2011

Parallel algorithm for solving integer linear programs

David O. Torrey Jr.
Michigan Technological University

A Parallel Algorithm for Solving Integer Linear Programs

By

David O. Torrey, Jr.

A Report

Submitted in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE IN MATHEMATICAL SCIENCES

Michigan Technological University

2011

This report, "A Parallel Algorithm for Solving Integer Linear Programs", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN MATHEMATICAL SCIENCES.

Mathematical Sciences

Signatures:

Report Advisor

Donald L. Kreher

Department Chair

Mark S. Gockenbach

Date

Abstract

Linear programs, or LPs, are often used in optimization problems, such as improving manufacturing efficiency or maximizing the yield from limited resources. The most common method for solving LPs is the Simplex Method, which will yield a solution, if one exists, but over the real numbers. From a purely numerical standpoint, it will be an optimal solution, but quite often we desire an optimal *integer* solution. A linear program in which the variables are also constrained to be integers is called an *integer linear program* or ILP. It is the focus of this report to present a parallel algorithm for solving ILPs. We discuss a serial algorithm using a breadth-first branch-and-bound search to check the feasible solution space, and then extend it into a parallel algorithm using a client-server model. In the parallel mode, the search may not be truly breadth-first, depending on the solution time for each node in the solution tree. Our search takes advantage of pruning, often resulting in super-linear improvements in solution time. Finally, we present results from sample ILPs, describe a few modifications to enhance the algorithm and improve solution time, and offer suggestions for future work.

Acknowledgments

I would like to thank Michigan Technological University for offering the opportunity to take classes for free as an employee. Without that, I would not have reached the point where I even considered pursuing an advanced degree.

To my current employer, ThermoAnalytics, my thanks for the funding and work schedule flexibility to allow me time to complete my degree. Had it not worked out, I may have dropped the idea in its early stages.

To my advisor, Dr. Kreher, thank you for encouraging me to continue in the area of discrete mathematics and for putting up with a non-traditional graduate student who worked, shall we say, at his own pace. You didn't give up on me when I expected you to, and that helped me stay the course.

And finally, to my wife Katie, thanks for taking up the slack on evenings and weekends, goading me into working when I didn't want to while still making sure the kids knew who I was, and putting up with descriptions of mathematical concepts, algorithmic nuances, and programming subtleties in excruciatingly vivid detail. I couldn't have survived it without your support.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction, Notation, and Background

Linear programs are often used to optimize manufacturing efforts or maximize the use of limited resources. For example, a factory might like to maximize their output, profit, or efficiency given constraints such as the cost of raw materials, time required to produce various products, and the profitability of each part. The solution to such a problem is the quantity of each product desired, and the maximized target figure. Given a set of limited resources, a set of possible products or goals, and the cost of building each product or achieving each goal, a linear program will produce the optimum quantities of product[6].

Definition 1. *Let $A = [a_{ij}]$ be an $m \times n$ matrix where $a_{ij} \in \mathbb{R}$, $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ be vectors. Then a linear program (LP) is defined as finding $X \in \mathbb{R}^n$ such that we maximize $Z = c^T X$, subject to $AX \leq b$ [6].*

Example 1. *The set of equalities in the following linear program can be visualized as the solid cube with edge length 10.5, having one corner at the origin. For simplicity, we choose to maximize the sum of the coordinates:*

$$\begin{aligned} \textit{maximize } Z = x_1 + x_2 + x_3 \\ x_1 \leq 10.5 \\ x_2 \leq 10.5 \\ x_3 \leq 10.5 \\ -x_1 \leq 0 \\ -x_2 \leq 0 \\ -x_3 \leq 0 \end{aligned}$$

An obvious solution to the problem is $Z = 31.5$, which occurs at $x_1 = x_2 = x_3 = 10.5$.

In matrix form, we can state the problem as follows:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 10.5 \\ 10.5 \\ 10.5 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

The solution X is called *feasible* if it satisfies $AX \leq b$, and *optimal* if there is no other feasible solution yielding a larger value of $Z = c^T X$.

Clearly, depending on the tightness of the constraints, there may be multiple feasible solutions. There may even be multiple feasible solutions that yield the same optimal objective function value (profit, for example). With fewer variables and constraints, it is quite often possible to generate a feasible solution simply by inspection, though it may not necessarily be optimal. For more complicated linear programs, or for computerized application, we need a generalized method.

1.1 Simplex Method

The most common method for solving LPs is the simplex method. In order to take advantage of it, the problem statement is converted into standard form [6]. Surplus and slack variables are added to the inequalities, changing them to strict equalities and adding the condition that $x_i \geq 0$; and Z is negated so that the goal is minimization. The definitions of *feasible* and *optimal* are appropriately reworded to reflect standard form. It is this standard form that our program expects as input.

Definition 2. ***Standard Form*** *Let A, c, b, and X be defined as before. We add variables $s_1, \ldots, s_m$ to produce $A' = [A|I_m], c' = [c|0\ldots], X' = [x|s]$ and define $Z' = [-Z|0\ldots]$ such that the problem can be stated as finding X such that we minimize $Z' = c'^T X$, subject to $A'X = b$ and $x_i \geq 0$ [6].*

It is convenient to represent A, b, and c as a consolidated tableau, for purposes of variable passing and data management. We define the tableau as:

$$\text{Tableau} \quad := \quad \left[\begin{array}{c|c} A & b \\ \hline c^T & 0 \end{array}\right] \tag{1.1}$$

Example 2. *Example 1, adjusted to standard form would be:*

$$\begin{aligned} \textit{minimize } Z = -x_1 - x_2 - x_3 \\ x_1 + s_1 = 10.5 \\ x_2 + s_2 = 10.5 \\ x_3 + s_3 = 10.5 \\ x_i \geq 0, s_i \geq 0 \end{aligned}$$

The corresponding tableau is:

$$\left[\begin{array}{cccccc|c} 1 & 0 & 0 & 1 & 0 & 0 & 10.5 \\ 0 & 1 & 0 & 0 & 1 & 0 & 10.5 \\ 0 & 0 & 1 & 0 & 0 & 1 & 10.5 \\ \hline -1 & -1 & -1 & 0 & 0 & 0 & 0 \end{array}\right]$$

Note the absence of explicit constraints for $x_i \geq 0$, as they are implied by standard form. Strictly speaking, any original x_i's which were unrestricted would require substitution of $x_i = x_i' - x_i''$ where $x' \geq 0, x'' \geq 0$ to ensure non-negative values for all variables. For simplicity, we are assuming the original variables all had implied non-negativity constraints. Also note the addition of slack variables s_i and the resulting identity submatrix in the tableau. This tableau is the input for the simplex algorithm.

The top-level of the simplex algorithm itself consists of three phases, described

here in pseudo-code: [6]

Algorithm 1.1.1: SIMPLEX($Tableau, nVars, nEqs$)

```
Unbounded ← false
Infeasible ← false
PHASE0(Tableau, nVars, nEqs)
if Infeasible
  then { output ("Program is infeasible after phase 0")
       { return
PHASE1(Tableau, nVars, nEqs)
if Infeasible
  then { output ("Program is infeasible after phase 1")
       { return
PHASE2(Tableau, nVars, nEqs)
if Unbounded
  then { output ("Program is unbounded after phase 2")
       { return
Z ← −Tableau[nEqs + 1, nVars + 1]
for j ← 1 to nVars
  do X[j] ← 0
for i ← 1 to nEqs
  do X[pivots[i]] = Tableau[i, nVars + 1]
return (X, Z)
```

The phases themselves are described as follows. For a more thorough discussion and sample coding of all three phases, see [6].

Phase 0: Find a basic solution, or show that the program is infeasible. This solution may not meet the added constraint $x_i \geq 0$.

Phase 1: Create a basic feasible solution from the basic solution, if necessary, or show that the program is infeasible. This solution meets all the constraints, but may not be optimal.

Phase 2: Improve the basic feasible solution (minimize Z) to get the final solution, or show that the program is unbounded.

The simplex algorithm as presented so far will yield a solution, if one exists, over the real numbers, $x_i \in \mathbb{R}$. From a purely numerical standpoint, it will be an optimal solution, though others may exist with the same Z for different values of X. Quite often, however, we desire an integer solution. In the simple factory case above, it would be difficult and not particularly useful to produce a fraction of

various products. In other applications, particularly combinatorial or graph theory applications, the problem itself may simply be discrete, and thus an integer solution to the model is appropriate. However, the simplex method alone does not provide a deterministic algorithm for finding integer solutions.

A linear program in which the variables X are also constrained to be integers is called an *integer linear program* (ILP). It is the focus of this report to present a parallel algorithm for solving ILPs.

1.2 Linear Relaxation

An obvious integer solution to Example 2 is $Z = -30$, occurring at $x_1 = x_2 = x_3 = 10$. We note that the function defining Z is a simple coordinate sum, and choose the largest integer coordinates that satisfy all constraints. This example is simple, however, and in practice, the integer solution is not always so obvious (or even necessarily near the real solution). Another obvious method would be to check all feasible integer combinations, but such an exhaustive search on the solution space would be impractical for larger problems.

To find integer solutions to a linear program in general, we can use the concept of linear relaxation. We fix some initial assignment $x_1, \ldots, x_i$ to be integers but "relax" the integer constraint and allow the remaining $x_{i+1} \ldots x_m$ to be any real value, and solve the remaining program. We can take advantage of the fact that any integer solution of the original problem is also a solution over the reals. In fact, for a given initial assignment, the best possible integer solution can be no better than the best possible real one. Furthermore, if during the initial phase of the search, we find an integer solution (i.e. where the remaining $x_{i+1}, \ldots, x_n$ happen to be found to be integers), we then have an upper bound on the optimal value of Z. Continuing in the search, we can ignore cases where the real solution has a Z value greater than our bound (recall that we are minimizing Z). We may further improve our bound with new integer solutions, and our hope is that this allows for significant pruning of the search.

Applying this method to Example 2, we would begin by setting $x_1 = 0$ (the minimum value for x_1), adjusting the tableau, and solving the remaining LP. This would yield the solution $(0, 10.5, 10.5)$ with $Z = -21$. We proceed by setting $x_1 = 1$ and repeating the process. At $x_1 = 10$, we have $Z = -31$ and at $x_1 = 11$ the problem becomes infeasible.

Note that we haven't found any integer solutions yet (all so far involve $x_2 = x_3 = 10.5$), and thus cannot prune the search. We start over with $x_1 = 0$, but also set $x_2 = 0$. We adjust the tableau and solve to find the solution $Z = -10.5$ at $(0, 0, 10.5)$. This round of linear relaxation, setting two variables at a time, continues until the problem yields $Z = -30.5$ at $(10, 10, 10.5)$.

The last round begins by setting all three variables to be integers, and ranging through the feasible values. Obviously, all solutions in this round will be integer ones,

but the process has resulted in an exhaustive search. We did not find any integer solutions until the last round of linear relaxation, and although our hope is to avoid this, it does also depend on the structure of the problem to some extent.

We present another example, showing how the tableau is affected by the linear relaxation:

Example 3. *Consider the following ILP, given first in equation form, then in our standard tableau form with slack variables:*

$$\begin{aligned} 4x - 5y &\leq 10 \\ x + 12y &\leq 48 \\ x, y &\geq 0 \\ \textit{maximize } & y \end{aligned}$$

$$\left[\begin{array}{cccc|c} 4 & -5 & 1 & 0 & 10 \\ 1 & 12 & 0 & 1 & 48 \\ \hline 0 & -1 & 0 & 0 & 0 \end{array}\right]$$

This can be easily visualized in two dimensions as intersecting lines. We want to minimize $-y$ (or, more intuitively, maximize y). Drawing this out is straightforward, and we can shade the region representing the inequalities so as to visualize the feasible solution space. Clearly, the optimal solution is $(x, y) = (0, 4)$, where the objective function takes on the value $Z = -y = -4$.

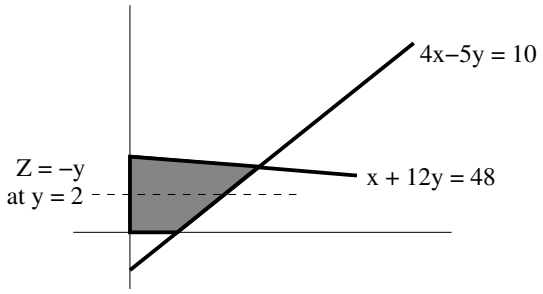

Figure 1.1: Graphical representation of Example 3

To generalize the process, we'll denote the variables as $x_1 = x, x_2 = y, x_3,$ and x_4, where the latter two are the slack variables. Applying our process of linear relaxation, we would start by setting $x_1 = 0$ and solving the remaining LP over the reals. The tableau with $x_1 = 0$ is simply:

$$\left[\begin{array}{ccc|c} -5 & 1 & 0 & 10 \\ 12 & 0 & 1 & 48 \\ \hline -1 & 0 & 0 & 0 \end{array}\right]$$

We apply the standard simplex algorithm to get $x_2 = 4$ and $Z = -4$. This happens to be an integer solution, and so we know that the final Z will be at most -4. Being an integer solution, we do not need to iterate on x_2 for $x_1 = 0$ either.

Continuing to iterate, setting $x_1 = 1$, we get this tableau:

$$\left[\begin{array}{ccc|c} -5 & 1 & 0 & 6 \\ 12 & 0 & 1 & 47 \\ \hline -1 & 0 & 0 & 0 \end{array}\right]$$

Applying the simplex algorithm, we get $x_2 \approx 3.92$ and $Z = -3.92$. This is not an integer solution, but also has a larger Z value, and thus we won't need to iterate on x_2 for $x_1 = 1$. Without the pruning, and only the two variables, that would have resulted in an exhaustive search.

We now continue with $x_1 = 2, 3, \ldots$ until the LP with tableau

$$\left[\begin{array}{ccc|c} -5 & 1 & 0 & 10 - 4x_1 \\ 12 & 0 & 1 & 48 - x_1 \\ \hline -1 & 0 & 0 & 0 \end{array}\right]$$

becomes infeasible. It is infeasible when $x_1 = 7$. For each iterative choice for x_1, there were no more integer solutions, and each successive value of Z became larger. Had we not stumbled on an integer solution at $x_1 = 0$, and thus set an upper bound on Z, the process would continue by iterating on x_2 for each $x_1 = 0, 1, 2, \ldots, 6$.

Of course, it is also possible that an ILP has no solution because it is either unbounded or infeasible over the reals, but that can be found out by finding the solution over the reals first. In our parallelization method, that initial run over the reals is also necessary for other reasons, as we will see later in Section 3.4.

Finally, the "worst" scenario for an integer linear program would be one where there is a real solution, but no integer solutions. In this case, the search will not be pruned and we end up with an exhaustive search. In a majority of applications, however, this is probably the least likely case, while it may be more common in theoretical problems.

Linear relaxation can be described programmatically as a recursive process:

Algorithm 1.2.1: LINEARRELAXATION($Tableau, nVars, nEqs, X, nFixed$)

```
comment: status variables
Done ← false
Found ← false
Prune ← false
comment: set up the new tableau
for i ← nFixed + 2 to nVars
  do { for j ← 1 to nEqs + 1
       do newTableau[i − 1][j] ← Tableau[i][j]
comment: substitute for the new fixed variable
if nFixed = nVars
  then Done ← true
nFixed ← nFixed + 1
X[nFixed] ← 0
while Prune = false and Done = false
      { newTableau ← SUBTABLEAU(Tableau, nVars, nEqs, X, nFixed)
        (X, Z) ← SIMPLEX(newTableau, nVars − 1, nEqs)
        if Infeasible
          then Prune ← true
          else if Unbounded
               then { Done ← 1
                    { return ( false )
          else if INTEGERSOLUTION(X)
               then { comment: check for improvement
                    { if !Found or (Z < Z_B)
                      then { Z_B = Z
                           { X_B = X
                           { Prune ← true
                           { Found ← true
        LINEARRELAXATION(Tableau, nVars, nEqs, X, nFixed)
        X[nFixed] ← X[nFixed] + 1
return (Found)
```

Chapter 2

Existing Methods

Much of the prior work on such algorithms occurred in the late 1980s through the late 1990s, with much general parallelization research going back to the 1970s. It is unclear whether the speed improvements in hardware, the massive parallelization possible with today's computer systems, or simply that current algorithms are adequate to solving today's ILPs led to the gap in research since the late 1990s.

Many papers reference [7], which covers the broader topic of Integer and Combinatorial Optimization in general. Chapter II.4, section 2 outlines the basic approach we take here, and defines some theoretical optimizations for breaking up problems and choosing branching variables, such as "degradations" and "penalties". We took a different approach to breaking up the work, while following later, less computationally-intensive methods for branching.

Also in 1988, [3], a parallel solution that achieved super-linear efficiency on several test problems using a hybrid branch-and-bound and cutting-plane method was presented. At each node of the tree, the problem was split into two sub-problems and offered back in sort of a queue. It is not clear to us if there is any optimization in the choice of branching variables, or specifically how a node is "fathomed". In this model, there is a notion of a master process in the sense that one processor performs certain setup and pre-calculation, while the rest of the algorithm depends on set of shared resources, termed the "monitor". Our algorithm is similar, but incorporates a dedicated master process which maintains control over the work queue and assimilates the results from the client processors. We similarly left out potential optimizations of variable choice and branching priorities initially, in an effort to achieve simple proof-of-concept and implementation. Many suggestions for improving our approach are listed in Section 6.

The 1993 survey paper [5] provides a high-level overview of the concepts involved in branch-and-bound parallelism in general, of which the ILP is a subset. Our code falls into the "Parallelism of type 2" classification, and is a "Strategy on request" "Asynchronous Single Pool" design, in the terminology of this survey. We have a master process, but the clients build the branch-and-bound tree, adding work while iterating at each node. The master may later remove work as obsolete, which is how

we hope to achieve super-linear improvements in performance.

In 1997, [1] builds on the general algorithm in [7] and largely follows the tree structure and hybrid method of [3]. The authors incorporate the use of "penalties" from [7] in order to better optimize the choice of branching variable, but agrees with that source that many commercial solvers have abandoned such calculations as they have proven to be computationally-intensive for larger problems.

Finally, much of the structure and methodology of our algorithm follows closely to that presented in [6] from 2005. In particular, our method incorporates Bland's Anti-cycling Rule to avoid cycling while converging on a solution. This rule is applicable to the general simplex algorithm, and is not particular to ILPs or linear relaxation.

Chapter 3

Serial Implementation

The serial implementation is essentially a branch-and-bound search on the feasible solutions over the integers. We hope that the search can be pruned by taking advantage of the previously-mentioned concepts within the simplex algorithm and linear relaxation.

3.1 Tableau

We took advantage of indexing conventions in the C programming language, whereby an array of length n is referenced by indices $[0, \ldots, n-1]$, to allow the use of a single tableau variable for all phases of the simplex algorithm[6]. The description given in Figure 1.1 forms the bulk of a simple doubly-indexed array, but with the upper left entry from A in cell $(1, 1)$ instead of $(0, 0)$. Row 0 and column 0 are used in Phase 1 to store the additional costing variables and optimal value function. There is somewhat of a trade-off between simplicity in function calls by passing a single tableau variable versus complexity in indexing during matrix operations due to the dual-use of the array.

3.2 Solution Tree

The solution space for the search can be viewed as a tree with levels $l = 0 \ldots m$ [7]. At each level $l \geq 1$, we create a node for each feasible value of x_l, given the values that have already been set higher in the tree for $x_1 \ldots x_{l-1}$. A sample tree is given as Figure 3.1. The top node, $l = 0$, has no real meaning in this model as far as assigning any value to any x_i.

At each node, we use linear relaxation with $x_1 \ldots x_l$ as integers and solve for the remaining $x_{l+1} \ldots x_n$ using the simplex method. There are four possible results, and we employ a decision tree with backtracking:

1. If the program is unbounded at any point, then the entire original problem is

unbounded and we are done. We have chosen integers for X_0 and because the program is unbounded at this stage, can choose any integers we wish for some particular $x_i, i > l$.

2. If the program is infeasible, we can prune the search below that node, as no feasible solutions exist for that particular value of x_l. We backtrack and continue the search on a different branch of the tree. [7]

3. At this point, the program has a solution, so we check to see if it happens to be integer. If so, we have a bound on Z. If this is an improvement on an existing bound (or if we have no existing bound), we keep the value.

   Regardless, for an integer solution, we also prune the search. Below this node, we know the best solution and we know it to be integer.

4. If the solution is not integer, we simply add nodes below for values of x_{l+1} as long as $l < n$, and continue searching the tree.

5. Once we have exhausted all feasible solutions ($l = n$), we backtrack up to the nearest level with unsolved nodes.

We also refer to these nodes as "cases", which abstracts it away from a tree and makes the transition to an optimized parallel algorithm a bit more intuitive later. We refer to the level of relaxation in a case by indicating the number of variables explicitly set to be integers, as in a "2-variable case" or "5-variable case".

There are a few aspects of this serial algorithm worth covering in greater detail.

3.3 Queuing

We used queuing rather than a true tree structure in the serial implementation, because it simplified the later conversion to parallel. The queue starts out with a single node, and the algorithm pushes additional nodes onto it as more feasible cases are found. The algorithm terminates when the queue is empty, and if we have an integer solution at that point, it is the optimal one.

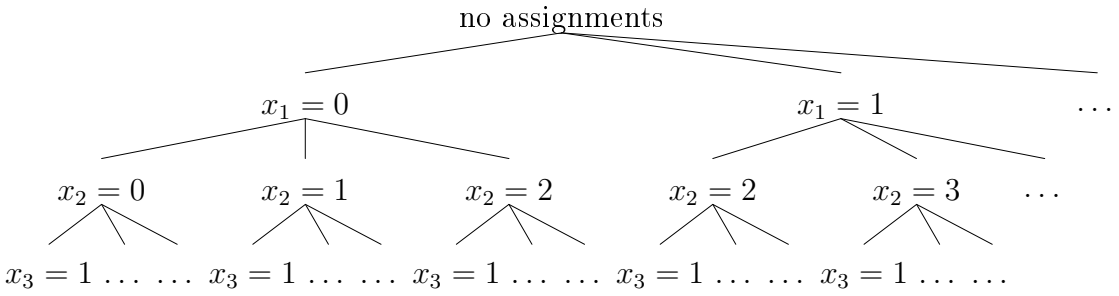

Figure 3.1: Example solution tree

The net effect of the queuing is that the overall search is breadth-first, while each case generates more cases one level deeper. Our claim is that this allows for faster pruning of the search by providing a broader distribution of work early in the search. Quite often, fixing relatively few variables as integers in a linear program will result in the entire solution being integer[7].

Finally, the queuing model allowed for simpler memory management in the actual code. Dynamic allocation of queue nodes for cases and a linked-list for the queue itself obviate knowing the size of the tree in advance.

The description given in Algorithm 1.2.1 is a depth-first search. Algorithm 3.3.1 shows the modified pseudo-code for linear relaxation using a queue, thereby changing the process to breadth-first. The GetCase() and PutCase() functions simply take data from the queue (a "pop") or place data on the queue (a "push"), and the queue itself uses a linked-list to model a first-in-first-out (FIFO) stack. The actual queue management code can be found in the appendix.

Algorithm 3.3.1: QUEUEDLINEARRELAXATION($Tableau, nVars, nEqs, Queue$)

```
Done ← false
Prune ← false
while (nFixed, X, Z_case) ← GETCASE(Queue)
    do { comment: don't bother unless there is possible improvement
         if Found and (Z_case ≥ Z_found)
           then Prune ← true
         nFixed ← nFixed + 1
         X[nFixed] ← 0
         while Prune = false and Done = false
             do { newTableau ← SUBTABLEAU(Tableau, nVars, nEqs, X, nFixed)
                  (X, Z) ← SIMPLEX(newTableau, nVars − 1, nEqs)
                  if Infeasible
                    then Prune ← true
                  comment: Infeasible here means look no further on this branch

                    else if Unbounded
                    then { Done ← 1
                           return
                  comment: Unbounded anywhere is unbounded, period. We are done.

                    else if INTEGERSOLUTION(X)
                    then { comment: check for improvement
                           if !Found or (Z < Z_best)
                             then { Z_best = Z
                                    X_best = X
                                    Prune ← true
                                    Found ← true

                    else if nFixed ≠ nVars
                    then PUTCASE(Queue, nFixed, X, Z)
                  comment: We have more potential, queue for later
                  X[nFixed] ← X[nFixed] + 1
return
```

3.4 Feasibility

At each level of the tree, we enumerate feasible values of x_l – that is, values of x_l for which there exists a solution to the relaxed problem having fixed integer values for $x_1, \ldots, x_l$ and real values for $x_{l+1}, \ldots, x_n$. This is not as obvious as it sounds, for we cannot just start at $x_l = 0$ and stop when we reach our first infeasible result. It's quite possible that the feasible range for x_l is something like $2 \leq x_l \leq 10$, for example.

To get around this, we again take advantage of the simplex solution over the reals. At the previous level $l - 1$, we solved for the remaining variables, including x_l, over the reals. Let x_{l_R} represent the optimum real value of x_l for a given case. To find the possible range of integers for assignment to x_l, then, we split the work into two ranges: upward from $\lceil x_{l_R} \rceil$ and downward from $\lfloor x_{l_R} \rfloor$. This is done to more efficiently limit the range of each variable: had we started at $x_i = 0$, we would need to iterate upward until the LP first becomes feasible, then continue until it is not. We continue trying each integer value until the solution on the remaining variables is infeasible. In practice, one of the two cases generated is often infeasible from the start because the optimal real value is near or at a boundary constraint.

3.5 Pruning the Search

Pruning of the search depends on finding an integer solution. It is quite possible that the search for a given program might not prune until very late, or indeed, not at all (no integer solution). In the initial phase, before any bounds have been found, we do get values for the real solution. Those values are another bound, but in this serial implementation, are not used. It may be possible to take advantage of them to a limited extent for imposing a priority on the queue.

Again viewing the solution space as a tree, where level l of the tree consists of nodes corresponding to all feasible values of x_l for the set values of $x_1 \ldots x_{l-1}$, pruning is done in the literal arboreal sense. For example, consider the sample tree given in Figure 3.1, and suppose that setting $x_1 = 0, x_2 = 2$ makes the problem infeasible. Obviously, there is no point in continuing down that branch. The search is pruned at that node, cutting off the tree below that point, yielding Figure 3.2. The search would then continue with the node $x_1 = 1$.

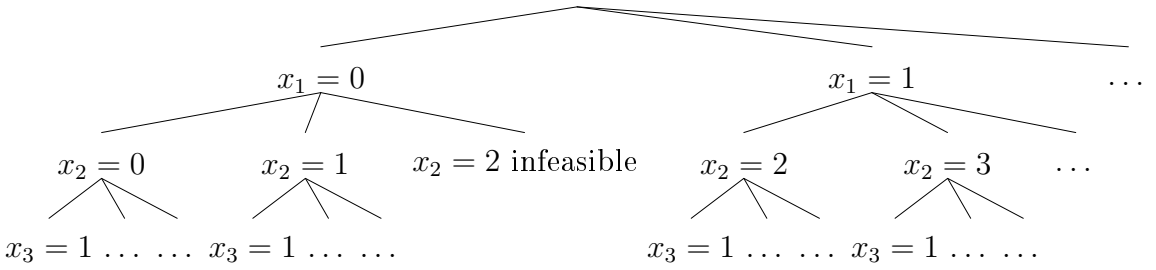

Figure 3.2: Pruned solution tree

With the use of a queue to store cases for later evaluation, pruning the tree for infeasibility amounts to not pushing the case under current evaluation onto the queue.

Further pruning can happen if we encounter an integer solution. That solution provides an upper bound on Z, and we further prune branches of the tree that have Z values which cannot improve on that bound. With the queue structure in this situation, a case is pulled off, found to have no improvement over the known Z bound, and simply dropped.

Chapter 4

Parallel Implementation

In the parallel implementation, we break down the program structurally to allow the use of multiple processors. Again, the intention is to speed up pruning of the search. At the very least, we should achieve a linear improvement in the speed of the search itself. The appropriate method for a given problem depends on the quantity of the data, the type and complexity of the algorithm, and the amount of bandwidth required for data access and communication.

4.1 Parallel Methods

Parallel algorithms take advantage of problem structure in order to speed up processing by spreading the work over multiple compute nodes. There are several general methods to accomplish this, covered briefly here. All methods reduce to effectively breaking up the dataset to be considered in some fashion (termed "grain size"), and establishing an appropriate communication and control system between processing nodes (the "topology") [4].

One simple form is to manually break up the problem. This may be a matter of giving a subset of data to be processed to each of several compute nodes, then comparing the results. Testing and analysis problems may fall into this category, and the unmodified serial algorithm is employed in parallel on each dataset. This method has a large grain size and the advantage of not having to develop a specialized parallel algorithm, but only affords a linear improvement on speed. There is no communication necessary between compute nodes in this model.

Shared memory parallelization often relies on numerical methods to perform large calculations in parallel. Part of the algorithm itself breaks down the model or data, rather than having to do it manually, and all processing nodes access the single dataset directly [4]. This method is generally limited to specialized hardware, often a single physical machine with several CPUs and an larger amount of memory. This method is by far the fastest, because memory access is at local bus speeds, and because of the generally small-grain breakdown of data. However, it is limited in scalability

by the physical specifications of the machine, in particular the bandwidth available for inter-node communication. Larger systems may take advantage of specialized, or even commodity, network connections in order to extend the parallelization to other physical machines. Examples of programming libraries and compilers for this model include MPI (Message Passing Interface) and UPC (Uniform Parallel C).

A recent example of specialized hardware for shared-memory applications is the GPU – Graphics Processing Unit. Once simply dedicated to running a computer's graphical display, modern GPUs are designed with broader computation in mind. Units with upwards of 400 processor nodes are not uncommon, and there is work toward standard programming interfaces underway. The processing nodes are simple, and very specifically designed for small-grain algorithms.

Distributed memory parallelization lies at the other end of the spectrum, where each processing node has its own dedicated exclusive memory, and runs a copy of the algorithm. The individual nodes can either negotiate among themselves, as in the shared-memory model, or there may be a dedicated "master" node handing out work sets to other nodes for actual processing. A client-server algorithm is an example of the latter, and offers somewhat of a middle-ground when the problem space has obvious discrete blocks of computational work. A master server distributes pieces of work at the request of several clients, giving the advantage of scale over several physical machines. The limiting factor here is the speed of communication between the master and each client, while the scalability is only limited by the how many clients the master can track. MPI is also applicable in this model, as it allows generic communication between compute nodes, independently from any shared hardware resources such as memory.

4.2 Client-Server

In our problem, we have a tree structure in which each node in the tree requires running the simplex algorithm on a matrix and comparing the results with those of other nodes (a decidedly large grain size). The amount of data required to describe a case is minimal (simple topology) and together with the initial tableau describes an independent chunk of computation work, thus the client-server model suits it very well. MPI may be a good choice for communication, but it is simpler to implement the first version with a straightforward text-based network command protocol and a single master node. In addition, given the speed of the simplex algorithm on large LPs, we expect a majority of algorithm time to be taken in the computation rather than the communication.

In our algorithm, the master initially distributes copies of the original data, and then hands out cases, accepts the results and handles potential search pruning, while the clients run the simplex algorithm. The only significant computation the master does is for the initial case, to find out if the problem is infeasible or unbounded, or else determine the optimal real value of x_1.

This model has been successfully used in many high-profile community-processing efforts, including Seti@Home[2] and Folding@Home[8]. In our code, the network communication consists of simple text-based protocol. The master process is started first, and given a file name and number of clients to expect. The file contains the matrix description of the linear program. As each client connects to the master, it is immediately given a copy of that file, and then waits to be issued a case. The rest of the process is controlled by a simple protocol using the command set in Table 4.1.

Table 4.1: Client-Server Protocol

Command	Issued by	Definition	Response(s)
request	client	Client requests a case from the master for processing	Master responds with 'case', 'done', or 'wait'
case nFixed X_{l_R} Z_{l_R}	master	Master gives a case to client	Client processes case and returns results with 'results'
done	master	All cases have been handled	Client disconnects and exits
wait	master	Master tells client to wait a predetermined amount of time	Client waits a predetermined time, then repeats the original request
bye	client	Client announces intention to disconnect	Master removes client from list and redistributes work as appropriate
results	client	Client announces results of processing	Master queues, prunes, or stops processing as appropriate

4.3 Traversal of the Solution Space

The parallel implementation, when combined with the queue structure for the solution tree, produces a traversal that is really neither depth-first, nor breadth-first. Consider the depth-first labeling of the tree in Figure 4.1.

A depth-first search would traverse it in lexicographical order, from node A to node V. The queue structure in the serial implementation, however, effectively produces a breadth-first search, beginning with A, N, and then B, F, J, ...

In the parallel implementation, each client works at one node evaluating feasible cases to create nodes below it, just as in the serial version. Depending on the feasibility

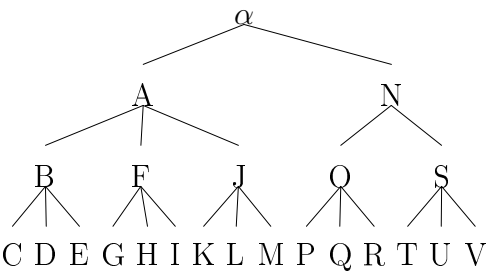

Figure 4.1: Labeled solution tree

of each node and the time it takes to solve each case, however, the client that put a case on the queue might very well not be the one that ends up checking it later. Each client will traverse in a somewhat random pattern, but since all clients use the same work queue, all feasible cases are eventually handled. For example, two clients running through the tree in Figure 4.1 might evaluate cases as follows, where Q designates a possible work queue at various points:

1. $Q = [\alpha]$: CPU1 gets the single initial case, iterates on x_1, and produces nodes A and N ($Q = [AN]$). By the time CPU1 is done iterating, CPU2 has already started on A, so CPU1 continues at N, iterating on x_2, and produces O and S.

2. Meanwhile, CPU2 started out idle, waiting for work. Once the queue started to fill, it was assigned node A, and produced nodes B, F, and J. Combined with the work CPU1 is producing, the queue could be just about any interleaving of B, F, J, and O, S, such as $Q = [BOFSJ]$.

3. Whichever client completes its run on x_2 first would get node B, iterate on x_3, and produce C, D, and E. The other client might get node O (as in the possible Q given above), and the resulting queue could be something like $Q = [FSJCDPQER]$.

4. ...and so on, with each client pulling the next available piece of work from the queue, and the results of all clients being pushed onto the queue in the order of generation.

Note that in step 3, it is possible that node O might come up for processing before nodes F and J, thus the search is not exactly breadth-first. Nodes F and J will get handled in the order in which they were queued, provided no improved bound on Z has been found in the meantime. If such a bound has been found, any node having a higher optimal Z value over the reals would be pruned when it comes up for consideration, saving processing time. It is worth mentioning in particular that this potential bound might be found by any one of the clients, but affects all clients from

that point onward. The master keeps track of these bounds and integer solutions, and checks future cases against them when handing out work to its clients.

4.4 Defining "Done"

One concern with the combination of queued cases to represent a solution space, and client-server communication to distribute work, is how to know when the entire algorithm is "done". Obviously, this happens when there is no more work to be processed, however, that is not necessarily equivalent to having an empty queue. Rather, we are done when the queue is empty and all work distributed to clients has been completed and reported. This is redundant in the serial model, since there is essentially one client, but important in the parallel one.

Initially, the queue only has one case, and all clients request work. As we saw in the description in section 4.3, only one client gets anything to process, and the other clients must wait, rather than exit – there will likely be plenty of work once we know the feasible range of x_1. We accomplish this by keeping track of the case we have assigned to each client in a separate array, and are "done" when the main work queue and client work array are empty. We also gain some robustness in the ability to reassign cases, should a client disconnect without reporting its results.

A secondary issue with completion concerns the simplex algorithm itself. Under certain circumstances, the simplex algorithm may cycle – that is, it does not terminate on a solution, but rather continues in an endless cycle. One common method to counter this is to apply Bland's Rule[6], whereby pivots are chosen by specific criteria during phases II and III, guaranteeing that the algorithm will not cycle.

Chapter 5

Empirical Analysis and Conclusions

In the course of this project, several hurdles were encountered in stages. In early runs, it became clear that zero-one matrices, such as those found in many graph theory problems, did not lend themselves to the work breakdown used here. Specifically, each variable did not have much of a feasible range, and a feasible case would generate only one or two ($x_{i+1} \in \{0, 1\}$) cases for further investigation. Also, these problems seemed to exhibit a tendency to arrive at an integer solution very early and with little or no room for improvement, and not actually test out the branching and parallelization of the solver.

Finding example problems proved to be much more difficult than expected. There are a few example problems and problem sets mentioned in the references [1, 3], but the format necessitated a conversion utility in order to prepare the LP for our solver, and a code change to the solver itself in order to better handle slack variables. Furthermore, the sample problems were given as general LPs, and as such did not list an integer solution (or indeed, if one even existed). On one run of such an LP, the work queue consumed all available memory over the span of two days, and crashed the master node – without so much as a token integer solution. The best alternative was to contrive examples with verifiable integer solutions, such as the cube, box, and house problems discussed below.

Motivated by the long run-times of the older sample problems (where it is not clear an integer solution even exists), we went ahead and implemented some simple queuing and relaxing optimization ideas, but found them not as helpful as expected. More are mentioned in section 6. The results were surprising at first, but obvious upon further inspection. One may expect that placing an ordering on the queue, such as by bound, would improve solution time. In practice, though, this resulted in a depth-first search. While solving a relaxed case with fixed variables $x_1 \ldots x_3$, for instance, there are likely several feasible values for x_4 that have very similar bounds. Queuing these cases based on those bounds would result in the master handing out nodes farther down this branch of the solution tree, rather than more broadly across it. Along with this behavior, the simple linked-list structure employed for the queue quickly became cumbersome while adding cases for future consideration, due to the

linear nature of traversal while comparing bounds. An alternate data structure could avoid this, and allow further evaluation of queue prioritization. An index into the queue based on the sorting criteria would preclude a full linear search, but a binary tree might be a better choice to allow faster insertions.

On the linear relaxation side, it seemed likely that different column orderings of the tableau might affect solution time. We wrote a pre-processor that looks at three criteria: affect on Z (smallest non-zero c_i), number of applicable constraints (rows of A where $A_{ij} \neq 0$), and level of constraint (smallest ratio of A_{ij} to b_i for a given i). The rationale for the first is to maximize the effect on Z in hope that it will lead to early pruning. For the latter two, we hope to look first at variables with smaller feasible ranges (or more constraints), and get a broad view of possible Z values early in the run.

Finally, the input format itself was modified slightly to allow the specification of which x_i must be integers. This was done to allow distinction of actual program variables from slack variables (which are not generally subject to the integer requirement), but has the added benefit of making the solver able to handle mixed integer linear programs (MILPs) as well as ILPs.

5.1 Tabulation of Results

In each example problem, we optimized the column order several ways for comparison of various methods. The notations under the "PreP" column, such as "zcr", indicate the sorting method applied to the tableau columns, as discussed earlier, where "z" sorts by the largest effect a given x_i has on Z, "c" sorts by the number of constraints in which x_i appears, and r sorts by the smallest ratio of A_{ij}/b_i for constraints in which x_i appears. Thus, the notation "zcr" means the tableau columns were sorted first by z, then by c, then by r. Similarly, a minus sign in front of any sort letter indicates the sort was reversed for that criterion.

We ran the example problems under varying number of processors, from 1 to 8, to gauge the effect of parallelization. As a check for possible variation in run-time, we ran versions (CPUs and pre-processing) of a few cases several times and found the performance results very consistent. The system used for the runs was a quad-core Intel i7 920 running at 2.67GHz, and having 6GB of main memory. The quad-core CPU had hyperthreading enabled, making it appear as though it were an 8-core CPU with (at least for our use) corresponding performance. Both master and client processes were run on the same machine, due to the relatively small CPU-load of the master and low memory-usage of the clients.

The metrics tracked and reported are number of clients (Cl), number of cases queued (Qd), number of cases offered to clients (Of), number rejected (Rj), number pruned (Pr), and elapsed time (Et). As aggregate measurements of efficiency, we define algorithm time (At) to be $Cl * Et$, and rate of cases per second (Rt) to be Qd/Et. We expect linearity for Et and Rt, proportional to the number of processors.

We expect a relatively constant value for Rt/Cl for a given LP, dependent on the size of the tableau (and thus, number of pivots). The cases presented here have small tableaus, as the parallel algorithm employed is more greatly affected by the tree size rather than the number of pivots.

5.2 Case 1: Cube

This problem is the tableau given as Example 2. Due to the symmetry in this problem, none of the pre-processing sorts have any effect. Results are summarized in Table 5.1.

Of note, there were no cases rejected until 4 client processors were employed; and the improvement in solving rate (Rt) was not linear. This is a relatively small LP, with a small solution space. By changing the right-hand side of the constraints to be 100.5 instead of 10.5, we should see longer solution times and better linearity. Results are shown in Table 5.2, and indeed, the solving rate is quite linear. However, the time-to-solution (At) from one to two processors was super-linear, and did not improve much with the addition of more processors. This super-linearity is a by-product of the queuing, where some 3-variable cases were considered before finishing all 2-variable ones, and is what we hope will happen to improve solution time.

As mentioned above, the symmetry of this problem defeats two of the pre-processor's sorting criteria. If we now further modify the LP to have differing right-hand side values, we can test the ratio sorting, "r". The reverse sort is denoted with a "-r" in the table. Each variable still only appears in a single constraint, and all variables have the same coefficient in Z, so the "z" and "c" sortings have no effect. Visually, we are simply changing our cube to a box whose dimensions are 100.5, 200.5, and 300.5. Results are given in Table 5.3, and we see sorting by the minimum ratio generally improved solution time and resulted in fewer queued cases. This makes sense, as the use of linear relaxation on this LP results in an exhaustive search of the solution space. If we start with the variable having the largest feasible range (i.e. the longest side of the box), and work backwards to the variable with the smallest range (and recall that cases are split into up- and down-ranges), we end up considering at least $2 * 300 * 200 = 120000$ cases (in a single-processor run). If, as with the "r" sorting, we start with the variable having the smallest range, we end up considering just over $2 * 100 * 200 = 40000$ cases.

To test the "z" sort, we revert to the plain larger cube, and modify the coefficients in Z. These results are summarized in Table 5.4, but most of the performance differences resulted from the algorithm iterating through a list of integer solutions in the case of "-z", due to the relatively small change induced in the value of Z by the x_i having the smallest coefficient.

Table 5.1: 10.5 unit cube results

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
none	1	266	26	0	240	6.36	6.36	41.7
	2	266	26	0	240	4.68	9.36	56.6
	4	232	26	17	205	3.17	12.7	72.9
	8	226	30	20	196	2.58	20.6	87.2

Table 5.2: 100.5 unit cube results

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
none	1	20606	206	0	20400	420	420	49.1
	2	5306	56	75	5250	55.7	111	95.2
	4	4916	56	270	4860	26.9	108	183
	8	4906	59	275	4847	14.4	115	341

Table 5.3: 100.5 x 200.5 x 300.5 unit box results

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
r	1	40806	206	0	40600	825	825	49.5
	2	10508	56	74	10452	108	215	97.6
	4	9716	56	470	9660	50.9	204	191
	8	9702	57	477	9645	26.5	212	366
-r	1	121606	606	0	121000	2456	2456	49.5
	2	11314	58	72	11256	116	232	97.7
	4	10304	56	376	10248	53.8	215	191
	8	9718	58	669	9660	26.5	212	367

Table 5.4: Minimize $Z = -100x_1 - 10x_2 - x_3$ over 100.5 unit cube

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
z	1	20606	206	0	20400	420.4	420.4	49.0
	2	5308	56	74	5252	55.7	111.4	95.3
	4	4916	61	270	4855	27.0	108.0	182.0
	8	4904	58	276	4846	14.4	115.5	339.5
-z	1	20606	504	0	20102	434.1	434.1	47.5
	2	5390	412	2962	4978	64.6	129.3	83.4
	4	5004	412	3155	4592	31.4	125.6	159.4
	8	4994	412	3160	4582	16.7	133.6	299.0

5.3 Case 2: House

As a slightly more complicated test case, we produced the "house". Again, able to be visualized in three dimensions where the familiar (x, y, z) correspond to (x_1, x_2, x_3) in the LP, this figure resembles a house in the $x - y$ plane, and has a slanted face. We have chosen two functions for Z. In the first LP, designated "House 1", the real solution is at the house's peak $(4, 9.5, 1.5)$ with $Z = -9.5$, while the integer solution is just below it at $(4, 9, 2)$ with $Z = -9$. These results are seen in Table 5.5. The second LP, designated "House 2", has a real solution at the top of the right wall $(7.5, 8.1875, 0)$ with $Z = -88.75$, while the integer solution is still near the peak at $(5, 9, 0)$ with $Z = -87.5$. These results are given in Table 5.6. This is a simple example of where the integer solution is not intuitively near the real solution.

Example 4. *Tableau for House 1:*

$$T = \left[\begin{array}{rrrrrrrr|r} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 7.5 \\ -3 & 8 & 0 & 0 & 1 & 0 & 0 & 0 & 64 \\ 3 & 8 & 0 & 0 & 0 & 1 & 0 & 0 & 88 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 11 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & .5 \\ \hline 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Example 5. *For House 2, we change Z to be nearly parallel to one of the constraints:*

$$T = \left[\begin{array}{rrrrrrrr|r} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 7.5 \\ -3 & 8 & 0 & 0 & 1 & 0 & 0 & 0 & 64 \\ 3 & 8 & 0 & 0 & 0 & 1 & 0 & 0 & 88 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 11 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & .5 \\ \hline -3.1 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Table 5.5: House 1 results

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
crz	1	16	8	0	8	0.6	0.6	24.8
	2	16	8	0	8	1.3	2.6	12.1
	4	16	8	0	8	1.3	5.3	12.1
	8	16	8	0	8	1.3	10.6	12.1
czr	1	22	4	0	18	0.6	0.6	36.6
	2	22	4	0	18	1.0	2.1	21.1
	4	22	4	0	18	1.0	4.2	21.2
	8	22	4	0	18	1.0	8.3	21.2

Table 5.6: House 2 results

PreP	Cl	Qd	Of	Rj	Pr	Et	At	Rt
crz	1	16	10	0	6	0.8	0.8	20.9
	2	16	10	0	6	1.0	2.1	15.4
	4	16	10	0	6	1.0	4.2	15.3
	8	16	10	0	6	1.0	8.3	15.4
zcr	1	22	4	0	18	0.6	0.6	36.6
	2	12	4	5	8	1.2	2.4	10.0
	4	10	4	6	6	1.2	4.8	8.3
	8	10	4	6	6	1.2	9.6	8.3

5.4 Depth-first Enhancement

The final enhancement considered for this solver was an attempt to quickly find an integer solution that may be near the real one. Any such solution will allow pruning of the tree, and can only help improve solution times.

We call this method a drill, and it is essentially a binary depth-first search down either side of the real solution. It is run first and only once, by injecting a special case into the queue as the first case. The rest of the queued work then follows as described previously. It is binary in the sense that at each level l it follows two paths to the next level down, setting $x_l = \lceil x_{l_R} \rceil$ and $x_l = \lfloor x_{l_R} \rfloor$. It only generates bounds based in integer solutions (if found), and does not create further cases for queuing. Once the client processor running the drill has finished, it joins the other clients working on the rest of the queue as usual.

Using the house in Example 4, the solver would consider only $x_1 = 3$ and $x_1 = 4$ at the first level. At the next level for $x_1 = 4$, it would only look at $x_2 = 9$ and $x_2 = 10$, and so on, until all integer variables have been fixed, or no solution is found. If an integer bound is found, the master will be able to prune the queue for subsequent processing.

In Figure 5.1, we show this process graphically. The nodes in bold indicate the actual relaxed solution found. For instance, at $x_1 = 3$ and $x_2 = 9$, the algorithm arrives at $x_3 = 2$ without having to explicitly set x_3 to be an integer. The nodes in parentheses are shown for completeness to indicate where a non-integer solution exists, but are not visited by the drill because they are not integer-valued. Nodes which are struck-through are infeasible.

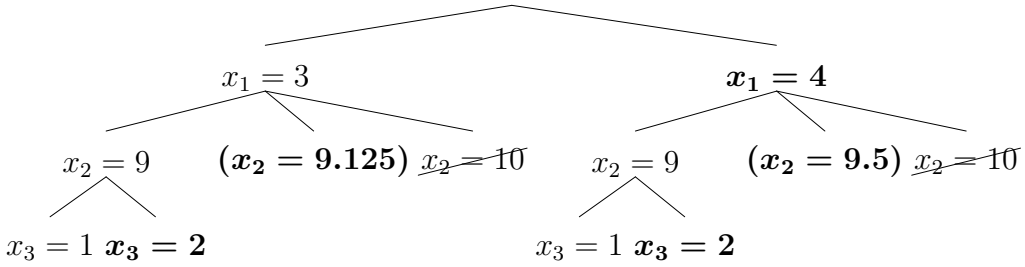

Figure 5.1: Nodes checked by drill

The drill would examine every integer node in the tree in Figure 5.1, and would find the integer solutions at $(3, 9, 1)$, $(3, 9, 2)$, $(4, 9, 1)$, and $(4, 9, 2)$. All have $Z = -9$, which sets a nice upper bound for minimizing Z. In fact, all of these solutions *are* optimal integer solutions for this particular ILP, and the subsequent algorithm will not improve on them, but will instead prune most of the tree.

In many LPs, this method actually finds the optimal integer solution because it happens to be near the optimal real solution. In the case of the house in Example 5, it does not, because the integer solution has $x_1 = 5$, while the drill process would

only consider $x_1 = 7$ and $x_1 = 8$. However, the drill does find an integer solution at $(7, 8, 0)$ with $Z = -8$, which provides a fairly tight bound for the final solution. Thus, any integer solution will improve the solver time by providing a bound for pruning early in the run.

5.5 Conclusions

Without a large number of example ILPs, it is difficult to draw many solid conclusions about the efficiency or effectiveness of the algorithm in this approach. Out of the six examples created and analyzed, most showed super-linear improvements in solution time going from one to two processors. However, the examples were all 3-dimensional, and this may be due to timing alone. Any 3-variable case (one where 3 variables are being set to integer values) queued will result in an integer solution, and thus, an upper bound on Z. If such a case is queued prior to other 2-variable cases, and produces a relatively tight bound on Z, this could drastically reduce solution time.

The pre-processing options showed some promise, with the greatest difference seen in Table 5.3. Again, however, this may have been a by-product of the simple 3-dimensional examples. The pre-processing for that example was type "r", and essentially caused an exhaustive search to iterate through variables with narrower feasible ranges first.

The drill concept showed the most promise in achieving reliably super-linear performance overall. It does not take long to run, because of its limited scope, and any resulting bound may drastically increase pruning and reduce solution time. Furthermore, it may be that such a search often discovers the optimal integer solution, if it commonly lies near the real solution. In this sense, the advantage of a parallel algorithm over a serial one would at best boil down to the ability to consider more cases per second – a linear improvement in the final search.

Chapter 6

Future Projects

Several ideas came to mind as the example code was produced, requiring what could be significant modification to large amounts of the code, and not directly in line with the purpose of this report. They are listed here as possibilities for future work and development of the parallel algorithm.

6.1 Improving Linear Relaxation Methodology

The concepts behind linear relaxation are straightforward; however, in practice, there are some caveats from an efficiency standpoint. At each level of the tree, we need to use a submatrix A_l of A as well as adjust b for the fixed values X_0. The process behind the simplex algorithm is destructive – it alters the entries in the tableau while solving – thus we need to preserve A somehow. The obvious method is to copy A and make the changes, but this becomes quite inefficient for programs with large numbers of variables and/or constraints. The vast majority of the copied data hasn't changed since the last copy. The time spent copying may rise proportionally to the time spent solving, due to the matrix manipulation routines, but overall there is room for improvement in setting up each case. A careful transition from one case to another, searching depth-first, may yield a solver routine that references the data that has changed separately, while still preserving the original matrix in its entirety. Essentially, this pushes the functionality of SubTableau() down into the simplex algorithm itself and trades inefficiency for some additional complexity.

6.2 Drill Often

Our implementation of the drill was a single run at the beginning of the work queue. We noted that in Example 5 the integer solution was not near enough to the real solution to be found by the drill process. It may be desirable to run the drill more than once, from different starting points, in an attempt to establish a bound for Z. The criteria for subsequent drilling could be based on the number of cases processed,

the elapsed run-time, or some definition of distance from the coordinates used in the initial run. This would likely be most useful when no integer bound has been established, or when the existing bound is not resulting in significant pruning.

6.3 Multiple Solutions

It is possible to have multiple optimal solutions to a linear program. This simply amounts to having several X sets yielding the same value for Z. Consider the 2-variable problem in Section 1.2, had the upper bound been simply $x_2 = 4$. The simplex algorithm, as implemented here, does not provide for multiple optimal solutions, and instead just produces the first such solution it finds. There are known methods for finding multiple solutions that could be incorporated if required [7].

Depending on the application, the user may wish to have the first solution, all solutions, or only a subset of solutions with certain extra conditions satisfied. In the latter instance, the program could offer alternatives and allow the user to choose the direction to take.

6.4 Optimized Queuing

The queue implemented here is a simple FIFO – First-In, First-Out. A few modifications were tested, but the improvements were minimal and short-lived. Simply prioritizing on the upper bound led to a depth-first search, while trying to guarantee a pure breadth-first search caused performance problems once the queue became very large.

Still, while the algorithm skips cases whose bounds have already been superseded (once an integer solution is found), there may be room for improvement prioritizing cases. The queue might take the form of an index into the existing queue structure, or an entirely different structure such as a balanced binary tree. Either should require only modification of the PUTCASE() and GETCASE() routines. New cases are assigned some sort of score, and inserted into the tree where appropriate. Examination of various scoring methods might result in an improved algorithm, being one in which the pruning has a greater limiting effect on the search.

Suggestions for scoring methods to test include ordering by Z, ordering by l (which happens to a large extent already, implicitly by design), and ordering by the number of possible values for a given x_i (ascending or descending).

6.5 Optimize the Choice of Fixed Variables

Our code simply starts fixing variables at x_1, and proceeds in order as far as it needs to go. According to [7], there are some heuristics that may be employed in choosing which variables to fix first. The internal manipulation to achieve this may require

significant coding, however, the user can also manually reorder variables if he or she is aware of any benefit to a specific ordering.

6.6 Matrix Routines

In this code, we wrote our own matrix manipulation. The operations are simple, being largely just pivoting and producing submatrices, but the use of a matrix manipulation library such as BLAS or LAPACK may still realize a performance improvement. In addition to time, such libraries may employ memory-optimized storage for matrices, reducing the footprint of larger linear programs. The change in coding to do this would be significant, as the use of such libraries likely involves specialized data types and function calls.

6.7 Network Routines

The actual client-server protocol and supporting code was a source of several particularly nagging bugs. In hindsight, it would have been worthwhile to investigate existing client-server modules to see if one could be adapted, rather than develop our own, educational though the process was. The end result, however, was quite flexible, where clients can come and go as they please, and the master automatically adjusts and re-queues work appropriately. Our protocol is also simple and ASCII-based, allowing easy debugging of the interactions.

6.8 Other Parallel Methods

The original choice of a client-server model seems to be appropriate, however, it may still be worthwhile to examine other methods. There may be other areas of the code which could be further parallelized, such as matrix manipulation. Shared-memory models for the queue might produce performance improvements over that of the network protocol. Using MPI in place of explicit sockets might allow for different flexibility and additional robustness.

References

[1] J. Libano Alonso, H. Schmidt, and V. N. Alexandrov, *Parallel branch and bound algorighms for integer and mixed integer linear programming problems under pvm*, Recent advances in parallel virtual machine and message passing interface, 4th European PVM/MPI Users' Group Meeting **1332** (1997), 313–320.

[2] D. Anderson and D. Werthimer, *Seti@home project*, `http://setiathome.berkeley.edu`.

[3] R. Boehning, R. Butler, and B. Gillett, *A parallel integer linear programming algorithm*, European Journal of Operational Research **34** (1988), 393–398.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving problems on concurrent processors, volume 1*, Prentice Hall, 1988.

[5] B. Gendron and T. G. Crainic, *Parallel branch-and-bound algorithms: Survey and synthesis*, Operations Research **Vol. 42, No. 6** (1994), 1042–1066.

[6] W. Kocay and D. Kreher, *Graphs, algorithms, and optimization*, Chapman & Hall/CRC, 2005.

[7] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization*, Wiley & Sons, 1988.

[8] V. Pande et al., *Folding@home project*, `http://folding.stanford.edu/`.

Appendix A

Code Listings

A.1 Overview

The code comprising the pre-processor and parallel algorithm, including the drill routine, is presented here. The pre-processor and conversion utility for the MPS format were written in Perl, while the parallel algorithm itself was written in C. For the latter, a Makefile is included. The C code compiles into a single executable which acts as a master or client, depending on the command line invocation.

Individual files are as follows:

Makefile used to compile the algorithm code, by typing "make"

client.c client portion of the algorithm

drill.c drill routines

main.c initial program logic to check syntax and invoke master or client code

master.c master portion of the algorithm

phases.c phases of the simplex algorithm

queue.c queue management routines

queue.h definitions and function templates for the queue

solver.c main solver routine which handles cases and invokes the simplex algorithm on them

solver.h definitions and function templates for algorithm

utils.c miscellaneous file and matrix routines

solver resulting single executable

mps2mat convert MPS format to our format

optimize optimize column order in a tableau

A.2 Usage

Usage of the executables, where italicized command-line arguments are optional:

solver -m filename ***numclients***

> Initiate a solver run as the master. **f**ilename is the name of the file containing the ILP. numclients is optional, but if given, the master waits until that many clients have connected before beginning the run.

solver -c hostname

> Initiate a solver run as a client. **h**ostname is the name of the master machine to connect to.

solver -f filename

> Initiate a solver run only for overall feasibility. Only runs a single simplex instance. **f**ilename is the name of the file containing the LP.

mps2mat ***-revcost*** **mpsfile > matfile**

> Convert the MPS format found in **m**psfile to our tableau format and put the result in **m**atfile. If the optional -revcost is given, take the cost (Z) row to be the negative of that found in **m**psfile. The MPS format did not have a mechanism to specify whether to maximize or minimize a given LP, leaving the choice up to the operator.

optimize method matfile > newmatfile

> Re-order columns in **m**atfile by **m**ethod and put the resulting tableau in **n**ewmatfile. **m**ethod is any combination and order of 'z', 'c', and 'r', such as 'zcr', 'crz', etc...

A.3 Parallel Solver

Listing A.1: Makefile

```
CFLAGS=-g
LDFLAGS=-lm

default: solver

solver: main.o client.o master.o solver.o phases.o utils.o queue.o drill
    .o

main.o: main.c solver.h

master.o: master.c solver.h

client.o: client.c solver.h

solver.o: solver.c solver.h

drill.o: drill.c solver.h

phases.o: phases.c solver.h

utils.o: utils.c solver.h

queue.o: queue.c queue.h

clean:
  rm -f *.log *.o solver

tar: parallel.tar.gz

parallel.tar.gz: *.c *.h Makefile
  tar cvfz parallel.tar.gz *.c *.h Makefile
```

Listing A.2: main.c

```
#include "solver.h"

FILE *LOG, *IN, *OUT;
int MASTER;
int phase;
int *optX, *fixed;
int found = 0;
double best = 0;
int fileRows, fileCols;

int main(int argc, char *argv[])
{

  char filename[256], logname[256];
  int i, minclients;

  // decide master vs. client
  if(((argc < 3) && !strcmp(argv[1],"-c")) || ((argc < 3) && !strcmp(
      argv[1],"-m"))) {

    fprintf(stderr, "USAGE: %s {-m filename #clients | -c hostname}\n",
        argv[0]);
    fprintf(stderr, "\tset up master (-m) or client (-c) process\n");

  } else  {

    strcpy(filename, argv[2]);
    strcpy(logname,filename);
    strcat(logname,argv[1]);
    strcat(logname,".log");
    LOG = fopen(logname,"w");

    stamp;

    if(!strcmp(argv[1],"-f")) { // feasibility check only

      debug("feasibility check\n");
      feasibility(argv[2]);

    } else if(!strcmp(argv[1],"-c")) { // we have a hostname, become
        client and join

      debug("becoming client\n");
      client(argv[2]); // pass the hostname

    } else { // must be the master

      debug("becoming master\n");
      minclients = 0;
      if(argc > 3) minclients = atoi(argv[3]);
```

```
      master(argv[2],minclients); // pass the filename

      if(found) {

        printf("Integer solution found:\n");
        for(i=0; i < fileCols-1; i++) {

          printf("\tX[%d] = %d\n", i+1, optX[i]);

        }

        printf("\tZ = %f\n", best);

      } else {

        printf("No integer solutions found\n");

      }

    }

    fclose(LOG);

  }

  exit(0);

} // main
```

Listing A.3: master.c

```
#include "solver.h"

int qsize = 0;
int feas_only = 0;
queue *q;

int feasibility(char *filename)
{

  int rc;

  feas_only = 1;
  rc = master(filename,0);

  printf("done with feasibility.\n");

  return(rc);

}

int master(char *filename, int minclients)
{

  double **tab, **inittab; // tableau
  int *prows; // pivot column by row
  int vars; // number of variables
  int eqs; // number of inequalities
  int r, rc;
  case_t *up_case, *dn_case, *dr_case;
  FILE *fp;

  // now figure out initial case over the reals

  stamp;
  fp = fopen(filename, "r");
  if(fp) {

    tab = read_file(fp,&eqs,&vars);
    fclose(fp);

  } else {

    perror(filename);
    exit(0);

  }

  stamp;

  optX = malloc(vars * sizeof(int));
```

```
  for(r=0; r < vars; r++) {

    optX[r] = 0;

  }

//  printf("initial tableau:\n");
//  print_table(tab,eqs,vars);

  // sync threads and start solving

  printf("setting up initial case\n");

  q = newqueue();

  prows = malloc((eqs+2) * sizeof(int));
  for(r=1; r<= eqs; r++) prows[r]=-1;
  up_case = malloc(sizeof(case_t));
  dr_case = malloc(sizeof(case_t));
  dn_case = malloc(sizeof(case_t));
  up_case->numfixed = dn_case->numfixed = dr_case->numfixed = 0;
  up_case->fixed = dn_case->fixed = dr_case->fixed = NULL;
  up_case->nextvar = dn_case->nextvar = dr_case->nextvar = 0;
  up_case->direction = 1;
  dn_case->direction = -1;
  dr_case->direction = 0;
  up_case->tableau = dn_case->tableau = dr_case->tableau = NULL;

  // compute real solution using up_case for now
  subtableau(tab, eqs, vars, up_case);
  rc = simplex(up_case->tableau, prows, eqs, vars);
  stamp;
  if(rc == FOUND) {

    printf("program has real solution\n");
    for(r = 1; r <= eqs; r++) {

      if(prows[r] != -1) {

        printf("\tX[%d] = %f, Z contrib %f\n", prows[r], up_case->
            tableau[r][vars+1],
          up_case->tableau[r][vars+1] * tab[eqs+1][prows[r]]);

      }

      if(prows[r] == 1)  {

        up_case->nextvar = up_case->tableau[r][vars+1];
        dn_case->nextvar = up_case->tableau[r][vars+1];
        dr_case->nextvar = up_case->tableau[r][vars+1];
```

```
      }

    }

    printf("\tZ = %f\n", up_case->tableau[r][vars+1]);
    printf("initial X[1] = %.3f\n", up_case->nextvar);
    up_case->bound = dn_case->bound = -up_case->tableau[r][vars+1];

#ifdef USE_DRILL
    // put drill case on the queue
    putcase(q,dr_case); qsize++;
#endif
    // put two cases on the queue, searching in opposite directions.
    putcase(q,up_case); qsize++;
    putcase(q,dn_case); qsize++;

    // initial tableau was just for real solution.  not needed for queue
        .
    for(r=0;r<=eqs;r++) free(up_case->tableau[r]);
    up_case->tableau = NULL;

    if(!feas_only) { // start the queue

      printf("waiting for connections...\n");
      rc = queuemgr(tab,minclients);

      printf("done solving.\n");

    }

  } else { // we're done early

    if(found) { // we found an optimal integer solution somewhere

      puts("integer solution found");

      printf("\tZ = %f\n", best);

      for(r=0; r < vars; r++) {

        printf("\tX_%d = ", r+1);
        printf("%d\n",optX[r]);

      }

    } else if(rc == UNBOUNDED) { // program was unbounded

      puts("program is unbounded");

    } else if(rc == INFEASIBLE) { // program was infeasible
```

```
      puts("program is infeasible");
    } else { // shouldn't get this far
      puts("unspecified error");
    }
  }
  return(0);
}
int queuemgr(double **tab, int minclients) {
  int s, s2, addrlen;
  char hostname[256], *inaddr, *args;
  struct sockaddr_in addr, addr2;
  struct hostent *host;
  struct addrinfo *address;

  FILE *fp;
  char buf[256], cmdline[256];
  int done;
  fd_set fds;
  int i,j;
  case_t *CASE, *upnewc, *dnnewc;
  client_t *client;
  int clicnt, numclients;
  int clientrc, assigned, vars, rc;
  double clientZ;
  queue *cq;
  node *cn;
  int r, on;

  int cases_queued, cases_offered, cases_pruned, cases_rejected;
  double rate_queued;
  struct timeval runstart, runstop;
  suseconds_t runtime;

  cases_queued = qsize; // initial case
  cases_offered = cases_pruned = cases_rejected = 0;
  runstart.tv_sec = 0;

  // set up the server socket and start listening
  printf("setting up master socket\n");
  gethostname(hostname, sizeof(hostname));
  if((s = socket(PF_INET,SOCK_STREAM,0)) < 0) {
    perror ("socket()");
```

```
    exit (errno);
  }

  printf("getting address for hostname '%s'\n", hostname);
  if(!(host = gethostbyname ((char*)hostname))) {
    perror("gethostbyname()");
    exit(errno);
  }

/*
  if(getaddrinfo(hostname, NULL, NULL, &address)) {
    perror("getaddrinfo()");
    exit(errno);
  }
*/

  printf("config socket\n");
  addr.sin_family = AF_INET;
  addr.sin_port = htons((unsigned short)SOLVER_PORT);
  memcpy((char*)&addr.sin_addr.s_addr, host->h_addr_list[0],host->
      h_length);
  on = 1;
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  printf("binding to socket\n");
  if(bind(s,(struct sockaddr*)&addr,sizeof(addr)) <0) {
//  if(bind(s, address->ai_addr, address->ai_addrlen) <0) {
    close(s);
    perror("bind()");
    exit(errno);
  }

  stamp;
  printf("listening for connections\n");
  if(listen(s,1)) {
    perror("listen()");
    exit(errno);
  }


  stamp;
  //gettimeofday(&runstart,NULL);

  clicnt = 0; numclients = 0; assigned = 0;
  cq = newqueue();

  done = 0;
  while(!done) {

/* Now we have a connection.  Let's talk...
 * 1) build up select() fd_sets, check for read blocking
```

```
 * 2) read a request and process it for each client
 */

    //fgets(buf, 255, stdin); // uncomment to allow manual control of
        interations

    if(clicnt) {
      printf("Cl : Sz/Qd/Of : Rj/Pr/Ib : Et Rt  %d : %d / %d / %d : 
          %d / %d / %d : %d %.0f  \r",
        numclients, qsize, cases_queued, cases_offered,
        cases_rejected, cases_pruned,
        (int)best, runstop.tv_sec - runstart.tv_sec,
        (double)cases_queued / (double)(runstop.tv_sec - runstart.tv_sec
            ));
    }

    FD_ZERO(&fds);

    //printf("adding listener socket to list\n");
    FD_SET(s,&fds); // add the listener socket

    if(minclients && (clicnt >= minclients) && !runstart.tv_sec) {

      gettimeofday(&runstart,NULL);

    }

    if(clicnt >= minclients) for(cn = cq->first; cn; cn = cn->next) {

      client = cn->c;

      if(client) {

//        printf("adding client %d to list\n", client->clinum);
        FD_SET(client->fd, &fds);

      }

    }

    select(FD_SETSIZE, &fds, NULL, NULL, NULL);

    if(FD_ISSET(s,&fds)) {

      debug("adding new client\n");

      client = malloc(sizeof(client_t));

      s2 = accept(s,(struct sockaddr*)&addr2,&addrlen);
      inaddr = inet_ntoa(addr2.sin_addr);
```

```
client->fd = s2;
client->addr = addr2;
client->inaddr = (char*)strdup(inaddr);
client->CASE = NULL;

if (!(fp = fdopen(s2,"r"))) {
  perror("fdopen() for read");
  close(s2);
  exit(errno);
} else {

  client->in = fp;
  setbuf(client->in, NULL);

}

if (!(fp = fdopen(dup(s2),"w"))) {

  perror("fdopen() for write");
  close(s2);
  exit(errno);

} else {

  client->out = fp;
  setbuf(client->out, NULL);

}

fprintf(LOG,"connection from '%s'\n",inaddr);

client->clinum = clicnt++;
putclient(cq,client); numclients++;

// spit tableau in original format to client
fprintf(client->out, "%d %d\r\n",clicnt,numclients);
fprintf(client->out, "begin\r\n");
fprintf(client->out, "%d %d\r\n", fileRows, fileCols);
for(i = 0; i <= fileRows; i++) {

  for(j = 1; j <= fileCols; j++) {

    fprintf(client->out, "%f ", tab[i][j]);

  }

  fprintf(client->out, "\r\n");

}

fprintf(client->out, "end\r\n");
```

```
      fflush(client->out);

    } // add client

    for(cn = cq->first; cn; cn = cn->next) {

      client = cn->c;

      if(client && FD_ISSET(client->fd, &fds)) {

//      printf("handling client %d (%s):\n", client->clinum, client->
    inaddr);

      fgets(buf, 255, client->in);
      strtok(buf,"\n\r");
      //printf("\treceived '%s'\n", buf);

      if(!strncmp(buf,"request",7)) { // client requests case

        //printf("\tclient requests case\n");
        if(client->CASE) {

          //printf("\tfreeing previous case\n");
          free(client->CASE->fixed);
          free(client->CASE);
          client->CASE = NULL;
          assigned--;

        }

        // grab one from the queue and relay it to client
        // it's possible the queue is empty, but we're not done

        CASE = getcase(q); if(CASE) qsize--;

        // prune as we go
        while(CASE && found && (best <= CASE->bound)) {

          free(CASE->fixed);
          free(CASE);
          cases_pruned++;
          CASE = getcase(q); if(CASE) qsize--;

        }

        if(CASE) {

          fprintf(client->out, "case %d %lf %d %lf\r\n", CASE->numfixed,
            CASE->nextvar, CASE->direction, CASE->bound);
          fflush(client->out);
```

```
      for(i = 0; i < CASE->numfixed; i++)
        fprintf(client->out, "%d ", CASE->fixed[i]);

      fprintf(client->out, " end\r\n");
      fflush(client->out);

      client->CASE = CASE;
      assigned++;
      cases_offered++;

    } else if(!assigned) { // the work queue is really empty

      //printf("\tinforming client we are done\n");
      fprintf(client->out, "done\r\n");
      fflush(client->out);

    } else {

      //printf("\ttelling client to wait, assigned = %d\n", assigned
          );
      fprintf(client->out, "wait\r\n");
      fflush(client->out);

    }

  } else if(!strncmp(buf, "results", 7)) { // client has results

    //printf("\tclient has results\n");

    CASE = client->CASE;

    // format: "results flag Z-value" then "X-values"
    // if results have potential, take them, otherwise skip
    // flag = INFEASIBLE/UNBOUNDED/FOUND/INTFOUND

    rc = sscanf(buf,"results %d %lf\r\n", &clientrc, &clientZ);

    if(clientrc == INFEASIBLE) { // do nothing

      fprintf(client->out, "thanks\r\n"); fflush(client->out);

    } else if(clientrc == UNBOUNDED) { // we're done, entirely

      fprintf(client->out, "thanks\r\n"); fflush(client->out);
      done++;

    } else if(clientrc == FOUND) { // check against possible int
        soln, maybe accept
```

```
if(!found || (best > clientZ)) { // accept unless no
    improvement possible

  fprintf(client->out, "accepted\r\n"); fflush(client->out);

  // make 2 new cases (up and down) and put work on queue
  // CASE->numfixed is what we gave.  newc->numfixed should
      increment
  upnewc = malloc(sizeof(case_t));
  dnnewc = malloc(sizeof(case_t));
  upnewc->numfixed = dnnewc->numfixed= CASE->numfixed+1;
  upnewc->fixed = malloc(upnewc->numfixed * sizeof(int));
  dnnewc->fixed = malloc(upnewc->numfixed * sizeof(int));
  upnewc->bound = dnnewc->bound = clientZ;

  // copy in what we originally gave the client
  for(i = 0; i < CASE->numfixed; i++) {

    upnewc->fixed[i] = CASE->fixed[i];
    dnnewc->fixed[i] = CASE->fixed[i];

  }

  // read the next fixed value, and the following real value
  fscanf(client->in, "%d %lf", &(upnewc->fixed[upnewc->
      numfixed-1]), &(upnewc->nextvar));
  dnnewc->fixed[dnnewc->numfixed-1] = upnewc->fixed[upnewc->
      numfixed-1];
  dnnewc->nextvar = upnewc->nextvar;

  upnewc->direction = 1;
  dnnewc->direction = -1;
  putcase(q,upnewc); qsize++; cases_queued++;
  if(dnnewc->nextvar > 0.0) {

    putcase(q,dnnewc); qsize++; cases_queued++;

  } else {

    free(dnnewc->fixed);
    free(dnnewc);

  }

  fgets(buf,255,client->in); // read rest of line (CR/NL most
      likely)

} else {

  fprintf(client->out, "nothanks\r\n"); fflush(client->out);
  cases_rejected++;
```

```
  }

} else if(clientrc == INTFOUND) { // check against int soln,
    maybe keep

  if(!found || (best > clientZ)) { // accept unless no
      improvement possible
    // must be *integer* improvement to matter


    fprintf(client->out, "accepted\r\n"); fflush(client->out);
    fscanf(client->in,"%d", &vars);

    if(!client->CASE->direction) { // drill results

      for(i = 0; i < vars; i++) {

        fscanf(client->in, " %d", &optX[i]);

      }

      fgets(buf,255,client->in); // read 'end' line

    } else {

      for(i = 0; i < CASE->numfixed; i++) {

        optX[i] = CASE->fixed[i];

      }

      for(; i < vars; i++) {
        fscanf(client->in, " %d", &optX[i]);
        printf("X[%02d] = %2d,\t", i, optX[i]);
      }

      fgets(buf,255,client->in); // read 'end' line
      //printf("\n");

    }

    best = clientZ;
    found++;

  } else {

    fprintf(client->out, "nothanks\r\n"); fflush(client->out);

  }
```

```
        }

      } else if(!strncmp(buf, "bye", 3)) { // client is exiting

        //printf("\tclient exiting\n");
        fclose(client->out);
        fclose(client->in);

        //printf("\tclosed file descriptors for client\n");
        if(client->CASE) { // did not finish its workload, requeue

          //printf("\trequeuing existing case\n");
          putcase(q,client->CASE); qsize++;
          assigned--;

        }

        free(client); cn->c = NULL; numclients--;

      } else {

        printf("\n\tunrecognized input '%s'\n\tdropping client %d\n",buf
            ,client->clinum);
        if(client->CASE) {

          //printf("\trequeuing existing case\n");
          putcase(q,client->CASE); qsize++;
          assigned--;

        }

        free(client); cn->c = NULL; numclients--;

      }

    }

    }

    // we are done when there are no more clients and no more work
    if(!numclients && !qsize) done++;

    gettimeofday(&runstop,NULL);

  }

  printf("\nalgorithm completed normally\n");

  runtime = (runstop.tv_sec - runstart.tv_sec) * 1000000 + runstop.
      tv_usec - runstart.tv_usec;
```

```
    printf("queued %d, offered %d, rejected %d, pruned %d\n", cases_queued
        , cases_offered, cases_rejected, cases_pruned);
    printf("total main loop time %d microseconds\n", runtime);
    // searchable log line
    rate_queued = (double)cases_queued * 1000000;
    rate_queued /= (double)runtime;
    fprintf(LOG,"STATS %d,%d,%d,%d,%.1lf,%.1lf,%.1lf\n",minclients,
      cases_queued, cases_offered, cases_rejected, cases_pruned,
      (double)runtime/1000000.0, (double)runtime / 1000000.0 * minclients,
          rate_queued);

    return(0);

}
```

Listing A.4: client.c

```
#include <signal.h>
#include "solver.h"

static FILE *IN, *OUT;

int client(char* master) {

  double **tab; // tableau
  int *prows; // pivot column by row
  int vars; // number of variables
  int eqs; // number of inequalities
  int *intreq;
  int r, c, rc;
  struct sigaction sigact, oldsigact;
  sigset_t sigset;

  // trap signals for clean exit from master
  sigemptyset(&sigset);
  sigact.sa_handler = &terminate;
  sigact.sa_mask = sigset;
  sigact.sa_flags = 0;
  sigact.sa_restorer = NULL;
  sigaction(SIGTERM, &sigact, &oldsigact);
  sigaction(SIGINT, &sigact, &oldsigact);

  // minimal initial setup, get tableau and parameters
  tab = initial_setup(master,&eqs,&vars);

  // pull out slack variable flags from row 0
  intreq = (int*)malloc(vars * sizeof(int));
  for(c = 1; c <= vars; c++) {

    intreq[c-1] = tab[0][c] ? 1 : 0;

    printf("variable %d %s required to be integer\n", c, intreq[c-1] ? "
        is" : "is not");

  }

//  printf("initial tableau:\n");
  //print_table(tab,eqs,vars);

  prows = malloc((eqs+2) * sizeof(int));
  for(r=0; r < eqs + 2; r++) prows[r] = -1;

  // sync threads and start solving
  printf("solving...\n");
  rc = solver(tab, prows, eqs, vars, intreq);

  printf("done solving.\n");
```

```
  return(0);

}

case_t *requestcase() // get a case from the master
{

  char resp[256];
  case_t *CASE;
  int i, blank, done;
  fd_set fds;

  resp[0] = 0;
  done = 0;

  do {

    //printf("requesting case\n");
    fprintf(OUT,"request\r\n"); fflush(OUT);

    //printf("reading response from master\n");
    fgets(resp, 255, IN);
    strtok(resp,"\r\n");
    //printf("server said '%s'\n", resp);

    if(!strncmp(resp,"case",4)) { // we've got work to do...

      done = 1;

    } else if(!strncmp(resp, "wait", 4)) { // we need to wait for work

      //printf("waiting for work from master\n");
      sleep(1);

    } else {

      done = 2;

    }

  } while(!done);

  if(done == 2) {

      fprintf(OUT,"bye\r\n"); fflush(OUT);
      return(NULL);

  }

  // read results and return a case_t*
```

```
  CASE = malloc(sizeof(case_t));
  sscanf(resp, "case %d %lf %d %lf", &(CASE->numfixed), &(CASE->nextvar)
      , &(CASE->direction), &(CASE->bound));
  CASE->fixed = malloc(CASE->numfixed * sizeof(int));
  for(i=0; i < CASE->numfixed; i++) {

    fscanf(IN,"%d",&(CASE->fixed[i]));

  }

//  printf("\ngot case: fixed = %d, nextvar = %lf, direction = %s, bound
    = %lf\n", CASE->numfixed, CASE->nextvar, (CASE->direction == 1 ? "
   up" : "down"), CASE->bound);

  fgets(resp,255,IN); // catch EOL chars

  return(CASE);

}

int announce(case_t* CASE, int rc, int vars, double *X, double Z) //
   announce a solution to the master
{

  int i, rc2;
  char resp[256];

  //printf("announcing results to server\n");
  rc2 = 0;

  fprintf(OUT,"results %d %f\r\n", rc, Z);
  fflush(OUT);

  fgets(resp, 255, IN);
  //printf("server said '%s'\n", resp);

  if(!strncmp(resp,"accepted",8)) {

    rc2++;

    if(!CASE->direction) { // this was just a bound

      fprintf(OUT, "%d", vars);
      for(i = 0; i < vars; i++) {
        if(i < CASE->numfixed) fprintf(OUT," %d ", CASE->fixed[i]);
        else fprintf(OUT," %.0f ", round(X[i]));
      }

      fprintf(OUT, "end\r\n"); // trailing 'end' for sync

    } else if(rc == FOUND) {
```

```
      fprintf(OUT, "%d %lf\r\n", (int)X[CASE->numfixed-1], X[CASE->
          numfixed]);
//       printf("sending new case X[%d] = %d (next = %f)\n", CASE->
    numfixed-1, (int)X[CASE->numfixed-1], X[CASE->numfixed]);

    } else { // must be INTFOUND (since the server is interested)

//       printf("sending integer solution, %d vars starting with %d\n",
    vars, CASE->numfixed-1);
      fprintf(OUT, "%d", vars);
      for(i = CASE->numfixed-1; i < vars; i++) {
        fprintf(OUT," %.0f ", round(X[i]));
//         printf("X[%d] = %.0f\t", i, round(X[i]));
      }


      fprintf(OUT, "end\r\n"); // trailing 'end' for sync
      printf("\n");

    }

    fflush(OUT);

  }

  return(rc2);

}

double **initial_setup(char *hostname, int *eqs, int *vars) //
    initialize our copy of the tableau and parameters
{

  int s, clinum,numclients;
  char buf[256];
  struct sockaddr_in addr;
  struct hostent *host;

  if((s = socket(PF_INET,SOCK_STREAM,0)) < 0) {
    perror ("socket()");
    exit (errno);
  }

  if(!(host = gethostbyname (hostname))) {
    perror("gethostbyname()");
    exit(errno);
  }

  addr.sin_family = AF_INET;
  addr.sin_port = htons((unsigned short)SOLVER_PORT);
```

```
  memcpy(&addr.sin_addr.s_addr, host->h_addr_list[0],host->h_length);

  printf("connecting to '%s'\n", hostname);
  if(connect(s, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
    perror("connect()");
    exit(errno);
  }

  if (!(IN = fdopen(s,"r"))) {
    perror("fdopen() for read");
    exit(errno);
  }

  if(!(OUT = fdopen(dup(s),"w"))) {
    perror("fdopen() for write");
    exit(errno);
  }

  setbuf(IN,NULL);
  setbuf(OUT,NULL);

  MASTER = s;

  printf("reading initial tableau\n");
  fscanf(IN,"%d %d", &clinum, &numclients);

  printf("I am client %d of %d\n", clinum,numclients);

  return(read_file(IN, eqs, vars));

}

void terminate(int signum) {

  printf("SIGTERM caught, terminating client\n");
  fprintf(OUT,"bye\r\n"); fflush(OUT);
  exit(0);

}
```

Listing A.5: solver.c

```
#include "solver.h"

int solver(double **tab, int *prows, int eqs, int vars, int *intreq)
{

  double **mytab;
  case_t *CASE, *newc;
  double *X, Z;
  int myvars, rc, r, c, newx, isol, keepranging;
  int startx;

  X = malloc(vars * sizeof(double));

  while(CASE = requestcase()) {

    rc = UNKNOWN;
    for(r=0; r < vars; r++) X[r] = 0;

    // a 'case' is a set of fixed X values known to contain
    // feasible solutions farther down the tree (i.e. fixing more
    // X values). Here, we take a case, and fix one more X, at
    // several values starting from 0, creating more cases.
    //
    // linear optimization being linear by definition, we know that
    // once a given X value makes the program infeasible, we can
    // stop trying in that direction.
    //
    // we also know that the program is entirely unbounded if
    // *any* unbounded case is found, so we drop out immediately

    startx = (int)ceil(CASE->nextvar);
    if(CASE->direction == -1) startx--;

    keepranging = (startx >= 0);
    if(CASE->direction)
      for(newx = startx; keepranging && (newx >= 0); newx += CASE->
          direction) {

      // create new case, fixing one more column
      newc = malloc(sizeof(case_t));
      newc->numfixed = CASE->numfixed+1;
      newc->fixed = malloc(newc->numfixed * sizeof(int));
      newc->direction = CASE->direction;
      for(c=0; c < CASE->numfixed; c++) newc->fixed[c] = CASE->fixed[c];
      newc->fixed[newc->numfixed-1] = newx;

//       printf("\n-------------------------------------------------\
    nchecking case, fixed = ( ");
      printf("fixed = (");
      for(c=0;c < newc->numfixed; c++) {
```

```
        printf("%d ", newc->fixed[c]);

      }

      printf(") %lf         \r",CASE->bound);

      newc->tableau = NULL;
      subtableau(tab,eqs,vars,newc);
      mytab = newc->tableau;
      myvars = vars - newc->numfixed;

      //printf("initial sub-tableau (%d free vars):\n",myvars);
      phase = 0; // for print_table()
      //print_table(mytab,eqs,myvars);

      rc = simplex(mytab, prows, eqs, myvars);

      if(rc == FOUND) {

        // grab X and Z
        Z = -mytab[eqs+1][myvars+1]; // the portion from *this*
            subtableau
        for(c=newc->numfixed; c <= myvars; c++) X[c] = 0;
        for(r=1; r <= eqs; r++) {

          if(prows[r] != -1) {

            X[newc->numfixed + prows[r] - 1] = mytab[r][myvars+1];

          }

        }

        for(c=0; c < newc->numfixed; c++) { // "fixed" portion

          X[c] = (double)newc->fixed[c];
//          Z += tab[eqs+1][c] * X[c];

        }

        // compare and store
        newc->bound = Z;

        // check for integer solutions
        isol = 1;
        for(r=newc->numfixed; r < vars; r++) {

          if(intreq[r] && ((X[r] - floor(X[r])) > EPSILON)) {
//            printf("X[%d] is not quite integer\n",r);
            isol = 0;
```

```
            }

          }

          if(isol) {

            rc = INTFOUND;
            printf("\nFOUND INTEGER SOLUTION: Z = %d\n", (int)round(Z));

          }
#ifdef ADD_SLACK
          announce(newc,rc,vars-eqs,X,Z);
#else
          announce(newc,rc,vars,X,Z);
#endif

        } // FOUND solution

        // do we keep going?
        // stop on integer solution found
        //if(rc == INTFOUND) keepranging = 0;

        // stop on unbounded, always
        if(rc == UNBOUNDED) keepranging = 0;

        // stop if past known feasible
        if(rc == INFEASIBLE) keepranging = 0;

        for(c=0; c < eqs + 2; c++) free(newc->tableau[c]);
        free(newc->tableau);
        free(newc->fixed);
        free(newc);

      } // for new X

      else {

        drill(tab,prows,eqs,vars,intreq,CASE); // drill instead of sweep
        printf("\n");

      }

//      printf("\n-------------------------------------------------\ndone
    ranging X_%d\n\n", CASE->numfixed+1);

      free(CASE->fixed);
      free(CASE); // uncommenting this seems to wreak havoc with glibc, no
          idea why.

    } // while
```

```
    printf("\nfinished all cases\n");
    return rc;
} // solver
```

Listing A.6: drill.c

```
#include "solver.h"

int drilldepth = 0;

int drill(double **tab, int *prows, int eqs, int vars, int *intreq,
    case_t *CASE)
{

  double **mytab;
  case_t *newc;
  double *X, Z;
  int myvars, rc, rc2, r, c, newx, isol, keepranging, keepdrilling;
  int startx;

  X = malloc(vars * sizeof(double));

    rc = UNKNOWN;
    for(r=0; r < vars; r++) X[r] = 0;

    // a 'case' is a set of fixed X values known to contain
    // feasible solutions farther down the tree (i.e. fixing more
    // X values).  Here, we take a case, and fix one more X, at
    // several values starting from 0, creating more cases.
    //
    // linear optimization being linear by definition, we know that
    // once a given X value makes the program infeasible, we can
    // stop trying in that direction.
    //
    // we also know that the program is entirely unbounded if
    // *any* unbounded case is found, so we drop out immediately

    startx = (int)ceil(CASE->nextvar);

    keepranging = (startx >= 0);
    keepdrilling = (drilldepth < vars);
    for(newx = startx; (newx > 0) && (newx >= startx-1); newx--) {

      // create new case, fixing one more column
      newc = malloc(sizeof(case_t));
      newc->numfixed = CASE->numfixed+1;
      newc->fixed = malloc(newc->numfixed * sizeof(int));
      for(c=0; c < CASE->numfixed; c++) newc->fixed[c] = CASE->fixed[c];
      newc->fixed[newc->numfixed-1] = newx;

//       printf("\n-----------------------------------------------\
    nchecking case, fixed = ( ");
      printf("depth %d fixed = (", drilldepth);
      for(c=0;c < newc->numfixed; c++) {

        printf("%d ", newc->fixed[c]);
```

```
}

printf(")␣%lf␣␣␣␣␣␣␣␣\r",CASE->bound);

newc->tableau = NULL;
subtableau(tab,eqs,vars,newc);
mytab = newc->tableau;
myvars = vars - newc->numfixed;

//printf("initial sub-tableau (%d free vars):\n",myvars);
phase = 0; // for print_table()
//print_table(mytab,eqs,myvars);

rc = simplex(mytab, prows, eqs, myvars);

if(rc == FOUND) {

  // grab X and Z
  Z = -mytab[eqs+1][myvars+1]; // the portion from *this*
      subtableau
  for(c=newc->numfixed; c <= myvars; c++) X[c] = 0;
  for(r=1; r <= eqs; r++) {

    if(prows[r] != -1) {

      X[newc->numfixed + prows[r] - 1] = mytab[r][myvars+1];

    }

  }

  for(c=0; c < newc->numfixed; c++) { // "fixed" portion

    X[c] = (double)newc->fixed[c];

  }

  // compare and store
  newc->bound = Z;

  // check for integer solutions
  isol = 1;
  for(r=newc->numfixed; r < vars; r++) {

    if(intreq[r] && ((X[r] - floor(X[r])) > EPSILON)) {
      isol = 0;
    }

  }
```

```
      if(isol) {

        rc = INTFOUND;
        rc2 = announce(newc,rc,vars,X,Z);
        if(!rc2) keepdrilling = 0; // server had better bound

      } else {

        newc->nextvar = X[newc->numfixed];

      }

    } // FOUND solution

    // do we keep going?

    // stop on unbounded, always
    if(rc == UNBOUNDED) keepranging = 0;

    // stop if past known feasible
    if(rc == INFEASIBLE) keepranging = 0;

    // drilldown
    if(keepdrilling) {

      newc->direction = 0;
      drilldepth++;
      drill(tab, prows, eqs, vars, intreq, newc);
      drilldepth--;

    }

    for(c=0; c < eqs + 2; c++) free(newc->tableau[c]);
    free(newc->tableau);
    free(newc->fixed);
    free(newc);

  } // for new X

  return rc;

} // solver
```

Listing A.7: phases.c

```
#include "solver.h"

extern FILE *LOG;

int simplex(double **tab, int *prows, int eqs, int vars)
{

  int rc = 0;

  // phase 0 - row-reduce to find basis solution
  //printf("starting phase 0\n");
  if(phase0(tab,prows,&eqs,vars)) {

    printf("program is infeasible after phase 0\n");
    rc = INFEASIBLE;

  } else {

    //printf("phase 0 complete:\n");
    //print_table(tab,eqs,vars);

    // phase 1 - make basis solution feasible
    //printf("starting phase 1\n");
    if(rc = phase1(tab,prows,eqs,vars)) {

      if(rc == INFEASIBLE) {

        //printf("program is infeasible after phase 1\n");
        print_table(tab,eqs,vars);

      } else {

        //printf("program is unbounded after phase 1\n");

      }

    } else {

      //printf("phase 1 complete:\n");
      print_table(tab,eqs,vars);

      // phase 2 - optimize final solution
      //printf("starting phase 2\n");
      if(phase2(tab,prows,eqs,vars)) {

        //printf("program is unbounded after phase 2\n");
        rc = UNBOUNDED;

      } else {
```

```
        //printf("phase 2 complete:\n");
        //print_table(tab,eqs,vars);

        rc = FOUND;

      } // phase 2

    } // phase 1

  } // phase 0

  return(rc);

} // simplex

int phase0(double **tab, int *prows, int *rows, int cols)
{

  int r,c,i,j,rc;
  int looking;

  phase=0;

  // find and pivot on the first non-zero entry for each column

  for(r = 1; r <= *rows; r++) {

//    debugf("\tlooking for pivot in row %d\n", r);

    for(c=1; (c <= cols) && !tab[r][c]; c++);

    if(c > cols) { // we have zeros on the left

      if(tab[r][cols+1]) { // but non-zero on the right

        rc = 1;

      } else { // entire row is zero, drop it

        fprintf(LOG,"row %d is all zeros, reducing rank and collapsing
            tableau rows\n",r);
        free(tab[r]);

        for(i=r+1; i <= (*rows)+1; i++) {

          tab[i-1] = tab[i];

        }

        (*rows)--;
        r--;
```

```
      }

    } else { // we have a pivot (r,c)

      pivot(tab,prows,*rows,cols,r,c);

    }

  }

  return(0);

} // phase0


int phase1(double **tab, int *prows, int rows, int cols)
{

  int r,c;
  int i;
  double min;
  int rmin;

  phase=1;

  // do we even need a phase 1? find most negative RHS value, pivot on
     phase1 column
  min = 0;
  rmin = 0;
  for(i=1; i<= rows; i++) {

    if(tab[i][cols+1] < min) {

      min = tab[i][cols+1];
      rmin = i;

    }

  }

  if(!rmin) {

    fprintf(LOG, "no phase 1 required\n");
    return 0;

  }

  // set up extra row (use r=0) and column (use c=0)
  for(i=1; i <= rows; i++) {
```

```
    tab[i][0] = (tab[i][cols+1] < -EPSILON ? -1 : 0);

  }

  tab[0][0] = 1;

  for(i=1; i<= cols; i++) {

    tab[0][i] = 0;

  }

  tab[0][cols+1] = 0;

//  fprintf(LOG, "initial phase 1 tableau:\n");
  print_table(tab,rows,cols);

  //printf("initial pivoting on (%d,0):\n",rmin);
  pivot(tab,prows,rows,cols,rmin,0);
  print_table(tab,rows,cols);
  for(c=1; c<= cols; c++) {

    if(tab[0][c] < -EPSILON) { // find minimum ratio and pivot there
//printf("\tlooking at col %d\n",c);

      // skip negative and zero entries in column c
      for(r=1; (r <= rows) && (tab[r][c] < EPSILON); r++);
      if(r > rows) return(UNBOUNDED);

      min = tab[r][cols+1] / tab[r][c]; // initial minimum ratio
      rmin = r; // is on this row
      r++; // skip to next row

      for(;r <= rows; r++) { // check the rest of column c

        if((tab[r][c] > EPSILON)
          && (tab[r][cols+1] / tab[r][c] < min)) {

          // only positive (r,c) entries are considered

          min = tab[r][cols+1] / tab[r][c];
          rmin = r;

        }

      }

      //printf("pivoting on (%d,%d) where min = %f\n", rmin,c,min);
      pivot(tab,prows,rows,cols,rmin,c);
      //print_table(tab,rows,cols);
      c = 0;
```

```
    }

  }

  if(tab[0][cols+1]) {

    return(INFEASIBLE);

  }

  return(0);

} // phase1

int phase2(double **tab, int *prows, int rows, int cols)
{

  phase=2;

  int i,r,c;
  double min;
  int rmin;

  for(c=1; c <= cols; c++) {

    if(tab[rows+1][c] < -EPSILON) { // find minimum ratio and pivot
        there

      //fprintf(LOG,"found negative cost entry in column %d = %f\n",c,
          tab[rows+1][c]);

      // find first positive entry in column
      for(r=1; (r <= rows) && (tab[r][c] < EPSILON); r++);

      // if none, problem is unbounded
      if(r > rows) return(UNBOUNDED);

      //fprintf(LOG,"\tfirst positive entry is in row %d = %f\n",r,tab[r
          ][c]);

      // else, start with that row as having the minimum ratio
      min = tab[r][cols+1] / tab[r][c];
      rmin = r;
      r++;

      // continue looking for row with minimum ratio
      for(;r <= rows; r++) {

        if((tab[r][c] > EPSILON) && (tab[r][cols+1] / tab[r][c]) < min)
            {
```

```
            min = tab[r][cols+1] / tab[r][c];
            rmin = r;

          }

        }

        //fprintf(LOG,"\tmin ratio is %f, pivot in (%d,%d)\n",min,rmin,c);
//        fprintf(LOG,"%d %d :: (cost,%d) = %f, row %d => %f / %f = %f\n",
    rmin,c,c,tab[rows+1][c],rmin,tab[rmin][cols+1],tab[rmin][c],min);
        pivot(tab,prows,rows,cols,rmin,c);
        c = 0;

      }

    }

    return(0);

} // phase2
```

Listing A.8: queue.c

```
#include "solver.h"

extern FILE *LOG;

queue *newqueue()
{

  queue *q;

  q = malloc(sizeof(queue));

  q->first = NULL;
  q->last = NULL;

  return(q);

}

void *getcase(queue *q)
{

  void *next = NULL;
  node *oldq;

  if(q && q->first) {

    // pop the queue
    oldq = q->first;
    q->first = oldq->next;
    if(q->first) q->first->prev = NULL;

    next = oldq->c;

    free(oldq);

  }

  return(next);

}

int putcase(queue *q, void *c)
{

  node *n;

  if(q) {

    // create new queue node
    n = malloc(sizeof(node));
```

```
    n->c = c;

    // push case on the end
    // empty queue?
    if(!q->first) {

      q->first = n;
      q->last = n;

      n->prev = NULL;
      n->next = NULL;

    } else {

      q->last->next = n;

      n->prev = q->last;
      n->next = NULL;
      q->last = n;

    }

  } else {

    debug("failed push:  queue not defined.\n");

  }

  return(0);

}

int putclient(queue *q, void *c)
{

  return(putcase(q,c));

}
```

Listing A.9: utils.c

```
#include "solver.h"

extern FILE *LOG;

void stamp()
{

return;
  time_t t;

  time(&t);
  debug(ctime(&t));
  puts(ctime(&t));

}

double **read_file(FILE *TAB, int *rows, int *cols)
{

  int r,c;
  //FILE *TAB;
  double **tab;
  char line[256]; // could be a problem for long lines before 'begin'
  int slack = 0;

  do {

    fscanf(TAB,"%s",line);

  } while(!feof(TAB) && strcmp(line,"begin"));


  if(feof(TAB)) {

    fprintf(stderr, "error reading file\n");
    exit(-1);

  }

  fscanf(TAB,"%d %d",&fileRows,&fileCols);

  *rows = fileRows; *cols = fileCols;

  fprintf(LOG, "reading %d rows of %d columns\n", fileRows, fileCols);
      fflush(LOG);

#ifdef ADD_SLACK
  slack = *rows - 1; // need slack variable for each equation
#endif
```

```
  // allocate the tableau
  // 'rows' includes the value function.  add two more for pivots and
     phase1
  fprintf(LOG, "actual malloc() is %d rows of %d columns\n", fileRows+2,
      fileCols+1+slack); fflush(LOG);
  tab = malloc(((*rows)+2) * sizeof(double*));
  for(r=0; r <= (*rows)+1; r++)
    tab[r] = malloc(((*cols)+1+slack) * sizeof(double));

  for(r=0; r <= *rows; r++) {

    //debugf("reading row %d\n", r);

    for(c=1; c <= *cols+slack; c++) {

      //debugf("\tcol %d\n", c);

      // col0 = 0col[1 .. vars] = read, col[1+slack .. 2*slack]
      // col[1 .. vars] = read from file
      // col[vars+1 .. vars+slack] = 1 for all rows but last
      // col[last] = read from file

      fscanf(TAB,"%lf",&tab[r][c]);

//       printf("%lf ", tab[r][c]);

    }

//     printf("\n");

  }

  do {

    //fgets(line,255,TAB);
    fscanf(TAB,"%s",line);

  } while(!feof(TAB) && strcmp(line,"end"));

  fgets(line,255,TAB);

  //fclose(TAB); // don't close it here, since we didn't open it
     ourselves

  // adjust for actual number of equations (rows-1) and variables (cols
     -1)
  (*rows)--;
  (*cols)--;
#ifdef ADD_SLACK
  *cols += slack;
#endif
```

```
  fprintf(LOG, "read %d rows, %d cols from file, added %d slack, new eqs
      /vars = %d/%d\n",
    fileRows, fileCols, slack, *rows, *cols); fflush(LOG);

  return(tab);

} // read_file()

void print_table(double **tab, int rows, int cols)
{

  return;

  if((cols > 60) || (rows > 30)) {

    printf("table too large for display\n");
    return;

  }

  int r,c;

  // print tableau
  for(r=(phase == 1 ? 0 : 1); r <= rows+1; r++) {

    for(c=(phase == 1 ? 0 : 1); c <= cols+1; c++) {

        fprintf(LOG,FORMAT, tab[r][c]);

    }

    fprintf(LOG,"\n");

  }

} // print_table

int pivot(double **tab, int *prows, int rows, int cols, int prow, int
    pcol)
{

  int r,c;
  double pval,cval, delta;

//  fprintf(LOG,"pivoting tableau of size (%d,%d) on cell (%d,%d)\n",
    rows,cols,prow,pcol);
  //print_table(tab,rows,cols);

  if(!tab[prow][pcol]) {
```

```
    fprintf(LOG, "cannot pivot on a zero value");
    return(-1);

  }

  // scale the pivot row itself first
  pval = tab[prow][pcol];
  for(c=0; c <= cols+1; c++)
    tab[prow][c] /= pval;

  for(r=0; r <= rows+1; r++) {

    cval = tab[r][pcol];
    if(r != prow) for(c=0; c <= cols+1; c++) {

      //fprintf(LOG, "\ttab[%d][%d] = %.1f",r,c,tab[r][c]);

      tab[r][c] -= tab[prow][c] * cval;
      delta = fabs(tab[r][c] - round(tab[r][c]));

      if(delta < EPSILON) tab[r][c] = round(tab[r][c]);

      //fprintf(LOG, " --> %.1f\n",tab[r][c]);

    }

    // avoid rounding issues
    //tab[r][pcol] = (r == prow ? 1 : 0);

//    fprintf(LOG, "\n");

  }

  prows[prow] = pcol;
  //fprintf(LOG,"\tafter:\n");
  //print_table(tab,rows,cols);

  //fprintf(LOG, "\tnew obj value = %f\n", tab[rows+1][cols+1]);

} // pivot

int subtableau(double **tab, int eqs, int vars, case_t *adj)
{

  // make a copy of the tableau for pivoting.
  // eqs & vars describe tab
  // adj contains information on how many columns to skip (numfixed)

  int r,c;
  int rows, cols;
  //double **t;
```

```
  rows = eqs + 2; // one for phase1, one for Z
  cols = vars + 1; // one for RHS, one for phase1

//  printf("numfixed = %d, subtableau size is %d rows, %d cols\n", adj->
    numfixed, rows, cols);

  //if(adj) cols -= adj->numfixed;

if(adj->tableau) printf("adj->tableau is not null\n");
  adj->tableau = (double**)malloc(rows * sizeof(double*));
//if(adj->tableau == ENOMEM) perror("first malloc() in subtableau()
    failed");
  for(r=0; r < rows; r++) {

    adj->tableau[r] = malloc((cols+1) * sizeof(double));

    for(c=cols; c; c--) {
//fprintf(LOG,"adjusting (%d,%d) for fixed = %d\n",r,c,(adj ? adj->
    numfixed : 0));

      if(!adj || !adj->numfixed) { // subtableau is entire tableau

        adj->tableau[r][c] = tab[r][c];

      } else if(c <= adj->numfixed) { // compensate on RHS for fixed
          vars

        adj->tableau[r][cols-adj->numfixed] -= tab[r][c] * adj->fixed[c
            -1];

      } else { // shift remaining unfixed vars to the left

        adj->tableau[r][c-adj->numfixed] = tab[r][c];

      }

    }

  }

  print_table(adj->tableau, eqs, vars - adj->numfixed);
  return(0);

} // subtableau
```

Listing A.10: solver.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <math.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/select.h>
#include <arpa/inet.h>

#include <sys/time.h>
#include <time.h>

#include "queue.h"

#define MAX_CLIENTS 255

#define SOLVER_PORT 11221
#define SOLVER_PORT_STR "11221"
//#define USE_DRILL 1

extern double **read_file(FILE*, int*, int*);
extern void print_table(double**, int, int);
extern int subtableau(double**, int, int, case_t*);

extern int master(char*,int);
extern case_t *requestcase();
extern int announce(case_t*, int, int, double*, double);
extern double **initial_setup(char*, int*, int*);
extern int queuemgr(double**,int);

extern int solver(double**, int*, int, int, int*);

extern int simplex(double**, int*, int, int);
extern int phase0(double**, int*, int*, int);
extern int phase1(double**, int*, int, int);
extern int phase2(double**, int*, int, int);
extern int pivot(double**, int*, int, int, int, int);
extern void stamp();
extern void terminate(int);

// for rounding to zero
#define EPSILON 1e-5

// per-cell format
#define FORMAT "%6.2f "

extern FILE *LOG;
```

```
extern int MASTER;
extern int phase, fileRows, fileCols;
extern int found;
extern double best;
extern int *optX, *fixed;
extern queue *q;

#define UNKNOWN 0
#define INFEASIBLE 1
#define UNBOUNDED 2
#define FOUND 3
#define INTFOUND 4

#define debug(x) fprintf(LOG,x); fflush(LOG);
#define debugf(x,y) fprintf(LOG,x,y); fflush(LOG);
```

Listing A.11: queue.h

```
// work queue
typedef struct {

  int numfixed;
  int *fixed;
  double nextvar;
  int direction;
  double bound;
  double **tableau;

} case_t;

// client queue
typedef struct {

  struct sockaddr_in addr;
  char *inaddr;
  int fd;
  FILE *in, *out;
  case_t *CASE;
  int clinum;

} client_t;

typedef struct queue_node_t node;
struct queue_node_t {

  void *c;
  node *prev, *next;;

};

typedef struct {

  node *first, *last;

} queue;

extern void *getcase(queue*);
extern int putcase(queue*, void*);
extern int putclient(queue*, void*);
extern queue *newqueue();

#define isempty(Q) (!(Q && Q->first))
```

A.4 Pre-processing Scripts

Listing A.12: mps2mat

```
#!/usr/bin/perl

$card = undef;
$rows = $cols = 0;
$minmax = 1; # minimize by default

if($ARGV[0] eq '-revcost') {

  $minmax = -1; # problem needs to have cost row reversed
  shift @ARGV;

}

LINE:
while($line = <>) {

  chomp $line;
  if($line =~ /^\s+/) { # sub-line

    next LINE unless $card;

    if($card eq 'ROWS') {

      ($rel, $rname) = split(" ",$line);

      $rnames{$rname} = $rows++;

      # we want all constraints to be "="
      # so add slack variables, -1 for ">=", 1 for "<="
      $sname = "slack $rname";
      $slacks{$rname} = -1 if($rel eq 'G');
      $slacks{$rname} = 1 if($rel eq 'L');
      $cnames{$sname} = $cols++ if($slacks{$rname});

      $rels{$rname} = 1;

      # remember the cost row
      $rels{$rname} = $minmax  if($rel eq 'N');
      $costrname = $rname if($rel eq 'N');

    } elsif($card eq 'COLUMNS') {

      ($cname, $pairs) = split(" ",$line,2);
      @pairs = split(" ",$pairs);

      $cnames{$cname} = $cols++ unless exists($cnames{$cname});
```

```
        while($rname = shift @pairs) {

          $coeff = shift @pairs;
          $coeff *= $rels{$rname};

          $mat[$rnames{$rname}][$cnames{$cname}] = $coeff;

          $sname = "slack $rname";
          $mat[$rnames{$rname}][$cnames{$sname}] = $slacks{$rname} if(
              $slacks{$rname});

        }

      } elsif($card eq 'RHS') {

        @pairs = split(" ", $line);
        shift @pairs unless($#pairs % 2); # dummy placeholder string

        while($rname = shift @pairs) {

          $rhs = shift @pairs;
          $rhs *= $rels{$rname};
          $mat[$rnames{$rname}][$cols] = $rhs;

        }

      }

    } else {

      $card = (split(" ",$line))[0];

    }

}

print STDERR "final matrix size, including cost and RHS is $rows rows by
    $cols cols\n";

$cols++; # to account for RHS column

print "begin\n$rows $cols\n";
ROW:
for $r (0 .. $rows-1) {

  @row = ();

COL:
  for $c (0 .. $cols-1) {

    $val = $mat[$r][$c];
```

```
      $val = 0 unless $val;
      push @row, $val;

    }

    $rline = join(' ',@row);
    if($rnames{$costrname} == $r) { # print the cost row last

      $costline = $rline;

    } else {

      print $rline . "\n";

    }

}

print $costline . "\nend\n";

exit 0;
```

Listing A.13: optimize

```
#!/usr/bin/perl

# optimize a tableau based on various rules.
# move more "important" columns to the left:
# * process larger entries in the cost row first (affects Z)
# * columns with small ratios of RHS to column entry (limits range)
# * columns with a large number of non-zero entries (more constraints)

# read in tableau file
local $order = shift @ARGV;
@lines = <>;
chomp @lines;
shift @lines; # begin
pop @lines; # end
($rows, $cols) = split(" ", shift @lines);

# convert to doubly-indexed array
foreach $r (0 .. $rows) {

  $row = shift @lines;
  @row = split(" ", $row);

  push @tab, [ @row ];

}

# create transpose so we can do column ops
foreach $r (0 .. $rows) {

  foreach $c (0 .. $cols-2) { # skip RHS

    $trans[$c][$r] = $tab[$r][$c];

  }

}

@sorted = sort optimize @trans;

# now transpose again to get final result
foreach $r (0 .. $rows) {

  foreach $c (0 .. $cols-2) {

    $tab[$r][$c] = $sorted[$c][$r];

  }

}
```

```
print "begin\n$rows $cols\n";
print join(" ", @{$tab[$_]}) . "\n" for (0 .. ($rows));
print "end\n";

sub optimize {

  @a = @{$a};
  @b = @{$b};

  # cost row entries
  $acost = $a[$#a];
  $bcost = $b[$#b];

  # non-zero entry count
  $acnt = scalar grep { $_ } @a;
  $bcnt = scalar grep { $_ } @b;

  # minimum ratio
  $aminrat = undef;
  $bminrat = undef;
  foreach $r (1 .. $rows-1) { # skip slackvar and cost rows (first and
      last)

    $rhs = $tab[$r][$cols-1];
    $arat = $a[$r] ? $rhs/$a[$r] : undef;
    $brat = $b[$r] ? $rhs/$b[$r] : undef;

    $aminrat = $arat if(defined $arat && (($arat < $aminrat) || !defined
        ($aminrat)));
    $bminrat = $brat if(defined $brat && (($brat < $bminrat) || !defined
        ($bminrat)));

  }


  my @sort = split(//, $order);
  my %sorts = (
    i => ($b[0] <=> $a[0]), # force integer variables to the left
    r => ($aminrat <=> $bminrat),
    z => (($acost && $bcost) ? ($acost <=> $bcost) : 0) || # both non-
        zero, straight compare
         ($acost ? -1 : 0) || # non-zero acost, zero bcost
           ($bcost ? 1 : 0), # vice-versa
    c => ($bcnt <=> $acnt)
  );

  my $rc = 0;
  foreach $s ('i', @sort) {

    $rc = $rc || $sorts{$s};
```

```
  }

#print "minrat = $aminrat/$bminrat, count = $acnt/$bcnt, cost = $acost,
    $bcost\n";
#print "rc = $rc\n\n";

  return $rc;

  return (
    ($bminrat <=> $aminrat) || # min RHS ratio
    (($acost && $bcost) ? ($acost <=> $bcost) : 0) || # both non-zero,
        straight compare
    ($acost ? -1 : 0) || # non-zero acost, zero bcost
    ($bcost ? 1 : 0) || # vice-versa
    ($bcnt <=> $acnt) || # simple constraint count
    0
  );

}
```