Integer Least Squares Search and Reduction Strategies

Stephen Breen

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

August, 2011

A thesis submitted to McGill University
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

DEDICATION

To my patient and supportive parents Gwendolyn and Stephen Breen

ACKNOWLEDGEMENTS

I would like to acknowledge :

– Xiao-Wen Chang, my supervisor for his excellent guidance and many helpful comments and ideas.
– NSERC and McGill University for their generous support.
– My lab mates in the Scientific Computing lab for the numerous times that they have helped me through difficult problems.
– My co-workers at MedRunner who have been very understanding with my lack of availability over the past two years.
– Finally, my friends and family who have remained close over time and distance.

ABSTRACT

This thesis is concerned with integer least squares (ILS) problems, also referred to as closest vector problems. One often used approach to solving ILS problems is the discrete search method, which typically involves two stages, the reduction and the search. The main purpose of the first stage is to make the second stage faster.

For the box-constrained integer least squares problem, the reduction strategy involves column reordering of the the given matrix. There are currently two algorithms for column reordering that are most effective for the search stage. Both use all available information of the problem, but they were derived respectively from geometric and algebraic points of view and look different. In this thesis we first modify one to make it more computationally efficient and easier to comprehend. Then we prove the modified one and the other actually give the same column reordering in theory. After that we propose a new mathematically equivalent algorithm, which is more computationally efficient and is still easy to understand. This thesis then extends the idea to ordinary integer least squares (OILS) problems. A new reduction algorithm which combines the well-known Lenstra–Lenstra–Lovász (LLL) reduction and the column reordering strategy is proposed. The new reduction can be much more effective than the LLL reduction sometimes.

The thesis also reviews some common search algorithms. A new algorithm is proposed, which is based on two previous algorithms, the depth-first search and the best-first search. This hybrid algorithm makes use of the advantages of both algorithms, is more efficient than either one and is easier to implement than previous hybrid algorithms.

ABRÉGÉ

Dans le plus compliqué des cas, le problème de la méthode de moindres carrés en nombres entiers (ILS) est un non-déterministe polynomial ardu. Depuis que sa solution détient de nombreuses applications pratiques, plusieurs algorithmes ont été proposées pour résoudre ce problème et certaines de ses variantes, tel le problème ILS sous contrainte de boîte (BILS). Il ya généralement deux étapes pour résoudre le problème ILS : la réduction et la recherche. Évidemment, nous aimerions résoudre les instances du problème ILS aussi rapidement que possible ; cependant, la pluplart de la littérature ne compare pas le temps d'exécution et les "flop counts" des algorithmes. Ils utilisent plutôt une mesure abstraite (le nombre de nœuds visités durant la recherche) qui ne correspond pas toujours au temps d'exécution des algorithmes. Cette thèse passe en revue plusieurs des stratégies de réduction et de recherche pour les problèmes ILS et BILS. En comparant le temps d'exécution de différentes algorithmes de recherche, nous pouvons noter les avantages de chacune, ce qui nous permet de proposer une nouvelle stratégie de recherche plus efficace qui est une combinaison des deux autres. Il sera également démontré que deux stratégies de réduction BILS très efficaces sont semblables en théorie ; nous proposons alors une nouvelle stratégie de réduction BILS équivalente aux autres, tout en étant plus efficace. Enfin, il sera démontré que la réduction BILS peut être appliquée au problème de ILS pour atteindre un amélioration significative en performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

This thesis deals with efficient algorithms for solving integer least squares (ILS) problems, sometimes referred to as closes vector problems (CVP). This problem has been well studied in mathematics for over 100 years [12]. The ordinary ILS problem has been proven to be NP hard [17], meaning that the time needed to solve the problem can depend exponentially on the problems size. Since ILS problems arise in a number of time sensitive practical applications, it is worthwhile to study algorithms which can solve them quickly. In this thesis an efficient class of algorithms for solving ILS problems will be studied. New algorithms will also be proposed which offer decreased computational or memory cost when used to solve problems that arise in some practical applications.

In this chapter an introduction to ordinary real least squares and integer least squares problems will be given; the distinction between the two will be made clear. Some applications for the ILS problem will be mentioned and then previous work which has dealt with solving these problems efficiently will be reviewed.

1.1 Notation

This section introduces the notation which will be used throughout this thesis. Matrices will be denoted by capital letters while vectors and scalars will be in lower case. Subscripts will be used to identify elements of vectors and matrices and also columns of matrices. For example, if we have a matrix A, a_j denotes the j^{th} column and $a_{i,j}$ denotes the element in row i, column j. For a vector v, v_j denotes the j^{th} element. The identity matrix of size n will be denoted as I_n or I and the i^{th} column of this identity matrix is denoted as the vector e_i. Occasionally MatLab like notation will be used to denote subsets of rows/columns in a matrix or vector. For example $A_{:,j:k}$ denotes the matrix formed by columns $j, j+1, \ldots, k$ of the matrix A.

Rounding of a real valued scalar or the elements of a real valued vector to the nearest integer(s) will be written as $\lfloor x \rceil$, where x is the scalar or vector. Suppose we have some set of integers $\mathcal{B}$, rounding of x to the nearest value in this set will be denoted as $\lfloor x \rceil_{\mathcal{B}}$.

For a random vector v following a normal distribution with 0 mean and covariance matrix $\sigma^2 I$, we write $v \sim N(0, \sigma^2 I)$.

1.2 List of Abbreviations

The following is a list of abbreviations which will often be used in this thesis:

- LS: Least Squares.
- ILS: Integer Least Squares.
- RLS: Real Least Squares.
- OILS: Ordinary Integer Least Squares (no constraints on the solution).
- BILS: Box constrained Integer Least Squares (box constraint on the solution).
- MIMO: Multiple Input, Multiple Output.
- SE: The Schnor-Euchner Enumeration search strategy for ILS problems, [21].
- BFS: Best First Search, a tree search algorithm.
- DFS: Depth First Search, another tree search algorithm.
- CH: Chang and Han's algorithm for BILS reduction.
- SW: Su and Wassell's algorithm for BILS reduction.
- LLL: The Lenstra-Lenstra-Lovasz reduction strategy.

1.3 Ordinary Real Least Squares Problem

Consider the following linear model,

$$y = Hx + v, \tag{1.1}$$

where $y \in \mathbb{R}^m$ is the observation vector, $H \in \mathbb{R}^{m \times n}$ is called the "design matrix" and has full column rank, $x \in \mathbb{R}^n$ is the unknown parameter vector and $v \in \mathbb{R}^m$ is the noise vector with $v \sim N(0, \sigma^2 I) \in \mathbb{R}^m$. To estimate x from (1.1), an often used approach is to solve

the following ordinary real least squares problem:

$$\min_{x \in \mathbb{R}^n} \|y - Hx\|_2^2 . \tag{1.2}$$

The solution to this problem, x_{RLS} has the property that it is the maximum likelihood estimator of the true parameter vector x. This means that x_{RLS} is more likely than any other vector to be equal to the true, unknown parameter vector x.

If we expand equation (1.2) and set its gradient to 0, we will arrive at the well known "normal equations", which can be written in matrix form as,

$$H^T H x = H^T y. \tag{1.3}$$

The solution to these normal equations satisfies the following:

$$x_{\mathrm{RLS}} = (H^T H)^{-1} H^T y.$$

Some numerical methods to compute the solution of these least squares problems efficiently and reliably can be found in [4].

Least squares estimation has numerous applications in many fields in science, engineering and economics.

1.4 Integer Least Squares Problems

In some applications, the unknown parameter vector x in equation (1.1) is an integer vector, to estimate x in this case, we instead solve the following integer least squares (ILS) problem:

$$\min_{x \in \mathbb{Z}^n} \|y - Hx\|_2. \tag{1.4}$$

We no longer have a closed-form representation for the optimal solution x_{ILS} in this case, in fact, the problem is provably NP-hard [17].

Sometimes the integer parameter vector x is subject to constraints. One example of such constraints is the following set of box constraints:

$$x \in \mathcal{B}, \tag{1.5}$$

$$\mathcal{B} = \mathcal{B}_1 \times \cdots \times \mathcal{B}_n, \tag{1.6}$$

$$\mathcal{B}_i = \{x_i \in \mathbb{Z} : l_i \leq x_i \leq u_i, l_i \in \mathbb{Z}, u_i \in \mathbb{Z}\}. \tag{1.7}$$

The constrained integer least squares problem corresponding to these constraints is reffed to as the box-constrained integer least squares (BILS) problem.

Even though the problem (1.4) is NP-hard, we still have some hope to get solutions quickly for certain instances. The problems that arise in many practical applications can often be solved in reasonable amounts of time if efficient algorithms are used.

In lattice theory, the matrix H is called the "lattice generator matrix". This is because it generates a lattice $\mathcal{L}(H)$ defined as follows:

$$\mathcal{L}(H) = \{Hx : x \in \mathbb{Z}^n\}. \tag{1.8}$$

The columns of H are called the basis vectors of the lattice. A single lattice may have many different sets of basis vectors or equivalently, many different generator matrices H. Figure 1–1 shows an example of a 2 dimensional lattice with two different sets of basis vectors.

In this context, the OILS and BILS problems can be thought of as trying to locate the closest lattice point to some real point y. This is why the problem is sometimes referred to as the closest vector problem (CVP).

1.5 Applications

One application of the OILS problem arises in GPS, where carrier phase measurements are used. In GPS, there are two types of measurements that can be used to determine the position of a receiver, code phase and carrier phase. Code phase measurements can give

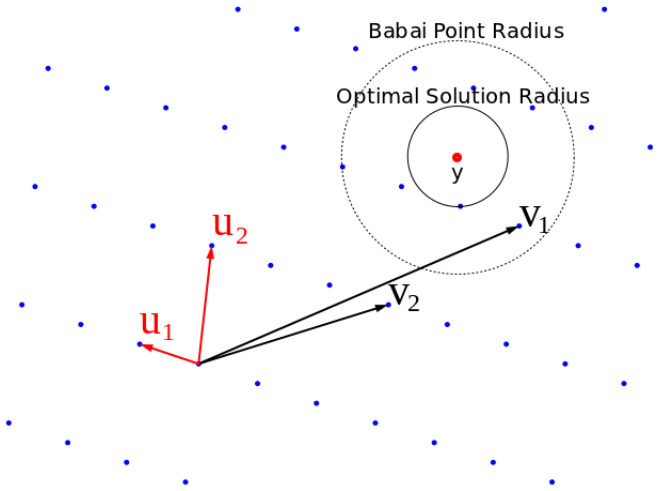

Figure 1–1: An example of a lattice with two different sets of basis vectors.

accuracy to a few meters, while carrier phase are accurate to centimeters. To make use of the more accurate carrier phase measurements, we must know how many cycles the carrier wave has gone through between a satellite and a receiver. GPS can only directly know the fractional part of the number of cycles and the difference in the number of cycles between two successive measurements. The problem of estimating the integer part of the number of cycles is called integer ambiguity estimation and one of the usual approaches taken to solve it is to construct and solve an ILS problem. For details on how the problem is formed and the usual approaches taken to solve this problem, see [27].

Some important applications such as MIMO (multiple-input, multiple-output) wireless signal decoding depend on the solution of the BILS problem. MIMO refers to the case where a wireless system has multiple input antennas transmitting a signal which is received by multiple output antennas. The purpose of a MIMO system is to maximize the throughput of the communications channel. Throughput is defined as "the average rate of successful message delivery over a communication channel".

The signal received is our input vector y from equation (1.1), it has undergone some linear transformation by the known "channel matrix" H (design matrix) and some noise has been introduced during the transmission. Originally, we know that each element of

x came from some finite set of symbols that we may want to transmit or receive (we model this property with $\mathcal{B}$). The purpose of such a system is to maximize throughput, however, the overall throughput of the system depends on how quickly and accurately we can solve the BILS problem. As the number of antennas increases, the BILS problem becomes the bottleneck on the throughput. In order to obtain estimates for x quickly we may use suboptimal solutions to the BILS problem, but under the assumption that the noise has 0 mean and is normally distributed, the BILS solution is more likely than any other possible solution to be the true integer parameter vector x [13]. For this reason, we say that a receiver which is decoding transmissions using an algorithm which solves the BILS problem exactly achieves "optimal performance"; performance refers to the likelihood that the vector found by the decoder is equal to the transmitted vector x. For more information on this problem refer to [14].

Other applications of OILS and BILS include cryptography, lattice design and number theory. For a more detail on some of these applications see [12].

1.6 Previous Work

Due to the important applications of the OILS and BILS problems, much work has gone into solving them efficiently. There are three main families of algorithms to solve these problems exactly. One family uses "voronoi cells", and will not be considered in this thesis because they are known to be relatively inefficient **??**. Another family of algorithms can be referred to as "real relaxation based branch and bound" methods or RRBB methods. Using these algorithms to solve ILS problems is a relatively new idea and no comprehensive comparisons have been done to study their efficiency, although some preliminary tests indicate they can be quite efficient or inefficient in some circumstances **??**. The last main family of algorithms that solve ILS problems exactly are "discrete search methods". This thesis will focus on studying these discrete search methods since they are often used in applications. There have also been some algorithms proposed that yield fast,

approximate solutions to the problems, some giving statistical bounds on the likelihood of error. These algorithms can sometimes be referred to as "randomized" algorithms. This thesis however will only deal with finding the optimal solution to the problem. For more details on the randomized algorithms and Voronoi cell algorithms, see [12] and references therein.

The discrete search based approaches which will be studied in this thesis try to find the optimal solution by enumerating vectors in the search space one element at a time until the optimal solution is found. The order in which the elements of the solution vectors are enumerated plays a critical role in how long the search algorithm takes to run; the reasons for this will become clear later.

1.6.1 Search Strategies

In Chapter 2, it is shown that the discrete search process is equivalent to a tree search problem. The tree in question has a depth of n; each node on a path from root to leaf (where we define leaves as nodes at depth n) represents a valid element in the solution vector. Therefore the tree has exponential width. To see an example of a subset of the nodes in such a tree, refer to Figure 2–1. The most widely used algorithm for the OILS search process, the Schnorr Euchner (SE) enumeration [21], can be thought of as a fairly straight forward depth first search in such a tree. The SE search is a simple modification to a previous algorithm known as the Pohst enumeration [20]. Other tree search algorithms may also be used to solve the problem with varying degrees of efficiency.

In the literature, some modifications to the best first tree search strategy have also been proposed to solve this problem. A few such algorithms are given in [19], [23], [11], [8] and [28]. When doing a tree search, the disadvantage of the best first approach is that the memory requirements can be exponential in the worst case and there is a significant overhead to visit each tree node. This is compared to the depth first search where the memory requirements are linear in the problem size n and there is very little computational cost to

visit a node in the tree. The advantage of the best first approach is that it is guaranteed to visit the least number of nodes in the tree. Some of the papers listed above propose a pure best first search, while others try to make some sort of trade off to achieve lower memory usage.

There have been some attempts to compare different discrete search algorithms for the OILS search process. One such paper is [19]. The authors here devise a common framework (based on a tree search) that many search algorithms can be described within and from there they can do a comparison on the estimated computational complexity of each. Unfortunately through this theoretical comparison, we can only relate the number of nodes in the search tree that are visited by various algorithms, this does not fully consider the amount of time processing each node which is often a computational bottleneck.

There have been other suggestions to improve the discrete search based algorithms as well. In [18], it is proposed that by using lower bounds on the residual from the optimal solution, we can shrink the search space (equivalently, prune the search tree). A few such lower bounds are given for special cases of the OILS problem and one for the general OILS and BILS problem. Unfortunately, the computational complexity of computing some of these bounds can be prohibitive since it adds to the processing that must be performed at each node in the tree. Overall it seems that these lower bounds do not offer a decrease in computational complexity during the search process if the search is performed on a reduced problem.

Another method that attempts to shrink the search space is given in [3]. They propose a simple stopping criteria for the search process that in theory should allow it to terminate earlier. Unfortunately, the bound derived here is not tight enough to be useful in practice and is rarely or never satisfied. Also it is proposed that after computing the bound, one can increase it by some amount with a low risk of achieving a suboptimal solution; this thesis will not consider this since it is only concerned with the exact solution of the OILS and BILS problems.

1.6.2 Reduction Strategies

As mentioned previously, the purpose of the OILS and BILS reduction is to transform the problem to an equivalent, but easier to solve one. This section will review some existing reduction strategies.

One very effective reduction strategy is known as the Korkine-Zolotarev (KZ) reduction. Unfortunately the KZ reduction is prohibitively expensive to compute, it requires solving a number of NP hard problems. The only time this reduction may be practical is when a very large number of input vectors y must be decoded for the same matrix H. For more information on the KZ reduction see [1]. The standard reduction algorithm used in practice for reducing unconstrained OILS problems is the "Lenstra-Lenstra-Lovasz" or LLL reduction [16]. It is usually much more efficient to compute than the KZ reduction, in fact any KZ reduced matrix is also LLL reduced. For more detail on how the LLL reduction and KZ reduction are related, again see [1]. The LLL reduction has been proven to converge in a finite number of steps and to have average case polynomial complexity when the basis vectors are normally distributed inside the unit sphere in $\mathbb{R}^m$. There are also results that bound the complexity based on the condition number of the input matrix. For more details on the bounds,complexity and convergence of LLL reduction, see [25].

In [26], it was found that many of the operations used in the original LLL algorithm are not always required as they do not affect the search process when the SE algorithm is used. The new reduction that results from applying only a subset of the operations is called the partial LLL reduction.

Unfortunately, neither the LLL reduction or partial LLL reduction are applicable to the BILS problem. For the BILS problem, there are other reduction strategies that focus only on permuting the columns of the matrix H, where the LLL reduction performs some other operations that are described in Chapter 3. We can separate reduction algorithms which simply try and find some optimal permutation of the columns of the matrix H into two

categories, those that only use the information contained in the matrix H, and algorithms that use both the information in H and the vector y.

Two algorithms that only use the information in H are the "Vertical-Bell Laboratories Layered Space-Time" or V-BLAST column reordering [10] and the "Sorted QR Decomposition" or SQRD column reordering [24]. An examination of these two algorithms reveals very similar motivations behind each.

Algorithms that use the information in both H and y are a fairly new development. The first was proposed by Su and Wassell in 2005 [22], and the second by Chang and Han in 2008 [6]. Numerical results show that these algorithms can offer great improvements over the previous reductions that use only the information contained in H.

1.7 Objectives and Contribution

This thesis will study both the search and reduction steps of the discrete search based ILS algorithms. For the reduction step, the LLL reduction [16] is the strategy most used in practice for the OILS problem. How the LLL reduction theoretically relates to the OILS problem has been studied in detail, and it also yields excellent results in practice. Unfortunately, for the BILS problem, the LLL reduction should not be used, the reason for this is described in Chapter 3. For the BILS problem, we are limited to performing only column permutations on the matrix H. There are a few algorithms which calculate how we should permute the columns of H, some of which were briefly introduced in section 1.6. Two more recent developments, [6] and [22], use both the matrix H and the vector y to calculate the permutations. These algorithms have shown excellent results. In this thesis it will be proven that these two algorithms are theoretically equivalent. Knowing that they are equivalent, we can use the best ideas from both to create a new reduction strategy that is faster than either of the originals and is numerically stable (the faster of the two original algorithms is not). Another advantage of these algorithms being equivalent is that since one had a geometric motivation and the other was derived algebraically, we now have both

geometric and algebraic justification for why the column orderings given by these algorithms should help speed up the search process. Also, the algorithm in [22] was derived through a geometric motivation and as such is described in terms of geometry in the original paper; this thesis will provide an algebraic explanation for the algorithm presented in [22] and offer some improvements to the original. The work which has been done with these reduction algorithms for this thesis has been accepted to the IEEE Globecom 2011 conference [5].

The motivation for the permutation based reduction strategies is not specific to the BILS problem; in theory these reduction strategies should reduce the run time for OILS problems as well. However, the very effective LLL reduction provides better results than using permutations alone. One way to think about what each type of algorithm is doing is, the LLL reduction finds a new set of shorter and more orthogonal basis vectors, while the permutation based reductions are just finding an ordering for the basis vectors that performs well in the search process. Consider Figure 1–1 suppose the original matrix H has columns v_1 and v_2, the LLL reduced matrix may have columns u_1 and u_2. The permutation based reductions are limited to simply changing the ordering so that $v_1 = v_2$ and $v_2 = v_1$. It is known that shorter more orthogonal basis vectors are preferable in the search process. By first performing LLL reduction to get a good set of basis vectors, and then applying a permutation based reduction to reorder them, we can sometimes greatly improve the performance of the search process. In this thesis, the strategy of first applying a LLL reduction, and then column permutations will be explored.

With many different algorithms for both the reduction and search process, it is not always clear how they relate. In [19] the authors propose a tree search framework to relate some of the various discrete search algorithms. This tree search framework makes the trade off between memory usage and the number of nodes visited in the tree search clear. Also we can see that as the memory usage increases, the computational complexity of visiting a node does as well. This thesis will consider combining two efficient search strategies

in a new way which is not considered by this framework. A parameter will control the influence of each individual algorithm. Using actual run time simulation results from a comparison of this combined algorithm with various settings of its parameter, we can see the strengths of each of the different approaches. This allows us to tune the new combined algorithm so that it outperforms either of the originals in some practical applications. This also provides some insight into the runtime performance comparison of each algorithm individually.

1.8 Outline

The rest of the thesis will be organized as follows;

In Chapter 2, the Schnorr-Euchner (SE) enumeration algorithm [21] will be presented in detail, much of the remainder of the thesis will use ideas and notation which comes from this algorithm. Also, since the reduction processes are trying to optimize the search process, it is critical to first understand the search process before considering the reduction.

In Chapter 3, an explanation will be given for why we need different reduction strategies for OILS and BILS problems. Then, strategies for reducing OILS and BILS problems will be presented separately. A new BILS reduction strategy will be introduced. Also it will be explained how we can take advantage of the new BILS reduction for many OILS problems.

In Chapter 4, some other notable search algorithms and modifications to the basic SE enumeration will be given. Also, a new hybrid search algorithm is proposed which combines two of the original algorithms in order to take advantage of the positive features of each.

Finally, Chapter 5 will give a summary and highlight areas where some future work could be done.

Chapter 2
Schnorr-Euchner Enumeration

As mentioned previously, there are two steps to solving ILS problems by the discrete search approach, reduction and search. The purpose of the reduction step is to transform the problem to an easier but equivalent one so the search proceeds more quickly. Therefore the reduction is performed before the search. In this thesis the details on the search process are presented first for easier understanding. The reason for this is that the reduction manipulates the matrix H in such a way to optimize it for the search process, therefore understanding the search process is crucial to understanding the reduction.

There are many search algorithms that have been proposed to solve the OILS problem. One of the most effective algorithms in terms of both overall runtime and memory consumption is the Schnorr-Euchner enumeration [21]. The SE search strategy is an improvement to a prior strategy called the Pohst enumeration strategy. For more detail on the Pohst algorithm and how it relates to the SE algorithm, see [1]. In this chapter, the SE algorithm is presented in detail, since concepts from it will be used throughout the remainder of the thesis.

Let H have the QR decomposition

$$H = [Q_1, Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where

$$\underset{n}{[Q_1,} \ \underset{m-n}{Q_2}\,] \in \mathbb{R}^{m \times m}$$

is orthogonal and $R \in \mathbb{R}^{n \times n}$ is upper triangular. Then, with $\bar{y} = Q_1^T y$ the OILS problem (1.4) is reduced to

$$\min_{x \in \mathbb{Z}^n} \|\bar{y} - Rx\|_2. \tag{2.1}$$

The goal of the search strategy is to enumerate the values for the elements of x until the optimal solution x_{ILS} is found. We would like to enumerate as few integer points as possible, $x \in \mathbb{Z}^n$ while still guaranteeing the optimal solution. Suppose the optimal solution satisfies,

$$\min_{x \in \mathbb{Z}^n} \|\bar{y} - Rx\|_2 < \beta. \tag{2.2}$$

We will discuss some good ideas for choosing an initial β later. The inequality (2.2) defines an ellipsoid in terms of x or a hyper-sphere in terms of the lattice points $w = Rx$ with radius β. For this reason, the problem is sometimes referred to as "Sphere Decoding". For an example of what the geometry may look like, refer to Figure 1–1 and consider the two circles.

Define

$$c_k = (\bar{y}_k - \sum_{j=k+1}^{n} r_{kj} x_j)/r_{kk}, \ k = n, n-1, \ldots, 1, \tag{2.3}$$

where when $k = n$ the sum in the right hand side does not exist. Notice that c_k depends on $x_{k+1:n}$. Then (2.2) can be rewritten as

$$\min_{x \in \mathbb{Z}^n} \sum_{k=1}^{n} r_{kk}^2 (x_k - c_k)^2 < \beta^2, \tag{2.4}$$

which implies the following set of inequalities:

$$\text{level } k: \quad r_{kk}^2 (x_k - c_k)^2 < \beta^2 - \sum_{i=k+1}^{n} r_{ii}^2 (x_i - c_i)^2, \tag{2.5}$$

for $k = n, n-1, \ldots, 1$. Equation (2.5) shows that valid values for c_k depend on the value of $c_{k+1:n}$. This suggests that to find an integer point which satisfies the inequality 2.2 we could start by choosing c_n since it does not depend on any previously chosen values. After c_n is chosen we may calculate which values would satisfy the inequality for c_{n-1} and so on. The SE search process will now be described.

We begin the search process at level n as mentioned above, therefore the valid values for x_n only depend on the initial β. Choose $x_n = \lfloor c_n \rceil$, the nearest integer to c_n. If the

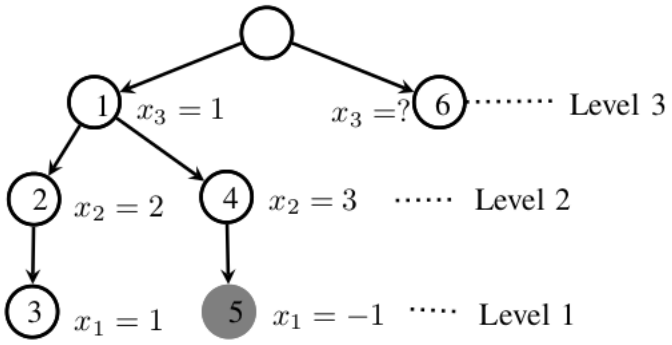

Figure 2–1: An example of the search process with solution $x = [-1, 3, 1]^T$.

inequality (2.5) with $k = n$ is not satisfied then it will not be satisfied for any integer. This means β was chosen to be too small initially so it must be enlarged and the search restarted. With x_n fixed, we can move to level $n-1$ and choose $x_{n-1} = \lfloor c_{n-1} \rceil$ with c_{n-1} calculated as in equation (2.3). At this point it is possible that the inequality (2.5) is no longer satisfied. If this is the case, we must move back to level n and choose x_n to be the second nearest integer to c_n. We continue this procedure until we reach level 1, moving back a level if ever the inequality in (2.5) for the current level is no longer satisfied. When we reach level 1, we will have found an integer point $\hat{x}$. Since $\hat{x}$ must satisfy (2.4), we can use it to define a new, smaller radius which will effectively shrink the search space since there will be fewer valid choices for values of x. To do this, update

$$\beta = \|\bar{y} - R\hat{x}\|_2$$

and try to find a better integer point which lies in the new, smaller ellipsoid. We do this by continuing the search process from the current level after the radius has been tightened. The next step in the process is to move back to level 2 since no new choice for x_1 will give a better solution. Finally in the search process, when we can no longer find any x_n to satisfy (2.5) with $k = n$, the search is complete and the last integer point $\hat{x}$ found is the solution. If we initially set $\beta = \infty$ the first point $\hat{x}$ that we find is known as the Babai integer point.

The above search process is actually a depth-first tree search in a tree of height n, see Figure 2–1, where the number in a node denotes the step number at which the node is encountered. Each edge going from level i to $i-1$ represents fixing x_{i-1} to some value, and each edge has a weight which is given by $r_{ii}^2(z_i - c_i)^2$. We call the sum of the edge weights from the root to any node in the tree the "cost" to visit that node. This cost should not be confused with computational cost, when we discuss computational cost it will be mentioned explicitly. Notice that if we take the sum of these weights from $i, \ldots, n$ we simply get the partial residual squared for fixing $x_{i:n}$, which is defined as

$$\|R_{i:n,i:n}x_{i:n} - \bar{y}_{i:n}\|_2^2 .$$

In this tree we define all leaves to have depth n. For example, we would not consider node 6 in Figure 2–1 a leaf, it does not have children in the figure simply because none of its possible children satisfied the constraint from the search process (2.4). This implies that each leaf in the tree represents an integer vector $x \in \mathbb{Z}^n$ and its cost is the squared residual

$$\|Rx - \bar{y}\|_2^2 .$$

This means that the OILS problem is equivalent to finding the lowest cost leaf in the tree. The SE enumeration takes advantage of the fact that we can easily calculate the lowest cost child of any given node to make the depth first search more efficient. Compared to a standard tree search problem where to find the lowest weight edge leaving a given node you may have to check each edge individually. In fact we can easily visit the children of a node in order of increasing cost, that is what we are doing when we initially choose $x_k = \lfloor c_k \rceil$, and next choose it to be the second and third nearest integer to c_k.

A modification of the SE enumeration can be used to solve the BILS problem. To ensure that we remain within the box constraint, instead of choosing $x_k = \lfloor c_k \rceil$ at step k, we choose $x_k = \lfloor c_k \rceil_{\mathcal{B}_k}$, where $\mathcal{B}_k$ comes from (1.5). Also recall that the notation $\lfloor c_k \rceil_{\mathcal{B}_k}$ denotes rounding c_k to the nearest integer in the set $\mathcal{B}_k$. Suppose $x_{k-1:n}$ are fixed,

then we must also ensure that as we visit the node corresponding to the second nearest integer (and all subsequent integers) for x_k that we remain within the box constraint. This is trivial to accomplish, we simply stop incrementing x_k if we hit the upper bound, and stop decrementing it if we hit the lower bound. If all values for x_k that are within the box constraint have been used up but we are still within the area defined by the ellipsoid, we move back to level $k-1$. Following this process will yield the optimal BILS solution. For more information on the BILS implementation of the SE algorithm, see [6].

The choice of the initial value for β can have a large impact on the number of nodes visited during the search. If β is initially chosen to be infinity, the first point found by the SE search is the Babai integer point. One simple method for choosing an initial β which may provide better results than infinity is as follows. Let x_{RLS} be the real LS solution, then calculate the residual

$$\left\| R\lceil x_{\text{RLS}} \rfloor - \bar{y} \right\|_2 .$$

We may use this residual as the initial β since the optimal solution must have a residual less than or equal to it. While the search process is never slower when using this initial β, often it will just end up being equivalent to using infinity since the Babai integer point is usually a better estimate than the rounded real least squares solution. In Chapter 3 another idea for choosing an initial β will be introduced.

Chapter 3
Reduction Strategies

This chapter focuses on the reduction stage for solving ILS problems. The goal is to modify the matrix H and vector y in such a way to preserve the original solution but make the problem easier. First the types of operations that we may perform on H and y will be described. The operations that are allowed are different for OILS and BILS problems, the difference and reason for it will be explained. Next some BILS reduction strategies will be described, we will show that two of the more effective strategies in the literature are actually equivalent and present a new algorithm that is based on combining ideas from these two. Finally, it was found that some of the ideas from the BILS reduction algorithms can be applied to the OILS problems as well. This idea will be explored and some numerical results presented.

Consider the OILS problem defined in equation (1.4). The goal of the reduction is to modify the matrix H in such a way that we still obtain the same solution x, but in fewer steps. With this goal, it is essential to know what types of operations we are allowed to perform on the matrix H so that the solution x is not modified. The first type of operation to consider is the orthogonal transformation. Suppose we apply some orthogonal matrix Q from the left, then it is easy to see that equation (1.4) becomes:

$$\|Q^T(y - Hx)\|_2 = \|(y - Hx)\|_2.$$

The second type of transformation that we may apply to the matrix H is any unimodular matrix. Unimodular matrices are square, integer matrices with determinant 1 or -1. We can show that the inverse of a unimodular matrix is an integer matrix through basic linear algebra; this property is very useful for our application. Consider applying such a

unimodular matrix Z to H from the right, equation (1.4) becomes:

$$\min_{x \in \mathbb{Z}^n} \|HZZ^{-1}x - y\|_2 \tag{3.1}$$

$$= \min_{z \in \mathbb{Z}^n} \|\bar{H}z - y\|_2, \tag{3.2}$$

where $\bar{H} = HZ$ and $z = Z^{-1}x$. When we solve this new OILS problem, we obtain some solution z. If Z is a known unimodular matrix, we can solve for x by computing $x = Zz$, this gives us the OILS solution to the original problem. Note that if Z were not unimodular, we would have no guarantee that Zz would be integer.

Finally, it is worth mentioning that permutation matrices are unimodular since they will be used extensively throughout this chapter. It is obvious that the effect on the solution vector x from applying a permutation matrix to H from the right is to reorder the elements of the the solution x, this is important when considering the BILS problem.

When reducing the BILS problem we must be more careful. Consider the constraints on the solution x, defined in equation (1.5). Applying orthogonal matrices to H from the left has no effect on x, so the constraints are also unaffected. Applying permutation matrices to H from the right will reorder the elements of the solution x, so we must reorder the elements in the constraint vectors as well, which is a trivial operation. However if we apply a general unimodular matrix Z to H from the right, we can no longer easily enforce the constraints on the solution; the simple box constraints become complicated. To see why this is the case, recall the SE search process. Suppose the search is currently at some level k and we would like to know to which value we should fix z_{k-1} (note that z denotes $Z^{-1}x$). We know that

$$l_{k-1} \leq x_{k-1} \leq u_{k-1},$$

but to determine such a bound for z_{k-1}, we would need to know the entire vector z in order to compute $Z_{k-1,:}z$. At this point in the search, we only know the last k elements of it. The only way to complete the search process is to ignore the bounds on z, find a potential solution, compute $x = Zz$, and then check if each element of x is within the bounds. This

is extremely inefficient since many potential solutions could be eliminated very early on in the search if we were able to enforce the constraints during the search process. It is for this reason that when reducing BILS problems, we only consider orthogonal transformations and column permutations.

3.1 BILS Reduction Algorithms

Due to the difficulty of applying general unimodular matrices to reduce the BILS problem, the algorithms in this section will focus on finding some permutation of the columns of the matrix H in order to optimize the search process.

3.1.1 Previous Reductions

The "Vertical-Bell Laboratories Layered Space-Time" or V-BLAST algorithm [10] mentioned in section 1.6 is a commonly used strategy to calculate the column permutations for H. Suppose we are working with R, which comes from the QR decomposition of H. Recall that the product of the diagonal elements of an upper triangular matrix is equal to the matrices determinant, and this value is invariant under permutation. This means that any time we swap two columns in the matrix H and re-calculate the QR decomposition, one diagonal element of R will always increase and the other will decrease.

The goal of the V-BLAST algorithm is to proceed from $k = n, \ldots, 1$ and find the column h_p from the set of columns $h_1, \ldots, h_k$ such that when h_p and h_k are swapped, the magnitude of the diagonal element $|r_{kk}|$ is maximal among the k choices. There is an efficient algorithm to compute such a column ordering which is described in [7].

In [6], the "Sorted QR Decomposition" or SQRD column reordering strategy originally presented in [24] for the same purpose as V-BLAST, was proposed for the purpose of BILS reduction. In the SQRD algorithm, we perform the QR decomposition from columns $k = 1, \ldots, n$, at each step choosing as the k^{th} column the column from the set $k, \ldots, n$ which gives the smallest magnitude for the diagonal element $|r_{kk}|$. This should yield large

$|r_{kk}|$ toward the end of the matrix since the product of the diagonal elements is a constant. Note that both SQRD and V-BLAST only use the information in the matrix H.

In [22], Su and Wassell considered the geometry of the BILS problem for the case that H is nonsingular and proposed a new column reordering algorithm (to be called the SW algorithm from here on for convenience) which uses all information of the BILS problem. Unfortunately, the geometric interpretation of this algorithm is hard to understand. Probably due to page limit, the description of the algorithm is very concise, making efficient implementation difficult for ordinary users.

This thesis will give some new insight of the SW algorithm from an algebraic point of view. Some modifications will be made so that the algorithm becomes more efficient and easier to understand and furthermore it can handle a general full column rank H. It is worth mentioning that the SW algorithm is not numerically stable. The numerical stability is not necessarily crucial since a wrong answer just results in a different set of permutations for the columns of H where any set of permutations is allowable. Until this point, other column reordering algorithms only considered the matrix H.

Independently Chang and Han in [6] proposed another column reordering algorithm (which will be referred to as CH). Their algorithm also uses all information of the BILS problem and the derivation is based on an algebraic point of view. It is easy to see from the equations in the search process exactly what the CH column reordering is doing and why we should expect a reduced complexity in the search process. The detailed description of the CH column reordering is given in [6] and it is easy for others to implement the algorithm. But our numerical tests and theoretical analysis indicated that CH has a higher complexity than SW, when SW is implemented efficiently. Our numerical tests also showed that CH and SW *almost* always produced the same permutation matrix P, the only exception is when then matrix H is very ill conditioned.

In this section it will be shown that the CH algorithm and the (modified) SW algorithm give the same column reordering in theory. This is interesting because both algorithms

were derived through different motivations and we now have both a geometric justification and an algebraic justification for why the column reordering strategy should reduce the complexity of the search. Furthermore, using the knowledge that certain steps in each algorithm are equivalent, we can combine the best parts from each into a new algorithm. The new algorithm has a lower flop count than either of the originals. This is important to the successive interference cancellation decoder, which computes a suboptimal solution to the BILS problem. When computing suboptimal solutions, the overall runtime may not be so dominated by the search, and the complexity of the reduction can become a bottleneck. The new algorithm can be interpreted in the same way as CH, so it is easy to understand.

In the following subsections, the CH and SW algorithms will be described in detail. This is necessary to understanding the proof of their equivalence and to see the motivation for the new algorithm. Also, a new algebraic interpretation and some improvements will be given for the SW algorithm. Finally the proof of equivalence of CH and SW will be given, and the new algorithm will be presented.

3.1.2 CH Algorithm

The CH algorithm first computes the QR decomposition of H, then tries to reorder the columns of R. The motivation for this algorithm comes from observing equation (2.5). If the inequality is false we know that the current choice for the value of x_k given $x_{k+1:n}$ are fixed is incorrect and we prune the search tree (we do not need to explore the subtree for the current node). We would like to choose the column permutations so that it is likely that the inequality will be false at higher levels in the search tree, this way we waste less time exploring solutions that are not optimal. The CH column reordering strategy does this by trying to maximize the left hand side of (2.5) with large values of $|r_{kk}|$ and minimize the right hand side by making $|r_{kk}(x_k - c_k)|$ large for values of $k = n, n-1, \ldots, 1$.

Here we describe step 1 of the CH algorithm, which determines the last column of the final R (or equivalently the last column of the final H). Subsequent steps are the same

but are applied to a subproblem that is one dimension smaller. In step 1, for $i = 1, \ldots, n$ we interchange columns i and n of R (thus entries of i and n in x are also swapped), then return R to upper-triangular by a series of Givens rotations applied to R from the left, which are also applied to $\bar{y}$. The following example demonstrates the process of returning the matrix R to upper triangular after column 5 is swapped with column 2 in a 5×5 matrix.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \\ & \times & & \times & \\ & \times & & & \end{bmatrix} \tag{3.3}$$

Equation 3.3 shows the matrix directly after the column swap. We want to restore it to upper triangular. To do so, we start by using $n-i$ Givens rotations to zero the subdiagonal elements in column i, in this case $i = 2$. Each Givens rotation is an orthogonal matrix which adds multiples of two rows to eachother, for our purposes, we would like to add only multiples of adjacent rows. A Givens rotation that uses row k to zero element j in row $k+1$ is defined in equation (3.4), denote such a Givens rotation as $G_{k,k+1}$:

$$\begin{bmatrix} I_{k-1} & & & \\ & c & -s & \\ & -s & c & \\ & & & I_{n-k-1} \end{bmatrix}, \quad c^2 + s^2 = 1 \tag{3.4}$$

where

$$c = \frac{R_{k,j}}{\sqrt{R_{k,j}^2 + R_{k+1,j}^2}}$$

and

$$s = \frac{R_{k+1,j}}{\sqrt{R_{k,j}^2 + R_{k+1,j}^2}}.$$

We can use rotations $G_{n-1,n}, G_{n-2,n-1}, \ldots, G_{i,i+1}$ to zero the subdiagonal elements in the i^{th} column, however this creates subdiagonal elements in columns $i+1, \ldots, n$. Equation (3.5) shows the matrix after applying this first round of Givens rotatations.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \\ & & & \times & \end{bmatrix} \tag{3.5}$$

We can now use a second round of Givens rotation to eliminate the new subdiagonal entries and restore the matrix to upper triangular. We use the rotations in the following order,

$$G_{i+1,i+2}, G_{i+2,i+3}, \ldots, G_{n-1,n}$$

where each rotation should zero the subdiagonal element in the corresponding column.

To avoid confusion, we denote the new R after restoring the upper triangular structure by $\hat{R}$ and the new $\bar{y}$ by $\hat{y}$, where $\hat{y}$ is $\bar{y}$ with the Givens rotations applied to it. We then compute $c_n = \hat{y}_n / \hat{r}_{n,n}$ and

$$x_i^c = \arg\min_{x_i \in \mathcal{B}_i} |\hat{r}_{nn}(x_i - c_n)| = \lfloor c_n \rceil_{\mathcal{B}_i}, \tag{3.6}$$

where the superscript c denotes the CH algorithm. Let $\bar{x}_i^c$ be the second closest integer in $\mathcal{B}_i$ to c_n, i.e.,

$$\bar{x}_i^c = \lfloor c_n \rceil_{\mathcal{B}_i \setminus x_i^c}.$$

Define

$$\text{dist}_i^c = |\hat{r}_{nn}(\bar{x}_i^c - c_n)|, \tag{3.7}$$

which represents the partial residual given when x_i is taken to be $\bar{x}_i^c$. Let $j = \arg\max_i \text{dist}_i^c$. Then column j of the original R is chosen to be the n^{th} column of the final R. With the corresponding updated upper triangular R and $\bar{y}$ (here for convenience we have removed

hats), the algorithm then updates $\bar{y}_{1:n-1}$ again by setting

$$\bar{y}_{1:n-1} := \bar{y}_{1:n-1} - r_{1:n-1,n} x_j$$

where $x_j = x_j^c$. Choosing x_j to be x_j^c here is exactly the same as what the search process does. We then continue to work on the subproblem

$$\min_{\tilde{x} \in \mathbb{Z}^{n-1}} \left\| \bar{y}_{1:n-1} - R_{1:n-1,1:n-1} \tilde{x} \right\|_2 , \tag{3.8}$$

where

$$\tilde{x} = [x_1, \ldots, x_{j-1}, x_n, x_{j+1}, \ldots x_{n-1}]^T$$

satisfies the corresponding box constraint. The pseudocode of the CH algorithm is given in Algorithm 1.

To determine the last column, CH finds the permutation to maximize $|r_{nn}(\bar{x}_i^c - c_n)|$. Using $\bar{x}_i^c$ instead of x_i^c ensures that $|\bar{x}_i^c - c_n|$ is never less than 0.5 but also not very large. This means that usually if $|r_{nn}(\bar{x}_i^c - c_n)|$ is large, $|r_{nn}|$ is large as well and the requirement to have large $|r_{nn}|$ is met. Using x_i^c would not be a good choice because $|x_i^c - c_n|$ might be very small or even 0, then column i would not be chosen to be column n even if the corresponding $|r_{nn}|$ is large and on the contrary a column with small $|r_{nn}|$ but large $|x_i^c - c_n|$ may be chosen.

Now we will consider the complexity of CH. The significant computational cost comes from line 9 in Algorithm 1, which requires $6(k-i)^2$ flops. If we sum this cost over all loop iterations and add the cost of the QR decomposition by Householder transformations, we get a total complexity of $0.5n^4 + 2mn^2$ flops.

3.1.3 SW Original Algorithm

The motivation for the SW algorithm comes from examining the geometry of the search process.

Algorithm 1 CH Algorithm - Returns p, the column permutation vector

$p := 1 : n$
$p' := 1 : n$
Compute the QR decomposition of H: $\begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} H = \begin{bmatrix} R \\ 0 \end{bmatrix}$ and compute $\bar{y} := Q_1^T y$
for $k = n$ to 2 **do**
  $maxDist := -1$
  for $i = 1$ to k **do**
    $\hat{y} := \bar{y}_{1:k}$
    $\hat{R} := R_{1:k,1:k}$
    swap columns i and k of $\hat{R}$, return it to upper triangular with Givens rotations, also apply the Givens rotations to $\hat{y}$
    $x_i^c := \lfloor \hat{y}_k / \hat{r}_{k,k} \rceil_{\mathcal{B}_i}$
    $\bar{x}_i^c := \lfloor \hat{y}_k / \hat{r}_{k,k} \rceil_{\mathcal{B}_i \setminus x_i^c}$
    $dist_i^c := |\hat{r}_{k,k} \bar{x}_i^c - \hat{y}_k|$
    if $dist_i^c > maxDist$ **then**
      $maxDist := dist_i^c$
      $j := i$
      $R' := \hat{R}$
      $y' := \hat{y}$
    end if
  end for
  $p_k := p'_j$
  Interchange the intervals $\mathcal{B}_k$ and $\mathcal{B}_j$
  Intechange entries k and j in p'
  $R_{1:k,1:k} := R'$
  $\bar{y}_{1:k} := y' - R'_{1:k,k} x_j^c$
end for
$p_1 := p'_1$

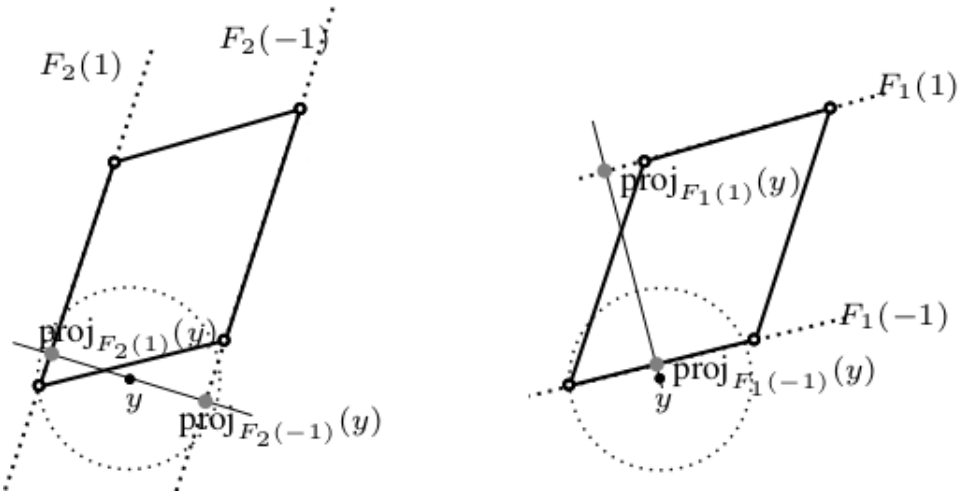

Figure 3–1: Geometry of the search with two different column ordering.

Figure 3–1 shows a 2-D BILS problem; 3–1(a) represents the original column ordering and 3–1(b) is after the columns have been swapped.

In the SW algorithm $H = [h_1, \ldots, h_n]$ is assumed to be square and non-singular. Let

$$G = [g_1, \ldots, g_n] = H^{-T}.$$

For any integer α, [22] defines the affine sets,

$$F_i(\alpha) = \{w \mid g_i^T(w - h_i\alpha) = 0\}.$$

The lattice points generated by H occur at the intersections of these affine sets. Let the orthogonal projection of a vector s onto a vector t be denoted as $\text{proj}_t(s)$, then the orthogonal projection of some vector s onto $F_i(\alpha)$ is

$$\text{proj}_{F_i(\alpha)}(s) = s - \text{proj}_{g_i}(s - h_i\alpha).$$

Therefore the orthogonal distance between s and $F_i(\alpha)$ is

$$\text{dist}(s, F_i(\alpha)) = \|s - \text{proj}_{F_i(\alpha)}(s)\|_2.$$

In [22], the points labeled $\text{proj}_{F_2(1)}(y)$ and $\text{proj}_{F_2(-1)}(y)$ in Figure 3–1 are called residual targets and "represent the components [of y] that remain after an orthogonal part has been projected away."

Note that $F_2(\alpha)$ in Figure 3–1 is a sublattice of dimension 1. Algebraically it is the lattice generated by H with column 2 removed. It can also be thought of as a subtree of the search tree where $x_2 = \alpha$ has been fixed. In the first step of the search process for a general case, x_n is chosen to be

$$x_n = \arg\min_{\alpha \in \mathcal{B}_n} \text{dist}(y, F_n(\alpha));$$

thus $F_n(x_n)$ is the nearest affine set to y. Actually the value of x_n is identical to $\lfloor c_n \rceil_{\mathcal{B}_n}$ given in Chapter 2, which will be proven later. Then y is updated as

$$y := \text{proj}_{F_n(x_n)}(y) - h_n x_n.$$

If we look at Figure 3–1, we see that the projection $\text{proj}_{F_n(x_n)}(y)$ moves y onto $F_n(x_n)$, while the subtraction of $h_n x_n$ algebraically fixes the value of x_n. This is necessary because in subsequent steps we will not consider the column h_n.

We now apply the same process to the new $n-1$ dimensional search space $F_n(x_n)$. If at some level i,

$$\min_{\alpha \in \mathcal{B}_i} \text{dist}(y, F_i(\alpha))$$

exceeds the current search radius, we must move back to level $i+1$. When the search process reaches level 1 and fixes x_1, it updates the radius to $\text{dist}(y, F_1(x_1))$ and moves back up to level 2.

Note that this search process is mathematically equivalent to the one described in Chapter 2; the difference is that it does projections because the generator matrix is not assumed to be upper-triangular. Computationally the former is more expensive than the latter since the cost for QR decomposition is only incurred once and we avoid the expensive projections.

To see the motivation of the SW algorithm for choosing a particular column ordering, consider Figure 3–1. Suppose the search algorithm has knowledge of the residual for the optimal solution (the radius of the circle in the diagram). With the column ordering chosen in (a), there are two possible choices for x_2, leading to the two dashed lines $F_2(-1)$ and $F_2(1)$ which cross the circle. This means that we will need to compute x_1 for both of these choices before we can determine which one leads to the optimum solution. In (b), there is only one possible choice for x_1, leading to the only dashed line $F_1(-1)$ which crosses the circle, meaning we only need to find x_2 to find the optimum solution. Since the projection resulting from the correct choice of x_2 will always be within the sphere, it makes sense

to choose the ordering which maximizes the distance to the second best choice for x_2 in hopes that the second nearest choice will result in a value for

$$\min_{\alpha \in \mathcal{B}_2} \operatorname{dist}(y, F_2(\alpha))$$

outside the sphere and the dimensionality can be reduced by one. For more detail on the geometry, see [22].

The following will give an overview of the SW algorithm as given in [22] but described in a framework similar to what was used to describe CH. In the first step to determine the last column, for each $i = 1, \ldots, n$, we compute

$$x_i^s = \arg\min_{\alpha \in \mathcal{B}_i} \operatorname{dist}(y, F_i(\alpha)) = \arg\min_{\alpha \in \mathcal{B}_i} |y^T g_i - \alpha| = \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i}, \tag{3.9}$$

where the superscript s stands for the SW algorithm. Let $\bar{x}_i^s$ be the second closest integer in $\mathcal{B}_i$ to $y^T g_i$, i.e.,

$$\bar{x}_i^s = \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i \setminus x_i^s}.$$

Let $j = \arg\max_i \operatorname{dist}(y, F_i(\bar{x}_i^s))$. Then SW chooses column j as the last column of the final reordered H, updates y by setting

$$y := \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s$$

and updates G by setting

$$g_i := \text{proj}_{F_j(0)}(g_i)$$

for all $i \neq j$. After G and y have been updated, the algorithm continues to find column $n-1$ in the same way etc. The pseudo-code of the SW algorithm is given in Algorithm 2.

Su and Wassell did not say how to implement the algorithm and did not give a complexity analysis. The parts of the computational cost we must consider for implementation

Algorithm 2 SW Algorithm - Returns p, the column permutation vector

$p := 1 : n$
$p' := \{1, 2, \ldots, n\}$
$G := H^{-T}$
for $k = n$ to 2 **do**
  $maxDist := -1$
  for $i \in p'$ **do**
    $x_i^s := \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i}$
    $\bar{x}_i^s := \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i \setminus x_i^s}$
    $\text{dist}_i^s := \text{dist}(y, F_i(\bar{x}_i^s))$
    if $dist_i^s > maxDist$ **then**
      $maxDist := dist_i^s$
      $j := i$
    end if
  end for
  $p_k := j$
  $p' := p' \backslash j$
  $y := \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s$
  for $i \in p'$ **do**
    $g_i := \text{proj}_{F_j(0)}(g_i)$
  end for
end for
$p_1 := p'$

occur in lines 9 and 19. Note that

$$\mathrm{dist}(y, F_i(\bar{x}_i^s)) = \|\mathrm{proj}_{g_i}(y - h_i\bar{x}_i^s)\|_2$$

and

$$\mathrm{proj}_{F_j(0)}(g_i) = g_i - \mathrm{proj}_{g_i} g_i,$$

where

$$\mathrm{proj}_{g_i} = g_i g_i^\dagger = g_i g_i^T / \|g_i\|^2.$$

A naive implementation would first compute proj_{g_i}, requiring n^2 flops, then compute

$$\|\mathrm{proj}_{g_i}(y - h_i\bar{x}_i^s)\|_2$$

and

$$g_i - \mathrm{proj}_{g_i} g_i,$$

each requiring $2n^2$ flops. Summing these costs over all loop iterations we get a total complexity of $2.5n^4$ flops. In the next subsection we will simplify some steps in Algorithm 2 and show how to implement them efficiently.

3.1.4 SW Algorithm Interpretation and Improvements

In this section we give new algebraic interpretation of some steps in Algorithm 2, simplify some key steps to improve the efficiency, and extend the algorithm to handle a more general case. All line numbers refer to Algorithm 2.

First we show how to efficiently compute dist_i^s in line 9. Observing that $g_i^T h_i = 1$, we have

$$\mathrm{dist}_i^s = \|g_i g_i^\dagger (y - h_i\bar{x}_i^s)\|_2 = |y^T g_i - \bar{x}_i^s| / \|g_i\|_2. \tag{3.10}$$

Note that $y^T g_i$ and $\bar{x}_i^s$ have been computed in lines 7 and 8, respectively. So the main computational cost of computing dist_i^s is the cost of computing $\|g_i\|_2$, requiring only $2n$

flops. For $k = n$ in Algorithm 2,

$$y^T g_i = y^T H^{-T} e_i = (H^{-1} y)^T e_i,$$

i.e., $y^T g_i$ is the i^{th} entry of the real solution x for $Hx = y$. The interpretation can be generalized to a general k.

In line 19 Algorithm 2,

$$\begin{aligned} g_i^{\text{new}} &\equiv \text{proj}_{F_j(0)}(g_i) \\ &= (I - \text{proj}_{g_j}) g_i = g_i - g_j (g_j^T g_i / \|g_j\|_2^2). \end{aligned} \tag{3.11}$$

Using the last expression for computation needs only $4n$ flops (note that $\|g_j\|_2$ has been computed before, see (3.10)). We can actually show that the above is performing updating of G, the Moore-Penrose generalized inverse of H after we remove its j^{th} column. This will allow us to interpret the subsequent steps of SW as operating on a subproblem. For proof of this, see [9].

In line 17 of Algorithm 2,

$$\begin{aligned} y^{\text{new}} &\equiv \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s = (y - g_j g_j^\dagger (y - h_j x_j^s)) - h_j x_j^s \\ &= (I - \text{proj}_{g_j})(y - h_j x_j^s). \end{aligned} \tag{3.12}$$

This means that after x_j is fixed to be x_j^s, $h_j x_j^s$ is combined with y (the same as CH does) and then the vector is projected to the orthogonal complement of the space spanned by g_j. We can show that this guarantees that the updated y is in the subspace spanned by the columns of H which have not yet been chosen. This is consistent with the assumption that H is nonsingular, which implies that the original y is in the space spanned by the columns of H. However, it is not necessary to apply the orthogonal projector $I - \text{proj}_{g_j}$ to $y - h_j x_j^s$ in (3.12). The reason is as follows. In Algorithm 2, y^{new} and g_i^{new} will be used only for

computing $(y^{\text{new}})^T g_i^{\text{new}}$ (see line 7). But from (3.11) and (3.12)

$$\begin{aligned}(y^{\text{new}})^T g_i^{\text{new}} &= (y - h_j x_j^s)^T (I - \text{proj}_{g_j})(I - \text{proj}_{g_j}) g_i \\ &= (y - h_j x_j^s)^T g_i^{\text{new}}.\end{aligned}$$

Therefore, line 17 can be replaced by $y := y - h_j x_j^s$. This not only simplifies the computation but also is much easier to interpret—after x_j is fixed to be x_j^s, $h_j x_j^s$ is combined into y as in the CH algorithm. Let $H_{:,1:n-1}$ denote H after its j^{th} column is removed. We then continue to work on the subproblem

$$\min_{\check{x} \in \mathbb{Z}^{n-1}} \|y - H_{:,1:n-1}\check{x}\|_2, \tag{3.13}$$

where

$$\check{x} = [x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n]^T$$

satisfies the corresponding box constraint. Here $H_{:,1:n-1}$ is not square. But there is no problem to handle it, see the next paragraph.

In [22], H is assumed to be square and non-singular. In our opinion, this condition may cause confusion, since for each k except $k = n$ in Algorithm 2, the remaining columns of H which have not been chosen do not form a square matrix. Also the condition restricts the application of the algorithm to a general full column rank matrix H, unless we transform H to a nonsingular matrix R by the QR decomposition. To extend the algorithm to a general full column rank matrix H, we need only replace line 3 by $G := (H^\dagger)^T$. This extension has another benefit. We mentioned before that the updating of G in line 19 is actually the updating of the Moore-Pernrose generalized inverse of the matrix formed by the columns of H which have not been chosen. So the extension makes all steps consistent.

To reliably compute G for a general full column rank H, we can compute the QR decomposition $H = Q_1 R$ by the Householder transformations and then solve the triangular system $RG^T = Q_1^T$ to obtain G. This requires $(5m - 4n/3)n^2$ flops. Another less reliable but more efficient way to do this is to compute $G = H(H^T H)^{-1}$. To do this efficiently we

would compute the Cholesky decomposition $H^T H = R^T R$ and solve $R^T R G^T = H^T$ for G by using the triangular structure of R. The total computational cost for computing G by this method can be shown to be $3mn^2 + \frac{n^3}{3}$. If H is square and nonsingular, we would use the LU decomposition with partial pivoting to compute H^{-1} and the cost is $2n^3$ flops.

For the rest of the algorithm if we use the simplification and efficient implementations mentioned above, we can show that it needs $4mn^2$ flops.

We see the modified SW algorithm is much more efficient than both the CH algorithm and the SW algorithm implemented in the naive way we mentioned in the previous subsection.

3.1.5 Proof of Equivalence of SW and CH

In this subsection we prove that CH and the modified SW produce the same set of permutations for a general full column rank H. To prove this it will suffice to prove that $x_i^s = x_i^c$, $\bar{x}_i^s = \bar{x}_i^c$, $\mathrm{dist}_i^s = \mathrm{dist}_i^c$ for $i = 1, \ldots, n$ in the first step which determines the last column of the final reordered H and that the subproblems produced for the second step of each algorithm are equivalent.

Proving $x_i^s = x_i^c$ is not difficult. The only effect the interchange of columns i and n of R in CH has on the real LS solution is that elements i and n of the solution are swapped. Therefore x_i^c is just the i^{th} element of the real LS solution rounded to the nearest integer in $\mathcal{B}_i$. Thus, with equations (3.6) and (3.9),

$$x_i^c = \lfloor (H^\dagger y)_i \rceil_{\mathcal{B}_i} = \lfloor e_i^T H^\dagger y \rceil_{\mathcal{B}_i} = \lfloor g_i^T y \rceil_{\mathcal{B}_i} = x_i^s. \tag{3.14}$$

Therefore we also have $\bar{x}_i^c = \bar{x}_i^s$.

In CH, after applying a permutation P to swap columns i and n of R, we apply V^T, a product of the Givens rotations, to bring R back to a new upper triangular matrix, denoted by $\hat{R}$, and also apply V^T to $\bar{y}$, leading to

$$\hat{y} = V^T \bar{y}.$$

Thus

$$\hat{R} = V^T R P$$

and

$$\hat{y} = V^T \bar{y} = V^T Q_1^T y.$$

Then

$$H = Q_1 R = Q_1 V \hat{R} P^T,$$

$$H^\dagger = P \hat{R}^{-1} V^T Q_1^T,$$

$$g_i = (H^\dagger)^T e_i = Q_1 V \hat{R}^{-T} P^T e_i = Q_1 V \hat{R}^{-T} e_n,$$

and

$$\|g_i\|_2 = \|\hat{R}^{-T} e_n\|_2 = 1/|\hat{r}_{nn}|.$$

Therefore, with (3.10) and (3.7)

$$\begin{aligned} \mathrm{dist}_i^s = \frac{|y^T g_i - \bar{x}_i^s|}{\|g_i\|_2} &= |\hat{r}_{nn}||y^T Q_1 V \hat{R}^{-T} e_n - \bar{x}_i^s| \\ &= |\hat{r}_{nn}||\hat{y}_n/\hat{r}_{nn} - \bar{x}_i^s| = |\hat{r}_{nn}(c_n - \bar{x}_i^c)| = \mathrm{dist}_i^c. \end{aligned} \tag{3.15}$$

Now we consider the subproblem (3.8) in CH and the subproblem (3.13) in SW. We can easily show that $R_{1:n-1,1:n-1}$ in (3.8) is the R-factor of the QR decomposition of $H_{:,1:n-1}P$, where $H_{:,1:n-1}$ is the matrix given in (3.13) and P is a permutation matrix such that $\breve{x} = P\tilde{x}$, and that $\bar{y}_{1:n-1}$ in (3.8) is the multiplication of the transpose of the Q_1-factor of the QR decomposition of $H_{:,1:n-1}P$ and y in (3.13). Thus the two subproblems are equivalent. To see proof of this, simply observe the following:

$$HP = QRP \tag{3.16}$$

$$= QV^T VRP \tag{3.17}$$

Where V is the product of Givens rotations which restores R to an upper triangular matrix after the permutation P. Therefore, VRP is upper triangular and obviously equivalent

to the matrix used in the second step of CH by its definition; it is just a permutation applied to R which is then restored to upper triangular through the product of Givens rotations V. The equivalence of y in the second step should be simple to deduce.

3.1.6 New Algorithm

Now that we know the two algorithms are equivalent, we can take the best parts from both and combine them to form a new algorithm. The main computational cost in CH is to interchange the columns of R and return it to upper-triangular form using Givens rotations. When we determine the k^{th} column, we must do this k times. We can avoid all but one of these column interchanges by computing x_i^c, $\bar{x}_i^c$ and dist_i^c directly using the equations from SW.

After the QR decomposition of H, we solve the reduced BILS problem (2.1). We need only consider how to determine the last column of the final R. Other columns can be determined similarly. Here we use the ideas from SW. Let $G = R^{-T}$, which is lower triangular. By (3.14), we compute for $i = 1, \ldots, n$

$$x_i = \left\lfloor \bar{y}^T G_{:,i} \right\rceil_{\mathcal{B}_i} = \left\lfloor \bar{y}_{i:n}^T G_{i:n,i} \right\rceil_{\mathcal{B}_i}, \ \bar{x}_i = \left\lfloor \bar{y}_{i:n}^T G_{i:n,i} \right\rceil_{\mathcal{B}_i \setminus x_i},$$
$$\mathrm{dist}_i = |\bar{y}_{i:n}^T G_{i:n,i} - \bar{x}_i| / \|G_{i:n,i}\|_2.$$

Let $j = \arg\max_i \mathrm{dist}_i$. We take a slightly different approach to permuting the columns than was used in CH. Once j is determined, we set

$$\bar{y}_{1:n-1} := \bar{y}_{1:n-1} - r_{1:n-1,j} x_j.$$

Then we simply remove the j^{th} column from R, and restore it to upper triangular using Givens rotations. We then apply the same Givens rotations to the new $\bar{y}$. In addition, we must also update the inverse matrix G. This is very easy, we can just remove the j^{th} column of G and apply the same Givens rotations that were used to restore the upper triangular structure of R. To see this is true notice that removing column j of R is mathematically

equivalent to rotating j to the last column and shifting columns $j, j+1, \ldots, n$ to the left one position, since we will only consider columns $1, 2, \ldots, n-1$ in subsequent steps. Suppose P is the permutation matrix which permutes the columns as described, and V^T is the product of Givens rotations to restore R to upper-triangular. Let $\hat{R} = V^T RP$ and set $\hat{G} = \hat{R}^{-T}$. Then

$$\hat{G} = (V^T RP)^{-T} = V^T R^{-T} P = V^T GP.$$

This indicates that the same V and P, which are used to transform R to $\hat{R}$, also transform G to $\hat{G}$. Since $\hat{G}$ is lower triangular, it is easy to verify that $\hat{G}_{1:n-1,1:n-1} = \hat{R}^{-T}_{1:n-1,1:n-1}$. Both $\hat{R}_{1:n-1,1:n-1}$ and $\hat{G}_{1:n-1,1:n-1}$ will be used in the next step.

After this, as in the CH algorithm, we continue to work on the subproblem of size $n-1$. The advantages of using the ideas from CH are that we always have a lower triangular G whose dimension is reduced by one at each step and the updating of G is numerically stable as we use orthogonal transformations. The lower triangular structure of G and the upper triangular structure of R allow us to save computation as the algorithm progresses. The pseudocode of the new algorithm is given in Algorithm 3.

Here we consider the complexity analysis of the new algorithm. If we sum the costs in Algorithm 3 over all loop iterations, we get a total of $\frac{7n^3}{3} + 2mn^2$ flops in the worst case. The worst case is very unlikely to occur, it arises when $j = 1$ each iteration of the outer loop. In the average case however, j is around $k/2$ and we get an average case complexity of $\frac{4n^3}{3} + 2mn^2$ flops. In both cases, the complexity is less than the complexity of the modified SW algorithm.

3.2 OILS Reduction Algorithms

For the OILS problem, the most common reduction strategy is to apply the "Lenstra-Lenstra-Lovász" or LLL reduction [16] to the matrix H. There are a few ways to describe the LLL reduction process and what it means for a matrix H to be LLL reduced. In this

Algorithm 3 New algorithm

Compute the QR decomposition of H by Householder transformations: $\begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} H = \begin{bmatrix} R \\ 0 \end{bmatrix}$
and compute $\bar{y} := Q_1^T y$ $(2(m - n/3)n^2$ flops)
$G := R^{-T}$ $(\frac{n^3}{3}$ flops)
$p := 1 : n$
$p' := 1 : n$
for $k = n$ to 2 **do**
  $maxDist := -1$
  for $i = 1$ to k **do**
    $\alpha = y_{i:k}^T G_{i:k,i}$
    $x_i := \lfloor \alpha \rceil_{\mathcal{B}_i}$ $(2(k - i)$ flops)
    $\bar{x}_i := \lfloor \alpha \rceil_{\mathcal{B}_i \setminus x_i}$
    $\text{dist}_i = |\alpha - \bar{x}_i| / \|G_{i:k,i}\|_2$ $(2(k - i)$ flops)
    if $dist_i > maxDist$ **then**
      $maxDist := dist_i$
      $j := i$
    end if
  end for
  $p_k := p'_j$
  Interchange the intervals $\mathcal{B}_k$ and $\mathcal{B}_j$
  Interchange entries k and j in p'
  Set $\bar{y} := \bar{y}_{1:k-1} - R_{1:k-1,j} x_j$
  Remove column j of R and G, and return R and G to upper and lower triangular by Givens rotations, respectively, and then remove the last row of R and G. The same Givens rotations are applied to $\bar{y}$.
  $(6k(k - j)$ flops)
end for
$p_1 = p'_1$

thesis, we will look at the LLL algorithm as a matrix factorization,

$$\begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} HZ = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

, where $Q = [Q_1, Q_2] \in \mathbb{R}^{m \times m}$ is orthogonal, R is upper triangular, and $Z \in \mathbb{Z}^{n \times n}$ is unimodular. After this special QRZ decomposition (the QRZ decomposition is not unique), the matrix R is LLL reduced i.e.,

$$|r_{k-1,j}| \leq \frac{1}{2} |r_{k-1,k-1}|, \quad j = k : n, k = 2 : n \tag{3.18}$$

$$\delta r_{k-1,k-1}^2 \leq r_{k-1,k}^2 + r_{k,k}^2 \tag{3.19}$$

From (3.18) which is known as the "size reduction" condition and (3.19) which is called the "Lovasz" condition, we can easily obtain the following inequality:

$$|r_{k-1,k-1}| \leq \frac{2}{\sqrt{4\delta - 1}} |r_{k,k}|. \tag{3.20}$$

The parameter δ can take on values from the interval $(\frac{1}{4}, 1]$. Looking at equation (3.20), we can obtain some sense of why using a LLL reduced matrix R in the search process should yield a performance improvement. In many applications, we use $\delta = 1$, this is the maximum value for δ and forces the most strict ordering of the diagonal, however as δ increases the LLL algorithm may take longer to converge. From here on when we discuss the LLL algorithm, we will assume $\delta = 1$.

We know from previous discussion about the box constrained reductions that it is desirable to have large diagonal elements, with the largest possible diagonal elements toward the end, $|r_{11}| <, \ldots, < |r_{nn}|$. Unfortunately this ordering can not always be obtained. The inequality implied by the LLL conditions, (3.20) does not impose this ordering strictly but it does ensure that it is not off by too much. In [26] the authors demonstrate that some other properties of the LLL reduction also help with the search process. One of these

relates to the distribution of the diagonal. They show that the minimum and maximum diagonal element should not be too far apart. This is consistent with the requirement for large diagonal elements since the product of the diagonal is constant. They also show that we should try and reduce the super diagonal entries of the matrix, which the LLL reduction accomplishes.

These properties allows us to prune the search tree at higher levels without wasting computational effort on suboptimal solutions. Unfortunately these properties are not always possible to obtain. However, the equation (3.20) does give us a guarantee about the relative sizes of the diagonal elements and the LLL conditions give some bounds on the size of the super diagonal.

3.2.1 Computing the LLL Reduction

This section will give details on how the LLL reduction described above can be computed. It is interesting to note that the LLL reduction is not unique, other methods to compute it exist and may yield different but still valid results.

In the following paragraphs the matrix operations that will be used by the LLL reduction will be described. Once the operations which the reduction uses have been presented, the LLL algorithm will be described and pseudocode given to compute it.

Integer Gauss Transformations.

One special type of unimodular matrix is an integer Gauss transformation (IGT), which can be defined as follows:

$$Z_{ij} = I - \mu e_i e_j^T, \quad \mu \in \mathbb{Z}. \tag{3.21}$$

We would like to know how these transformations affect an upper triangular matrix R. Suppose we apply such an IGT to R from the right, this gives:

$$\bar{R} = RZ_{ij} = R - \mu R e_i e_j^T. \tag{3.22}$$

The overall effect of this transformation on the matrix R is that the j^{th} column has some integer multiple of the i^{th} column subtracted from it, therefore:

$$\bar{r}_{kj} = r_{kj} - \mu r_{ki}, \quad k = 1, \ldots, i. \tag{3.23}$$

If we take $\mu = \lfloor \frac{r_{ij}}{r_{ii}} \rceil$, it should be clear that $|\bar{r}_{ij}| \leq \frac{1}{2}|r_{ii}|$, so given a particular column j in an upper triangular matrix R, we should be able to use $j-1$ IGTs to satisfy the size reduction condition (3.18).

Permutations.

After doing IGTs, there is no guarantee that the Lovasz condition, (3.19) will be satisfied, often it is not. In this case, we must permute the columns of R in order for the condition to hold. If $|r_{k-1,k-1}| > \sqrt{r_{k-1,k}^2 + r_{k,k}^2}$, then we permute columns k and $k-1$. After performing the column permutation, R will no longer be upper triangular. To restore the upper triangular structure of R, we can apply Givens rotations as we did in section 3.1. In this case however, only one Givens rotation will be required to zero a single sub diagonal element.

After the permutation, the Lovasz condition, (3.19) will hold. After performing this permutation, we also have the guarantee that value of $|r_{kk}|$ will increase and $|r_{k-1,k-1}|$ will decrease, therefore the resulting matrix will have something closer to an increasing diagonal.

By using the above permutations and IGTs together, we can devise an algorithm to satisfy both of the LLL conditions (3.18) and (3.19). We will start by letting $H = [Q_1, Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix}$ denote the QR decomposition of the matrix H. We will work with the columns of R from right to left, starting with column $k = n$. The idea is to move to the left so that at any step k, the columns $k+1 : n$ satisfy the LLL conditions. In the k^{th} step, we start by using IGTs to make sure column k satisfies the size reduction condition (3.18),

$|r_{ik}| < \frac{1}{2}|r_{ii}|, \quad i = k-1 : -1 : 1$. If the Lovasz condition in (3.19) holds, we move to column $k-1$, otherwise column $k-1$ and k are swapped with a column permutation and R is brought back to upper triangular as described in subsection 3.2.1. After applying a column permutation, we must move back to column $k+1$ since it is possible that the permutation we applied will cause the conditions on the previous column to no longer be satisfied. When we reach column 1, we know the matrix R must be LLL reduced. Algorithm 4 describes this process. As mentioned previously in subsection 1.6.2, in theory this algorithm terminates in a finite number of iterations (although this has not been proven for floating point arithmetic). There are also some bounds on the complexity, for more detail on the complexity and convergence, see [25].

Algorithm 4 LLL Algorithm - Returns R the LLL reduced upper triangular matrix and Z a product of IGTs and permutations

Compute the QR decomposition of H: $\begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} H = \begin{bmatrix} R \\ 0 \end{bmatrix}$
$Z := I_n$
$k := n$
while $k \geq 2$ **do**
  if $k > n$ **then**
    $k := n$
  end if
  Compute IGTs so that column k satisfies $|r_{ik}| < \frac{1}{2}|r_{ii}|, \quad i = k-1 : -1 : 1$, apply the IGTs to Z and R
  if $r_{k,k}^2 + r_{k-1,k}^2 < r_{k-1,k-1}^2$ **then**
    Swap columns k and $k-1$ in R
    Swap columns k and $k-1$ in Z
    $k := k+1$
  else
    $k := k-1$
  end if
end while

3.2.2 New OILS Reduction

In section 3.1, the motivation for the permutation based reductions was given. While these algorithms make use of the box constraint, the original motivation for them does not

rely on it. We may apply these permutation based strategies to unconstrained problems by simply setting the constraints to -infinity and +infinity.

Applying a permutation based reduction algorithm directly to the matrix H however may yield results that are much worse than those given by LLL on average, this was confirmed through numerical experiments. There are a few reasons for why this is the case. First, since the permutation based reductions do not use IGTs, we can not reduce the super diagonal entries. In [26] it was shown that reducing super diagonal entries is important for the search process. Also, we can not control the distribution of the diagonal entries without IGTs, and using only permutations there are no strict guarantees on the behavior of the diagonal elements like we get from the LLL conditions, (3.20). Another way to think about this is to consider the geometry. The permutation based reduction only reorders a given set of basis vectors in an attempt to optimize the ordering for the search. The LLL algorithm actually finds a new, better set of basis vectors. See Figure 1–1 for a picture of what this may look like. The LLL algorithm however has no knowledge of the input vector y, since we have seen that the optimal ordering for the columns of R depends on y, it is reasonable to assume that we should be able to find better column orderings for the search process than the one which LLL gives.

The basic idea behind the proposed solution is simple, apply the LLL reduction to find a new set of basis vectors, then apply the permutation strategy given in subsection 3.1.6 to reorder the basis vectors. We will call this reduction strategy "LLL+PERMU" from here on for convenience. Unfortunately the new upper triangular matrix after the permutation reduction is applied may no longer be LLL reduced. In many cases after applying the LLL+PERMU strategy we will see a very significant decrease in search time compared to LLL reduction alone, although other times there will be no significant difference. The difference in search time between LLL and LLL+PERMU depends both on the matrices and the vector y (which has a random component). Numerical experiments indicate that when the matrix H is generated in some ways, and the standard deviation of the noise is within

a certain range, we should apply the permutation reduction, but not when H is generated in some other ways, or when the noise is too high. The performance improvement in the search process can be quite significant on average for some practical cases.

Denote the Babai integer point from Chapter 2 by the vector z_0. This is the first point found during the SE search process. If the initial radius β is chosen to be larger than the residual

$$\|Rz_0 - \bar{y}\|_2 ,$$

then β will not prune any nodes from the search tree. This means that the Babai integer point effectively defines the initial radius of the search process when the initial value chosen for β is too large. Since the Babai integer point is usually a good estimate at the ILS solution and is cheap to compute, often we simply use an initial $\beta = \infty$ which for the search process is equivalent to using the residual from the Babai integer point. The number of nodes that the SE search visits in the search tree is strongly related to the initial radius, the ordering of the columns and the shape of the lattice (the shape of the lattice defines how many integer points are within the sphere defined by the radius). It is obvious how the initial radius and shape of the lattice relate to the search process, a smaller radius for a fixed lattice results in a smaller search space. If the lattice points are densely packed, even a small sphere radius could include many of them. To see this, again consider Figure 1–1. Looking at the "Babai Point Radius" in the figure, we can see that there are many potential solutions that lie within it, these would all be valid candidate solutions if we had started with the "Babai Point Radius" as our initial radius. If this initial radius had been smaller, there would be fewer potential solutions and therefore the search space would be smaller as well.

We will usually obtain different Babai integer points from the two reduction strategies, LLL and LLL+PERMU. Since we know that the search time is related to the initial radius which is often defined by the Babai integer point as mentioned above, one way to estimate whether the search process will proceed faster after permutations is to compare the initial

radius in both cases. Figure 3–2 shows such a comparison. Looking at Figure 3–2, we can see that after permutations, the Babai integer point in this case is at least as good as the one before permutation. Often it is significantly better. This is because the permutation based reduction strategies work directly with the Babai integer point which depends on the vector $\bar{y}$. With the permutations, we are trying to maximize the probability that the Babai integer point is the ILS solution. The result is that the Babai integer point should have smaller residual in many cases.

Our goal is to complete the search process as quickly as possible. At this point we have two ILS problems which have the same optimal solution, one is LLL reduced and the other comes from applying the permutation strategy to the first, e.g., LLL+PERMU. These two problems will often have different residuals to the Babai integer point. A good strategy to try and optimize the search process is to compare the residual to the Babai integer point from these two equivalent ILS problems and perform the search process on the one with the smaller residual. Numerical results comparing this strategy to performing the search on the LLL reduced matrix each time can be found in section 3.2.3.

Unforunately, there are certain special types of ill conditioned problems where the above strategy does give an improvement in search speed. Even though the new ILS problem after LLL+PERMU gives a better Babai integer point, the search process may visit a slightly larger number of nodes. Some characteristics and examples of such problems are given in section 3.2.3. In these cases, we may use the permutation strategy just to help set an initial search radius β as was discussed in Chapter 2. Then we may perform the search on the LLL reduced matrix R.

We may use the residual from the Babai integer point obtained from the LLL+PERMU reduction directly as the initial β in the LLL reduced problem. Using this residual in the search process on the LLL reduced problem when it is smaller than the residual to the Babai integer point given by LLL reduction alone is guaranteed to result in a shorter search time. From here on we will refer to this strategy as LLL+BABAI. Since the permutation

reduction process is very fast compared to the search, usually the overall run time, which is the time for the search plus the time for reduction, will be faster as well. There are some cases where we obtain better results by using the matrix from the LLL+PERMU strategy in the search, and some where it is better to use the LLL+BABAI strategy. Some comparisons can be found in section 3.2.3.

3.2.3 OILS Numerical Experiments

In this subsection, some numerical results will be presented to compare the speed of the search process after applying the reduction strategies presented in section 3.2. We will compare the LLL reduction, LLL reduction plus permutations (LLL+PERMU), and the LLL reduction using the initial radius β given by the permutation strategy (LLL+BABAI). For all of the search time results the time to find the Babai integer point is not included.

Figure 3–2 which was discussed in the previous subsection shows the residual to the Babai integer point before and after the PERMU strategy has been applied. This figure shows numerical results for 200 matrices $H \in \mathbb{R}^{50\times 50}$ generated randomly with their elements picked from a normal distribution, $N(0,1)$. The elements in the vectors $x \in \mathbb{Z}^n$ were picked uniformly on the interval $[-10, 10]$ and the vector y was generated as $y = Hx + v$, where v is normally distributed with mean 0 and standard deviation 0.5 in this case.

Figure 3–3 shows the average search time when the three reduction strategies (LLL, LLL+PERMU, LLL+BABAI) are used to reduce some normally distributed, random square matrices $H \in \mathbb{R}^{n\times n}$ with $n = 35 : 5 : 50$. The horizontal axis displays varying values for σ which is the standard deviation of the noise vector v. The vertical axis is the average time taken for the search process over 200 runs where on each run a new matrix H, vector x and noise vector v were generated.

Looking at Figure 3–3, we see that when the noise is small, the search process is too fast for any extra reduction to make a difference. As the noise becomes larger, LLL+PERMU

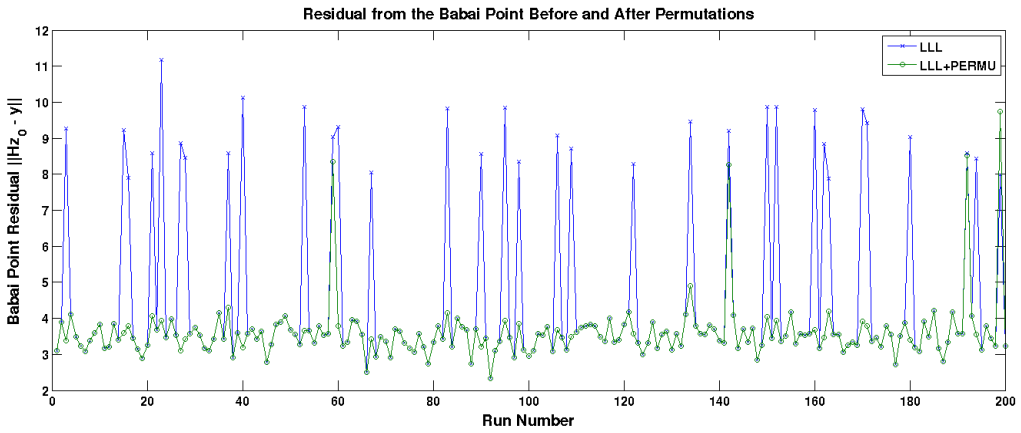

Figure 3–2: LLL Reduction vs LLL+PERMU. Residual from the Babai points.

and LLL+BABAI start to offer a significant advantage in terms of search time. LLL+PERMU and LLL+BABAI perform similarly at moderate levels of noise in this case. As the noise rises further, the search time after LLL+PERMU seems to become about the same or slightly worse than after LLL, however LLL+BABAI maintains a significant advantage in all cases.

To better understand the behavior of the search process with the three reduction strategies, Figure 3–4 shows the search time for 200 runs on a few select problem sizes and levels of noise. Notice that when the noise is at a lower level like 0.5, the LLL reduction has many spikes in its runtime, where LLL+PERMU and LLL+BABAI encounter these spikes much less frequently. Most of these spikes can be explained by looking at the residual to the Babai integer point. When there is a spike in the LLL time it is usually the case that the Babai integer point after LLL reduction resulted in a large residual, while the LLL+PERMU and LLL+BABAI reductions gave a smaller residual to the Babai integer point. The reason for this was explained in subsection 3.2. When the noise becomes larger like around 0.8 we see large variance in the runtime from all three reduction methods. This is because when the noise becomes too large, the difference between the residuals to the Babai points between LLL and LLL+PERMU becomes smaller.

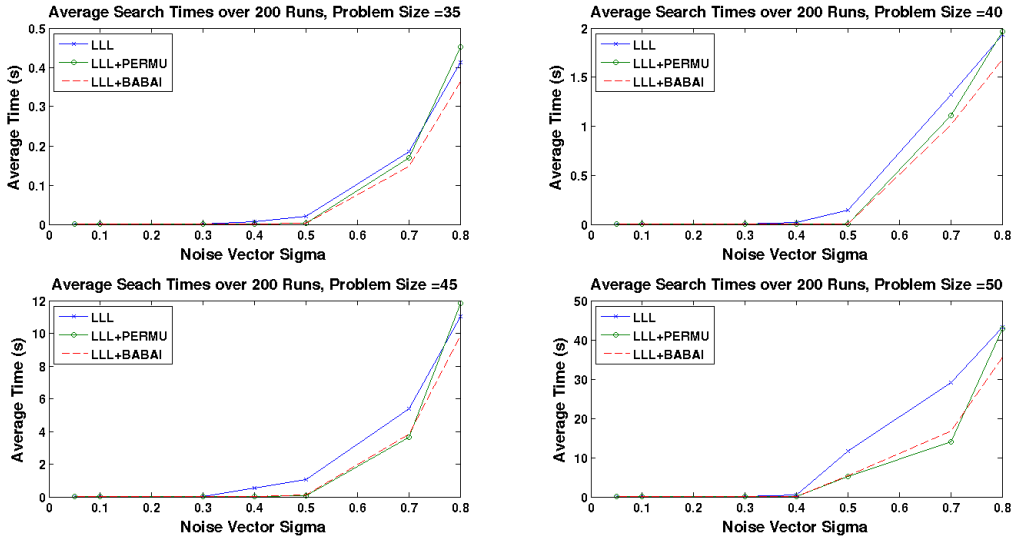

Figure 3–3: LLL Reduction vs LLL+PERMU vs LLL+BABAI. Average times over 200 runs for various problem sizes and noise levels.

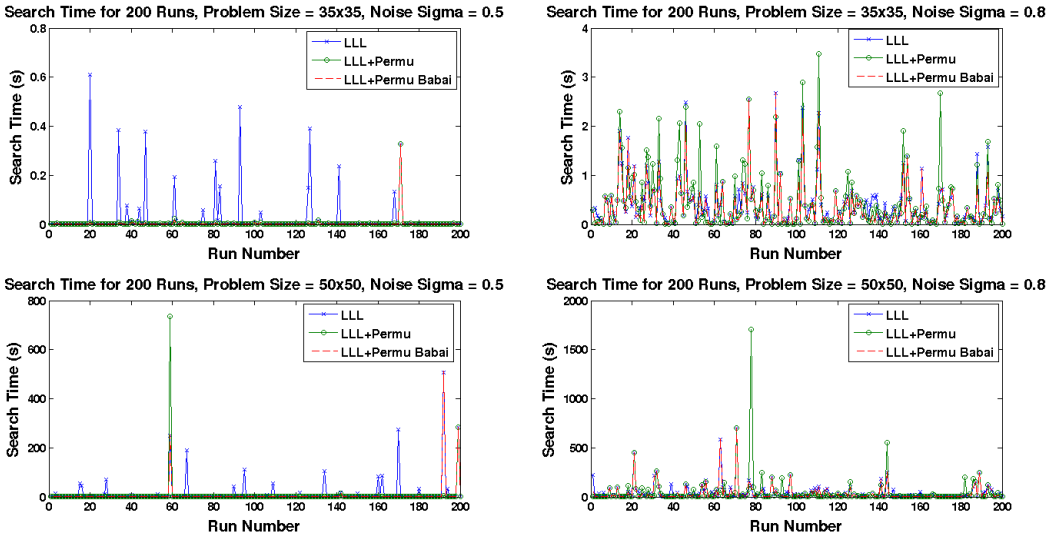

Figure 3–4: LLL Reduction vs LLL+PERMU vs LLL+BABAI. Search time for 200 runs with some different sizes and noise levels.

Table 3–1: Success Rate (out of 200) for LLL and LLL+PERMU on various problem sizes and levels of noise.

	35×35		40×40		45×45		50×50	
	LLL	LLL+PERMU	LLL	LLL+PERMU	LLL	LLL+PERMU	LLL	LLL+PERMU
0.05	200	200	200	200	200	200	200	200
0.1	200	200	200	200	200	200	200	200
0.3	199	200	200	200	200	200	200	200
0.4	192	200	196	200	193	200	196	199
0.5	178	198	175	198	175	194	167	196
0.7	89	132	75	120	79	141	89	137
0.8	44	76	45	77	42	86	44	94

Table 3–1 shows how many times out of 200 runs that the Babai integer point was equal to the true solution x for both LLL reduction and LLL+PERMU. The vertical axis of the table shows the various levels of noise, σ, and the horizontal gives the different problem sizes. If we take this as a percentage it is known as the "success rate", it gives the probability that the Babai integer point is equal to the true parameter vector. This is an important number for GPS applications where we are required to solve OILS problems. In GPS the success rate can be estimated based on the problem. If the estimated success rate is high enough, the expensive OILS problem is not solved, the Babai integer point is simply used as an estimate for the parameter vector instead. In communications applications the success rate is generally not considered, instead there are other measure like the "bit error rate" and "symbol error rate". The success rate is shown here because we are considering OILS problems and communications applications deal with BILS problems. Here we see that the LLL+PERMU strategy can offer a significantly better success rate in all cases tested; even in cases where the search process may be slower on average if we use the permuted R.

Looking at these numerical results, we see that when the noise is both too low, or too high there seems to be no big difference between the reduction strategies. When the noise is too low this is because the problem becomes too easy, all three reductions result in a very fast search speed. When the noise is too high the explanation is a little bit more

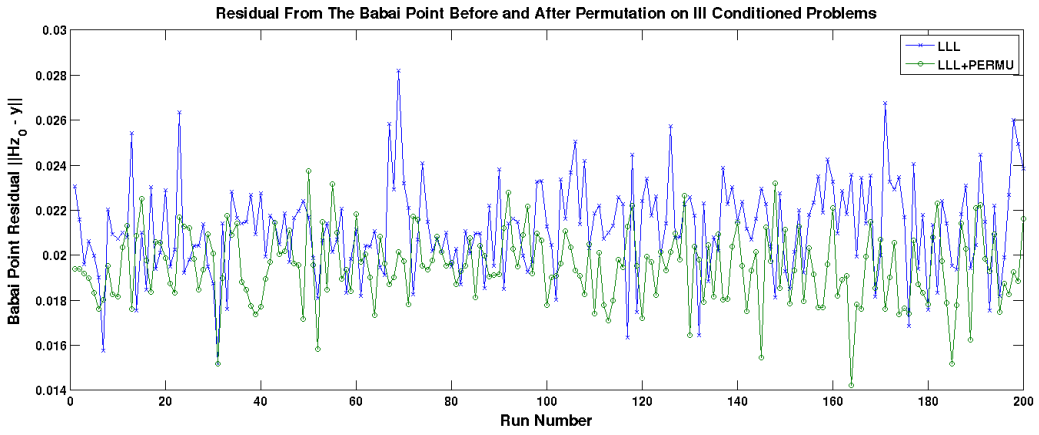

Figure 3–5: LLL Reduction vs LLL + PERMU. Residual from the Babai points on ill conditioned problems.

complicated. We have performed numerical experiments which concluded that as the noise becomes high, the new upper triangular matrix given by LLL+PERMU becomes farther from a LLL reduced matrix, e.g., the Lovasz condition (3.19) is less likely to be satisfied. Since we know that the properties of LLL are desirable in the search process, it makes sense that if they are lost the search may slow down. This problem still needs further investigation for a more rigorous explanation.

The next set of numerical results demonstrates a case where the permutation strategy should not be used. In Figures 3–5 and 3–6 the matrix H is generated as follows. We construct a diagonal matrix $D = diag(d_{ii})$, where $d_{ii} = 10^{-\frac{(i-1)*4}{(n-1)}}$. Recalling that the condition number of a matrix is the first singular value divided by the last one; a matrix with condition number 10^4 is constructed by forming the product of the SVD, $H = UDV^T$. Here U and V are the Q factors from the QR decomposition of two random matrices. We generate x and y in the same way as the previous numerical experiments.

Figure 3–5 shows the residual for the Babai integer point both before and after permutation where $H \in \mathbb{R}^{40\times 40}$ and $\sigma = 0.5$. Comparing to Figure 3–3, we notice that it is different. First, the residual both before and after permutation is much smaller in magnitude. The residual after permutation is still slightly better than before on average, but the difference is much less significant than Figure 3–3. The residual to the Babai integer point

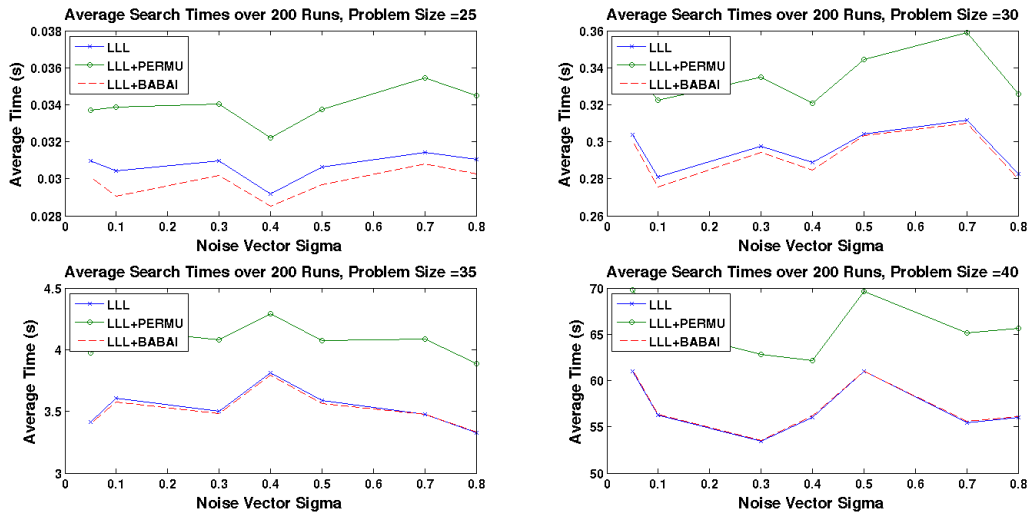

Figure 3–6: LLL Reduction vs LLL +PERMU vs LLL + BABAI, average search time on ill conditioned problems.

after LLL reduction alone has less variance relative to LLL+PERMU compared to Figure 3–3. Out of the 200 runs in Figure 3–5, not once was the Babai integer point equal to the true solution; compared to the corresponding example in Figure 3–2 where $167/200$ times the Babai integer point was the correct solution before permutations were performed.

Figure 3–6 shows the average search times for these ill conditioned problems. We see that LLL+PERMU is consistently slightly worse than the other reduction strategies. In such ill conditioned cases, using LLL+BABAI is the best strategy.

Chapter 4
Alternate Search Strategies

In this chapter some more search strategies will be reviewed. The best first search which has been mentioned previously will then be presented in detail. In the literature there have been a few attempts to control the memory usage of the BFS, one of these will be looked at and a new one will be presented along with some numerical results.

Chapter 2 described the most common search strategy used to solve both the OILS and BILS problems. Also, the search process was shown to be equivalent to a tree search problem where we must find the minimum cost leaf in a tree of depth n with potentially exponential width.

While the depth first search corresponding to the SE algorithm is usually very fast, it is not optimal in terms of the number of nodes visited during the search process. Let $x^p_{i:n}$ denote the node in the search tree that results from fixing $x_{i:n}$ to some particular set of values, where the superscript p stands for partial. We may call this a partial solution. The squared partial residual given by this partial solution is just its cost in the search tree and can be defined as

$$\|R_{i:n,i:n}x^p_{i:n} - \bar{y}_{i:n}\|_2^2 .$$

Consider the search problem only as a tree search, assume we have the nodes and edges with their weights but have no other knowledge of the problem. Then we can define an optimal algorithm for an ILS problem as one which visits only nodes in the search tree with partial residuals that are less than or equal to the optimal ILS solutions residual. Therefore all nodes visited by an optimal search process satisfy the following inequality:

$$\|R_{i:n,i:n}x^p_{i:n} - \bar{y}_{i:n}\|_2^2 \leq \|Rx_{ILS} - \bar{y}\|_2^2 . \tag{4.1}$$

Here R and $\bar{y}$ come from reducing an ILS problem defined by H and y, and x_{ILS} denotes the optimal ILS solution, which depending on the noise may not be the same as the original integer vector.

Assuming no other knowledge of the search problem beyond the tree, any search algorithm must at least visit this set of nodes to guarantee there is no other leaf in the tree with a smaller residual than the one found (recall from Chapter 2 leaves all lie at depth n). This means all optimal search algorithms as defined above visit the same set of nodes but possibly in a different order. The SE search presented in Chapter 2 is not optimal since some of the nodes which it visits may not satisfy (4.1).

There are in fact ways to reduce the number of nodes visited so that we may visit fewer nodes than the optimal algorithms as defined above. One technique involves using lower bounds on the optimal solutions residual to prune nodes from the search tree, see e.g., [18]. Unfortunately our numerical tests indicate that existing methods which use this technique incur too much per-node overhead to provide faster solutions to *reduced* OILS and BILS problems.

One algorithm that satisfies the requirement in (4.1) is called the "Best First Search", which will be referred to as BFS from here on for convienience. This algorithm has been proposed in the literature a number of times, [11] and [23] are two examples, although they appear different on the surface. Later in this chapter a detailed description of the BFS will be given. For now it is enough to note that the BFS pays a price to achieve the goal of visiting the minimum number of nodes, it must permanently store each node visited during the search in memory and always keep track of the node with minimum cost (where cost is defined as in Chapter 2). Also, when the nodes are later accessed from memory, since they are not stored in a simple manner like the SE search, it is likely that we will incur cache penalties when accessing them. For these reasons it is not always practical to use the BFS strategy since hardware applications may have very limited memory and slow processors.

Unfortunately, some of the literature such as [8] does not fully consider the overhead of finding the minimum cost node when comparing search algorithms, instead only considering limiting the memory usage of BFS while still visiting as few nodes as possible. Even though such algorithms may visit fewer nodes than an SE search, the extra time spent at each node may cause them to still run slower. In section 4.2 the algorithm from [8] will be described. Comparing the number of nodes visited by two different search processes may not always be meaningful if one process spends much more time on each node than the other.

The following sections will explain the BFS, quickly overview a couple of attempts that have been made to control the memory usage of the BFS, and then propose a new idea for combining the BFS with the SE search in order to limit memory usage and decrease computational complexity.

4.1 Best First Search

Recall from Chapter 2 that we will explore the search tree which has depth n and each edge has a weight. Define a nodes cost as the sum of the weights of all edges traversed to reach the node, that is the length of the path between the node and the root. This cost is equal to the partial residual squared for fixing $x_{i:n}$ to a set of values, i.e.,

$$\|R_{i:n,i:n}x^p_{i:n} - \bar{y}_{i:n}\|_2^2 .$$

We would like to find the leaf with minimal cost. For the BFS, we define a node's "next best child" as the child of that node with the lowest cost that has not yet been visited. This is simple to compute; the first "next best child" for a node is just $\lfloor c_k \rceil$ as defined in Chapter 2. The second "next best child" for the node is $\lfloor c_k \rceil \pm 1$, such that we get the second nearest integer to c_k, we proceed in this way, always taking the next nearest integer to define the "next best child". This is the exact same strategy that the SE search uses.

The core data structure used to efficiently implement a best first search is called a priority queue. The priority queue is an abstract data structure whose basic operations are $insert(element, cost)$, and $findmin()$. The $insert(element, cost)$ operation adds an element to the queue with some real value cost. The $findmin()$ operation finds and removes the element with minimum cost from the queue. The implementation details are not particularly relevant to this thesis, but it is important to note that there are various implementations used in practice. The usual cost for the $insert(element, cost)$ operation is $\theta(1)$ and for the $findmin()$ operation $\theta(log(N))$ where N is the number of elements currently in the queue. For more information on priority queues and their implementation see [2].

As mentioned above, for an ILS application, we can quickly find the next best child of a given node, it is the node which corresponds to the next choice of x_k chosen as in the SE algorithm. The "first best child" of a node at level $k-1$, assuming $x_{k-1:n}$ are fixed, is always the node corresponding to $x_k = \lfloor c_k \rceil$.

In the first step of the BFS, we initialize an empty priority queue and define the root node as having a cost of 0 and being at level $n+1$. The "next best child" of the root node corresponds to $x_n = \lfloor \bar{y}_n / R_{n,n} \rceil$ and the cost to visit this child is equal to the partial residual squared $(R_{n,n}x_n - \bar{y}_n)^2$, we call this cost the "next best child cost". The root node will store its current "next best child" and "next best child cost" and then be inserted into the priority queue. Elements in this priority queue are always sorted by their "next best child cost".

In the next step, we visit the first child of the root, x_n. First, we perform the $findmin()$ operation on the priority queue, since currently the root is the only element, it will be returned. We would like to visit the first child of the current node (which is now the root). To visit a node involves calculating the "next best child" and the "next best child cost", therefore we must compute these quantities for the node corresponding to x_n (which is the first child of the root). The "next best child" of x_n will be given by $x_{n-1} = \lfloor c_{n-1} \rceil$ and its

cost is the partial residual squared:

$$\left\| R_{n-1:n,n-1:n} x^p_{n-1:n} - \bar{y}_{n-1:n} \right\|_2^2 .$$

We then insert x_n into the priority queue with its "next best child" and "next best child cost". Notice that if we expand the cost as follows, $(R_{n,n}x_n - \bar{y}_n)^2 + (R_{n-1,n}x_n + R_{n-1,n-1}x_{n-1} - \bar{y}_{n-1})^2$ then the first term in the cost is just the cost of the parent node; this means to calculate the cost for a child at level i, we only need to add the $(n-i+1)^{th}$ term from the sum defined by

$$\|R_{i,i:n}x^p_{i:n} - \bar{y}_i\|_2^2$$

to the cost of the parent.

Since we are now visiting the first child of the root, we must generate the new "next best child" for the root node, $x_n = \lfloor c_n \rceil \pm 1$ and the cost for this child (also the partial residual), $(R_{n,n}x_n - y_n)^2$. We may now insert the root back into the priority queue with the newly calculated "next best child cost" and "next best child".

At this step, the next node to be visited will be the one at the top of the priority queue with the smallest "next best child cost". Currently there are two nodes in the priority queue, one at level $n+1$ (the root) and one at level n. If the former has a smaller "next best child cost", we will visit the second best child of x_{n+1}, the new x_n next. Otherwise we will visit the first child of the node x_n, which is at level $n-1$.

By proceeding in this way we always visit the nodes in the order of increasing partial residual. The next node we visit is always the one with the next smallest partial residual (in the whole tree) from the previous one, even if those 2 nodes are at different levels. This should be obvious, consider the case where there exists a node in the tree that we have not yet visited which has a smaller partial residual than the current node at the top of the priority queue. This means that the parent of the node in question also had a smaller partial residual than the current node and its grandparent etc. up to the root. If this were the case, however, we would be visiting that node instead of the current node, so it is a

contradiction. In this way we can guarantee that the first time we find a leaf in the tree, it must be the leaf with the smallest squared residual and is therefore the solution. Also, at the point where we find a leaf, we know we have visited only and all of those nodes that have a partial residual within the radius defined by the optimal solutions residual.

It should also be noted that trivial modifications to this BFS search algorithm can be made so that it will solve the BILS problem. We just need to make sure that at every step, the value chosen for x_k is within the proper bounds defined in equation (1.5).

4.2 Controlling BFS Memory Usage

In section 4.1, notice that in each step a new node is added to the priority queue, but nodes are never removed (the $findmin()$ operation removes a node, but we just update the cost and insert it back in). This means that each node visited during the search is kept permanently in the priority queue. Since the number of nodes visited can potentially be exponential in n this can become a problem for two reasons. The first and most obvious reason is memory usage. The second which is often overlooked is that, as the priority queue grows, the computational complexity to maintain it is increased at each step; the computational cost for each operation is $\theta(log(N))$, where N is the number of nodes visited so far. Also consider that the computational cost to visit a node in an efficient implementation of the SE algorithm is only about $2k$ flops, where k is the level of the node in the tree. The BFS must still perform the operations the SE performs in addition to maintaining the priority queue.

With the rough analysis of the computational cost in the previous paragraph, it should be obvious that just because the BFS visits fewer nodes than SE, it does not mean it will necessarily be faster. In both [28] and [8] the authors try to find a balance between the number of nodes that we keep in memory at any given time, and visiting the smallest number of nodes possible; this section will briefly describe the approach taken in [8]. Note that the algorithm described here is slightly different in that an extra parameter has

been eliminated, according to the authors results, this parameter seems to always make the performance worse anyways.

Their algorithm is only slightly different from the BFS. The authors do not use the concept of a priority queue, and instead use ordered lists to find the node with minimal cost at each step. Suppose we have a list S in which nodes are stored, also this list is sorted by each nodes level in the search tree (lower levels are toward the back). This list, like the priority queue in the BFS starts with only the root node. At each step, a new node is added to the back of the list by finding the node with minimal cost in S (this takes $|S|$ operations) and visiting its "next best child". Now suppose we allow the user to specify some parameter α. Instead of scanning the whole list S to find the node with the lowest "next best child cost" to visit next, we simply look at the last α elements in S (which correspond to the α lowest nodes in the tree) and make our decision based on these. Such a strategy forces the BFS to proceed down the tree much faster than it would otherwise.

Unfortunately, when a leaf is reached, we no longer have the guarantee that it is the optimal solution. We do know however that we no longer have to consider any of the last α nodes in S since they all must have a cost greater than the cost of the leaf we had just found. We may remove the last α nodes in S, update the search radius to be the residual given by this leaf, and continue the search process. Any time a node is visited with a "next best child cost" that is greater than the current search radius, we may discard the last α nodes in S. When S is empty we may terminate with the optimal solution.

The behavior of this algorithm is such that when $\alpha = 1$ it degrades to a SE search. When $\alpha = \infty$ it becomes a BFS. Unfortunately implementing a BFS using a list is very inefficient as will be discussed later. The authors also give some good bounds on the amount of memory this algorithm requires based on how the parameter α is chosen. They present some results, but do not give flop counts or CPU time, instead focusing on memory usage and the number of nodes visited.

There are a few drawbacks to this algorithm. One is that it is not clear how to implement this in practice. Consider setting the parameter α to a relatively high number. When α is higher, we will visit fewer nodes since it will be closer to the BFS, and we will discard more nodes each time we have the opportunity to discard. Unfortunately as α gets larger, a naive list based implementation becomes impractical. Scanning through α elements in a list at each step could impose significant overhead. Also consider that S must be sorted by the levels of the nodes in the search tree. If we wish to add a new node which has a level larger than the smallest leveled nodes currently in S, we must move all of the lower nodes in the list one place to the right in memory, again this could take α operations. This suggests that we may want to use a priority queue based implementation for large alpha and a list based implementation for small alpha. A priority queue implementation of this algorithm is not very straight forward however since we have the extra condition that the node at the top of the priority queue should have the lowest cost and a level in the tree corresponding to the parameter α.

With these problems, it is worthwhile to explore some other options to limit the memory usage of the BFS, while trying to decrease the amount of time spent processing each node in the tree.

4.3 Combining BFS and SE Search

Here a new method for controlling the memory usage of the BFS will be presented. Like the method mentioned in the previous section [8] it is a modification to the BFS which relies on a parameter supplied by the user.

One of the main problems with the SE search occurs when elements high up in the tree (e.g. $x_n, x_{n-1}, \dots$) are initially chosen incorrectly. We must explore the entire subtree looking for the ILS solution, visiting many nodes that may have partial residuals higher than the residual given by the ILS solution. Eventually we make our way back up the tree

to correct the mistake. The BFS doesn't have this problem since it does not restrict itself to visiting only children of the current node.

In order to avoid making expensive mistakes at high levels, we may consider dividing the search tree into two parts, cutting it at some level α which is to be supplied by the user. Nodes with level greater or equal to α will belong to one part and less than α another. For example, consider figure 2–1. If we set $\alpha = 2$, the nodes labeled $1, 6, 2$ and 4 would belong to the top group, while nodes 3 and 5 would belong to the bottom. We begin with a BFS on the top part of the tree, when the BFS reaches a node at level $k = \alpha$, it uses an SE search to find the optimal solution in the subtree of that node. Suppose we are at some node in the BFS which is given by fixing $x_{k:n}$ to some particular values (so we are at level $k = \alpha$), then we solve the subproblem given by the following using the SE search.

Let

$$\hat{y} = \bar{y}_{1:k-1} - R_{1:k-1,k:n} x_{k:n},$$

where $\bar{y}$ and R come from one of the reduction strategies described in Chapter 3. Note that the search process in Chapter 2 also combines the known part of x with R and $\bar{y}$ as it progresses in the same way. Then assuming that $x_{k:n}$ are fixed, we would like to find the $x_{1:k-1}$ to minimize the following:

$$\min_{x_{1:k-1} \in \mathbb{Z}^{k-1}} \|\hat{y} - R_{1:k-1,1:k-1} x_{1:k-1}\|_2.$$

Call the estimate $x_{1:n}$ which is given by combining the solution to the above subproblem ($x_{1:k-1}$) with the nodes found by the BFS ($x_{k:n}$) the "current best integer point". We can continue the BFS until we reach another node at level α. If at some point during the BFS the next node in the priority queue has a "next best child cost" greater than the cost of the current best integer point found by a previous SE search, we may terminate the search process knowing that the current best integer point is the optimal ILS solution. When the BFS eventually visits another node at level α, we again perform an SE search to find the

optimal solution in the subtree of this node, however this time we may use a sphere constraint to terminate the SE search early i.e., we can set a good initial β. We can use the current best integer point to set the initial β for the search on this new subproblem. To do this we simply set the initial β for this new search equal to the cost of the current best integer point minus the cost of the node at level α we are visiting, where we will start our SE search. This ensures that any leaf found by the current SE search must have a lower cost than the current best integer point. If this SE search finds a leaf, we may update the current best integer point to correspond to this new leaf. It is possible that the current SE search will terminate without finding a leaf, this means that the corresponding subproblem does not yield a better solution than the current best integer point. Continuing this process, eventually we will reach a point where the next node to be visited in the BFS has a cost greater than the current best integer point, at this point we may terminate with the optimal solution.

With this approach and an appropriate choice of the parameter α, we can hope to avoid making expensive mistakes at higher levels in the search tree. We also will incur a much smaller overhead to maintain the priority queue, since it will not contain nearly as many nodes if α is chosen to be near n, and with the smaller priority queue, there will be less memory usage.

There are some useful tricks which can be used in conjunction with the combined search process to improve performance. Consider that the first SE search will very often be the one that costs the most in terms of run time. This is because we do not have a good value for an initial β to use as a stopping condition and must solve the subproblem completely, even if the subproblem may be very ill conditioned. The best case occurs when the first subproblem we solve yields the ILS solution, in this case we get the smallest possible sphere radius to use in subsequent SE searches as a stopping criteria, and therefore we can prune more nodes from the search tree. To make this more likely to occur, we may delay solving the first subproblem as follows. When we visit a node at level α, we can

decide not to solve the subproblem stemming from this node right away. Instead we put the node into a list of "unsolved subproblems" and continue the BFS. One example of when we may not want to solve a subproblem immediately is when the Babai integer point given by that subproblem has a large residual. Eventually we need to perform the SE search to find a leaf. Once a suitable node is visited by the BFS (one with a relatively low residual to the Babai integer point is a good choice), we perform the SE search and find a leaf in the tree corresponding to a potential solution. Now we simply go back to the list of "unsolved subproblems" and solve them all, using the residual from the previous solution as an initial sphere radius. The unsolved problems that had Babai points yielding high residuals are now likely to terminate much more quickly because not many nodes should be within the current sphere radius if the subproblem that we chose to solve gave a good solution. This strategy has not been explored in full detail and still requires some future research on the best way to utilize it.

Finally, it should be mentioned that while this combined BFS and SE search strategy has been described in a way to solve OILS problems, it is also applicable to BILS problems. Only minor modifications must be made in order to ensure the elements of x satisfy the constraints defined in (1.5).

The advantage of this new algorithm over the one presented in [8] which was described in section 4.2 is that the algorithm in [8] has no clear and efficient implementation. The implementation you should use depends how you would like to choose the parameter α. Even for a fixed α it is not always obvious exactly how the algorithm should be implemented efficiently. The new algorithm presented here does not have this problem. The implementation is straight forward and should always use a priority queue, no matter how the parameter is to be chosen.

Some strategies for choosing the parameter α based on the input problem were tested. Unfortunately choosing α based on properties of the problem has proven to be somewhat difficult, although it can be chosen reliably through numerical experiments as will

be shown in the next section. How to choose α for a given problem will be considered in future research.

4.4 Numerical Testing Results

In this section, numerical results will be given to compare the SE search, BFS and the combination of the two. To this point, there has not been much said about how to choose the parameter α. We need to choose α low enough to avoid mistakes at high levels in the tree, but high enough so that we get the advantages of the SE search being faster at processing each node. Note that when $\alpha = n$, we have a pure SE search, and when $\alpha = 1$, the combined search strategy degrades to a BFS.

For the first set of numerical results in this section, the matrices are simply generated randomly with elements drawn from a normal distribution, $N(0, 1)$. The integer vectors x have elements uniformly distributed in the interval $[-10, 10]$ and the vectors y are generated as $y = Hx + v$ where v is normally distributed with some standard deviation σ. All matrices in this section will be LLL reduced before the search process begins.

Figure 4–1 shows runtime results as the parameter α varies for 50 different randomly generated OILS problems of size 50×50. Here the noise vector has $\sigma = 0.8$, which is quite high. The noise was chosen to be high so that the runtime would be significant, when the noise is too low all of the search algorithms will be very fast. The x-axis which varies from $1, \ldots, 50$ represents the choices of the parameter α. The y-axis gives the search time in seconds, and each line down the z-axis represents one of the 50 random problems. On the far right of the graph where α is small, we have a search process equivalent to the BFS and toward the left as α gets larger it is equivalent to the SE search. Notice that on both ends there are sometimes spikes in the run time, while if we follow the line defined by $\alpha = 40$, we see that there is a consistent trough in the runtime. This is interesting and tells us that for this particular type of problem, it is best to choose the parameter α somewhere

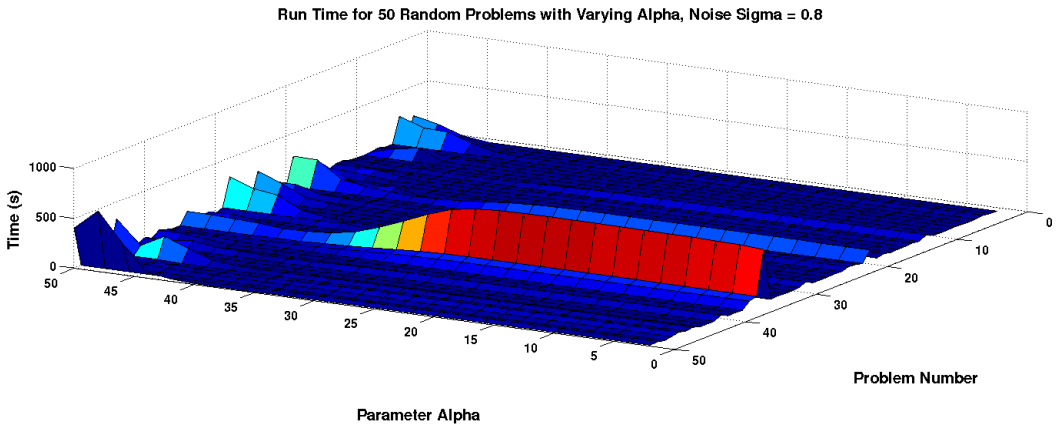

Figure 4–1: Run time results for the combined search process with varying parameter α on random problems.

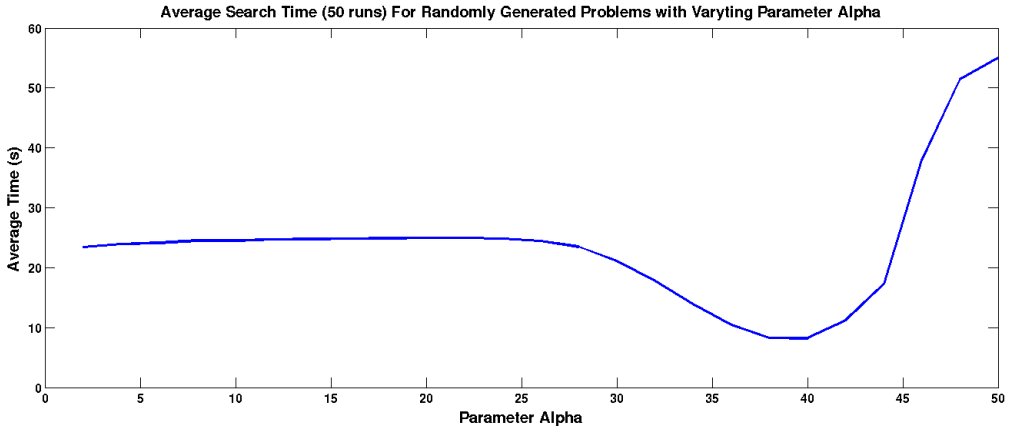

Figure 4–2: Average run time results for the combined search process with varying parameter α on random problems.

around 40. Also notice that choosing the parameter slightly incorrectly is usually not a big deal, there is a fairly wide range of good choices.

Figure 4–2 shows the mean search time over all 50 runs for each setting of the parameter α. Notice that the minimum occurs around $\alpha = 40$. This confirms our choice in the previous paragraph.

We know the search behaves fairly consistently as the parameter α changes, and there is a wide range for good choices of the parameter; to choose α for some new type of problem generated in some other way, it is recommended to simply run it a few times with various parameter settings and choose the one that performs best.

Figure 4–3 shows some numerical results from running the new combined search process on ill conditioned problems. These problems were generated in the same way as the ill-conditioned problems in section 3.2.3 and Figures 3–5 and 3–6. The problem size here is 35×35 and the noise vector was generated with $\sigma = 0.3$. Here we choose the noise vector to have a smaller standard deviation because these ill conditioned problems take much longer to solve. The best results are obtained when α is near n which means we start the SE search process immediately. These examples illustrate that the best first search is not always better than the SE search even though it visits fewer nodes in the search tree.

Some reasons for this can be found by examining the number of nodes visited during the search process. First, we look at the ratio $\frac{\textit{Nodes visited by BFS}}{\textit{Nodes visited by SE Search}}$. For the previous set of numerical results where the new combined search achieved significantly better results, this ratio was 0.44 on average, for the new set of ill-conditioned problems, it is 0.8 on average. This means that there is not as big a difference between the best first search and SE search as far as the number of nodes visited is concerned. The fact that the BFS explores 20% fewer nodes on average is not enough to offset the extra time spent on each node.

Also with these ill conditioned problems, the run times are all on a similar scale. If we look at the variances of the run times between the two figures, the variance for the ill-conditioned problems is only about 170, while the randomly generated problems have a runtime variance of nearly 5000. These variances were calculated as the variance of all of the run times in each figure. If we look at Figure 4–1 we can see that most of the time the SE search is fast, it is only 8 or 9 out of the 50 randomly generated problems where it becomes very slow. This small number of bad cases makes it much worse on average.

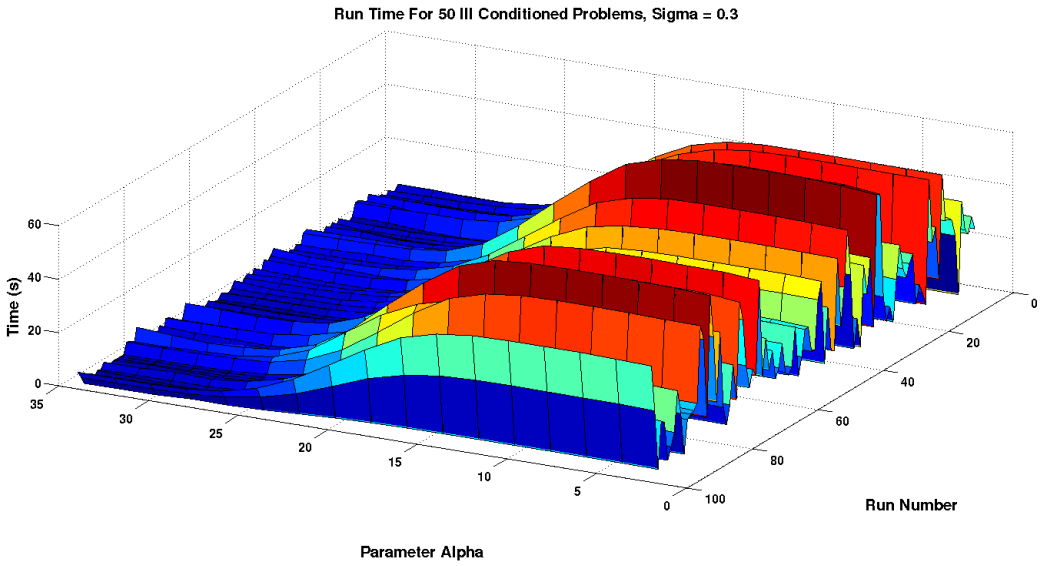

Figure 4–3: Run time results for the combined search process on ill conditioned problems with varying parameter α.

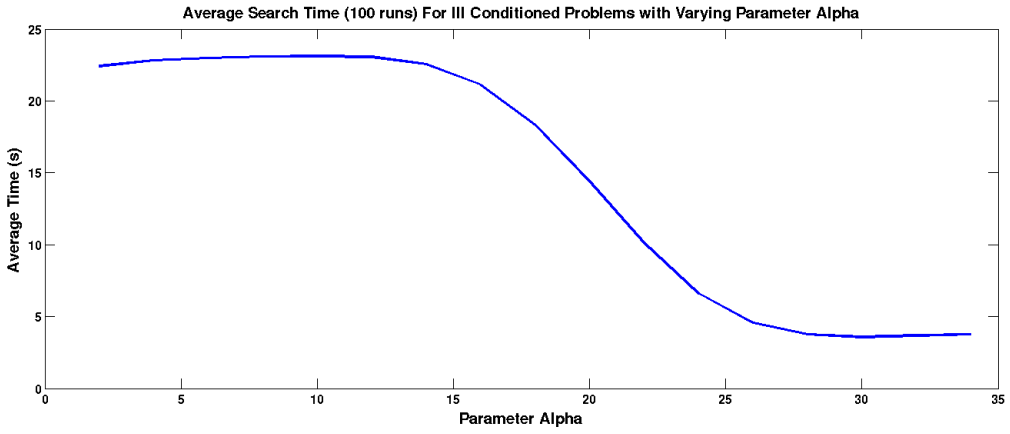

Figure 4–4: Average run time results for the combined search process with varying parameter α on ill conditioned problems.

Chapter 5
Conclusions and Future Work

This thesis focused on improving both the reduction and search processes in solving OILS and BILS problems. In Chapter 3 it was shown that two effective reduction strategies in the literature for the BILS problem are theoretically equivalent. Using the knowledge of their equivalence a new reduction strategy was proposed that uses ideas from each to compute the same solution more efficiently and in a numerically stable manner. Also in Chapter 3, it was shown that under some practical conditions, this new reduction process can be used to improve the performance of the OILS search process as well. This improvement seems to be a result of the Babai integer point given by the new reduction having both a smaller residual and higher success rate (defined as the probability that the Babai integer point is equal to the true solution x). Since the Babai integer point has a tendency to have smaller residual and higher success rate after applying these permutations, we can conclude that for both OILS and BILS problems, applications which use the Babai integer point as an estimate to the ILS solution should apply this permutations strategy in order to improve their success rate.

Some more investigation into the new reduction algorithm is still needed. The new reduction presented in subsection 3.2.2 has two phases, first we do the LLL reduction which uses IGTs and permutations together to achieve some properties in the upper triangular matrix R. The second phase applies another set of permutations to R to try and achieve some different properties. We would like to investigate reduction algorithms that try to consider both sets of these properties in a single phase instead of having two disconnected phases. This will require a deeper understanding of exactly why the new LLL+PERMU reduction gives good results.

In Chapter 4 the "Best First Search" approach was described. The BFS has the property that it visits the fewest nodes of any search algorithm. Unfortunately the BFS is not always ideal because of its high memory usage and complexity per node visited in the search tree. Previous attempts to improve upon the BFS were mentioned and a new one introduced.

A previous attempt at improving the BFS, [8], involved limiting the amount of memory used by the BFS using a user supplied parameter. Unfortunately it is difficult to implement efficiently which leads to a high per-node complexity. The new algorithm introduced in this thesis tries to address both the problem of memory usage and high complexity per node while remaining easy to implement. Run time results were given comparing this new algorithm to the SE search process and the improvement was shown to be significant for some practical cases.

It would be interesting to see if some theory could be worked out for how to choose the parameter α in the new search method. It is currently unclear how to choose it given a particular ILS problem. Since the optimal value for the parameter seems somewhat stable for fixed problem type, size and levels of noise, this may be possible. Also it is possible that we could do away with the parameter and dynamically decide to stop doing a BFS and start the SE search when some condition is met. For the time being it is recommended to simply compare a small number of test runs with various settings of the parameter in order to choose it for some specific applications.

References

[1] E. Agrell, T. Eriksson, A. Vardy, K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[3] L. Lampe, A. Schenk, R. Fischer. A stopping radius for the sphere decoder and its application to MSDD of DPSK. *IEEE Communications Letters*, 13:465–467, 2009.

[4] Å. Björck. *Numerical Methods for Least Squares Problem*. Philadelphia: SIAM, 1996.

[5] S. Breen, X.-W. Chang. Column reordering for box-constrained integer least squares problems. In *IEEE Globecom 2011*, 2011.

[6] X.-W. Chang, Q. Han. Solving box-constrained integer least-squares problems. *IEEE Transactions on Wireless Communications*, 7(1):277–287, 2008.

[7] X.-W. Chang, C.C. Paige. Euclidean distances and least squares problems for a given set of vectors. *Applied Numerical Mathematics*, 57:1240–1244, 2007.

[8] W. Fichtner, C. Studer, A. Burg. A unification of ml-optimal tree-search decoders. In *In Proceedings of IEEE Signals, Systems and Computers*, 2007.

[9] R. E. Cline. Representations for the generalized inverse of a partitioned matrix. *Journal of the Society for Industrial and Applied Mathematics*, 12(3):588–600, September 1964.

[10] G. J. Foscini, G. D. Golden, R. A. Valenzuela, P. W. Wolniansky. Simplified processing for high spectral efficiency wireless communication employing multi-element arrays. *IEEE Journal on Selected Areas in Communications*, 17(11):1841–1852, November 1999.

[11] T. Fukatani, R. Matsumoto, T. Uyematsu. Two methods for decreasing the computational complexity of the MIMO ML decoder. In *Proceedings of International Symposium on Information Theory and its Applications*, pages 34–38, Parma, Italy, October 2004.

[12] G. Hanrot, X. Pujol, D. Stehlé. Algorithms for the shortest and closest lattice vector problems. In *International Workshop on Coding and Cryptology*, 2011.

[13] S. Hanssian. Successs rates of integer parameter estimates in linear models. Master's thesis, McGill University, School of Computer Science, 2011.

[14] M. Jankiraman. *Space-Time Codes and MIMO Systems*. Artech House Publishers, 2004.

[15] W. Ku. Lattice basis reduction and integer least squares problems. Master's thesis, McGill University, School of Computer Science, 2011.

[16] A.K. Lenstra, J.H.W. Lenstra, L. Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[17] D. Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Transactions on Information Theory*, 47:1212–1215, 2001.

[18] B. Hassibi M. Stojnic, H. Vikalo. Speeding up the sphere decoder with h-infinity and sdp inspired lower bounds. *IEEE Transactions On Signam Processing*, 56:712–726, 2008.

[19] A.D. Murugan, H. El Gamal, M. O. Damen, G. Caire. A unified framework for tree search decoding: rediscovering the sequential decoder. *IEEE Transactions on Information Theory*, 52(3):933–953, 2006.

[20] M. Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *ACM SIGSAM Bulletin*, 15:37–44, February 1981.

[21] C.P. Schnorr, M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.

[22] K. Su, I. Wassell. A new ordering for efficient sphere decoding. In *IEEE International Conference on Communications*, volume 3, pages 1906–1910, 2005.

[23] Z. Zhou J. Wang Weiyu Xu, Youzheng Wang. A computationally efficient exact ml sphere decoder. In *Proceedings of IEEE Globecom 2004*, 2004.

[24] D. Wubben, R. Bohnke, J. Rinas, V. Kuhn, K.D. Kammeyer. Efficient algorithm for decoding layered space-time codes. *IEEE Electronics Letters*, 37(22):1348–1350, October 2001.

[25] D. Wubben, D. Seethaler, J. Jalden, G. Matz. Lattice reduction - a survey with applications in wireless communications. *IEEE Signal Processing Magazine*, May:70–92, 2011.

[26] X. Xie, X.-W. Chang, M. A. Borno. Partial lll reduction. In *IEEE Globecom*, 2011.

[27] G. Xu. *GPS: theory, algorithms, and applications*. Springer-Verlag, 2007.

[28] Z. Yan, Y. Dai. Memory-constrained ml-optimal tree search detection. In *In Proceedings of IEEE Information Sciences and Systems*, 2008.