INFORMS Journal on Computing
Vol. 11, No. 2, Spring 1999

0899-1499/99/1102-0173 $05.00

A Computational Study of Search Strategies for Mixed Integer Programming

J. T. LINDEROTH AND M. W. P. SAVELSBERGH / *School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0205, Email: jeff@akula.isye.gatech.edu;mwps@isye.gatech.edu*

(Received: October 1997; revised: February 1998, April 1998, May 1998; accepted: September 1998)

The branch-and-bound procedure for solving mixed integer programming (MIP) problems using linear programming relaxations has been used with great success for decades. Over the years, a variety of researchers have studied ways of making the basic algorithm more effective. Breakthroughs in the fields of computer hardware, computer software, and mathematics have led to increasing success at solving larger and larger MIP instances. The goal of this article is to survey many of the results regarding branch-and-bound search strategies and evaluate them again in light of the other advances that have taken place over the years. In addition, novel search strategies are presented and shown to often perform better than those currently used in practice.

The effectiveness of the branch-and-bound procedure for solving mixed integer programming (MIP) problems using linear programming relaxations is well documented. After the introduction of this procedure in the 1960s,[14, 10, 26] researchers in the 1970s examined strategies for searching the branch-and-bound tree in an efficient manner.[15, 7, 14–16, 29] With a few exceptions (notably Nygreen[31] and Hajian and Mitra[19]), research in past decades has strayed from developing and examining effective search techniques and instead focused on improving the bound obtained by the linear programming relaxation.[9, 34, 35] The goal of this paper is to reexamine search techniques in light of the many advances made in the field over the years.

One major change over the past decades is in the hardware on which MIPs are solved. Computers of today are orders of magnitudes faster than their counterparts of yesteryear, allowing us to expend more computational effort in the solution of a MIP instance. Furthermore, computers of today are equipped with a great deal more memory than in the past, so a search strategy designed to limit memory size may not be of much use today.

A second change since the 1970s is in the theory behind solving MIPs. In the 1980s, the usefulness of combining cutting planes and branch-and-bound was demonstrated. Nearly all industrial strength MIP codes now contain some sort of cut generation in addition to branch-and-bound. Dramatic improvements in the simplex algorithm have also been made that allow for fast reoptimization of an LP, regardless of the change in LP-relaxation from one node to the next. Also, many MIP codes now use a heuristic method to attempt to find feasible solutions. Each of these practical and theoretical improvements impacts the effectiveness of a particular strategy for searching the branch-and-bound tree.

What advances in the future will allow researchers to solve larger MIP instances? Undoubtedly, theoretical advancements will continue to have a major impact. Another major impact will come from implementing the current branch-and-bound algorithms on parallel computers. In a wide range of fields, the introduction of parallel computers consisting of many microprocessors has made possible the solutions of problems impossible to consider solving on a single processor. Many researchers have studied the effects, both good and bad, of dividing the search space for an optimization problem among many processors.[23, 24, 38] Key to achieving an effective algorithm on parallel computers is an effective means for exploring the search space. Further, in a parallel environment, information found on one processor may be useful in guiding the search on a different processor. Sharing information on a distributed memory architecture requires that a message be passed, for which some overhead is incurred. It therefore is useful to know what types of information are especially useful for guiding the search and how much of this information must be gathered and shared. This is yet another motivation for restudying search strategies for MIP.

The purpose of this article is two-fold—to survey existing ideas for searching the branch-and-bound tree and to offer some new techniques for performing the search. Extensive computational experiments will be presented comparing the old and new methods. We begin by briefly reviewing the branch-and-bound algorithm for solving a mixed integer program using linear programming relaxations. In Section 2, we discuss and compare various "branching rules," or methods of subdividing the search space. In Section 3 different "node selection schemes," or search-ordering methods, are presented and compared. Finally, in Section 4 we draw some conclusions from our work and give some directions for future research.

1. LP-Based Branch-and-Bound

A mixed integer program (MIP) can be stated mathematically as follows:

$$\text{Maximize } z_{MIP} = \sum_{j \in I} c_j x_j + \sum_{j \in C} c_j x_j$$

Subject classifications: Computational Programming
Other key words: Computational algorithms, branch-and-bound

$$\text{subject to} \quad \sum_{j \in I} a_{ij} x_j + \sum_{j \in C} a_{ij} x_j \leq b_i \quad i = 1, \ldots m$$
$$l_j \leq x_j \leq u_j \qquad j \in N$$
$$x_j \in \mathbb{Z} \qquad j \in I$$
$$x_j \in \mathbb{R} \qquad j \in C,$$

where I is the set of integer variables, C is the set of continuous variables, and $N = I \cup C$. The lower and upper bounds l_j and u_j may take on the values of plus or minus infinity.

The term "branch-and-bound" was originally coined by Little et al.[28] in their study of such an algorithm to solve the traveling salesman problem. However, the idea of using a branch-and-bound algorithm for integer programming using linear programming relaxations was proposed somewhat earlier by Land and Doig.[26] The process involves keeping a list of linear programming problems obtained by relaxing some or all of the integer requirements on the variables x_j, $j \in I$. To precisely define the algorithm, let us make a few definitions. We use the term *node* or *subproblem* to denote the problem associated with a certain portion of the feasible region of MIP. Define z_L to be a lower bound on the value of z_{MIP}. For a node N^i, let z_U^i be an upper bound on the value that z_{MIP} can have in N^i. The list $\mathcal{L}$ of problems that must still be solved is called the *active set*. Denote the optimal solution by x^*. Algorithm 1 is an LP-based branch-and-bound algorithm for solving MIP.

Algorithm 1 The Branch-and-Bound Algorithm

0. **Initialize.**
   $\mathcal{L} = \text{MIP}$. $z_L = -\infty$. $x^* = \emptyset$.
1. **Terminate?**
   Is $\mathcal{L} = \emptyset$? If so, the solution x^* is optimal.
2. **Select.**
   Choose and delete a problem N^i from $\mathcal{L}$.
3. **Evaluate.**
   Solve the LP relaxation of N^i. If the problem is infeasible, go to Step 1, else let z_{LP}^i be its objective function value and x^i be its solution.
4. **Prune.**
   If $z_{LP}^i \leq z_L$, go to Step 1. If x^i is fractional, go to Step 5, else let $z_L = z_{LP}^i$, $x^* = x^i$, and delete from $\mathcal{L}$ all problems with $z_U^i \leq z_L$. Go to Step 1.
5. **Divide.**
   Divide the feasible region of N^i into a number of smaller feasible regions $N^{i1}, N^{i2}, \ldots, N^{ik}$ such that $\cup_{j=1}^{k} N^{ij} = N^i$. For each $j = 1, 2, \ldots, k$, let $z_U^{ij} = z_{LP}^i$ and add the problem N^{ij} to $\mathcal{L}$. Go to Step 1.

The description makes it clear that there are various choices to be made during the course of the algorithm, namely, how do we select which subproblem to evaluate, and how do we divide the feasible region. A partial answer to these two questions will be provided in the next two sections. When answering these questions, our main focus will be to build *robust* strategies that work well on a wide variety of problems.

2. Problem Division

Since our algorithm is based on linear programming relaxations, we need to enforce linear constraints to divide the current feasible region as required in Step 5 of Algorithm 1. However, adding constraints to the linear programming formulation leads to the undesirable result of increasing the size of the linear program that must be solved as we progress. A more efficient way to divide the region is to change the bounds on variables. In the following sections, we present the two most common (and obvious) schemes for partitioning the feasible region by changing variables' bounds.

2.1 Variable Dichotomy

For MIP, the obvious way to divide the feasible region of N^i is to choose a variable j that has fractional value x_j^i in the solution to the LP relaxation of N^i and impose the new bounds of $x_j \leq \lfloor x_j^i \rfloor$ to define one subregion and $x_j \geq \lceil x_j^i \rceil$ to define another subregion. We call this branching on a *variable dichotomy*, or simply branching on a variable.

If there are many fractional variables in the solution to the LP relaxation of N^i, we must choose one variable to define the division. Because the effectiveness of the branch-and-bound method strongly depends on how quickly the upper and lower bounds converge, we would like to branch on a variable that will improve these bounds. It seems difficult to select a branching variable that will affect the lower bound. Doing this amounts to heuristically finding feasible solutions to MIP and is very problem specific. However, there are ways to attempt to predict which fractional variables will most improve the upper bound when required to be integral. These prediction methods fall into two general categories. The first category includes methods that attempt to estimate the change (or degradation) of the objective function value and the second category includes those that provide a lower bound on the degradation of the objective function value.

2.1.1 Estimation Methods

Estimation methods work as follows: with each integer variable x_j, we associate two quantities P_j^- and P_j^+ that attempt to measure the per unit change in objective function value if we fix x_j to its rounded down value and rounded up value, respectively. Suppose that $x_j^i = \lfloor x_j^i \rfloor + f_j^i$, with $f_j^i > 0$. Then by branching on x_j, we will estimate a change of $D_j^{i-} = P_j^- f_j^i$ on the down branch of node i and a change of $D_j^{i+} = P_j^+ (1 - f_j^i)$ on the up branch of node i. Bénichou et al.[5] call the values P_j^- and P_j^+ *down* and *up pseudocosts*.

One way to obtain values for P_j^- and P_j^+ is to simply use the observed degradation in objective function value. Let N^{i-} and N^{i+} denote the nodes for the down and up branches of node N^i, then compute the pseudocosts as

$$P_j^- = \frac{z_{LP}^{i-} - z_{LP}^i}{f_j^i} \text{ and } P_j^+ = \frac{z_{LP}^{i+} - z_{LP}^i}{1 - f_j^i}.$$

Two more questions need to be answered before we can implement a branching scheme based on pseudocosts:

- To what value should the pseudocosts be initialized?
- How should the pseudocosts be updated from one branch to the next?

Table I. MIPLIB Instances

Name	Rows	Columns	No. of Integer Variable	No. of Binary Variables	No. of Continuous Variables
air04	823	10757	10757	ALL	0
arki001	1048	1388	538	415	850
bell3a	123	133	71	39	62
bell5	91	104	58	30	46
gesa2	1392	1224	408	240	672
harp2	112	2993	2993	ALL	0
l152lav	97	1989	1989	ALL	0
mod011	4480	10958	96	ALL	0
pp08a	136	240	64	ALL	176
qiu	1192	840	48	ALL	792
qnet1	503	1541	1417	1288	124
rgn	24	180	100	ALL	80
stein45	331	45	45	ALL	0
vpm2	234	378	168	ALL	210

The question of initialization is an important one. By the very nature of the branch-and-bound process, the branching decisions made at the top of the tree are the most crucial. As pointed out by Forrest et al.,[14] if at the root node we branch on a variable that has little or no effect on the LP solution at subsequent nodes, we have essentially doubled the total amount of work required.

An obvious method for initialization is to simply let the pseudocosts for a variable be the value of its objective function coefficient, since if a variable were unrelated to the other variables in the problem, its pseudocost would be precisely its objective function coefficient.

Bénichou et al.[5] and Gauthier and Ribière[15] experimented with explicitly computing the pseudocosts of each variable. They report that doing so can be effective in reducing the number of nodes evaluated, but that the time spent computing these values explicitly is significant. In fact, Gauthier and Ribière conclude that the computational effort is too significant compared to the benefit obtained.

Even though today's situation may be slightly different, due to faster computers and better LP solvers, it clearly indicates that care has to be taken when pseudocosts are explicitly computed for each variable. In doing our experiments, we noted that often only a small percentage of the integer variables are ever fractional in the solution to a linear programming relaxation and an even smaller percentage of integer variables are ever branched on. Therefore, if pseudocosts are going to be explicitly computed, then they should be computed only for the fractional variables as needed.

Yet another alternative, suggested by Eckstein,[13] keeps track of the average value of the pseudocost on the up and down branches. For each variable that has yet to be arbitrated, the pseudocosts are set to these average values. This method has the disadvantage that variables not yet branched on are ranked in importance only by how fractional they are.

In the course of the solution process, the variable x_j may be branched on many times. How should the pseudocosts be updated from one branch to the next? Bénichou et al.[5] state that the pseudocosts vary little throughout the branch-and-bound tree, and suggest fixing P_j^- and P_j^+ to the values observed the first time when variable x_j is branched on. Alternatively, as suggested in Forrest et al.,[14] one could also fix the pseudocosts to the values obtained from the last time when x_j was branched on. Forrest et al.[14] and Eckstein[13] suggest averaging the values from all x_j branches.

We performed an experiment to verify the observations of Bénichou et al.,[5] i.e., that the pseudocosts are relatively constant throughout the branch-and-bound tree. Suppose the (either up or down) pseudocost P_j is some linear function of the number of times N_j variable x_j is branched on. We can express this relationship as

$$P_j = \beta_0 + \beta_1 N_j .$$

For the set of instances shown in Table I (taken from MIPLIB[6]), we explicitly computed the regression coefficients β_0 and β_1 for each variable and direction on which we chose to branch more than seven times. This gave us 693 variable-direction pairs. For these 693, zero was in the 95% confidence interval of the regression coefficient β_1 673 times, which would imply there was statistical reason to believe that the pseudocosts are constant throughout the branch-and-bound tree for these variable-direction pairs. However, we also observed that from node to node, pseudocosts can vary significantly. In Figure 1, we plot the observed pseudocosts as a function of the number of times we branch on a specific variable. Therefore, we believe that updating the pseudocosts by averaging the observations should be the most effective.

The issues of initialization of pseudocosts and updating of pseudocosts are unrelated. Generally once a variable is

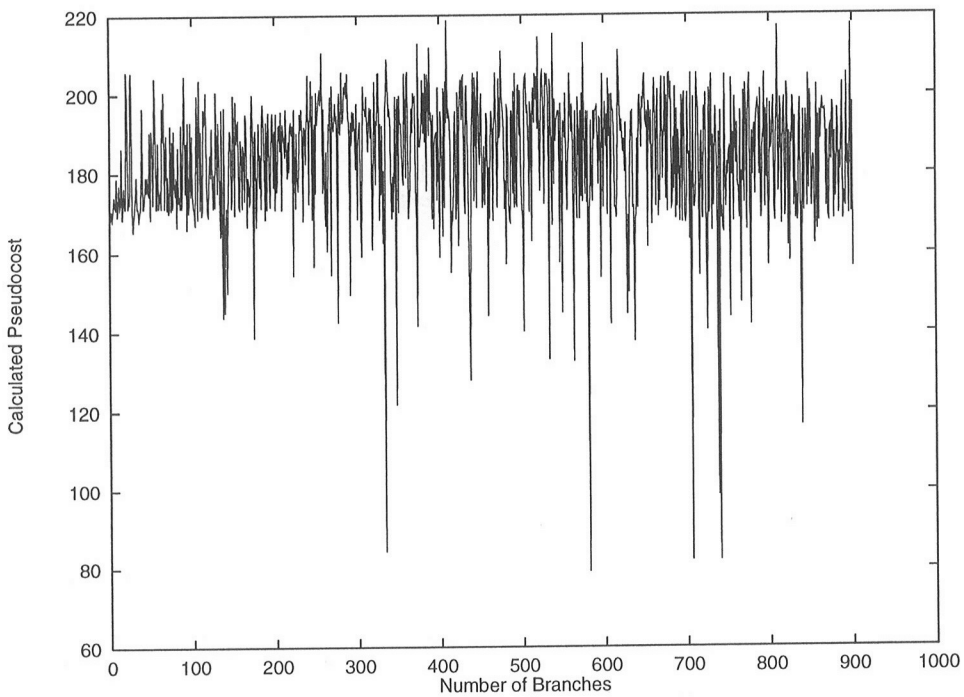

Figure 1. Observed pseudocosts as a function of number of branches; problem pp08a, variable 219.

branched on, the initial pseudocost value is discarded and replaced by the true (observed) pseudocost. Therefore, we will deal with the initialization and update issues separately.

Throughout the course of this article, we will be introducing experiments aimed at establishing the effectiveness of different search strategy techniques for mixed integer programming. The techniques under investigation were incorporated into the mixed integer optimizer MINTO (v2.0).[30] Since one of the main focuses of this article is to establish how search strategies interact with new, sophisticated, integer programming techniques, we have used the advanced features MINTO has to offer in performing the experiments. These features include preprocessing and probing, automatic cut generation, reduced cost fixing, and row management. To solve the linear programs that arise, we have used CPLEX v4.0.[8]

Many of our branching experiments are aimed at determining the best way in which to perform a particular aspect of a specific branching rule. For experiments of this type, we have limited the test instances to a suite of 14 problems from the newest version of MIPLIB.[6] The instances were chosen more or less at random, but exhibit a wide range of problem characteristics. Table I shows the instances in the test suite. This is admittedly a small sample of problems, and the reader should be cautioned against drawing any sweeping conclusions about the effectiveness of a particular method based on such a small sample. However, we use experiments based on this small test suite only to guide us as to the best manner in which to perform one particular aspect of a specific branching rule, and we hope that results from these experiments show trends consistent with our intuition. In Section 2.1.6, where we compare the effectiveness of various branching rules, we use a much larger test set for our experiment.

Table II. Characteristics of All Computational Experiments

- Code compiled with the IBM XLC compiler, optimization level −O2.
- Code run on an RS/6000 Model 590.
- CPU time limited to one hour.
- Memory limited to 100 mb.

Unless otherwise noted, all experiments were run with the settings shown in Table II. Other characteristics about the experiments will be mentioned as needed.

We now describe an experiment that aims at establishing the best pseudocost initialization method. Since determining the best initialization method is our goal here, we have fixed the updating method in these runs to be the averaging suggestion. We branch on the variable x_j for which $D_j^- + D_j^+$ is the largest. This choice will be discussed in more detail in Section 2.1.4. As previously stated, the main focus of choosing a branching variable is to choose one that will most improve the upper bound of the child nodes from the parent. By setting z_I to the value of the optimal solution to the problem in our computational experiments, we minimize factors other than branching that determine the size of the branch-and-bound tree. Just for completeness, we men-

Table III. Summary of Pseudocost Initialization Experiment

Initialization Method	Avg. Ranking
Obj. Coef.	2.93
Averaged	3.07
Computed All	2.50
Computed Fractional	1.50

tion that we use the "best bound" node selection rule, where at the **Select** portion of the branch-and-bound algorithm, we choose to evaluate the node $N^i \in \mathcal{L}$ with the largest value of z_{lt}^i.

Tables detailing the results for each instance in the experiments were too long to be included here and can be found in Linderoth and Savelsbergh.[27] Instead, we will be including summary tables that rank the performance of the techniques under investigation. We rank techniques related to branching methods as follows:

- A method that proves the optimality of the solution is ranked higher than one that does not.
- If two methods prove the optimality of the solution, the one with shorter computation time is ranked higher.
- If two methods do not prove the optimality of the solution, the one with smaller final gap is ranked higher.
- Ties are allowed.

Table III gives a summary of the results of our experiment comparing different pseudocost initialization methods. The pseudocosts were initialized with objective function coefficients, by averaging observations, by explicitly computing them for all variables at the root node, and explicitly computing them for the fractional variables only as needed.

Examination of the results shows that explicitly computing initial pseudocosts for fractional variables as needed is clearly the best method. This result is different than the conclusion reached by Gauthier and Ribière.[15] The faster simplex algorithm and computers of today now make it possible to invest more effort into the (often very important) initial branching decisions.

We conclude that a good pseudocost initialization strategy should allow for initially explicitly computing pseudocosts, take care not to expend too much computation time accomplishing this task, and allow for spending more time computing explicit pseudocosts at the top of the branch-and-bound tree where branching decisions are more crucial. After further experimentation, we adopted the following pseudocost initialization strategy. Let T be the maximum amount of time per node we are willing to spend to initialize pseudocosts for variables on which we have yet to branch. In this time, we wish to gain useful branching information on all fractional variables. We therefore impose a limit L on the number of simplex iterations used in solving the linear program necessary to compute one particular pseudocost. Let γ be an estimate of the number of simplex iterations performed per unit time, obtained by

$$\gamma \equiv \frac{\text{Number of iterations needed to solve the initial LP}}{\text{Time to solve the initial LP}}.$$

Let η be the number of fractional variables in initial LP solution. Then we compute L as

$$L = \frac{T\gamma}{2\eta}.$$

As we develop the branch-and-bound tree, if there is a fractional variable x_j upon which we have never branched, we perform L simplex pivots after fixing the bounds of this variable to $\lfloor x_j \rfloor$ and $\lceil x_j \rceil$ in order to explicitly determine the pseudocosts.

Gauthier and Ribière[15] also proposed a pseudocost initialization strategy that uses a limited number of simplex iterations, but our approach is fundamentally different. They purposely limit the number of simplex iterations to a small number, while we set T to a large number hoping to be able to compute a "true" pseudocost. T is two minutes in the current implementation.

We now turn our attention to the question of how to update the pseudocosts from one branch to the next. As mentioned above, our initial experiments lead us to believe that updating the pseudocosts by averaging the observations would be the most computationally effective.

We empirically verified this conjecture by solving the instances of MIPLIB in Table I where the pseudocosts were updated by averaging the observations, setting the pseudocost to the first observed value, and setting the pseudocost to the last observed value. For these runs, we have initialized the pseudocosts by our strategy that explicitly computes them, with a limit on the number of iterations used. As in our previous experiment, we branch on the variable x_j for which $D_j^{i-} + D_j^{i+}$ is the largest, set z_l to be the known optimal solution to the problem, and we use the best bound node selection rule. Table IV shows the average ranking for the different pseudocost update methods over all of the instances. For the full results of the experiment, see Linderoth and Savelsbergh.[27]

Table IV. Summary of Pseudocost Update Experiment

Update Method	Avg. Ranking
First	2.43
Last	1.64
Average	1.43

From the results of the experiment we see that our intuition is correct. For the most part, it seems to be best to average the degradations when branching on a variable to determine its pseudocosts, and the overhead necessary to perform the averaging does not outweigh its benefits.

In distributed memory parallel computer architectures, the overhead necessary to perform true averaging of pseudocosts increases dramatically, since different processors calculate different nodes (and hence different pseudo-

costs). The fact that the "Last" updating technique is not significantly outperformed by the "Average" updating technique leads us to believe that true averaging may not be necessary for computational effectiveness on parallel architectures.

For the remainder of this article, when we refer to *pseudocost branching*, this will imply that we have used our strategy of explicitly computing the initial pseudocosts with a simplex iteration limit, and we update the pseudocosts by averaging the observations.

2.1.2 Lower Bounding Methods

Branching strategies that provide a lower bound on the objective function degradation work much the same way as estimation strategies. For each variable, we find quantities L_j^- and L_j^+ that provide lower bounds on the decrease in objective function value if we branch down and up on variable x_j. This idea originated with the work of Dakin,[10] Healy,[21] and Davis et al.[11] Driebeek[12] shows how to compute values for L_j^- and L_j^+ by implicitly performing one dual simplex pivot. Breu and Burdet provide computational evidence that lower bounding methods can be beneficial,[7] however branching methods based on simple lower bound calculations have fallen out of favor in recent years.[2, 14] As Beale states, "the fact remains that practical problems tend to have several nonbasic variables with zero reduced costs, when these methods are largely useless."[2]

Over the years since Beale made this statement, much work has been done in the area of generating strong valid inequalities for MIP and incorporating these inequalities into a branch and cut scheme.[9, 17, 32, 34] With the advent of these sophisticated cutting planes, one may suspect to have fewer nonbasic variables with zero reduced cost. The rationale for this statement is as follows. Non-basic variables with zero reduced cost correspond to alternative optimal solutions to the linear programming relaxation. Cutting planes meant to separate a fractional LP solution from the convex hull of integer points may also separate alternative LP-optimal solutions. Figure 2 graphically depicts this phenomenon in two dimensions. We also empirically verified this observation by determining the percentage of nodes where a non-zero lower bound on the degradation is found when both cuts are added and not added to the formulation. We used the lower bound obtained from implicitly performing one dual simplex pivot. Table V shows the percentage of the first 250 nodes of the branch-and-bound tree that have useful dual estimates when both cuts are added and not added to the formulation. This experiment was run on all the problems of MIPLIB, but only the instances where the difference in percentage is greater than 4% are reported. The results indicate that somewhat more non-zero estimates are obtained with cutting planes, but there are no dramatic improvements.

One may make the argument that since rows are being added to the linear programming formulation, it is likely that a greater number of dual simplex pivots are required for the child node to satisfy the bound we have imposed. Hence, the lower bound obtained by implicitly performing one dual simplex pivot bears less relation to the true degradation.

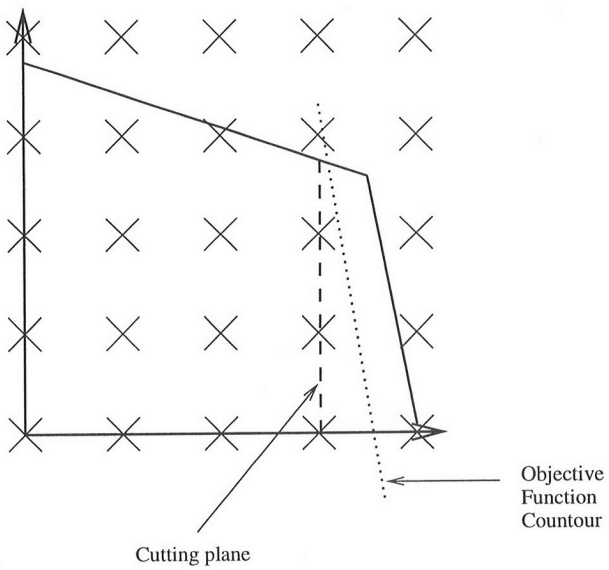

Figure 2. A valid inequality that cuts off alternative optimal solutions.

However, we will show that the lower bounds obtained in this fashion can be useful in determining a branching variable.

Instead of implicitly performing one dual simplex pivot, a number of actual dual simplex pivots can be performed for each variable. The change in objective function again provides a lower bound on the true degradation. A strategy similar to this is what Bixby[1] calls *strong branching*. Strong branching selects a set of "good" variables on which one may choose to branch and performs a number of dual simplex pivots on each variable in this set. Strong branching has been shown to be an effective branching rule for large set partitioning problems, traveling salesman problems, and some general integer programs.[1, 8] The idea of performing multiple dual simplex pivots is closely related to the pseudocost initialization idea we propose. However, there are two main differences. For pseudocost initialization, the dual simplex pivots are performed only once for each variable, regardless of how many times the variable was branched on. Further, for pseudocost initialization, a large number of pivots are performed in the hope of completely resolving the problem.

Tomlin[37] has shown that Driebeek's lower bounding method, i.e., implicitly performing one dual simplex pivot, can be improved by taking into account the integrality of the nonbasic variables. In that case, the quantities L_j^- and L_j^+ no longer provide lower bounds on the degradation of the objective function with respect to the optimal value of the LP relaxation at the child node, but on the degradation of the objective function with respect to the optimal value of the IP at the child node. Tomlin also observed that these lower bounds are dominated by bounds derived from Gomory cuts.

Recently, Günlük[18] has proposed another extension of Driebeek's lower bounding method, which he calls *knapsack*

Table V. Percentage of Useful Lower Bound Estimates

Problem	Cuts?	Percent of Nodes With Useful Estimates
danoint	Y	98.40
danoint	N	94.00
enigma	Y	5.13
enigma	N	23.00
lseu	Y	100.00
lseu	N	92.00
misc03	Y	66.45
misc03	N	90.69
misc07	Y	78.05
misc07	N	92.62
mod010	Y	100.00
mod010	N	65.79
p0201	Y	81.20
p0201	N	66.00
p2756	Y	80.00
p2756	N	66.25
pk1	Y	38.40
pk1	N	44.40
rentacar	Y	100.00
rentacar	N	0.00
rgn	Y	24.56
rgn	N	0.85
rout	Y	88.71
rout	N	71.08
vpm1	Y	9.38
vpm1	N	100.00

branching. Knapsack branching is similar to the penalty improvement idea of Tomlin,[37] since both methods take into account the integrality of the non-basic variables. However, knapsack branching takes the integrality into account in a more involved way. In order to determine the value L_j^- or L_j^+, a knapsack problem must be solved. Günlük suggests a method for solving these knapsack problems efficiently.

We solved the MIPLIB instances in Table I using the various lower bounding methods in order to compare performance. The lower bounding branching methods we chose to compare were to perform one dual simplex pivot on all fractional variables, 10 dual simplex pivots on all fractional variables, 25 dual simplex pivots on a subset of the fractional variables (i.e., strong branching), and knapsack branching. To determine the subset $I' \subseteq I$ of variables on which to perform 25 dual simplex pivots, we used the following strategy. Given a fractional LP solution, let $L = \max\{f_j : f_j \leq 0.5, j \in I\}$ and $U = \min\{f_j : f_j \geq 0.5, j \in I\}$. We chose

$$I' \equiv \{j \in I : 0.8L \leq f_j \leq U + 0.2(1 - U)\}. \quad (1)$$

For each of the lower bounding methods, we branched on the variable x_j for which $L_j^+ + L_j^-$ was the largest. If $L_j^+ + L_j^- = 0\ \forall x_j \in I$, then we chose to branch on the fractional variable $x_j \in I'$ with largest objective function coefficient. We used the best bound node selection rule. Table VI shows a summary of the experiment. The full results are given in Linderoth and Savelsbergh.[27]

Table VI. Summary of Lower Bound Based Branching Rules Experiment

Branching Rule	Avg. Ranking
1 pivot (all)	1.64
10 pivots (all)	2.79
Strong	3.14
Knapsack	2.36

From the results of this experiment we make the following observations:

- It is in general too costly to perform 10 dual simplex pivots on all fractional variables.
- Strong branching can be highly effective on some problems, but the effectiveness is impacted greatly by the ability to select a suitable subset of variables on which to perform a number of dual simplex pivots.
- Performing one dual simplex pivot seems to be the best of the lower bounding based branching methods, which is a somewhat surprising result.

2.1.3 Combining Estimates and Lower Bounds

Getting lower bounding estimates on the degradation by performing one or more dual simplex pivots can be an expensive operation, but it gives us insight as to how the objective function value will change when branching on a variable *given the current formulation at a node*. We consider this "local" branching information. In contrast, pseudocost information is more "global" since these values are averages of observations taken throughout the branch-and-bound tree.

We want to consider combining this local and global branching information in some way. Specifically, we wish to estimate the true (up or down) degradation when branching on a variable x_j as

$$\hat{D}_j = \beta_0 + \beta_1 D_j + \beta_2 L_j, \quad (2)$$

D_j is the degradation measure taken from pseudocosts, and L_j is a lower bound on the degradation.

Since we only use these estimates $\hat{D}_j$ to rank the variables, the parameter β_0 is of no importance to us. The weights β_1 and β_2 could be chosen a priori, but defining the estimate in this way gives us the opportunity to create a dynamic branching scheme by altering these parameters as the search progresses. Given that we have evaluated n nodes, we do a regression to find the parameters β_1 and β_2 that give the best linear fit of the degradation $\hat{D}$ as a function of the pseudocost degradation D_j and lower bound on the degradation L_j. Doing a full regression at every node of the branch-and-bound tree may be too computationally expensive, but updating a regression from one observation to the next is

relatively easy. The parameters β_1 and β_2 can be obtained from the solution to the linear system

$$\begin{bmatrix} n & \Sigma D^i & \Sigma L^i \\ \Sigma D^i & \Sigma (D^i)^2 & \Sigma D^i L^i \\ \Sigma L^i & \Sigma D^i L^i & \Sigma (L^i)^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \Sigma \hat{D}^i \\ \Sigma P^i \hat{D}^i \\ \Sigma L^i \hat{D}^i \end{bmatrix}, \quad (3)$$

where the superscript ^{i} on D, L, or $\hat{D}$ is meant to denote the value of this quantity at node i, regardless of which variable was branched on. This system has a closed form solution involving only a few arithmetic operations, so computing the regression parameters is not too costly. If the lower bound on the degradation or the pseudocost is zero, we do not update the regression coefficients, because this would have the effect of "biasing" the coefficients so that the importance of the lower bound L_j in the calculation was weighted too heavily.

2.1.4 Using Degradation Estimates

Once we have computed estimates or bounds on the degradation of the objective function given that we branch on a specific variable, we still must decide how to use this information to make our branching choice. Our goal is to maximize the difference in LP value of the relaxation from a parent to its children, but since there are two children of each parent, there are different measures of change. Gauthier et al.[15] suggest trying to maximize the sum of the degradation on both branches, i.e., branch on the variable x_{j^*} with

$$j^* = \arg\max_j \{D_j^+ + D_j^-\}$$

or

$$j^* = \arg\max_j \{L_j^+ + L_j^-\}$$

This is the rule that we have used in our previous branching experiments. Bénichou et al.[5] and Beale[2] suggest instead to branch on the variable for which the smaller of the two estimated degradations is as large as possible. That is,

$$j^* = \arg\max_j \{\min\{D_j^+, D_j^-\}\}$$

or

$$j^* = \arg\max_j \{\min\{L_j^+, L_j^-\}\}.$$

Eckstein[13] suggests combining these ideas by branching on the variable

$$j^* = \arg\max\{\alpha_1 \min\{D_j^+, D_j^-\} + \alpha_2 \max\{D_j^+, D_j^-\}\} \quad (4)$$

or

$$j^* = \arg\max\{\alpha_1 \min\{L_j^+, L_j^-\} + \alpha_2 \max\{L_j^+, L_j^-\}\}. \quad (5)$$

Note that we can maximize the sum of the degradation on both branches by letting $\alpha_1 = \alpha_2 = 1$ in equation (4) or (5), and we can maximize the minimum degradation on both branches by letting $\alpha_1 = 1$, $\alpha_2 = 0$.

Table VII shows a summary of the effect of varying the parameters α_1 and α_2 and solving the MIPLIB test instances in Table I. The full results can be found in Linderoth and Savelsbergh.[27] For these runs, we have computed an estimated degradation as suggested by equation (2) with $\beta_1 = \beta_2 = 1$. We have set z_L to be the known optimal solution to the problem and used the best bound node selection rule. From the experiment we draw the following conclusions about using the degradation estimates or lower bounds to choose a branching variable:

Table VII. Summary of Computational Results On Using Degradations to Compute Branching Variables

(α_1, α_2)	Avg. Ranking
(1, 0)	4.71
(10, 1)	2.86
(2, 1)	1.86
(1, 1)	2.86
(1, 2)	3.64
(1, 10)	4.29

- Both the up and down degradations should be considered.
- The importance of a variable is more related to the smaller of the two degradations.
- Using $(\alpha_1, \alpha_2) = (2, 1)$ in equation (4) or (5) appears to be a good choice.

2.1.5 Non-estimate-Based Branching Rules

One further branching rule of note is suggested by Padberg and Rinaldi[33] and modified slightly by Jünger, Reinelt, and Thienel.[22] We call this method *enhanced branching*. Recall from (1) the set I' of variables on which we chose to make dual simplex pivots for strong branching. Enhanced branching chooses to branch on the variable $x_j \in I'$ for which the objective function coefficient is the largest. A variation of this method is the default branching rule invoked by MINTO (v2.0).[30]

2.1.6 Computational Results

This subsection describes a comprehensive experiment to compare the effectiveness of various branching rules. Table VIII describes the branching rules we studied. We solved each instance of MIPLIB using each of these branching rules. To minimize the effects of factors other than branching in proving optimality of the solution, we have fixed z_L to be the known optimal solution to the problem. We evaluate nodes in best bound order. When estimates or lower bounds of the degradation are used in the branching decision, we use them by taking $(\alpha_1, \alpha_2) = (2, 1)$ in the formula (4) or (5).

The instance *nwo4* of MIPLIB was excluded from the runs, since it required too much memory for our machine. The instance *dano3mip* was also excluded from the runs since the linear program is extremely difficult to solve—only 3 or 4 nodes can be evaluated in an hour. This left us with a test suite of 57 problem. The full results of the experiment can be found in Linderoth and Savelsbergh.[27] We call a problem instance *hard* if not all branching methods could prove the optimality of the solution in less than two minutes. In this

Table VIII. Branching Rules Investigated

Branching Method	Description
B1	Branch on variable closest to ½.
B2	Enhanced branching.
B3	Determine penalties by implicitly performing one dual simplex pivot. Strengthen penalties for integer variables as suggested by Tomlin.[37]
B4	Pseudocost-based branching.
B5	Use the estimates of degradation as in equation (2), with $\beta_1 = 1$, $\beta_2 = 1$.
B6	Use the estimates of degradation as in equation (2), and dynamically update the coefficients β_1 and β_2 by solving the system (3).
B7	Knapsack branching.

Table IX. Summary of Branching Results for all MIPLIB Instances

Method	Ranking (Min., Avg., Max.)	Problems Solved	Computation Time (sec.)
B1	(1, 5.77, 7)	8	97327
B2	(1, 5.32, 7)	11	91323
B3	(2, 4.29, 7)	14	78467
B4	(1, 2.26, 7)	19	65061
B5	(1, 2.41, 6)	19	65743
B6	(1, 2.32, 5)	19	65625
B7	(4, 4.97, 7)	14	79372

experiment, there were 34 hard instances. A summary of the experiment for the hard instances is given in Table IX.

Based on our branching rule experiments, we make the following observations:

- The use of pseudocosts in an intelligent manner is essential to solve many of the problems.
- Combining pseudocosts with lower bounding information seems to improve the robustness of the branching method at a relatively small computational price.
- There is no branching method which clearly dominates the others (note that almost all methods came in last at least once), so a sophisticated MIP solver should allow many different options for selecting the branching variable.

2.2 GUB Dichotomy

When the problem has generalized upper bound (GUB) constraints of the form $\Sigma_{j \in T} x_j = 1$ (or $\Sigma_{j \in T} x_j \leq 1$) for some $T \subseteq I$, another problem subdivision scheme used in practice is called branching on a *GUB Dichotomy*. Here, a subset $T' \subseteq T$ for which the solution of the LP relaxation x^i at node i satisfies $0 < \Sigma_{j \in T'} x_j^i < 1$ is chosen. The constraint $\Sigma_{j \in T'} x_j = 0$ is enforced in one subregion, and the constraint $\Sigma_{j \in T \setminus T'} x_j = 0$ is enforced in the other subregion. Note that these constraints can again be enforced by fixing variables' bounds. When there exists a logical ordering of the variables in the set T, this set is sometimes called a *special ordered set* (SOS) and hence this division method is sometimes called *SOS branching*.

Table X. Warehouse Sizes and Costs

Size	Cost
10	100
20	180
40	320
60	450
80	600

One advantage of branching on a GUB constraint instead of a variable is that the branch and bound tree is more "balanced." Suppose we have some GUB $\Sigma_{j \in T} x_j = 1$, and we choose to branch on a single variable j^*. If x_{j^*} is not an "important" variable, then it is likely that the set of feasible solutions for the node with $x_{j^*} = 0$ is very nearly the same as the set of feasible solutions for the original node. In this case, we have made little progress in our search.

A second advantage of branching on a GUB constraint occurs when the GUB is actually a SOS. In this case, the fractional LP solution may suggest which variable will be one in the optimal solution, and thereby demonstrate a good set T' on which to base the branching dichotomy. An example will make this point clear. Suppose we are modeling a facility location problem in which we must decide on the size of a warehouse to build. The choices of sizes and their associated cost are shown in Table X.

Using binary decision variables $x_1, x_2, \ldots, x_5$, we can model the cost of building the warehouse as

$$\text{COST} \equiv 100x_1 + 180x_2 + 320x_3 + 450x_4 + 600x_5 .$$

The warehouse will have size

$$\text{SIZE} \equiv 10x_1 + 20x_2 + 40x_3 + 60x_4 + 80x_5 ,$$

and we have the SOS constraint

$$x_1 + x_2 + x_3 + x_4 + x_5 = 1 .$$

If a linear programming solution has $x_1 = 0.35$ and $x_5 = 0.65$, then it might be trying to suggest building a warehouse of size

$$\text{SIZE} = 0.35(10) + 0.65(80) = 55.5 ,$$

in which case a sensible set on which to base the branching dichotomy would be $T' = \{1, 2, 3\}$. This means that our dichotomy is based on whether to build a warehouse of size less than 40 or larger than 60.

In our example, we assigned an order to the variables based on their coefficients in the "size" constraint. In general

SOS branching, constraints like the "size" constraint are termed *reference rows*. If the coefficients $a_1, a_2, \ldots a_{|T|}$ in the reference row are ordered such that $a_1 \leq a_2 \leq \ldots \leq a_{|T|}$, then a sensible set on which to base the branching dichotomy is

$$T' = \{j : a_j \leq \sum_{j \in T} a_j x_j^*\}.$$

The index

$$j' \equiv \arg\min_{j \in T \setminus T'} \{a_j\}$$

is usually called the *branch point* of the SOS.

Generalizing pseudocosts to help determine on which GUB to branch is not entirely straightforward. One simple idea is to extend the definition of down and up pseudocosts to apply to a GUB. Given that we have chosen an appropriate subset $T' \subseteq T$ on which to base our dichotomy, we can define the down and up pseudocosts for this GUB to be

$$P_j^- = \frac{z_{LP}^{i-} - z_{LP}^i}{\sum_{j \in T'} f_j^i} \text{ and } P_j^+ = \frac{z_{LP}^{i+} - z_{LP}^i}{1 - \sum_{j \in T'} f_j^i}.$$

This pseudocost definition does not take into account how the dichotomy was formed. Gauthier et al.[15] suggest assigning pseudocosts for each branch point in a SOS. Beale[2] gives a "row-based" method for determining both lower bounds and an estimate on the degradation if a GUB is branched on. Tomlin[36] extends the idea of performing one implicit dual simplex pivot to a set of variables, and performing a number of dual simplex pivots to obtain a lower bound on the degradation can also be applied in this case.

When a problem has GUB constraints, we are faced with the question of whether to branch on a GUB or on a variable. This question has received little attention in the literature. As suggested by Beale and Forrest,[3] if one uses a "row-based" method for determining estimates or lower bounds on the degradation of objective function value, then comparing the usefulness of branching on a GUB or a variable is straightforward.

2.2.1 Computational Results

To determine the value of GUB branching when there is no logical order to the variables in a GUB we compare GUB branching and plain variable branching on the instances of MIPLIB where there is a significant number of GUB constraints. There are 13 such instances. For this experiment, we adopt the following GUB branching strategy. We choose to branch on the GUB containing the greatest number of fractional variables in the current LP solution x^*. If there is no GUB containing at least 3 fractional variables, then we branch instead on a variable, using the adaptive combined estimate and lower bound strategy (B6). If we choose to branch on a GUB, the set T' on which the dichotomy is based is chosen in such a way so as to make $\Sigma_{j \in T'} x_j^*$ close to 0.5. Table XI we show a summary of an experiment. The full results can be found in Linderoth and Savelsbergh.[27]

From the results, we see that a straightforward approach to GUB branching seems to not be as effective as branching on a variable. Thus, if there is no logical order to the variables in a GUB, one should either branch on a variable, or make use of the more advanced GUB branching techniques detailed in this section.

Table XI. GUB vs. Variable Branching

Method	Ranking (Min., Avg., Max.)	Problems Solved	Computation Time (sec.)
B4	(1, 1.83, 3)	11	11829
B6	(1, 1.83, 3)	11	12324
GUB	(1, 2.33, 3)	8	14917

3. Node Selection

We now deal with the "select" portion of the branch-and-bound algorithm. When we make a decision to branch, we are solely concerned about maximizing the change in z_{LP}^i between a node N^i and its children. In selecting a node, our purpose is twofold: to find good integer feasible solutions or to prove that no solution better than our current one with value z_L exists. Therefore, the quality of the current solution value z_L is an important factor in determining which node to select for evaluation. Hence the decision of whether or not a heuristic procedure is used in order to obtain good integer feasible solutions is also a factor that must be considered when choosing a node selection rule. If a heuristic procedure is used, then node selection rules that emphasize proving that no better solution exist rather than finding improved integer feasible solutions may be preferred. Since many of the early branch-and-bound codes for solving MIP did not contain a heuristic as part of their solution procedure, existing ideas for node selection deserve more exploration. Here, we provide a brief survey of node selection methods. We categorize the node selection methods as *static* methods, *estimate-based* methods, *two-phase* methods, and *backtracking* methods. In addition, we introduce a new calculation on which to base estimation methods, and we perform experiments to test the effectiveness of the various methods.

3.1 Static Methods

A popular way to choose which subproblem to explore is to choose the one with the largest value of z_{LP}^i. There are theoretical reasons for making this choice, since for a fixed branching rule, selecting problems in this way minimizes the number evaluated nodes before completing the search. This node selection rule is usually called *best-first* or *best-bound* search.

At the other extreme is a selection rule called *depth-first* search. As the name suggests, the solution space is searched in a depth first manner.

Both these methods have inherent strengths and weaknesses. Best-first search will tend to minimize the number of nodes evaluated and at any point during the search is attempting to improve the global upper bound on the problem. Therefore best-first search concentrates on proving that no solution better than the current one exists. Memory requirements for searching the tree in a best-first manner may become prohibitive if good lower bounds are not found early, leading to relatively little pruning of the tree. Also, the

search tree tends to be explored in a breadth-first fashion, so one linear program to solve has little relation to the next—leading to higher node evaluation times.

Depth-first search overcomes both these shortcomings of best-first search. Searching the tree in a depth first manner will tend to minimize the memory requirements, and the changes in the linear program from one node to the next are minimal—usually just changing one variable's bound. Depth-first search has another advantage over best-first search in finding feasible solutions since feasible solutions tend to be found deep in the search tree. Depth-first search was the strategy proposed by Dakin[10] and Little et al.,[28] primarily due to the small memory capabilities of computers at that time. Despite its advantages, depth first search can lead to extremely large search trees. This stems from the fact that we may evaluate a good many nodes that would have been fathomed had a better value of z_L been known. For larger problems, depth first search has been shown to be impractical.[14] However, this conclusion was made in the days before primal heuristics were incorporated into most MIP codes, so depth first search deserves to be reexamined.

3.2 Estimate-Based Methods

Neither best first search nor depth first search make any intelligent attempt to select nodes that may lead to improved integer feasible solutions. What would be useful is some estimate of the value of the best feasible integer solution obtainable from a given node of the branch-and-bound tree. The *best projection* criterion, introduced by Hirst[20] and Mitra[29] and the *best estimate criterion* found in Bénichou et al.[5] and Forrest et al.,[14] are ways to incorporate this idea into a node selection scheme.

The best projection method and the best estimate method differ in how they determine an estimate of the best solution obtainable from a node. Given an estimate E^i, they both select the node in the active set for which this value is largest. For any node N^i, let $s^i \equiv \Sigma_{j \in I} \min(f_j, 1 - f_j)$ denote the sum total of its integer infeasibilities. Also, let the root node of the branch-and-bound tree be denoted by N^0. The best projection criterion for node selection is to choose the node with the highest value of

$$E_i = z_U^i + \left(\frac{z_L - z_U^0}{s^0} \right) s^i . \tag{6}$$

The value $\lambda = (z_L - z_U^0)/s^0$ can be thought of as the change in objective function value per unit decrease in infeasibility. Note that this method requires that there be a value of z_L.

The estimate obtained by the best projection method does not take into account which variables are fractional or the individual costs for satisfying each variable. A natural extension of the best projection idea would be to use pseudo-costs in obtaining an estimate of the value of the best solution obtainable from a node. This extension is what is known as the best estimate criterion. Here, the estimate of the best solution obtainable from a node is

$$E_i = z_U^i + \sum_{j \in I} \min(|P_j^- f_j|, |P_j^+ (1 - f_j)|). \tag{7}$$

This estimate has the advantage that it does not require a value of z_L.

3.3 Two-Phase Methods

Since we have two goals in node selection: finding good feasible solutions and proving that no better feasible solutions exist, it is natural to develop node selection strategies that switch from one goal to the other in the course of the algorithm. In the first phase, we are interested in determining good feasible solutions, while in the second phase, we are interested in proving that the solutions we obtained in the first phase are good or optimal. Perhaps the simplest "two-phase" algorithm is to perform depth first search until a feasible solution is found, then switch to best first search. A slight variation of this strategy is used by Eckstein.[13]

Forrest et al.[14] and Beale[2] propose a two-phase method that first chooses nodes according to the best-estimate criterion. Once a feasible solution is found, they state that it is better to select nodes that maximize a different criterion, known as the *percentage error*. The percentage error can be thought of as the amount by which the estimate of the solution obtainable from a node must be in error for the current solution x^* to not be optimal. The percentage error of a node i is

$$PE_i = 100 \, \frac{z_L - E_i}{z_U^i - z_L} .$$

3.4 Backtracking Methods

Define a *superfluous node* as a node N^i that has $z_{LP}^i < z^*$. Searching the tree in a best first manner will ensure that no superfluous nodes are evaluated. If, however, one can be assured that all (or most) of the superfluous nodes will be fathomed (which is the case if $z_L = z^*$), the memory and speed advantages of depth first search make this method the most preferable. Various authors have proposed strategies that attempt to go depth first as much as possible while minimizing the number of superfluous nodes evaluated.[15, 7, 8, 15] Given some estimate E_0 of the optimal objective function value z^*, the tree is searched in a depth first fashion as long as $z_{LP}^i > E_0$. If $z_{LP}^i \leq E_0$, then a node is selected by a different criterion such as best-first or best-estimate. The methods differ in the manner in which they obtain E_0 and in which criterion they use when deciding to backtrack.

For backtracking methods, we see there is a need for accuracy in the estimation of the optimal solution E_i obtainable from a node. From these values, we can estimate $E_0 = \max_{i \in J} E_i$. The accuracy of E_0 is important, but perhaps more important is noticing the effect of an inaccurate E_0. If the estimate is too large, then the tree will be searched in a best-first or best-estimate fashion and none of the advantages of going depth-first are obtained. If the estimate is too small, then the tree is searched in a more depth-first fashion and many superfluous nodes may be evaluated. This is a point to be kept in mind when deciding on reasonable estimates to use to guide the search.

The estimate of the best solution obtainable from a node given by (7) assumes that we will always be able to round a fractional variable to an integer in a manner that is the least detrimental to the objective, which is somewhat optimistic.

Table XII. Average Relative Error in Estimation of Optimal Solution

		Best Projection				Pseudocost				Adjusted Pseudocost			
Problem	Trials	n_o	E_o (%)	n_u	E_u (%)	n_o	E_o (%)	n_u	E_u (%)	n_o	E_o (%)	n_u	E_u (%)
bell3a	5	4	0.20	1	0.00	5	0.46	0	0.00	5	0.46	0	0.00
bell5	9	0	0.00	9	1.67	9	0.39	0	0.00	9	0.27	0	0.00
blend2	10	8	5.34	2	1.17	10	7.59	0	0.00	10	6.80	0	0.00
dcmulti	10	2	0.09	8	0.23	6	0.04	4	0.00	3	0.07	7	0.01
gesa2	10	0	0.00	10	0.46	10	0.15	0	0.00	5	0.02	5	0.12
gesa3_o	8	4	0.02	4	0.03	1	0.00	7	0.16	0	0.00	8	0.42
l152lav	4	1	0.28	3	0.45	2	0.25	2	0.61	0	0.00	4	1.99
misc07	5	1	6.70	4	16.78	4	15.77	1	1.81	3	12.10	2	16.47
mod008	10	4	7.45	6	2.07	8	4.98	2	0.23	8	4.34	2	0.75
p0201	8	2	0.98	6	0.74	1	0.90	7	5.22	1	0.90	7	15.35
pk1	10	4	9.85	6	38.24	10	92.09	0	0.00	10	85.67	0	0.00
rgn	5	3	1.91	2	0.17	5	8.07	0	0.00	5	6.63	0	0.00
stein45	10	0	0.00	10	6.72	10	19.21	0	0.00	10	17.84	0	0.00
vpm2	10	5	2.60	5	1.64	10	4.46	0	0.00	10	3.56	0	0.00
TOTAL	114	38	3.70	76	5.45	91	15.23	23	1.79	79	15.81	35	4.40

A more realistic estimate would be the following:

$$E_i = z_U^i + \sum_{j \in I: f_j \leq 0.5} (f_j P_j^- q_j + (1 - f_j) P_j^+ (1 - q_j)) + \sum_{j \in I: f_j > 0.5} (f_j P_j^- (1 - q_j) + (1 - f_j) P_j^+ q_j), \quad (8)$$

where q_j is the "probability" that we will be able to round a fractional solution to the closest integer and obtain a feasible integer solution.

An obvious question is how to calculate q_j for a variable. Since we are using a primal heuristic to generate feasible solutions, we can get an estimate of the percentage of variables that can be rounded to the nearest integer as follows. When an LP-based successive rounding heuristic starting from a fractional point x produces a feasible integer point $\hat{x}$, we define the *flip-percentage* ζ as

$$\zeta \equiv \frac{|\{j \in I : |\hat{x}_j - x_j| > 0.5\}|}{|I|}. \quad (9)$$

Regression studies showed that ζ calculated as in (9) is approximately constant for a given problem instance over a wide range of feasible solutions. To obtain a less optimistic estimate E_i, we could use equation (8), where $q_j = 1 - \zeta$.

We make two modifications to q_j in order to more accurately reflect the probability. The first of the modifications is based on the intuition that fractional variables close to integer values are more likely to be able to be rounded to this value in a feasible solution. The second of our modifications to q_j is based on the following notion. Suppose we have found a number of feasible solutions, and in each of these solutions $x_j = 1$. We might conjecture that there is something inherent in the problem structure that will force $x_j = 1$ in a feasible solution.

We performed a study to compare the various estimation methods. At certain nodes of the branch-and-bound tree, we computed estimates of the best solution obtainable from that node using the estimate from the Best Projection method (6), the pseudocost estimate (7), and the "adjusted pseudocost" estimate (8). We then solved the problem with the formulation at that node to optimality for comparison purposes. For a given estimated method, let n_o be the number of times the method overestimated the solution, and let the average relative error of overestimation be E_o. Likewise, let n_u be the number of times the method underestimated the solution, and let the average relative error of underestimation be E_u. Table XII shows a summary of the results of this study.

The experiments show that the best projection estimate often underestimates the true solution, and the pseudocost estimate usually overestimates the true solution. The adjusted pseudocost estimate also usually overestimates the solution, but not by as much as the regular pseudocost estimate. These characteristics are not that important by themselves, but have an impact on the performance of the backtracking methods that use them.

3.5 Branch Selection

Typical branching is based on a dichotomy that creates two new nodes for evaluation. The node selection scheme must also answer the question of how to rank the order of evaluation for these nodes. Schemes that prioritize the nodes based on an estimate of the optimal solution obtainable from that node have a built-in answer to this question, since distinct estimates are assigned to the newly created nodes. For schemes that do not distinguish between the importance of the two newly created nodes, such as depth first search, researchers have made the following suggestion. Suppose that we have based the branching dichotomy on the variable x_{j^*}, then we select the down node first if $f_{j^*}^i < 1 - f_{j^*}^i$ and the up node first otherwise.[25] If estimates are not available, we will use this rule for selecting whether to evaluate the down or up child of a node.

Table XIII. Node Selection Rules Investigated

Node Selection Method	Description
N1	Best Bound.
N2	Depth First.
N3	Depth First until a solution is obtained, then Best Bound.
N4	Best Estimate (normal pseudocost) until a solution is obtained, then Percentage Error.
N5	Best Projection.
N6	Best Estimate (normal pseudocost).
N7	Best Estimate (adjusted pseudocost).
N8	Backtrack. Best Projection Estimate. When backtracking, select node by best bound criterion.
N9	Backtrack. Best Estimate (normal pseudocost). When backtracking, select node by best bound criterion.
N10	Backtrack. Best Estimate (adjusted pseudocost). When backtracking, select node by best bound criterion.
N11	Backtrack. Best Projection Estimate. When backtracking, select node by best estimate criterion.
N12	Backtrack. Best Estimate (normal pseudocost). When backtracking, select node by best estimate criterion.
N13	Backtrack. Best Estimate (adjusted pseudocost). When backtracking, select node by best estimate criterion.

3.6 Computational Results

We performed an experiment to compare many of the node selection rules we have discussed. Each of the problems in MIPLIB was solved using the node selection methods detailed in Table XIII. For a branching method, we use the adaptive regression method B6 in Table VIII. All advanced features of MINTO were used, which includes a diving heuristic that is invoked every 10 nodes of the branch-and-bound tree.

As in the branching methods experiment the instances *nwo*4 and *dano3mip* were excluded from this experiment. In addition, the instance *arki*001 was also excluded, since during the heuristic phase, we encounter a linear program that takes more than one hour of CPU time to solve. This left us with 56 problems in the test suite. Table XIV shows a summary of the results of this experiment. The full results are in Linderoth and Savelsbergh.[127] When ranking the performance of a node selection method on a given instance, we used the following criteria:

- Methods are ranked first by the value of the best solution obtained.
- If two methods find the same solution, the method with the lower provable optimality gap is ranked higher.
- If both methods find the same solution and optimality gap, they are ranked according to computation time.
- Ties are allowed.

In computing the rankings, instances where each node selection method was able to prove the optimality of the solution and the difference between the best and worst methods' computation times was less than 30 seconds were excluded. This left us with 33 instances.

From the tables it is difficult to determine a clear winner among the node selection methods, but we can make the following observations:

- Pseudocost-based node estimate methods or combining a pseudocost-based estimate method in backtracking seems to be the best idea for node selection.
- Backtracking methods that select the node with the best estimate when backtracking generally outperform those methods that choose the best bound node when backtracking.
- There is no node selection method that clearly dominates the others (note that almost all methods came in last at least once), so a sophisticated MIP solver should allow many different options for selecting the next node to evaluate.
- Even in the presence of a primal heuristic, depth-first search performs poorly in practice.

4. Conclusions

We have examined a wide variety of branching methods and node selection rules for mixed integer programming. The strategies were examined in conjunction with many of the advanced features found in today's sophisticated MIP solvers to see if strategies developed decades ago are still practical today. In general, we conclude that the early methods are indeed still practical today. Especially pseudocost-based methods, when used intelligently, seem to be very beneficial.

There is no one search strategy that will work best on all problem instances. Hopefully the results here will help guide researchers and practitioners in selecting suitable branching and node selection rules for their problems.

Some important topics for further investigation are:

- Can the "local" and "global" branching information from simplex pivots and pseudocosts be combined in a more intelligent way than by simple linear regression.
- Can we develop intelligent ways to choose a "good" subset of variables for strong branching?
- Can we get better estimates of the optimal solution obtainable from a node?
- Can we develop search strategies that adapt themselves based on the observed behavior for a given problem instance?

Table XIV. Summary of Node Selection Method Experiment

Method	Ranking (Min., Avg., Max.)	Times No Solution Found	Times Optimal Solution Found	Computation Time (sec.)
N1	(1, 6.67, 13)	2	21	68189
N2	(1, 8.42, 13)	1	16	80005
N3	(1, 7.12, 13)	1	20	72207
N4	(1, 6.12, 13)	2	23	72324
N5	(1, 7.03, 13)	0	20	79082
N6	(1, 5.67, 12)	2	24	66147
N7	(1, 5.94, 13)	1	21	67823
N8	(1, 7.00, 13)	1	20	74375
N9	(1, 6.85, 12)	1	21	66475
N10	(1, 6.94, 13)	1	21	68106
N11	(1, 7.42, 13)	1	15	78568
N12	(1, 5.70, 13)	1	23	65861
N13	(1, 5.42, 11)	1	24	67944

5. Acknowledgments

The authors would like to acknowledge Alper Atamtürk for many useful discussions and an anonymous referee for suggestions leading to a more concise presentation.

References

1. R. BIXBY, 1996. *Personal communication.*
2. E.M.L. BEALE, 1979. Branch-and-Bound Methods for Mathematical Programming Systems, in *Discrete Optimization II*, P.L. Hammer, E.L. Johnson, and B.H. Korte (eds.), North Holland Publishing Co., Amsterdam.
3. E.M.L. BEALE and J.J.H. FORREST, 1976. Global Optimization Using Special Ordered Sets, *Mathematical Programming 10*, 52–69.
4. E.M.L. BEALE and R.E. SMALL, 1966. Mixed Integer Programming by a Branch-and-Bound Method, in *Proceedings IFIP Congress 65*, W.H. Kalenich (ed.), Volume 2.
5. M. BÉNICHOU, J.M. GAUTHIER, P. GIRODET, G. HENTGES, G. RIBIÈRE, and O. VINCENT, 1971. Experiments in Mixed-Integer Linear Programming, *Mathematical Programming 1*, 76–94.
6. R.E. BIXBY, S. CERIA, C.M. MCZEAL, and M.W.P. SAVELSBERGH, 1998. An Updated Mixed Integer Programming Library: MIPLIB 3.0, *Technical Report TR98-03*, Department of Computational and Applied Mathematics, Rice University, available from *http://www.caam.rice.edu/~bixby/miplib/miplib.html*
7. R. BREU and C.A. BURDET, 1974. Branch-and-Bound Experiments in Zero-One Programming, *Mathematical Programming 2*, 1–50.
8. CPLEX Optimization, Inc., 1995. *Using the CPLEX Callable Library.*
9. H. CROWDER, E.L. JOHNSON, and M.W. PADBERG, 1983. Solving Large Scale Zero-One Linear Programming Problems, *Operations Research 31*, 803–834.
10. R.J. DAKIN, 1965. A Tree Search Algorithm for Mixed Programming Problems, *Computer Journal 8*, 250–255.
11. R.E. DAVIS, D.A. KENDRICK, and M. WEITZMAN, 1967. A Branch-and-Bound Algorithm for Zero-One Mixed Integer Programming Problems, *Technical Report Development Economic Report 69*, Center for International Affairs, Harvard University.
12. N.J. DRIEBEEK, 1966. An Algorithm for the Solution of Mixed Integer Programming Problems, *Management Science 12*, 576–587.
13. J. ECKSTEIN, 1994. Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5, *SIAM Journal on Optimization 4*, 794–814.
14. J.J.H. FORREST, J.P.H. HIRST, and J.A. TOMLIN, 1974. Practical Solution of Large Scale Mixed Integer Programming Problems with UMPIRE, *Management Science 20*, 736–773.
15. J.M. GAUTHIER and G. RIBIÈRE, 1977. Experiments in Mixed-Integer Linear Programming Using Pseudocosts, *Mathematical Programming 12*, 26–47.
16. A.M. GEOFFRION and R.E. MARSTEN, 1972. Integer Programming Algorithms: A Framework and State-of-the-Art Survey, *Management Science 18*, 465–491.
17. Z. GU, G.L. NEMHAUSER, and M.W.P. SAVELSBERGH, 1998. Cover Inequalities for 0-1 Linear Programs: Computation, *INFORMS Journal on Computing 10*, 427–437.
18. O. GÜNLÙK, 1996. A Branch-and-Cut Algorithm for Capacitated Network Design Problems, *unpublished manuscript.*
19. M.T. HAJIAN and G. MITRA, 1995. Design and Testing of an Integrated Branch-and-Bound Algorithm for Piecewise Linear and Discrete Programming Problems, *Technical Report TR/01/95*, Brunel, the University of West London, London.
20. J.P.H. HIRST, 1969. Features Required in Branch-and-Bound Algorithms for (0-1) Mixed Integer Linear Programming, *unpublished manuscript.*
21. W.C. HEALY, JR., 1964. Multiple Choice Programming, *Operations Research 12*, 122–138.
22. M. JÜNGER, G. REINELT, and S. THIENEL, 1994. Provably Good Solutions for the Traveling Salesman Problem, *Zeitschrift für Operations Research 40*, 183–217.
23. T. LAI and S. SAHNI, 1983. Anomalies in Parallel Branch-and-Bound Algorithms, in *Proceedings of the 1983 International Conference on Parallel Processing.*
24. T.H. LAI and A. SPRAGUE, 1986. A Note on Anomalies in Parallel Branch-and-Bound Algorithms with One-to-One Bounding Functions, *Information Processing Letters 23*, 119–122.
25. A. LAND and S. POWELL, 1979. Computer Codes for Problems of Integer Programming, in *Discrete Optimization II*, P.L. Hammer, E.L. Johnson, and B.H. Korte (eds.), North Holland Publishing Co., Amsterdam.

26. A.H. LAND and A.G. DOIG, 1960. An Automatic Method for Solving Discrete Programming Problems, *Econometrica 28*, 497–520.
27. J.T. LINDEROTH and M.W.P. SAVELSBERGH, 1997. A Computational Study of Search Strategies in Mixed Integer Programming, *Technical Report TLI-97-12*, Georgia Institute of Technology.
28. J.D.C. LITTLE, K.G. MURTY, D.W. SWEENEY, and C. KAREL, 1963. An Algorithm for the Traveling Salesman Problem, *Operations Research 21*, 972–989.
29. G. MITRA, 1973. Investigation of Some Branch-and-Bound Strategies for the Solution of Mixed Integer Linear Programs, *Mathematical Programming 4*, 155–170.
30. G.L. NEMHAUSER, M.W.P. SAVELSBERGH, and G.C. SIGISMONDI, 1994. MINTO, a Mixed INTeger Optimizer, *Operations Research Letters 15*, 47–58.
31. B. NYGREEN, 1991. Branch-and-Bound with Estimation Based on Pseudo Shadow Prices, *Mathematical Programming 52*, 59–69.
32. M. PADBERG, T.J. VAN ROY, and L. WOLSEY, 1985. Valid Linear Inequalities for Fixed Charge Problems, *Operations Research 33*, 842–861.
33. M.W. PADBERG and G. RINALDI, 1991. A Branch-and-Cut Algorithm for the Solution of Large Scale Traveling Salesman Problems, *SIAM Review 33*, 60–100.
34. T.J. VAN ROY and L.A. WOLSEY, 1987. Solving Mixed Integer 0-1 Programs by Automatic Reformulation, *Operations Research 35*, 45–57.
35. M.W.P. SAVELSBERGH, 1994. Preprocessing and Probing Techniques for Mixed Integer Programming problems, *ORSA Journal on Computing 6*, 445–454.
36. J.A. TOMLIN, 1970. Branch-and-Bound Methods for Integer and Non-Convex Programming, in *Integer and Non-linear Programming*, J. Abadie (ed.), North Holland, Amsterdam.
37. J.A. TOMLIN, 1971. An Improved Branch-and-Bound Method for Integer Programming, *Operations Research 19*, 1070–1075.
38. J.M. TROYA, and M. ORTEGA, 1989. Study of Parallel Branch-and-Bound Algorithms with Best-Bound-First Search, *Parallel Computing 11*, 121–126.