Using an interior point method in a branch and bound algorithm for integer programming.

July 7, 1992.

Brian Borchers
Department of Mathematics
Weir Hall
New Mexico Tech
Socorro, NM 87801
505-835-5393
borchers@jupiter.nmt.edu

John E. Mitchell
Department of Mathematical Sciences
Rensselaer Polytechnic Institute
Troy, NY 12180
518-276-6915
mitchj@rpi.edu

Subject Classification: Programming: integer: algorithms: branch-and-bound.

Abstract

This paper describes an experimental code that has been developed to solve zero-one mixed integer linear programs. The experimental code uses a primal–dual interior point method to solve the linear programming subproblems that arise in the solution of mixed integer linear programs by the branch and bound method. Computational results for a number of test problems are provided.

Introduction

Mixed integer linear programming problems are often solved by branch and bound methods. Branch and bound codes, such as the ones described in [7, 11, 12], normally use the simplex algorithm to solve linear programming subproblems that arise. In this paper, we describe an experimental branch and bound code for zero–one mixed integer linear programming problems that uses an interior point method to solve the LP subproblems.

This project was motivated by the observation that interior point methods tend to quickly find feasible solutions with good objective values, but take a relatively long time to converge to an accurate solution. For example, figure 1 shows the sequence of dual objective values generated during the solution of a sample LP by the primal–dual method. After five iterations, the solution is dual feasible and a lower bound is available. After ten iterations, the dual solution has an objective value within 2% of the optimal objective value. However, it takes eighteen iterations to solve the problem to optimality. Furthermore, the solution estimates generated by an interior point method tend to converge steadily to an optimal solution. Figure 2 shows the sequence of values of the variable x_{18} for the sample LP. After about ten iterations, it becomes apparent that x_{18} is converging to a value near 0.6.

Within a branch and bound algorithm, accurate solutions to the LP subproblems are often not needed. Any dual solution with a higher objective value than a known integer solution provides a bound sufficient to fathom the current

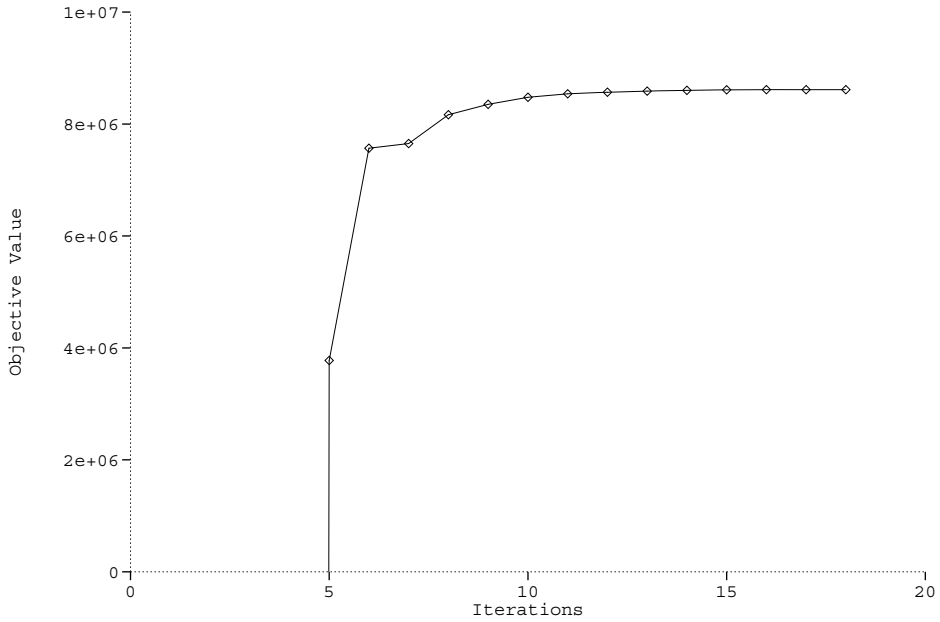

Figure 1: Lower bounds for a sample problem.

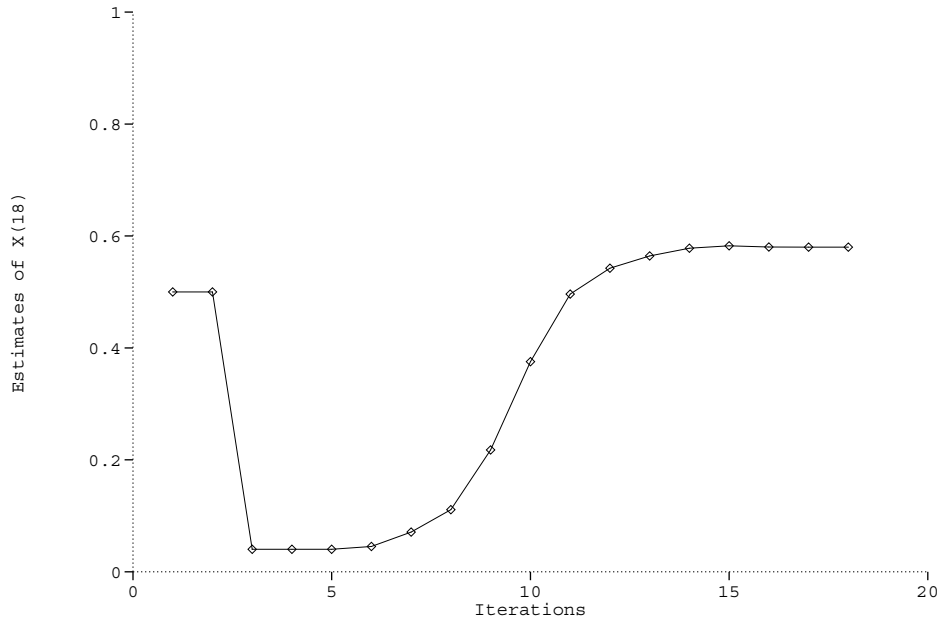

Figure 2: Estimates of x_{18}.

subproblem. Furthermore, if it becomes apparent that the optimal solution to the current subproblem includes an integer variable at a fractional value, we can branch early without solving the current subproblem to optimality. As we shall see later, this early branching significantly improves the performance of our branch and bound code.

However, the simplex method has a significant advantage over interior point methods within a branch and bound algorithm. In the branch and bound algorithm we can warm start the solution of each new subproblem with the optimal solution from the parent subproblem. Using the simplex method, a new optimal solution can usually be found after only a few simplex iterations. With interior point methods, warm starts are not as effective.

In order to determine whether the advantages of using an interior point method outweigh the advantages of using the simplex method, we have developed an experimental code that uses the primal–dual interior point method and early branching. We have tested this code on a number of sample problems and compared the performance of the code to the simplex based branch and bound code in IBM's Optimization Subroutine Library (OSL). These computational results are the first contribution of the paper. The second contribution of the paper is the early branching idea, which might be applicable to other branch and bound algorithms.

The paper is organized as follows. In section 1, we describe the interior point method that we have used to solve the subproblems. We describe the

branch and bound algorithm in section 2. We present computational results for a number of sample problems in section 3. Our conclusions are presented in section 4.

1 Solving the LP Subproblems

The experimental code solves problems of the form

$$
\begin{aligned}
(P) \qquad & \min \quad c^T x \\
& \text{subject to} \quad Ax = b \\
& \qquad\qquad\quad x \geq 0 \\
& \qquad\qquad\quad x \leq u.
\end{aligned}
$$

Some of the x variables are restricted to the values zero and one. Here A is an m by n matrix, c is an n by 1 vector, and b is a m by 1 vector. Some of the primal variables may be unbounded. For these variables, we write $u_i = \infty$. We introduce slack variables s so that the constraint $x \leq u$ can be rewritten as $x + s = u$, together with the nonnegativity constraint $s \geq 0$.

The LP dual of this problem can be written as

$$
\begin{aligned}
(D) \qquad & \max \quad b^T y - u^T w \\
& \text{subject to} \quad A^T y - w + z = c \\
& \qquad\qquad\quad w, z \geq 0.
\end{aligned}
$$

If x_i is an unbounded primal variable, then we fix w_i at zero and let $s_i = \infty$.

Within the branch and bound algorithm, we will solve a number of LP relaxations of the original problem (P), using the primal–dual method described

in [3, 14, 15]. The primal–dual method is summarized in the following algorithm.

Algorithm 1

Given an initial primal solution x, s *with* $x, s > 0$ *and* $x + s = u$, *and an initial dual solution* w, y, z, *with* $w, z > 0$, *repeat the following iteration until the convergence criterion has been met.*

1. *Compute* μ.

2. *Compute* $\Theta = (X^{-1}Z + S^{-1}W)^{-1}$ *and* $\rho(\mu) = \mu(S^{-1} - X^{-1})e - (W - Z)e$.

3. *Compute the Newton's method steps* Δx, Δy, Δs, Δw, *and* Δz *using*

$$\begin{aligned}
(A\Theta A^T)\Delta y &= (b - Ax) + A\Theta((c - A^T y - z + w) + \rho(\mu)) \\
\Delta x &= \Theta(A^T \Delta y - \rho(\mu) - (c - A^T y - z + w)) \\
\Delta s &= -\Delta x \\
\Delta z &= X^{-1}(-XZe + \mu e - Z\Delta x) \\
\Delta w &= S^{-1}(-SWe + \mu e - W\Delta s).
\end{aligned}$$

4. *Find step sizes* α_p *and* α_d *that ensure* $x + \alpha_p \Delta x > 0$, $s + \alpha_p \Delta s > 0$, $z + \alpha_d \Delta z > 0$, *and* $w + \alpha_d \Delta w > 0$. *If possible, make full steps with* $\alpha_p = 1$ *and* $\alpha_d = 1$.

5. *Let* $x = x + \alpha_p \Delta x$, $s = s + \alpha_p \Delta s$, $y = y + \alpha_d \Delta y$, $w = w + \alpha_d \Delta w$, *and* $z = z + \alpha_d \Delta z$.

For an initial solution, we use the method described in [14]. In step 2, we compute μ using the method of [3]. We consider the current solution feasible if

$$\frac{\|b - Ax\|}{1 + \|x\|} < 10^{-8}$$

and

$$\frac{\|c - A^T y + w - z\|}{1 + \|y\| + \|w\| + \|z\|} < 10^{-8}.$$

We terminate the primal–dual algorithm and declare the current solution optimal when the current solution is primal and dual feasible, and when

$$\frac{|c^T x - (b^T y - u^T w)|}{1 + |b^T y - u^T w|} < 10^{-8}.$$

This ensures that the solution is primal and dual feasible, and that the duality gap is small relative to the objective value.

The most difficult part of the computation is calculating Δy in the primal–dual step. As long as the constraint matrix A is sparse and has no dense columns, we can hope that the matrix AD^2A^T will also be sparse. The experimental codes take advantage of this sparsity by saving the matrix in sparse form and making use of routines from the Yale Sparse Matrix Package (YSMP) or IBM's Extended Scientific Subroutine Library (ESSL) to solve the systems of equations [4, 8]. We have found that the routines from YSMP are more effective on very sparse problems, while the routines from ESSL work best on relatively dense problems.

If the linear programming problem is primal or dual infeasible, then the algorithm will loop without ever finding a feasible solution. However, in our

branch and bound algorithm it is reasonable to assume that the initial LP relaxation of the problem will be primal and dual feasible. As a result, all of the other subproblems in the branch and bound tree will be at least dual feasible, and we need only detect LP subproblems which are dual unbounded and primal infeasible.

In theory, we would know that a dual subproblem was unbounded if the current dual solution was dual feasible, $\Delta w \geq 0, \Delta z \geq 0$, and $b^T \Delta y - u^T \Delta w > 0$. However, because of numerical problems in the calculation of Δw and Δz, this does not work well in practice. We have developed an alternative procedure for detecting infeasible subproblems.

After each primal–dual iteration we compute directions $\Delta \bar{w}$ and $\Delta \bar{z}$ by

$$\Delta \bar{w}_i = \begin{cases} \Delta w_i - \min(\Delta w_i, \Delta z_i) & \text{if } u_i < \infty \\ 0 & \text{otherwise} \end{cases}$$

and

$$\Delta \bar{z}_i = \begin{cases} \Delta z_i - \min(\Delta w_i, \Delta z_i) & \text{if } u_i < \infty \\ \Delta z_i & \text{otherwise} \end{cases}$$

If the previous dual solution was feasible, and if $b^T \Delta y - u^T \Delta \bar{w} > 0$, then Δy, $\Delta \bar{w}$ and $\Delta \bar{z}$ give us a direction in which the dual solution is feasible and improving. If $\Delta \bar{w}$ and $\Delta \bar{z}$ are nonnegative, then we can move as far as we wish in this improving direction, and the dual problem is unbounded.

Within the branch and bound algorithm, we will need to restart the primal–dual method after fixing a zero–one variable at zero or one. We could simply

use the last primal and dual solutions to the parent subproblem, but very small values of x_i, s_i, w_i or z_i can lead to numerical problems. Instead, we use a warm start procedure similar to the one described in [13]. If x_i or s_i is less than a tolerance ξ, then we set the variable to ξ, and adjust the slack as needed. If w_i or z_i is less than ξ, and x_i has an upper bound, then we add ξ to both w_i and z_i. This helps to retain dual feasibility. If x_i has no upper bound and z_i is less than ξ, then we add ξ to z_i. Our code uses $\xi = 0.1$.

2 The Branch and Bound Algorithm

A flowchart of our branch and bound algorithm appears in figure 3. The algorithm maintains one large data structure. This is the tree of subproblems. Each leaf node in the tree is a record containing a description of which variables have been fixed and the best known primal and dual solutions for that subproblem. The dual solution consists of $m + n$ numbers, while the primal solution consists of n numbers. (We store only x, y and w. The values of s and z can be calculated when they are needed.) In contrast, a simplex based branch and bound code would normally save only a list of the variables that had been fixed at zero or one and a listing of the variables in the current basis. Thus our algorithm uses somewhat more storage than a simplex based code.

In addition to the problem data and the branch and bound tree, the algorithm maintains two variables. The upper bound ub is the objective value for

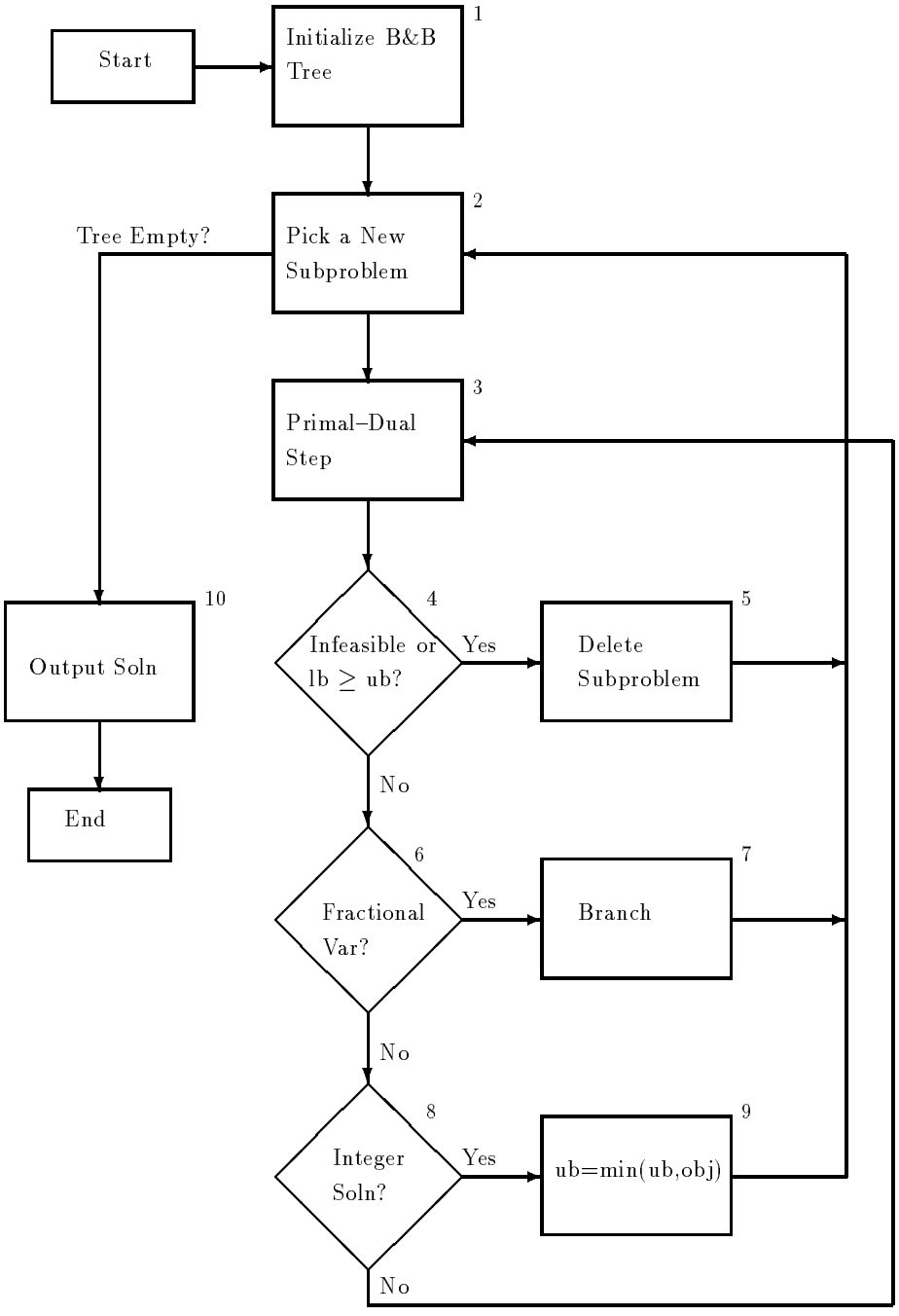

Figure 3: The branch and bound algorithm.

the best known integer solution. It is initialized to $+\infty$. For each subproblem, we obtain lower bounds, lb, which can be used to fathom a subproblem when $lb \geq ub$.

In step 2, our algorithm uses a depth first search strategy to pick subproblems until an integer solution has been found. From then on, it picks the remaining subproblem with the lowest estimated objective value. These estimates are based on pseudocosts, similar to those described in [11, 12, 16].

In step 6, we use a heuristic to determine if any of the zero–one variables appear to be converging to fractional variables. If the heuristic determines that one of the zero–one variables is approaching a fractional value, then the algorithm branches on the current subproblem to create two new subproblems. The heuristic could be wrong in a number of ways: the current subproblem could have an integer optimal solution, the current subproblem could be infeasible, or the optimal value of the current subproblem could be higher than ub, which would allow us to fathom the current subproblem. In each of these cases, the algorithm would have to consider more subproblems than if it had not branched early. However, the algorithm will eventually recover from such a mistake, since the zero–one variables will eventually all be fixed at zero or one, and at that point each subproblem will be solved to optimality, fathomed by lower bound, or shown to be infeasible.

Our heuristic is based on the work of El-Bakry, Tapia, and Zhang [5], who have shown that for a fractional variable x_i, the ratios $\frac{x_i^{k+1}}{x_i^k}$ and $\frac{s_i^{k+1}}{s_i^k}$ go to

one as the solution approaches optimality. Furthermore, the ratios $\frac{w_i^{k+1}}{w_i^k}$ and $\frac{z_i^{k+1}}{z_i^k}$ go to zero as the solution approaches optimality. Although the version of the primal–dual method used by El-Bakry, Tapia, and Zhang is slightly different from the version that we use, these ratios still provide a good heuristic indication of whether or not a variable is tending to a fractional value.

Our version of the heuristic is as follows. First, we do not search for fractional variables until we have a solution which is dual feasible and within 10% of primal feasibility. Furthermore, we do not search for fractional variables if it appears that the optimal value of the current subproblem might be large enough to fathom the subproblem by bound. We base this decision on the last two dual objective values. If $(b^T y^{k+1} - u^T w^{k+1}) + ((b^T y^{k+1} - u^T w^{k+1}) - (b^T y^k - u^T w^k)) > ub$, then we delay searching for fractional variables in hopes of fathoming the current subproblem by bound. Although this prevents the algorithm from branching as early as possible, this safeguard acts to reduce the total number of subproblems solved.

Once the above conditions are met, we declare a zero–one variable x_i fractional when it satisfies

$$\begin{aligned} |\tfrac{x_i^{k+1}}{x_i^k} - 1| &< 0.1 \\ |\tfrac{s_i^{k+1}}{s_i^k} - 1| &< 0.1 \\ \tfrac{w_i^{k+1}}{w_i^k} &< 0.6 \\ \tfrac{s_i^{k+1}}{s_i^k} &< 0.6. \end{aligned}$$

In this heuristic, we put more stress on the ratios of x and s, because we have

found that the ratios of the w and z variables do not go to zero as quickly as the ratios of the x and s variables go to one.

3 Computational Results

We tested our experimental codes on seven sets of sample problems. The first three sets of sample problems consist of randomly generated set covering problems. The fourth set of sample problems is a set of capacitated facility location problems taken from the literature [1]. Problem set five consists of two problems given to us by AT&T Bell Laboratories. Problem set six consists of the problem khb05250 from the MIPLIB collection of test problems[2]. Problem set seven consists of the problem misc06 from the MIPLIB collection of test problems.

Table I gives statistics on size of these test problems. For each problem, we give the number of rows, columns, and zero–one variables. We also give statistics on the density of the constraint matrix A, the density of $A\Theta A^T$, and the Cholesky factors of $A\Theta A^T$ after minimum degree ordering.

All computations were performed in double precision on an IBM ES/9580 under the AIX/370 operating system. The experimental codes were written in Fortran and compiled with the VS Fortran version 2 compiler and VAST2 preprocessor [9, 10]. CPU Times were measured with the CPUTIME subroutine of the Fortran run-time library. These times exclude the time required to read in the problems and to output solutions. For comparison, we also solved the

sample problems using routines from release 2 of IBM's Optimization Subroutine Library (OSL) [11].

The optimal objective function values and solutions from OSL were given to seven significant digits. The experimental codes were designed to find an optimal objective value good to about eight significant digits. In all cases, the experimental codes produced the same integer solutions as OSL. The optimal objective values generally matched to seven significant digits, although there are slight differences in the seventh digit in some cases.

For each set of problems, we first solved the LP relaxations of the problems using the OSL simplex routine EKKSSLV, the OSL primal–dual routine EKKBSLV, and our implementation of the primal–dual method, called SOLVELP. Figure 4 shows how the two implementations of the primal–dual method performed in comparison to OSL's simplex method. For the problems in problem sets one, two, and three, the primal–dual codes were competitive with EKKSSLV, while on problems sets four, five, six, and seven the EKKSSLV routine was faster. The

Problem Set	rows	columns	0–1 variables	density		
				A	$A\Theta A^T$	L
1	410	4020	10	0.49%	5.00%	5.29%
2	420	8040	20	0.48%	9.31%	9.74%
3	810	8010	10	0.25%	2.56%	2.70%
4	66	832	16	2.97%	19.9%	26.5%
5	198	1927	25	1.97%	32.5%	46.6%
6	101	1324	24	1.96%	26.9%	31.5%
7	820	2404	112	0.33%	1.15%	3.37%

Table I: Sample problem statistics.

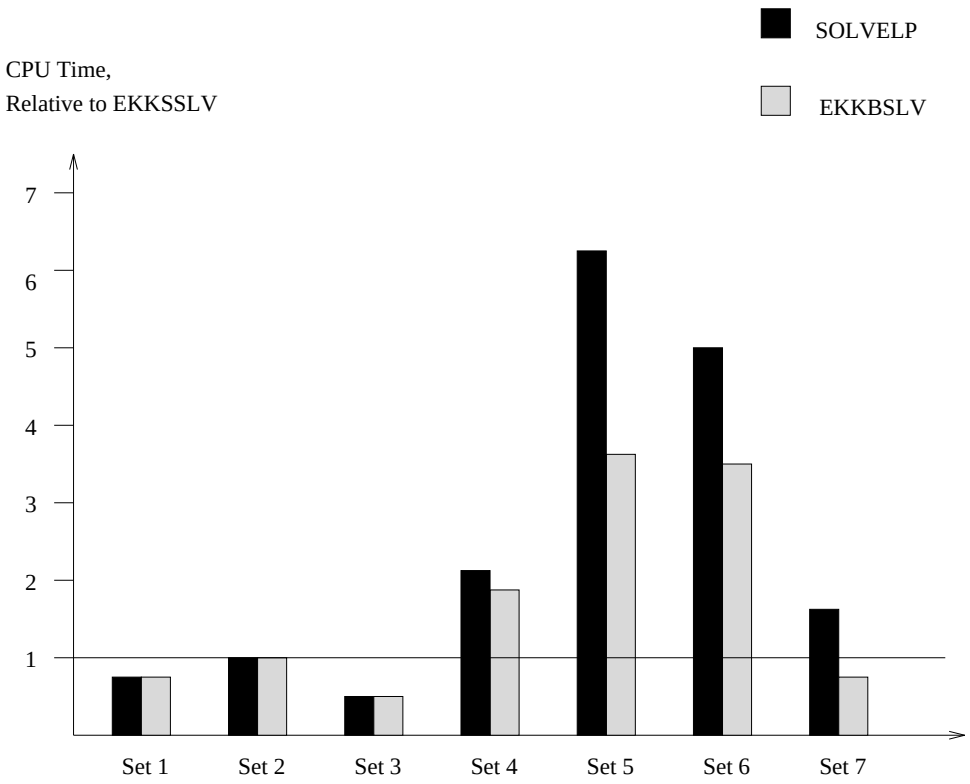

Figure 4: LP solution times, relative to OSL simplex.

OSL implementation of the primal–dual method is generally faster than our experimental implementation of the primal–dual method. This can be explained by more sophisticated sparse matrix linear algebra routines and by OSL's use of the predictor–corrector scheme [6].

In order to determine the effectiveness of the early branching heuristics, two versions of the experimental code were developed. The first version of the code, called BB, uses early branching as described in the previous section.

The second version of the code, called FULLBB, solves each subproblem to optimality without any early branching. Thus we can determine the effectiveness of the early branching heuristics.

We then solved the sample problems with the two experimental codes and with the OSL subroutine EKKMSLV. EKKMSLV is a sophisticated branch and bound code that uses the dual simplex method to solve the linear programming subproblems. Figure 5 shows the number of subproblems solved by the two versions of the experimental code, relative to the number of subproblems solved by EKKMSLV. In general, the two experimental codes solved about as many subproblems as OSL. Furthermore, the version of the code with early branching did not solve more subproblems than the version of the code without early branching. This indicates that the use of early branching did not significantly increase the number of subproblems solved.

Problem seven is unusual. Although this problem had over a hundred zero-one variables than any of the other test problems, EKKMSLV was able to solve the problem with only 228 LP subproblems. The BB code solved 309 subproblems, while the FULLBB code solved 637 subproblems. For this problem, it appears that our procedure for selecting subproblems to be solved was not as effective as OSL's.

Early branching was able to reduce the number of iterations per subproblem used by the experimental code. Figure 6 shows the number of iterations per subproblem for the two experimental codes. In each case, the BB code, with

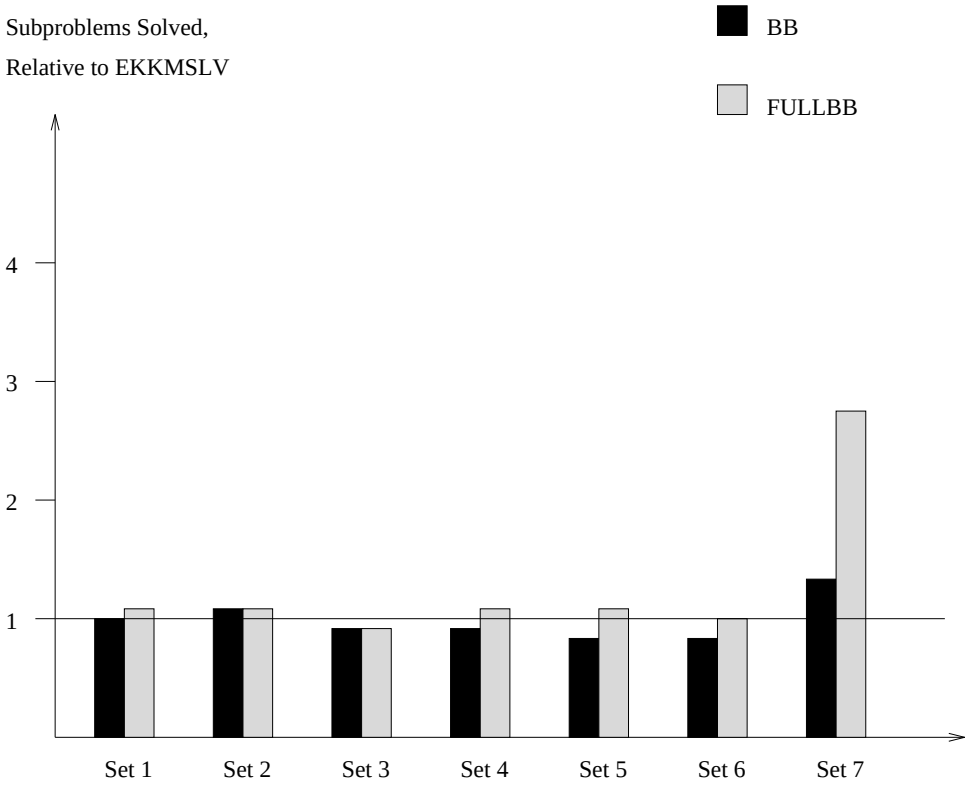

Figure 5: Subproblems solved, relative to EKKMSLV.

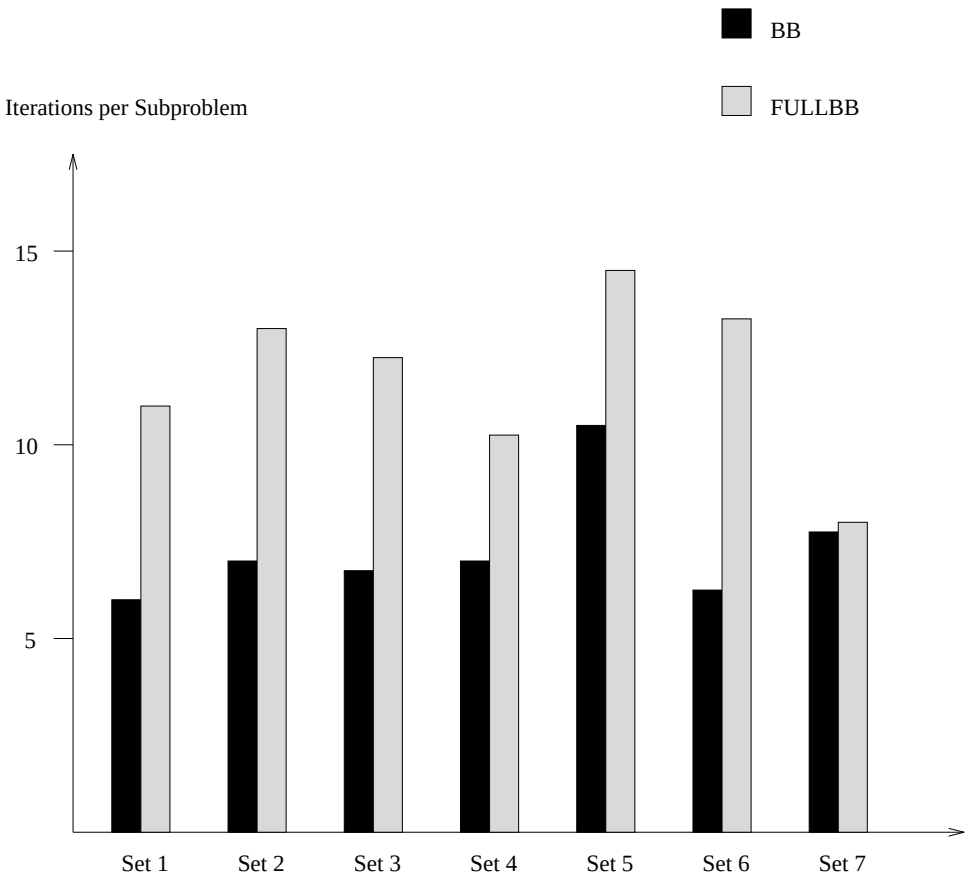

Figure 6: Iterations per Subproblem.

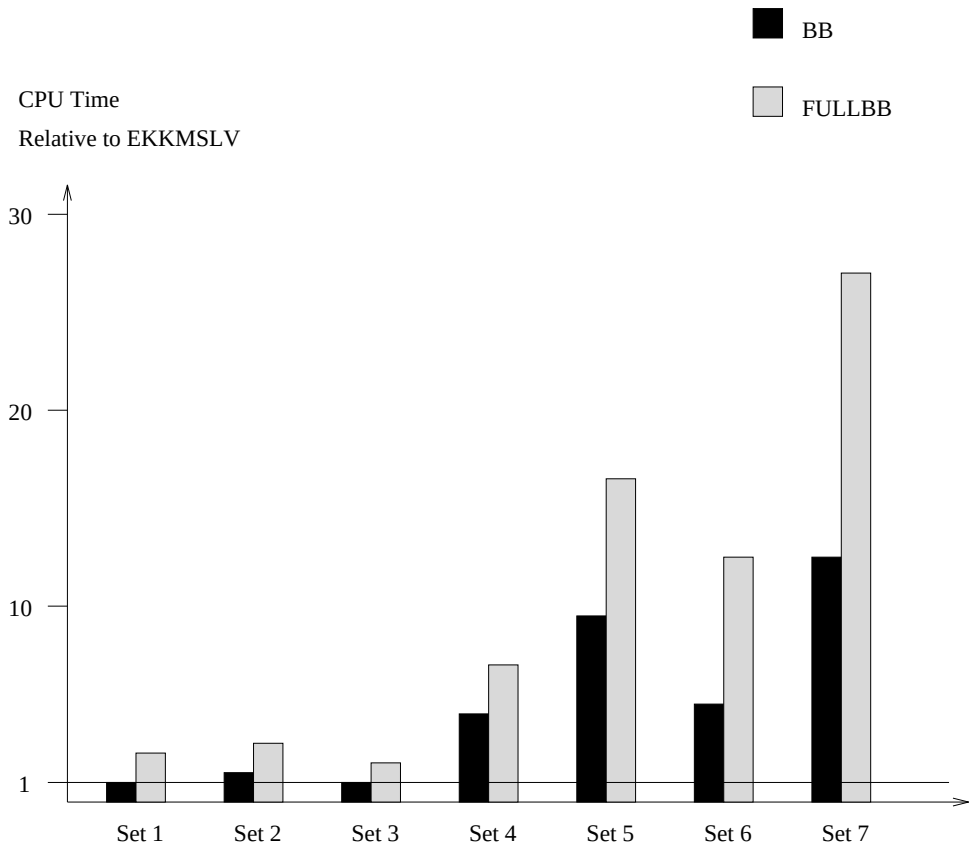

Figure 7: CPU times, relative to EKKMSLV.

early branching, used fewer iterations per subproblem. In most cases, the BB code used significantly fewer iterations per subproblem than FULLBB.

Figure 7 shows the total CPU time used by the experimental codes, relative to the CPU time used by EKKMSLV. On problem sets one, two, and three, the experimental code BB is reasonably competitive with EKKMSLV. On problem sets four, five, six, and seven, OSL dominates. However, OSL's simplex method was signficantly faster than our primal–dual codes in solving the LP relaxations of these problems, and it is not surprising that the simplex based branch and bound code is superior.

4 Conclusions

The experimental code was not competitive with OSL's simplex based branch and bound procedure on most of the test problems. Since the experimental code and EKKMSLV generally solved about the same number of subproblems, it appears that the advantage of warm starting the simplex method in EKKMSLV outweighed the advantages of early branching in the experimental code.

However, there are a number of ways in which the experimental code could be improved. First, the implementation of the primal–dual method could be speeded up. In some cases, our primal–dual LP solver took twice as long to solve the LP relaxation of a problem as OSL's implementation of the primal–dual method. Second, a better procedure for warm starting the primal–dual

method could be developed. This is an area in which very little work has been done to date. Third, improvements in the heuristics for detecting fractional variables might be possible. Fourth, the experimental code is lacking in support for special ordered sets, implicit enumeration, and other techniques that can speed up the branch and bound algorithm.

Furthermore, our computational testing has been limited to relatively small problems. The largest test problem that we have solved has fewer than 1000 rows. Since interior point methods for linear programming are generally more effective on larger problems, it is possible that a branch and bound code using an interior point method would be superior to a conventional branch and bound code on larger problems.

Although the computational results in this paper are not immediately encouraging, we feel that progress in interior point methods for linear programming or the need to solve larger mixed integer programming problems will make further research worth while.

The second contribution of this paper was the early branching technique. In most cases, the code with early branching solved about the same number of subproblems as the code without early branching. At the same time, the code with early branching used significantly fewer iterations per subproblem. As a result, the experimental code with early branching was significantly faster than the code without early branching. The early branching idea might also be applicable to branch and bound algorithms for other problems, such as mixed

integer nonlinear programming problems.

Acknowledgements

This research was partially supported by ONR Grant number N00014–90–J–1714.

References

[1] Umit Akinc and Basheer M. Khumawala, 1977. *An Efficient Branch and Bound Algorithm for the Capacitated Warehouse Location Problem*, **Management Science 23:6**, 585–594.

[2] Robert E. Bixby, E. Andrew Boyd, and Ronni R. Indovina, 1992. *MIPLIB: A Test Set of Mixed Integer Programming Problems*, **SIAM News 25:20**, 16.

[3] In Chan Choi, Clyde L. Monma, and David F. Shanno, 1990. *Further Developments of a Primal-Dual Interior Point Method*, **ORSA Journal on Computing 2:4**, 304–311.

[4] S. C. Eisenstat, M.C. Gurshy, M.H. Schultz, and A.H. Sherman, 1982. *The Yale Sparse Matrix Package, I. The Symmetric Codes*, **International Journal for Numerical Methods in Engineering 18**, 1145–1151.

[5] A. S. El-Bakry, R. A. Tapia, and Y. Zhang, 1991. *A Study of Indicators for Identifying Zero Variables in Interior-Point Methods*, Technical Report TR91-15, Rice University.

[6] J. J. H. Forrest and J. A. Tomlin, 1992. *Implementing Interior Point Linear Programming Methods in the Optimization Subroutine Library*, **IBM Systems Journal 31:1**, 26–38.

[7] A. M. Geoffrion and R. E. Marsten, 1972. *Integer Programming Algorithms: A Framework and State-of-the-Art Survey*, **Management Science 18**, 465–491.

[8] IBM, 1988. *Engineering and Scientific Subroutine Subroutine Library Guide and Reference*, Kingston, NY.

[9] IBM, 1990. *VAST-2 for VS Fortran: User's Guide*, Kingston, NY.

[10] IBM, 1990. *VS Fortran Version 2, Language and Library Reference, Release 5*, Kingston, NY.

[11] IBM, 1991. *IBM Optimization Subroutine Library Guide and Reference*, Kingston, NY, third edition.

[12] A. Land and S. Powell, 1979. *Computer Codes for Problems of Integer Programming*, In P. L. Hammer, E. J. Johnson, and B. H. Korte, editors, **Discrete Optimization II**, Annals of Discrete Mathematics. North Holland, New York.

[13] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno, 1990. *Starting and Restarting the Primal-Dual Interior Point Method*, Technical Report SOR 90-14, Princeton University.

[14] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno, 1991. *Computational Experience With a Primal–Dual Interior Point Method for Linear Programming*, **Linear Algebra and Its Applications 152**, 191–222.

[15] Kevin A. McShane, Clyde L. Monma, and David Shanno, 1989. *An Implementation of a Primal-Dual Interior Method for linear Programming*, **ORSA Journal on Computing 1:2**, 70–83.

[16] R. Gary Parker and Ronald L. Rardin, 1988. *Discrete Optimization*, Academic Press, Boston.