Lookahead Branching for Mixed Integer Programming

Wasu Glankwamdee
Jeffrey Linderoth
Lehigh University

Report No. 06T-004

Lookahead Branching for Mixed Integer Programming

Wasu Glankwamdee
Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University
200 W. Packer Ave.
Bethlehem, PA 18015

wag3@lehigh.edu jtl3@lehigh.edu

October 12, 2006

Abstract

We consider the effectiveness of a *lookahead* branching method for the selection of branching variable in branch-and-bound method for mixed integer programming. Specifically, we ask the following question: by taking into account the impact of the current branching decision on the bounds of the child nodes *two* levels deeper than the current node, can better branching decisions be made? We describe methods for obtaining and combining bound information from two levels deeper in the branch-and-bound tree, demonstrate how to exploit auxiliary implication information obtain in the process, and provide extensive computational experience showing the effectiveness of the new method.

1 Introduction

A mixed integer program (MIP) is the problem of finding

$$z_{MIP} = \max\{c^T x + h^T y : Ax + Gy \leq b, x \in \mathbb{Z}^{|I|}, y \in \mathbb{R}^{|C|}\}, \qquad \text{(MIP)}$$

where I is the set of integer-valued decision variables, and C is the set of continuous decision variables. The most common algorithm for solving MIP, dating back to Land and Doig [20], is a branch-and-bound method that uses the linear programming relaxation

of MIP to provide an upper bound on the optimal solution value (z_{MIP}). Based on the solution of the relaxation, the feasible region is partitioned into two or more subproblems. The partitioning processes is repeated, resulting in a tree of relaxations (typically called a *branch-and-bound tree*) that must be evaluated in order to solve MIP. See [26] or [31] for a more complete description of the branch-and-bound method for MIP.

A key decision impacting the effectiveness of the branch-and-bound method is *how* to partition the feasible region. Typically, the region is divided by *branching on a variable.* Branching on a variable is performed by identifying a decision variable x_j whose solution value in the relaxation ($\hat{x}_j$) is not integer-valued. The constraint $x_j \leq \lfloor \hat{x}_j \rfloor$ is enforced in one subproblem, and the constraint $x_j \geq \lceil \hat{x}_j \rceil$ is enforced in the other subproblem. In a given solution $(\hat{x}, \hat{y})$ to the LP relaxation of MIP, there may be many decision variables for which $\hat{x}_j$ is fractional. A natural question to ask is on *which* of the fractional variables should the branching dichotomy be based.

The effectiveness of the branch-and-bound method strongly depends on how quickly the upper bound on z_{MIP}, obtained from the solution to a relaxation, decreases. Therefore, we would like to branch on a variable that will reduce this upper bound as quickly as possible. In fact, a long line of integer programming research in the 1970's was focused on developing branching methods that estimated which variables would be most likely to lead to a large decrease in the upper bound of the relaxation after branching [4, 18, 25, 6, 16, 17].

In the 1990's, in connection with their work on solving large-scale traveling salesperson instances, Applegate *et al.* proposed the concept of *strong branching* [2]. In strong branching, the selection of a branching variable is made by *tentatively* selecting each variable from a potential set $\mathcal{C}$ of candidates to be the branching variable and observing the change in relaxation value after performing a fixed, limited number of dual simplex pivots. The intuition behind strong branching is that if the subproblem bounds change significantly in a limited number of simplex pivots, then the bound will also change significantly (relative to other choices) should the child node relaxations be fully resolved. Strong branching has been shown to be an effective branching rule for many MIP instances and has been incorporated into many commercial solvers, e.g. CPLEX [9] and XPRESS [10]. In *full strong branching*, the set $\mathcal{C}$ is chosen to be the set of all fractional variables in the solution of the relaxation, and there is no upper limit placed on the number of dual simplex pivots performed. Full strong branching is a computationally expensive method, so typically $\mathcal{C}$ is chosen to be a subset of the fractional variables in the relaxation solution, and the number of simplex pivots performed is small.

When employed judiciously, strong branching has been shown to be a very effective

branching method. In fact, researchers *outside* the realm of mixed integer linear programming, but still where branch-and-bound is employed, have also recently begun to use the idea of strong branching to aid the branching decisions. Vandenbussche and Nemhauser use strong branching to determine on which complementarity condition to branch in an algorithm for solving nonconvex quadratic programs [30], and Fampa and Anstreicher use strong branching to reduce the size of an enumeration tree of Steiner topologies in their algorithm for solving the multidimensional Steiner Tree problem [15].

The fact that strong branching can be a powerful, but computational costly, technique has led some researchers to consider weaker forms of strong branching that only perform the necessary computations at certain nodes. For example, Linderoth and Savelsbergh [22] suggest to perform the strong branching computations for variables that have yet to be branched upon. The commercial package LINDO performs strong branching at all nodes up to a specified depth d of the branch-and-bound tree [23]. This work was improved by Achterberg, Koch, and Martin, in a process called *reliability branching* in which the choice of the set $\mathcal{C}$ and the number of pivots to perform is dynamically altered during the course of the algorithm [1].

The fundamental motivation of this paper is to consider the exact *opposite* question as that of previous authors. That is, instead of performing *less* work than full strong branching, what if we performed *more*? Specifically, by taking into account the impact of the current branching decision on the bounds of the child nodes *two* levels deeper than the current node, can we make better branching decisions? The intuition behind studying this question is to view strong branching as a greedy heuristic for selecting the branching variable. By considering the impact of the branching decision not just on the child subproblems, but on the grandchild subproblems as well, can we do better? And if so, at what computational cost?

Obviously, obtaining information about the bounds of potential child nodes two levels deeper than the current node may be computationally expensive. In this work, we will for the most part focus on the question of *if* attempting to obtain this information is worthwhile, rather than on how to obtain the information in a computationally efficient manner. However, even if obtaining this information is extremely costly, we note two factors that may mitigate this expense. First, in codes for mixed integer programming that are designed to exploit significant parallelism by evaluating nodes of the branch-and-bound tree on distributed processors [14, 21, 28], in the initial stages of the algorithm, there are not enough active nodes to occupy available processors. If obtaining information about the impact of branching decisions at deeper levels of the tree is useful, then these idle processors could be put to useful work by computing this infor-

mation. Second, as noted by numerous authors [16, 22], the branching decisions made at the top of the tree are the most crucial. Perhaps the "expensive" lookahead branching techniques need only be done at for the very few first nodes of the branch-and-bound tree.

We are not aware of a work that considers the impact of the branching decision on grandchild nodes. Anstreicher and Brixius considered a "weak" (but computationally efficient) form of two-level branching as one of four branching methods described in [7]. In the method, k_1 "pivots" are made to consider one branching decision; then, using dual information akin to the penalties of Driebeek [12], one more "pivot" on a second branching entity is considered. This paper is an abbreviated version of the Master's Thesis of Glankwamdee [19], wherein more complete computational results can be found.

The paper has two remaining sections. In Section 2, we explain the method for gathering branching information from child and grandchild nodes, and we give one way in which this information can be used to determine a branching variable. We also show that auxiliary information from the branching variable determination process can be used to tighten the LP relaxation and reduce the size of the search tree. In Section 3, we present methods to speed up the lookahead branching method. Extensive computational experiments are performed to determine good parameter settings for practical strong branching and lookahead methods. These branching methods are compared to that of MINTO, a sophisticated solver for mixed integer programs.

2 Lookahead Branching

In this section, the question of whether or not significantly useful branching information can be obtained from potential grandchild nodes in the branch-and-bound tree is examined. We explain our method for gathering this information and describe a simple lookahead branching rule that hopes to exploit the branching information obtained.

Figure 1 shows a potential two-level expansion of the search tree from an initial node. The set $\mathcal{F}$ is the set of fractional variables in the solution to the initial LP relaxation (x^*). By definition, the solution value of an infeasible linear program is denoted as $z_{LP} = -\infty$, and the lower bound on the optimal solution value z_{MIP} is denoted at z_L. If the constraint $x_i \leq \lfloor x_i^* \rfloor$ is imposed on the left branch, and the relaxation is resolved, a solution of value z_i^- is obtained, and there is a set of variables $\mathcal{F}_i^- \subseteq I$ that takes fractional values. We use the parameter $\xi_i = 1$ to indicate if the left branch would be pruned (i.e. if $z_i^- \leq z_L$); otherwise $\xi_i^- = 0$. Similarly, if the bound constraint $x_i \geq \lceil x_i^* \rceil$ is imposed on the right branch, a solution of value z_i^+ is obtained, a set of variables

$\mathcal{F}_i^+ \subseteq I$ is fractional, and the parameter ξ_i^+ indicates if the child node would be pruned.

Continuing to the second level in Figure 1, if the variable $j \in \mathcal{F}_i^-$ is chosen as the branching variable for the left child node, then the solution values for the two grandchild nodes are denoted as z_{ij}^{--} and z_{ij}^{-+}, and the indicator parameters ρ_{ij}^{--} and ρ_{ij}^{-+} are set to 1 if the corresponding grandchild nodes would be pruned, otherwise the indicators are set to 0. The notation for grandchild nodes on the right is similar.

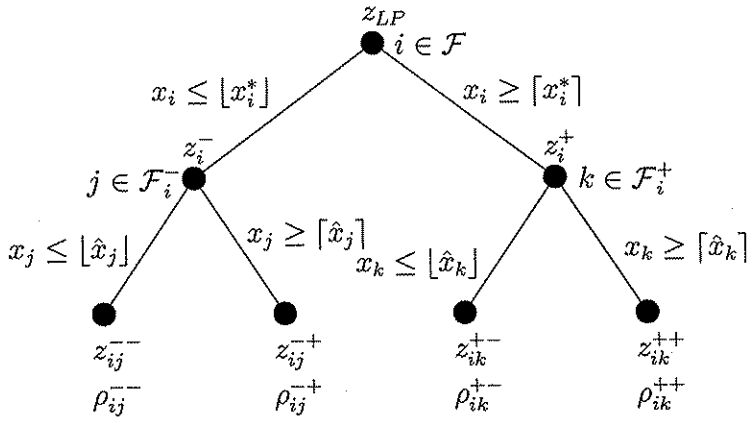

Figure 1: Notations for Lookahead Search Tree

2.1 Branching Rules

Once the branching information from the grandchild nodes is collected, there is still the question of to how to use this information to aid the current branching decision. Two reasonable objectives in choosing a branching variable are to minimize the number of grandchild nodes that are created, and to try to decrease the LP relaxation bounds at the grandchild nodes as much as possible. Various combinations of these objectives were explored in [19]. To keep the exposition short, we mention only one such method here. To precisely define the rule, we will use the following definitions. Let

$$\mathcal{G}_i^- \stackrel{\text{def}}{=} \{j \in \mathcal{F}_i^- | \rho_{ij}^{--} = 0, \rho_{ij}^{-+} = 0\} \tag{1}$$

and

$$\mathcal{G}_i^+ \stackrel{\text{def}}{=} \{k \in \mathcal{F}_i^+ | \rho_{ik}^{+-} = 0, \rho_{ik}^{++} = 0\} \tag{2}$$

be the sets of indices of fractional variables in child nodes both of whose grandchild nodes would not be pruned. To combine the progress on bound reduction of two child nodes into one number, we use the weighting function

$$\mathcal{W}(a,b) \stackrel{\text{def}}{=} \{\mu_1 \min(a,b) + \mu_2 \max(a,b)\}, \tag{3}$$

as suggested by Eckstein[13]. In this paper, the parameters of the weighting function are set to $\mu_1 = 4$ and $\mu_2 = 1$. Linderoth and Savelsbergh verified empirically that these weights resulted in good behavior over a wide range of instances [22]. Let the reduction in the LP relaxation value at the grandchild nodes be denoted by

$$D_{ij}^{s_1 s_2} \stackrel{\text{def}}{=} z_{LP} - z_{ij}^{s_1 s_2}, \text{ where } s_1, s_2 \in -, +. \tag{4}$$

Note that $D_{ij}^{s_1 s_2} \geq 0$. The symbol η_i counts the total number of potential grandchild nodes that would be fathomed if variable i was chosen as the branching variable i.e.

$$\eta_i \stackrel{\text{def}}{=} \sum_{j \in \mathcal{F}_i^-} (\rho_{ij}^{--} + \rho_{ij}^{-+}) + \sum_{k \in \mathcal{F}_i^+} (\rho_{ik}^{+-} + \rho_{ik}^{++}). \tag{5}$$

The two goals of branching, bound reduction and node elimination, are combined into one measure through a weighted combination. The branching rule employed in the experiments chooses to branch on the variable i^* that maximizes this weighted combination namely

$$i^* = \arg\max_{i \in \mathcal{F}} \left\{ \max_{j \in \mathcal{F}_i^-} \{\mathcal{W}(D_{ij}^{--}, D_{ij}^{-+})\} + \max_{k \in \mathcal{F}_i^+} \{\mathcal{W}(D_{ik}^{+-}, D_{ik}^{++})\} + \lambda \eta_i \right\}, \tag{6}$$

$$\text{where } \lambda = \frac{1}{|\mathcal{G}_i^-|} \sum_{j \in \mathcal{G}_i^-} \mathcal{W}(D_{ij}^{--}, D_{ij}^{-+}) + \frac{1}{|\mathcal{G}_i^+|} \sum_{k \in \mathcal{G}_i^+} \mathcal{W}(D_{ik}^{+-}, D_{ik}^{++}) \tag{7}$$

is the average (weighted) reduction in LP value of all potential grandchild nodes in the sets $\mathcal{G}_i^-$ and $\mathcal{G}_i^+$. This value of λ was chosen to give the terms in equation (6) the same scale. Note that in equation (6), the variables $j \in \mathcal{F}_i^-$ and $k \in \mathcal{F}_i^+$ that maximize the weighted degradation of the grandchild nodes LP relaxation value may be different. To implement full strong branching, we let

$$D_i^s \stackrel{\text{def}}{=} z_{LP} - z_i^s, \text{ where } s \in -, +, \tag{8}$$

and we branch on the variable

$$i^* = \arg\max_{i \in \mathcal{F}} \{\mathcal{W}(D_i^-, D_i^+)\}. \tag{9}$$

2.2 Implications and Bound Fixing

When computing the LP relaxation values for many potential child and grandchild nodes, auxiliary information is obtained that can be useful for tightening the LP relaxation and reducing the size of the search tree.

2.2.1 Bound Fixing

When tentatively branching on a variable x_i, either in strong branching or in lookahead branching, if one of the child nodes is fathomed, then the bounds on variable x_i can be improved. For example, if the child node with branching constraint $x_i \geq \lceil x_i^* \rceil$ is infeasible ($\xi_i^+ = 1$), then we can improve the upper bound on variable i to be $x_i \leq \lfloor x_i^* \rfloor$. Likewise, if there exists no feasible integer resolution for a variable j after branching on a variable i, then the bound on variable i can be set to its complementary value. The exact conditions under which variables can be fixed are shown in Table 1.

Condition	Implication
$\xi_i^- = 1$	$x_i \geq \lceil x_i^* \rceil$
$\xi_i^+ = 1$	$x_i \leq \lfloor x_i^* \rfloor$
$\rho_{ij}^{--} = 1$ and $\rho_{ij}^{-+} = 1$	$x_i \geq \lceil x_i^* \rceil$
$\rho_{ik}^{+-} = 1$ and $\rho_{ik}^{++} = 1$	$x_i \leq \lfloor x_i^* \rfloor$

Table 1: Bound Fixing Conditions

2.2.2 Implications

By examining consequences of fixing 0-1 variables to create potential grandchild nodes, simple inequalities can be deduced by combining mutually exclusive variable bounds into a single constraint. The inequality identifies two variables, either original or complemented, that cannot simultaneously be 1 in an optimal solution. For example, if variables x_i and x_k are binary decision variables, and the lookahead branching procedure determines that branching "up" on both x_i and x_k (i.e. $x_i \geq 1, x_k \geq 1$) results in a subproblem that may be pruned, then the inequality $x_i + x_k \leq 1$ can be safely added to the LP relaxation at the current node. These inequalities are essentially additional edges in a local *conflict graph* for the integer program [29, 3]. As a line of future research, we intend to investigate the impact of adding these edges to the local conflict graph, and performing additional preprocessing. Further *grandchild inequalities* can be added if any of the grandchild nodes would be pruned as specified in Table 2.

Condition	Inequality
$\rho_{ij}^{--} = 0$	$(1 - x_i) + (1 - x_j) \leq 1$
$\rho_{ij}^{+-} = 0$	$(1 - x_i) + x_j \leq 1$
$\rho_{ik}^{+-} = 0$	$x_i + (1 - x_k) \leq 1$
$\rho_{ik}^{++} = 0$	$x_i + x_k \leq 1$

Table 2: Grandchild Implications

2.3 Characteristics of Computational Experiments

The lookahead branching rule has been incorporated into the mixed integer optimizer MINTO v3.1, using the `appl_divide()` user application function that allows the user to specify the branching variable [27]. In all the experiments, the default MINTO options, including preprocessing and probing, automatic cut generation, and reduced cost fixing, were used. For these experiments, the lower bound value z_L was initialized to be objective value of the (known) optimal solution. By setting z_L to the value of the optimal solution, we minimize factors other than branching that determine the size of the branch-and-bound tree. To solve the linear programs that arise, we use CPLEX(v8.1) [9]. To speedup the testing of the algorithm, we run the experiments on a Beowulf cluster at Lehigh University. The code was compiled with gcc version 2.96 (Red Hat Linux 7.1), and run on Intel(R) Pentium(R) III CPU, with clock speed 1133MHz. The CPU time was limited to a maximum of 8 hours, and the memory was limited to a maximum of 1024MB. We have limited initial test to a suite of 16 instances from MIPLIB 3.0 [5] and MIPLIB 2003 [24].

2.4 Computational Results

In our first experiment, we ran an implementation of full strong branching, with and without bound fixing and implications, and lookahead branching, with and without bound fixing and implications. The primary focus of the experiment is not on the speed of the resulting methods at this point, but instead on the following two questions:

- Does lookahead branching often make different branching decisions compared to full strong branching? If so, what are the positive impact of these branching decisions?

- Do bound fixing and grandchild inequalities coming from implications found in the lookahead branching procedure have a positive impact on the size of the search

tree?

Full details of the experimental runs can be found in Tables 4, 5 and 6 in the Appendix. To summarize the results of the experiments, we use performance profiles plotted in log scale, as introduced by Dolan and Moré [11]. A performance profile is a relative measure of the effectiveness of a solver s when compared to a group of solvers $\mathcal{S}$ on a set of problem instances P. To completely specify the performance profile, we need the following definitions:

- γ_{ps} is a quality measure of solver s when solving problem p,
- $r_{ps} = \gamma_{ps}/(\min_{s\in\mathcal{S}} \gamma_{ps})$, and
- $\rho_s(\tau) = |\{p \in P \mid r_{ps} \leq \tau\}|/|P|$.

Hence, $\rho_s(\tau)$ is the fraction of instances for which the performance of solver s was within a factor of τ of the best. A performance profile for solver s is the graph of $\rho_s(\tau)$. In general, the higher the graph of a solver, the better the relative performance. Eleven of the sixteen instances were solved to provable optimality by one of the four methods, and for these instances, we use the number of nodes as the quality measure γ_{ps}. Under this measure, $\rho_s(1)$ is the fraction of instances for which solver s evaluated the fewest number of nodes to verify optimality, and $\rho_s(\infty)$ is the fraction of instances for which solver s verified the optimality of the solution of value z_L. Figure 2 shows the performance profile plot for these eleven instances. *SB* and *LA* denote strong branching and lookahead branching respectively while *Implication* indicates that bound fixing and implications are added to the algorithms. Two conclusions can evidently be drawn from Figure 2.

1. Using bound fixing and grandchild inequalities can *greatly* reduce the number of nodes in the branch-and-bound tree, and
2. Neither full strong branching nor lookahead branching seems to significantly outperform the other in these tests.

The fact that full strong branching and lookahead branching seem to be of comparable quality is slightly surprising, more so when one considers the fact that lookahead branching *quite often* chooses to branch on a different variable than full strong branching does. In Table 3, the second column lists the percentage of nodes at which lookahead branching and full strong branching would make *different* branching decisions. For example, for the instance `qiu`, the two methods choose a different branching variable 96%

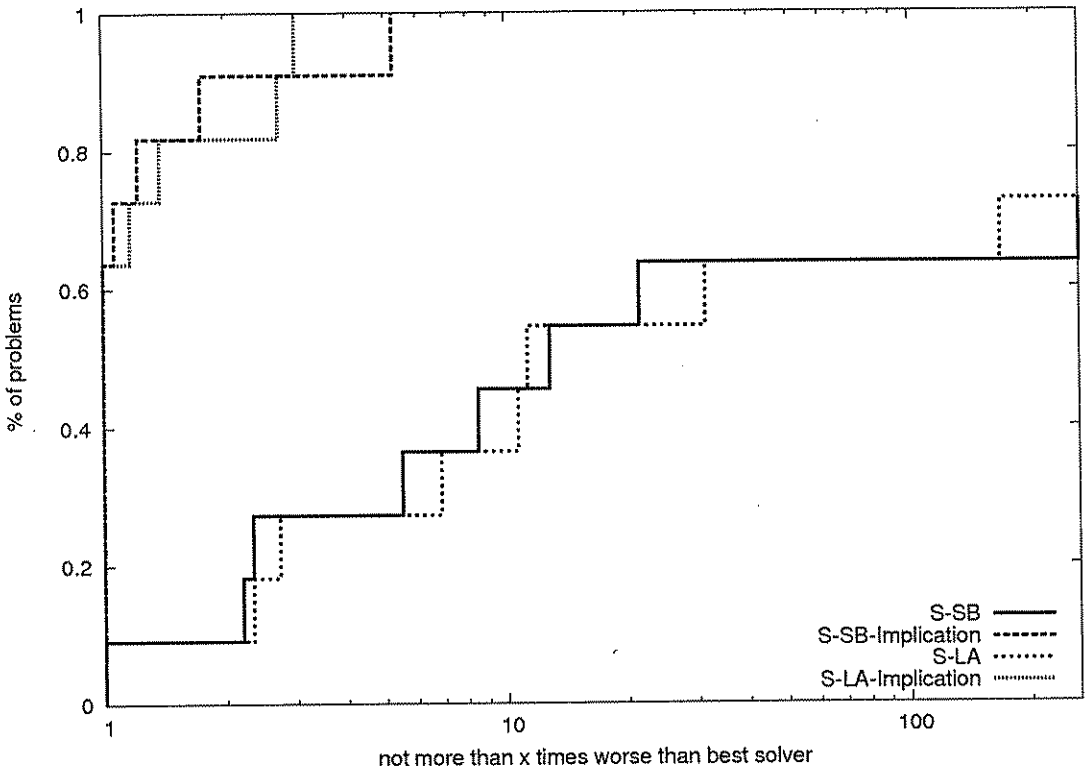

Figure 2: Performance Profile of Number of Evaluated Nodes in Solved Instances

of the time. Also in Table 3, we list the number of times a variable's bound was improved per node, and the number of grandchild inequalities per node that was added.

Name	% Diff	# of Bound Fixing (per node)	# of Inequalities (per node)
aflow30a	0.76	1.53	47.80
aflow40b	0.59	0.72	26.24
danoint	0.86	0.85	3.48
l152lav	0.25	2.56	267.56
misc07	0.73	1.32	32.35
modglob	0.50	1.00	17.13
opt1217	0.46	1.00	0
p0548	1.00	3.00	15.33
p2756	0	0.67	15.00
pk1	0.53	0.87	10.71
pp08a	0.75	1.02	0.86
qiu	1.00	3.60	120.60
rgn	0.71	0.64	2.75
stein45	0.60	1.19	72.08
swath	0.84	0.73	1.51
vpm2	0.54	0.83	8.81

Table 3: Lookahead Branching Statistics

For five of the sixteen instances, none of the branching methods was able to prove the optimality of the solution. For these instances, we use as our quality measure (γ_{ps}), the final integrality gap after eight hours of CPU time. Figure 3 shows the performance profile of the four branching methods on the unsolved instances. Again, we see that the bound fixing and grandchild (implications) inequalities can be very effective, and that lookahead branching and full strong branching are of relatively similar quality for these instances.

The final performance profile (in Figure 4) uses the solution time as the quality parameter for the eleven solved instances, and also includes a profile for the default MINTO branching rule. The profile demonstrates that

- As expected, the running time of the strong branching and lookahead branching are in general worse than the default MINTO.

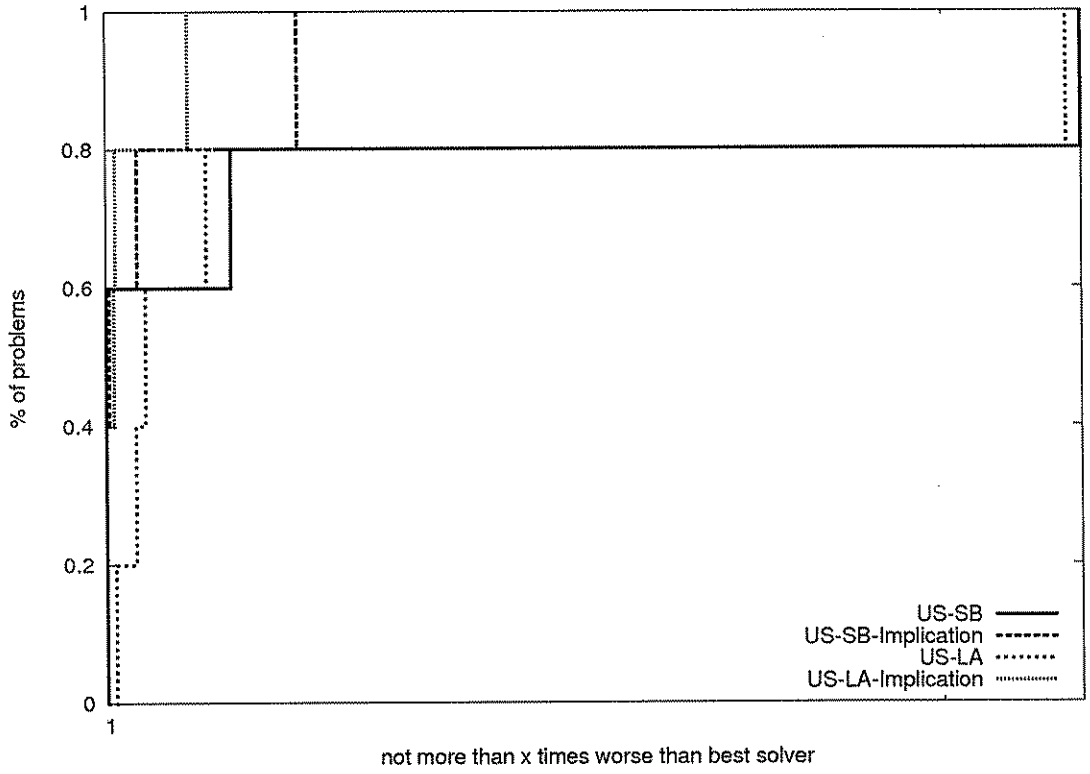

Figure 3: Performance Profile of Integrality Gap in Unsolved Instances

- However, the added implications and bound fixing help to solve the `pk1` instance which is unsolved with the default MINTO.

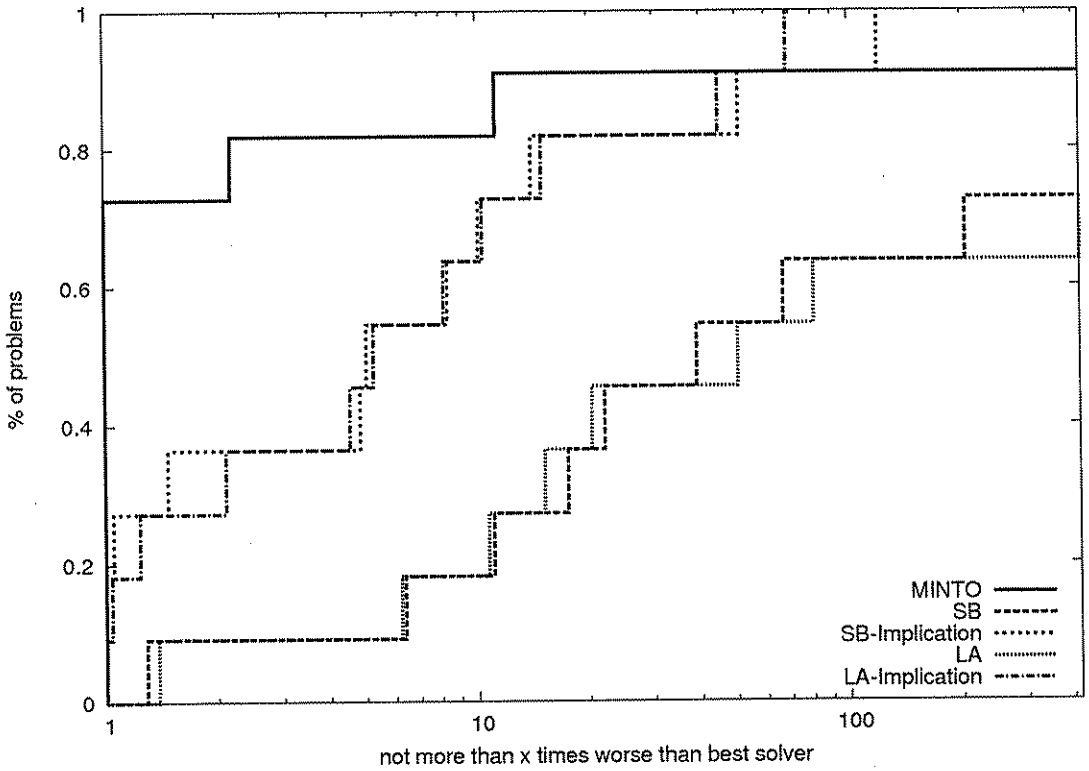

Figure 4: Performance Profile of Running Time in Solved Instances

3 Abbreviated Lookahead Branching

The initial experiment led us to believe that measuring the impact on grandchild nodes when making a branching decision can reduce the number of nodes of the search tree, in large part due to additional bound fixing and implications that can be derived. However, the time required to perform such a method can be quite significant. Our goal in this section is to develop a practical lookahead branching method. An obvious way in which to speed the process up is to consider fixing bounds on only certain pairs of variables, and then to limit the number of simplex iterations used to gauge the change in bound at the resulting grandchild nodes.

3.1 Algorithm

To describe the method employed, we use similar notation as for the original method described in Section 2. Figure 5 shows the notation we use for the values of the LP relaxations of the partially-expanded search tree, and the indicator variables if a particular grandchild node would be fathomed.

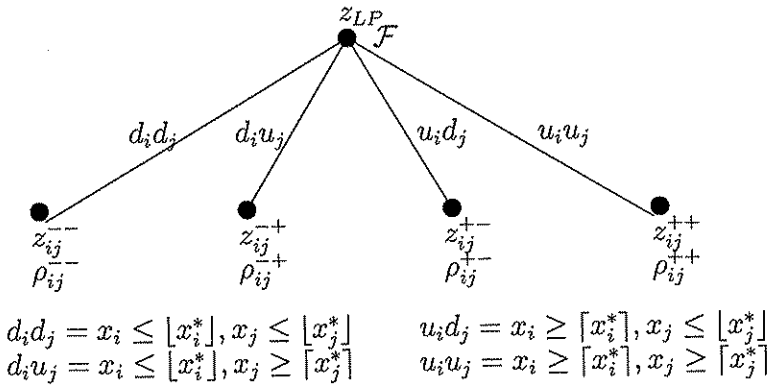

Figure 5: Notations for Abbreviated Lookahead Search Tree

For a pair of variables (x_i, x_j) whose values in the current LP relaxation are fractional, we create the four subproblems denoted in Figure 5 and do a limited number of dual simplex pivots in order to get an upper bound on the values $z_{ij}^{--}, z_{ij}^{-+}, z_{ik}^{+-}$, and z_{ik}^{++}. The obvious questions we must answer are *how* to choose the pair of variables (x_i, x_j), and *how many* simplex pivots should be performed.

3.2 Strong Branching Implementation

The questions of how to choose candidate variables and how many pivots to perform on each candidate must also be answered when implementing a (one-level) strong branching method. Since our goal is to show possible benefits of the lookahead method, we also implement a practical strong branching method with which to compare the abbreviated lookahead method. When implementing strong branching, there are two main parameters of interest:

- The number of candidate variables to consider, (or the size of the set $\mathcal{C}$), and
- the number of simplex pivots (down and up) to perform on each candidate.

Our choice to limit the size of the candidate set is based on how many fractional variables there are to consider in a solution $(\hat{x}, \hat{y})$ whose objective value is z_{LP}. Specifically, we let the size of this set be

$$|\mathcal{C}| = \max\{\alpha|\mathcal{F}|, 10\}, \tag{10}$$

where $\mathcal{F}$ is the set of fractional variables in $\hat{x}$, and $0 \leq \alpha \leq 1$ is a parameter whose value we will determine through a set of experiments. When branching, the variables are ranked from largest to smallest according to the fractionality of $\hat{x}_i$, i.e. the criteria $\min(f_i, 1 - f_i)$, where $f_i = \hat{x}_i - \lfloor \hat{x}_i \rfloor$ is the fractional part of $\hat{x}_i$. The top $|\mathcal{C}|$ variables are chosen as potential branching candidates. For each candidate variable x_i, β dual simplex iterations are performed for each of the down and up branch, resulting in objective values z_i^- and z_i^+. The variable selected for branching is the one with

$$i^* \in \arg\max_{i \in \mathcal{F}} \{\mathcal{W}(z_{LP} - z_i^-, z_{LP} - z_i^+)\}. \tag{11}$$

More sophisticated methods exist for choosing the candidate set $\mathcal{C}$. For example, the variables could be ranked based on the bound change resulting from one dual simplex pivot (akin to the penalty method of Driebeek [12]), or even a dynamic method, in which the size of the set considered is a function of the bound changes seen on child nodes to date, like the method of Achterberg, Koch, and Martin [1]. We denote by $\text{SB}(\hat{\alpha}, \hat{\beta})$ the strong branching method with parameters $\hat{\alpha}$ and $\hat{\beta}$. In our experiments, strong branching was implemented using the CPLEX routine `CPXstrongbranch()` [9].

3.3 Lookahead Branching Implementation

When implementing the abbreviated lookahead branching method, we must determine

- the number of candidate variable pairs to consider, and
- the number of simplex iterations to perform on each of the four grandchild nodes for each candidate pair.

Our method for choosing the set of candidate pairs D works as follows. First, a limited strong branching $\text{SB}(\hat{\alpha}, \hat{\beta})$ is performed, as described in Section 3.2. Then, the variables are ranked from largest to smallest using the same criteria as in strong branching, namely $\mathcal{W}(z_{LP} - z_i^-, z_{LP} - z_i^+)$. From these, the best γ candidates are chosen, and for *each* pair of candidate variables coming from the best γ, δ dual simplex iterations are performed on the four grandchild nodes, resulting in the values $z_{ij}^{s_1 s_2}$ and $\rho_{ij}^{s_1 s_2}$ of Figure 5. If $\hat{\alpha}$ and $\hat{\beta}$ are the parameters for the limited strong branching, and $\hat{\gamma}$, $\hat{\delta}$

are the parameters defining the size of the candidate set of variable pairs and number of pivots on each grandchild node to perform, then we will refer to the branching method as $\text{LA}(\hat{\alpha}, \hat{\beta}, \hat{\gamma}, \hat{\delta})$. Note that the set D consists of all pairs of the best γ candidate variables from the limited strong branching. It may not be necessary to consider each pair, and we will consider other mechanisms for choosing the variable pairs as a line of future research.

3.4 Computational Results

A first set of experiments was performed to determine good values for the branching parameters α, β, γ, and δ. Subsequently, we compared the resulting strong branching and abbreviated lookahead branching methods with the default branching scheme of MINTO v3.1. MINTO v3.1 uses a combination of (once-initialized) pseudocosts and the penalty method of Driebeek [12]. See Linderoth and Savelsbergh [22] for a complete explanation of this method. For these experiments, we used a test suite of 88 instances from MIPLIB 3.0 [5], MIPLIB 2003 [24], and instances available at COR@L [8]. Besides the test suite of problems, all other characteristics of these experiments are the same as those described in Section 2.3.

3.4.1 Strong Branching Parameters

Our first goal is to determine reasonable values for α and β to use in our strong branching method $\text{SB}(\alpha, \beta)$. Doing a search of the full parameter space for α and β would have required prohibitive computational effort, so instead we employed the following mechanism for determining reasonable default values for α and β. The number of simplex iterations was fixed to $\beta = 5$, and an experiment was run to determine a good value of α given that $\beta = 5$. Figure 6 shows a performance profile of this experiment, wherein we have implemented the branching rules $\text{SB}(\alpha, 5)$ for $\alpha = 0.25, 0.5, 0.75,$ and 1.0. The result of the experiment shows that $\alpha = 0.5$ gives good relative results. Namely, considering a half of the fractional variables as branching candidates results in good computational behavior.

Next, we ran an experiment comparing the branching rules $\text{SB}(0.5, \beta)$ for $\beta = 5, 10,$ and 25. Figure 7 summarizes the results of this experiment in the performance profile. There is no clear winner in this experiment, but the value $\beta = 10$ appears to perform reasonably well.

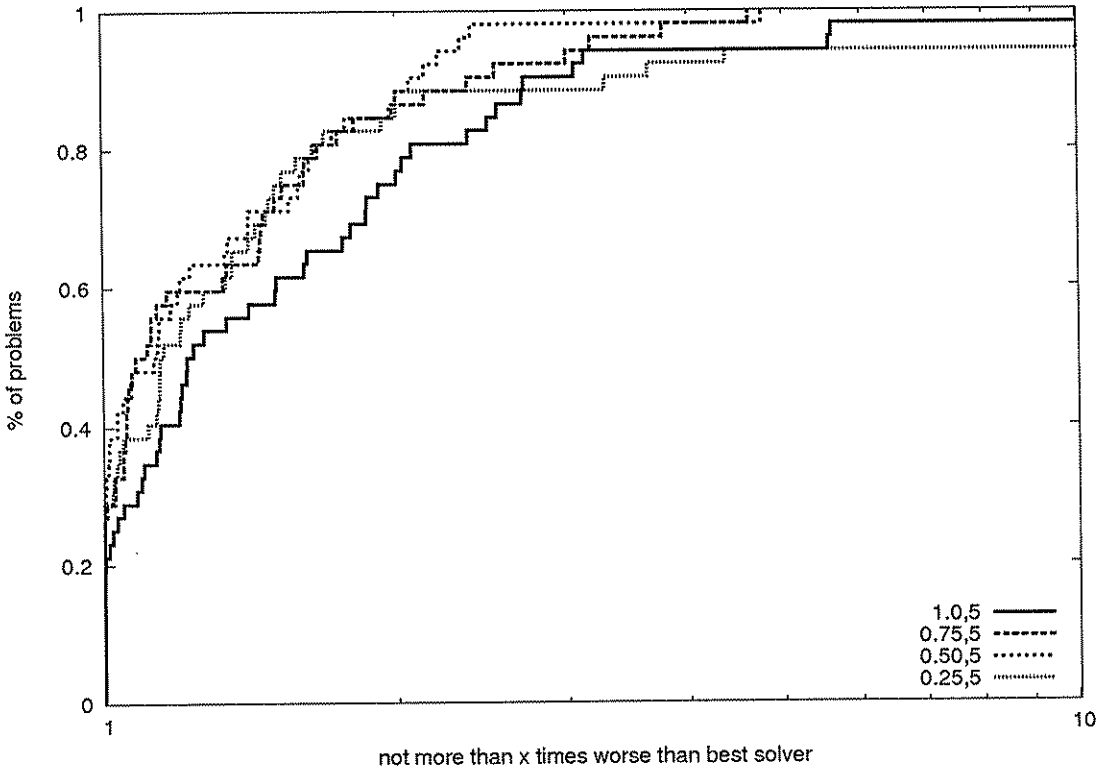

Figure 6: Performance Profile of Running Time as α Varies

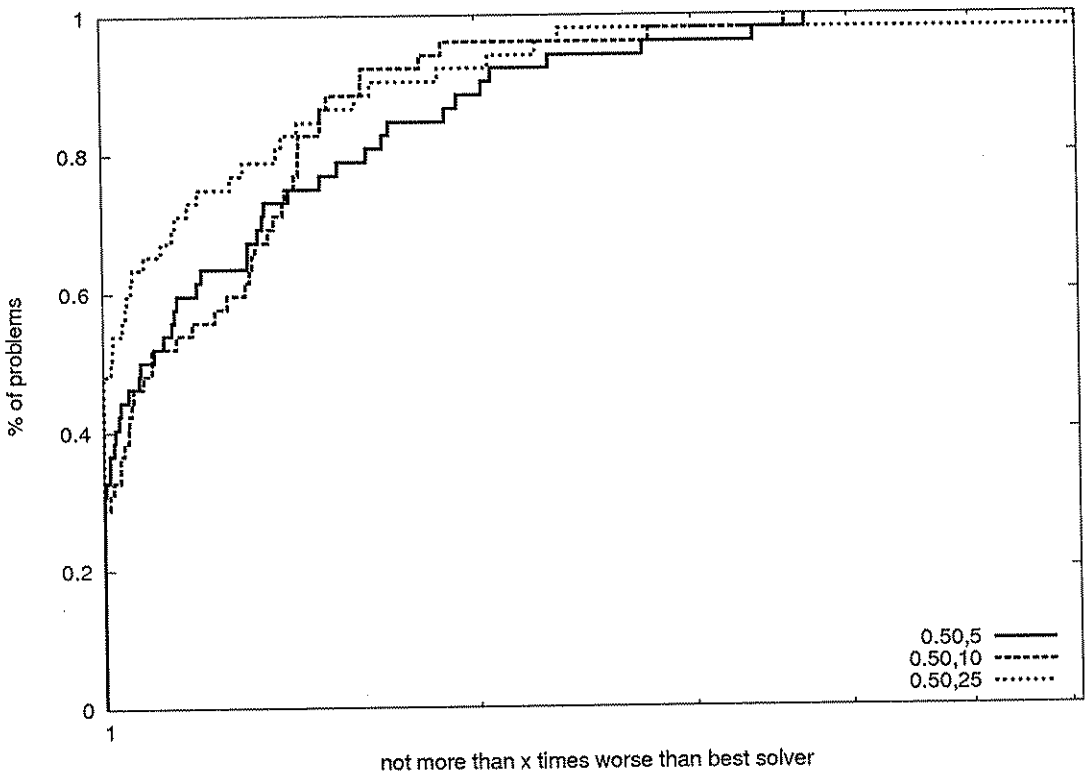

Figure 7: Performance Profile of Running Time as β Varies

3.4.2 Lookahead Branching Parameters

This experiment is designed to determine appropriate values for the number of branching candidates and the number of simplex pivots, i.e. parameters γ and δ respectively, in the abbreviated lookahead branching method, LA($\alpha, \beta, \gamma, \delta$). In this experiment, we have fixed the values for $(\alpha^*, \beta^*) = (0.5, 10)$ as determined in Section 3.4.1. To find appropriate values for γ and δ, we follow the similar strategy to the one that was used to determine α^* and β^*. First, we fix a value of $\delta = 10$, and compare the performance of branching rules LA(0.5, 10, γ, 10). The results of this experiment are summarized in Figure 8. We conclude from the experiment that $\gamma = 3$ is a reasonable parameter value.

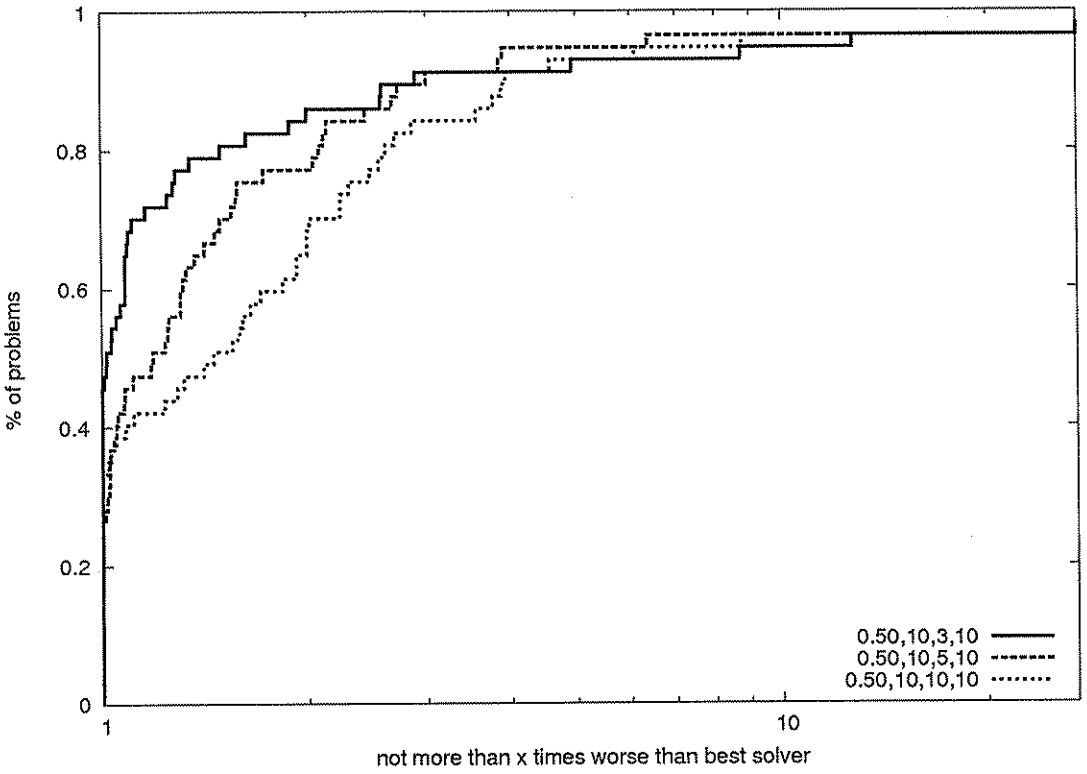

Figure 8: Performance Profile of Running Time as γ Varies

Given that $\gamma = 3$, the next experiment compared branching methods LA($0.5, 10, 3, \delta$) for $\delta \in \{5, 10, 15, 20, 25\}$. The results of this experiment are summarized in the performance profile in Figure 9. There is also no clear winner in this experiment, but we prefer the smaller number of simplex pivots. Therefore, we select the good parameter settings

for the abbreviated lookahead branching method to be $(\alpha^*, \beta^*, \gamma^*, \delta^*) = (0.5, 10, 3, 10)$.

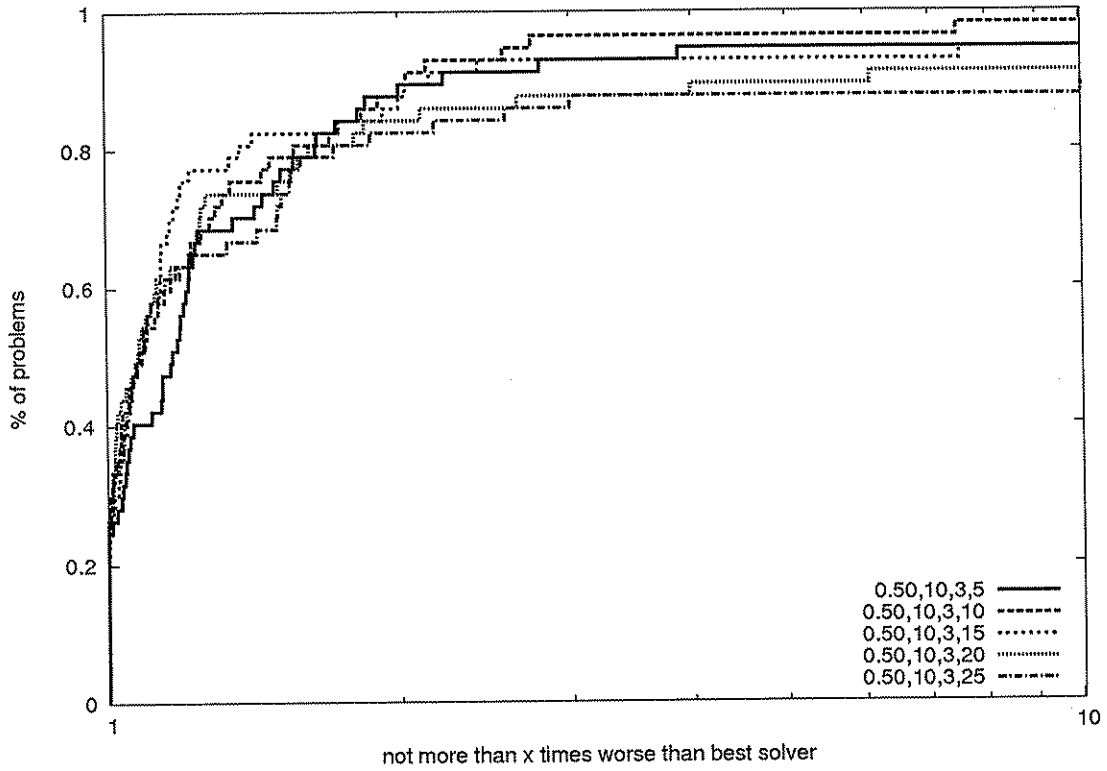

Figure 9: Performance Profile of Running Time as δ Varies

3.4.3 Full Strong Branching and Abbreviated Lookahead Branching Comparison

This experiment is aimed at determining if the limited grandchild information obtained from the abbreviated lookahead branching method could reduce the number of nodes in the search tree significantly. Namely, we compare the branching methods SB(α^*, β^*) and LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$. For sure, the abbreviated lookahead branching method will not be at all effective if the number of nodes in LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$ is not significantly less than SB(α^*, β^*) since the amount of work done to determine a branching variable is significantly larger in LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$ than for SB(α^*, β^*).

Figure 10 is the performance profile comparing the number of nodes evaluated in the two methods. The number of nodes in the abbreviated lookahead method is substan-

tially less than the strong branching. A somewhat more surprising result is depicted in Figure 11, which shows that LA($\alpha^*, \beta^*, \gamma^*, \delta^*$) also dominates SB($\alpha^*, \beta^*$) in terms of CPU time.

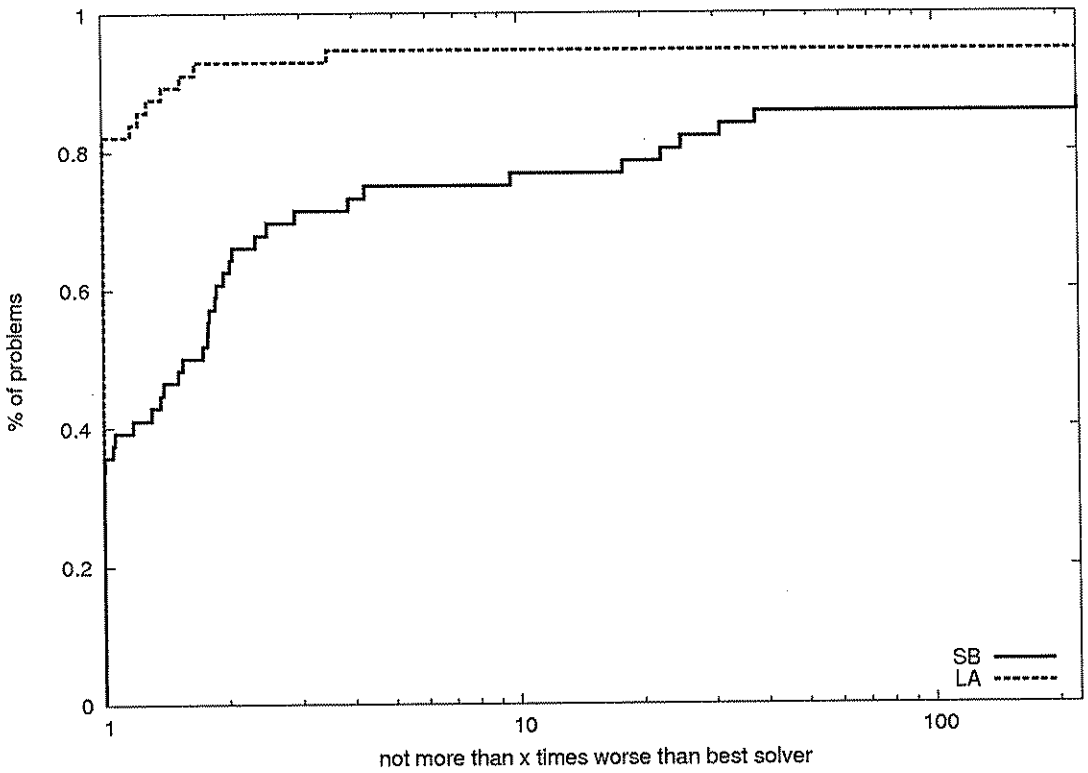

Figure 10: Performance Profile of Number of Evaluated Nodes for SB and LA

3.4.4 Final Comparison

The final experiment is the most practical one, aimed at determining for a fixed maximum number of simplex iterations, whether these iterations are most effectively used evaluating potential child node bounds or grandchild node bounds. Namely, we would like to compare a strategy SB(α_1, β_1) against a strategy LA($\alpha_2, \beta_2, \gamma, \delta$) for parameter values such that

$$2|\mathcal{C}_1|\beta_1 = 2|\mathcal{C}_2|\beta_2 + 4\frac{\gamma(\gamma-1)\delta}{2}. \tag{12}$$

The LHS of equation (12) is the maximum number of pivots that the method SB(α_1, β_1) will perform, where $|\mathcal{C}_1|$ is computed from equation (10). Similarly, the RHS

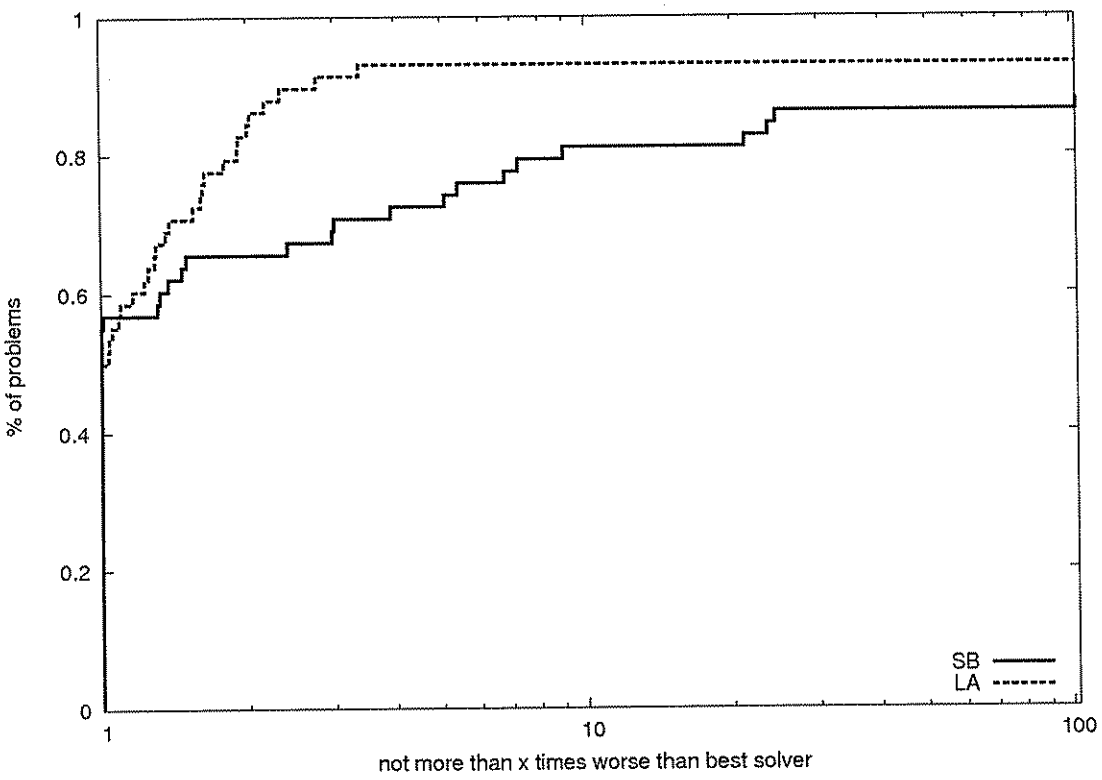

Figure 11: Performance Profile of Running Time for SB and LA

is the maximum number of pivots that the method LA($\alpha_2, \beta_2, \gamma, \delta$) can perform. For this experiment, γ and δ were fixed at the settings determined in Section 3.4.2, namely $\gamma^* = 3$ and $\delta^* = 10$. We used a value of $\alpha_1 = 1$ and $\alpha_2 = 0.5$ so that $|\mathcal{C}_2| = 0.5|\mathcal{C}_1|$. Finally, we set $\beta_1 = 25$, and β_2 is computed from equation (12).

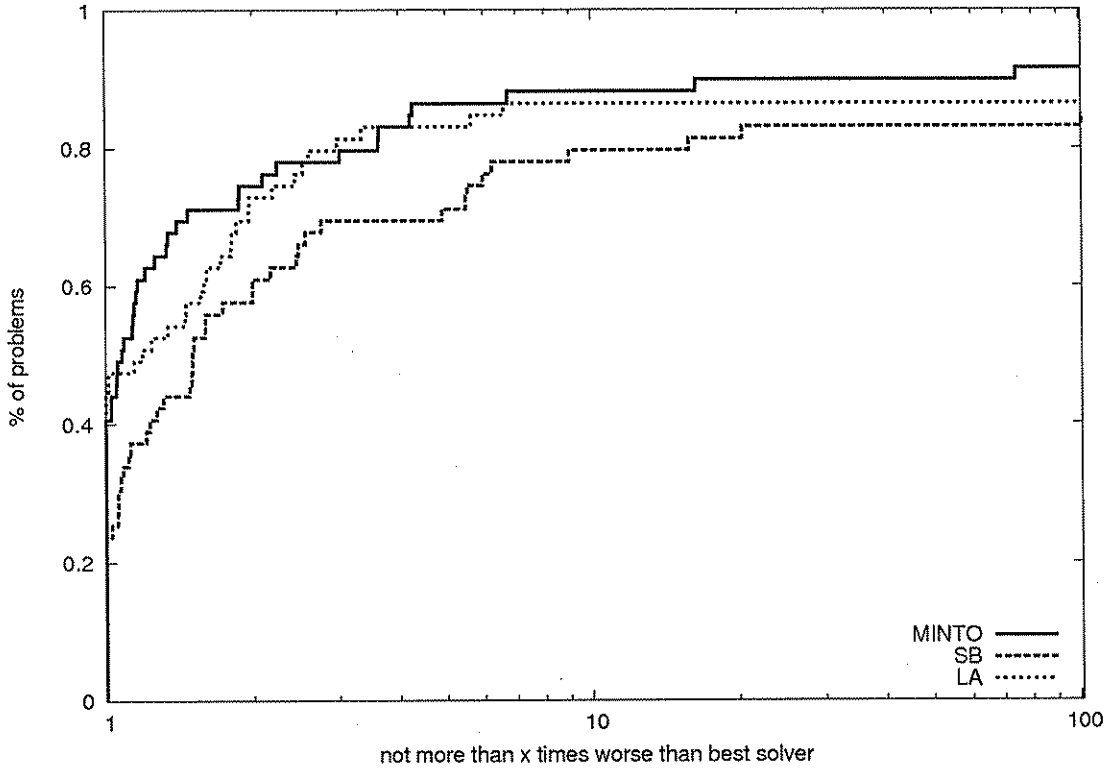

Figure 12: Performance profile of Running Time with Fixed Number of Pivots

The experimental results are summarized in the performance profile of Figure 12. For a fixed number of simplex iterations, abbreviated lookahead branching outperforms strong branching. Further, while not quite as effective as the default MINTO branching method, abbreviated lookahead branching appears to be a reasonable practical branching method if performed in a limited fashion.

4 Conclusion

We have asked and partially answered the question of whether or not consider branching information from grandchild nodes can result in smaller search trees. We have proposed

a method for gathering the information from grandchild nodes. We verified that this information can often be quite useful in reducing the total number of nodes in the search, can result in fixing bounds on variables, and can often give implications between variables. Finally, we showed that by the limiting number of simplex iterations or the number of fractional variables for which to generate the branching information, a similar branching decision can still be made, but with less computational effort. The resulting branching rule is of comparable quality to the advanced branching method available in the MINTO software system. From our experience, it seems unlikely that lookahead branching will be a good default branching scheme for general MIP, but for some classes of hard problem instances, the additional computational effort may well pay off.

Acknowledgement

The authors would like to thank the fellow members of the COR@L lab (`http://coral.ie.lehigh.edu`) for a variety of insightful discussions during the course of this work. This work is supported in part by the US National Science Foundation (NSF) under grants OCI-0330607 and DMI-0522796 and by the US Department of Energy under grant DE-FG02-05ER25694. Computational resources are provided in part by equipment purchased by the NSF through the IGERT Grant DGE-9972780.

Appendix

A Tables of Results

A value of -1 indicates that the instance was not solved within the time limit of 8 hours.

	Number of Evaluated Nodes			
Name	SB	SB-Implication	2-Level	2-Level-Implication
l152lav	193	11	281	9
p0548	3	3	3	3
rgn	1011	127	1296	119
stein45	16437	7491	20409	8755
vpm2	4883	381	4291	527
misc07	8173	163	5219	31
modglob	159	29	199	79
p2756	7	3	7	3
aflow30a	-1	15	-1	45
pk1	-1	24001	-1	13731
qiu	-1	5	-1	5

Table 4: Solved MIPLIB Instances

	Integrality Gap			
Name	SB	SB-Implication	2-Level	2-Level-Implication
opt1217	23.88	23.88	24.07	24.07
pp08a	10.63	10.66	11.55	11.38
aflow40b	13.25	6.91	13.09	5.90
danoint	4.38	4.38	4.49	4.41
swath	32.47	30.66	30.27	29.29

Table 5: Unsolved MIPLIB Instances

	Total Running Time				
Name	MINTO	SB	SB-Implication	2-Level	2-Level-Implication
l152lav	8.84	1808.68	1059.83	3678.86	606.83
p0548	0.21	0.27	0.31	0.29	0.26
rgn	2.91	32.13	24.18	31.13	23.66
stein45	296.94	19820.37	15107.89	23863.70	13323.79
vpm2	23.71	522.02	144.39	483.05	107.62
misc07	790.96	13852.87	8000.36	11993.02	8201.39
modglob	2.95	115.14	41.55	148.21	44.25
p2756	2.92	18.52	14.70	18.10	15.34
aflow30a	1842.31	-1	844.3	-1	1779.79
pk1	-1	-1	4928.79	-1	4685.71
qiu	5112.87	-1	454.33	-1	472.78

Table 6: Total Running Time of Solved MIPLIB Instances

References

[1] T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters, 33 (2004), pp. 42–54.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *On the solution of traveling salesman problems*, in Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians, 1998, pp. 645–656.

[3] A. Atamtürk, G. Nemhauser, and M. W. P. Savelsbergh, *Conflict graphs in solving integer programming problems*, European Journal of Operational Research, 121 (2000), pp. 40–55.

[4] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent, *Experiments in mixed-integer linear programming*, Mathematical Programming, 1 (1971), pp. 76–94.

[5] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh, *An updated mixed integer programming library: MIPLIB 3.0*, Optima, 58 (1998), pp. 12–15.

[6] R. Breu and C. A. Burdet, *Branch and bound experiments in zero-one programming*, Mathematical Programming, 2 (1974), pp. 1–50.

[7] N. Brixius and K. Anstreicher, *Solving quadratic assignment problems using convex quadratic programming relaxations*, Optimization Methods and Software, 16 (2001), pp. 49–68.

[8] *COR@L: Computational Optimization Research @ Lehigh*, 2005. `http://coral.ie.lehigh.edu`.

[9] CPLEX Optimization, Inc., *Using the CPLEX Callable Library, Version 9*, Incline Village, NV, 2005.

[10] Dash Associates, *XPRESS-MP Reference Manual*, 2004. Release 2004.

[11] E. Dolan and J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming, 91 (2002), pp. 201–213.

[12] N. J. Driebeek, *An algorithm for the solution of mixed integer programming problems*, Management Science, 12 (1966), pp. 576–587.

[13] J. Eckstein, *Parallel branch-and-bound methods for mixed integer programming*, SIAM News, 27 (1994), pp. 12–15.

[14] J. Eckstein, C. A. Phillips, and W. E. Hart, *PICO: An object-oriented framework for parallel branch-and-bound*, in Proc. Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications, 2001, pp. 219–265.

[15] M. Fampa and K. M. Anstreicher, *An improved algorithm for computing steiner minimal trees in Euclidean d-space*, working paper, Dept. of Management Sciences, University of Iowa, 2006.

[16] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin, *Practical solution of large scale mixed integer programming problems with UMPIRE*, Management Science, 20 (1974), pp. 736–773.

[17] J. M. Gauthier and G. Ribière, *Experiments in mixed-integer linear programming using pseudocosts*, Mathematical Programming, 12 (1977), pp. 26–47.

[18] A. M. Geoffrion and R. E. Marsten, *Integer programming algorithms: A framework and state-of-the-art survey*, Management Science, 18 (1972), pp. 465–491.

[19] W. GLANKWAMDEE, *Lookahead branching for mixed integer programming*, Master's thesis, Department of Indsutrial and Systems Engineering, Lehigh University, Bethlehem, PA, 2004.

[20] A. H. LAND AND A. G. DOIG, *An automatic method for solving discrete programming problems*, Econometrica, 28 (1960), pp. 497–520.

[21] J. T. LINDEROTH, *Topics in Parallel Integer Optimization*, PhD thesis, Georgia Institute of Technology, 1998.

[22] J. T. LINDEROTH AND M. W. P. SAVELSBERGH, *A computational study of search strategies in mixed integer programming*, INFORMS Journal on Computing, 11 (1999), pp. 173–187.

[23] LINDO SYSTEMS INC., *LINDO User's Manual*, Chicago, IL, 1998.

[24] A. MARTIN, T. ACHTERBERG, AND T. KOCH, *MIPLIB 2003*. Available from `http://miplib.zib.de`.

[25] G. MITRA, *Investigation of some branch and bound strategies for the solution of mixed integer linear programs*, Mathematical Programming, 4 (1973), pp. 155–170.

[26] G. NEMHAUSER AND L. A. WOLSEY, *Integer and Combinatorial Optimization*, John Wiley and Sons, New York, 1988.

[27] G. L. NEMHAUSER, M. W. P. SAVELSBERGH, AND G. C. SIGISMONDI, *MINTO, a Mixed INTeger Optimizer*, Operations Research Letters, 15 (1994), pp. 47–58.

[28] T. K. RALPHS, *SYMPHONY 5.0*, 2004. Available from `http://www.branchandcut.org/SYMPHONY/`.

[29] M. W. P. SAVELSBERGH, *Preprocessing and probing techniques for mixed integer programming problems*, ORSA Journal on Computing, 6 (1994), pp. 445–454.

[30] D. VANDENBUSSCHE AND G. L. NEMHAUSER, *A branch-and-cut algorithm for nonconvex quadratic programs with box constraints*, Mathematical Programming, 102 (2005), pp. 559–575.

[31] L. A. WOLSEY, *Integer Programming*, John Wiley and Sons, New York, 1998.