.14	6	0.01
sppnw14	1.75	2291	0.95	0.13	17	0.05
sppnw16	1.56	1317	0.99	0.01	38	0.31
sppnw17	1.94	1843	0.74	0.24	13	0.03
sppnw18	0.52	721	0.80	0.17	16	0.13
sppus02	0.29	791	0.34	0.33	12	0.01
sppus03	0.88	973	0.47	0.43	21	0.08
sppus04	1.25	865	0.50	0.20	54	0.66

Table 12 shows a typical profiling result of the instances. Most of the computation is spent on pricing even after our adjustment. Hence it is required to develop a better procedure to decrease the time spent choosing and adding columns.

Table 12: Profiling Results: COLS - sppnw14

Operation	percentage
addition of a column	79.0

Operation	Percentage
obtaining x_B^* and ρ^*	13.6
deletion of columns	5.8
other	1.6

The results on large instances are not included in this chapter. They require more than 300 second for all large instances as compared with CPLEX, which solves these instance within a minute. Hence we designed a variation of COLS so that we can improve its convergence rate. From experience, we have found that the COLS algorithm with small M converges to a non optimal solution (error) quickly. So we tried using dynamic M values to solve the large problem quickly. We start M at 1.0×10^4 and we double M every 3-400 iterations until M reaches 1.0×10^8. The results are listed in the Table 13. It is still much slower than the CPLEX Primal and Dual simplex solvers, but it gives significant improvement compared to the results without the dynamic M value method. Because of this initial success, a potential future research direction is determining M dynamically as a function of the instance size, solution progress, and/or problem data.

Table 13: CPU Time: COLS($M = 1.0 \times 10^6$) - large

	CPLEX primal		CPLEX dual		COLS	
Instance	time	iter.	time	iter.	time	iter.
sppaa01	34.65	28517	8.97	4107	75.08	3715
sppaa02	9.59	18919	0.77	1067	8.14	894
sppaa03	38.75	37595	4.24	2476	49.86	2438
sppaa04	4.19	10101	2.37	1578	16.73	1605
sppaa05	25.62	25760	4.56	2846	48.93	2562
sppaa06	21.02	30577	1.77	1463	27.78	1737
sppnw01	1.66	3270	0.53	130	1.61	1161
sppnw02	3.80	5339	1.37	154	2.42	1919

5.3 Concluding Remarks

The COLS algorithm shares many core feature with LSPD, such as its sparse matrix computation and re-factorization routine. The computational results show that it works

pretty well even without upper-bound implementation. But more research is required to improve convergence results for larger problems. One important research direction for the future is to incorporate a primal long-step algorithm in the upper-bound version, since it is used to get better convergence results in primal simplex.

CHAPTER VI

ALGORITHMS FOR A LARGE SCALE LINEAR PROGRAMMING

6.1 Primal-Dual Subproblem Method

The primal-dual subproblem method (PDSUB) was developed by [21]. PDSUB is designed for large scale optimization problems such as airline crew scheduling problems [2], which can be formulated as set-partitioning problems and consists of millions of columns and a few hundreds of rows.

This algorithm starts with a primal feasible solution and a dual feasible solution and moves toward satisfying the complementary slackness conditions. We want to solve a linear programming problem in standard form

$$\begin{aligned} \min \quad & cx & \\ \text{s.t.} \quad & Ax & = b \\ & x & \geq 0, \end{aligned} \qquad (\mathbf{P})$$

and its dual

$$\begin{aligned} \max \quad & \pi b & \\ \text{s.t.} \quad & \pi A & \leq c, \end{aligned} \qquad (\mathbf{D})$$

where c^T and x are the column vectors in $\mathbb{R}^n$, b and π^T are column vectors in $\mathbb{R}^m$, and A is a matrix of size $m \times n$. Assume that $(\mathbf{P})$ has matrix A which has millions of columns and several hundreds of rows, which is a standard size of airline crew pairing models. Because of the huge aspect ratio of A, column generation is a reasonable solution approach. PDSUB was developed to solve this kind of problem efficiently. PDSUB showed better convergence results in [20] than simplex based column generation approach.

PDSUB starts with a dual feasible point π and a subset $\tilde{A}$ of columns in A which corresponds to a feasible solution $\tilde{x}$ where $\tilde{A}\tilde{x} = b, \tilde{x} \geq 0$. Every iteration of PDSUB starts

with solving

$$\begin{aligned} \min \quad & \tilde{c}\tilde{x} & \\ \text{s.t.} \quad & \tilde{A}\tilde{x} &= b \\ & \tilde{x} &\geq 0, \end{aligned} \qquad (\mathbf{Psub})$$

where $\tilde{A}$ is a $m \times k$ matrix which is a subset of A, and $\tilde{c}^T$ and $\tilde{x}$ are column vectors in $\mathbb{R}^k$ corresponding to the columns of $\tilde{A}$. Let ζ^* be a dual feasible solution in (**Psub**). If dual point ζ^* is feasible for (**Psub**) and (**P**) then we have primal and dual optimal solutions $\tilde{x}^*$ and ζ^*. Assume that ζ^* is not dual feasible for (**P**). Observe that $\zeta^* b > \pi b$. Then we update π with

$$\pi' = \theta\pi + (1-\theta)\zeta^*,\ 0 \leq \theta \leq 1. \qquad (34)$$

π' must satisfy dual feasibility of (**D**). Hence

$$\pi' A_j = \theta\pi A_j + (1-\theta)\zeta^* A_j \leq c_j,$$

which gives

$$\theta \bar{c}_j^{\pi} + (1-\theta)\bar{c}_j^{\zeta} \geq 0,$$

where $\bar{c}_j^{\pi} = c_j - \pi A_j$ and $\bar{c}_j^{\zeta} = c_j - \zeta^* A_j$. So θ needs to satisfy following inequality:

$$\theta \geq \frac{-\bar{c}_j^{\zeta}}{\bar{c}_j^{\pi} - \bar{c}_j^{\zeta}},$$

for all columns having $\bar{c}_j^{\zeta} < 0$. Hence we choose θ with

$$\theta = \min\{\frac{-\bar{c}_j^{\zeta}}{\bar{c}_j^{\pi} - \bar{c}_j^{\zeta}} |\, \bar{c}_j^{\zeta} < 0\}.$$

Since ζ^* is dual infeasible for (**D**), we have $\theta > 0$. $\theta < 1$ since there is at least one $\bar{c}_j^{\zeta} < 0$. If not, θ can be increased infinitely with dual feasibility (**D**) so that (**P**) become infeasible. Then it contradicts the assumption that we have a feasible solution in (**P**). This operation improves the dual feasible point since $\pi' b = \theta\pi b + (1-\theta){\zeta^*}^T b$ and $\zeta b > \pi b$.

After updating π with (34), we need to update $\tilde{A}$ with all the columns with $c_j - \pi' A_j < \epsilon$, where ϵ has a positive value. To guarantee convergence we enforce that all the columns in the previous $\tilde{A}$ should be included in the new $\tilde{A}$.

Algorithm 4 PDSUB : min $cx, s.t.\ Ax = b, , x \geq 0$

1: Dual feasible solution π and a set of columns $\tilde{A}$ which includes primal feasible basis.
2: **loop**
3: Construct matrix $\tilde{A}$ with the columns of A with $c_j - \pi A_j \epsilon_{(>0)}$.
4: Solve min $\{\tilde{c}\tilde{x}\ :\ \tilde{A}\tilde{x} = b, \tilde{x} \geq 0\}$,
and denote the primal solution by $\tilde{x}^*$ and dual solution by ζ^*.
5: **if** ζ^* is dual feasible **then**
6: STOP. $\tilde{x}$ and ζ^* are primal and dual optimal solutions.
7: **end if**
8: $\pi \leftarrow \theta\pi + (1-\theta)\zeta^*$ where $\theta = min\{-\bar{c}_j^{\zeta}/(\bar{c}_j^{\pi} - \bar{c}_j^{\zeta})|\bar{c}_j^{\zeta} < 0\}$
9: **end loop**

New columns with $c_j - \pi' A_j < \epsilon$ are added to $\tilde{A}$ and the dual solution is strictly improved in each iteration. Thus we have a smaller duality gap every iteration so that we have finite convergence since we have finite choices of basis.

If a primal feasible basis is not available, we need to find a primal feasible matrix $\tilde{A}$ before PDSUB starts. We want to find a primal feasible matrix $\tilde{A}$ along with improving our dual solution. This can be done in a way similar to the primal-dual algorithm. In each iteration before obtaining primal feasible basis, we solve

$$\begin{aligned} \min \quad & \textstyle\sum \tilde{x}_j \\ \text{s.t.} \quad & \tilde{A}\tilde{x} = b \\ & \tilde{x} \geq 0. \end{aligned}$$

and denote the dual solution of this ζ^*. After solving this we need to update π with

$$\pi' = \pi + \theta\zeta^*, \tag{35}$$

satisfying dual feasibility so that θ is chosen as

$$\theta = \min\{\frac{\bar{c}_j^{\pi}}{\zeta^* A_j} : \zeta^* A_j > 0\}.$$

Based on the new π, $\tilde{A}$ is reconstructed with the columns having $c_j - \pi' A_j < \varepsilon$ until we have a primal feasible basis.

6.2 Least-Square Subproblem Method

The least-squares subproblem method (LSSUB) is the same as PDSUB except for the way subproblems are solved. LSSUB uses non-negative least-squares based algorithms to solve

subproblems. Crew pairing problems generally have primal feasible solutions since columns (legal parings) are generated to include current airline feasible pairings, which is definitely feasible and should be included in $\tilde{A}$. But in our case, we don't have an available primal feasible solution in the beginning.

Phase I of LSSUB is different from Phase I of PDSUB since LSSUB uses LSPD as its subproblem solver,

$$\begin{aligned} \min \quad & \zeta^T \zeta & \\ \text{s.t.} \quad & \tilde{A}\tilde{x} + \zeta && = b \\ & \tilde{x} && \geq 0. \end{aligned} \qquad (\mathbf{Lsub})$$

The dual is updated with the formula described in (35).

NNLS is very quick to find a primal feasible solution for (**P**). Preliminary computations on the problem indicate that obtaining a good dual point before obtaining a primal feasible solution is more important for the overall convergence than obtaining a primal feasible solution in a short time. Hence we delay the convergence to the primal feasible solution so that we can get a better quality dual solution. This can be done by controlling ϵ for constructing $\tilde{A}$.

In the second phase, the COLS algorithm is applied to solve the subproblem.

6.2.1 Computational Results

Two instances, **RJmod** and **RJmx**, were generated from flight schedules, and their sizes are listed in Table 14. Both have more than five million columns

The detailed execution results are in Table 15. Phase I obtains a primal feasible solution and Phase II obtains the optimal solution of the problem. Because of the large number of columns LSSUB spend most of its computation time in pricing, and solving the subproblems takes only 10% of the overall computation time.

We could obtain a primal feasible solution in one major iteration. But to get a better dual feasible solution, we delayed the primal feasibility by controlling ϵ. We started with very small ϵ and gradually increased it.

For the comparison, CPLEX Sifting solver is used to solve these problems. CPLEX Sifting solver with primal simplex has better results than with CPLEX Sifting solver with dual simplex. The results are in Table 16. LSSUB performs around four times better in

Table 14: Instances: RJmod and RJmax

	RJ mod	RJ max
row	212	219
col	5052633	5091544
obj. value	21090.8	22958.0

Table 15: Execution Details: Larger-Scale

		RJmod		RJmax	
		LSSUB	PDSUB	LSSUB	PDSUB
Phase I	Dual Obj.	20064.3	19802.1	21944.4	21683.5
	# iter	5	9	5	10
	Solving time	4.4	3.9	6.7	4.2
Phase II	# iter	3	6	3	5
	Solving time	41.8	19.8	56.7	21.1
	Pricing time	78.0	145.4	77.7	147.7

Table 16: Execution Comparison: LSSUB and CPLEX Sifting Algorithm

	RJ mod(cpu sec.)	RJ max(cpu sec.)
CPLEX Sifting	561.5	554.3
LSSUB	124.3	141.27

running time. It is not quite a fair comparison since LSSUB have advantages for finding better subproblem sizes. However this comparison shows LSSUB has relevance for the further research.

For more comparison, we have solved PDSUB with CPLEX algorithm. The second Phase is not the main point of the comparison since we already showed that CPLEX is quicker to solve general linear programming problem. In the first phase of LSSUB requires less iteration. This is expected result since the first phase is solved by NNLS and its improving direction is steepest direction. In the second phase, COLS algorithm iterates like simplex algorithm such that the dependent columns are enter the basis and some of the columns are dropped from the basis. This type of the iteration in COLS algorithm takes more time than entering independent columns. This is the part remained for the future research.

REFERENCES

[1] Ahuja, R. K., Magnanti, T. L., and Olin, J. B., *Network Flows.* Prentice Hall, 1993.

[2] Annal, R., Tanga, R., and Johnson, E. L., "A global approach to crew-pairing optimization," *IBM Systems Journal*, vol. 31, no. 1, pp. 71–78, 1992.

[3] Barnes, E., Chen, V., Gopalakrishnan, B., and Johnson, E. L., "A least-square primal-dual algorithm for solving linear programming problems," *Operations Research Letters*, vol. 30, pp. 289–294, 2002.

[4] Beasly, E., "OR-library" http://mscmga.ms.ic.ac.kr/info.html .

[5] Björck, A., *Numerical Methods for Least Squares Problem.* Society for Industrial and Applied Mathematics, 1996.

[6] Chvátal, V., *Linear Programming.* W. H. Freeman and Company, 1980.

[7] Coleman, T. F., Edenbrandt, A., and Gilbert, J. R., "Predicting fill for sparse orthogonal factorization," *Journal of the Association for Computing Machinery*, vol. 33, no. 3, pp. 517–532, 1986.

[8] Davis, T. A., Gilbert, J. R., Larimore, S., and Ng, E., "A column approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software*, vol. 30, no. 3, pp. 353–376, 2004.

[9] Duff, I. S., "Pivot selection and row ordering in givens reduction on sparse matrices," *Computing*, vol. 13, pp. 239–248, 1974.

[10] Duff, I. S. and Reid, J. K., "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Transactions on Mathematical Software*, vol. 9, no. 3, pp. 302–325, 1983.

[11] Forrest, J. and Tomlin, J., "Updated triangular factors of the basis to maintain sparsity in the product form simplex method," *Mathematical Programming*, vol. 2, pp. 263–278, 1972.

[12] Forrest, J. and Tomlin, J., "Implementing the simplex method for the optimization subroutine library," *IBM Systems Journal*, vol. 31, pp. 11–25, 1992.

[13] Gentleman, W. M., "Error analysis of QR decompositions by givens transformations," *Linear Algebra and Its Applications*, vol. 10, pp. 189–197, 1975.

[14] George, A. and Heath, M. T., "Solution of sparse linear least squares problems using givens rotations," *Linear Algebra and its Applications*, vol. 34, pp. 60–83, 1980.

[15] George, A. and Liu, J. W. H., "A fast implementation of the minimum degree algorithm using quotient graphs," *ACM Transactions on Mathematical Software*, vol. 6, no. 3, pp. 337–358, 1980.

[16] George, A. and Ng, E., "On row and column orderings for sparse least squares problems," *SIAM Journal on Numerical Analysis*, vol. 20, pp. 326–344, 1983.

[17] Golub, G., "Numerical methods for solving linear least squares problems," *Numerische Mathematik*, vol. 7, pp. 206–216, 1965.

[18] Golub, G. H. and Loan, C. F. V., *Matrix Computation*. The Johns Hopkins University Press, third ed., 1996.

[19] Gopalakrishnan, B., *Least-square Methods for Solving Linear Programming Problems*. PhD thesis, Georgia Institute of Technology, June 2002.

[20] Hu, J., *Solving Linear Programs Using Primal-dual Subproblem Simplex Method and Quasi-explicit Matrices*. PhD thesis, Georgia Institute of Technology, June 1996.

[21] Hu, J. and Johnson, E. L., "Computational results with a primal-dual subproblem simplex method," *Operations Research Letters*, vol. 25, pp. 149–157, 1999.

[22] Klingmana, D., Napier, A., and Stutz, J., "Netgen - a program for generating large scale capacitated assignment, transportation, and minimum cost ow networks," *Management Science*, vol. 20, pp. 814–820, 1974.

[23] Kuhn, H. W., "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[24] Kuhn, H. W. and Tucker, A. W., eds., *Linear Inequalities and Related Systems*. Princeton University Press, 1956.

[25] Lawson, C. L. and Hanson, R. J., *Solving Least Squares Problems*. Prentice-Hall, 1974.

[26] Leichner, S. A., Dantzig, G. B., and Davis, J. W., "A strictly improving linear programming phase I algorithm," *Annals of Operations Research*, vol. 47, pp. 409–430, 1993.

[27] Liu, J. W. H., "On general row merging schemes for sparse givens transformation," *SIAM Journal of Sci. Stat. Comput.*, vol. 7, no. 4, pp. 1190–1211, 1986.

[28] Markowitz, H. M., "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, pp. 255–269, 1957.

[29] Maros, I., *Computation Technique of the Simplex Method*. Kluwer Academics Publisher, 2003.

[30] Matstoms, P., "Sparse linear least squares problems in optimization," *Computational Optimization and Applications*, vol. 7, pp. 89–110, 1997.

[31] Papadimitriou, C. H. and Steiglitz, K., *Combinatorial Optimization*. Dover Publications, 1998.

[32] Skorobohatyj, G., "`MP-test` data: A collection of test data for various mathematical programming problems," 2003. http://elib.zib.de/pub/Packages/mp-testdata/ .

[33] 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.

[34] YANNAKAKIS, M., "Computing the minimum fill-in is NP-complete," *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, pp. 77–79, 1991.

VITA

Seunghyun Kong was born in Seoul, South Korea in March 1974. He obtained his bachelor's degree from Seoul National University, in Material Science and Engineering. After he finished his military service, he joined master program in Industrial and Systems Engineering at Georgia Institute of Technology in 2000. After completing master degree, he enters the Ph.D program in the same school and will be awarded his doctorate in May 2007. He started his first job at SAS institute, Cary, NC, as a research and development staff from March 2007.