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 explored 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. We also prove that two very effective BILS reduction strategies are theoretically equivalent and propose a new BILS reduction that is equivalent to the others but more efficient.

ABRÉGÉ

The text of the abstract in French begins here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 Least Squares Problem

Consider the following linear model for some observation vector y,

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

Where $y \in \mathbb{R}^m$, $H \in \mathbb{R}^{m \times n}$ is called the "design matrix" and has full column rank, and $v \in \mathbb{R}^m$ is a noise vector which we assume is normally distributed with mean 0 and covariance matrix $\sigma^2 I$. We would like to find the unique solution $x \in \mathbb{R}^m$ which minimizes the least squares residual,

$$\|Hx - y\|_2^2 . \tag{1.2}$$

This is called the least squares (LS) problem. 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,

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

$$x = (H^T H)^{-1} H^T y. \tag{1.4}$$

The solution of the least squares problem has numerous applications in many fields of science and engineering.

1.2 Integer Least Squares Problems

The integer least squares (ILS) problem is a modification of the LS problem where the solution vector $x \in \mathbb{Z}^m$ is an unknown integer vector. We no longer have a closed-form solution for x in this case, in fact, the problem is provably NP-Hard in the worst case. The ILS problem can be expressed as:

$$\min_{x \in \mathbb{Z}} \|y - Hx\|_2. \tag{1.5}$$

A modification to the ILS problem is the box-constrained integer least squares problem (BILS). Here we have the following constraint on the solution vector,

$$x \in \mathcal{B} \tag{1.6}$$

$$\mathcal{B} = \mathcal{B}_1 \times \cdots \times \mathcal{B}_n \tag{1.7}$$

$$\mathcal{B}_i = \{x_i \in \mathbb{Z} : l_i \leq x_i \leq u_i, l_i \in \mathbb{Z}, u_i \in \mathbb{Z}\}. \tag{1.8}$$

Even though the problem is NP-Hard, we still have some hope to get solutions quickly. In [7] the authors prove that under reasonable assumptions on the variance in the noise, the ILS problem will have an expected polynomial complexity using standard search algorithms.

The usual approach to solving an ILS 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 A 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.

1.3 Applications

Some important applications such as MIMO wireless signal decoding depend on the solution of the BILS problem. MIMO stands for “multiple-input multiple-output”, it refers to the case where a wireless system has multiple input antennas transmitting a signal which is received by multiple output antennas. The signal received is our input vector y from (1.1), it has undergone some linear transformation by the known “channel matrix” A (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 we can solve the BILS problem. Of course we need not solve the BILS problem exactly, however 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 solution x. For this reason, we say that a receiver which is decoding transmissions using a BILS algorithm achieves “optimal performance”, where performance refers to the likelihood that the vector found by the decoder is equal to the transmitted vector x.

Another application of the ILS problem arises in global positioning systems 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 an integer, we can form a linear model for this system and obtain it by solving an ILS problem.

Other applications of BILS and ILS include cryptography and lattice design. For any application where the elements of x are known to be integer, we should use ILS. If the elements of x are drawn from some finite set, BILS is appropriate.

1.4 Previous Work

Due to the important applications of the BILS and ILS problems, much work has gone into solving them efficiently. There have been many algorithms proposed that yield a fast, approximate solution to the problem, some giving statistical bounds on the likelihood of error. This thesis however will only deal with finding the optimal solution to the problem. Even for finding the optimal solution to the ILS problem, there are a few different approaches that are quite different from each other. This thesis will further be restricted to considering only what are known as 'enumeration based' approaches which are most often used in practice because of their efficiency and simplicity. The enumeration based approaches, as the name implies, try to find the optimal solution by enumerating vectors in the search space until all possible solutions but one are eliminated.

1.4.1 Search Strategies

In chapter 2, it will be shown that the enumeration process can be reduced to a tree search problem. The most widely used algorithm, the Schnorr Euchner (SE) enumeration [11] can be thought of as a fairly straight forward depth first search in such a tree. Other tree search algorithms may also be used to solve the problem.

In the literature, some modifications to the best first tree search strategy have also been proposed, a few such proposals are given in [10], [13], [6], [3] and [15]. 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 and there is very little cost to visit a node in the tree. The advantage of the best first approach is that it can guarantee 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 enumeration algorithms for the ILS search process. One such paper is [10]. 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.

In [9], 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 ILS problem and one for the general ILS and BILS problem. Unfortunately, the computational complexity of computing these bounds can be prohibitive, this will be discussed later in the thesis.

There have been other suggestions to improve the enumeration based search algorithms as well. One such method [1] proposes 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.

1.4.2 Reduction Strategies

The standard reduction algorithm used in practice for the unconstrained ILS problems is the LLL reduction [8]. 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, the LLL reduction is not 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. We can separate algorithms that try and find these permutations 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 VBLAST [5] and SQRD [14]. 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 [12] in 2005, and then [2] in 2008. Numerical results show that these algorithms can offer great improvements over the previous reductions that use only the information contained in H.

1.5 Objectives and Contribution

With many different algorithms for both the reduction and search process, it is not always clear how they relate. In [10] the authors propose a tree search framework to compare various search algorithms. This is an interesting idea, but results from such a comparison may not relate to real world run time since they do not consider the time spent at each step of each algorithm, only how many steps are taken to solve the problem. This thesis will consider some of the various search strategies and do a more realistic comparison of actual run time performance. Using the results from such a comparison, we can see the advantages of the different approaches and use them to devise a new hybrid search algorithm.

For the reduction step, the LLL reduction [8] is the strategy most used in practice for the ILS problem. How the LLL reduction relates to the ILS 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 described in section 1.4. Two more recent developments, [2] and [12], 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. 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 SW algorithm 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 SW algorithm and offer some improvements to the original.

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 ILS problems as well. However, the 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 these vectors that performs well 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.

1.6 Outline

The rest of the thesis will be organized as follows;

In chapter 2, the Schnorr-Euchner (SE) enumeration algorithm [11] 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 ILS problems. Then, strategies for reducing BILS and ILS problems will be presented separately.

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 basic 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

There are many search algorithms that have been proposed to solve the unconstrained ILS problem. One of the most effective algorithms in terms of both overall runtime and memory consumption is the Schnorr-Euchner enumeration [11]. In this section, 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 factorization

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

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 ILS problem (1.5) is reduced to

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

We would like to enumerate as few elements as possible, $x \in \mathbb{Z}^n$ while still guaranteeing the optimal solution. Suppose we start with some initial bound on the error,

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

We will discuss some better ideas for choosing an initial β in later chapters, however one simple method is to choose x to be the real LS solution with each element rounded to the nearest integer, and then calculate the residual. 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".

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. Then (2.2) can be rewritten as

$$\sum_{k=1}^{n} r_{kk}^2 (x_k - c_k)^2 < \beta,$$

which is equivalent to the following set of inequalities:

$$\text{level } k: \ \ r_{kk}^2 (x_k - c_k)^2 < \beta - \sum_{i=k+1}^{n} r_{ii}^2 (x_i - c_i)^2, \tag{2.4}$$

Define for $k = n, n-1, \ldots, 1$.

We begin the search process at level n. Choose $x_n = \lfloor c_n \rceil$, the nearest integer to c_n. If the inequality (2.4) with $k = n$ is not satisfied, it will not be satisfied for any integer, this means β was chosen to be too small, it must be enlarged. 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 (2.4) 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 for the current level is no longer satisfied. When we reach level 1, we will have found an integer point $\hat{x}$. We then update $\beta = \|\bar{y} - R\hat{x}\|_2^2$ and try to find a better integer point which lies in the new, smaller ellipsoid. Finally in the search process, when we can no longer find any x_n to satisfy (2.4) 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 m, see Fig. 2, 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 $m..i$ we simply get the partial residual for fixing $x_{i:m}$, which is defined as $\|R_{i:m,i:m}x_{i:m} - \bar{y}_{i:m}\|_2^2$. This implies that each leaf in the tree represents an integer vector $x \in \mathbb{R}^m$ and its weight is the residual $\|Rx - \bar{y}\|_2^2$. This means that the ILS 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. 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 (1.6). Suppose $x_{k-1} : x_n$ 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.

CHAPTER 3
Reduction Strategies

Consider the ILS problem (1.5). 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 (1.5) becomes:

$$\min_{x\in\mathbb{Z}} \|QHx - y\|_2 \tag{3.1}$$

$$= \min_{x\in\mathbb{Z}} \|Q^T(QHx - y)\|_2 \tag{3.2}$$

$$= \min_{x\in\mathbb{Z}} \|Hx - \bar{y}\|_2 \tag{3.3}$$

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 $+/-1$. Let Z be some unimodular matrix. Another property of unimodular matrices is that the equation $Zx = b$ always has an integer solution x if b is integer. This property is very useful for our application. Consider applying such a unimodular matrix Z to H

from the right, (1.5) becomes:

$$\min_{x \in \mathbb{Z}} \|HZx - y\|_2 \tag{3.4}$$

$$= \min_{\hat{x} \in \mathbb{Z}} \|H\hat{x} - y\|_2 \tag{3.5}$$

$$\tag{3.6}$$

When we solve this new ILS problem, we will obtain some solution $\hat{x}$. If Z is a known unimodular matrix, we can solve the system $Zx = \hat{x}$ to find x, the ILS solution to the original problem. Note that if Z were not unimodular, we would have no guarantee that the solution x would be integer.

Finally, it is worth mentioning that we may apply permutation matrices to H as well, although they are just a special case of unimodular matrices. It is obvious that the effect on the solution from applying a permutation matrix is only to re-order the elements of the solution vector x.

When reducing the BILS problem we must be more careful. Consider the constraints on the solution x, (1.6). 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

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 V-BLAST algorithm [5] mentioned in 1.4 is a commonly used strategy to calculate the column permutations for H. Suppose we are working with R, which comes from the QR factorization 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 factorization, 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 diagonal element r_{kk} is maximal There are efficient algorithms to compute such a column ordering, one such algorithm will become obvious later in this section.

In [2], the SQRD column reordering strategy originally presented in [14] 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 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 [12], 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 allowed. Until this point, other column reordering algorithms only considered the matrix H.

Independently Chang and Han in [2] 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 [2] and it is easy for others to implement the algorithm. But our numerical tests 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.

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. 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 factorization of H, then tries to reorder the columns of R. The motivation for this algorithm comes from observing equation (2.4). 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 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. The CH column reordering strategy does this by trying to maximize the left hand side of (2.4) with large values of $|r_{kk}|$ and minimize the right hand side by making $|r_{kk}(x_k - c_k)|$ large for values of $k = n, n-1, \ldots, 1$.

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

$$\begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & x & x & x & \\ & x & & x & \\ & x & & & \end{bmatrix} \tag{3.7}$$

Equation 3.7 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 (3.8), denote such a Givens rotation as $G_{k,k+1}$:

$$\begin{bmatrix} I_{k-1} & & & \\ & c & -s & \\ & -s & c & \\ & & & I_{n-k-1} \end{bmatrix}, \quad c^2 + s^2 = 1 \tag{3.8}$$

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 $G_{n-1,n}, G_{n-2,n-1} \dots G_{i,i+1}$ to zero the subdiagonal elements in the i^{th} column, however this will create subdiagonal elements in columns $i + 1 \dots n$. Equation (3.9) shows the matrix after applying this first round of Givens rotatations.

$$\begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & x & x & \\ & & & x & \end{bmatrix} \tag{3.9}$$

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 $\hat{y}$. We then compute $c_n = \hat{y}_n / \hat{r}_{n,n}$ and

$$x_i^c = \arg \min_{x_i \in \mathcal{B}_i} |\hat{r}_{nn}(x_i - c_n)| = \lfloor c_n \rceil_{\mathcal{B}_i}\,, \tag{3.10}$$

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

$$\text{dist}_i^c = |\hat{r}_{nn}(\bar{x}_i^c - c_n)|, \tag{3.11}$$

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

$$\min_{\tilde{x} \in \mathbb{Z}^{n-1}} \left\| \bar{y}_{1:n-1} - R_{1:n-1,1:n-1} \tilde{x} \right\|_2\,, \tag{3.12}$$

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

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

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 factorization by Householder transformations, we get a total complexity of $0.5n^4 + 2mn^2$ flops.

3.1.3 SW Original Algorithm

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

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, [12] defines the affine sets, $F_i(\alpha) = \{w \mid g_i^T(w - h_i\alpha) = 0\}$. The lattice points generated by H occur at the intersections of these affine sets. Let the orthogonal projection of a vector s onto a vector t be denoted as $\text{proj}_t(s)$, then the orthogonal projection of some vector s onto $F_i(\alpha)$ is $\text{proj}_{F_i(\alpha)}(s) = s - \text{proj}_{g_i}(s - h_i\alpha)$. Therefore the orthogonal distance between s and $F_i(\alpha)$ is $\text{dist}(s, F_i(\alpha)) = \|s -$

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

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

$\text{proj}_{F_i(\alpha)}(s)\|_2$. In [12], 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.

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 find 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 [12].

The following will give an overview of the SW algorithm as given in [12] but described in a framework similar to what was used to describe CH. In the first step to determine the last column, for each $i = 1, \ldots, n$, we compute

$$x_i^s = \arg\min_{\alpha \in \mathcal{B}_i} \operatorname{dist}(y, F_i(\alpha)) = \arg\min_{\alpha \in \mathcal{B}_i} |y^T g_i - \alpha| = \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i}, \tag{3.13}$$

where the superscript s stands for the SW algorithm. Let $\bar{x}_i^s$ be the second closest integer in $\mathcal{B}_i$ to $y^T g_i$, i.e., $\bar{x}_i^s = \left\lfloor y^T g_i \right\rceil_{\mathcal{B}_i \setminus x_i^s}$. Let $j = \arg\max_i \operatorname{dist}(y, F_i(\bar{x}_i^s))$. Then SW chooses column j as the last column of the final reordered H, updates y by setting $y :=$ proj$_{F_j(x_j^s)}(y) - h_j x_j^s$ and updates G by setting $g_i :=$ proj$_{F_j(0)}(g_i)$ for all $i \neq j$. After G and y have been updated, the algorithm continues to find column $n-1$ in the same way etc. The pseudo-code of the SW algorithm is given in Algorithm 2.

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

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

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

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

$$\mathrm{dist}_i^s = \|g_i g_i^\dagger (y - h_i \bar{x}_i^s)\|_2 = |y^T g_i - \bar{x}_i^s| / \|g_i\|_2. \tag{3.14}$$

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 for $Hx = y$. The interpretation can be generalized to a general k.

In line 19 Algorithm 2,

$$\begin{aligned} g_i^{\text{new}} &\equiv \text{proj}_{F_j(0)}(g_i) \\ &= (I - \text{proj}_{g_j}) g_i = g_i - g_j (g_j^T g_i / \|g_j\|_2^2). \end{aligned} \tag{3.15}$$

Using the last expression for computation needs only $4n$ flops (note that $\|g_j\|_2$ has been computed before, see (3.14)). 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. For proof of this, see [4].

In line 17 of Algorithm 2,

$$\begin{aligned} y^{\text{new}} &\equiv \text{proj}_{F_j(x_j^s)}(y) - h_j x_j^s = (y - g_j g_j^\dagger (y - h_j x_j^s)) - h_j x_j^s \\ &= (I - \text{proj}_{g_j})(y - h_j x_j^s). \end{aligned} \tag{3.16}$$

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 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 (3.16). 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 (3.15) and (3.16)

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

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

$$\min_{\check{x} \in \mathbb{Z}^{n-1}} \|y - H_{:,1:n-1} \check{x}\|_2, \tag{3.17}$$

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 [12], 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 factorization. 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 factorization $H = Q_1 R$ 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 factorization $H^T H = R^T R$ and solve $R^T R G^T = H^T$ for G by using the triangular structure of R. The total 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 factorization 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 (3.10) and (3.13),

$$x_i^c = \lfloor (H^\dagger y)_i \rceil_{\mathcal{B}_i} = \lfloor e_i^T H^\dagger y \rceil_{\mathcal{B}_i} = \lfloor g_i^T y \rceil_{\mathcal{B}_i} = x_i^s. \tag{3.18}$$

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 (3.14) and (3.11)

$$\begin{aligned}\mathrm{dist}_i^s = \frac{|y^T g_i - \bar{x}_i^s|}{\|g_i\|_2} &= |\hat{r}_{nn}||y^T Q_1 V \hat{R}^{-T} e_n - \bar{x}_i^s| \qquad (3.19)\\ &= |\hat{r}_{nn}||\hat{y}_n/\hat{r}_{nn} - \bar{x}_i^s| = |\hat{r}_{nn}(c_n - \bar{x}_i^c)| = \mathrm{dist}_i^c.\end{aligned}$$

Now we consider the subproblem (3.12) in CH and the subproblem (3.17) in SW. We can easily show that $R_{1:n-1,1:n-1}$ in (3.12) is the R-factor of the QR factorization of $H_{:,1:n-1}P$, where $H_{:,1:n-1}$ is the matrix given in (3.17) and P is a permutation matrix such that $\check{x} = P\tilde{x}$, and that $\bar{y}_{1:n-1}$ in (3.12) is the multiplication of the transpose of the Q_1-factor of the QR factorization of $H_{:,1:n-1}P$ and y in (3.17). Thus the two subproblems are equivalent.

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.

After the QR factorization of H, we solve the reduced ILS 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 (3.18), 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. We give the pseudocode of the new algorithm in Algorithm 3.

Algorithm 3 New algorithm

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

Here we consider the complexity analysis of our 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 ILS Reduction Algorithms

For the unconstrained ILS problem, the most common reduction strategy is to apply the LLL reduction [8] 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 LLL algorithm as a matrix factorization, $H = QRZ$, where Q is orthogonal, R is upper triangular, and Z is unimodular. After this QRZ factorization, the matrix R will be LLL reduced. We can say an upper triangular matrix is LLL reduced if it satisfies the following properties:

$$|r_{k-1,j}| \leq \frac{1}{2} r_{k-1,k-1} \tag{3.20}$$

$$\sigma r_{k-1,k-1}^2 \leq r_{k-1,k}^2 + r_{k,k}^2 \tag{3.21}$$

$$j = k : n, k = 2 : n \tag{3.22}$$

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

$$|r_{k-1,k-1}| \leq \frac{2}{\sqrt{4\sigma - 1}} |r_{k,k}| \tag{3.23}$$

Looking at (3.23), 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 $\sigma = 1$. We know from previous discussion that it is desirable to have large diagonal elements, with the largest possible diagonal elements toward the end, $r_{11} < \cdots < r_{nn}$. The equation (3.23) 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 factorization described above can be computed. It is interesting to note that this factorization is not unique.

Integer Gauss Transformations

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

$$Z_{ij} = I - \mu e_i e_j^T, \quad \mu \in \mathbb{Z} \tag{3.24}$$

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:

$$\bar{R} = RZ_{ij} = R - \mu R e_i e_j^T \tag{3.25}$$

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

$$\bar{r}_{kj} = r_{kj} - \mu r_{ki}, \quad k = 1 \ldots i \tag{3.26}$$

If we take $\mu = \lfloor / frac r_{ij} r_{ii} \rceil$, it should be clear that $|\bar{r}_{ij}| \leq \frac{1}{2} r_{ii}$, so given a particular colum, we should be able to use IGTs to satisfy the first condition in equation (3.20).

Permutations

After doing IGTs, there is no guarantee that the second condition in (3.20) will be satisfied, often it is not. In this case, we must permute the columns of R in order for the conidtion 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. After the permutation, the second condition in (3.20) will hold. After performing this permutation, we also have the guarantee that element 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 the LLL conditions (3.20). We will start by letting $H = QR$ denote the QR factorization 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, $|r_{ik}| < \frac{1}{2} r_{ii}, \quad i = k - 1 : -1 : 1$. 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 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.

3.2.2 New Unconstrained ILS 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 unconstrained ILS problem as well. Applying the SW 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. 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. Unfortunately after applying the permutations, the LLL conditions in the new matrix $\hat{R} = PR$ may no longer be satisfied. Whether the search will be faster for $\hat{R}$ 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.

CHAPTER 4
Alternate Search Strategies

Chapter 2 described the most common search strategy used to solve both the ILS and BILS problems. Also, the search process was shown 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).

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

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

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.

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, [6] and [13] 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 explored during the search in memory and always keep track of the node with minimum cost. For this reason it is not always practical to use the BFS strategy, one reason is 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.

Unfortunately, some of the literature such as [3] 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.

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 integer 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 the 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" 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 $\left\| R_{n-1:n,n-1:n} x^p_{n-1:n} - \bar{y}_{n-1:n} \right\|_2^2$. 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. We then insert x_n into the priority queue with its "next child" and "next child cost".

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), $(R_{n,n}x_n^2 - y_n)$. 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 and one at level $n-1$. If the former is smaller, we will visit the second best child of x_n, the new x_{n-1} next, otherwise we will visit the best child of the node x_{n-1} 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. 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.

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 large 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 be faster. In both [15] and [3] 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 [3]. 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, and will 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 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 first α elements in S 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 first α nodes in S. When S is empty we may terminate with the optimal solution.

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 of 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 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.

4.3 Combining BFS and DFS

CHAPTER 5
Conclusions

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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