A Modified Simplex Method for Solving $Ax = b$, $x \geq 0$, for Very Large Matrices A Arising from a Calibration Problem

by

Zoë A. MacDonald

A research paper submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

A Modified Simplex Method for Solving $Ax = b$, $x \geq 0$, for Very Large Matrices A Arising from a Calibration Problem

Zoë A. MacDonald
Master of Science
Graduate Department of Computer Science
University of Toronto
2020

In the area of correlated multivariate discrete distributions, including, in particular, multivariate Poisson distributions, the calibration problem is the problem of finding a multidimensional discrete probability distribution with given marginals and a desired correlation matrix. This can be done by taking a convex combination of a set of "extreme" multivariate discrete distributions, each with the given marginals and an "extreme" correlation matrix. The problem of determining the coefficients for this convex combination can be solved by finding a solution to an underdetermined system of linear equations $Ax = b$, with the constraint $x \geq 0$.

At smaller dimensions, a solution to this underdetermined system of linear equations $Ax = b$, with the constraint $x \geq 0$, can be computed using phase 1 of the simplex method. However, the number of extreme distributions involved in such a problem increases exponentially as the dimensionality of the desired correlation matrix increases. Since each extreme distribution is associated with a column of the matrix A, the number of columns in the matrix A also increases exponentially as the dimensionality of the desired correlation matrix increases. This paper describes a variant of the simplex method for use on this specific problem, the cost of which, in most cases that we have tested, grows polynomially with the dimensionality of the desired correlation matrix (and polylogarithmically with the number of extreme distributions). This allows for solutions to be calculated with correlation matrices up to size 52×52 in a reasonable amount of time (approximately 20 minutes on a home desktop computer). This was the largest problem tested with this method, due to precision limits, but in principle there is no reason it would not work on even larger problems.

Contents

Chapter 1

Introduction

In the area of correlated multivariate discrete distributions, including, in particular, multivariate Poisson distributions, the calibration problem is the problem of finding a multidimensional discrete probability distribution with given marginals distributions and a desired correlation matrix.

To be more specific, let $X_1, X_2, \ldots, X_d$ be $d \geq 2$ correlated random variables and let

$$P_{i_1,i_2,\ldots,i_d} = \mathbb{P}(X_1 = i_1, X_2 = i_2, \ldots, X_d = i_d)$$

where, for $k \in \{1, 2, \ldots, d\}$, $i_k \in \{0, 1, 2, \ldots\}$. That is, P is the joint d-dimensional discrete probability distribution associated with the random variables $X_1, X_2, \ldots, X_d$.

For $k \in \{1, 2, \ldots, d\}$, the k^{th} marginal distribution associated with the d-dimensional discrete probability distribution P is $Q^{(k)}$, where

$$Q^{(k)}_{i_k} = \mathbb{P}(X_k = i_k) = \sum_{j \in \mathcal{I}_k} \sum_{i_j=0}^{\infty} P_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_d}$$

for $i_k \in \{0, 1, 2, \ldots\}$ and

$$\mathcal{I}_k = \{j : j \in \{1, 2, \ldots, d\} \text{ but } j \neq k\}$$

For k and $l \in \{1, 2, \ldots, d\}$, the correlation associated with the random variables X_k and X_l (and hence also associated with the d-dimensional discrete probability distribution P) is

$$C_{k,l} = \text{corr}(X_k, X_l) = \frac{\mathbb{E}[X_k X_l] - \mathbb{E}[X_k]\mathbb{E}[X_l]}{\sigma(X_k)\sigma(X_l)}$$

where $\mathbb{E}$ is the expectation operator and σ is the standard deviation operator.

We can assemble the correlations $C_{k,l}$, for k and $l \in \{1, 2, \ldots, d\}$, into a $d \times d$ correlation matrix C, where $C_{k,l}$ is the element in row k and column l of the matrix C. Note that C is the $d \times d$ correlation matrix associated with the d-dimensional discrete probability distribution P.

Note that, if the marginal distributions $Q^{(k)}$ and $Q^{(l)}$ of P are given, then $\mathbb{E}[X_k]$ and $\sigma(X_k)$ are completely determined by $Q^{(k)}$ and $\mathbb{E}[X_l]$ and $\sigma(X_l)$ are completely determined by $Q^{(l)}$, but $\mathbb{E}[X_k X_l]$ is not completely determined by the marginals of P. To be more specific,

$$\mathbb{E}[X_k X_l] = \sum_{i_k=0}^{\infty} \sum_{i_l=0}^{\infty} i_k i_l P^{(k,l)}_{i_k,i_l}$$

where

$$P^{(k,l)}_{i_k,i_l} = \mathbb{P}(X_k = i_k, X_l = i_l) = \sum_{j \in \mathcal{I}_{k,l}} \sum_{i_j=0}^{\infty} P_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{l-1},i_l,i_{l+1},\ldots,i_d}$$

for i_k and $i_l \in \{0, 1, 2, \ldots\}$, where

$$\mathcal{I}_{k,l} = \{j : j \in \{1, 2, \ldots, d\} \text{ but } j \neq k \text{ and } j \neq l\}$$

The calibration problem is to find a d-dimensional discrete probability distribution P with given marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$, and a given correlation matrix C.

Chiu et al. [3, Section 4] provide a method for solving the calibration problem described above. We briefly summarize their approach here. For further details, see [3].

To begin, define the "monotonicity structure"

$$\mathbf{e}^{(j)} = (e^{(j)}_1, e^{(j)}_2, \ldots, e^{(j)}_d)$$

by setting $e^{(j)}_1 = 0$ and

$$e^{(j)}_i = \begin{cases} 0 & \text{if } X_1 \text{ and } X_i \text{ are to be maximally correlated} \\ 1 & \text{if } X_1 \text{ and } X_i \text{ are to be minimally correlated} \end{cases}$$

for $i = 2, \ldots, d$. Note that, for $i = 2, \ldots, d$, there are two choices for the i^{th} component of each $\mathbf{e}^{(j)}$. Hence, there are $n = 2^{d-1}$ distinct monotonicity structures $\mathbf{e}^{(1)}$, $\mathbf{e}^{(2)}, \ldots, \mathbf{e}^{(n)}$.

For $j = 1, 2, \ldots, n$, we associate with each monotonicity structure $\mathbf{e}^{(j)}$ an "extreme" d-dimensional discrete probability distribution $P^{(j)}$ having the same marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$, as the desired d-dimensional discrete probability distribution P, but having an "extreme" correlation matrix $C^{(j)}$ with the property that

$$C^{(j)}_{k,l} = \begin{cases} \text{is the maximal possible value for } \operatorname{corr}(X_k, X_l) \text{ if } e^{(j)}_k = e^{(j)}_l \\ \text{is the minimal possible value for } \operatorname{corr}(X_k, X_l) \text{ if } e^{(j)}_k \neq e^{(j)}_l \end{cases}$$

How to compute the $C^{(j)}_{k,l}$, for k and $l \in \{1, 2, \ldots, d\}$ and $j \in \{1, 2, \ldots, n\}$, and the associated d-dimensional discrete probability distribution $P^{(j)}$ are described in [3, Section 4].

In passing, note that, since $C^{(j)}_{k,l}$ is a correlation, $C^{(j)}_{k,l} \in [-1, 1]$. However, even though we maximize $C^{(j)}_{k,l}$ if $e^{(j)}_k = e^{(j)}_l$ or minimize $C^{(j)}_{k,l}$ if $e^{(j)}_k \neq e^{(j)}_l$, it often happens that the maximal possible value for $C^{(j)}_{k,l} < 1$ and/or the minimal possible value for $C^{(j)}_{k,l} > -1$.

As noted in [3, Section 4], we can now reduce the calibration problem described in the beginning of this chapter to finding a set of coefficients $\omega_1, \omega_2, \ldots, \omega_n$ satisfying

$$\begin{aligned} \sum_{j=1}^{n} \omega_j C^{(j)} &= C \\ \sum_{j=1}^{n} \omega_j &= 1 \\ \omega_j &\geq 0 \end{aligned} \tag{1.1}$$

and then setting

$$P = \sum_{j=1}^{n} \omega_j P^{(j)} \tag{1.2}$$

Since each $P^{(j)}$, for $j = 1, 2, \ldots, n$, is a d-dimensional discrete probability distribution, it follows easily from (1.2) and the second and third lines of (1.1) that P is also a d-dimensional discrete probability distribution. Moreover, since each $P^{(j)}$, for $j = 1, 2, \ldots, n$, has the same marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$, it again follows easily from (1.2) and the second and third lines of (1.1) that P also has the required marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$. Furthermore, it follows easily from (1.1) and (1.2) that P has correlation matrix C.

Therefore, if we can find a set of coefficients $\omega_1, \omega_2, \ldots, \omega_n$ satisfying (1.1), then the d-dimensional discrete probability distribution P given by (1.2) satisfies the calibration problem described in the beginning of this chapter. That is, P has the specified marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$, and the required correlation matrix C.

On the other hand, if there are no coefficients $\omega_1, \omega_2, \ldots, \omega_n$ satisfying (1.1), then there is no d-dimensional discrete probability distribution P with marginal distributions $Q^{(k)}$, for $k = 1, 2, \ldots, d$, and correlation matrix C.

Note in passing that we can view the problem (1.1) as finding a convex combination of the extreme correlation matrices $C^{(j)}$ that is equal to the desired correlation matrix C. Also, we can view (1.2) as writing the desired d-dimensional discrete probability distribution as a convex combination of the extreme d-dimensional discrete probability distributions $P^{(j)}$, where each $P^{(j)}$ has correlation matrix $C^{(j)}$.

To find a set of coefficients $\omega_1, \omega_2, \ldots, \omega_n$ satisfying (1.1) or to show that no such set of coefficients exists, we re-formulate the conditions (1.1) as a constrained system of linear equations. To this end, note that each matrix $C^{(j)}$ is symmetric with 1's on its diagonal. This is also true for C. Therefore, each matrix $C^{(j)}$ is completely determined by the elements in its strictly upper-triangular part. Again, this is also true for C. Hence, we can "vectorize" each matrix $C^{(j)}$ by putting the elements $C^{(j)}_{1,l}$, for $l = 2, \ldots, d$, in the first row of the strictly upper-triangular part of $C^{(j)}$ in the first $d-1$ elements of a column vector $\hat{A}^{(j)}$. Then putting the elements $C^{(j)}_{2,l}$, for $l = 3, \ldots, d$, in the second row of the strictly upper-triangular part of $C^{(j)}$ in the next $d-2$ elements of $\hat{A}^{(j)}$. And so on. That is, we set

$$\hat{A}^{(j)}_{(2d-k)(k-1)/2+l-k} = C^{(j)}_{k,l}$$

for $k = 1, 2, \ldots, d-1$ and $l = k+1, \ldots, d$. Note that $\hat{A}^{(j)}$ is a column vector of length $\hat{m} = d(d-1)/2$.

Similarly, we can vectorize C by setting

$$\hat{b}_{(2d-k)(k-1)/2+l-k} = C_{k,l}$$

Note that $\hat{b}$ is also a column vector of length $\hat{m} = d(d-1)/2$. Now let $\hat{A}^{(j)}$ be the j^{th} column of the $\hat{m} \times n$ matrix $\hat{A}$. That is,

$$\hat{A} = [\hat{A}^{(1)}\ \hat{A}^{(2)}\ \dots\ \hat{A}^{(n)}]$$

Then the conditions (1.1) can be rewritten as

$$\begin{aligned} \hat{A}\omega &= \hat{b} \\ \mathbf{1}^T\omega &= 1 \\ \omega_j &\geq 0 \qquad \text{for } j = 1, 2, \dots, n \end{aligned} \tag{1.3}$$

where $\omega = (\omega_1, \omega_2, \dots, \omega_n)^T$ and $\mathbf{1} = (1, 1, \dots, 1)^T \in \mathbb{R}^n$.

Note in passing that we can view the problem (1.3) as finding a convex combination of the vectors $\hat{A}^{(j)}$ (representing the extreme correlation matrices $C^{(j)}$) that is equal to the desired vector $\hat{b}$ (representing the desired correlation matrix C).

We now transform (1.3) into a more standard form. First, replace ω by x, since x is more commonly used than ω as the vector of unknowns in linear algebra problems. Second, if $\hat{b}_i < 0$, multiply $\hat{b}_i$ by -1 and also multiply each element in the i^{th} row of $\hat{A}$ by -1. After this is done, all $\hat{b}_i \geq 0$. Third, add a row of 1's to the bottom of the matrix $\hat{A}$ and call the new matrix A; also add a 1 to the bottom of the vector $\hat{b}$ and call the new vector b. This incorporates the constraint $\mathbf{1}^T\omega = 1$ in (1.3) into the system for linear equations $Ax = b$. Note that A is an $m \times n$ matrix, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, where

$$\begin{aligned} m &= \hat{m} + 1 = \frac{d(d-1)}{2} + 1 \\ n &= 2^{d-1} \end{aligned}$$

After these transformations are performed, we can rewrite (1.3) as

$$\begin{aligned} Ax &= b \\ x &\geq 0 \end{aligned} \tag{1.4}$$

where $x \geq 0$ means $x_j \geq 0$ for all $j = 1, 2, \dots, n$.

The system (1.4) is in the form of the constraints that appear in many Linear Programming Problems (LPPs):

$$\min_{\substack{Ax=b \\ x\geq 0}} \bar{c}^T x \tag{1.5}$$

To start the simplex method to solve (1.5), we need a "feasible solution" to (1.4). Below, we adapt the method presented in [4, Section 13.5] for finding a solution to (1.4).

First we augment the system (1.4) as follows.

$$\begin{aligned} [A\ I]\begin{pmatrix} x \\ y \end{pmatrix} &= b \\ \begin{pmatrix} x \\ y \end{pmatrix} &\geq 0 \end{aligned} \tag{1.6}$$

where I is the $m \times m$ identity matrix and $y \in \mathbb{R}^m$. Clearly, $x = 0$ and $y = b$ is a solution to (1.6), since the second transformation described above ensures that $y = b \geq 0$. Moreover, if we can find a solution $(x^T\, y^T)^T$ to (1.6) with $y = 0$, then the associated x satisfies (1.4). So, let

$$\tilde{A} = [A\ I] \qquad z = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \bar{c} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

where $\tilde{A}$ is an $m \times (m+n)$ matrix, $z \in \mathbb{R}^{m+n}$ and $c \in \mathbb{R}^{m+n}$ with $\bar{c}_i = 0$ for $i = 1, 2, \ldots, n$ and $\bar{c}_i = 1$ for $i = n+1, n+2, \ldots, n+m$. Now consider the LPP

$$\min_{\substack{\tilde{A}z=b \\ z\geq 0}} \bar{c}^T z \tag{1.7}$$

As noted above, an initial feasible solution to (1.7) is $z = (x^T\, y^T)^T$ with $x = 0$ and $y = b$. So, we have an initial feasible solution to start the simplex method to solve (1.7). Given the constraint in (1.7) that $z \geq 0$, which implies that $y \geq 0$, we have

$$\bar{c}^T z = (0\,0\,\cdots\,0\,1\,1\,\cdots 1)\begin{pmatrix} x \\ y \end{pmatrix} = (0\,0\,\cdots\,0)\,x + (1\,1\,\cdots 1)\,y = \sum_{i=1}^{m} y_i \geq 0 \tag{1.8}$$

Suppose we solve the LPP (1.7) and we find that the minimizer $z^* = ((x^*)^T\,(y^*)^T)^T$ of (1.7) satisfies $\bar{c}^T z^* = 0$. Then, from (1.8), it follows that $y^* = 0$. Hence, as noted above, the associated x^* satisfies (1.4).

On the other hand, suppose we solve the LPP (1.7) and we find that the minimizer $z^* = ((x^*)^T\,(y^*)^T)^T$ of (1.7) satisfies $\bar{c}^T z^* > 0$. Then there is no solution to (1.4), since, if there were a solution $\hat{x}$ to (1.4), we could let $\hat{z} = (\hat{x}^T\,\hat{y}^T)^T$ with $\hat{y} = 0$ and note that $\tilde{A}\hat{z} = b$, $\hat{z} \geq 0$ and $\bar{c}^T \hat{z} = 0$, contradicting the assumption that the minimizer z^* of (1.7) satisfies $\bar{c}^T z^* > 0$.

So, for small problems (e.g., $d \leq 10$), we can solve the LPP (1.7) to either find a solution x to (1.4) or show that there is no solution to (1.4).

However, for medium to large sized problems (e.g., $d \approx 51$), the major difficulty arises that A has $n = 2^{d-1} = 2^{50} \approx 10^{15}$ columns. So, it would take an unacceptably large amount of time just to construct the matrix A (assuming we could store it). (Note that A is not usually sparse.)

The focus of this research paper is to use the structure of the calibration problem described at the beginning of this chapter to develop a modified version of the simplex method that solves the LPP (1.7) without ever generating all the columns of A or $\tilde{A}$.

The outline of this research paper is as follows. In Chapter 2, we provide a glossary of several terms used in this paper, for ease of reference. In Chapter 3, we review the simplex method using the "full tableau" approach. In Chapter 4, we propose a modified simplex method and compare it to the simplex method outlined in Chapter 3. In Chapter 5 we detail the algorithm for pivot column selection in this modified simplex method. In Chapter 6 we analyze the time and space complexity of our modified simplex method, and of some alternatives that were explored. Chapter 7 concludes the paper.

Finally, we note that, if there is a solution to (1.4), it usually is not unique. However, in this research paper, we are content to find any solution to (1.4). We leave to future work the interesting question of whether there is a "best" solution to (1.4) and how to find it.

Chapter 2

Glossary and Terms

Terms used throughout the paper are collected here as a quick reference. Some have been mentioned already, and others are explained in greater detail in later chapters.

Correlation Matrix. A $d \times d$ matrix, C, from which the problem derives. d is the primary parameter for categorizing the sizes of these problems ("a problem of size d" refers to one based on an $d \times d$ correlation matrix, C), and that terminology is used frequently in this thesis. For a problem of size d, the number of elements in the strictly upper-triangular part of the associated correlation matrix C is $\hat{\boldsymbol{m}} = \frac{d(d-1)}{2}$ Additionally, as a shorthand, we define $m = \hat{m} + 1$, as these figures both arise frequently.

Generating Vectors. The source of the resultant vectors, these have the form of a 1, followed by a pattern of 1s and 0s. The set of generating vectors contains all possible such vectors of 1s and 0s. For a problem of size d, the number of generating vectors is $n = 2^{d-1}$, and the length of the generating vectors is d. Each generating vector is associated with one extreme measure.

Resultant Vectors. The resultant vectors are the vectorized forms of the extreme correlation matrices $C^{(j)}$ used in the calibration (1.1). They can be calculated from the generating vectors by taking every pair of generating vector elements in top-down order, and where the pair is equal the resultant vector element is the maximal correlation of that pair, and where they are not it is the minimal correlation of that pair. The number of resultant vectors is $n = 2^{d-1}$ (one generated by each generating vector), and the size of each resultant vector is $\hat{m} = \frac{d(d-1)}{2}$. The resultant vectors (plus the convexity constraint) form the columns of the A matrix.

Target Vector. A vector specific to the flipping algorithm, for which we want to find a resultant vector to give a negative dot product of near maximal magnitude.

Objective Vector. The objective vector, $\hat{b}$ is defined to be a vectorized version of the desired correlation matrix C, as detailed in Chapter 1. In the tableau, the top $\hat{m}$ elements of b are initialized to $\hat{b}$, with the remaining m^{th} element being 1 for the convexity constraint.

Rowops matrix. The "rowops" (short for "row operations") matrix, also called G, is the product of all tableau updates, such that multiplying G by a column of $\hat{A}$ provides an up-to-date version of that column and its current reduced cost, as if it had been in the tableau since its initalization.

Algorithm Names:

- The "Full Tableau Simplex method" is a version of the standard simplex algorithm which is similar in tableau arrangement to our truncated tableau simplex method. Unlike the truncated tableau simplex method, the full tableau contains the entire A-matrix, which means it also can choose the exact column of A to minimize the reduced cost at each simplex iteration.

- The "Truncated Tableau Simplex method" is our algorithm. It stores only the active columns of A in memory, and can dynamically calculate other columns when they need to be accessed. Its version of the tableau is called the truncated tableau.

- The "Flipping Algorithm" is a part of the truncated tableau simplex method that finds a resultant vector with a low reduced cost value in logarithmic-in-n time in the average case. It is so named because it operates by testing "flips" of the signs of elements of generating vectors to search for resultant vectors to introduce to the active set.

$\boldsymbol{A, b, c, E, i}$. In the context of the simplex tableau, these refer to its five parts. $\boldsymbol{A}$ and $\boldsymbol{E}$ form the main section of the tableau. At initialization, $\boldsymbol{A} = A$, $\boldsymbol{E} = I$ (the identity matrix), and $\boldsymbol{b} =$ b, of $Ax + Ez = b$, after pre-processing to fit Simplex method constraints (requiring b to be entirely non-negative). $\boldsymbol{c}$ is the reduced cost vector. In the truncated simplex tableau, only a subset of $[A|E]$ is included in the tableau at any given time, but the term A always refers to the entire set of resultant vectors. For reasons that are detailed later, for each column j, c_j is initially $-\sum_{i=1}^{m} a_{i,j}$. Finally, $\boldsymbol{i}$ is a row added to the bottom of the truncated tableau: $\mathbf{i}_j$ is the index of the column in A associated with the column j in the truncated tableau. All columns of E have $\mathbf{i}_j = 0$, as this index-tracking is not necessary for those columns.

$\boldsymbol{\hat{A}, \hat{b}, \hat{m}}$. These "hat" symbols represent the value before being pre-processed for the truncated tableau simplex method. The symbols $\hat{A}$ and $\hat{b}$ represent A and b without the added convexity constraint. Additionally, $\hat{m} = m - 1$ represents the sizes of these original columns and vectors.

$\boldsymbol{O(g(n)), \Omega(g(n)), \Theta(g(n))}$. These symbols represent classes of functions, in terms of how their value increases as their input n increases. If some function $f(n)$ is $O(g(n))$, then $f(n)$ grows at most as fast as $cg(n)$ as n approaches infinity, for some constant c. Likewise, if $f(n)$ is $\Omega(g(n))$, then $f(n)$ grows at least as fast as $cg(n)$ as n approaches infinity, for some positive constant c. If $f(n)$ is $\Theta(g(n))$, then $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

$\boldsymbol{\bar{c}}$. The original cost vector defining the cost function of a given minimization problem. For all the problems tackled in this research paper, $\bar{c} = (0, 0, \ldots, 0, 1, 1, \ldots 1)^T$, so this is not discussed at length. This is usually referred to with the symbol c in LPPs, but that is also used to refer to the c-vector of a simplex tableau, so we use $\bar{c}$ for the cost vector to distinguish them.

Chapter 3

Review of Full Tableau Simplex Method

This implementation is based on Nasser M. Abbasi's free-for-use Matlab Simplex code, specifically its first phase [1]. There are many small details in a simplex tableau which may change from implementation to implemention, so we explain it in this chapter in a way that is consistent with the truncated tableau method, introduced in the following chapter.

3.1 The Initial Tableau

The tableau initialization function accepts the arguments A and b. Using these it constructs a simplex tableau. For our purposes, A is a matrix that contains all our resultant vectors, as columns, with a row of "1"s added as the bottom row, as the convexity constraint. b is initialized from the *objective vector* with a 1 added to the bottom of the vector, again for the convexity constraint.

This version largely follows the description of the first phase of the simplex method in [2, Chapter 3]. That book also contains some example problems.

The vector c denotes the opportunity cost (or "reduced cost") of bringing a column of A into the basis.

In the tableau, c is a row vector underneath A, such that column j of A, $A_{*,j}$, has a corresponding c_j-value. This c_j is initialized to the negative sum of each element in the column $A_{*,j}$: $c_j = -(a_{1,j} + a_{2,j} + \cdots + a_{m,j})$, though this equality is not guaranteed to hold as the simplex method progresses. This does not apply to the columns of E, which are prevented from re-entering the basis, so we do not need to consider their reduced cost and count every column of E's reduced cost as 0. This is not typical of the "standard" simplex method, but makes some things simpler later on. Though the standard simplex method might sometimes reintroduce a column of E into the basis, setting the costs associated with columns of E to 0 stops it from doing so in our variant of the full tableau method. Note that this never prevents our method from finding a solution (this is discussed further in Section 4.5).

In a pre-processing step (explained in detail in Section 4.2.1), the tableau is altered such that $b_i \geq 0$ for all $i \in [1, \hat{m}]$. This allows us to initialize the auxiliary matrix E to an identity matrix.

Compositing A, b, c, and the auxiliary matrix E creates the initial $(m+1) \times (n+m+1)$ tableau, T_0:

$$T_0 = \begin{array}{ccccccc} a_{1,1} & \dots & a_{1,n} & e_{1,1} & \dots & e_{1,m} & b_1 \\ a_{2,1} & \dots & a_{2,n} & e_{2,1} & \ddots & e_{2,m} & b_2 \\ \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots \\ a_{m,1} & \dots & a_{m,n} & e_{m,1} & \dots & e_{m,m} & b_m \\ c_1 & \dots & c_n & 0 & 0 & 0 & 0 \end{array}$$

When it is initialized, the E section of the tableau is in the form of an $m \times m$ identity matrix, and forms the initial basis of the tableau. Basis columns of $[A|E]$ are associated with variables $x_i > 0$ and $y_j > 0$ in the solution of $Ax + Ey = b$, $x \geq 0$ and $y \geq 0$. After initialization, the columns of E are not distinguished from columns of A, so for the sake of brevity we uniformly represent generic tableau elements with a.

3.2 Simplex Iterations

At each iteration, the following steps are taken by the algorithm:

First: identify the most negative c-value in the tableau T_i shown below. In the truncated tableau simplex method this is not a trivial operation, but for this version all columns are accessible in the tableau, so it is simply a call to the "min" function. We'll call this minimal c-value "c_j". This j^{th} column is called the pivot column.

Second: we need to identify the pivot element, and thus the pivot row. The pivot is an element, $a_{p,j}$ in the pivot row p and pivot column j, for which some positive b_p divided by $a_{p,j}$ is minimal. That is, $p = index(minimum((\frac{1}{a_{1,j}}, \dots, \frac{1}{a_{m,j}}) \circledcirc (b_1, \dots, b_m)))$, for $b_p > 0$ where $\circledcirc$ denotes elementwise multiplication.

Third: once the pivot row is known, that row of the tableau is normalized by dividing all elements in row p by $a_{p,j}$. This makes the pivot element $a_{p,j} = 1$ (see tableau T_{i+1}).

$$T_i = \begin{array}{ccccccc} a^i_{1,1} & a^i_{1,2} & \dots & a^i_{1,j} & \dots & a^i_{1,n} & b^i_1 \\ a^i_{2,1} & a^i_{2,2} & \dots & a^i_{2,j} & \dots & a^i_{2,n} & b^i_2 \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a^i_{p,1} & a^i_{p,2} & \dots & a^i_{p,j} & \dots & a^i_{p,n} & b^i_p \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a^i_{m,1} & a^i_{m,2} & \dots & a^i_{m,j} & \dots & a^i_{m,n} & b^i_m \\ c^i_1 & c^i_2 & \dots & c^i_j & \dots & c^i_n & 0 \end{array}$$

$$T_{i+1} = \begin{array}{ccccccc} a^i_{1,1} & a^i_{1,2} & \dots & a^i_{1,j} & \dots & a^i_{1,n} & b^i_1 \\ a^i_{2,1} & a^i_{2,2} & \dots & a^i_{2,j} & \dots & a^i_{2,n} & b^i_2 \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a^i_{p,1}/a^i_{p,j} & a^i_{p,2}/a^i_{p,j} & \dots & a^i_{p,j}/a^i_{p,j} & \dots & a^i_{p,n}/a^i_{p,j} & b^i_p/a^i_{p,j} \\ \vdots & \vdots & & \vdots & & \vdots & \vdots \\ a^i_{m,1} & a^i_{m,2} & \dots & a^i_{m,j} & \dots & a^i_{m,n} & b^i_m \\ c^i_1 & c^i_2 & \dots & c^i_j & \dots & c^i_n & 0 \end{array}$$

Then, for all k except $k = p$, element $a_{k,j}$ is reduced to zero by multiplying the pivot row by $a_{k,j}$ and subtracting it from row k, as seen in tableau T_{i+2}. For the sake of legibility, we define the shorthand $s_{x,y} = a^{i+1}_{x,j} a^{i+1}_{p,y}$. Note that, since $a^{i+1}_{p,j} = 1$, this has the effect of making each element in the j^{th} column equal 0, except for $a_{p,j}$, which remains equal to 1. This pivot column becomes a new basis column, replacing the basis column which previously had a 1 in the pivot row. By this process of replacement, the tableau always has m basis columns, which form columns of an identity matrix.

$$T_{i+2} = \begin{array}{ccccccc}
a^i_{1,1} - s_{1,1} & a^i_{1,2} - s_{1,2} & \dots & a^i_{1,j} - s_{1,j} & \dots & a^i_{1,n} - s_{1,n} & b^i_1 - a^i_{1,j} b^{i+1}_p \\
a^i_{2,1} - s_{2,1} & a^i_{2,2} - s_{2,2} & \dots & a^i_{2,j} - s_{2,j} & \dots & a^i_{2,n} - s_{2,n} & b^i_2 - a^i_{2,j} b^{i+1}_p \\
\vdots & \vdots & & \vdots & & \vdots & \vdots \\
a^{i+1}_{p,1} & a^{i+1}_{p,2} & \dots & 1 & \dots & a^{i+1}_{p,n} & b^{i+1}_p \\
\vdots & \vdots & & \vdots & & \vdots & \vdots \\
a^i_{m,1} - s_{m,1} & a^i_{m,2} - s_{m,2} & \dots & a^i_{m,j} - s_{m,j} & \dots & a^i_{m,n} - s_{m,n} & b^i_m - a^i_{m,j} b^{i+1}_p \\
c^i_1 - c^i_j a^{i+1}_{p,1} & c^i_2 - c^i_j a^{i+1}_{p,2} & \dots & c^i_j - c^i_j & \dots & c^i_n - c^i_j a^{i+1}_{p,n} & 0
\end{array}$$

These iterations repeat until the vector of reduced costs $c \geq 0$, at which point an optimal solution has been found [2, Section 3.1]. This algorithm always terminates [2, Section 3.4].

3.3 The Final Tableau

When the algorithm terminates, the final tableau is of this form:

$$T_f = \begin{array}{ccccccc}
a_{1,1} & \dots & a_{1,n} & e_{1,1} & \dots & e_{1,m} & b_1 \\
a_{2,1} & \dots & a_{2,n} & e_{2,1} & \ddots & e_{2,m} & b_2 \\
\vdots & \ddots & \vdots & \vdots & & \vdots & \vdots \\
a_{m,1} & \dots & a_{m,n} & e_{m,1} & \dots & e_{m,m} & b_m \\
c_1 & \dots & c_n & 0 & 0 & 0 & 0
\end{array}$$

If all the basis vectors are associated with columns of A, then all auxiliary variables, y_i, are 0 and we have found a solution to $Ax = b, x \geq 0$. If some basis vectors are columns of E, then some auxiliary variables are positive, and there is no solution to $Ax = b, x \geq 0$. See the discussion following equation (1.8) for a more detailed explanation of this important point.

From this final tableau we can recover x in the typical manner for the simplex method: set $x_i = 0$ if column $A_{*,i}$ is not in the final basis, and set $x_j = b_k$ if column $A_{*,j}$ is in the final basis, where $A_{k,j} = 1$. (because of the form of the basis columns, this matches each x_j to a unique b_k). This x is a solution to $Ax = b, x \geq 0$.

The description above of the full tableau simplex method is quite brief. However, this algorithm is described in detail in many books. See, for example, [4, Chapter 13].

Chapter 4

Truncated Tableau Simplex Method

4.1 Overview of the Truncated Tableau Simplex Method

To solve the correlation problem described in Chapter 1, it would be simple to solve the associated LPP, as discussed in Chapter 3. However, because the problems we want to tackle are so large (with $d = 52$, there are $2^{51} \approx 2.25 \times 10^{15}$ columns of A, far too many to store in memory or perform calculations on), we need a way to run the simplex method where time and space costs are not dependent upon n, the number of columns of A. A revised simplex method is described in [2, Section 3.3] which purports to operate in $O(m^2)$ memory and $O(m^2)$ best-case time, however it notes that this is only true of the space-complexity because in "most large-scale problems that arise, A is very sparse and can be stored compactly." Our A matrix is certainly not sparse, so this method is not useful for our purposes. Additionally, its method for providing $O(m^2)$ best-case time gives no method for providing "effective" reduced cost values — simply calculating them one at a time, at random, and proceeding when a negative one is found. Though this does seem to provide polynomial-time performance in the cases we tested, it still leaves much room for improvement (a more detailed cost comparison between this "revised simplex" and our truncated tableau simplex method is given in Section 6.2.6.)

To accomplish a true reduction in time- and space-complexity, we need a way to dynamically calculate the information which the simplex method would otherwise read from the tableau. Namely, updated columns of A, and their reduced costs.

In the previously-described implementation of the simplex method, the tableau contains all the data associated with the minimization problem. In our truncated tableau simplex method, we use instead a *truncated tableau*, which explicitly stores only the current basis from the full tableau described in Chapter 3 alongside the b-column, with a new row, i, added to the bottom of the tableau, to store each A-column's index in $\hat{A}$.

To account for the information we've removed from the tableau, we need to introduce additional functions to make the algorithm work without having direct access to every value in A and c. The first key to solving this issue comes from the fact that we change the tableau exclusively by elementary row operations. We can "save" these row operations in a "rowops" matrix, G, which is initialized to the identity matrix and updated on each iteration to incorporate all row operations done on the tableau. Then, to replace a column in A with one that is not currently in the tableau, we can simply multiply it by G to apply all previous row operations to it. The details of this procedure are given in Section 4.3.

By this method, we can avoid storing and performing row operations on the large quantity of inactive columns of A until we need them. When a column enters the basis, we can multiply it by the rowops matrix, G, and insert it into the tableau, as if it had been there all along.

The second challenge arising from the truncated tableau is in using the c-row to find negative cost coefficients. Without storing the entire c-row (which contains either $n = 2^{d-1}$ elements or $n+m$ elements, depending on how you implement the method), we cannot simply search it to find the most negative cost coefficient at each step (and even if we could, it would be a prohibitively expensive $\Omega(2^{d-1})$ operation). Calculating which A-column has the minimum c-value (given the information available to the truncated tableau) is likely an NP-complete problem (see Section 6.3 for details). Fortunately, the *most* negative coefficient is not required for each iteration to make progress. A "sign-flipping" approximation algorithm which usually runs in polynomial time in d is detailed in Chapter 5. Using this algorithm, each iteration can efficiently identify a column of A with a negative c-value to use as the pivot.

Though it is difficult to say precisely how this negative c-value compares to the minimum possible c-value, experimental analysis reveals that the number of iterations required by the truncated tableau method is usually approximately $d^{2.5} \approx m^{1.25}$ (for $d < 52$, at least). Since there are m basis vectors, the best case scenario would require m iterations. Thus, the flipping algorithm's imprecision adds approximately at most $m^{1.25} - m$ iterations, a small price to pay for replacing a $\Omega(2^{d-1})$ operation with the cost of finding and updating the newly introduced column of A, which is a $O(d^4)$ operation.

When the basis is composed entirely of columns of A, the algorithm terminates and we can extract a solution to our initial problem.

4.2 Truncated Simplex Tableau Initialization

The construction of this initial truncated tableau, T_0, is fairly straightforward. The bulk of the tableau is made up of m columns of E, the auxiliary matrix. As before, the column vector b is the rightmost column of the tableau.

The row vector c, representing each column's reduced cost, is placed in the row below A. This is somewhat of an abstraction, as the columns of the tableau are always the basis columns, and thus have a reduced cost of 0, but it is trivial to include as a double-check that nothing has gone awry, and to maintain some of the form of the standard tableau.

The row vector i is added below c. These values are initialized to 0 as well (as they represent each column's index in $\hat{A}$, and does not apply to columns of E), but as columns of A are brought into the basis, the i-row tracks each column's index so that the final tableau can be properly interpreted.

$$T_0 = \begin{array}{ccccc} e_{1,1} & e_{1,2} & \dots & e_{1,m} & b_1 \\ e_{2,1} & e_{2,2} & \dots & e_{2,m} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ e_{m,1} & e_{m,2} & \dots & e_{m,m} & b_m \\ c_1 & c_2 & \dots & c_m & 0 \\ i_1 & i_2 & \dots & i_m & 0 \end{array} = \begin{array}{ccccc} 1 & 0 & \dots & 0 & b_1 \\ 0 & 1 & \dots & 0 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & b_m \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{array}$$

4.2.1 Removing Negative b-Values

Our implementation of the simplex method requires that the linear programming problem it solves be in a form such that there are no negative values in b. For the standard simplex algorithm, this is easily arranged as follows. If $b_i < 0$, then multiply b_i and each element in row i of A by -1. However each swap in the sign of b also changes the predictable structure of $\hat{A}$ that arises from the monotonicity structures as described in Chapter 1. This poses a challenge, because the flipping algorithm only works by exploiting our knowledge of this structure. So it is necessary to carefully "cheat" the flipping algorithm, transforming columns of A back to the unaltered values of $\hat{A}$ for certain calculations and then reversing that transformation without invaliding those calculations.

First, a global variable is defined to be a vector of size m, *flipped_rows* $= (1, 1, 1, \ldots, 1, 1, 1)^T$. This is used to keep track of which rows have flipped signs.

During the creation of the inital tableau, when a negative b_i is found and b_i and each element in row i of A are multiplied by -1, we also set *flipped_rows*$_i = -1$. Going forward, we can elementwise-multiply any column of the tableau with *flipped_rows* to swap it between this "canonical simplex form" (with no negative b-values) and "original $\hat{A}$ form" (with possible negative b-values and the predictable pattern structure).

Specifically for the flipping algorithm, the target vector is elementwise-multiplied by *flipped_rows*, putting it in original $\hat{A}$ form, and then the column of $\hat{A}$ calculated by the flipping algorithm is put in canonical simplex form before being introduced to the tableau. Empirical tests have validated the accuracy of this method, and the algorithm works equally well with b-vectors which do or do not have negative initial values.

4.3 Bringing an Inactive Vector Into the Tableau

Since we have truncated a majority of the tableau's columns, and do not update them between simplex iterations, we need some way to update those columns when they enter the active set so that they have the value that they would have had if they had been in the tableau all along. A column j of the tableau consists of a column of A, which we call $A_{*,j}$, its reduced cost c_j, and its index i_j. The index is not affected by the updating process, so it can be ignored for now.

We could simply store and repeat each individual row operation that had been done in the tableau up to that point, but the tableau operations themselves make up a majority of the algorithm's cost, and this could further increase it. As enumerated in Section 3.2, updating a column of the tableau by one step in this way requires m multiplications and additions, and one division. This makes it a $\Theta(d^2)$ operation.

Each simplex iteration would increase the number of sequential column updates by 1, making each "new" column cost more than the previous one. Thus, if there are I iterations, then the total cost of all column updates would be $\Theta((1 + 2 + \cdots + I)d^2) = \Theta(I^2 d^2)$.

As noted in Section 6.2, I is well approximated by a $\Theta(d^{2.5})$ function in the average case, which would give $\Theta(d^7)$ average-case cost. The average-case cost incurred by all other parts of the algorithm is well approximated by a $\Theta(d^{6.5})$ function, so this is quite undesireable. Additionally, the number of iterations varies based on the specific problem, so even if I did have a "good" growth rate in the average case, this implementation would exacerbate worst-case performance.

Fortunately, linear algebra gives us a simple way to implement a more efficient update procedure with a cost-per-iteration that remains constant no matter how many iterations have occured. This is because all changes made to the tableau during simplex iterations are made with elementary row operations. Each elementary row operation can be represented by matrix multiplication, so we can characterize every change to the tableau T as having been made by some update matrix U, such that the i^{th} tableau update can be represented as $U_iT_{i-1} = T_i$. Since the section of T being updated has columns with a height of $m+1$, all U matrices are $(m+1)\times(m+1)$.

Additionally, the properties of matrix multiplication mean that we can chain these updates to form a single matrix $U_iU_{i-1}\cdots U_2U_1 = G_i$ (and update it step by step, such that $G_i = U_iG_{i-1}$) such that we could take the original tableau and update it to T_i with a single matrix multiplication $G_iT_0 = T_i$. Equivalently, we can take any column of A and update it to its i^{th} version in the tableau, as if it had been part of the tableau the entire time. We call the $(m+1)\times(m+1)$ matrix G_i the "row ops" matrix for iteration i of the truncated tableau simplex method.

Each simplex iteration involves two sequential updates to the tableau: normalizing the pivot row, and then eliminating the pivot column. For the i^{th} iteration, with some pivot $p_{j,k}$ in row j and column k, the corresponding update matrix D_i to normalize the pivot row is the identity matrix with the 1 at element (k,k) replaced with $1/p_{j,k}$, and V_i, which eliminates the pivot column, is as follows, where each $a_{x,j}$ is the element in the x^{th} row of the pivot column of the current tableau:

$$V_i = \begin{matrix} 1 & 0 & \cdots & -a_{1,j} & \cdots & 0 & 0 \\ 0 & 1 & & -a_{2,j} & & & 0 \\ \vdots & & \ddots & \vdots & & & \vdots \\ & & & 1 & & & \\ \vdots & & & \vdots & \ddots & & \vdots \\ 0 & & & -a_{m,j} & & 1 & 0 \\ 0 & 0 & \cdots & -c_j & \cdots & 0 & 1 \end{matrix}$$

Now let $U_i = V_iD_i$.

This method gives another additional small bonus: we can also characterize the calculation of reduced costs c as an elementary row operation, and make that the first update to the tableau, U_0:

$$U_0 = \begin{matrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 1 & 0 \\ -1 & -1 & \cdots & -1 & 1 \end{matrix}$$

So we can skip having to perform that step separately with each new column and we can simply include it in $G_i = U_iU_{i-1}\cdots U_2U_1U_0$.

Much of this matrix multiplication process is somewhat theoretical, however. The replacement scheme of the tableau means that the $[A|E]$ section of the truncated tableau only contains the basis, which is always in the form of an identity matrix, so the tableau updates are only necessary to update the b section. Updating G is also necessary to determine the target vector which is passed to the flipping algorithm to minimize the reduced cost, as discussed in Chapter 5.

This process dominates the cost of the truncated tableau simplex algorithm. Each iteration's computation of $G_i = U_i G_{i-1}$ requires $\Theta(m^2) = \Theta(d^4)$ additions and multiplications, (assuming we don't take advantage of G_i being sparse for small i). For further details, see the full cost analysis in Section 6.1.

4.4 Truncated Simplex Tableau Iterations

In each iteration, the bit flipping algorithm is used to find a negative cost coefficient (see Chapter 5 for the details on that process). Once this negative cost coefficient is found, the column of A associated with it has its convexity constraint added to it and is multiplied by the rowops matrix G to update it in accordance with the current tableau. It then replaces a column in the truncated tableau (and, thus, in the basis). The choice of which column to replace is made by calculating the pivot row of the entering column and removing the column in the old basis with a 1 in that row. Because of this method of replacement, the column exiting the truncated tableau is also the one exiting the basis. Thus, at every step the truncated tableau contains only the basis.

Because the tableau only ever contains the basis, the only changes made by the "row operations" involved in a tableau update are to the pivot column and b-vector, which is a very efficient calculation. The majority of the calculation time of a simplex iteration is spent updating G (as described in the previous section), by repeating the row operations described in Section 3.2.

4.4.1 Anti-cycling

For certain problems, it is possible for the simplex algorithm to encounter a phenomenon known as cycling [4, Section 13.5], where the algorithm takes *degenerate steps* which do not reduce the objective function $\bar{c}^T x$, and eventually returns to an earlier basis. From there the method repeats the steps it took when it was first at the earlier basis, causing the tableau to loop through the same set of basis vectors, never making further progress.

Cycling is considered to be rare outside of relaxations of integer programming problems [4, Section 13.5] and we never encountered a problem where cycling occurred. So, in this test version of our method, we have not included an anti-cycling strategy, but we believe one could easily be added to a "production" version of our method.

4.4.2 Interpreting the Final Tableau

$$T_f = \begin{array}{ccccc} 1 & 0 & \dots & 0 & b_1 \\ 0 & 1 & \dots & 0 & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 & b_m \\ 0 & 0 & \dots & 0 & 0 \\ i_1 & i_2 & \dots & i_m & 0 \end{array}$$

This process is nearly unchanged from the full tableau simplex method. If the algorithm terminates with each basis vector associated with a column of A, then we can recover x as follows. Set $x_i = 0$ for all A-columns $A_{*,i}$ which are not in the final basis, and set $x_j = b_k$ for all A-columns $A_{*,j}$ which are in the final basis, where $A_{k,j} = 1$ in the basis vector $A_{*,j}$ (because of the form of the basis columns, this matches each x_j to a unique b_k). Then $\sum_{j=1}^{n} A_{*,j}x_j = b$. Thus, x satisfies our constrained system of linear equations $Ax = b, x \geq 0$.

4.5 Example Problem with Truncated Tableau

This example combines the features of this algorithm into one cohesive example in order to walk through a small problem in the way that it would actually be solved by the truncated tableau simplex method. The only aspect lacking from this example is that we use the actual values of the reduced costs to determine a pivot column, rather than finding one via the flipping algorithm defined in the next chapter.

First, let's establish the problem. For this example, we use A and b for which the convexity constraint has already been added to the bottom row, and the second row of A has had its signs flipped (since the original b-vector's second value is -0.3):

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 0.15 \\ 0.3 \\ 0.2 \\ 1 \end{bmatrix} \tag{4.1}$$

This particular example does not precisely match the shape of an A matrix which would actually arise from the callibration problem described in Chapter 1, but is illustrative of a typical constrained linear problem having more columns than rows, while not being too large for a simple illustrative example.

The details of the LPP that is solved in this example are discussed in Chapter 1, starting from equation (1.4). In particular, we show the process for finding a solution to the LPP (1.7) for the given A and b.

The initial tableau is segmented as such:

Initial Tableau segments:

E	A	b
c_E	c_A	0
i_E	i_A	0

The second block column, the block column in black, is comprised of columns not stored in memory. These are called "inactive" or "the inactive set", while the parts of the tableau in memory are called "active" or "the active set". While in this example the inactive set is relatively small, in actual problems this section makes up nearly all of the tableau, as the inactive set has $n = 2^{d-1}$ columns and the active set has $m = d(d-1)/2 + 1$ columns. What is shown in this example are the values which are generated if an inactive column is brought into the active set. These are the same values that they would have if they had remained loaded in the tableau since the first iteration and updated on every iteration of the full tableau simplex method described in Chapter 3. This updating process in the truncated tableau simplex method that is performed when an inactive column is brought into the active set is described in Section 4.3.

The initial tableau begins only having loaded E, which is initially the identity matrix, and b, which remains loaded throughout. To fit the format of the tableau, the matrix E has a row of c-values and a row of i-values below it, but these are always initalized to zero. The c-values of the E-columns are effectively always zero because they are initialized as zero, and there is no method by which a column of

E can be re-introduced to the basis once it has been removed (the flipping algorithm, which decides what column in the inactive set enters the basis, does not consider columns of E). They could be considered to have non-zero c-values when in the inactive set, but it would be more accurate to imagine them simply discarded entirely when they leave the active set in the first place.

Since re-introducing a column of E is never necessary to reach a solution in the phase one simplex method, the algorithm is simply not given the tools to do so. To see why this is the case, consider that every time a column of E exits the basis the simplex method could be “restarted” with the current tableau. That is, imagine a new simplex tableau initialized to be equal to the “old” tableau: its initial feasible basis would be the basis of the previous tableau, and thus consist of both columns of E and columns of A. The simplex method never requires creating new artificial variables after initialization, so all the columns of E that exited the basis in previous tableaus would be gone forever. In that way, we can see “reintroducing columns of E to the basis” as being equivalent to “creating new artificial variables after initialization”, in that neither are strictly necessary to reach a solution.

The i-values of E-columns are also all set to 0, as these values represent the index of the column in $\hat{A}$, a value which does not apply to columns of E.

Recall that the c_A-values represent their reduced cost — the amount by which they would change the cost function by being introduced to the basis. The initial calculation of this value is explained in Section 3.1. The i_A-values represent the index of each column in the $\hat{A}$ matrix, and “track” the columns as they get shuffled between the active and inactive sets.

As the algorithm progresses, a column’s location can change by being brought into and out of the active set. Notice, however, that the loaded part of the active set is always in the form of an identity matrix. This is because of the active-set replacement algorithm. When a new column enters the active set, one must exit. This exiting column is always chosen to be the one with the same pivot row as the entering column. Therefore, the row updates of each simplex iteration do not change anything in the truncated tableau except for the pivot column and b, as all other values in the pivot row of the truncated tableau are zero.

This means we could eliminate most of the truncated tableau, keeping just the i-vector and b-vector. We’ve opted not to do this, as it would not reduce the calculation cost by much (since the bulk of the truncated tableau does not need any updates), and the tableau remains a good abstraction for understanding and organizing the algorithm.

For the LPP considered in this example, the initial tableau is T_0, shown below. As the method progresses, follow the i-values between each iteration to see which columns are added to or removed from the basis.

$$T_0 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0.15 \\
0 & 1 & 0 & 0 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 0.3 \\
0 & 0 & 1 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0.2 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & -2 & 0 & -4 & -2 & 0 & 2 & -2 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 0
\end{array}$$

For the first iteration, we spell out the mathematical operations in detail; for later iterations, we just give the final result after each iteration. Note that only the sections of the tableau in blue are actually loaded into the computer memory. The arithmetic done in the black section simply represents the new values those columns would take if they were updated to the current tableau. In the truncated tableau simplex method, these updates are not actually performed until an inactive column is selected to be added to the active set. However, we perform the updates of the inactive columns on each iteration here so that the reader can more easily follow the performance of the algorithm.

$$T_{0.5} = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0.15 \\
1 - 1 & 1 & 0 & 0 & -1 - 1 & -1 - 1 & 0 - 1 & 1 - 1 & -1 + 1 & -1 + 1 & 1 + 1 & 1 + 1 & 0.3 - 0.15 \\
1 - 1 & 0 & 1 & 0 & 1 - 1 & -1 - 1 & 0 - 1 & -1 -1 & 1 + 1 & -1 + 1 & 1 + 1 & -1 + 1 & 0.2 - 0.15 \\
1 - 1 & 0 & 0 & 1 & 1 - 1 & 1 - 1 & 0 - 1 & 1 - 1 & 1 + 1 & 1 + 1 & 1 + 1 & 1 + 1 & 1 - 0.15 \\
\hline
-4 + 4 & 0 & 0 & 0 & -2 + 4 & 0 + 4 & 0 & -2 + 4 & 0 - 4 & 2 - 4 & -2 - 4 & 0 - 4 & 0 \\
\hline
3 & 0 & 0 & 0 & 1 & 2 & 0 & 4 & 5 & 6 & 7 & 8 & 0
\end{array}$$

For this half-update, column 3 of A is the pivot column, and the top row is the pivot row.

$$T_1 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0.15 \\
0 & 1 & 0 & 0 & -2 & -2 & -1 & 0 & 0 & 0 & 2 & 2 & 0.15 \\
0 & 0 & 1 & 0 & 0 & -2 & -1 & -2 & 2 & 0 & 2 & 0 & 0.05 \\
0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 2 & 2 & 2 & 2 & 0.85 \\
\hline
0 & 0 & 0 & 0 & 2 & 4 & 0 & 2 & -4 & -2 & -6 & -4 & 0 \\
\hline
3 & 0 & 0 & 0 & 1 & 2 & 0 & 4 & 5 & 6 & 7 & 8 & 0
\end{array}$$

$$T_2 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 0 & 0.5 & 0 & 0 & -1 & 0.5 & -1 & 0.175 \\
0 & 1 & 0 & 0 & -2 & 0 & 0 & 2 & -2 & 0 & -1 & 2 & 0.1 \\
0 & 0 & 1 & 0 & 0 & -1 & -0.5 & -1 & 1 & 0 & 0.5 & 0 & 0.025 \\
0 & 0 & 0 & 1 & 0 & 2 & 0 & 2 & 0 & 2 & -1 & 2 & 0.8 \\
\hline
0 & 0 & 0 & 0 & 2 & -2 & 0 & -4 & 2 & -2 & 0 & -4 & 0 \\
\hline
3 & 0 & 7 & 0 & 1 & 2 & 0 & 4 & 5 & 6 & 0 & 8 & 0
\end{array}$$

$$T_3 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 0 & 0.5 & 0 & 0 & -1 & 0.5 & -1 & 0.175 \\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0.5 & -1 & 0 & -0.5 & 1 & 0.05 \\
0 & 0 & 1 & 0 & -1 & -1 & -0.5 & 0.5 & 0 & 0 & 0 & 1 & 0.075 \\
0 & 0 & 0 & 1 & 2 & 2 & 0 & -1 & 2 & 2 & 0 & 0 & 0.7 \\
\hline
0 & 0 & 0 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 0 & 0 & 0 \\
\hline
3 & 4 & 7 & 0 & 1 & 2 & 0 & 0 & 5 & 6 & 0 & 8 & 0
\end{array}$$

$$T_4 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 1 & 0 & 0.5 & 0 & 0 & -1 & 0.5 & -1 & 0.175 \\
0 & 1 & 0 & 0 & 1 & 0 & 0.5 & 0.5 & -1 & -1 & 0 & 0 & 0.225 \\
0 & 0 & 1 & 0 & 1 & -1 & 0 & 0.5 & 0 & -1 & 0.5 & 0 & 0.25 \\
0 & 0 & 0 & 1 & -2 & 2 & -1 & -1 & 2 & 4 & -1 & 2 & 0.35 \\
\hline
0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & -2 & -4 & 0 & -2 & 0 \\
\hline
1 & 4 & 7 & 0 & 3 & 2 & 0 & 0 & 5 & 6 & 0 & 8 & 0
\end{array}$$

$$T_5 = \begin{array}{cccc|cccccccc|c}
1 & 0 & 0 & 0 & 0.5 & 0.5 & 0.25 & -0.25 & 0.5 & 0.25 & 0.25 & -0.5 & 0.2625 \\
0 & 1 & 0 & 0 & 0.5 & 0.5 & 0.25 & 0.25 & -0.5 & 0.25 & -0.25 & 0.5 & 0.3125 \\
0 & 0 & 1 & 0 & 0.5 & -0.5 & -0.25 & 0.25 & 0.5 & 0.25 & 0.25 & 0.5 & 0.3375 \\
0 & 0 & 0 & 1 & -0.5 & 0.5 & -0.25 & -0.25 & 0.5 & 0.25 & -0.25 & 0.5 & 0.0875 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 4 & 7 & 6 & 3 & 2 & 0 & 0 & 5 & 0 & 0 & 8 & 0
\end{array}$$

We know that T_5 is the final tableau, as all E-columns have been removed from the basis.

From the final tableau, we can determine the convex combination of the columns of $\hat{A}$ that equal $\hat{b}$. It is well-known from the Simplex Method that

$$\sum_{j=1}^{m} b_j \hat{A}_{*,i_j} = \hat{b}$$

where b_j is the j^{th} element in the b vector on the right side of T_5, i_j is the j^{th} element in the i vector at the bottom of T_5 and $\hat{A}_{*,i_j}$ is the column i_j in the matrix $\hat{A}$.

In our example, we can see that the i-values of the active set are 1, 4, 7, and 6 in that order. Thus, our final convex combination is as follows:

$$b_1\hat{A}_{*,1} + b_2\hat{A}_{*,4} + b_3\hat{A}_{*,7} + b_4\hat{A}_{*,6} = \hat{b}$$

That is,

$$0.2625 \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} + 0.3125 \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0.3375 \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix} + 0.0875 \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.15 \\ 0.3 \\ 0.2 \end{bmatrix}$$

Thus, constructing $\hat{b}$ as a convex combination of columns of $\hat{A}$.

Chapter 5

"Flipping" Algorithm

When using a version of the simplex method that includes a full tableau, the pivot column of each iteration is simply chosen to be the one with the most negative reduced cost, which can be determined with a call to the *min* function. However, since the non-basis columns of A are not loaded in the tableau, we need some other way to identify columns with low reduced costs. This is where the "flipping" algorithm described in this chapter comes in.

This algorithm uses the patterns between the generating vectors and their resultant vectors to search the resultant vectors efficiently. Rather than exhaustively searching through the entire set of resultant vectors, we establish simple rules that allow an algorithm to "generate" a resultant vector having a low reduced cost, while guaranteeing that the vector being calculated corresponds to an actual column of $\hat{A}$. The flipping algorithm makes use of columns of $\hat{A}$ rather than A, as the convexity constraint that distinguishes them is the same for all columns, so does not affect the choice of pivot column.

Specifically, we use the fact that for a column of $\hat{A}$ in a problem of size d (thus, a column of size $\hat{m}$), the first $d-1$ elements of that column can be any combination of "high" and "low" values. Additionally, each such combination appears in exactly one column of $\hat{A}$, and thus uniquely defines a column.

Our first concern is how the reduced cost of a column of $\hat{A}$ is calculated. As described in Section 4.3, we know that when a column of $\hat{A}$ enters the tableau it must be updated by being multiplied by the current rowops matrix G_j. This update includes the calculation of the reduced cost. A column of $\hat{A}$ is first augmented by adding a 1 as the bottom element (the convexity constraint) to put it in the form of a column of A, and then further augmented by adding to the bottom of that column of A an additional element that represents the initial reduced cost of that column of A. Since G is initalized to include this calculation of the reduced cost (as described in Section 4.3), we initalize the reduced cost to 0. We call this fully augmented but not yet updated column $(T_0)_{*,i}$.

Then, for a given tableau column $(T_j)_{*,i} = G_j(T_0)_{*,i}$, the bottom element of $(T_j)_{*,i}$ gives its reduced cost. From the rules of matrix multiplication, we know that this bottom element must equal $(G_j)_{m+1,*} \cdot (T_0)_{*,i}$, where $(G_j)_{m+1,*}$ denotes the bottom row of G_j. Thus, we do not need to update an entire column to check its reduced cost: we can simply calculate this dot product.

We could use this method to check randomly selected columns of $\hat{A}$ until we find one with a negative reduced cost, and have that column enter the basis. This is very similar to an algorithm described in [2, Section 3.3], an analysis of which is done in Section 6.2.6. The flipping algorithm makes use of information specific to the calibration problem described in Chapter 1 to achieve a reduced time-cost compared to this random selection.

While the dot product method can be done with the actual "high" and "low" values found in the columns of $\hat{A}$, the flipping algorithm makes use of an efficiency shortcut that requires these values be "rounded" to 1 and -1 respectively. No other kinds of rounding are involved in this algorithm, so going forward, when we refer to a "rounded" $\vec{a}$-vector, this is the particular method of rounding being referred to. The loss in accuracy made by this rounding is quite low in practice, and the trade-off of a faster, but slightly less accurate calculation goes in favour of this rounding (see the flipping algorithm's cost analysis in Section 6.2.1 for details).

The flipping algorithm also makes use of the bottom row of G. However, the bottom row of G is longer than a column of $\hat{A}$. Fortunately, since the bottom two values of every column of T_0 associated with a column of $\hat{A}$ are identical to each other (specifically, the bottom two elements are 1 and 0), we do not need to consider them when comparing columns of $\hat{A}$. (See the discussion around the equation for U_0 in Section 4.3 for a more detailed discussion of this.) Thus, we can truncate the two rightmost elements of the bottom row of G. We refer to the vector that results from this as the *target vector*, $\vec{t} \in \mathbb{R}^{\hat{m}}$, where $\hat{m} = d(d-1)/2$. From here, we aim to find a column of $\hat{A}$ (let's call such a column $\vec{a} \in \mathbb{R}^{\hat{m}}$) such that $\vec{t} \cdot \vec{a}$ is minimized.

While these two values truncated from $\vec{t}$ do not affect the relative value of $\vec{t} \cdot \vec{a}$ when comparing between different $\vec{a}$-vectors, they do affect the absolute value. For each index j, the true reduced cost is $c_j = \vec{t} \cdot \vec{a}_j + s$, where s is constant for a given G_i.

Because of this, it is possible for the flipping algorithm to select some index j for which $\vec{t} \cdot \vec{a}_j < 0$, but for which $\vec{t} \cdot \vec{a}_j + s \geq 0$. Or the opposite could occur: the flipping algorithm may find that the smallest $\vec{t} \cdot \vec{a_j}$ that it has found is positive, but the true reduced cost $c_j = \vec{t} \cdot \vec{a_j} + s < 0$. To address these and similar errors introduced by the rounding method mentioned earlier, the flipping algorithm double-checks its final result by recalculating the reduced cost of the column it selects. If this reduced cost is nonnegative for whatever reason, a more exact (but less efficient) algorithm called the advanced flipping algorithm is used. That algorithm is detailed in Section 5.1.

For the purposes of this algorithm, $\vec{t}$ is rearranged into an upper-triangular square matrix. We call this matrix the "gameboard" because this algorithm plays out somewhat like a board game (such as *Reversi*) where players aim to flip tiles to their colour, but do not have precise control over the flips which are done. In our case, we want to flip the signs of elements in the gameboard to make the gameboard sum as low as possible, but we can only flip entire rows and columns at once.

For a problem of size d, the way to initialize this $(d-1) \times (d-1)$ gameboard matrix is to take the first $d-1$ entries of $\vec{t}$ and place them into the main diagonal of the gameboard (with the first entry in the (1,1) position, second in the (2,2) position, etc). After that, each subsequent entry of the vector is added to the upper triangle of the matrix, left to right, proceeding from the top row to the bottom row. The lower triangle of the matrix is not used: it is just left as 0. Consider this example for a $d = 5$ problem:

$$\vec{t} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 5 & 6 & 7 \\ 0 & 2 & 8 & 9 \\ 0 & 0 & 3 & 10 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

This is essentially reversing the vectorization process of the columns of $\hat{A}$ described in Chapter 1, and this is why we can benefit by rounding $\vec{a}$ to 1s and -1s. With this rounding, we do not need to reference the values of elements of any particular $\vec{a}$. We can swap between different $\vec{a}$s simply by changing the first $d-1$ elements which, in this rounding scheme, determine the values of all remaining elements, as illustrated in the following example.

$$\vec{a} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ d \\ ab \\ ac \\ ad \\ bc \\ bd \\ cd \end{bmatrix} \longrightarrow \begin{bmatrix} a & ab & ac & ad \\ 0 & b & bc & bd \\ 0 & 0 & c & cd \\ 0 & 0 & 0 & d \end{bmatrix}$$

That is, each element in the strictly upper triangular part of the gameboard is the product of some unique pair of the first $d-1$ elements. This is due to how the correlation matrices are generated from the monotonicity structures, also explained in Chapter 1. Each pair corresponds to exactly one such element. This is elegantly represented in the gameboard form, as each main-diagonal element shares its variable with every element on its row and its column, and each strictly upper-triangular element shares one variable with the main diagonal element on its row, and its other variable with the main diagonal element on its column. Thus, when we flip the sign of a main diagonal variable, we flip the sign of all elements on its row and column.

Therefore, we can calculate $\vec{t} \cdot \vec{a}$ as follows.

$$\vec{t}\cdot\vec{a} = \begin{bmatrix}1\\2\\3\\4\\5\\6\\7\\8\\9\\10\end{bmatrix}\cdot\begin{bmatrix}a\\b\\c\\d\\e\\f\\g\\h\\i\\j\end{bmatrix} = sum\begin{bmatrix}1a & 5ab & 6ac & 7ad\\ 0 & 2b & 8bc & 9bd\\ 0 & 0 & 3c & 10cd\\ 0 & 0 & 0 & 4d\end{bmatrix}$$

where *sum* is the sum of all the elements in the gameboard and each of a, b, c, d is either $+1$ or -1.

Now we have our gameboard. If we were to take its sum right now with each a, b, c, d equal to $+1$, that would be the dot product of $\vec{t}$ with the $\vec{a}$ vector of all 1s (for every d-value, the $\hat{A}$-column of index n is always $\vec{a}_n = (1, 1, \ldots, 1, 1)$, so we know we are starting from a legitimate $\vec{a}$).

From this construction, we can see what changes we can make to the gameboard while keeping $\vec{a}$ a valid column of A: any variable along the main diagonal can be flipped (a, b, c, or d in this example), which flips all values on its row and column, as these elements (and only these elements) are the ones which share that variable.

Consider again this example gameboard:

$$\begin{bmatrix}1 & 5 & 6 & 7\\ 0 & 2 & 8 & 9\\ 0 & 0 & 3 & 10\\ 0 & 0 & 0 & 4\end{bmatrix}$$

In the first step of the flipping algorithm, the following candidate flips would be calculated:

$$\begin{bmatrix}-1 & -5 & -6 & -7\\ 0 & 2 & 8 & 9\\ 0 & 0 & 3 & 10\\ 0 & 0 & 0 & 4\end{bmatrix} = 17, \begin{bmatrix}1 & -5 & 6 & 7\\ 0 & -2 & -8 & -9\\ 0 & 0 & 3 & 10\\ 0 & 0 & 0 & 4\end{bmatrix} = 7, \begin{bmatrix}1 & 5 & -6 & 7\\ 0 & 2 & -8 & 9\\ 0 & 0 & -3 & -10\\ 0 & 0 & 0 & 4\end{bmatrix} = 1, \begin{bmatrix}1 & 5 & 6 & -7\\ 0 & 2 & 8 & -9\\ 0 & 0 & 3 & -10\\ 0 & 0 & 0 & -4\end{bmatrix} = -5$$

In this instance, the flip on the $(4, 4)$ element would be selected due to it resulting in the minimum gameboard sum, the matrix generated by that flip would become the new gameboard, and the flipping algorithm would proceed from there.

With these tools in hand, the algorithm works like this:

1. Establish a starting vector. While the vector of all 1s is a legitimate starting vector, as previously mentioned, there is a quick shortcut we can take for a small increase in efficiency. Since the first $d-1$ elements of $\vec{a}$ can be flipped arbitrarily, we can simply flip them such that all elements on the main diagonal of the gameboard are negative. This gives a small but measureable increase over

starting naïvely from the vector of all 1s.

2. Calculate the results of potentially swapping each main-diagonal element, in terms of the change to the gameboard sum.

3. Compare each of these results. Whichever one lowers the gameboard sum the most is applied to the gameboard, repeat step 2 from the updated gameboard. If no potential flips lower the gameboard sum, proceed to step 4.

4. The current gameboard is considered the "final gameboard" and the flipping algorithm terminates. The column of $\hat{A}$ associated with the final gameboard is then determined based on which flips have been done (this calculation is discussed after the example problem below).

After step 4, the true reduced cost of the column of $\hat{A}$ associated with the final gameboard is calculated to be $c_i = (G_j)_{m+1,*} \cdot (T_0)_{*,i}$, where $(G_j)_{m+1,*}$ denotes the bottom row of G_j and $(T_0)_{*,i}$ denotes the augmented tableau column associated with the i^{th} column of $\hat{A}$. If this true reduced cost is nonnegative, we proceed to the advanced flipping algorithm, which is described in the next section. If this true reduced cost is negative, the index of the $\vec{a}$ determined by the current gameboard is identified, and that $\vec{a}$ is updated with G, and inserted into the tableau.

Let's see a complete example of the flipping algorithm in action. For this example, each flipped element is shown in red, and the matrix selected in step 3 has its sum shown in green.

The original target vector and associated gameboard for this example associated with a problem of size $d = 6$ is shown below. This particular problem was selected to take a few flipping iterations to solve (for problems of size $d = 6$, usually only two flips are needed) to properly show the flipping algorithm in action. For this reason, we skip the shortcut mentioned in step 1, and instead show all flips starting from $\vec{a} = (1, 1, \ldots, 1, 1)^T$.

$$\vec{t} = \begin{bmatrix} 8 \\ 6 \\ 10 \\ 5 \\ 10 \\ 2 \\ -11 \\ 1 \\ 4 \\ 7 \\ -13 \\ 4 \\ 4 \\ 1 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 2 & -11 & 1 & 4 \\ 0 & 6 & 7 & -13 & 4 \\ 0 & 0 & 10 & 4 & 1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}$$

Note that the "flip" function shown in this example shows the difference between the given matrix and the original matrix it was "flipped" from, rather than the full sum of that matrix. Though comparing total sums does bring us through the same series of flips, this comparison means we can easily distinguish between increases and decreases in our total sum, and terminate when every possible flip fails to decrease the sum.

First flip:

$$flip(\begin{bmatrix} -8 & -2 & 11 & -1 & -4 \\ 0 & 6 & 7 & -13 & 4 \\ 0 & 0 & 10 & 4 & 1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = -8$$

$$flip(\begin{bmatrix} 8 & -2 & -11 & 1 & 4 \\ 0 & -6 & -7 & 13 & -4 \\ 0 & 0 & 10 & 4 & 1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = -12$$

$$flip(\begin{bmatrix} 8 & 2 & 11 & 1 & 4 \\ 0 & 6 & -7 & -13 & 4 \\ 0 & 0 & -10 & -4 & -1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = -22$$

$$flip(\begin{bmatrix} 8 & 2 & -11 & -1 & 4 \\ 0 & 6 & 7 & 13 & 4 \\ 0 & 0 & 10 & -4 & 1 \\ 0 & 0 & 0 & -5 & -6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = -6$$

$$flip(\begin{bmatrix} 8 & 2 & -11 & 1 & -4 \\ 0 & 6 & 7 & -13 & -4 \\ 0 & 0 & 10 & 4 & -1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = -50$$

Second flip:

$$flip(\begin{bmatrix} -8 & -2 & 11 & -1 & 4 \\ 0 & 6 & 7 & -13 & -4 \\ 0 & 0 & 10 & 4 & -1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 8$$

$$flip(\begin{bmatrix} 8 & -2 & -11 & 1 & -4 \\ 0 & -6 & -7 & 13 & 4 \\ 0 & 0 & 10 & 4 & -1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 4$$

$$flip(\begin{bmatrix} 8 & 2 & 11 & 1 & -4 \\ 0 & 6 & -7 & -13 & -4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = -18$$

$$flip(\begin{bmatrix} 8 & 2 & -11 & -1 & -4 \\ 0 & 6 & 7 & 13 & -4 \\ 0 & 0 & 10 & -4 & -1 \\ 0 & 0 & 0 & -5 & 6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 18$$

$$flip(\begin{bmatrix} 8 & 2 & -11 & 1 & 4 \\ 0 & 6 & 7 & -13 & 4 \\ 0 & 0 & 10 & 4 & 1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = 50$$

Third flip:

$$flip(\begin{bmatrix} -8 & -2 & -11 & -1 & 4 \\ 0 & 6 & -7 & -13 & -4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = -36$$

$$flip(\begin{bmatrix} 8 & -2 & 11 & 1 & -4 \\ 0 & -6 & 7 & 13 & 4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 32$$

$$flip(\begin{bmatrix} 8 & 2 & -11 & 1 & -4 \\ 0 & 6 & 7 & -13 & -4 \\ 0 & 0 & 10 & 4 & -1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 18$$

$$flip(\begin{bmatrix} 8 & 2 & 11 & -1 & -4 \\ 0 & 6 & -7 & 13 & -4 \\ 0 & 0 & -10 & 4 & 1 \\ 0 & 0 & 0 & -5 & 6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 34$$

$$flip(\begin{bmatrix} 8 & 2 & 11 & 1 & 4 \\ 0 & 6 & -7 & -13 & 4 \\ 0 & 0 & -10 & -4 & -1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = 46$$

Fourth flip:

$$flip(\begin{bmatrix} 8 & 2 & 11 & 1 & -4 \\ 0 & 6 & -7 & -13 & -4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 36$$

$$flip(\begin{bmatrix} -8 & 2 & -11 & -1 & 4 \\ 0 & -6 & 7 & 13 & 4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 40$$

$$flip(\begin{bmatrix} -8 & -2 & 11 & -1 & 4 \\ 0 & 6 & 7 & -13 & -4 \\ 0 & 0 & 10 & 4 & -1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 62$$

$$flip(\begin{bmatrix} -8 & -2 & -11 & 1 & 4 \\ 0 & 6 & -7 & 13 & -4 \\ 0 & 0 & -10 & 4 & 1 \\ 0 & 0 & 0 & -5 & 6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = 38$$

$$flip(\begin{bmatrix} -8 & -2 & -11 & -1 & -4 \\ 0 & 6 & -7 & -13 & 4 \\ 0 & 0 & -10 & -4 & -1 \\ 0 & 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}) = 30$$

This gives a final matrix, and thus

$$sum(\begin{bmatrix} -8 & -2 & -11 & -1 & 4 \\ 0 & 6 & -7 & -13 & -4 \\ 0 & 0 & -10 & -4 & 1 \\ 0 & 0 & 0 & 5 & -6 \\ 0 & 0 & 0 & 0 & -10 \end{bmatrix}) = -60$$

For this particular problem, this happens to be the optimal solution.

Once the flipping algorithm terminates, we need to determine the index of the column of $\hat{A}$ that the final gameboard corresponds to. Because of the way the $\hat{A}$-columns are arranged, there is a simple algorithm to convert a set of flips to the index. For this example, the first, third, and fifth elements on the main diagonal were flipped, so the corresponding elements of $\vec{a}$ must be flipped from its original state (of all 1s): $[-1, 1, -1, 1, -1]^T$. If we treat the -1s as 0s and read it as a binary number, we get an index of 10. We can do this because the arrangement of $\hat{A}$'s columns is arbitrary, so any consistent bijection between indices and columns is valid. This method creates a bijection because for any value of d, in the associated $\hat{A}$ matrix there exists exactly one column of $\hat{A}$ for each combination of high and low values for the first $d - 1$ elements.

After the index is determined, the associated column of the tableau is updated with G to calculate its true reduced cost. As mentioned earlier, if the true reduced cost is negative, then the associated column of A is selected to be inserted into the basis. If the true reduced cost is nonnegative, then the result is not used and the advanced flipping algorithm attempts to find a column of $\hat{A}$ with negative reduced cost.

5.1 Advanced Flipping Algorithm

In rare cases, the flipping algorithm does not find a column of $\hat{A}$ with negative reduced cost. This can occur for various reasons discussed in the previous section. When this happens, the flipping algorithm's result is thrown out and the advanced flipping algorithm (AFA) is used instead. The AFA is a replacement of the flipping algorithm which works on a similar principle, but does away with the flipping algorithm's shortcuts. Because of this, it runs slightly slower in the average case, and has exponential-time worst case performance. However, the advanced flipping algorithm is guaranteed to find a solution if one exists. It is known that the simplex algorithm, while usually efficient, has poor worst-case performance, and the AFA is where that inevitable risk of poor performance resurfaces in the truncated tableau simplex method.

In the worst case scenario, the AFA requires $\Omega(2^{d-1})$ arithmetic operations, essentially searching through each column of $\hat{A}$ one by one for a negative reduced cost. Thankfully, our experience and tests suggest that the circumstances for exponential-time performance are extremely rare. In all the tests run in the course of developing this algorithm and testing its time-costs on randomized problems, this did not occur even once.

Conceptually, the AFA works similarly to the original flipping algorithm: both compare all $\vec{a}$-vectors "adjacent" to a candidate vector, make the one with the greatest reduction to the reduced cost the new candidate vector, and repeat until no more improvements can be found. The differences between the two are in which "adjacent" vectors are compared, and how reduced costs are calculated.

The AFA does away with the flipping algorithm's 1/-1 approximation to each element of the resultant vectors. Instead of using a gameboard, each dot product is calculated individually. This ensures that no effective candidates are being lost by the rounding. This calculation also includes the second-to-last element of the bottom row of G which is truncated in the flipping algorithm, so the reduced costs calculated by the AFA are the true values (the last element would only be multiplied by 0 in the dot product for every $\vec{a}$, so can still be safely truncated).

Aside from these changes, the AFA proceeds similarly to the flipping algorithm until it finds a candidate vector for which no adjacent vector will reduce the reduced cost. If that candidate vector's reduced cost is negative, the AFA terminates and that vector is selected to be inserted into the basis. However, if that candidate vector's reduced cost is nonnegative, the AFA considers all vectors which are two flips away from the current candidate vector. If this fails, it considers three simultaneous flips, and so, until a reduction is found. Once that reduction is found, the AFA proceeds with single flips as before.

Putting those differences together, we can describe the advanced flipping algorithm step-by-step. Before beginning, we initialize an "adjacency value" $f = 1$, which is referenced in step 2 and defines which vectors should be compared at that step. Since the first $d-1$ elements of a resultant vector correspond to the main diagonal of the flipping algorithm gameboard, step 2 is very similar to the flipping algorithm when $f = 1$.

1. Establish a starting vector. As in the flipping algorithm, the vector of all 1s is a legitimate starting vector, but the same shortcut mentioned there can also be used here.

2. Calculate the reduced cost of all resultant vectors which have the sign of all but f of their first $d-1$ elements in common.

3. Compare each of these results. Whichever adjacent vector lowers the reduced cost the most is applied to the gameboard, and then step 2 is repeated from the updated board and f is set to 1. If no adjacent vectors reduce the reduced cost, proceed to step 4.

4. If the reduced cost of the current candidate vector is nonnegative, increment f and repeat step 2. Otherwise, update the candidate vector and select it to be inserted into the basis.

Since the AFA works directly with the true values of columns of $\hat{A}$ and their indices, the algorithm will already know the true reduced cost and the index, and the flipping algorithm's recalculation of these values is not necessary.

Considering all possible trios of flips, or potentially even larger sets, is an expensive calculation. For a problem of size d, calculating the reduced cost of all vectors which are f flips "away" from the candidate vector requires evaluating $\binom{d-1}{f}$ dot products. Fortunately, our empirical results indicate that more than two simultaneous flips are only necessary so rarely that the possibility cannot be effectively analyzed from our numerical results, and it is even rarer that it requires so many simultaneous flips as to provide a nontrivial increase in the truncated tableau simplex method's total runtime. Regardless of the low risk, these potential costs of the advanced flipping algorithm are analyzed in Sections 6.2.2 and 6.2.4.

Chapter 6

Cost Analysis

All tests were done on a desktop computer with an Intel i7-6700k CPU running at 4GHz. The Scilab *timer()* function was used, which measures time by counting CPU cycles used by the program, rather than real time, making it more consistent for measuring tests (for instance, a software update running in the background does not affect results). This means that all results are consistent relative to each other, but may not be consistent with real time.

6.1 Analysis of Constant Costs

The precise amount of time and iterations required by the program varies depending on the problem being tackled. However, in each iteration the updating of the tableau and rowops matrix is constant for a given d.

Recall that the truncated tableau is comprised of an $m \times m$ square matrix, which is a subset of A and E, as well as vectors b and c, each of length m, and that rowops (the matrix which is the product of our elementary row operations) is an $(m+1) \times (m+1)$ matrix. In each iteration, the tableau update requires normalizing the pivot row (this requires one division operation, to update the b-value on the pivot row), then subtracting a dynamic product from all-but-one element of both the b-column and the pivot column. However, all operations on the pivot column can be skipped, because they necessarily result in a column of the form $[0, 0, \ldots, 1, \ldots, 0, 0]^T$. This gives a total cost of $\hat{m}$ subtractions and $\hat{m}+1$ multiplications/divisions. There is no cost to "updating" the active set section of the truncated tableau, as it begins and ends each iteration in the form of an identity matrix.

The update to the rowops matrix is far more costly. In each iteration, this costs m division operations: one for each element in the pivot row of rowops. However, the majority of the calculation time comes from the multiplications in the following step. Every element of rowops requires that a new product be calculated and then subtracted from it — except for the pivot row (which has already been adjusted) and the pivot column (since the subtraction would be multiplied by the pivot of 1, that multiplication is skipped). Thus, the rowops update requires m^2 multiplications, which dwarfs all previously mentioned calculation costs. This gives a time complexity of $\Theta(m^2) = \Theta(d^4)$ per iteration. In terms of n, the width of $\hat{A}$, the growth is polylogarithmic: $\Theta(\log_2(n)^4)$. Each of these multiplications accompanies a subtraction, in addition to the subtractions involved in updating the pivot column, so there are also $\Theta(d^4)$ addition/subtraction operations.

To update the full tableau would normally have a time complexity of $\Theta(mn) = \Theta(d^2 \times 2^d)$, so the truncated tableau simplex method provdes a great time-cost reduction over the full tableau simplex method.

Thus, the lower bound cost of this algorithm is $\Omega(d^4)$ operations per iteration. To find the true cost per iteration, it is necessary to determine if calculations from the flipping algorithm or advanced flipping algorithm could exceed an $O(d^4)$ time complexity. We find that this excessive cost may occur in the worst case with the advanced flipping algorithm, but, in all the tests we ran, the average time complexity is indeed $\Theta(d^4)$ per simplex iteration. Later in this chapter we examine the details of these other potential costs. The other factor in the total cost will be the number of simplex iterations, which we determine empirically in the next section.

6.2 Counting Iterations

The following table gives the average of the results of ten problems for each d-value from $d = 10$ to $d = 52$. While the number of iterations required is obviously an integer value for a given problem, the iterations and flips are noted to one decimal place as they represent the average of ten test problems. For these test problems, maximum correlation values are chosen as random uniform values in [0.9,1) and minimum correlation values are chosen as uniform random values in the range (-1,-0.9]. Smaller tests with other ranges (with uniform random numbers in ranges as wide as (-1,-0.7] and [0.7,1) as stand-ins for maximum and minimum correlations, which is a wider range than would be expected in actual data) have produced very similar results. Therefore the results of this sort of test can be reasonably extrapolated to these types of problems as well.

d	time	Iterations	time/iter	flips	flips/iter
10	0.56	64.4	0.0087	229.1	3.56
11	0.72	82.3	0.0087	322.1	3.91
12	0.92	103.1	0.0089	436.6	4.23
13	1.15	128.2	0.0090	597.5	4.66
14	1.45	154.1	0.0094	773.4	5.02
15	1.95	181.7	0.0107	974.0	5.36
16	2.36	224.1	0.0105	1271.3	5.67
17	2.58	245.6	0.0105	1478.7	6.02
18	3.69	275.8	0.0134	1730.7	6.28
19	4.13	328.2	0.0126	2206.5	6.72
20	4.59	358.5	0.0128	2507.8	7.00
21	5.62	418.2	0.0134	3065.7	7.33
22	6.57	460.7	0.0143	3584.6	7.78
23	8.41	520.0	0.0162	4125.1	7.93
24	8.81	579.8	0.0152	4859.7	8.38
25	10.07	627.3	0.0160	5469.9	8.72
26	12.66	700.5	0.0181	6329.7	9.04
27	14.83	793.4	0.0187	7498.3	9.45
28	17.47	866.3	0.0202	8428.0	9.73
29	19.63	926.3	0.0212	9381.1	10.13
30	24.25	1023.7	0.0237	10718.7	10.47
31	26.30	1090.4	0.0241	11749.5	10.78
32	30.17	1194.7	0.0253	13223.4	11.07
33	34.24	1265.3	0.0271	14622.4	11.56
34	42.59	1439.2	0.0296	17032	11.83
35	47.61	1514.8	0.0314	18493.8	12.21
36	50.29	1622.1	0.0310	20335.4	12.54
37	58.95	1759.1	0.0335	22692.8	12.90
38	67.31	1829.1	0.0368	24079.9	13.16
39	79.12	1989.9	0.0398	27028.5	13.58
40	91.27	2122.4	0.0430	29654.0	13.97
41	101.24	2280.6	0.0444	32694.6	14.34
42	123.35	2394.9	0.0515	35268.2	14.73
43	142.14	2519.3	0.0564	37870.1	15.03
44	149.53	2650.7	0.0564	40863.7	15.42
45	180.98	2823.5	0.0641	44462.7	15.75
46	213.54	3022.0	0.0707	48592.6	16.08
47	265.62	3127.0	0.0849	51612.9	16.51
48	287.54	3338.9	0.0861	56190.8	16.83
49	348.91	3518.8	0.0992	60552.8	17.21
50	386.07	3715.6	0.1039	65091.1	17.52
51	397.99	3934.7	0.1011	70735.8	17.98
52	467.91	4124.5	0.1134	75801.9	18.38

From these numbers, the function that best predicts the number of simplex iterations is $iter \approx d^{2.5}/4.8$, giving $\Theta(d^{2.5})$ iterations (see Figure 6.1 (b)). Though the times for smaller d-values are a little erratic (likely due to minor constant costs dominating the total calculation time), as d increases the total times approach the predicted value $t \approx d^{6.5}/(2.95 \times 10^8)$ CPU-seconds, giving a total cost which grows polynomially in d and polylogarithmically in n: $\Theta(d^{6.5}) = \Theta(\log_2(n)^{6.5})$ (see Figure 6.1 (a)).

The "flips" and "flips/iter" headers refer to the number of "flipping iterations" (where one flipping iteration includes the calculation of the value of each potential flip from the current $\vec{a}$, and taking the most negative value as the new candidate $\vec{a}$), as opposed to the actual number of candidate flips that were checked (each flipping iteration considers $d-1$ candidate flips, so the total number of candidate flips is flips$\times(d-1)$).

The number of flipping iterations per simplex iteration clearly grows linearly with d, as $flips/iter \approx 0.35 * d$ is a strong approximation for all values in that column (see Figure 6.1 (c)). Since each flipping iteration represents $d-1$ candidate flips we can estimate that, out of n-columns of A, each call to the flipping algorithm checks approximately $0.35d^2 \approx 0.35(\log_2 n)^2$ columns of A (with some potential duplicates).

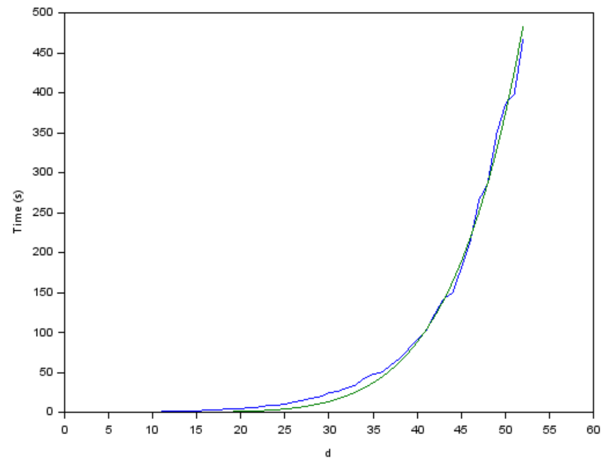
6.2.1 Flipping Algorithm Costs

In each flipping iteration, $d-1$ possible flips are explored. Calculating the results of each possible flip requires d additions, followed by d comparisons to find the most negative option. One such flip is implemented per iteration and each implemented flip requires d additional sign-flips. Thus, each iteration of the flipping algorithm requires $\Theta(d^2)$ additions and $\Theta(d)$ sign-flips. Since it runs $\approx 0.35 * d$ times per simplex iteration (see Figure 6.1 (c)), the cost per simplex iteration of the flipping algorithm is $O(d^3)$ additions and $O(d^2)$ sign-flips. This cost is dwarfed by the $\Theta(d^4)$ cost of the rowops update at each iteration, and indeed in the empirical data we see that the total time spent in the flipping algorithm is relatively small. It's likely the algorithm could be further optimized, but as it is already so cheap this would provide only a minor overall improvement to the truncated tableau simplex method.

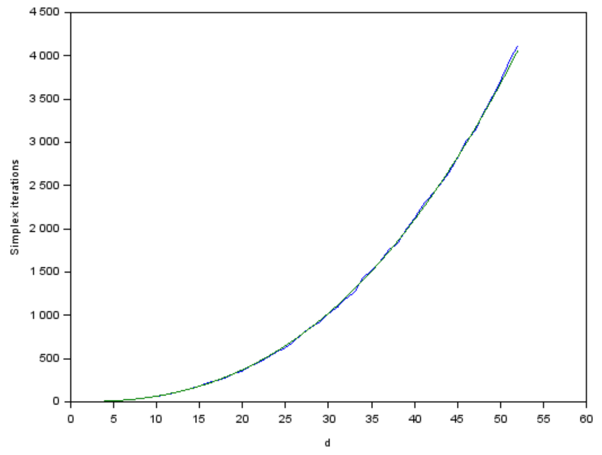
6.2.2 Advanced Flipping Algorithm Costs

The advanced flipping algorithm is rarely used, so it is difficult to gather data about its performance, or make specific claims about how often it is needed. For instance, 500 random test problems (at sizes $d = 10, 30, 50$ were run, and the advanced flipping algorithm was not required to solve even one of them. Because of this, the cost analysis should be seen as being worst-case, rather than significiantly impacting the average case.

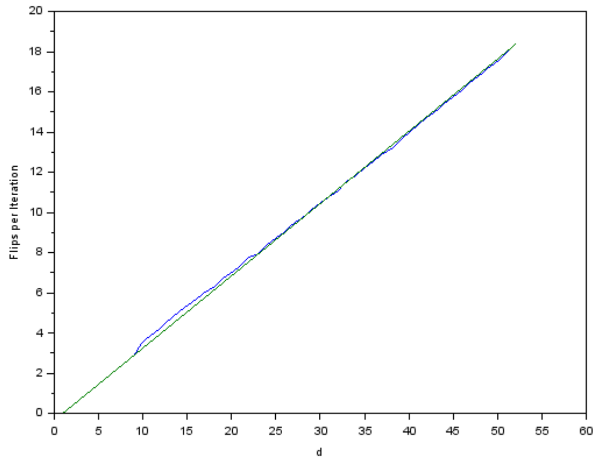
The cost of each use of the advanced flipping algorithm iteration depends on the greatest number of simultaneous flips that are needed. Each dot product costs m multiplications and $\hat{m}$ additions, and is thus a $\Theta(d^2)$ operation. For an iteration calculating f simultaneous flips, the number of dot products calculated is d-choose-f, equivalent to $\Theta(d^f)$ for $f \leq d/2$, giving a total cost for the advanced flipping algorithm of $\Theta(d^{f+2})$, where f is the greatest number of simultaneous flips explored. The further implications of this are explored in Section 6.2.4.



(a) Time



(b) Simplex iterations



(c) Flips per simplex iteration

Figure 6.1: plots of cost estimation functions (green) compared to the actual data (blue).

6.2.3 Flipping Algorithm Variant Comparisons

This table shows the results of a test which compared the total number of simplex iterations required to solve five randomly generated problems (for each given d-value) using the rounded flipping algorithm, the exact-value flipping algorithm, and the random selection algorithm given in the description of the revised simplex method in [2, Section 3.3].

d	flip (rounded)	flip (exact)	random
3	5.2	5.0	5.0
4	8.4	8.4	12.4
5	13.0	12.4	15.6
6	20.2	19.8	41.0
7	28.0	29.6	62.8
8	37.6	38.0	83.4
9	52.8	51.8	139.8
10	62.0	65.4	188.2
11	84.4	79.6	274.6
12	101.0	106.6	368.6
13	125.0	123.2	424.8
14	152.6	154.2	568.6
15	165.6	195.2	713.8
16	195.6	213.8	950.6
17	231.2	263.6	1091.0
18	288.6	285.0	1300.2
19	323.4	332.8	1434.2
20	374.2	365.2	1683.8
21	412.0	412.6	2174.2
22	469.8	454.0	2392.2
23	515.6	531.2	2835.0
24	556.0	586.2	3071.0
25	646.8	647.6	3581.0
26	723.8	723.6	4120.0
27	781.0	781.2	4626.8
28	858.6	854.0	5072.8
29	963.8	980.8	5904.4
30	1001.8	1055.4	6137.6

We can see here that the rounding done by the flipping algorithm has a negligible effect on the number of simplex iterations required compared to using the exact values, sometimes even showing superior performance. Conversely, the random-selection algorithm is clearly inferior to both versions of the flipping algorithm.

The rounded flipping algorithm is used because it requires slightly less computation and is simpler to implement.

6.2.4 Total Costs and Empirical Comparison

Clearly, in the average case the dominating factor is the updating of the rowops matrix, with $\Theta(d^4)$ multiplications per simplex iteration, and $\Theta(d^{2.5})$ simplex iterations, for a total of $\Theta(d^{6.5})$ operations. This aligns with the empirical data of the average runtimes, and in terms of n, the number of columns in the $\hat{A}$ matrix, this is $\Theta(\log(n)^{6.5})$. Since the simplex algorithm's average case performance is normally polynomial-in-n, this is a big improvement.

In the worst-case, the running time of the algorithm can deteriorate significantly because of the excessive run time potentially required by the advanced flipping algorithm. If a single iteration requires a search through all resultant vectors in the space of four simultaneous flips, this would cost $\Theta(d^6)$ multiplications, and if more than four simultaneous flips are required, this would increase further and push the total cost to $\Omega(d^7)$ or higher.

The absolute worst case scenario (being that the only resultant vector with a more negative cost than the current candidate vector requires all of the $d-1$ possible flips to identify) would require a dot product calculation for every single resultant vector, meaning $d^2 \times 2^{d-1}$ total multiplications — an obviously unfeasible quantity, except for small values of d. However, nothing near these worst-case scenarios has ever occured in testing for problems which have solutions. Though, if there is no x such that $Ax = b, x \geq 0$, then this can occur. Depending on the specific implementation and what problems are being solved, it may be necessary to implement safety measures to ensure only problems which have valid solutions are attempted or to abort the computation if a resultant vector with a negative reduced cost cannot be found in a reasonable amount of time.

6.2.5 Full Tableau Cost Estimations And Comparisons

How much of an improvement is the truncated tableau simplex method over the full tableau simplex method? The relevant differences are that in the full tableau simplex method the full tableau is kept in memory and updated on each iteration as described in Chapter 3, the flipping method is replaced with a (linear time) call to the function *min()*, and the size of the tableau is $(m+1) \times (m+n+1)$.

Since each element in the A section of the tableau is changed by a multiplication in each simplex iteration, there are $\Theta(mn)$ multiplications per iteration. Since there are m rows and no more than one A-column can enter the basis per iteration, there must be at least m simplex iterations. Thus, the total number of multiplications is at least $m^2n \approx d^4/4 \times 2^{d-1} = d^4 \times 2^{d-3}$.

Based on our table of empirical results, the calculation time of the truncated tableau simplex method is dominated by the multiplications. With the average of $d^{2.5}/4.8$ simplex iterations, the number of multiplications for the rowops matrix is $\approx d^4/4 \times d^{2.5}/4.8 \approx d^{6.5}/20$ and the total calculation time is approximately $\approx d^{6.5}/(2.95 \times 10^8)$ seconds. Based on this, we can extrapolate that (on the test system, a desktop computer with a 4GHz Intel i7-6700k processor) each multiplication costs approximately 5×10^{-10} seconds.

Thus, the time to complete a problem of size d using the full tableau simplex method can be approximated as $t \approx d^4 \times 2^{d-3} \times 5 \times 10^{-10}$ seconds. We can then estimate that, with the full tableau simplex method, this type of problem would take over an hour at $d = 27$, over a day at $d = 31$, and over a year at $d = 38$. At just $d = 67$, it would take longer than 14 billion years, the age of the universe.

6.2.6 Revised Simplex Cost Comparison

The revised simplex method and its time and space complexity are described in [2, Section 3.3]. It claims to have $O(m^2)$ space complexity and (best-case) $O(m^2)$ time complexity per iteration. If this were true, it would make our method quite redundant. However, these figures rely on some assumptions which do not bear out in practice for our particular problem.

Firstly, it bases its $O(m^2)$ space complexity on the assumption of a very sparse A-matrix, as that is typical for many large linear programming problems. Though some other sort of compression might be possible (as the A matrix has only $d(d-1)$ distinct values, from the maximum and minimum correlations), our A matrix typically has no 0-entries, and so is obviously not sparse.

If we implement space-complexity saving methods of the truncated tableau simplex method alongside the other parts of the revised simplex algorithm (for the sake of testing them), we find that the polynomial best-case time complexity does bear out — though it exhibits worse performance than our method. This is mainly because it lacks a method for finding an "effective" reduced cost coefficient, instead choosing at random until finding a negative one. While this is usually slightly more time-efficient per simplex iteration than the flipping method, the flipping method's cost is already a very small part of the overall costs, and the random method increases the number of simplex iterations from $\approx d^{2.5}/4.8$ to $\approx d^3/4$. For instance, this formula and our empirical results indicate that this change results in an 8-fold time-cost increase at $d = 50$, increasing further as d increases.

6.2.7 Truncated Tableau Time Extrapolation

Past $d = 52$, we reach the limits of integer precision on Scilab's default settings, which limits us from being able to uniquely index the 2^{d-1} columns of A in a straightforward way (as well as approaching sizes at which it's unfeasible to run a large number of tests to analyse in a short span of time), but aside from that impediment we see no reason why this algorithm would have disproportionate trouble with greater values of d. From the empirical data, we know that for an average-case d-size problem, the time to solve it is given by the function $t \approx d^{6.5}/(2.95 \times 10^8)$, in CPU-seconds on the test system.

Extrapolating this, problems would take over an hour at $d = 71$, over a day at $d = 115$, and over a year at $d = 285$, on hardware equivalent to the test system.

6.2.8 Space Costs

Even in the worst case, the space required by the truncated tableau simplex algorithm is $\Theta(d^4)$, most of which is required by the $(m+1) \times (m+1)$ rowops matrix and the $(m+2) \times (m+1)$ active part of the tableau. The identity matrix representing the basis vectors of the active part of the tableau is $m \times m$. If space were at a premium, the identity matrix could stored in a sparse format or removed altogether and just used implicitly, cutting total space requirements approximately in half.

All other necessary storage is also of static size, and much smaller than $\Theta(d^4)$: the gameboard is a $d-1 \times d-1$ matrix, a few m-length vectors are stored long term (such as *flipped_rows*), and no more than a few temporary ones are in use simultaneously.

In its current state the program has some functions that collect diagnostic information, which do grow in space-cost at a constant rate as the algorithm progresses, but none of these are required for the program to function, and if space is a concern they can be easily disabled.

6.3 An Exact, Polynomial-time Flipping Algorithm

The flipping algorithm that we propose in Chapter 5 is very effective and runs in polynomial time in d, but it is not guaranteed to find the A-column with most-negative reduced cost. This might lead one to wonder if it is possible to precisely determine the most negative reduced cost in a time polynomial in d. If it is, it would be possible to precisely replicate the steps of the standard simplex algorithm in logarithmic-in-n time.

To investigate this, let's define the problem faced: given a vector $\vec{v} = \{v_1, v_2, \ldots, v_m\} \in \mathbb{R}^m$, we must choose a vector $x = \{x_1, x_2, \ldots, x_{d-1}\}$, where each $x_i \in \{-1, 1\}$ such that the following expression is maximized:

$$x_1v_1 + x_2v_2 + \cdots + x_{d-1}v_{d-1} + (x_1x_2)v_d + (x_1x_3)v_{d+1} + \cdots + (x_{d-2}x_{d-1})v_m \tag{6.1}$$

This ignores the additional complication of the true values of the $\hat{A}$-columns not being 1/-1, but for now we investigate this simpler case.

This can be seen as a specific version of the satisfiability problem, one which would be called "weighted monotone MAX-2-XOR-SAT". That is, a satisfiability problem where a conjunction of clauses is given, and each clause has some weight and at most two literals (which cannot be negations) linked by an XOR operation, and the objective is to maximize the value of the weights (where a true clause adds its weight to the sum and a false clause does not).

The investigation of this problem is beyond the scope of this thesis, and there has not been much literature on this very specific instance of the satisfiability problem, but one comment on StackExchange [5] gives a reduction of MAX-CUT to (unweighted, non-monotone) MAX-2-XOR-SAT. Since MAX-CUT is NP-hard, if this reduction is correct, then MAX-2-XOR-SAT is NP-hard, and therefore the (strictly more difficult) weighted MAX-2-XOR-SAT, is also NP-hard.

Additionally, we are not aware of any NP-hard satisfiability problems where the addition of a monotonicity constraint renders the problem solvable in polynomial time. Thus, while it is beyond the scope of this research paper to prove it, it seems very likely that the problem faced by the flipping algorithm is NP-hard, and a polynomial approximation is most fitting for our purposes.

Chapter 7

Conclusion

The algorithm presented in this paper provides great time- and space-complexity improvements over the full tableau simplex method for solving the constrained linear problem $Ax = b, x \geq 0$, that arises from the calibration problem described in Chapter 1, reducing it from exponential complexity to polynomial time in d for all the test problems we have run. This is accomplished by exploiting patterns in the way correlation matrices are generated.

Further improvements in running time may be possible, as both the rowops matrix update and flipping algorithm are highly parallelizable.

This greatly expands the size of calibration problems which can be solved in reasonable timeframes, and improves the speed at which smaller problems that were already solvable can be solved, allowing for analysis of far more complex correlation matrices.

Chapter 8

Index: Code

Code for the Scilab implementation of this algorithm is accessible here: `https://github.com/zoeada/correlation_calibration/blob/master/correlation.sci`

Bibliography

[1] Nasser M. Abbasi. Basic Matlab implementation of the simplex matrix algorithm. `https://www.12000.org/my_notes/simplex/index.htm`, May 2017.

[2] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

[3] Chiu, M., K. Jackson, and A. Kreinin. Backward simulation of correlated multivariate mixed Poisson processes. `https://arxiv.org/pdf/2007.07976.pdf`, July 2020.

[4] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, New York, 2nd edition, 2006.

[5] rotia. MIN-2-XOR-SAT and MAX-2-XOR-SAT: are they NP-hard? `https://cs.stackexchange.com/q/38300`, Feb 2015.