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

ACKNOWLEDGEMENTS

ABSTRACT

In the worst case the integer least squares (ILS) problem is NP-Hard. Since its solution has many practical applications, there have been a number of algorithms proposed to solve it and some of its variations e.g., the box-constrained ILS problem (BILS). There are typically two stages to solving an ILS problem, the reduction and the search. Obviously we would like to solve instances of the ILS problem as quickly as possible, however most of the literature does not compare the run-time or FLOP counts of the algorithms; instead they use a more abstract metric (the number of nodes visited during the search). This metric does not always coincide with the algorithms run-time. This thesis will review some of the most effective reduction and search strategies for both the ILS and BILS problems. By comparing the run-time performance of some search algorithms, we are able to see the advantages of each, which allows us to propose a new, more efficient search strategy that is a combination of two others. It will also be proven that two very effective BILS reduction strategies are theoretically equivalent and a new BILS reduction that is equivalent to the others but more efficient will be proposed. Finally, it will be shown that the BILS reduction can be applied to the ILS problem as well to provide a significant performance improvement.

ABRÉGÉ

The text of the abstract in French begins here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

This thesis will deal with efficient algorithms for solving the integer least squares (ILS) problem. In some areas this problem is referred to as "Sphere Decoding", or the "Closest Vector Problem" (CVP). This problem has been well studied in mathematics for over 100 years. The ILS problem has been proven to be NP hard [?], 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 complexity 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 examined and then previous work which has dealt with solving these problems efficiently will be reviewed.

1.1 Notation

This section will introduce the notation which will be used throughout this thesis. Matrices will be denoted using 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 and the i^{th} column of this identity matrix can be written 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 will be written as $\rceil x \lfloor$, where x is the vector or scalar. Suppose we have some set of integers β, rounding to the nearest value in this set will be denoted as $\rceil x \lfloor_\beta$.

Some of the variables that will be discussed in this thesis will be normally distributed. To describe a scalar or vector whose elements are drawn from a normal distribution, we can write $v \sim N(0, \sigma^2 I)$. This would mean a vector v has its elements drawn from a normal distribution with mean 0 and standard deviation $sigma$. The first parameter (in this case 0) specifies the mean and the second gives the covariance matrix for the normal distribution in question.

1.2 Ordinary Real Least Squares Problem

Consider the following linear model,

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

Where $y \in \mathbb{R}^m$, $H \in \mathbb{R}^{m \times n}$ is called the "design matrix" and has full column rank, and

$$v \sim N(0, \sigma^2 I) \in \mathbb{R}^m$$

is a noise vector. We would like to find the estimate to the solution x which we will call x_{RLS} that minimizes the least squares residual,

$$\min_{x_{RLS}} \|Hx_{RLS} - y\|_2^2 . \tag{1.2}$$

Under the assumption that v is normally distributed, the solution to this problem, x_{RLS} has the property that it is the maximum likelihood estimator for 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 (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,

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

The solution to these normal equations will satisfy the following:

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

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

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

1.3 Integer Least Squares Problems

The ordinary integer least squares (OILS) problem is a modification of the LS problem where the solution vector $x \in \mathbb{Z}^n$ is an unknown integer vector as follows:

$$\min_{x_{ILS} \in \mathbb{Z}^n} \|y - Hx_{ILS}\|_2. \tag{1.4}$$

We no longer have a closed-form solution to find the optimal solution x_{ILS} in this case, in fact, the problem is provably NP-Hard [**?**].

Sometimes the solution vector x_{ILS} will be subject to constraints. One example of such constraints is given by the box-constrained integer least squares problem (BILS). Here we have the following constraint on the solution vector,

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

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

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

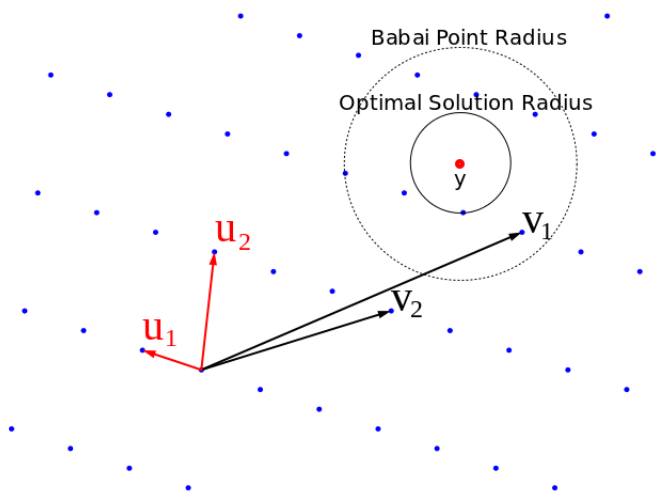

Figure 1–1: An example of a lattice with two different sets of basis vectors.

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

One of the well known approaches for solving an OILS or BILS problem consists of two phases, reduction and search. In the reduction phase, we transform the problem into an equivalent, but easier one. This typically involves manipulations on the design matrix H such as column permutations or integer gauss transformations to try and achieve certain properties. After reduction, we proceed to the search phase where we try to enumerate the possible solutions in an efficient manner.

In lattice theory, the matrix H is called the "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 thesis will not focus on the geometrical lattice interpretation of the problem; for more details on it, see [1].

1.4 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 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 the satellite and the receiver. The number of cycles will be real but we would like to estimate the integer part of the real number, we can form a linear model for this system and obtain an estimate of this integer by solving an OILS problem. For more information on this problem, see [**?**].

Some important applications such as MIMO wireless signal decoding depend on the solution of the BILS problem. MIMO stands for "multiple-input multiple-output" and it 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 (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 will depend 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 [**?**]. 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 [**?**].

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

1.5 Previous Work

Due to the important applications of the BILS and OILS problems, much work has gone into solving them efficiently. To solve ILS problems exactly there are three main families of algorithms. One family uses "Voronoi cells", we will not consider algorithms in this family because they are known to be relatively inefficient. Another family of algorithms can be referred to as "Continuous Relaxation Branch and Bound 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 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 known to be efficient. There have also been some algorithms proposed that yield a fast, approximate solution 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 [**?**]. For details on the CRBB methods see [**?**].

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 all possible solutions but one are eliminated. The order in which the elements of the solution vectors are enumerated plays a critical role in how long the search algorithm will take to run; the reasons for this will become clear later.

1.5.1 Search Strategies

In Chapter 2, it will be shown that the discrete search process is equivalent to a tree search problem. The tree in question will have a depth of n; each node on a path from root to leaf represents a valid element in the solution vector. Therefore the tree will have exponential width. To see an example of such a tree, refer to Figure 2–1. The most widely used algorithm for the OILS search process, the Schnorr Euchner (SE) enumeration [13], 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 [**?**]. 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 [12], [15], [8], [5] and [17]. 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. Compared to the depth first search where the memory requirements are linear in the problem size n and there is very little cost to visit a node in the tree. The advantage of the best first approach is that it is guaranteed to explore 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 [12]. 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 will be explored by various algorithms, this does not 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 [11], 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 [2]. 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.

1.5.2 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 the unconstrained OILS problems is the LLL reduction [10]. 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 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 these bounds see [**?**].

In [**?**], 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 will be described in 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 [7] and the “Sorted QR Decomposition” or SQRD column reordering [16]. 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 [14] in 2005, and then [4] in 2008. Numerical results show that these algorithms can offer great improvements over the previous reductions that use only the information contained in H.

1.6 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 [10] 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 will be 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 1.5. Two more recent developments, [4] and [14], 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 how 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 [14] 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 [14] and offer some improvements to the original. This work was accepted to IEEE Globecom 2011 [**?**].

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 may simply change 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 re-order 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 [12] 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.

1.7 Outline

The rest of the thesis will be organized as follows;

In Chapter 2, the Schnorr-Euchner (SE) enumeration algorithm [13] 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 BILS and OILS problems. Then, strategies for reducing BILS and OILS 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 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, reduction and search. The purpose of the reduction step is to transform the problem to an easier but equivalent one so the search will proceed more quickly, therefore the reduction is performed before the search. In this thesis the details on the search process will be 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 [13]. The SE search strategy is an improvement to a prior strategy called 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 will be 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

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

which implies the following set of inequalities:

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

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 we could start by choosing c_n since it does not depend on 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 depends on the initial β. Choose $x_n = \lfloor c_n \rceil$, the nearest integer to c_n. If the inequality (2.5) with $k = n$ is not satisfied, it will not be satisfied for any integer, this

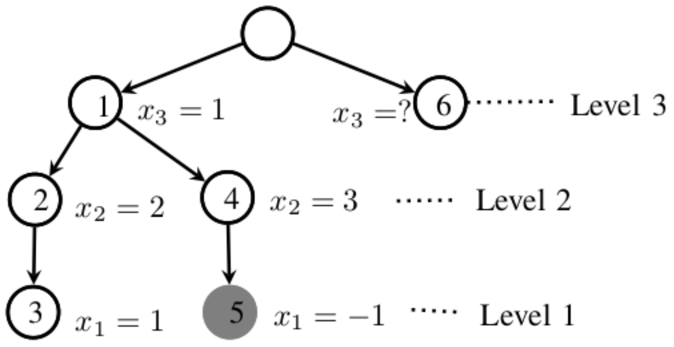

Figure 2–1: An example of the search process with solution $x = [-1, 3, 1]^T$.

means β was chosen to be too small, 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 (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 will 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

$$\beta = \|\bar{y} - R\hat{x}\|_2^2$$

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 will be 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 process 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 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 will have a weight which is given by $r_{ii}^2(z_i - c_i)^2$. Notice that if we take the sum of these weights from $m, \dots, i$ we simply get the partial residual for fixing $x_{i:n}$, which is defined as

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

This implies that each leaf in the tree represents an integer vector $x \in \mathbb{Z}^n$ and its weight is the residual

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

This means that the OILS problem is equivalent to finding the lowest-weight 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 cost 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 weight or 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 explore 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 [4].

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 will be the Babai 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

$$\|R\lceil x_{RLS}\rfloor - \bar{y}\|_2^2 .$$

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 will never be slower when using this initial β, often it will just end up being equivalent to using infinity since the Babai 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 will focus 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 will be explained. Next some BILS reduction strategies will be described, we will show that two of the more efficient 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 results presented.

Consider the OILS problem (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 (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, (1.4) becomes:

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

$$= \min_{\hat{z} \in \mathbb{Z}^n} \|Hz - y\|_2 \tag{3.2}$$

$$\tag{3.3}$$

When we solve this new OILS problem, we will obtain some solution z. If Z is a known unimodular matrix, we can solve the system $Zx = z$ to find $x = Z^{-1}z$, the OILS solution to the original problem. Note that if Z were not unimodular, we would have no guarantee that $Z^{-1}z$ 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 re-order 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, (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 re-order the elements of the solution x, so we must re-order 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 $\hat{x}_{k-1}$ (note that $\hat{x}$ denotes Zx). We know that

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

but to determine such a bound for $\hat{x}_{k-1}$, we would need to know the entire vector $\hat{x}$ in order to compute $Z^{-1}_{k-1,:}\hat{x}$. 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 $\hat{x}$, find a potential solution, compute $x = Z^{-1}\hat{x}$, 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 [7] mentioned in 1.5 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 [**?**].

In [4], the "Sorted QR Decomposition" or SQRD column reordering strategy originally presented in [16] for the same purpose as V-BLAST, was proposed for this purpose. 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 [14], 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 [4] 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 [4] and it is easy for others to implement the algorithm. But our numerical tests and theoretical analysis indicated 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 becomes 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 $k = n, n-1, \dots, 1$.

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.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \\ & \times & & \times & \\ & \times & & & \end{bmatrix} \tag{3.4}$$

Equation 3.4 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 (3.5), denote such a Givens rotation as $G_{k,k+1}$:

$$\begin{bmatrix} I_{k-1} & & & \\ & c & -s & \\ & -s & c & \\ & & & I_{n-k-1} \end{bmatrix}, \quad c^2 + s^2 = 1 \tag{3.5}$$

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 will create subdiagonal elements in columns $i+1, \ldots, n$. Equation (3.6) shows the matrix after applying this first round of Givens rotatations.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \\ & & & \times & \end{bmatrix} \tag{3.6}$$

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

$$x_i^c = \arg\min_{x_i \in \mathcal{B}_i} |\hat{r}_{nn}(x_i - c_n)| = \lfloor c_n \rceil_{\mathcal{B}_i}, \tag{3.7}$$

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.,

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

Define

$$\text{dist}_i^c = |\hat{r}_{nn}(\bar{x}_i^c - c_n)|, \tag{3.8}$$

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

$$\min_{\tilde{x} \in \mathbb{Z}^{n-1}} \left\| \bar{y}_{1:n-1} - R_{1:n-1,1:n-1} \tilde{x} \right\|_2 , \tag{3.9}$$

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 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.

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

```
1: $p := 1 : n$
2: $p' := 1 : n$
3: 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 $\bar{y} := Q_1^T y$
4: for $k = n$ to 2 do
5:     $maxDist := -1$
6:     for $i = 1$ to $k$ do
7:         $\hat{y} := \bar{y}_{1:k}$
8:         $\hat{R} := R_{1:k,1:k}$
9:         swap columns $i$ and $k$ of $\hat{R}$, return it to upper triangular with Givens rotations,
           also apply the Givens rotations to $\hat{y}$
10:        $x_i^c := \lfloor \hat{y}_k / \hat{r}_{k,k} \rceil_{\mathcal{B}_i}$
11:        $\bar{x}_i^c := \lfloor \hat{y}_k / \hat{r}_{k,k} \rceil_{\mathcal{B}_i \setminus x_i^c}$
12:        $dist_i^c := |\hat{r}_{k,k} \bar{x}_i^c - \hat{y}_k|$
13:        if $dist_i^c > maxDist$ then
14:            $maxDist := dist_i^c$
15:            $j := i$
16:            $R' := \hat{R}$
17:            $y' := \hat{y}$
18:        end if
19:    end for
20:    $p_k := p'_j$
21:    Interchange the intervals $\mathcal{B}_k$ and $\mathcal{B}_j$
22:    Intechange entries $k$ and $j$ in $p'$
23:    $R_{1:k,1:k} := R'$
24:    $\bar{y}_{1:k} := y' - R'_{1:k,k} x_j^c$
25: end for
26: $p_1 := p'_1$
```

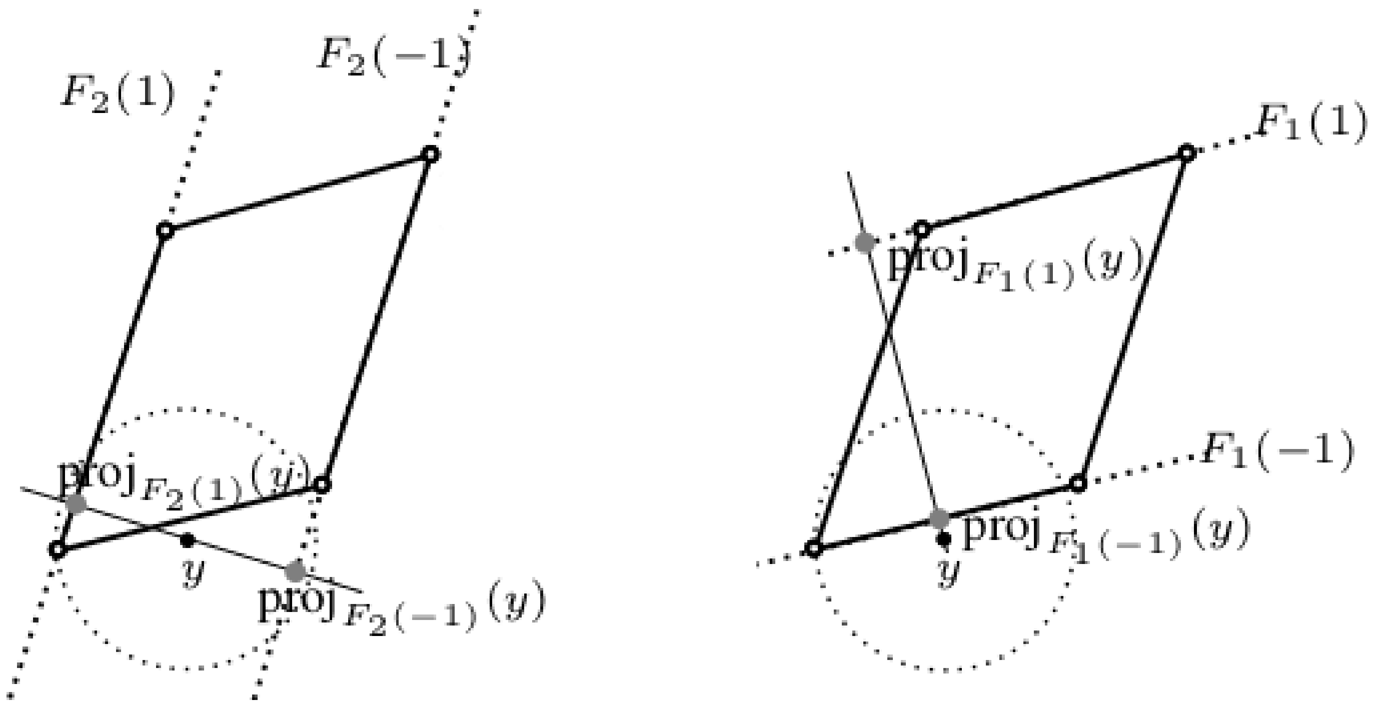
$F_2(1)$
$F_2(-1)$
$\mathrm{proj}_{F_2(1)}(y)$
y
$\mathrm{proj}_{F_2(-1)}(y)$
$F_1(1)$
$\mathrm{proj}_{F_1(1)}(y)$
$F_1(-1)$
$\mathrm{proj}_{F_1(-1)}(y)$
y

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

3.1.3 SW Original Algorithm

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

Fig. 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, [14] 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 $\mathrm{proj}_t(s)$, 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 [14], the points labeled $\text{proj}_{F_2(1)}(y)$ and $\text{proj}_{F_2(-1)}(y)$ in Fig. 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 Fig. 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 Fig. 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 Fig. 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 [14].

The following will give an overview of the SW algorithm as given in [14] 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

$$x_i^s = \arg\min_{\alpha \in \mathcal{B}_i} \operatorname{dist}(y, F_i(\alpha)) = \arg\min_{\alpha \in \mathcal{B}_i} |y^T g_i - \alpha| = \lfloor y^T g_i \rceil_{\mathcal{B}_i}, \tag{3.10}$$

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.,

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

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.

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

```
1:  $p := 1 : n$
2:  $p' := \{1, 2, \ldots, n\}$
3:  $G := H^{-T}$
4:  for $k = n$ to 2 do
5:      $maxDist := -1$
6:      for $i \in p'$ do
7:          $x_i^s := \lfloor y^T g_i \rceil_{\mathcal{B}_i}$
8:          $\bar{x}_i^s := \lfloor y^T g_i \rceil_{\mathcal{B}_i \backslash x_i^s}$
9:          $\text{dist}_i^s := \text{dist}(y, F_i(\bar{x}_i^s))$
10:         if $dist_i^s > maxDist$ then
11:             $maxDist := dist_i^s$
12:             $j := i$
13:         end if
14:     end for
15:     $p_k := j$
16:     $p' := p' \backslash j$
17:     $y := \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s$
18:     for $i \in p'$ do
19:         $g_i := \text{proj}_{F_j(0)}(g_i)$
20:     end for
21: end for
22: $p_1 := p'$
```

Su and Wassell did not say how to implement the algorithm and did not give a complexity analysis. The parts of the cost we must consider for implementation occur in lines 9 and 19. Note that

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

and

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

where

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

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

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

and

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

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

$$\mathrm{dist}_i^s = \|g_i g_i^\dagger (y - h_i\bar{x}_i^s)\|_2 = |y^T g_i - \bar{x}_i^s| / \|g_i\|_2. \tag{3.11}$$

Note that $y^T g_i$ and $\bar{x}_i^s$ have been computed in lines 7 and 8, respectively. So the main 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,

$$\begin{aligned} g_i^{\text{new}} &\equiv \text{proj}_{F_j(0)}(g_i) \\ &= (I - \text{proj}_{g_j}) g_i = g_i - g_j (g_j^T g_i / \|g_j\|_2^2). \end{aligned} \tag{3.12}$$

Using the last expression for computation needs only $4n$ flops (note that $\|g_j\|_2$ has been computed before, see (3.11)). 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 [6].

In line 17 of Algorithm 2,

$$\begin{aligned} y^{\text{new}} &\equiv \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s = (y - g_j g_j^{\dagger}(y - h_j x_j^s)) - h_j x_j^s \\ &= (I - \text{proj}_{g_j})(y - h_j x_j^s). \end{aligned} \tag{3.13}$$

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 (3.13). 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 (3.12) and (3.13)

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

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

$$\min_{\check{x} \in \mathbb{Z}^{n-1}} \|y - H_{:,1:n-1}\check{x}\|_2, \tag{3.14}$$

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 [14], 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 $H^TH = R^TR$ and solve $R^TRG^T = H^T$ for G by using the triangular structure of R. The total 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 a 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 $x_i^s = x_i^c$, $\bar{x}_i^s = \bar{x}_i^c$, $\text{dist}_i^s = \text{dist}_i^c$ 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 (3.7) and (3.10),

$$x_i^c = \lfloor (H^\dagger y)_i \rceil_{\mathcal{B}_i} = \lfloor e_i^T H^\dagger y \rceil_{\mathcal{B}_i} = \lfloor g_i^T y \rceil_{\mathcal{B}_i} = x_i^s. \tag{3.15}$$

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 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 (3.11) and (3.8)

$$\begin{aligned} \text{dist}_i^s = \frac{|y^T g_i - \bar{x}_i^s|}{\|g_i\|_2} &= |\hat{r}_{nn}||y^T Q_1 V \hat{R}^{-T} e_n - \bar{x}_i^s| \\ &= |\hat{r}_{nn}||\hat{y}_n/\hat{r}_{nn} - \bar{x}_i^s| = |\hat{r}_{nn}(c_n - \bar{x}_i^c)| = \text{dist}_i^c. \end{aligned} \tag{3.16}$$

Now we consider the subproblem (3.9) in CH and the subproblem (3.14) in SW. We can easily show that $R_{1:n-1,1:n-1}$ in (3.9) is the R-factor of the QR decomposition of $H_{:,1:n-1}P$, where $H_{:,1:n-1}$ is the matrix given in (3.14) and P is a permutation matrix such that $\check{x} = P\tilde{x}$, and that $\bar{y}_{1:n-1}$ in (3.9) is the multiplication of the transpose of the Q_1-factor of the QR decomposition of $H_{:,1:n-1}P$ and y in (3.14). Thus the two subproblems are equivalent. To see proof of this, simply observe the following:

$$HP = QRP \tag{3.17}$$

$$= QV^T VRP \tag{3.18}$$

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 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 (3.15), we compute for $i = 1, \ldots, n$

$$\begin{aligned} x_i &= \lfloor \bar{y}^T G_{:,i} \rceil_{\mathcal{B}_i} = \lfloor \bar{y}_{i:n}^T G_{i:n,i} \rceil_{\mathcal{B}_i}, \ \bar{x}_i = \lfloor \bar{y}_{i:n}^T G_{i:n,i} \rceil_{\mathcal{B}_i \backslash x_i}, \\ \mathrm{dist}_i &= |\bar{y}_{i:n}^T G_{i:n,i} - \bar{x}_i| / \|G_{i:n,i}\|_2. \end{aligned}$$

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 will permute 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 LLL reduction [10] 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 thesis, we will look at the

Algorithm 3 New algorithm

```
1: 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)
2: $G := R^{-T}$                                                   ($\frac{n^3}{3}$ flops)
3: $p := 1 : n$
4: $p' := 1 : n$
5: for $k = n$ to 2 do
6:    $maxDist := -1$
7:    for $i = 1$ to $k$ do
8:       $\alpha = y_{i:k}^T G_{i:k,i}$
9:       $x_i := \lfloor \alpha \rceil_{\mathcal{B}_i}$                              ($2(k-i)$ flops)
10:      $\bar{x}_i := \lfloor \alpha \rceil_{\mathcal{B}_i \setminus x_i}$
11:      $\text{dist}_i = |\alpha - \bar{x}_i| / \|G_{i:k,i}\|_2$                ($2(k-i)$ flops)
12:      if $dist_i > maxDist$ then
13:         $maxDist := dist_i$
14:         $j := i$
15:      end if
16:   end for
17:   $p_k := p'_j$
18:   Interchange the intervals $\mathcal{B}_k$ and $\mathcal{B}_j$
19:   Interchange entries $k$ and $j$ in $p'$
20:   Set $\bar{y} := \bar{y}_{1:k-1} - R_{1:k-1,j} x_j$
21:   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)
22: end for
23: $p_1 = p'_1$
```

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 QRZ decomposition, the matrix R is LLL reduced i.e.,

$$|r_{k-1,j}| \leq \frac{1}{2} |r_{k-1,k-1}| \tag{3.19}$$

$$\sigma r_{k-1,k-1}^2 \leq r_{k-1,k}^2 + r_{k,k}^2 \tag{3.20}$$

$$j = k : n, k = 2 : n \tag{3.21}$$

From (3.19) and (3.20) we can easily obtain the following inequality:

$$|r_{k-1,k-1}| \leq \frac{2}{\sqrt{4\delta - 1}} |r_{k,k}| . \tag{3.22}$$

Looking at (3.22), we can obtain some sense of why using a LLL reduced matrix R in the search process should yield a performance improvement. Usually in practice, we use $\delta = 1$, this is the maximum value for sigma and forces the most strict ordering of the diagonal. 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, $|r_{11}| <, \ldots, < |r_{nn}|$. This allows us to prune the search tree at higher levels without wasting computational effort on suboptimal solutions. The equation (3.22) gives us a guarantee about the relative sizes of the diagonal elements. In practice, the diagonal will usually end up being mostly increasing.

3.2.1 Computing the LLL Reduction

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

Integer Gauss Transformations

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

$$Z_{ij} = I - \mu e_i e_j^T, \quad \mu \in \mathbb{Z}. \tag{3.23}$$

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 will give:

$$\bar{R} = RZ_{ij} = R - \mu R e_i e_j^T. \tag{3.24}$$

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:

$$\bar{r}_{kj} = r_{kj} - \mu r_{ki}, \quad k = 1, \dots, i. \tag{3.25}$$

If we take $\mu = \lfloor \frac{r_{ij}}{r_{ii}} \rceil$, it should be clear that $|\bar{r}_{ij}| \le \frac{1}{2}|r_{ii}|$, so given a particular column in an upper triangular matrix R, we should be able to use m IGTs to satisfy the first condition given in equation (3.19).

Permutations

After doing IGTs, there is no guarantee that the second LLL condition, (3.20) 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 will 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 second condition, (3.20) 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.

LLL Reduction

By putting subsections 3.2.1 and 3.2.1 together, we can devise an algorithm to satisfy both of the LLL conditions (3.19) and (3.20). We will start by letting $H = QR$ 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 first LLL condition (3.19), $|r_{ik}| < \frac{1}{2}|r_{ii}|, \quad i = k-1 : -1 : 1$. If the second inequality in 3.20 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 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. It is also know that this algorithm will terminate in a finite number of steps, for more detail and an alternate explanation of the process, see the original paper [10].

3.2.2 New OILS Reduction

In subsection 3.1.3, the motivation for the SW algorithm was given. While the SW algorithm does make use of the box constraint, the original motivation for their algorithm applies to the OILS problem as well. Similarly, the motivation for the CH algorithm also

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

```
1: Compute the QR decomposition of $H$: $\begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} H = \begin{bmatrix} R \\ 0 \end{bmatrix}$
2: $Z := n \times n$ Identity matrix
3: $k := n$
4: while $k \geq 2$ do
5:     if $k > n$ then
6:         $k := n$
7:     end if
8:     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
9:     if $r_{k,k}^2 + r_{k-1,k}^2 < r_{k-1,k-1}^2$ then
10:        $P :=$ Permutation matrix to swap column $k$ and $k-1$
11:        $R := RP$
12:        $Z := ZP$
13:        $k := k + 1$
14:    else
15:        $k := k - 1$
16:    end if
17: end while
```

applies to the unconstrained problem; there is nothing specific about either motivation that only applies to box constrained problems.

Applying the SW or CH algorithm directly to the matrix H however may yield results that are much worse than those given by LLL on average. The reason for this is that the SW algorithm 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 (shorter and more orthogonal) and a reasonably good ordering for those 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 new reduction strategy given in subsection 3.1.6 to re-order the basis vectors. We will call this reduction strategy "LLL+PERMU" from here on for convenience. Suppose we have $H = QRZ$. Unfortunately after applying the permutations, the LLL conditions in the new upper triangular matrix VRP may no longer be satisfied. Whether the search will be faster for VRP or R seems to be hard to predict consistently. It 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 permutations, 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, therefore further investigation into this reduction strategy is needed.

Recall the Babai point from Chapter 2, denote the Babai point by the vector z_0. This is the first point found during the SE search process. The residual

$$\|Rz_0 - \bar{y}\|_2^2$$

defines the initial radius of the search process. The number of nodes that the SE search will visit 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 will be 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 a different Babai point after applying permutations to the design matrix H (or equivalently R). Since we know that the search time is related to the initial

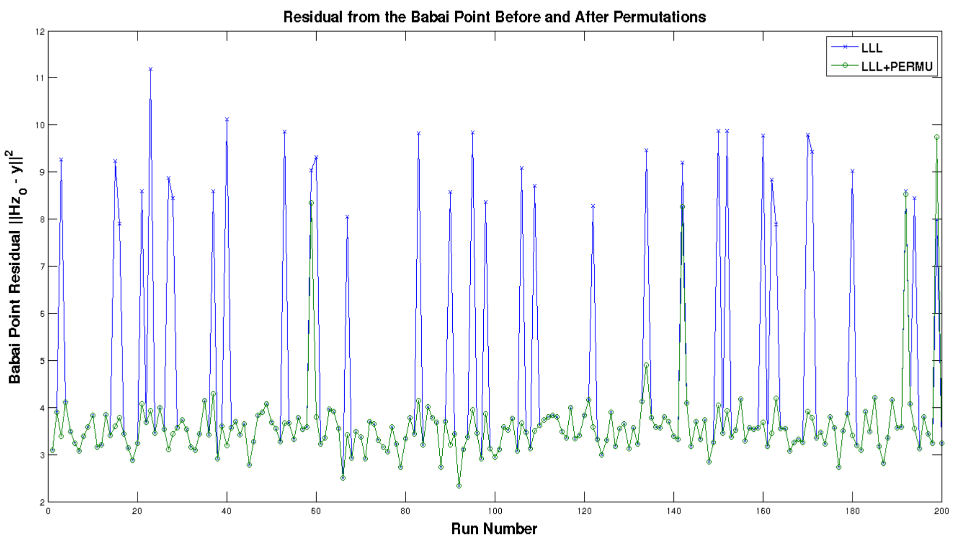

Figure 3–2: LLL Reduction vs LLL+PERMU. Residual from the Babai points.

radius which is defined by the Babai point, one way to estimate whether the search process will proceed faster before or after permutations is to compare the initial radius in both cases. Figure 3–2 shows such a comparison for 200 matrices $H \in \mathbb{R}^{50\times 50}$ generated randomly with their elements picked from a standard normal distribution. The elements in the vectors $x \in \mathbb{Z}^n$ were picked uniformly between $-10, \ldots, 10$ 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.

Looking at Figure 3–2, we can see that after permutations, the Babai point in this case is at least as good as the one before permutation. Often it is significantly better. For cases where the Babai point is better, we can choose to use the permuted R for the search process. Numerical results comparing this strategy to performing the search on a LLL reduced matrix can be found in section 3.2.3.

Unforunately, there are certain special types of ill conditioned problems where the above strategy does not work very well. Even though the new matrix R after permutations gives a better Babai point, the search process visits more 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.

Since we only apply orthogonal transformations to the matrix R and vector y with our permutation strategy, we can use the residual from the Babai point in the permuted problem directly in the problem before permutation. The equation

$$\|VRPz_0 - Vy\|_2^2 = \|RPz_0 - y\|,$$

where V is a product of Givens rotations and P is a permutation matrix, shows this. This means the β given by this strategy will never be too small, forcing us to restart the search process. Using this residual in the search process on the LLL reduced problem when it is smaller than the residual to the Babai point given by LLL reduction alone is guaranteed to result in a faster 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 permuted matrix R in the search, but as a general method to solve OILS problems, it is safer to use the LLL reduced R with the initial radius β given by the permutation strategy. Some comparisons can be found in section 3.2.3.

3.2.3 OILS Reduction Results

In this subsection, some results will be presented to compare the speed of the search process after applying the reduction strategies presented previously in this chapter. 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).

Figure 3–3 shows the three reduction strategies compared when run on normally distributed, random matrices H of sizes from 35×35 to 50×50. The horizontal axis displays varying values for σ which is the noise parameter. The vertical axis is the average time

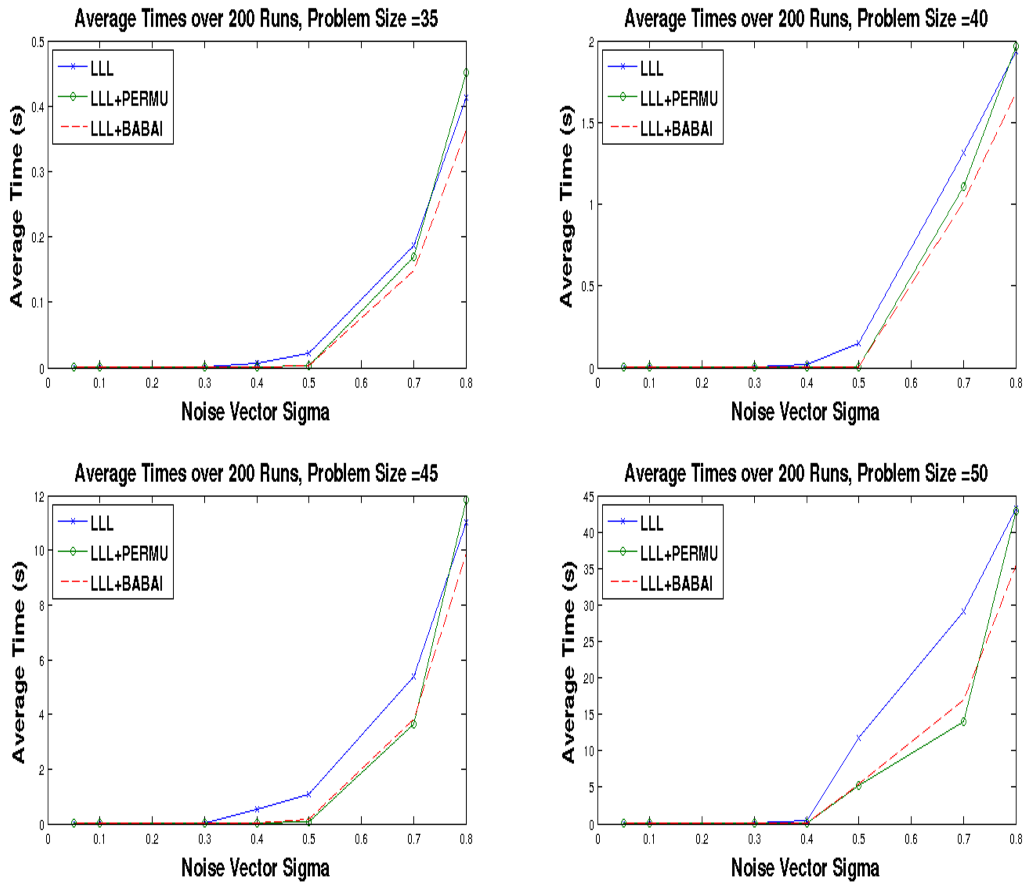

Figure 3–3: LLL Reduction vs LLL Reduction and Permutations vs LLL Reduction with Permutation Initial Radius.

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 starts 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, LLL+PERMU actually becomes worse than LLL, however LLL+BABAI maintains a significant advantage in all cases.

Tables 3–1 and 3–2 show how many times out of 200 runs that the Babai point was equal to the true solution x. If we take this as a percentage it is known as the "Success

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

	0.05	0.1	0.3	0.4	0.5	0.7	0.8
35×35	200	200	199	192	178	89	44
40×40	200	200	200	196	175	75	45
45×45	200	200	200	193	175	79	42
50×50	200	200	200	196	167	89	44

Table 3–2: Success Rate (out of 200) for LLL+PERMU on various problem sizes and levels of noise.

	0.05	0.1	0.3	0.4	0.5	0.7	0.8
35×35	200	200	200	200	198	132	76
40×40	200	200	200	200	198	120	77
45×45	200	200	200	200	194	141	86
50×50	200	200	200	199	196	137	94

Rate". This is an important number because many practical applications may decide not to solve the OILS problem completely and instead just use the Babai point as an estimate at the solution. 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.

It is not fully clear why LLL+PERMU fails to perform well in the search process when the noise is high. One reason is because the matrix R after the permutations is usually further from a LLL reduced matrix when the noise is high as opposed to when the noise is low. This has been confirmed through numerical experiments where when the second inequality in 3.20 was not satisfied, the difference $r^2_{k-1,k-1} - (r^2_{k,k} + r^2_{k-1,k})$ was added to a sum. For a LLL reduced matrix this sum taken from $1, \ldots, n$ should be 0, as the matrix becomes further from LLL reduced the sum will increase. A rough explanation as to why this happens is as follows; when the noise is very low the permutation algorithm finds the column permutations that maximize the diagonal entries in the order from $n, \ldots, 1$. As the noise rises, this property is lost. To see this consider the motivation for the CH algorithm in 3.1.2 and consider what happens when $y = Hx$. One goal of the LLL reduction is also to maximize the diagonal entries in order from $n, \ldots, 1$ in a way, it however also considers

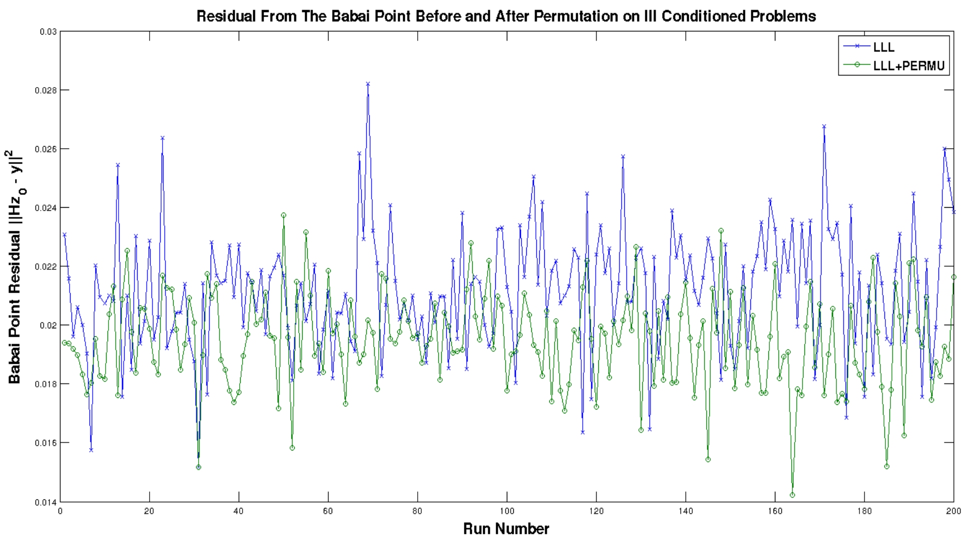

Figure 3–4: LLL Reduction vs LLL + PERMU. Residual from the Babai points on ill conditioned problems.

the off diagonal and previous diagonal entry. When the noise is lower, the permutation algorithm is likely to do fewer permutations since the ordering is more likely to already be somewhat satisfied since we started with a LLL reduced matrix. Therefore the matrix is likely to remain closer to a LLL reduced matrix.

The next set of results demonstrates a case where the permutation strategy should not be used. In figures 3–4 and 3–5 the matrix H is generated by first generating two random, normally distributed matrices A and B. Then the QR decomposition of each matrix is taken, so we will have $A = Q_A R_A$ and $B = Q_B R_B$. Next we generate a diagonal matrix D, 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; we will construct a matrix with condition number 10^4 by forming the product of the SVD, $H = Q_A D Q_B$. We generate x and y in the same way as the previous results.

Figure 3–4 shows the residual for the Babai point both before and after permutation where $H \in \mathbb{R}^{40\times 40}$ and $\sigma = 0.5$. Comparing to Figure 3–2, we notice that it is much different. First, the residual both before and after permutation is much smaller in magnitude. The residual after permutation is slightly better than before on average, but the difference

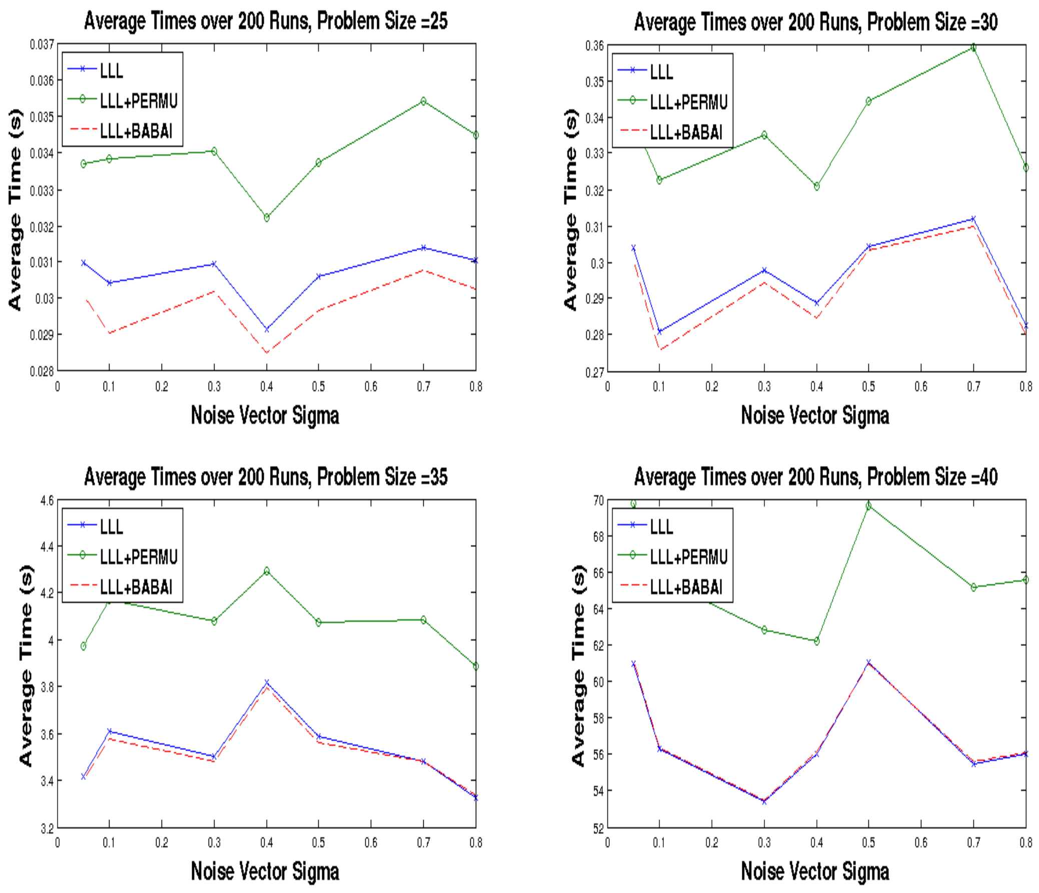

Figure 3–5: LLL Reduction vs LLL +PERMU vs LLL + BABAI on ill conditioned problems.

is nearly insignificant. With the smaller residual one may expect that the Babai point in this case is more likely to be equal to the true solution x. This however is not the case at all. Out of the 200 runs in Figure 3–4, not once was the Babai point equal to the true solution; compared to the corresponding example in Figure 3–2 where $167/200$ times the Babai point was the correct solution before permutations were performed.

Knowing that the Babai point does not seem to be significantly better when the matrix H is generated in this way, we can predict that the run time performance of the search process will probably be worse in the case where we use LLL+PERMU and about the same for LLL+BABAI. Figure 3–5 confirms this.

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 height n with potentially exponential width. The difference between this tree search and more general tree search problems is that we can easily visit the children of a given node in increasing order of cost (partial residual); we can compute the next child to visit in constant time.

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, we may call this a partial solution. Then the partial residual given by this partial solution is equivalent to its cost in the search tree and can be defined as

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

An optimal algorithm for an ILS problem visits only nodes in the search tree with partial residuals that are less than the optimal ILS solutions residual. Therefore all nodes explored by the optimal search process will satisfy the following:

$$\left\| R_{i:n,i:n} x^p_{i:n} - \bar{y}_{i:n} \right\|_2^2 \leq \left\| R x_{ILS} - \bar{y} \right\|_2^2 , \tag{4.1}$$

where x_{ILS} here denotes the optimal ILS solution, which depending on the noise may not be the same as the original integer vector.

Assuming no other special knowledge of the search problem, any other search algorithm must at least explore this set of nodes to guarantee there is no other leaf in the tree with a lower residual than the one found. There are in fact ways to reduce the number of nodes visited beyond this point, one technique involves using lower bounds on the optimal solutions residual. Unfortunately after numerical testing it was found that these methods 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, [8] and [15] 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 expanding 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. 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 nodes. For these reasons it is not always practical to use the BFS strategy, one problem is that often hardware applications have limited memory, another is that when the number of nodes explored becomes very large, the overhead of finding the one with minimum cost becomes significant and this must be done at each iteration.

Unfortunately, some of the literature such as [5] does not properly consider the overhead of finding the minimum cost node when comparing search algorithms, instead only considering the goal of limiting the memory usage of BFS while still exploring as few nodes as possible. Comparing the number of nodes explored 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

A quick review from Chapter 2; we will explore the search tree which has depth n and each edge has a cost to traverse. Define a given nodes cost as the sum of the costs of all the 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 for fixing $x_{i:n}$ to a set of values, e.g.

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

We would like to find the leaf with minimal cost. For the BFS, we will define a nodes "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 next will be $\lfloor c_k \rceil \pm 1$, 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.

As mentioned above, for an ILS application, we can quickly find a given nodes "next best child", it is the node corresponding 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 $x_k = \lfloor c_k \rceil$, where c_k comes from (2.3).

In the first step of the BFS, we initialize an empty priority queue, pq 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 will correspond to $x_n = \lfloor \bar{y}_n / R_{n,n} \rceil$ and the cost to visit this child will be equal to the partial residual $(R_{n,n}x_n - \bar{y}_n)^2$, we will 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 pq, elements in this priority queue will always be sorted by their "next best child cost".

In the next step, we will visit the first child of the root, x_{n-1}. 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 will be the partial residual:

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

We then insert x_n into the priority queue with its "next child" and "next child cost". Notice if we expand the cost as follows, $(R_{n,n}x_n - \bar{y}_n)^2 + (R_{n-1,n}x_n + R_{n-1,n-1}x_{n-1} - \bar{y}_{n-1})^2$ that 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 i^{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 roots new "next best child", $x_n = \lfloor c_n \rceil^{+}_{-}1$ 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 in the queue.

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 haven't yet visited which has a smaller partial residual than the current node at the top of the priority queue. This means the node in questions parent 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 explored only and all of those nodes that have a partial residual within the radius of the optimal hyper-sphere.

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 1.5.

4.2 Controlling BFS Memory Usage

In 4.1, notice that in each step a new node is added to the priority queue, but nodes are never removed (yes, 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 will be 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 over looked is, as the priority queue grows, it costs more and more to maintain it at each step; the cost for each operation is $\theta(log(N))$, where N is the number of nodes visited so far. Depending on the implementation of the priority queue, there could also be some significant constants involved with this cost meaning that for even small N we are paying a significant overhead to maintain the priority queue. Also consider that the cost to visit a node in an efficient implementation of the SE algorithm is only about $2k$, where k is the level of the node in the tree.

With the rough cost analysis in the previous paragraph, it should be obvious that just because the BFS will explore fewer nodes than SE, does not mean it will necessarily be faster. In both [17] and [5] 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 [5]. 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. It is not clear why they included it in their paper and it would just take longer to explain.

The idea 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 will take $|S|$ 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 BFS. 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 explore 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 nodes levels 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 tree one place to the right in memory, again this could cost 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 nodes 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, but at the same time also try 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 in [5] it is quite simple and 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. $x_n, x_{n-1}, \ldots$) 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 cutting the search tree at some level α which will be supplied by the user. We begin with a BFS, when the BFS reaches a node at level $k = \alpha$, it uses a SE search to find the optimal solution in that nodes subtree. Suppose we are at some node in the BFS which is given by fixing $x_{k+1:n}$ to some particular values (so we are at level $k = \alpha$), then we will solve the subproblem given by the following using the SE search:

$$\hat{y} = y_{1:k} - R_{1:k,k+1:n} x_{k+1:n} \tag{4.2}$$

$$\min_{x_{1:k} \in \mathbb{Z}^k} \|\hat{y} - R_{1:k,1:k} x_{1:k}\|_2 \tag{4.3}$$

By solving this subproblem, we can find the optimal solution assuming $x_{k+1:n}$ are fixed. With the solution calculated by the SE algorithm, we can continue the BFS until we reach another node at level α. Next time we visit a node at level α, we again perform an SE search to find the optimal solution in this nodes subtree, however this time we may use a sphere constraint to terminate the SE search early; if the previous SE search found a leaf with some residual r, then we can terminate this SE search when there are no nodes with residual less than r remaining to visit, we need not find a leaf. 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 sphere radius (given by the residual from the best leaf visited), 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 have no sphere constraint to use as a stopping condition and must solve the subproblem completely. The best case occurs when the first subproblem we solve contains the ILS solution, in this case we get the smallest possible sphere radius to use in subsequent SE searches as a stopping criteria, and will therefore prune more nodes. 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, 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 is when the Babai point given by that subproblem has a large residual. Eventually we will 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 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.

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.

4.4 Search Strategy Results

In this section, 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. This section will attempt to address this question through some numerical experiments. 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 results in this section, the matrices will simply be generated randomly with elements drawn from a standard normal distribution. The vectors x will have elements uniformly distributed between $-10, \dots, 10$ and the vectors y are generated as $y = Hx + v$ where v is normally distributed with some standard deviation σ.

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 was has $\sigma = 0.8$, which is quite high. The x-axis which varies from $1, \dots, 50$ 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 $\alpha = x = 40$, 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 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 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 3–4 and 3–5. The problem size here is 35×35 and the noise vector was generated with $\sigma = 0.3$. As with our new reduction strategy, this new search is ineffective on these types of matrices. The best results are

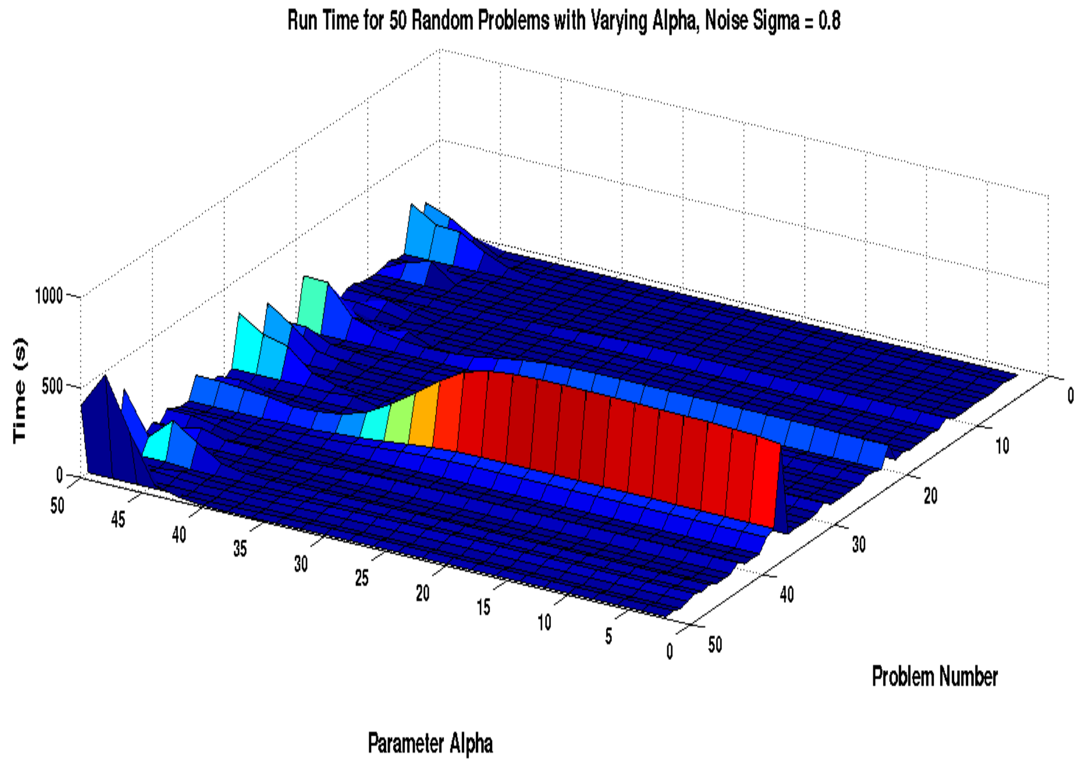

Figure 4–1: Run time results for the combined search process with varying parameter α on random problems.

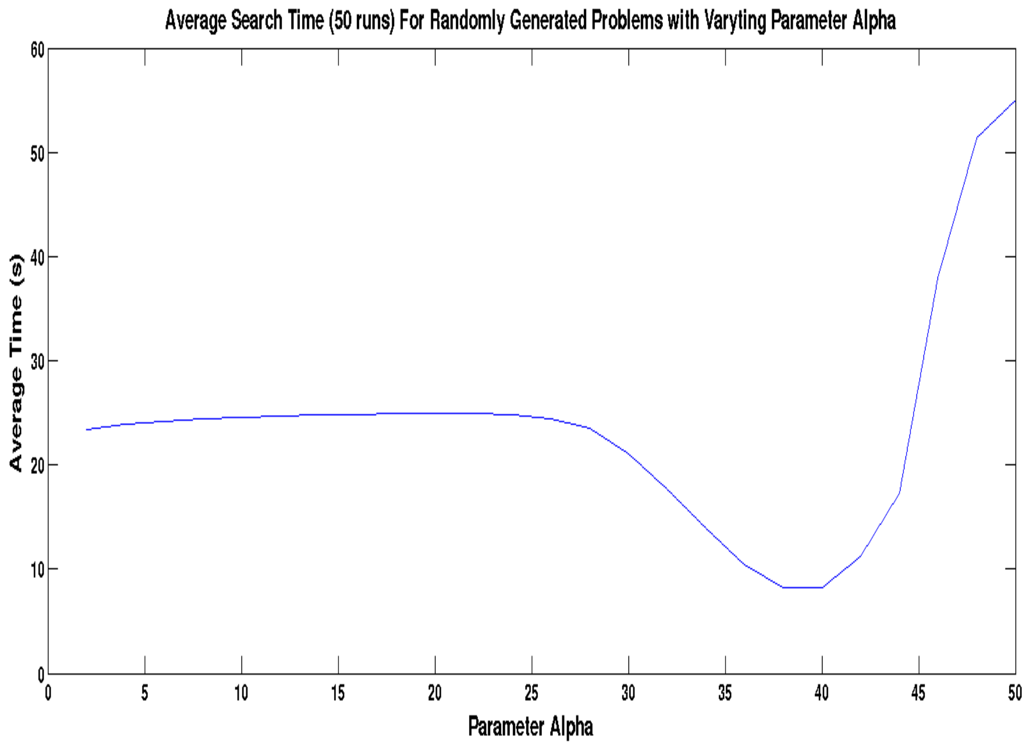

Figure 4–2: Average run time results for the combined search process with varying parameter α on random problems.

obtained when $\alpha = n$ which means we start the SE search process immediately, this is obviously equivalent to an SE search. Figures 4–3 and 4–4 both confirm this. 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 results where the new combined search achieved significantly better results, this ratio was 0.44 on average, for the new set of ill-conditioned results, it is 0.8 on average. This means that there is not as much difference between the best first search and SE search as far as the number of nodes visited is concerned. The fact that the BFS is exploring 20% fewer nodes on average is not enough to offset the extra time spent on each node. This is a property of these problems, it means that the Babai point is closer to the ILS solution. In the case where the ILS solution and the Babai point are the same, we get the same number of nodes visited by the BFS and SE search.

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. 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.

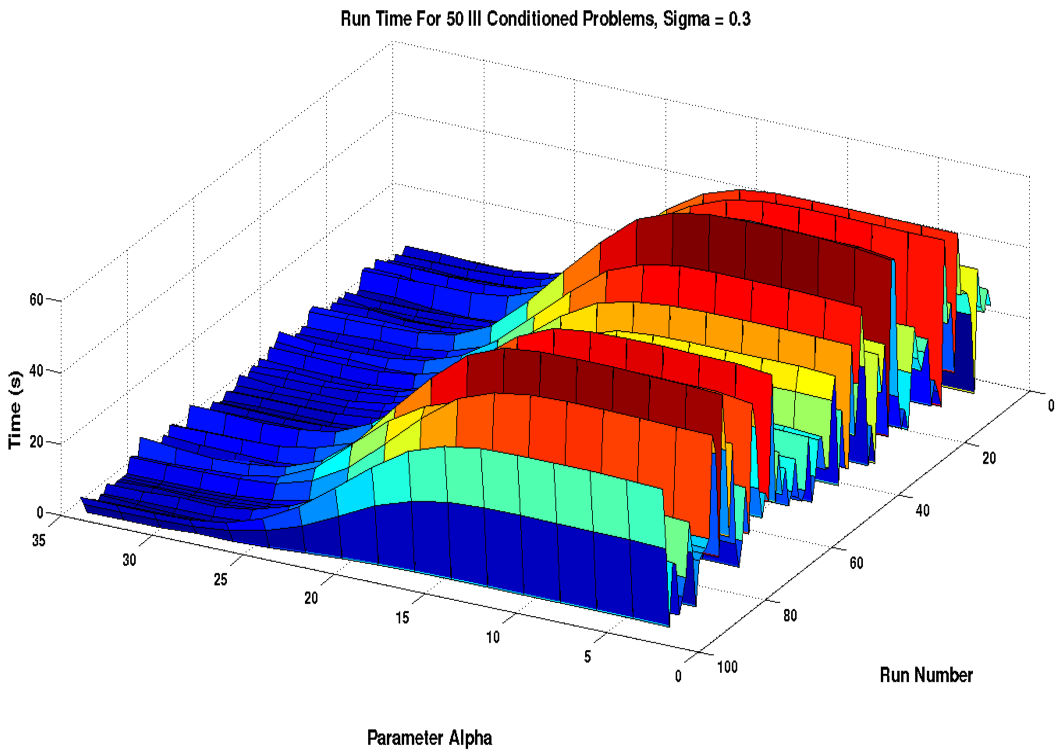

Figure 4–3: Run time results for the combined search process on ill conditioned problems with varying parameter α.

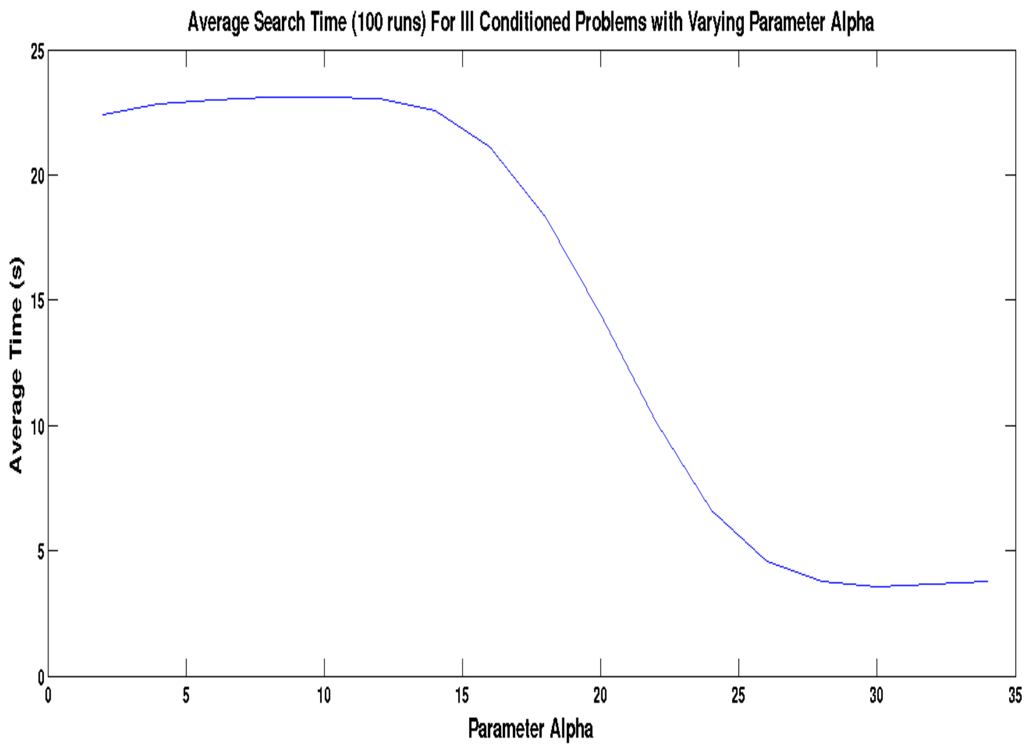

Figure 4–4: Average run time results for the combined search process with varying parameter α on ill conditioned problems.

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 for the BILS problem are in fact 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 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 point after the permutations having both a lower residual and higher success rate (defined as the probability that the Babai point is equal to the true solution x). Since the Babai point has a tendency to have lower residual and higher success rate after applying these permutations, we can conclude that for both BILS and OILS problems, applications which use the Babai 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 permutation reduction algorithm is still needed. It is not quite clear exactly when we should perform the search on the permuted matrix R and when we should search the LLL reduced R. It is also possible that there exists another reduction strategy that can combine the properties of the permutations and LLL reduction. This was investigated but nothing significant was found. We know the optimal ordering of the basis vectors can depend on y, it is also possible that the optimal basis vectors themselves could depend on y as well, the relationship however is unclear. It would also be interesting if we could use some easy to calculate properties of the matrix H, vector y and noise vector standard deviation σ to decide whether we should use the permutations or not before we even calculate them. Such a general result would be very useful in determining what types of practical applications this new reduction applies to.

In Chapter 4 the "Best First Search" approach was described. The BFS has the property that it will visit 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.

One of the previous attempts, [5] involved limiting the amount of memory used by the BFS using a user supplied parameter, unfortunately it is difficult to implement efficiently in some cases 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.

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 problem type, size and level of noise.

References

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

[2] Lutz Lampe Andreas Schenk, Robert Fischer. A stopping radius for the sphere decoder and its application to msdd of dpsk. *IEEE Communications Letters*, 13:465–467, 2009.

[3] Mazen Al Borno. Reduction in solving some integer least squares problems. Master's thesis, McGill University, 2010.

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

[5] Wolfgang Fichtner Christoph Studer, Andreas Burg. A unification of ml-optimal tree-search decoders. In *In Proceedings of IEEE Signals, Systems and Computers*, 2007.

[6] Randall 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.

[7] G. J. Foscini, G. D. Golden, R. A. Valenzuela, and 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.

[8] T. Fukatani, R. Matsumoto, and 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.

[9] Babak Hassibi and Haris Vikalo. On the sphere-decoding algorithm i. expected complexity. In *IEEE Transactions on Signal Processing*, volume 53, August 2005.

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

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

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

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

[14] Karen Su and Ian J. Wassell. A new ordering for efficient sphere decoding. In *IEEE International Conference on Communications*, volume 3, pages 1906–1910, 2005.

[15] Zucheng Zhou Jing Wang Weiyu Xu, Youzheng Wang. A computationally efficient exact ml sphere decoder. In *Proceedings of IEEE Globecom 2004*, 2004.

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

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