There Are 10 Kinds of Problems in the World: Those That Are Binary and Those That Aren’t

Shane Garrick
Carroll College, Helena, MT
There Are 10 Kinds of Problems in the World: 
Those That Are Binary and Those That Aren’t 

Shane Garrick 

April 24, 2006
This thesis for honors recognition has been approved for the Department of Mathematics, Engineering, and Computer Science.

Director: Dr. Mark Parker
Date: 4/24/06

Reader: Dr. Holly Zullo
Date: 4/24/06

Reader: Joan Stottlemyer
Date: 4/24/06
Abstract

Binary integer programming is a class of algorithms that are used to solve problems where we have several yes/no decisions, along with some constraints on which decisions we can make and a value associated with each decision. A classic example is the knapsack problem, where we have to choose what to carry in the knapsack. For each possible item, we can carry it or not. We are also limited by space and weight, and some objects will have a higher value than others. This paper examines binary integer programming and a simple computer implementation of a technique for solving such problems, called the Multiphase Dual Algorithm, which was developed by Dr. Fred Glover in 1965. We also test the execution time of this program compared to an implementation of a branch-and-cut algorithm developed by COIN-OR. Results show that this Multiphase Dual program executes in about the same time as the branch-and-cut algorithm on small problems, but starts to become slower on large programs. This indicates that although the Multiphase Dual algorithm may not be the fastest possible algorithm, it can solve binary integer programming problems in a reasonable time. We conclude by discussing other parts of the algorithm whose implementation would increase its efficiency.
# Contents

1 Introduction .................................................. 1

2 Binary Integer Programming ............................. 1

3 Exhaustive Search ........................................ 2

4 Binary Multiphase Dual Algorithm .................. 3
   4.1 Binary Search Tree Enumeration .................. 4
   4.2 Fathoming Tests Part 1 ............................ 7
   4.3 Surrogate Constraint ............................... 10
      4.3.1 Other S-Constraint Information .......... 10
      4.3.2 Finding a Strong S-Constraint .......... 11
   4.4 Fathoming Tests Part 2 ........................... 18
   4.5 Other Parts .......................................... 19
   4.6 Overall Algorithm .................................. 19
      4.6.1 Settling the Problem ....................... 21
      4.6.2 Removing Non-Binding Constraints ...... 22
      4.6.3 Setting a Variable ........................... 22
      4.6.4 Testing the Current Solution ............ 23
      4.6.5 Backtracking .................................. 24
      4.6.6 Example ......................................... 24

5 Program Design ........................................... 27
   5.1 ClpConstraint Class ................................ 27
      5.1.1 Summary ...................................... 27
      5.1.2 Details ........................................ 28
   5.2 ClpBinaryMultiphaseDual Class ................ 28
      5.2.1 Summary ...................................... 28
      5.2.2 Details ........................................ 29

6 Performance Tests ....................................... 29
   6.1 Testbed .............................................. 30
   6.2 Programs ............................................ 30
   6.3 Test Problems ...................................... 30
   6.4 Collected Data ...................................... 30
   6.5 Results ............................................. 31

7 Conclusion ................................................. 31
   7.1 Current Status .................................... 31
   7.2 Future Improvements ............................... 31
## A Program Header Files

<table>
<thead>
<tr>
<th>Section</th>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>IndexOrder.hpp</td>
<td>32</td>
</tr>
<tr>
<td>A.2</td>
<td>ClpConstraint.hpp</td>
<td>32</td>
</tr>
<tr>
<td>A.3</td>
<td>ClpBinaryMultiphaseDual.hpp</td>
<td>36</td>
</tr>
</tbody>
</table>
1 Introduction

There is a class of problems in which we have to choose between two options for several decisions. For example, if I go home, for every item I have, I have to choose to take it or not. There are also some requirements for what I can take, such as I cannot take more items than my car has space for and the weight of the items I choose to take cannot be more than my car can lift. Each item that I can take will also have value for having it: for example, including games will have a higher value than including homework, and I would like to maximize the value of the items I take. This problem also assumes that each decision is independent of all other decisions. For example, if I bring a Nintendo and a Nintendo controller home, we consider the values of those to be the sum of their individual values, although they would be more valuable together. Similar assumptions are also made about the constraints; i.e., combinations of items only weigh as much as the sum of their weights. Such a problem in which we have several binary decisions is called binary integer programming. A more practical example of a binary integer programming problem is crew scheduling, and companies like IBM and Northwest Airlines face such problems [MIPLIB].

Such a problem might seem trivial because there is only a finite number of possible sets of items I can take. We can find the value of each set of items I can take, throw out the ones that are too heavy or take too much space, and use the remaining set with the highest value. Unfortunately, if we add another item, then there are twice as many possible sets we would have to examine. Even if we can quickly test if a set meets all requirements, if we add more items, the time to test all possible sets will increase quickly. This paper goes through a description and computer implementation of a binary multiphase dual algorithm, which was proposed by Dr. Fred Glover [Glover, 1965]. This algorithm is similar to checking every possible set of items except we can throw away some sets early because we know that they will break one of the requirements. We go through some of the formal details of binary integer programming, then some simple ways of solving it; then we examine the multiphase dual algorithm and go through a few examples. Afterwards, we finish with a brief description of the computer implementation and test results, then finish with a few descriptions of how this program could be improved.

2 Binary Integer Programming

More formally, the binary integer programming problem is as follows:

Maximize \( cx \)
subject to \( Ax \boxdot b, \ x_i = 0, 1 \) \hspace{1cm} (1)

where \( \boxdot \) is the sense of a constraint for each row, either \( \leq \), \( \geq \), or \( = \).
\( x \) is the \( n \times 1 \) vector of the items that we can choose to include or not, where \( x_i = 1 \) if we choose to include item \( i \) or \( 0 \) if we do not include it. The individual \( x_i \) are known
as the decision variables, or sometimes just variables since they are the only variables in this problem. The whole vector $x$ is known as a solution.

$c$ is a $1 \times n$ vector of the benefits, where we have a benefit of $c_i$ to our total profit from including $x_i$. $cx$ is the total profit, sometimes referred to as value, for a given solution $x$. Higher profits are better than smaller profits, as long as the constraints are met, as explained below.

$A$ is an $m \times n$ matrix and $b$ is an $m \times 1$ matrix; both describe the constraints of the problem. First, we say that $m \parallel n$ is true for column vectors $m$ and $n$ if $m_i \parallel n_i$ for all $i$, where $\parallel$ is the sense of the inequality as explained above. If a solution $x$ allows $Ax \parallel b$ to be true, then the solution is feasible and we can consider it for a possible maximum solution. If this solution does not fit those constraints, then it is infeasible and not a candidate for the optimum solution. This is also a formal way of writing the constraints. If we multiply everything in the array, we have a system of linear inequalities:

$$
\begin{align*}
    a_{1,1} \cdot x_1 & + a_{1,2} \cdot x_2 & + \cdots & + a_{1,n} \cdot x_n & \geq, \leq, \text{or} & = b_1 \\
    a_{2,1} \cdot x_1 & + a_{2,2} \cdot x_2 & + \cdots & + a_{2,n} \cdot x_n & \geq, \leq, \text{or} & = b_2 \\
    \vdots & & \vdots & & \vdots & \\
    a_{m,1} \cdot x_1 & + a_{m,2} \cdot x_2 & + \cdots & + a_{m,n} \cdot x_n & \geq, \leq, \text{or} & = b_m
\end{align*}
$$

The goal of this optimization problem is to find a feasible solution whose profit is as great or greater than the profit of any other feasible solution.

We also look at a specific form of the binary integer programming problem:

$$
\begin{align*}
\text{maximize} & \quad cx \\
\text{subject to} & \quad Ax \leq b, \quad x_i = 0, 1
\end{align*}
$$

(2)

The difference between (1) and this problem is the latter has only $\leq$ constraints, while the former can have any type. It is easy to convert between the different types of constraints. For $\geq$ constraints, such as $x_1 + x_2 + \cdots + x_n \geq b$, we can derive a $\leq$ constraint by multiplying that row by -1 to get $-x_1 - x_2 - \cdots - x_n \leq -b$. In the case of an equality constraint, such as $x_1 + x_2 + \cdots + x_n = b$, we make two new constraints: $x_1 + x_2 + \cdots + x_n \leq b$ and $x_1 + x_2 + \cdots + x_n \geq b$. Since the only point where the last two inequalities intersect is on the equation, this is a valid replacement. Then we replace the $\geq$ constraint with a $\leq$ constraint, as explained previously.

Now that we have a firm definition of the problem, we look at the brute force method for solving it.

### 3 Exhaustive Search

One technique that we can use to solve this problem is to check all possible solutions. Since each variable $x_i$ can be only 0 or 1, there are a finite number of solutions. We can test all possible solutions, reject those that do not meet all of the constraints, and
record the feasible solution that gives us the highest objective function value. There are several ways to organize all possible solutions to evaluate all of them. One is the binary search tree, which is also similar to the binary multiphase dual algorithm, so we will examine it further here.

A binary search tree splits the current binary integer programming problem into two sub-problems: one problem with a variable set to 1 and another problem with the same variable set to 0. Then each of those sub-problems is recursively divided into more sub-problems until all decision variables are set. Eventually, that procedure will go through all possible solutions. An example with three decision variables is shown in figure (1).

The problem with an exhaustive search is that if we add another decision variable to the problem, we have twice as many solutions to check: the previous problem with the new variable set to 1 and the previous problem with the new variable set to 0. The number of solutions we test is $2^n$, where $n$ is the number of decision variables in the problem.

For comparison, while I was running some of my tests on the multiphase dual, I found the algorithm tested about a million solutions in a couple of hours. If an exhaustive search were to test solutions at the same rate, it would take a couple of hours to test all of the solutions for a problem with log 1,000,000 $\approx$ 20 decision variables. The smallest problem in our test set (except for the hand examples) has 27 decision variables, so this method is impractical for our purposes. With the discussion of the exhaustive search done, we are now ready to examine the binary multiphase dual algorithm.

4 Binary Multiphase Dual Algorithm

There are two main unique parts to the multiphase dual method: a constraint that is a combination of all of the constraints, called a surrogate constraint, or s-constraint for short, and a series of tests that we can use to tell if the sub-problem we are exploring has a feasible solution and if some variables have to be forced to 1 or 0 to maintain feasibility[Glover, 1965]. First, we will give an overview of the enumeration
of the binary search tree which will allow us to use these two techniques. Then we will explore these two parts and show how they fit together with a binary search to find the optimal solution.

4.1 Binary Search Tree Enumeration

As explained before, the binary multiphase dual algorithm is similar to a binary search tree. Here, we explain some terminology and other details about how we implement the binary search tree in this algorithm.

In the original problem, all variables that are not set are referred to as free. When we set a variable to 0 or 1 for a sub-problem, we call that variable fixed. We fix variables that were free until we run out of free variables or we are in a situation in which we know the current sub-problem has no feasible solutions. The reason this algorithm is multiphase is because it works in several phases where more variables are fixed. Once we reach a stopping point in examining a sub-problem, we backtrack and set the most recently fixed variables to free and look at a different sub-problem.

The way we keep track of where we are in the binary search tree is through a stack list which contains information about the current solution we are examining and variables that are fixed. It is a stack because when we are going through the algorithm, we include newly fixed variables at the end of the list, and we also look at the end of the list for the next sub-problem when we are done with the current sub-problem. If our list has the term i, then \( x_i = 1 \). Similarly, if our list has the term \( i \), then \( x_i = 0 \). We also underline a term in the list to indicate that we do not have to explore the other sub-problem. This situation can arise from the test we describe later or because we have already examined that sub-problem.

For an example of a list, let us look at 234. The 2 at the start of the list means the first variable we fixed was \( x_2 = 1 \). The 3 shows that the second choice we made is to set \( x_3 = 0 \) and we will not check the sub-problem where \( x_3 = 1 \). The next term shows \( x_4 = 1 \) and we will not test the sub-problem where it is 0. At this point, if there are only four total decision variables, then we have to set \( x_1 \) in order to come to a complete solution.

We go through a list like this by starting with a blank list, which shows that we start with all variables free. Then we select a variable to fix and add it to our list. We always fix variables to 1 initially because we can quickly test the solution if it is set to 0. Then we continue picking free variables and fixing them until we come to a terminal situation, which means we have no more free variables to fix. Our list should contain a value for all variables that we can test. This is similar to going to the bottom of the tree in figure (1). Then we backtrack and remove all variables from the end of the list that are underlined. The analogy with the binary search tree is this is going back up the tree until we reach a branch that we have not explored. At this point, the end of the list has a variable that is not underlined or the list is empty. If there is a variable in the list, we set the value of the variable to what it was not before (0 if it was 1 or 1 if it was 0); then we fix variables as we did previously. This is
like exploring a new branch in the binary search tree. In the case that every term in
our list is underlined and we have no list by the time that we are done backtracking,
then we have explored all possible solutions.

Here is an example of how this method proceeds if we look at a problem with
three variables:

\[ \underline{123} \quad \underline{123} \quad \underline{123} \quad \underline{123} \quad \underline{123} \quad \underline{123} \quad \underline{123} \quad \underline{123} \]

This set of solutions is also the same as what we would find in the tree in figure
(1). First, we set \( x_1 \) to 1, then \( x_2 \) to 1, then \( x_3 \) to 1, as shown in the first solution.
This is the same as going down the left side of the tree in figure (1). At this point, we
have explored all of that sub-problem, so we backtrack. The last term in the list is 3,
so we change that to \( \underline{3} \), as in solution 2, which means we look at what happens if \( x_3 \)
were set to 0 instead of 1. Once that is complete, we go back through the list. The 3
is already underlined, so we know all of the sub-problem containing all possibilities of
\( x_3 \) is complete, so we go back more. The next term is 2, so we change that to \( \underline{2} \) and
examine all possibilities where \( x_2 = 0 \) and \( x_3 = 0 \) or 1 (solutions 3 and 4). Once that
is done, we go back through the list. At this point (solution 4), our list is \( \underline{123} \). Both
2 and 3 are underlined, so we go back and change 1 to \( \underline{1} \). The second half of this
procedure is the same as the first half, except \( x_1 = 0 \) instead of 1. When we come to
the last solution, \( \underline{123} \), we back through all of the terms. When we do that, we know
that we have explored all possibilities in the problem and are finished.

There are a few methods to choose the next variable we fix. One is from the tests
that we will explain in the next section. They can decide if a variable needs to be
set in a certain way to keep a feasible solution. If we add a term from this, then we
underline it because we do not have to look at both sub-problems, only the one we
know will have a feasible solution. If there are no such variables, then we choose one,
using information from the s-constraint, that we believe will lead us to the optimal
solution, and set that variable equal to 1, not underlined. It will be set to 0 later
when we examine the other sub-problem.

Next, we look at an example problem to demonstrate how setting a variables in a
sub-problem works. Here is our example problem:

Maximize \[ 3x_1 + 5x_2 + 4x_3 + 1x_4 \]
Subject to \[ 8x_1 + 2x_2 + 2x_3 + 1x_4 \leq 7 \]
\[ 4x_1 + 4x_2 + 2x_3 + 6x_4 \leq 3 \]
\[ 2x_1 + 5x_2 + 2x_3 + 4x_4 \leq 8 \]

Let us say that we choose to set \( x_2 = 1 \). Our current list is 2, showing that we are
exploring the sub-problem where \( x_2 = 1 \), and we will later explore the sub-problem
where \( x_2 = 0 \). Then we replace \( x_2 \) with 1 in each constraint and subtract the new
constant term from both sides of the problem to get the following:
Maximize \[ 3x_1 + 5 + 4x_3 + 1x_4 \]
Subject to \[
\begin{align*}
8x_1 + 2x_3 + 1x_4 & \leq 5 \\
4x_1 + 2x_3 + 6x_4 & \leq -1 \\
2x_1 + 2x_3 + 4x_4 & \leq 3 
\end{align*}
\] (4)

Now, we have a new sub-problem with \( x_2 = 1 \). To satisfy the second constraint, we have to add non-negative numbers to get a negative number. Since there is no way to do that, there is no way to find a feasible solution to this sub-problem. The tests that we explain later will give more insight into situations in which there are no feasible solutions. Since there is no feasible solution, we go through the backtracking procedure. We start from the end of our list, 2, and undo the changes to the problem from all terms until the first term that is not underlined. Since our list has no underlined terms, we just undo the term 2, which results in problem (3). Then we explore the other sub-problem of the last term, which is setting \( x_2 = 0 \) fixed. Now our list becomes 2, indicating that we are examining possible solutions with \( x_2 = 0 \), and we already know the outcome if \( x_2 = 1 \) (no feasible solution). After setting \( x_2 = 0 \) in problem (3) and subtracting the constant terms from both sides, we obtain the following sub-problem:

Maximize \[ 3x_1 + 4x_3 + 1x_4 \]
Subject to \[
\begin{align*}
8x_1 + 2x_3 + 1x_4 & \leq 7 \\
4x_1 + 2x_3 + 6x_4 & \leq 3 \\
2x_1 + 2x_3 + 4x_4 & \leq 8 
\end{align*}
\] (5)

The new sub-problem differs from (3) only by the lack of the \( x_2 \) term. Since this is still a feasible problem, we continue this example by arbitrarily choosing to set \( x_3 \) to 1. Then we make the following sub-problem:

Maximize \[ 3x_1 + 4 + 1x_4 \]
Subject to \[
\begin{align*}
8x_1 + 1x_4 & \leq 5 \\
4x_1 + 6x_4 & \leq 1 \\
2x_1 + 4x_4 & \leq 6 
\end{align*}
\] (6)

Since \( x_3 = 1 \), we add the term 3 to our list, making the whole list \( 23 \). Then we continue the enumeration and test all sub-problems with \( x_1 \) and \( x_4 \) set to 0 or 1. Notice that the only feasible solution to the second constraint is \( x_1 = 0 \) and \( x_4 = 0 \), so our list will eventually become \( 2341 \) or \( 2341 \). When we reach that point, we still have a feasible solution, but no more free variables. We record the variables in this feasible solution and the solutions value, then backtrack. In backtracking, we undo the 4 and 1 in our list since those are underlined, and then we reach the 3 in our list. Since 3 is not underlined, we replace it with \( 3 \) and explore the sub-problems with \( x_3 = 0 \). Once we are done with the sub-problems of \( x_3 \), we backtrack. Since both the 3 and the 2 in our list are underlined, we backtrack through both of those, and we end up with an empty list, which indicates that our exploration of problem (3) is done.
One note about this method is when to check if a solution we have is a candidate for the optimal solution. We do not have a full solution until our list includes all decision variables, but we can say that any variable not specified in our list has a value of 0. If we adjust $b$ after each time we fix a variable, as in the previous examples, then we can quickly test if the current solution in the list is feasible by checking if all of the terms in $b$ are non-negative. If we set all free variables to 0 and if everything in $b$ is non-negative, then our constraints become $0 \leq b$, which is true if every element of $b$ is non-negative. Since we can quickly test if a solution is feasible when all free variables are 0, we always explore the sub-problem in which a variable is set to 1 before it is set to 0 because we have quick estimates if a variable is set to 0. Later, we will explain how to use a feasible solution as a constraint to skip exploring sub-problems for which there are no better solutions than one we have found. Finding a feasible solution quickly helps that constraint.

Next, we look at a few of the tests to tell if a sub-problem has a feasible solution.

### 4.2 Fathoming Tests Part 1

Here, we present the first four of eight tests. These tests can be understood without understanding the s-constraint, so we present them earlier than the other tests. These tests determine if there is a feasible solution for a given constraint in a sub-problem. For example, there are no feasible solutions to the constraint $1x_1 + 2x_2 + 3x_3 \leq -1$ since there is no way to add the positive numbers on the left side of that inequality to make a negative number.

Before we go into the details of the tests, here are a few notations that we commonly use:

- $\sum_{M, N} a_i$ is the largest sum we can get using as few as $M$ and as many as $N$ elements $a_i$ of $a$.
- $\sum_{M, N} a_i$ is the smallest sum we can get using as few as $M$ and as many as $N$ elements $a_i$ of $a$.

For example, if $a = 1, 2, 3, 4, -5$, then $\sum_{2, 4} a_i = 1 + 2 + 3 + 4 = 10$ and $\sum_{2, 4} a_i = -5 + 1 = -4$.

Next, we introduce $U$ and $L$, which help us make tests more effective.

$U$ is the upper bound on the number of free variables we can set to 1 and still have a feasible solution. Optimally, we want to pick a $U$ where we can find a feasible solution by setting $U$ variables, but setting $U + 1$ variables to 1 will always result in an infeasible solution. For example, if we have a single constraint $az \leq b_p$ with $n$ variables, then $U$ is the number in which $\sum_{U} a_i \leq b_p$ and $\sum_{U+1} a_i > b_p$ if $n$ is the size of $a$. For instance, in the constraint $-2x_1 + -3x_2 + 1x_3 + 2x_4 + 3x_5 + 4x_6 + 9x_7 \leq 2$, $U = 5$ because $\sum_{5} a_i = 1 \leq 2$, but $\sum_{6} a_i = 5 > 2$ if we choose the 6 terms with the smallest coefficients.

The tests examine all possible solutions using no more than $U$ variables set to 1. Any number will work for $U$ as long as there are no feasible solutions that have more than $U$ variables set, but our tests will be most effective if we can find the smallest
possible $U$. For the example above, we can use $U = 6$ or 7, but the tests will waste time checking solutions that have 6 or 7 variables set to 1 when those will not be feasible solutions. Picking a $U$ that is too small means the tests will skip examining some possible maximum solutions, so picking a $U$ that is too small will cause an error in the algorithm.

Unfortunately, finding a $U$ that fits all constraints at once is not as simple as finding it for a single constraint. There are a few methods for finding a strong $U$, such as using linear programming, using the s-constraint, and finding $U$ for every constraint in the problem. For simplicity, we always let $U$ be the number of free variables.

The counterpart to $U$ is $L$, the lower bound on the number of free variables that must be set to 1 to have a feasible solution to a constraint. The best choice for $L$ is a number for which we can find a feasible solution if we set $L$ or more variables to 1, but we cannot find a feasible solution by setting fewer variables to 1. We can also have a more relaxed $L$ by picking any number in which there are no feasible solutions if we set fewer than $L$ to 1, just as we can have more relaxed $U$s, but results will also not be as quick with relaxed $L$s. In a single constraint $ax \leq b_p$ with $n$ variables, this $L$ is the number in which $\sum_{L \leq i \leq n} a_i \leq b_p$ and $\sum_{L-1 \leq i < n} a_i > b_p$. For example, the constraint $-1x_1 + 2x_2 + 3x_3 + 2x_4 \leq -1$ has an $L$ of 1 because $\sum_{i=1}^{4} a_i = -2 \leq -1$, but $\sum_{0 \leq i \leq 4} a_i = 0 > -1$. $L$ is also hard to find for multiple constraints simultaneously. There are ways to estimate $L$ that are counterparts to the ways to estimate $U$. For simplicity, we always use 0 for $L$.

Once we have $U$ and $L$, we can run the following tests to determine whether we can force some variables to be set to 0 or 1. Test 1 is done whenever we find values for $U$ and $L$.

**Test 1** [Glover, 1965]/If $U < L$ or if $U = 0$, then we terminate exploration in the current branch.

This test should be done immediately after any change in $U$ and $L$ so we do not unnecessarily perform any of the more computationally intensive tests. If $U < L$, then one constraint requires that more variables are set to 1 than another constraint will allow. That conflict means that we cannot find a feasible solution, and we should terminate exploration into this part of the binary search tree. If $U = 0$, then no more free variables can be set to 1. We check the feasibility and the value of the current solution with all free variables set to 0. Either way, we stop exploration into this part of the search tree.

Tests 2-4 are done to every constraint $ax \leq b_p$ in an order that is specified when we go through the whole algorithm. Test 2 is the basic test to check if the current problem has no solution for the current constraint, while tests 3 and 4 use those results to shortcut the enumeration of the binary search tree before some variables are fixed.

**Test 2** [Glover, 1965]/If $\sum_{L \leq i \leq n} a_i > b_p$, then we terminate exploration of this branch.
The above test means that with the given restriction on the variables we can set by L and U, the smallest possible value of the left side will be greater than \( b_p \), so no solution will satisfy this constraint. This is a terminal situation and we should backtrack. For example, if we have the constraint \( 1x_1 + 2x_2 + 3x_3 + -1x_4 \leq -2 \) with \( L = 0 \) and \( U = 4 \), then we cannot find a feasible solution for this constraint because the smallest possible solution is -1 on the left side, which is not less than -2.

The next two tests are similar to test 2, except they try to force variables to either 0 or 1 when setting them to the other value will cause test 2 to fail. If a variable is set to either 0 or 1 by one of these tests, we underline the variable because that is the only way we can set the variable and maintain feasibility.

Test 3 /Glover, 1965/If \( a_k < \sum_{L+1-U+1} a_i - b_p \), where \( a_k \) is a term from \( \sum_{L,U} a_i \), then we fix \( x_k \) to 1.

This test checks if we can change one of the variables in the minimum constraint solution from 1 to 0 and still satisfy the constraint. One view is the right side of this inequality is the most slack we can have in the constraint if the variables are set to minimize this constraint (this value is usually negative). That slack allows the possibility of changing some of the variables that were set to 1 to be 0. Adding 1 to \( L \) and \( U \) reflects that we are changing a variable from 1 to 0, so we can set another variable to 1. If a variable in that summation is less than the spare room we have, the spare room provided by the variable is required to satisfy this constraint, so the variable is fixed to 1. Another way of viewing this is we can temporarily set \( x_k \) to 0 and run test 2. An example of when this test is effective is the constraint \( 1x_1 + 2x_2 + -15x_3 \leq -10 \). The smallest sum of the left side we can have is -15, so the spare room is \(-15 - (-10) = -5. -15 \) is the only term smaller than -5, so \( x_3 \) has to be 1.

Test 4 /Glover, 1965/If \( a_k > b_0 - \sum_{L-1,U-1} a_i \), where \( a_k \) is a term not from \( \sum_{L,U} a_i \), then we fix \( x_k \) to 0.

This test looks through all terms that were not included in finding the minimum sum in test 2 and checks if we can still have a feasible solution by changing one of them from 0 to 1. Once again, the right side is the amount of spare room this constraint can have if we find a solution that minimizes this constraint. Subtracting 1 from \( L \) and \( U \) reflects the possibility of changing a term that used to be 0 to 1. If any of those terms are greater than the amount of free space we have, then there is no possibility of setting that variable to 1 and still satisfying the constraint. We fix that variable to 0. For example, let us look at the constraint \( 1x_1 + 2x_2 + 15x_3 \leq 5 \). The smallest sum we can make is 0, so there is \( 5 - 0 = 5 \) space for free room. The term 15 is greater than 5, so \( x_3 \) has to be 0 to satisfy this constraint.

We should also note that sorting all of the variables by their coefficients helps us perform the \( \sum_{L,U} a_i \) operation quickly, as we can look at the ends of the sorted list to quickly find all of the variables that are affected by tests 3 and 4.
Next, we will look at the other major part of this algorithm: the surrogate constraint.

### 4.3 Surrogate Constraint

The *surrogate constraint*, or *s-constraint*, is used to make the tests run more efficiently. It has two main benefits for this algorithm: 1. We will likely find more results from this constraint than the normal constraints; and 2. It can estimate the optimal solution to the original problem, so we can estimate how we should set the variables early.

It acts as a single constraint that has some of the constraining powers of several constraints, which allows us to apply the fathoming tests to this one constraint and get results that we would find by testing several constraints. For example, it might be the case that the third constraint fixes a variable using test 3 or 4, and the fifth constraint might fix a different variable using test 3 or 4. Running tests 3 and 4 on a good s-constraint might fix both of those variables without having to test every constraint.

The s-constraint also gives us a good method for choosing the order to set the variables to quickly find an optimal solution. We have a method for quickly estimating the optimal solution to a problem that only has the s-constraint, and we expect the optimal solution to the original problem to be similar to the optimal solution to the s-constraint problem. If the first branch of our binary search tree follows the optimal solution to the s-constraint, we are likely to find the optimal solution quickly.

#### 4.3.1 Other S-Constraint Information

What gives the s-constraint these powers is that it is a non-negative linear combination of the normal constraints. We use the row vector of size $1 \times m$ $u \geq 0$ with at least one element greater than 0 to store that linear combination. If the s-constraint is $a \leq b_0$, then we find it with $a = uA$ and $b_0 = ub$.

One useful property of the s-constraint is every solution that is feasible to the original problem is also feasible in the s-constraint. That is because of the way the s-constraint is a linear combination of the normal constraints. If each constraint in the original problem has some slack before it becomes infeasible (i.e., the right side minus the left side of every inequality is non-negative), then the slack of the s-constraint is a linear combination of the slacks of the individual constraints. If a solution is feasible for all constraints, then all constraints have a non-negative slack, so the s-constraint will have a non-negative slack and will also be feasible. For an example, we can look at this problem:

\[
\begin{align*}
\text{Maximize} & \quad 3x_1 - 2x_2 + 1x_3 \\
\text{Subject to} & \quad 2x_1 - 2x_2 + 3x_3 \leq 1 \\
& \quad 4x_1 + 3x_2 + 5x_3 \leq 7 \\
& \quad 1x_1 + 1x_2 - 3x_3 \leq 0
\end{align*}
\]
After we derive the s-constraint from the original problem, we can make the s-constraint problem. Generically, it is this:

Maximize \( cx \)
Subject to \( ax \leq b_0, \ x_i = 0, 1 \) \( (8) \)

The first s-constraint we always examine is with \( u_i = 1 \) for all \( u_i \), so the specific s-constraint problem for \( (7) \) is as follows:

Maximize \( 3x_1 - 2x_2 + 1x_3 \)
Subject to \( 7x_1 + 2x_2 + 5x_3 \leq 8 \) \( (9) \)

If we pick a solution that is feasible for problem \( (7) \), such as \( x = [0, 0, 0] \), then we find that the first constraint has a slack of 1, the second has a slack of 7, and the third has no slack. The slack in the s-constraint is the linear combination \( u \) of the slack in the ordinary constraints. Since all of \( u \) is 1, the slack in the s-constraint is \( 1 + 7 + 0 = 8 \), which is still true if we plug that solution into \( (9) \). Since a constraint is satisfied if and only if it has non-negative slack, if all ordinary constraints are satisfied, all of them will have non-negative slack. When we run that through the non-negative linear combination defined by \( u \), we will still have non-negative slack in the s-constraint, so it is also satisfied by the given solution, and all feasible solutions in the original problem are also feasible for the s-constraint problem.

The converse is not true: the feasibility of a solution for the s-constraint does not imply the feasibility of the same solution for the original problem. That can happen if the slack for a couple of the individual constraints makes up for the negative slack taken by the infeasible constraints. For example, the solution \( x = [1, 0, 0] \) satisfies \( (9) \), but it violates the first and third constraints of \( (7) \).

Since the feasible solution set of the s-constraint is a superset of the feasible solution set of the original problem, the optimal solution of the s-constraint will always be at least as large as the optimal solution to the original problem when both use the same objective function since it always contains the optimal solution of the original problem. That also means that if the optimal solution for the s-constraint is also feasible for the original problem, then that is the optimal solution for the original problem. This helps us define what is a better, or stronger, s-constraint, but it also has the bonus that if we find an optimal solution to \( (8) \) that is also feasible for the original problem, then that solution is also optimal for the original problem.

4.3.2 Finding a Strong S-Constraint

Now that we have seen some of the useful properties of the s-constraint, we show how to obtain a strong s-constraint.

Criteria We consider s-constraint \( A \) to be stronger than s-constraint \( B \) if s-constraint \( A \) has a smaller optimal solution than s-constraint \( B \). Using \( (7) \) as an example prob-
lem, we will make a second s-constraint with \( u = [1, 0.5, 1] \) to compare with (9). The new s-constraint problem is as follows:

Maximize \( 3x_1 - 2x_2 + x_3 \)
Subject to \( 5x_1 + 0.5x_2 + 2.5x_3 \leq 4.5 \) \hspace{1cm} (10)

After trying every possible solution, we find the maximum solution to (9) is \( x = [1, 0, 0] \) with a value of 3, while the maximum solution to (10) is \( x = [0, 0, 1] \) with a value of 1. Since 1 < 3, we consider (10) to be the stronger s-constraint.

**Procedure** The basic procedure for finding a strong s-constraint is to find the optimal solution to (8), then find which constraints do not allow the optimal solution to (8). Then we adjust the combination of constraints we use, \( u \), to make a new s-constraint in which the violated constraints have a greater effect. We continue this until one of these conditions is met:

- We run through a predetermined number of iterations[Glover, 1965].
- The optimal solutions start to have higher (and more likely to be infeasible) values[Glover, 1965].
- The optimal solution for the s-constraint becomes feasible for (2)[Glover, 1965].

Next, we look at the specific steps we go through to find the s-constraint.

**Method 1**

1. Let \( u \) be the mx1 column vector that describes the combination of all of the constraints that make the s-constraint. This is initially set to 1 for all \( u_i \).

2. Maximize (8) for the current s-constraint.

3. If the maximum for the current s-constraint is greater than the maximum for the last s-constraint, then the last s-constraint is the one we use. That means that we are going farther away from finding an s-constraint with an optimal solution that will be feasible to (2). We also skip this step during the initial iteration.

4. If the optimal solution to (8) for the current s-constraint is feasible for (2), then we use the current s-constraint.

5. If we have run through a pre-set limit of iterations, then we quit and use the current s-constraint.
6. We adjust \( u \) to find a new \( s \)-constraint. First, we split \( u \) into two parts, \( F \), which is the part of \( u \) that represents constraints that are met, and \( G \), which is the part of \( u \) in which the constraints are violated. Then we can find \( f \), which is the sum of the amounts that the constraints are over-satisfied multiplied by the constraints' weights in \( F \). \( g \) is found similarly, except using the sum of how much the constraints are violated. Then we adjust \( G \) by multiplying it by \( \frac{L}{\theta} + \epsilon \), in which \( \epsilon \) is a small amount that we choose, and make the new \( u \) by combining \( F \) and the new \( G \).

7. Then we go back to step 2 and repeat the process with the new \( s \)-constraint from \( u \).

**Maximization for this Procedure** In order to run this procedure, we need an algorithm to quickly find an optimal solution to (8). The next theorem shows us how to make that a simpler problem, but we should also look at some informal justifications of why this works first.

A variable is *good* if we want to set it to 1 for a part of the problem, either the objective function or the constraint. Similarly, a variable is *bad* if we want to set it to 0. A variable is good for the objective function if its coefficient is positive since we want to maximize the objective function. Alternatively, a variable is bad for the objective function if it is negative. A variable is good for the constraint if it is negative because the constraint is a "less-than" constraint, so smaller numbers are better. The counterpart to that is positive numbers in the constraint are bad for the constraint. With this idea, we can justify how this next theorem works.

**Theorem 1** [Glover, 1965] If \( a_i \leq 0 \) and \( c_i \geq 0 \) then we set \( x_i \) to 1 because this variable is good for both the objective function and the constraint.

If \( a_i \geq 0 \) and \( c_i \leq 0 \), then we set \( x_i \) to 0 because this variable is bad for both the objective function and the constraint.

For an example of how to maximize a single constraint, let's look at the following problem:

Maximize \[ 3x_1 - 2x_2 + 1x_3 - 3x_4 + 5x_5 - 1x_6 \]
Subject to \[ 5x_1 + 3x_2 + 2x_3 - 1x_4 - 2x_5 - 3x_6 \leq 3 \]

(11)

Setting \( x_2 = 1 \) is bad for the constraint and the objective function because it reduces the objective function value and makes it harder to meet the constraint, so we set \( x_2 = 0 \). Similarly, setting \( x_5 = 1 \) adds to the objective function and eases the constraint, so we set \( x_5 = 1 \). None of the other variables is affected by theorem 1, so the problem becomes:

Maximize \[ 3x_1 + 1x_3 - 3x_4 - 1x_6 + 5 \]
Subject to \[ 5x_1 + 2x_3 - 1x_4 - 3x_6 \leq 5 \]

(12)
All of these variables are good for one part and bad for the other, so \( a_i, c_i \leq 0 \) or \( a_i, c_i \geq 0 \). We can normalize this so only one of those is true, or every remaining variable is good and bad for the same parts, by making a temporary vector \( v \), and finding the optimal solution to the following problem:

\[
\begin{align*}
\text{Maximize} & \quad v\bar{c} + \sum_i c_i \\
\text{Subject to} & \quad \bar{a}v \leq \bar{b}v_i = 0, 1 \\
\text{where} & \quad \bar{a}_i, \bar{c}_i = \begin{cases} a_i, c_i & \text{if } v_i = x_i \\ -a_i, -c_i & \text{if } v_i = 1 - x_i \end{cases} \\
& \quad \bar{b}_0 = b_0 - \sum /b_i
\end{align*}
\]

(13)

When \( \sum_i m_i \) is the sum of all \( n_i \) for which \( v_i = 1 - x_i \). To show how this works, we apply it to (12). First, we need to decide where \( u \) and \( x \) are different. Here, we let \( v_i = 1 - x_i \) if \( a_i, c_i \leq 0 \) or \( v_i = x_i \) if \( a_i, c_i \geq 0 \). We let the difference be \([0,0,1,1] \), which means we change \( x_4 \) and \( x_6 \). After we apply that change to (12), we get the following:

\[
\begin{align*}
\text{Maximize} & \quad 3v_1 + 1v_3 + 3v_4 + 1v_6 + 1 \\
\text{Subject to} & \quad 5v_1 + 2v_3 + 1v_4 + 3v_6 \leq 9
\end{align*}
\]

(14)

The major changes were that we added the coefficients of \( x_4 \) and \( x_6 \) to the constant term in the objective function and we subtracted the coefficients for \( x_4 \) and \( x_6 \) from the right side, along with changing the signs of those two variables. A sub-problem in this form is easier to solve than a sub-problem with variables that are both \( a_i, c_i \leq 0 \) and \( a_i, c_i \geq 0 \). After we solve for \( v \) in problem (14), we can convert \( v \) back to \( x \) with the array \([0,0,1,1] \) that we defined above. We continue by showing how to approximate a solution to problem (14).

All of the variables in the problem now have \( a_i, c_i > 0 \) or all \( a_i, c_i < 0 \). The next two theorems use this information to finish an approximate fractional solution.

**Theorem 2** (Glover, 1965) Assume \( a_i, c_i > 0 \). We let \( i_n \) be an ordering of \( \frac{c_i}{a_i} \) in which \( \frac{c_p}{a_p} > \frac{c_q}{a_q} \) implies \( p < q \). Then let \( r \) be the smallest integer for which \( \sum_{p \leq r} a_i \geq b_0 \). The optimal fractional solution to (8) is given by

\[
\begin{align*}
x_{i_p} &= \begin{cases} 1 & \text{if } p < r, \\ 0 & \text{if } p > r, \end{cases}
\end{align*}
\]

Then we assign the value of \( b_0 - (\sum_{p \leq r} a_i)/a_r \) to \( x_r \). If no such \( r \) exists, i.e., \( b_0 < 0 \), then this problem has no feasible solution.

For a better explanation, this theorem assumes that every variable is good for the objective function and bad for the constraint. \( c_i/a_i \) is the ratio of good to bad, or efficiency, if we choose variable \( x_i \). If \( c_i \) is high and \( a_i \) low, then setting \( x_i \) to 1 is more efficient (higher objective gain with a lower constraint cost) than if \( c_i \) is low and \( a_i \) is high. Then we use the index variables \( i_p \) to order \( x_i \) from most efficient to
least efficient. In the next part of this theorem, we choose the most efficient $x_i$s to set to 1 until we run out of constraint slack. Once we reach a variable that will not fit, instead of setting it to 0 or 1, we set it to a fraction between 0 and 1, depending on how much slack we have. Since the rest of the variables are less efficient than the ones we have chosen, we set them to 0.

To continue with our example problem, (14), we order the variables based on efficiency. The results are in Table 1. This table shows that $v_4$ is the most efficient variable, while $v_6$ is the least efficient.

The next part of our example of finding the optimal fractional solution is to show how to set the variables. $b_0 = 9$, so we add the constraint coefficients in the order specified by $i$ until the sum is greater than 9. We start with $v_4$, which has the coefficient $1 < 9$. Next we have $v_1$, which has a coefficient 5, which brings the total to $6 < 9$. The next most efficient variable is $v_3$, making the sum $8 < 9$. If we try to add the last term, $v_6$, the sum becomes $11 > 9$. Since four is the smallest number of terms that we added to make the sum larger than 9, $r = 4$. Therefore, $v_1$, $v_2$, and $v_3$ are set to 1 and nothing is set to 0 as specified in theorem 2. In the case of $v_4$, we use a fractional solution instead. We include as much of the variable as we can without violating the constraint. In this case, if we set $v_4$, $v_1$, and $v_3$ to 1, the total sum in the constraint is 8. $b_0$ is 9, so we have a slack of 1. The next most efficient variable is $v_6$, which has a coefficient of 3 in the constraint. We use the 1 unit of slack for $v_6$, so $v_6 = 1/3$. If we wanted the optimal fractional solution here, we would take the optimal $v = [1,1,1,1/3]$, undo the normalizing step, and and include the variables that we set early because they were good or bad to get the solution $x = [1,0,1,0,1,2/3]$.

Next, we present the dual theorem to theorem 2 if we decide that $a_i, c_i < 0$, or that all variables are bad for the objective function and good for the constraint.

**Theorem 3** [Glover, 1965] Assume $a_i, c_i < 0$. We let $i_n$ be the same ordering defined in the last theorem. Then let $s$ be the smallest integer for which $\sum_{p \geq s} a_{ip} \leq b_0$. The optimal fractional solution to (8) is given by

$$x_{ip} = \begin{cases} 1 & \text{if } p > s, \\ 0 & \text{if } p \leq s, \end{cases}$$

Then we assign the value of $1 - b_0 - \sum_{p \geq s} a_{ip}/a_{is}$ to $x_s$. If no such $s$ exists, i.e., $b_0 < \sum_{p} a_{ip}$, then this problem has no feasible solution.

The way to think of this last theorem is that we start with an infeasible solution with all variables equal to 0, and we need to set some to 1 to make it become feasible.

<table>
<thead>
<tr>
<th>Variable Index $t$</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_t/a_t$</td>
<td>0.6</td>
<td>0.5</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td>$i_t$</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Efficiency Orders used in Theorem 2
We start at the end of the list where $x_i$ is less efficient. We choose those because they have the biggest gain to easing the constraint while the smallest penalty to the objective function. In this situation, the variables at the end of the list are the most effective. When we reach a feasible solution, we take only a piece of the last variable. Our example uses the form specified in theorem 2, so there is no example here.

At this point, we need to find an actual binary solution. We can change the fraction optimal solution to an approximate integer solution with the following approximations.

**Approximation 1** [Glover, 1965] Let $i$ and $r$ be defined the same as in theorem 2, and let $a_i, c_i \geq 0$. Then we set $x_i$ to 0. This allows an amount of relaxation in the constraint. Then we run through the rest of the variables in the same order specified by $i$ starting with $i_{r+1}$ and set those to 1 if we have room in the constraints.

In our example with problem (14), this means first setting $v_6$ to 0. This brings us back to an integer solution and adds 1 unit of slack into the constraint. Then we continue through our list of variables searching for the variable whose coefficient will fit in the free space. In this example, there are no more less efficient variables after $v_6$, so no extra variables are set. As an example, if we had another variable $v_7$ with a constraint coefficient of 1 and an objective function coefficient of 0.1, then we would set that variable to 1. Since $c_7/a_7 = 0.1$, $v_7$ is the least efficient variable, and we would end up looking at that variable last, but it is the most efficient variable that can fit in the slack of the constraint, so we would set $v_7 = 1$.

**Approximation 2** [Glover, 1965] Let $i$ and $s$ be defined as in theorem 3, and let $a_i, c_i \leq 0$. Then we set $x_i$ to 1. This gives some relaxation in the constraint. Then we continue through the rest of the variables that were set to 0 and find the one that gives the smallest decrease in the objective function and would still allow the constraint to be feasible. In other words, we find the biggest $c_i/p \leq s$ and $a_i/p \leq \sum_{q \geq s} a_{iq} - b_q$. We set that variable to 1.

**Example** At this point, we have a method for a quick optimal binary solution approximation method that we can use with the s-constraint generation method. Next, we give a full example of generating an s-constraint with method 1 through one iteration with $\epsilon = 0.1$ for the following problem:

Maximize $3x_1 - 2x_2 + 4x_3 + 1x_4 - 2x_5$

Subject to $3x_1 - 2x_2 + 4x_3 - 2x_4 + 3x_5 \leq 4$
$2x_1 + 2x_2 + 2x_3 - 1x_4 - 2x_5 \leq 0$
$4x_1 + 5x_2 + 1x_3 - 1x_4 + 2x_5 \leq 7$ (15)

The first step is to let $u = [1, 1, 1]$, so our first s-constraint is

$9x_1 + 5x_2 + 7x_3 - 4x_4 + 3x_5 \leq 11$
In step 2, we maximize the current s-constraint with respect to the original objective function. Using thereom 1, we know the optimal solution will have $x_2 = 0$, $x_4 = 1$, and $x_5 = 0$. After those variables are removed, the remaining problem is the following:

\[
\begin{align*}
\text{Maximize} & \quad 3x_1 + 4x_3 + 1 \\
\text{Subject to} & \quad 9x_1 + 7x_3 \leq 15 
\end{align*}
\]  

(16)

Since it is the case for all variables that $a_i, c_i \geq 0$, we do not have to normalize this problem with (13). The next step of the maximization is to sort the variables based on efficiency as in theorem 2. The efficiency of $x_1$ is 1/3 while the efficiency of $x_3$ is 4/7, so $x_3$ is our most efficient variable. We find the optimal fractional solution sets these two variables to $[8/9, 1]$, as described in the second part of theorem 2. To get an approximate optimal integer solution, we use approximation 1. This sets $x_1 = 0$ to find a feasible solution. The algorithm then searches through the less efficient variables to find another variable to set to 1. Since there are no more variables, the end of approximation 1 is the solution $[0, 1]$. We include the results obtained from theorem 1 to get the full approximate optimal solution: $x = [0, 0, 1, 1, 0]$.

We skip step 3 since this is the first iteration and we have no previous s-constraint to compare the maximum to. In step 4, this solution is infeasible for the second constraint, so we do not quit. We also continue past step 5 since this is the first iteration and we have to check if the new s-constraint that we will make is better than the old one.

We adjust $u$ in step 6. $F = [1, 0, 1]$ because those are the parts of $u$ that satisfy the constraints. $G = [0, 1, 0]$ is the part of $u$ that violates the constraint. Then we find $f = 2 + 7 = 9$ (sum of over-satisfying the constraint), $g = 1$ (sum of exceeding the constraint), and the total change is $f/g + \epsilon = 9/1 + 0.1 = 9.1$. $u$ becomes $u = F + \text{change} \ast G = [1, 0, 1] + 9.1 \ast [0, 1, 0] = [1, 9.1, 1]$. Using this $u$, the new s-constraint is

\[
25.2x_1 + 21.2x_2 + 23.2x_3 + -12.1x_4 + -13.2x_5 \leq 11
\]  

(17)

This process continues by going back to step 2 of method 1, where we maximize the current s-constraint. To maximize this constraint, we first find $x_2 = 0$ and $x_4 = 1$ because of the first part of theorem 1. After we set those variables, adjust $b_0$ appropriately, and normalize, we have the following constraint:

\[
25.2v_1 + 23.2v_3 + 13.2v_5 \leq 23.1
\]  

(18)

If we ran through everything in theorem 2, we would first find the optimal fractional solution is $v = [0, 1, 0.98]$. Then when we use approximation 1 to find an integer solution, we set $x_5 = 0$ to get a feasible integer solution, and we end up with the solution $v = [0, 1, 0]$. After we reverse the normalizing process and combine all of the results, we find the solution is $x = [0, 0, 1, 1, 1]$. The value of this solution is 3, while the value of the approximate optimal solution to the last s-constraint is 5, so
we keep this constraint. Since this is the second iteration, we quit in step 5 and use (17) as our s-constraint with \( u = \{1, 9.1, 1\} \). But if we wanted to go through another iteration, we would find the total over-satisfying amount is 6, the under-satisfying amount is 1, and the first constraint is the only violated constraint, so we change the value of the first constraint in \( u \) by multiplying it by 6.1 to get \( u = \{6.1, 9.1, 1\} \). Then we would maximize and repeat the process in method 1.

Now that we have the s-constraint, we can examine more tests.

### 4.4 Fathoming Tests Part 2

Here are the tests that require the s-constraint. Tests 5, 6, and 7 work in a triplet like tests 2, 3, and 4, in which there is one test to check if there can be a feasible solution, and two tests that can force a variable to 0 or 1 based on results from the first test.

**Test 5** [Glover, 1965] We use the single constraint optimization technique as explained in theorem 2 or theorem 3. If we have found a feasible solution already, we compare the previously obtained maximum to \( c_0 \).

If this constraint would stop us from finding a better solution than we have already found, then we stop exploration of the current branch of the search tree and go back.

Tests 6 and 7 also work in a pair. They use test 5 to force a variable to 0 or 1 if we cannot find a better solution if the variable is set the other way.

**Test 6** [Glover, 1965] For all \( x_i \) in \( x \) that were set to 0, we change them to a value of 1 and find the new optimal solution for that constraint. If that solution is less than a solution we already have found, we know that changing that variable will make this constraint limit the objective function value to something worse that what we already have, so we have to fix this variable to 0.

**Test 7** [Glover, 1965] For all \( x_i \) in \( x \) that were set to 1, we change them to 0 and find the new optimal solution to that constraint. If the value of this solution is less than the value of a solution we already have, the constraint will stop us from finding a better solution if the variable is set to 0, so we fix it to 1.

Test 8 is used to tell if a constraint is non-binding, and in such a case, we temporarily remove it from the problem to make the problem simpler.

**Test 8** [Glover, 1965] If \( \sum_{L,U} a_i < b_0 \), then there is no way to set the variables to not meet this constraint, so we consider it non-binding.

Let us look at the constraint \( 1x_1 + 2x_2 + 3x_3 \leq 20 \) for an example. The largest sum of the numbers on the left side of that inequality is 6, which is less than 20. There is no way to find an infeasible solution to this constraint, so we can remove it
from the problem until we backtrack to a sub-problem with other variables that make it binding. Since this test is affected by how the variables are currently set, we can insert this constraint underlined into the list of fixed variables. This will also help us remember to include this constraint back into the problem when we are backtracking through that list.

### 4.5 Other Parts

There are a few other parts that make this algorithm run efficiently. One is using the objective function as another constraint. Since we are looking for a feasible solution that maximizes the objective function, we can throw out solutions that have a value that is less than the value of a feasible solution we have already found. A more mathematical description is as follows:

\[
    cx \geq b_M
\]

or

\[
    c_1x_1 + c_2x_2 + \cdots + c_nx_n \geq b_M
\]

When \(b_M\) is the right side of the objective function constraint. We can also write this as a “less-than” constraint by multiplying by -1 to be in the form of our other constraints:

\[
    -c_1x_1 - c_2x_2 - \cdots - c_nx_n \leq -b_M
\]

When we add the objective function constraint to our set of constraints, then we can test if we can find a better solution than the ones we have by applying tests 2-4.

We can also set \(b_M\) to 0 when we find a new solution if we adjust \(b_M\) when we backtrack and set variables. \(b_M\) will become the negative of the maximum solution when we backtrack to the point when all variables are free.

Now that we have all of the pieces of this algorithm, we are ready to see how they fit together.

### 4.6 Overall Algorithm

At this point, we have a set of tests that we can use to determine whether the current problem has a feasible solution and if any variables need to be set a certain way in order to maintain feasibility. We also have a way to estimate which way the variables should be set in order to find an optimal solution. The overall combination of these pieces is displayed in figure 2.

First, the binary multiphase dual algorithm generates the s-constraint. This is done once, and there is a trade-off between how long it takes to generate the s-constraint and how strong it is.
Figure 2: Overall Algorithm
Second, this algorithm repeats the tests on the constraints in the current problem to fix any variables that have to be fixed and possibly remove any non-binding constraints. The tests are repeated until we find they are returning no more results or we find that the current problem has no feasible solution. We call this part settling the problem. After this is done, there are three possibilities:

1. **No Feasible Solution** Sometimes we find there are no solutions that meet all of the constraints. In that case, we go to the backtracking phase.

2. **Simpler Problem** Most of the time, after the tests are finished, a few variables were fixed but many of the variables are still free. We pick a free variable to fix and settle the new sub-problem. The way we choose the variable is described later.

3. **Terminal Problem** Occasionally, we get to a point where test 1 shows we cannot set more variables to 1. In our case, that means we ran out of free variables, but it could be different if we used a stronger method of finding $U$. If test 1 shows $U = 0$, then we test the current solution to find if it is feasible and if its value is better than any previous maximums we have. Afterwards, we backtrack.

The first and third of these conditions are terminal. The second is non-terminal since there are still some free variables remaining. If it is terminal, then we determine whether the current solution is feasible and record it as a new maximum; if it is, then we backtrack. If backtracking shows we are done with the problem, then we quit. If we have a non-terminal condition after settling the problem, then we remove any non-binding constraints with test 8, set a variable, determine whether the new solution is feasible, then loop back to settling the problem. Next, we explain these parts in greater detail.

### 4.6.1 Settling the Problem

The point of settling the problem is to use our tests to fix all variables that will be fixed as quickly as we can. In doing this, we find there are three classes of constraints: normal constraints, the s-constraint, and the objective function constraint. We also find that our eight tests reduce to four groups that are computationally related. Test 1 is by itself and we use it after a variable is fixed. Test 8 is also by itself, but we use it in a different stage. Tests 2, 3, and 4 go together because test 2 is the main test, and tests 3 and 4 use test 2 to decide if a variable needs to be fixed to 0 or 1. Tests 5, 6, and 7 also go together since test 5 is the main test, and tests 6 and 7 use test 5 to fix variables to 0 or 1 if needed. When we run the groups of tests on a constraint (2, 3, and 4 or 5, 6, and 7), we run the first test to check if there is a feasible solution, then run the other two tests to find if there are any variables that need to be set.

Next, we need to know where these tests can be applied in relation to the constraints. Test 1 is independent of the constraints, so it is done after any variable is
fixed in any constraint. Tests 2, 3, and 4 can be done to any constraint, although
the objective function constraint requires a previously discovered maximum to work.
Tests 5, 6, and 7 are performed on only the s-constraint and require a known feasible
solution.

We also need to specify an order in which to run these tests. There are several
methods to do this, but some will fix variables faster than others. Since we expect the
s-constraint to return the most results, followed by the objective function constraint,
if applicable, we start our tests on those two functions. We give the tests the following
priorities:

1. Tests 2, 3, and 4 on the s-constraint
2. If we have a feasible solution, then tests 2, 3, and 4 on the objective function
   and tests 5, 6, and 7 on the s-constraint
3. Tests 2, 3, and 4 on all normal constraints

If we fix a variable after we are done with the tests on a constraint, we go back to
the start with the test on the s-constraint because we expect the s-constraint to return
better results. We repeat this procedure until a test returns a terminating result (no feasible solution or \( U = 0 \)), or none of the tests on the constraints return any results
after passing through all of the constraints. If there is a terminating result, we go to
the backtracking part and check for a new maximum solution if necessary. If there
are no results, then we go to the step where we remove the non-binding constraints.

4.6.2 Removing Non-Binding Constraints

To remove non-binding constraints, we run test 8 on all of the constraints except
the s-constraint and the objective function constraint. If there are some non-binding
constraints, then we remove those constraints from being active in the problem by
adding them to our list of fixed variables. Then we remove these constraints from the
s-constraint and go the the “Setting a Variable” step.

4.6.3 Setting a Variable

The goal of setting a variable is to fix a variable to 1 that we believe will be set
to 1 in the optimal solution to the problem. If we can find the optimal solution
quickly (although we will not know it immediately when we find it), then the objective
constraint will be more effective. Our method for this is similar to the method
about finding the optimal solution to the s-constraint because the s-constraint is a
summarization of all of the constraints, so we expect the optimal solution for it to be
similar to the optimal solution to the problem. We recall from section 4.3 that there
are three classifications of variables when we maximize a single constraint:

1. Good for both the constraint and the objective function
2. Bad for both the constraint and the objective function

3. Good for either the constraint or the objective function, and bad for the other

We can also divide the last item into two parts: good for the constraint and bad for the objective and good for the objective and bad for the constraint. Now we have four lists of variables. To find the next variable to fix to 1, we explain which list and which element we should choose.

1. Our first list that we should choose from is the list of variables that are good for both the constraint and the objective function. We choose a variable that has the highest objective to constraint ratio. This will give us a variable that we definitely want to set to 1 because it is good for everything in our s-constraint simplification of the problem.

2. If the last list is empty, we choose a variable from the good for one, bad for the other lists, since the list that is bad for the constraint and objective should be chosen last. The way we choose the variable depends on the feasibility of the current solution in the s-constraint. If the s-constraint is feasible if all free variables are 0 \((b_0 > 0)\), then we have some space to sacrifice in the constraint, so we choose an item from the list that is good for the objective and bad for the constraint. We choose the item that has the highest objective-to-constraint ratio in order to raise the objective value without making the constraints much more strict. For other case, in which the s-constraint is not feasible if all free variables are 0 \((b_0 \leq 0)\), we choose an item from the list that is good for the constraint and bad for the objective. This sacrifices the objective function in order to get a feasible solution. From that list, we choose the element with the highest constraint-to-objective ratio to give the most slack in our constraint without decreasing the objective too much.

3. If the list specified by the feasibility condition above is empty, then we pick a variable from the other list. We pick the variables in the same order.

4. If all lists with something good are empty, then we choose variables from the list in which the variables are bad for both the constraint and the objective function. We pick the variable with the largest constraint-to-objective ratio. This is the variable that will have the smallest reduction in constraint slack from how much objective value we give up.

Once we choose a variable, we set it to 1 and test the new solution.

4.6.4 Testing the Current Solution

We test the current solution in two different places, but both do essentially the same thing. We test if the current solution is feasible, and if it is, then we update the
information about the maximum solution. Since all free variables are assumed to be 0 when we test the current solution, we can quickly perform this test by checking if $0 \leq b$ and if $0 \leq -b_M$, since the left hand side of the constraints becomes 0 if all of the free variables are 0.

If this step comes after the "Setting a Variable" step, then we want to quickly check if we have a new maximum to update the objective function constraint before we go to settling the problem again. If we are testing the solution after a terminal condition, we are testing if the terminal condition was "no feasible solution" or "$U = 0." Then we update the maximum solution if necessary, then go to the backtracking stage.

### 4.6.5 Backtracking

After testing the current solution after a terminal condition, we backtrack. This is done as described in section 4.1. Once this is done, we should be either at a new subproblem to explore in the settle steps, or we might be done with the whole problem, in which case we print the results and quit.

### 4.6.6 Example

To understand this procedure better, we use it to solve (15) since we already have the s-constraint generated for it. We also simplify this problem by letting $U = n$ (the number of free variables), $L = 0$, and we skip tests 5, 6, and 7 since those tests require a lot of calculation that will distract from the process of solving the problem. Here is a repeat of the problem:

Maximize \[ 3x_1 + -2x_2 + 4x_3 + 1x_4 + -2x_5 \]
Subject to \[ 3x_1 + -2x_2 + 4x_3 + -2x_4 + 3x_5 \leq 4 \]
\[ 2x_1 + 2x_2 + 2x_3 + -1x_4 + -2x_5 \leq 0 \]
\[ 4x_1 + 5x_2 + 1x_3 + -1x_4 + 2x_5 \leq 7 \]
S-Constraint \[ 25.2x_1 + 21.2x_2 + 23.2x_3 + -12.1x_4 + -13.2x_5 \leq 11 \]

One note is that we write this problem in a form in which we can easily enumerate solutions and run tests. Although we may drop and add a column from the left side, the right side of the constraints is what is usually changed. Therefore, we can make a table with each constraint in its own column. At the bottom is what used to be the right side of the constraint. Then we can add a new row each time we set a variable. This problem is shown in table 2. The left column is the objective function constraint. The - show that we have not found a feasible solution. The middle three columns are the normal constraints. The right column is the s-constraint. We might add more columns on the right if we come to a case in which a constraint becomes non-binding and we start using a new s-constraint. After a row, we insert the list of variables that have been set. A table going through the complete problem is shown in table 3. Next, we discuss the operations in each entry of table 3.
<table>
<thead>
<tr>
<th>Objective Function Constraint (-c)</th>
<th>Normal Constraints</th>
<th>S-Constraints</th>
<th>Current Solution Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>-4</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>-4</td>
<td>4</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2: Tabular form of (19)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>25.2</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>2</td>
<td>5</td>
<td>21.2</td>
</tr>
<tr>
<td>-4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>23.2</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-1</td>
<td>-12.1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-2</td>
<td>2</td>
<td>-13.2</td>
</tr>
<tr>
<td>-4</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3: Solution Sequence to (19)

25
1) The original problem.

2) None of the tests yielded any results, so we go to the Setting a Variable step. $x_4$ in the constraint is good for everything, so we set that to 1. We also find that it is a feasible solution, so we set the limit of the objective function constraint $b_M = 0$, as described in section 4.5.

3) Once again, the tests had no results, and $x_3$ had the largest benefit in objective function for the restriction in the s-constraint, so we set it to 1.

4) Tests 2, 3, and 4 on the s-constraint show that we have to set $x_5 = 1$ and $x_1 = x_2 = 0$ in order to maintain a feasible s-constraint. Part of the resulting $b$ vector is negative, so we do not reset $b_M$ to 0.

5) We are at a terminal solution since every variable is set, so we backtrack until the first non-underlined term, 3. We underline it and negate it with the overline, then continue.

6) The tests show no results. The first constraint becomes $3x_1 - 2x_2 + 3x_3 \leq 6$. By test 8, we can remove this constraint. We add underlined constraint term $C_1$ to our solution, remove it from the set of constraints that we test, and remove its component from the s-constraint. This is shown by the lack of numbers in this column and the new column for the new s-constraint.

7) In the Setting a Variable step, $x_5$ is our next choice of variables to select from the new s-constraint.

8) and 9) Running tests 2, 3, and 4 on the objective function show we have to set $x_1 = 1$ and $x_2 = 0$ in order to find a solution with a better objective function value. This solution is feasible for all constraints since everything is positive, so we again reset $b_M$ to 0 to make a strong objective function constraint.

10) Backtrack and set $x_5 = 0$ fixed.

11) Test 4 on the s-constraint shows we have to set the remaining two free variables to 0 to keep the s-constraint feasible. The objective function constraint is violated, which means we do not update $b_M$. This also means that this solution has a worse objective function value than a previous solution.

12) Every term after 4 in the solution is underlined and we have a terminal solution, so we backtrack and set $x_4 = 0$ fixed. This involves including the first constraint back in the problem, so we revert back to our original s-constraint.
13) None of the constraints have any results from the tests, so we set \( x_3 = 1 \) in the Setting a Variable step.

14) Tests 2, 3, and 4 on the s-constraint show we have to set \( x_5 = 1 \) and \( x_1 = x_2 = 0 \) to prevent violating that constraint. We have also run out of free variables, so we check if this solution is feasible. The -3 in the first constraint column shows this is an infeasible solution, so we do not update \( b_M \).

15) The last solution was terminal, so we backtrack and set \( x_3 = 0 \) fixed.

16) We have to set \( x_1 = 0 \) because of test 4 on the s-constraint. After we do that, test 2 on the objective function constraint shows there is no solution that will satisfy that constraint, so this is a terminal solution and we backtrack. All terms are underlined, so we are done with this algorithm. When we quit, we find the maximum solution comes from line 9 because that is the last step where we reset \( b_M \) to 0. The solution there is \( x = [1, 0, 0, 1, 1] \) with a value of 2. We also note that that is the negative of the limit of the objective function constraint after we are done backtracking, \( -b_M = -(-2) = 2 \). That is because that constraint has become the same as \( cx \geq \) maximum solution.

Now that we have seen the hard way of going through this algorithm, we will discuss some of the computer implementation details.

5 Program Design

This main part of this program is the ClpBinaryMultiphaseDual class. It contains the objective function and the constraints, which are stored as ClpConstraints. It is responsible for loading the problem, along with running most of the algorithm. The other main class is the ClpConstraint, which holds the constraint and some other information to make the processing of the tests faster.

Next, we will take a bottom up approach to describe these classes to show what is good in the pieces before we move to the whole.

5.1 ClpConstraint Class

5.1.1 Summary

The ClpConstraint class holds information about a single constraint. It also holds information that makes processing of the tests faster. For example, the basic data structure, an array of a structure called IndexOrders, also contains a linked list that sorts that data from largest to smallest. When we run tests 2, 3, and 4 on a constraint, we can use that order to make the tests quicker. The linked list structure also allows us to quickly remove and insert elements from the list, which is used when we fix a variable in the problem.
5.1.2 Details

The main feature of this class is it holds the coefficients of the variables and right hand side of a constraint. It also does other management operations on those variables, such as the standard operations on a list, along with adjusting the right hand side when a variable is set and keeping the variables sorted.

One of the main benifits is how the variables are sorted. We use a linked list, as suggested by Dr. Glover [Glover, 2005]. This is good because when we set a variable and want to remove it from our list of sorted variables, a linked list will allow us to do so quickly.

The data structure for each node in our list is an IndexOrder. They hold data to manage three linked lists: one for the normal order, one for efficiency order, and one for the efficiency order in which +/- and -/ for objective/constraint matter. The first is used for tests 2, 3, and 4, the second is used for maximization, and the last is used for choosing the next variable to set during the Setting a Variable step. The first is used for all constraints, while the other two are used in only the s-constraint, but there are separate functions to prevent unnecessary sorting for most functions. There are also functions for sorting the IndexOrders for each list using bubble sort or quick sort.

In the area of functions, ClpConstraints have the normal functions for accessing data. They also have functions to sort by the various orders. They can also remove and add indexes and adjust the right hand side and the linked lists appropriately. There is also a maximum fractional solution and approximate integer solution function. The iterator functions are for the ClpBinaryMultiphaseDual class to access the elements in sorted order to run tests 2, 3, and 4. There are also constraint addition functions for when we generate s-constraints. Lastly, there is a function telling us which variable we should use in the Setting a Variable step for the s-constraint. Relevant header files, IndexOrder.hpp and ClpConstraint.hpp, appear in appendix A.

5.2 ClpBinaryMultiphaseDual Class

5.2.1 Summary

The ClpBinaryMultiphaseDual class is the main class that runs most of this algorithm. It is also a subclass of the ClpBinaryModel class, which is a subclass of the ClpModel class. The ClpModel class is the base class of the open-source COIN linear programming package (www.coin-or.org). We chose to make a subclass of this class because it has the functions to load a problem from an MPS file, which is a standard file format for mathematical programming problems and which all of our test data are stored in, and it can run the linear programming solver, which could be useful if we decide to use a stronger method for estimating $U$ and $L$ in a future version.

This class is responsible for most of the actions in this algorithm, such as running the tests on the constraints and keeping track of the fixed and free variables. First, it has functions for loading a problem from an MPS file. Then it can make the s-
constraint. It also runs the tests on the variables in the settle stage, along with finding the next variable to set and checking for a new maximum solution and backtracking. It also maintains the stack of the fixed variables and keeps track of the current solution and which constraints are binding.

To run this solver, first we make an instance of the class. Then we load the MPS file, whose function is also overloaded to convert the constraints that were stored in the COIN-OR sparse matrix to our local ClpConstraint format. The next step is to run the s-constraint generating function. Afterwards, a call to the solve function will print results as new, and better feasible solutions are discovered. The final output is the optimal solution and the time the program ran to find that solution.

5.2.2 Details

For functions in the background, this has all of the parts of the algorithm that the ClpConstraint class does not. A few of the major functions are the tests that are run on each constraint, functions to get U and L, and functions to deal with larger blocks of the algorithm, such as settling the variables, finding non-binding constraints, and setting the next variable if the tests had no results. There are also functions that deal with our solution stack. Most of these are called directly from the solve function.

In the background, there are two major concerns about how this implementation works: the way we store the solution stack and how results from the tests are passed. The solution stack is a static array of SetVarGroups that we use as a stack. Each SetVarGroup element includes the variable index, value, and if it is fixed or not. In the case that we have a constraint on the stack, there is an isNonBindConstraint variable that says this is a constraint instead of a variable, and a nonBindConstraint, which gives the index of the constraint. The newSc variable also points to the old s-constraint if we had to make a new one. This way, when we return to the old s-constraint, we do not have to worry about putting the variables back in the old order.

The second possible confusion is the test_ret enum. When we run the test, there are several strengths of results we can have. This return value can tell us if something was set during a test, or if the test shows the solution is infeasible and we should backtrack. Any place that uses this also recognizes that these variables are not equal, since if anything is set to one, then we have to do the actions for if something is set to one, and if any result returns infeasible, then the current problem is infeasible. The header file, ClpBinaryMultiphaseDual.hpp, appears in appendix A.

6 Performance Tests

Here, we describe the tests we used to compare this multiphase dual algorithm to another algorithm to see how well it performs
<table>
<thead>
<tr>
<th>Problem</th>
<th>Columns</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stein27</td>
<td>27</td>
<td>118</td>
</tr>
<tr>
<td>Stein45</td>
<td>45</td>
<td>331</td>
</tr>
</tbody>
</table>

Table 4: Test Problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Branch and Cut Time (Sec)</th>
<th>Multiphase Dual Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stein27</td>
<td>5.38</td>
<td>3.37</td>
</tr>
<tr>
<td>Stein45</td>
<td>295.07</td>
<td>785.62</td>
</tr>
</tbody>
</table>

Table 5: Execution Times

6.1 Testbed

These tests were performed on an Athlon 2500+ computer with 512 MB of RAM. The operating system is a base install of Slackware Linux 10.2 running the Linux 2.4.31 kernel. All tests were performed in multi-user mode after a clean boot.

6.2 Programs

The main program run here is the previously explained implementation of this binary multiphase dual algorithm. It is a basic driver that loads the problem from a specified MPS file, then solves it.

The program we tested it against is an implementation of a branch and cut algorithm from COIN-OR(www.coin-or.org).

6.3 Test Problems

The testset that we used comes from MIPLIB 3.0, which is a collection of mixed integer programming problems [MIPLIB]. We have chosen the problems in table 4 to test our programs on because they are all binary integer programming problems.

These problems were also rated as green in MIPLIB 2003. The definition of a problem rated as green was that it could be solved in under an hour with a commercial solver. We also tried a few problems rated yellow, but those did not appear to be solved in a reasonable time.

6.4 Collected Data

After running these programs on these test problems, we have the execution times in table 5.
6.5 Results

At the current stage, this binary multiphase dual algorithm can solve small problems in a reasonable amount of time, but as the problems become larger, this program starts taking proportionally more time than the COIN-OR branch and cut program. For example, the binary multiphase dual beats the cbc program 3 seconds to 6 seconds when running stein27, but in stein45, the cbc program takes 5 minutes while this binary multiphase dual program takes about 13 minutes.

7 Conclusion

7.1 Current Status

The current status of this program is that it is not as efficient as current binary integer programming techniques. For example, in the last section, we saw that this program scaled worse than the COIN-OR branch and cut program.

7.2 Future Improvements

There are several techniques that could improve the performance of this algorithm that were not tested in this program. For example, one of the major improvements would be the implementation of $U$ and $L$ in the program. We used the most relaxed forms of those, and using stricter values could increase performance. Another improvement could be in the $s$-constraint generation technique. In one paper from Dr. Glover, he mentions using an LP relaxation of the problem. The ClpBinary-MultiphaseDual class is derived from the linear programming class of COIN-OR, so implementing an LP relaxation $s$-constraint generation technique would be straightforward, but still not trivial. We could also adjust the number of iterations and $\epsilon$ in the current $s$-constraint generating method to try to find strong $s$-constraints. Another improvement would be how we put the pieces together in this algorithm. [Glover, 1965] explains that each test works, but there is still some room in how the tests are combined. Trying different modifications of this algorithm could show that one method works better than others.
A Program Header Files

A.1 IndexOrder.hpp

```cpp
#ifndef INDEXORDER_HPP
#define INDEXORDER_HPP

#include <math.h>

struct IndexOrder
{
    int index;
    double value;
    IndexOrder* low;
    IndexOrder* high;
    IndexOrder* lowRatio;
    IndexOrder* highRatio;
    IndexOrder* chooseLow;
    IndexOrder* chooseHigh;
};

void sortIndexOrders(IndexOrder* low, IndexOrder* high);
void sortIndexOrderRatios(IndexOrder* low, IndexOrder* high, const double* aux);
void bubbleSortIndexOrders(IndexOrder* low, IndexOrder* high);
void bubbleSortIndexOrderRatios(IndexOrder* low, IndexOrder* high, const double* aux);
void quickSortIndexOrders(IndexOrder* low, IndexOrder* high);
void quickSortIndexOrderRatios(IndexOrder* low, IndexOrder* high, const double* aux);

#endif
```

A.2 ClpConstraint.hpp

```cpp
#ifndef CLPCONSTRAINT_H
#define CLPCONSTRAINT_H

#include "IndexOrder.hpp"
#include <memory.h>
#include <math.h>
#include <stdio.h>

#endif
```
class ClpConstraint
{
private:
//main storage for the list
IndexOrder* mainlist;
int length;

//normal high/low pointers
IndexOrder high,low;

//for the efficient orders in theorems 1 and 2
//three groups, sort by efficiency
//some of these are also backwards
//because of the +/- division
IndexOrder zeroLowRatio,zeroHighRatio;
IndexOrder oneLowRatio,oneHighRatio;
IndexOrder mixLowRatio,mixHighRatio;

//sub-division of the mix orders above
//good for objective, bad for constraint
//(set to 1 in set var step)
IndexOrder choosePosLow,choosePosHigh;
//good for constraint, bad for objective
//(set to 0 in set var step)
IndexOrder chooseNegLow,chooseNegHigh;

//the iterator for running test 2, 3, and 4
//from the ClpBinaryMultiphaseDual class
IndexOrder *iterator;

//right hand side of this constraint.
double rhs;

public:
//constructor with length set
ClpConstraint(int l = 0);
virtual ~ClpConstraint();

//various functions to sort the variables
//normal order
void makeOrder();
//efficiency order from scratch
//(used in s-constraint only)
void makeRatioOrder(const double* objfunc);
//efficiency order for a new s-constraint
//(used in s-constraint only)
void resortRatioOrder(const double* objfunc);
//sort the zero and one lists separately because
//those do not need to be sorted sometimes
//(used in s-constraint only)
void makeOtherRatioOrder(const double* objfunc);
//split the mix order list into pos and neg
//(used in s-constraint only)
void splitMixOrder();

//self-descriptive
inline int getLength() const {return length;}
int setLength(int l);

//get and set the coefficient of variable i
double& value(int i);
double value(int i) const;

//get and set the right hand side
inline double getRhs() const {return rhs;}
void setRhs(double val);

//funcs to set and unset a variable
//these involve adjusting the rhs and adding or
//removing the variable from the linked list
void removeIndex(int i,double value);
void addIndex(int i,double value);

//max approximate integer solution
//maximizes using approx 1
//obj is an array that is the objective function
//objvalue is a reference param for the end value
//x is an array for the solution if it is there
//0 if OK or 1 if no feasible solution
int maximize(const double* obj,double& objvalue,
             double* x = NULL);

//max fraction solution
//params and vars same as
//maximize function
//except r, which points to the
//fractional part of the solution
//if it was inputed
int maxFracSolution(const double* obj, double& objvalue, double* x = NULL, IndexOrder** r=NULL);

//old function to make u
//now we just adjust the old constraint
//instead of making a new one
//void makeCombo(ClpConstraint* constraints, int numConstraints, const double* u, const double* usedVars = NULL);

//test how much a given solution satisfies the constraint
double testConstraint(double* solution);

//iterator functions for the minimum sum
//can go through the variables in order
//of big->small or small->big coefficients
//first, set high or low
//then move the other way
//until validIterator() returns false
//getIteratorIndex() and getIteratorValue()
//get information about the current node
void setIteratorHigh();
void setIteratorLow();
void moveIteratorHigh();
void moveIteratorLow();
int getIteratorIndex();
double getIteratorValue();
bool validIterator();

//for the s-constraint, this func returns the
//next variable we should set to 1 in the
//set a variable step of the algorithm
int getHighIndex();

//add a constraint, used for making the s-constraint
//c is the constant multiplier
//constraint is the constraint to add
void addConstraint(double c, const ClpConstraint &constraint);

//set constraint to 0
void setZero();
//void setEqual(const ClpConstraint &constraint);

//copier
ClpConstraint& operator= (const ClpConstraint & rhs);

#endif

A.3 ClpBinaryMultiphaseDual.hpp

#ifndef CLPBINARYMULTIPHASEDUAL_HPP
#define CLPBINARYMULTIPHASEDUAL_HPP

#include "ClpBinaryModel.hpp"
#include "IndexOrder.hpp"
#include "ClpConstraint.hpp"
#include "CoinTime.hpp"
#include <stdio.h>

//struct that hold information
//put in our solution stack
struct SetVarGroup
{
  int index;
  double value;
  bool fixed;
  ClpConstraint* newSc;
  int nonBindConstraint;
  bool isNonBindConstraint;
};

class ClpBinaryMultiphaseDual : public ClpBinaryModel
{
  private:
  //the normal constraints
  int numConstraints;
  ClpConstraint* constraints;

  //current s-constraint
  ClpConstraint* sc;
  //current const combo for s-constraint

double* ucombo;

//current objective function constraint,
//but only active if haveSolution is true
ClpConstraint* objConstraint;

//true if c0 (curMax) is valid, false if not
bool haveSolution;

//current max for the objective
//func constraint with a found solution
double curMax;
double curValue;

//the current solution that we are examining
double* curSolution;

/*the best found solution, only valid if haveSolution*/
double* curMaxSolution;

//our current solution stack
SetVarGroup *svStack;
int svStackSize;

//self-explanatory
int numFreeVars;
int numSetVars;
int U,L;

//for each row(constraint), 1 if binding, 0 if not
double* bindingConstraints;

//for each column(variable), 1 if free, 0 if not
double* freeVars;

//inverse objective func since this
//is a maximizing algorithm and
//MPS files are implied minimization
double* maxObjCoefficients;

public:

//return vals for test result functions
enum test_ret
{
    //nothing done
    TR NOTHING,
    //something was set, but not to 1
    TR NOTHINGSET,
    //var forced to 1, new solution
    TR ONESET,
    //constraint is infeasible
    TR INFEASIBLE
};

//normal constructor/destructor
ClpBinaryMultiphaseDual();
virtual ~ClpBinaryMultiphaseDual();

//main user functions

//overloaded readMps
//vars and return are same as Clp::readMps
int readMps (const char *filename,
            bool keepNames=false,
            bool ignoreErrors=false);

//generate s-constraint after loading problem
void genSConstraint(int maxIterations = 5,
                    double limit = 0.1);

//after everything is loaded, solve the problem
int solve(double* x, double& value);

//the rest of these are more of the
//background functions

//normal accessors
inline int getVarStackSize()
{ return svStackSize; }
inline int getNumSetVars()
{ return numSetVars; }
inline int getNumConstraints()
{ return numConstraints; }

//for the readMps function
//convert constraints from the COIN-OR sparse matrix to local format
int loadConstraints();

//convert minimize to maximize functions
//overloaded func to return maximizing objective
//instead of minimizing objective
const double* getObjCoefficients() const;
//make the objective function into a constraint
void genObjConstraint();

//testing functions
//getU and L get the value of U and L
int getU();
int getL();
//generate uses a technique to revise the values of U and L
int generateU();
int generateL();
//straight test, as describe in the paper
test_ret test234(ClpConstraint* c);
test_ret test567(ClpConstraint* c);
bool test8(ClpConstraint* c);

//functions that roughly correspond to boxes
//in the algorithm diagram in the paper
//settlevars function as describe in paper
test_ret settleVars();
//func to find non-binding constraints
//then re-adjusts the s-constraint
//and resorts
void findNonBindingConstraints();
//function to get the next var to set
//as describe in the paper
int getNextSet();

//solution stack manipulation functions
//set and pop vars and set constraint non-binding
//adjust the stack and all constraints
//var is var index, value is 0 or 1,
//fixed if fixed or not
int setVar(int var, double value, bool fixed);
//no value since we do not need it
int popVar(int &var, bool &fixed);
// i is constraint index
int setConstraintNonBinding(int i);

// check if next var on stack is fixed
// used when we are backtracking
bool isNextVarFixed();

// functions to test the solutions
double getCurSolutionValue();
bool testCurSolution();

// debug func to test if the s-constraint
// is being modified correctly after
// a constraint is found non-binding
bool testUCombo();

// print functions for debugging
void printConstraintOrderings();
void printSConstraintOrderings();
void printObjConstraintOrderings();
};

#endif
References

