Branch and Cut: An Empirical Study

A Dissertation
Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Industrial Administration
(Operations Research)

by Miroslav Karamanov
Tepper School of Business
Carnegie Mellon University

September 2006

Branch and Cut: An Empirical Study

Approved:

Gérard Cornuéjols, Chairman

Egon Balas

John Hooker

François Margot

Reha Tütüncü, External reader

Date approved: _____

To my parents Maria and Stoyan

To my wife Dilyana

Acknowledgments

This dissertation would not be possible without those whose love, understanding, experience, and knowledge have inspired and supported me during my Ph.D. study.

First of all, I would like to thank my advisor, Prof. Gérard Cornuéjols, for his advice and guidance. As a skilled researcher and excellent teacher, he provided me with much insight and stimulated my scientific curiosity. I am grateful for the inspiration, encouragement and support.

I would also like to thank Prof. Egon Balas and Prof. François Margot for the research collaborations and fruitful discussions. I thank Prof. John Hooker for his guidance and useful discussion on the importance of rigorous experimental design and statistical analysis in an empirical study. I thank all of them as well as Prof. Reha Tütüncü for participating in my committee.

I am indebted also to Prof. Rumen Raev for early on invoking my interest in mathematics and fostering my progress on my way to Carnegie Mellon.

There are many people who have made these six years in Pittsburgh very enjoyable. My best wishes go out to Tallys and Renata, and Atul and Kristen with thanks for all the time together. Thanks to Erlendur, Amitabh, Jochen, Kent, Luis, Xinming, Yanjun, Hakan, Teresa, Latife, Viswanath, Mohit, John, Vineet, Elina, Valentin and Gergana for their friendship and and support.

I am very grateful to my parents for their love and encouragement to follow my dreams, and support for every decision I made.

Last, but not least, I thank my wife Dilyana from the bottom of my heart for her never ending love, support, and patience. Whatever I have achieved, was achieved together.

Contents

Chapter 1

Introduction and preliminaries

1.1 Introduction

Many "real world" optimization problems have a discrete nature and can be formulated as Mixed Integer Linear Programming (MILP) problems. Various solution techniques for this class of problems have been proposed over the last 50 years. Still, obtaining exact solutions to these problems is difficult. The inherent complexity of MILP problems is partially explained by the fact that they are NP-hard. Nevertheless, the advances in the algorithms over the last decade and the increased power of computers improved dramatically our abilities to solve large problem instances. This dissertation proposes ideas for further improvement of the state-of-the-art solution techniques.

There are two major methods for solving Mixed Integer Linear Programming problems: cutting-plane algorithms and branch-and-bound algorithms. In a *cutting-plane algorithm*, valid inequalities that cut off the current solution of the Linear Programming relaxation are used to tighten the formulation until an integer feasible solution is found. This method can be traced back to the work of Danzig, Fulkerson, and Johnson [29] who apply it to successfully solve a 48-city traveling salesman problem. Gomory [34, 36, 35, 37] proposed the well-known *fractional cuts* and *mixed integer cuts*, as well as a general solution procedure for pure and mixed 0-1 programs. *Branch and bound* is a divide-and-conquer algorithm that searches the feasible set of the Linear Programming relaxation of the MILP problem. It implicitly enumerates the feasible solutions in a quest for a proof of optimality. This method originates with the work of Land and Doig [41] for general MILP and of Balas [10] for pure

0-1 programs.

Branch and bound has been the solution method of choice in the 70s and 80s. It has been implemented in all commercial software packages for MILP. During this period, cutting-plane algorithms were considered of less practical value as independent solution techniques. Their use was limited to tightening the formulation before the start of branch and bound in what is now called *cut and branch*.

It was in the early 90s when a more efficient combination of the two solution methods was proposed. Padberg and Rinaldi [55] were the first to propose generating cutting planes at the nodes of the search tree — a method they called *branch and cut*. They applied this algorithm to the traveling salesman problem generating combinatorial cuts and showed impressive results. Balas et al. [14] demonstrated that applying mixed integer Gomory cuts in a branch-and-cut framework provides a powerful algorithm for solving general MILP problems. Today, branch and cut is the state-of-the-art algorithm for this class of optimization problems. It is implemented in all MILP solvers.

Branch and cut is a complex algorithm in which cut generation, LP reoptimization and various search rules, such as node selection, branching variable selection, and child selection procedures, interact. The efficiency of the algorithm, measured by the solution time and even by the ability to solve a problem, is largely influenced by this interaction. The complex nature of the algorithm makes theoretical studies of particular components of the algorithm difficult to integrate into an overall understanding. Therefore, in our effort to evaluate our modifications to the algorithm, we resort to the common scientific method for analysis when dealing with a complex object: an empirical study.

One of the important decisions made in the branch-and-cut algorithm is the choice of a branching object. Traditionally, in general purpose MIP solvers, branching objects are variables — the "best" candidate is chosen among the integer variables with fractional values in the current basic solution. A more general approach is to branch on general split disjunctions. We propose an efficient implementation of this idea in Chapter 2. We select promising branching disjunctions based on a heuristic measure of disjunction quality. This measure exploits the relation between branching disjunctions and intersection cuts. In this work, we focus on disjunctions defining the mixed integer Gomory cuts at an optimal basis of the linear programming relaxation. The procedure is tested on instances from the

literature. Experiments show that branching on general disjunctions is more efficient than branching on variables for a majority of the instances.

Contemporary MILP solvers make extensive use of cuts in their branch-and-cut algorithms. Some widely used classes of general cuts, such as mixed integer Gomory, mixed integer rounding, and lift-and-project cuts, are essential for the efficient solution of large MILP problems. Fortunately, increased power of computers allows generating a large number of these cuts in a short interval of time. Unfortunately, adding all of these cuts to the formulation causes a range of negative effects. Aggressive cut generation increases the size of the formulation significantly and slows down the solution of the LP relaxations. It can cause numerical problems as well. Furthermore, it can deteriorate the facial structure of the polyhedron. The consequence is that future rounds of cuts are less deep and less efficient. In Chapter 3, we study the effect of cuts on the performance of branch and cut. We propose criteria for cut selection and a cut selection algorithm that decreases the negative effects. The novelty in our approach is that it evaluates the quality of the cuts as a group, as opposed to evaluating only individual properties. We test the performance of the cut selection routine empirically.

Research shows that the performance of branch and cut depends on the decisions with respect to cut generation and on the choice of branching rules, among other decisions. The effect of these choices on the size of the branching tree is significant but not very well studied. As a result, it is believed that predicting the running time of the algorithm is practically impossible. This is supported by real-life experience showing large variability among instances. In Chapter 4, we try to challenge this belief by showing that the size of the branching tree, therefore, the running time, can be roughly predicted in an early phase of the solution process. We consider a specific set of decision rules, the ones implemented in the out-of-the-box CPLEX 8.0, and construct a simple sampling procedure for predicting the number of nodes in the tree. We tested the procedure in cut-and-branch and branch-and-cut frameworks.

1.2 Branch and cut

In this section, we provide an outline of the branch-and-cut algorithm followed by a description of its key components. For a more detailed discussion on branch and cut, refer to Nemhauser and Wolsey [52] and Wolsey [58]. Recent computational studies of branching rules are Linderoth and Savelsbergh [43], and Achterberg et al. [3].

1.2.1 Outline of the algorithm

Consider the Mixed Integer Linear Program:

$$z^* = \min\{c^T x : Ax \geq b,\, x_j \in \mathbb{Z} \text{ for } j \in N_I\}, \tag{1.1}$$

where $c, x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and $N_I \subseteq N := \{1, 2, \ldots, n\}$. Let $X_{\text{MILP}} = \{x : Ax \geq b,\, x_j \in \mathbb{Z} \text{ for } j \in N_I\}$ denote the feasible set of (1.1). The linear relaxation of problem (1.1) is obtained by removing the integrality constraints:

$$z(P_{\text{LP}}) = \min\{c^T x : x \in P_{\text{LP}}\}, \tag{1.2}$$

where $P_{\text{LP}} = \{x : Ax \geq b\}$. Let $z(P_{\text{LP}}) = +\infty$ if $P_{\text{LP}} = \emptyset$.

A generic branch-and-cut algorithm is shown in Figure 1.1. The algorithm starts with a single subproblem to solve: the LP relaxation of the mixed integer linear program. During the course of the algorithm, new subproblems are created by *branching*: The solution of a subproblem may lead to the creation of two or more *children* subproblems. This process can be represented by a tree where the nodes correspond to subproblems and the edges represent the parent-child relation between subproblems. In this context, the terms node and subproblem are considered synonymous.

Detailed explanation of the different steps is provided below.

1.2.2 Branching

Node selection rules

The order in which we search the branching tree is of great importance for the efficiency of the branch-and-cut algorithm. This order is enforced by a rule for selecting the next node to be processed. It is essentially a rule for comparing two nodes. The unsolved, pending

Input: An MILP problem of the type (1.1).

Output: An optimal solution $x^* \in X_{\text{MILP}}$ and its objective value $z^* = c^T x^*$, or the conclusion that $X_{\text{MILP}} = \emptyset$, denoted by $z^* = +\infty$.

Initialize: Queue of active subproblems $Q := \{P_{\text{LP}}\}$,
$z^* := +\infty$.

1. If $Q = \emptyset$, exit by returning the optimal solution x^* with value z^*.
2. Select a problem P from the queue Q.
3. Solve the linear program $z(P) = \min\{c^T x : x \in P\}$ with optimal solution $\bar{x}(P)$.
4. Apply reduced cost fixing.
5. If cuts should be generated, goto Step 6, otherwise, goto Step 7.
6. Generate a round of cuts, $Dx \geq e$, where $D \in \mathbb{R}^{k \times n}$, $e \in \mathbb{R}^k$, and k is the number of generated cuts.
   Add them to the formulation: $P := P \cap \{x : Dx \geq e\}$.
   Goto Step 3.
7. If $z(P) \geq z^*$, goto Step 2.
8. If $\bar{x}(P) \in X_{\text{MILP}}$, update the incumbent: $x^* = \bar{x}(P)$ and $z^* := z(P)$.
9. Branching: Split P into subproblems and add them to Q.
   Goto Step 2.

Figure 1.1: Branch-and-cut algorithm

subproblems, also called *active* nodes, are kept in a pool. It is denoted by Q in Figure 1.1. The node selection rule induces ordering of the active nodes and, as a result, the pool of active nodes can be viewed as a priority queue where the first node is the most preferred.

Several rules have been explored in the literature and used in practice.

Depth-first search. This is the rule used by the algorithms by Balas [10], Dakin [28] and Little et al. [44]. According to this rule, new nodes are added to the front of the queue and the next node to be processed is the last added. An advantage of this strategy is its memory efficiency: the number of active nodes in the queue is never larger than the depth of the currently processed node. The main drawback is that the search may be conducted first in parts of the tree where there are no good feasible solutions, thus increasing the total

solution time.

Best bound. This is the rule used by the branch-and-bound algorithm of Land and Doig [41]. For every active subproblem, the branch-and-bound algorithm keeps a lower bound on the objective value. This lower bound is obtained from the objective value of the parent node or from strong branching if such has been applied. The best-bound rule selects the node with the smallest lower bound among all active nodes. This rule minimizes the number of explored nodes before completing the search. It guarantees that no node that could be pruned by the optimal objective value would be solved. As a drawback, memory requirements for a search by best-bound may be prohibitive. In addition, one subproblem to be optimized differs significantly from the previous subproblem solved (in contrast to depth-first search), which leads to a longer solution time.

Best-dive. This rule combines the strengths of the above two. It dives along a path until pruned, then selects the best-bound node and dives again. Diving in a depth-first fashion is computationally efficient. In addition, it helps find new feasible solutions faster. This leads to a faster improvement in the upper bound, hence a decrease the amount of enumeration. Choosing a best-bound node as a starting point of the next dive suggests that an area of the search tree with some potential for a good solution is explored. As a result, best-dive is a balanced and efficient search algorithm.

In diving, as well as in depth-first search, one of the two newly created nodes is explored next and the other one is sent to the queue. The most common rules for selecting the next processed node are the following:

Lowest LP bound: The child with smaller LP bound is selected.

Integer infeasibility: Let $I = \sum_{j \in N_I} \min\{\bar{x}_j(P) - \lfloor \bar{x}_j(P) \rfloor, \lceil \bar{x}_j(P) \rceil - \bar{x}_j(P)\}$ be the sum of the infeasibility of all integer variables. The child that minimizes I is selected.

Branching objects

The outcome of the solution of a subproblem $z(P)$ is one of the following:

- P is proven infeasible, i.e. $z(P) = +\infty$;
- the optimal solution is integer: $\bar{x}(P) \in X_{\text{MILP}}$;
- the optimal value is above the upper bound: $z(P) \geq z^*$, therefore, branching would not bring an improvement in the best known solution;

- the optimal value is below the upper bound: $z(P) < z^*$, and $\bar{x}(P) \notin X_{\text{MILP}}$.

In the first three cases, the subproblem is abandoned and the algorithm proceeds to the next subproblem in the queue Q. In the last case, the algorithm *branches*: creates several children subproblems by splitting the set P into disjoint subsets. For general MILP problems, the widely accepted way of branching divides the feasible set into two subsets by selecting an integer variable x_j with fractional value in the optimal solution of $z(P)$ and imposing new bounds on x_j: $x_j \leq \lfloor \bar{x}_j(P) \rfloor$ in one of the subproblems and $x_j \geq \lceil \bar{x}_j(P) \rceil$ in the other. We call this branching on a *variable dichotomy* or just branching on a variable.

In Chapter 2, we consider more general branching objects: split disjunctions. A split disjunction is a union of two disjoint halfspaces in $\mathbb{R}^n$, such that the union contains all points in $\mathbb{Z}^n$. We show that our procedure for branching on split disjunctions performs better than branching on variables when tested on a diverse set of test instances.

Variable selection rules

Various rules can be used to select the branching variable. All of them are instantiations of the following generic algorithm (Figure 1.2).

Input:	Current subproblem P with an optimal solution $\bar{x}(P) \notin X_{\text{MIP}}$.
Output:	An index $i \in N_I$ of a fractional variable.
Initialize:	Let $C = \{i \in N_I : \bar{x}_i \notin \mathbb{Z}\}$ be the set of candidate branching variables.
1.	For all candidates $i \in C$, calculate a quality measure $\psi_i \in \mathbb{R}$.
2.	Return an index $i \in C$ with $\psi_i = \max_{j \in C}\{\psi_j\}$.

Figure 1.2: Variable selection

Most fractional variable. Consider variable x_j and its value in the solution of the current subproblem, $\bar{x}_j(P)$. The *fractionality* of the variable is defined as: $\min\{\bar{x}_j(P) - \lfloor \bar{x}_j(P) \rfloor, \lceil \bar{x}_j(P) \rceil - \bar{x}_j(P)\}$. The *most-fractional* rule stipulates that the variable with maximum fractionality should be selected. Intuitively, the larger the fractionality, the larger the minimum improvement in the lower bounds of the children nodes. However, a recent computational study suggests that the fractionality has little correlation with the improvement

in the lower bound. Achterberg et al. [3] present experimental results showing that the effect of using this rule is close to that of randomly choosing a branching variable. Although this rule is still popular, more efficient variable selection rules, as measured by the solution time and the size of the branching tree, are in use today. Those are based on estimates of the deterioration in the objective value at the two children and include strong branching, pseudocost branching, and combinations of the two.

Strong branching. Strong branching was introduced by Applegate, et al. [8, 9] It works as follows. Let v and p be two positive integers. Given the solution of the LP relaxation at a node, make a list of v integer variables that have fractional values, using some variable selection rule, e.g. the v most fractional ones. (If there are fewer than v fractional variables, select all of them.) For each of these fractional variables, create the two children nodes and try to solve them performing at most p dual simplex pivots. Let the objective values obtained be z_1 and z_2. (These are valid lower bounds of the objective values of the two children.) Compute a quality measure $\psi(z_1, z_2)$ for the branching variable. Select the variable that maximizes the quality measure and branch on it.

The quality measures used for comparing candidate variables in strong branching and pseudocost branching are functions of the estimated lower bounds on the two children: z_1 and z_2. A typical form of such a function is $\psi(z_1, z_2) = \lambda \min(z_1, z_2) + (1 - \lambda) \max(z_1, z_2)$, for $0 \leq \lambda \leq 1$. Usually, larger weight is given to the first term. The values for λ proposed in the literature vary between 0.66 and 1.

Setting parameter p to a very large number leads to a complete solution of the two children for all considered branching variables. This provides the exact lower bounds for the children. If, in addition, parameter v is large enough so that all fractional variables are considered, we obtain the (locally) best variable to branch on, with respect to the chosen quality measure. This branching rule is called *full strong branching*.

Pseudocost branching. Similarly to strong branching, pseudocost branching estimates the deterioration in the objective value at the two children. Pseudocost branching is less time consuming than strong branching but the price for this are rougher estimates. It was introduced by Benichou et al. [19].

For each integer variable x_i, pseudocost branching keeps a history of the result of previous branchings on x_i. The history is used to estimate the pseudocost: the amount of change in the objective caused by one unit of change in x_i. Separate pseudocosts are kept

for upward and downward change of x_i. When x_i is a candidate for branching, estimates of the lower bounds of the children nodes, $\hat{z}_1$ and $\hat{z}_2$, are obtained from the pseudocosts and the amount of infeasibility. Then, a quality measure similar to the one in strong branching is applied in order to select the branching variable.

Hybrid strong/pseudocost branching. Variable selection rules that combine pseudocosts and strong branching have been developed as well. Strong branching is used to initialize the pseudocosts of a variable the first time it is considered for branching. In addition, strong branching may be used periodically to update the pseudocosts.

A version of a hybrid procedure, called *reliability branching* was recently proposed by Achterberg et al. [3]. It incorporates a measure of reliability of the pseudocosts.

1.2.3 Cutting

Cutting planes are inequalities valid for all feasible points but violated by the solution of the current subproblem. Adding cuts strengthens the formulation and improves the lower bound, which decreases the amount of enumeration. Cuts have a very important role for the solution of difficult problems but their excessive use may cause slowdown and numerical problems.

Cuts in branch-and-cut algorithms for general MILP problems are added in rounds: the algorithm adds a batch of cuts, reoptimizes the LP, then repeats the procedure or turns to branching. The most important decisions in this process concern the intensity of cut generation: How many rounds of cuts to generate? How many cuts to add in a round? At what nodes to generate cuts? A static strategy gives fixed answers that are applied universally to all problem instances. An example of such a strategy is: "Generate at most ten rounds of cuts at the root and one round at every 50th processed node. In every round, generate at most 100 cuts." A dynamic approach answers the above questions based on information collected in the course of the algorithm. E.g., the decision to generate one more round of cuts or switch to branching may depend on the progress made by previous rounds of cuts. Because of the great diversity of MILP problems and the large variation in the performance of branch and cut among problem instances, the dynamic approach is preferable. Unfortunately, not much research is available in this direction. We address the issue of cut selection in Chapter 3.

Another important question for the computational efficiency of branch-and-cut solvers is: When are cuts discarded from the formulation? Keeping all generated cuts throughout the solution process can cause tremendous slowdown. One solution is to remove cuts as soon as they become inactive (non-binding). An alternative strategy is to keep a cut until some constant number of iterations of inactivity, where an iteration is one reoptimization of the subproblem (after a round of cuts or in branching).

Chapter 2

Branching on general disjunctions

2.1 Introduction

Branch and cut is the most widely used algorithm for solving Mixed Integer Linear Programs. Its performance improved by several orders of magnitude in the last decade due to advances in hardware but mostly due to modifications in the algorithm. In this paper, we propose a modification in the branching routine.

One of the important decisions made in the branch-and-cut algorithm is the choice of a branching object. Traditionally, in general purpose MILP solvers, branching objects are variables — the "best" candidate is chosen among the integer variables that have a fractional value in the current basic solution. If an integer-constrained variable, x_j, has a fractional value $\bar{x}_j$, we impose the constraint $x_j \leq \lfloor \bar{x}_j \rfloor$ in one of the children and $x_j \geq \lceil \bar{x}_j \rceil$ in the other. This can be viewed as adding the constraints $\pi^T x \leq \pi_0$ and $\pi^T x \geq \pi_0 + 1$, respectively, where $\pi = e_j$, the j-th unit vector, and $\pi_0 = \lfloor \bar{x}_j \rfloor$. We propose to use a general integer vector π and $\pi_0 = \lfloor \pi^T \bar{x} \rfloor$ for branching, where $\pi_i \in \mathbb{Z}$ if x_i is an integer variable and 0 otherwise. We call a disjunction *simple* when $\pi = e_j$, for some j, and *general* otherwise. General disjunctions are also known as *split disjunctions* [25].

There is an evident trade-off between the two approaches. General disjunctions can lead to a smaller tree size. On the other hand, branching on variables produces LP subproblems that are easier to reoptimize because bounds on the variables do not increase the size of the basis. Branching on a general disjunction adds one row to the formulation of the children subproblems. When this is repeated at every node, the number of constraints can grow

notably leading to an increased solution time of each subproblem. In our experiments, we observe that the decreased tree size usually more than offsets this increase.

One difficulty with the application of the idea for branching on general disjunctions is the infinite number of general disjunctions that are violated by a given basic solution. Optimizing over this set would give the best results but is practically impossible. A natural objective would be to maximize the improvement in the lower bound as a result of branching (we assume here that MILP is a minimization problem). But there is no known way to measure this value before solving the children nodes and, hence, no way to formulate this problem. The intimate relation between split disjunctions and intersection cuts, introduced by Balas [11], provides a proxy for the change in the lower bound. The *depth* of an intersection cut, or *distance cut off*, is a reasonable measure of the cut quality. One may use the depth of the cut as a heuristic measure of the quality of the corresponding disjunction. But even maximizing the depth over the set of all intersection cuts is a difficult MILP problem.

In this paper, we consider a specific class of general disjunctions — the ones defining mixed integer Gomory cuts derived from the tableau [35]. The advantages of this class are that it is finite and fast to generate. Furthermore, disjunctions corresponding to Gomory cuts can be viewed as strengthened simple disjunctions. The algorithm we propose performs a heuristic pre-selection of the most promising disjunctions based on the distance cut off by the corresponding cut, followed by an exact evaluation of the quality of the pre-selected disjunctions. This idea can be applied to other classes of intersection cuts as well, e.g. lift-and-project[12], reduce-and-split [5], and mixed integer rounding cuts [47]. Our approach is explained in detail in Section 2.4. A review of related earlier work is present in Section 2.2. A short introduction to intersection cuts is included in Section 2.3.

In Section 2.5, we describe the experiments we conducted and their results. These experiments measure the gap closed after branching for a fixed number of levels by branch and bound, and show that general disjunctions perform better than simple ones. An interesting observation is that pruning of a child by infeasibility, which is a desirable effect, happens more often when branching on general disjunctions than when branching on simple disjunctions. In a final experiment, we study the performance of our algorithm in a cut-and-branch framework. We also test an algorithm combining branching on variables and general disjunctions.

There is no doubt that branching on general disjunctions can reduce the amount of

enumeration compared to branching on single variables, provided that we know the right disjunctions to branch on. In this paper, we show that such disjunctions can be generated and applied without much computational overhead.

2.2 Literature review

The idea of branching on general disjunctions is not novel. One approach proposed in the literature is to find "thin" directions in the polyhedron of feasible solutions, transform the space so that these directions correspond to unit vectors, and solve the problem in the new space by regular branch and bound, branching on the new variables. Transformed back to the original space, this corresponds to branching on general disjunctions. For detailed descriptions, refer to the algorithms for solving integer programming problems in fixed dimensions by Lenstra [39], Grötschel, Lovász, and Schrijver [38], and Lovász and Scarf [46]. Finding thin directions is done by lattice basis reduction based on the work of Lenstra, Lenstra, and Lovász [42]. This approach proved very efficient for some instances where branch and bound fails due to huge enumeration trees. Aardal et al. [1] applied a related algorithm, developed by Aardal, Hurkens, and Lenstra [2], to market split instances of the type proposed by Cornuéjols and Dawande [26]. They managed to solve instances much larger than those that could be solved by regular branch and bound. For a recent paper in this direction, see Mehrotra and Li [50].

Other examples of branching on general disjunctions are SOS branching and local branching. Given the presence of a Special Ordered Set (SOS) [15] constraint in the formulation (also called Generalized upper bound), branching can be done by replacing the original SOS constraint by a new SOS constraint, different in both children. This results in a significant reduction of the number of nodes that need to be enumerated. In their paper on local branching [32], Fischetti and Lodi propose a way to direct the search in the branch-and-bound algorithm. They branch on a special type of constraint that defines a neighborhood of the incumbent solution.

The methods cited above find a set of promising branching disjunctions before the start of branching or apply very specific types of general disjunctions. Our approach is to select general disjunctions at every node of the search tree based on a heuristic measure of their quality. Similar ideas have not been studied extensively in the literature. To our

knowledge, there is one related study. Owen and Mehrotra [54] propose branching on general disjunctions generated by a neighborhood search heuristic. The neighborhood contains all disjunctions with coefficients in $\{-1, 0, 1\}$ on the integer variables with fractional values at the current node. The quality of the disjunctions is evaluated by solving the children nodes in the spirit of strong branching.

Owen and Mehrotra tested their approach on 12 instances from MIPLIB 3.0 [20] and report a significant decrease in the total number of nodes in a majority of them compared to strong branching as implemented in CPLEX. The proposed procedure is not computationally efficient because of the large number of subproblems solved before each branching. Nevertheless, it emphasizes the important observation that branching on general disjunctions can decrease the size of the branching tree significantly.

The main differences between our approach and that of Owen and Mehrotra are:

- we consider a different class of general disjunctions, the ones defining mixed integer Gomory cuts, while Owen and Mehrotra propose $\{-1, 0, 1\}$-disjunctions.
- instead of an extensive heuristic search, we apply a two-phase disjunction selection procedure based on the depth of the corresponding intersection cuts in the first phase and on strong branching in the second. As a result,
- we propose a computationally efficient algorithm that competes with branching on single variables not only in terms of tree size but in terms of solution time.

2.3 Split disjunctions and intersection cuts

We present a summary of the theoretical foundations of intersection cuts. For a detailed discussion, refer to Balas [11] and Andersen, Cornuejols, and Li [5].

Consider the Mixed Integer Linear Program:

$$\text{(MILP)} \qquad \min\{c^T x : Ax = b,\ x \geq 0_n,\ x_j \text{ integer for } j \in N_I\}, \tag{2.1}$$

where $c, x, 0_n \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and $N_I \subseteq N := \{1, 2, \ldots, n\}$. Without loss of generality, assume A is of full row rank. The Linear Programming relaxation, denoted by (LP), is obtained from (MILP) by dropping the integrality constraint on x_j for $j \in N_I$. Let P_I and P denote the sets of feasible solutions to (MILP) and (LP), respectively. A

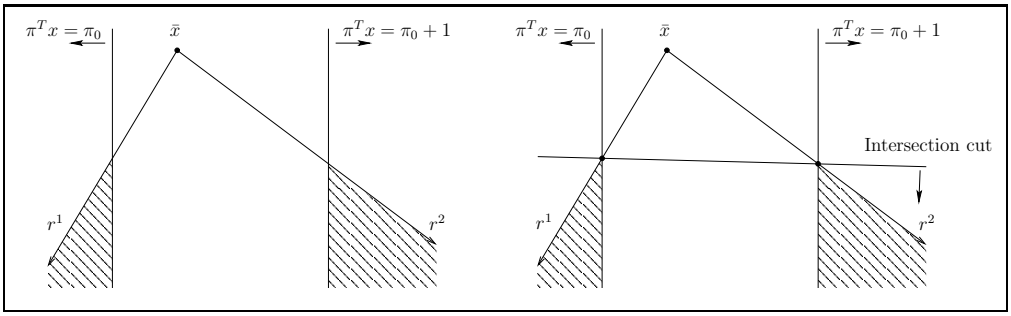

Figure 2.1: Deriving the intersection cut

basis for (LP) is an m-subset B of N such that the column submatrix of A induced by B is an invertible submatrix of A. Let $J := N \setminus B$ denote the index set of non-basic variables. A further relaxation of the set P with respect to a basis B is obtained by removing the non-negativity constraints on the basic variables. We denote it by $P(B)$:

$$P(B) := \{x \in \mathbb{R}^n : Ax = b \text{ and } x_j \geq 0 \text{ for } j \in J\}. \tag{2.2}$$

This set is a translate of a polyhedral cone: $P(B) = C + \bar{x}$, where $C = \{x \in \mathbb{R}^n : Ax = 0 \text{ and } x_j \geq 0 \text{ for } j \in J\}$ and $\bar{x}$ solves $\{x \in \mathbb{R}^n : Ax = b \text{ and } x_j = 0 \text{ for } j \in J\}$, i.e. $\bar{x}$ is the *basic solution* corresponding to the basis B. The cone C can be expressed also in terms of its extreme rays, r^j for $j \in J$: $P(B) = Cone(\{r^j\}_{j \in J}) + \bar{x}$, where $Cone(\{r^j\})$ denotes the polyhedral cone generated by vectors $\{r^j\}$. The extreme rays of $P(B)$ can be found from the simplex tableau corresponding to the basis B.

Define a *split disjunction* $D(\pi, \pi_0)$ to be a disjunction of the form $\pi^T x \leq \pi_0 \ \vee \ \pi^T x \geq \pi_0 + 1$, where $(\pi, \pi_0) \in \mathbb{Z}^{n+1}$ and $\pi_j = 0$ for $i \notin N_I$. Clearly, any feasible solution to (MILP) has to satisfy every split disjunction. Any violated split disjunction can be used to define a cutting plane that cuts off points of P violating the disjunction. The generation of this *intersection cut*, as defined by Balas [11], is exemplified in Figure 2.1 and explained below.

Given a split disjunction $D(\pi, \pi_0)$, let $F_{D(\pi,\pi_0)} := \{x \in \mathbb{R}^n : \pi^T x \leq \pi_0 \ \vee \ \pi^T x \geq \pi_0 + 1\}$ denote the set of points that satisfy the disjunction. Since $P_I \subseteq P(B) \cap F_{D(\pi,\pi_0)}$, a valid cut for $P(B) \cap F_{D(\pi,\pi_0)}$ is valid for P_I. In particular, the intersection cut is a half-space bounded by the hyperplane passing through the intersection points of $D(\pi, \pi_0)$ with the extreme rays of $P(B)$.

In order to find the intersection points, for all $j \in J$ we compute the scalars:

$$\alpha_j(\pi, \pi_0) := \begin{cases} -\frac{\varepsilon(\pi,\pi_0)}{\pi^T r^j} & \text{if } \pi^T r^j < 0, \\ \frac{1-\varepsilon(\pi,\pi_0)}{\pi^T r^j} & \text{if } \pi^T r^j > 0, \\ +\infty & \text{otherwise,} \end{cases} \tag{2.3}$$

where $\varepsilon(\pi, \pi_0) := \pi^T \bar{x} - \pi_0$ is the amount by which $\bar{x}$ violates the first term of the disjunction $D(\pi, \pi_0)$. The number $\alpha_j(\pi, \pi_0)$ for $j \in J$ is the smallest number α such that $\bar{x} + \alpha r^j$ satisfies the disjunction. In other words, $\bar{x} + \alpha_j(\pi, \pi_0) r^j$ lies on one of the disjunctive hyperplanes $\pi^T x = \pi_0$ and $\pi^T x = \pi_0 + 1$.

Now, the intersection cut associated with B and $D(\pi, \pi_0)$ is given by:

$$\sum_{j \in J} \frac{x_j}{\alpha_j(\pi, \pi_0)} \geq 1. \tag{2.4}$$

The Euclidean distance between $\bar{x}$ and this hyperplane is:

$$d(B, \pi, \pi_0) := \sqrt{\frac{1}{\sum_{j \in J} \frac{1}{(\alpha_j(\pi, \pi_0))^2}}} \tag{2.5}$$

This quantity, called *distance cut off* or *depth*, can be used as a measure of the quality of the cut. To our knowledge, it was first used by Balas, Ceria, and Cornuéjols [13].

An important result of Balas [11] is that Gomory cuts derived from the simplex tableau associated with B can be viewed as intersection cuts. Let $\bar{a}_{ij}$ be the entry of the simplex tableau in row i and column j. The mixed integer Gomory cut derived from the row in which x_i is basic can be obtained as an intersection cut from the disjunction $D(\hat{\pi}^i, \hat{\pi}^i_0)$:

$$\hat{\pi}^i_j := \begin{cases} \lfloor \bar{a}_{ij} \rfloor & \text{if } j \in N_I \cap J \text{ and } \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor \leq \bar{x}_i - \lfloor \bar{x}_i \rfloor, \\ \lceil \bar{a}_{ij} \rceil & \text{if } j \in N_I \cap J \text{ and } \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor > \bar{x}_i - \lfloor \bar{x}_i \rfloor, \\ 1 & \text{if } j = i, \\ 0 & \text{otherwise,} \end{cases} \tag{2.6}$$

$$\hat{\pi}^i_0 = \lfloor (\hat{\pi}^i)^T \bar{x} \rfloor$$

This disjunction can also be obtained by strengthening the simple disjunction $D(\pi^i = e_i,\ \pi^i_0 = \lfloor \bar{x}_i \rfloor)$ on the non-basic integer variables where the affected coefficients are modified so that the distance cut off is maximized.

2.4 Branching on general disjunctions

This work is inspired by the relation between branching disjunctions and intersection cuts at the optimal basic solution of the current LP relaxation. A violated split disjunction can be used for generating an intersection cut but it can be used for branching as well. A good intersection cut cuts deeply into the polyhedron of feasible solutions of the LP relaxation and improves the lower, Linear Programming bound. Our suggestion is that a split disjunction defining a deep cut is good for branching too. The LP lower bound is often an important determinant of the amount of enumeration needed to complete the solution. (Because of this, improving the lower bound is the aim of common rules for selecting branching variables implemented in current MILP solvers.) The improvement in the lower bound caused by branching on a split disjunction is no less than the improvement by the corresponding intersection cut. We show this below.

A routine for branching on general disjunctions requires a procedure for selecting the disjunction to branch on, which, in turn, requires a criterion for comparing the quality of disjunctions, i.e. a criterion for comparing some measure of improvement in the lower bound.

Measure of quality of a disjunction

A common rule for choosing a branching variable (simple disjunction) is to maximize some function of the two optimal objective values computed at the children nodes that would result from branching on this variable. Specifically, let $\bar{x}_1$ and $\bar{x}_2$ be the optimal solutions for the first and second child, respectively, and let $z(\bar{x}_1) = c^T\bar{x}_1$ and $z(\bar{x}_2) = c^T\bar{x}_2$ be the corresponding objective values. Two typical functions are $\min(z(\bar{x}_1), z(\bar{x}_2))$ or $\frac{1}{2}[z(\bar{x}_1) + z(\bar{x}_2)]$. Experiments with other options have been reported in the literature. As an example, Linderoth and Savelsbergh [43] show good results with the sum $\frac{1}{3}[\min(z(\bar{x}_1), z(\bar{x}_2)) + z(\bar{x}_1) + z(\bar{x}_2)]$, while Achterberg, Koch, and Martin [3] propose $\frac{1}{6}[4\min(z(\bar{x}_1), z(\bar{x}_2)) + z(\bar{x}_1) + z(\bar{x}_2)]$. The only exact way to compute these functions is to solve the linear programs at the children nodes. A commonly used method, called *strong branching*, does this for the candidate branching variables before choosing the "best" one. This is computationally expensive when applied to all integer variables with fractional values at the current basis and is impossible to apply when branching on general disjunctions because of the infinite number

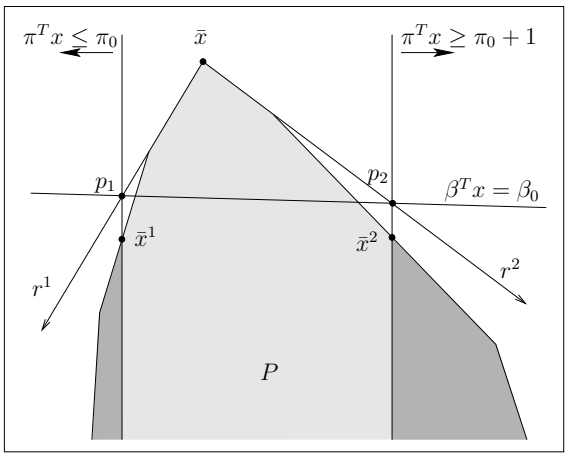

Figure 2.2: The LP bound obtained by branching is different from the one obtained by cutting.

of such disjunctions. A procedure for pre-selecting a small finite set of disjunctions is needed before strong branching can be applied. Such a procedure requires a heuristic measure of disjunction quality.

We consider the integrality gap closed, or equivalently $\min(z(\bar{x}_1), z(\bar{x}_2))$, an "exact" measure of the quality of a disjunction. Based on the relation between an intersection cut and the underlying disjunction, we propose to use the depth (distance cut off) of the cut as a proxy to this measure, since the distance cut off is correlated to the amount of integrality gap closed by adding the cut. Next, we show that the gap closed by branching on a split disjunction is always at least as large as the gap closed by the corresponding intersection cut.

Let $P(B)$ be defined as in (2.2) and consider a split disjunction $D(\pi, \pi_0)$. Let $\beta^T x \leq \beta_0$ be the intersection cut defined by $P(B)$ and $D(\pi, \pi_0)$. (An example is shown in Figure 2.2.) The feasible sets of the children are $F_1 := P \cap \{x \in \mathbb{R}^n : \pi^T x \leq \pi_0\}$ and $F_2 := P \cap \{x \in \mathbb{R}^n : \pi^T x \geq \pi_0 + 1\}$. Let $\bar{x}_1 := \arg\min\{c^T x : x \in F_1\}$ and $\bar{x}_2 := \arg\min\{c^T x : x \in F_2\}$ be the corresponding optimal basic solutions. Let $p_1 := \arg\min\{c^T x : x \in P(B) \text{ and } \pi^T x \leq \pi_0\}$ and $p_2 := \arg\min\{c^T x : x \in P(B) \text{ and } \pi^T x \geq \pi_0 + 1\}$. Then, $z(p_i)$ is a lower bound for $z(\bar{x}_i)$, for $i = 1, 2$, because $P(B) \supseteq P$. Therefore, the optimal solution of $\min\{c^T x : x \in P(B) \text{ and } \beta^T x \leq \beta_0\}$, provides a lower bound for any measure of disjunction quality

which is a convex combination of $z(\bar{x}_1)$ and $z(\bar{x}_2)$. Consequently, branching on a general disjunction can provide a better lower bound than the corresponding intersection cut. In this respect, we cannot substitute branching by adding the corresponding intersection cut.

In our procedure, we will use the distance cut off by the corresponding intersection cut as a heuristic measure of the quality of a disjunction. Other measures can be used as well. Future research in this direction should be fruitful.

Procedure for selecting the branching disjunction

We need a procedure for selecting promising split disjunctions for branching. As we discussed in the introduction, optimizing over the set of all split disjunctions is prohibitively expensive. One idea would be to come up with a heuristic search routine. Instead, we suggest to concentrate on a finite class of general disjunctions which we can enumerate — the set of split disjunctions defining mixed integer Gomory cuts, which we call *MIG disjunctions*. The reasons for our choice are the following. First, this set is not only finite but relatively small. Its cardinality at a given node of the branch-and-bound tree equals the number of integer variables with fractional values in the current basic solution. Second, these disjunctions are fast to obtain. They can be generated from the current tableau by a closed form formula (2.6). Third, as we mention in Section 2.3, these disjunctions can be viewed as strengthened simple disjunctions (with respect to the cut depth) which suggests that they could perform better.

The branching procedure we propose is shown in Figure 2.3. We consider the set $\mathcal{M}$ of all MIG disjunctions for a specific basic solution and select a subset $\mathcal{S}$ of it, containing the most promising disjunctions according to the chosen criterion for comparison. (Here, the distance cut off by the underlying intersection cut.) We limit the cardinality of $\mathcal{S}$ to k. In our tests, we use a constant k throughout the branching tree ($k = 10$). The parameter k can be used to manage the computational effort at different levels, e.g. a larger k can be used close to the root where branching decisions are more important and a smaller k in the deep levels. Finally, we apply strong branching to the disjunctions in $\mathcal{S}$.

The computational complexity of this procedure at each node is dominated by Step 3 and it is roughly equivalent to applying strong branching to the k most fractional variables.

At each node:

1. Generate the set $\mathcal{M}$ of all MIG disjunctions.
2. Select a subset $\mathcal{S} \subseteq \mathcal{M}$ of disjunctions with largest distance cut off, such that $|\mathcal{S}| \leq k$.
3. Apply strong branching to the disjunctions in $\mathcal{S}$ maximizing gap closed and choose a disjunction $D \in \mathcal{S}$. Branch on D.

Figure 2.3: Procedure for branching on MIG disjunctions

2.5 Experimental results

The test set for the experiments we report is the union of the Mixed Integer Linear Programming libraries MIPLIB 2.0 [53], MIPLIB 3.0 [20], and MIPLIB 2003 [4]. We exclude very easy instances that can be solved in less than 50 nodes by the algorithms we tested. We also exclude some very difficult instances — those for which less than 50 nodes can be processed in one hour. These two groups of instances are considered in the first experiment, where the gap closed at the root is compared but they are excluded from the subsequent experiments. We also exclude instances with zero integrality gap. The final number of instances in the test set is 80. This is a heterogeneous set of benchmark instances of different sizes and with different origins and applications. It serves as a good test-bed of our ideas.

All experiments are conducted on an IBM IntellistationZ Pro computer with an Intel Xeon 3.2GHz CPU and 2GB RAM. The MILP solver used is COIN-OR BCP, where some user methods are modified for the purpose of our experiment. The LP solver is COIN-OR CLP. Mixed integer Gomory cuts were generated using the cut generator in the library COIN-OR CGL.

In our experiments, we compare branching on single variables to branching on general disjunctions. When an instance is solved to optimality, we compare the solution time and the size of the branch-and-bound trees. When the solution of an instance is interrupted (due to time limit or bound on the depth of exploration), we compare the amount of integrality gap closed. We consider the *absolute gap closed*: the difference between the lower bound at interruption and the lower bound at the root node, and the *relative* (*percentage*) *gap closed*:

the absolute gap closed relative to the gap at the root.

In the first experiment, we study the gap closed after branching at the root node. In the second experiment, we study the gap closed and the number of active nodes left after branching for five levels. We apply pure branch and bound in these experiments in order to avoid the influence of adding different cutting planes. This ensures a clean comparison between the two branching procedures. Finally, we test our idea in a cut-and-branch framework and propose a combination of the two branching methods. We also study the increase in solution time per node caused by branching on general disjunctions.

2.5.1 Gap closed at the root

We first compare the gap closed at the root by ordinary branching on single variables and branching on MIG disjunctions. When branching on variables, we select the (up to) ten most fractional variables — those integer variables that have fractional parts closest to 0.5. Then, the variable to branch on is chosen by strong branching with objective to maximize the minimum gap closed in the children, i.e. the variable with maximum $\min(z(\bar{x}^1), z(\bar{x}^2))$ is selected. In this implementation, if branching on a variable creates an infeasible child, this variable is preferred to all others. For short, we call this setup for branching on simple disjunctions *SIMDI*.

When branching on general disjunctions, we select the (up to) ten MIG disjunctions with best distance cut off. Again, the object to branch on is chosen by strong branching following the same rules as before. This guarantees fair comparison for variables and general disjunctions since the number of branching candidates in both cases is the same. We call this setup for branching on general disjunctions *GENDI*.

The comparison of the absolute gap closed shows that GENDI performs better for 59 out of 108 instances, while SIMDI is better for 20 instances. For 90 instances the optimal objective value is known and this allows to compute the percentage gap closed. The average percentage gap closed by GENDI and SIMDI over this set of instances is 17.9% and 11.7%, respectively. If we consider only those 65 instances for which the gap closed by both methods differ, the average gap closed by GENDI and SIMDI is 16.8% and 8.1%, resp. Detailed results are given in Table A.1.

A closer look at the cases when one method substantially dominates the other shows

that for 11 instances GENDI closes a positive amount of gap while SIMDI cannot close any gap. In contrast, there are only 3 instances for which GENDI is unsuccessful while SIMDI manages to improve the lower bound. When we look at the instances for which both methods managed to improve the lower bound, the gap closed by GENDI is an order of magnitude larger than that of SIMDI for 12 instances, while SIMDI is an order of magnitude better for 2 instances. Some examples where GENDI is clearly more successful are: `10teams`, `blend2`, `fiber`, `gt2`, `mod010`, `momentum1`, `2`, and `3`, `msc98-ip`, `p0282`, `nw04`, and `swath`. Similarly, SIMDI dominates in: `arki001`, `bell3b`, `mzzv42z`, and `roll3000`.

These results are a strong indication that GENDI performs better than SIMDI. Next, we test whether these good results at the root proliferate throughout the tree by branching for five levels and by branching for up to two hours. Branching for five levels helps observe another good effect of branching on MIG disjunctions: a decrease in the number of active nodes.

2.5.2 Branching for five levels

In the second experiment, we branch at the top five levels of the branch and bound tree and compare the resulting gap closed. As before, GENDI performs better. This is mainly due to the larger gap closed by branching on general disjunctions, which we observed at the root as well. But now we observe an interesting secondary effect: branching on general disjunctions tends to produce more infeasible children, which additionally decreases the amount of enumeration. We record this phenomenon by counting the number of active nodes at the fifth level.

Detailed results of the experiment are shown in Table A.2. Summary results are shown in Table 2.1. Lines labeled "Average" contain the average value of the criterion. Lines labeled "Count better" contain the number of instances for which one method dominates the other according to the criterion.

In terms of gap closed, SIMDI dominates in 19 cases, GENDI in 48 cases out of 80. The average gap closed by SIMDI and GENDI is 24.3% and 29.1%, resp, over the set of 75 instances for which the optimal objective value is known. These results support our earlier observation that GENDI closes more gap.

It is interesting to observe that GENDI also produces a smaller number of active nodes

Table 2.1: Comparison of SIMDI and GENDI after five levels of branching

	SIMDI	GENDI
Absolute gap closed		
Count better	19	48
Percentage gap closed		
(Instances with known optimal solution)		
Average	24.3	29.1
Count better	18	46
Active nodes at level 5		
Average	25.8	18.0
Count better	10	43
Gap closed and active nodes together		
Count better	6	33

at the fifth level, on average. Out of the maximum possible 32 nodes, SIMDI generates 25.8 while GENDI generates 18. On this criterion, SIMDI performs better in 10 cases while GENDI does this in 43 cases. This effect is important not by itself but in combination with improvement in the gap. Combining both criteria, SIMDI results in a larger gap closed and a smaller number of active nodes in only six cases, while GENDI achieves this in 33 cases.

The reason for the smaller number of active nodes is that GENDI often generates disjunctions that produce only one feasible child. For some instances, this happens at every level of the branching tree, resulting in a single node at level five. Sometimes, this is combined with an impressive improvement of the gap closed over SIMDI, e.g. `manna81`, `p0033`, `set1al`, `set1cl`, `seymour`, and `swath`. See also `lseu`, `roll3000`, `set1ch`, and `sp97ar`.

The combination of a larger improvement in the gap and a smaller number of active nodes is a very desirable effect and it deserves more attention. Branching on a disjunction that generates only one feasible child is equivalent to adding a single cut to the formulation. One may argue that this cut would be added by a branch-and-cut algorithm and only the fact that we apply pure branch and bound leaves us such good opportunities. This

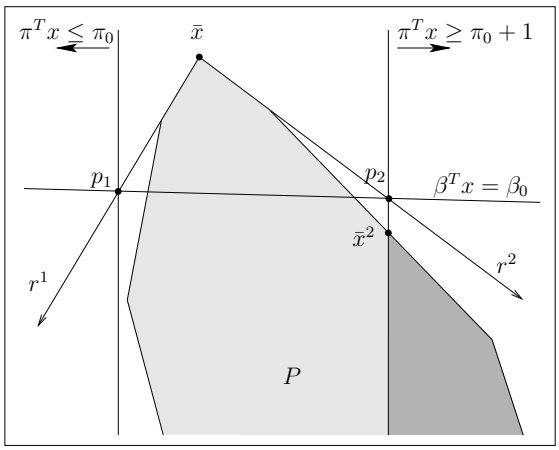

Figure 2.4: Disjunction with only one feasible child

could be true in some cases but in others the disjunction inequality is stronger than the corresponding MIG cut. Figure 2.4 is an example. The cut generation procedure considers the polyhedral cone pointed at $\bar{x}$, relaxing some of the constraints defining P, and generates the intersection cut $\beta^T x \leq \beta_0$. But it cannot detect the fact that one of the feasible sets of the children is empty. (Here, $P \cap \{x \in \mathbb{R}^n : \pi^T x \leq \pi_0\}$.) When branching on $D(\pi, \pi_0)$, we essentially add the cut $\pi^T x \geq \pi_0 + 1$, which is stronger than $\beta^T x \leq \beta_0$.

Consequently, branching on a general disjunction that generates only one child can be viewed as strengthening the underlying intersection cut. Thus, branching on a general disjunction cannot be substituted by adding the corresponding intersection cut even when one of the disjunctive sets is empty. When both disjunctive sets are non-empty, branching on a general disjunction can close more gap than the corresponding cut, as we showed in Section 2.4.

We do not consider branching on general disjunctions a substitute of cutting planes. Our procedure comes into play when branch and cut decides to start branching. And then, a routine that tends to avoid branching (and duplicating the problem) when some part of the gap can be closed by a cut could be very useful.

2.5.3 Cut and branch

Now, we test the performance of our approach in a cut-and-branch framework. We generate ten rounds of mixed integer Gomory cuts before proceeding to branching. Branching is done as before: at each node, up to ten branching objects are selected for strong branching. We make two minor modifications in the algorithm for branching on general disjunctions. The first aims at avoiding unnecessary increase in the size of the subproblems and the second aims at avoiding numerical problems. First, when the π vector of a disjunction is a singleton, we branch on that variable instead of adding explicit constraints of the type $x_j \leq \lfloor \bar{x}_j \rfloor$ or $x_j \geq \lfloor \bar{x}_j \rfloor + 1$. Second, we do not consider dense disjunctions for branching. We define dense disjunctions to be those whose vector $\pi \in \mathbb{Z}^n$ has support of cardinality greater than $\max(10, 0.1n)$. If all generated disjunctions at a node are dense, we branch on variables instead.

The limit on the solution time is two hours. Our goal is to solve the instances and compare the tree size and the running time. For those not solved to optimality, the amount of gap closed is compared. We also compute the increase in computing time caused by branching on general disjunctions by computing the ratio of the average solution time per node required by the two different branching schemes.

Branching on general disjunctions performs better than branching on variables for a majority of the instances but not for all of them. This suggests that a combined approach can build upon the advantages of both "pure" algorithms. We combine them in the following way: at every node of the branching tree, we select five most fractional variables and five disjunctions with maximum distance cut off. Then we apply strong branching to all these branching objects. We call this algorithm *COMBI* and compare it to the other two. Detailed results from this experiment are presented in Table A.3.

Out of 80 test instances, SIMDI solved 35 in the alloted time, GENDI solved 50, and COMBI solved 50. GENDI solved all instances solved by SIMDI except one, `stein45`. COMBI solved all instances solved by SIMDI and could not solve only one instance solved by GENDI, `bell4`.

We ranked the methods according to the amount of gap closed for each instance (see Table 2.2). SIMDI ranked first 46 times and GENDI 67 times. GENDI closed more gap than SIMDI in 33 cases while SIMDI dominated in 8 cases. In addition, we compute the ratios of

Table 2.2: Ranks according to gap closed.

Rank	SIMDI	GENDI	COMBI
1	46	67	59
2	2	7	18
3	32	6	3

Table 2.3: Geometric mean of the ratios of three parameters over all 80 instances.

Ratio	GENDI / SIMDI	COMBI / SIMDI	COMBI / GENDI
Absolute gap closed	1.15	1.14	0.99
Absolute gap closed by branching	1.26	1.23	0.97
Average solution time per node	1.24	1.29	1.04

the absolute gap closed by GENDI and SIMDI for all instances and look at the geometric mean. (Values of infinity, corresponding to zero gap closed by SIMDI, are excluded.) Since part of this gap is closed by the cuts added at the root, a better measure for comparing the different branching methods is the ratio between the gap closed only by branching. These ratios are shown in the second column of Table 2.3. GENDI closes 15% more gap than SIMDI on average, and if we consider only the gap closed by branching, the improvement is 26%. These results indicate that GENDI performs better than SIMDI on this test set.

The performance of the combined approach, COMBI, tends to be close to the better of the other two methods. It equaled the best gap closed by SIMDI and GENDI for 53 (out of 80) instances and even improved it in six cases. It ranked last only three times demonstrating more consistent behavior. Comparing the ratios of the gap closed (Table 2.3), we conclude that COMBI performs a few percent worse than GENDI on average but still outperforms SIMDI significantly.

Table 2.3 shows the ratio of the average solution time per node of the three branching procedures as well. We observe that GENDI requires 24% more time to solve a node than

Table 2.4: Geometric mean of the ratios of solution time and tree size over 35 instances solved by all methods

Ratio	GENDI / SIMDI	COMBI / SIMDI	COMBI / GENDI
Solution time	0.72	0.50	0.70
Tree size	0.65	0.42	0.65

SIMDI (including the time for strong branching). The corresponding statistic for COMBI is 29%. Careful analysis shows that the increased solution time per node caused by branching on general disjunctions is offset by the reduced amount of enumeration and the increased amount of integrality gap closed.

To make a more detailed analysis, we split the test set into three sets: instances solved by all methods, those not solved by any method, and the remaining. This analysis is presented below. In summary, the two algorithms using branching on general disjunctions solved 43% more instances within the two hour time limit compared to branching on variables. For those instances solved by all algorithms, GENDI and COMBI are more efficient in terms of solution time and tree size, on average. For the instances not solved by any method, GENDI and COMBI closed more gap for the same time. These observations are a strong indication that the proposed procedures for branching on general disjunctions perform better than branching on variables.

Instances solved by all methods

Thirty five instances were solved by the three algorithms which allows to compare the size of the branching trees and the total solution times. One instance, `manna81`, was solved solely by cutting planes and is excluded from this analysis.

Table 2.4 contains the geometric mean of the ratios of solution time and tree size. It shows that GENDI decreases the solution time by 28% compared to SIMDI. COMBI is more efficient and solves these instances twice as fast as SIMDI. The decrease in the tree size is even more significant. GENDI and COMBI explore 65% and 42% of the nodes explored

by SIMDI, respectively. At the two extremes, GENDI and COMBI are at least five times more efficient in solving `dcmulti`, `flugpl`, `gesa3`, `gesa3_o`, `lseu`, and `p0033`, while SIMDI is about five times faster with `air05` and `qnet1_o`. COMBI is faster than the other two algorithms in most cases. As an extreme example, it solves `nw04` in 48 seconds while SIMDI and GENDI require 60 and 113 minutes, respectively.

In conclusion, we observe that branching on general disjunctions causes significant reduction in the total solution time compared to branching on variables. The combined approach performs notably better than the other two algorithms on this test set.

Instances solved by some but not all methods

Fifteen instances were solved by some but not all algorithms. These are relatively difficult instances on which branching on general disjunctions performs significantly better and in many cases orders of magnitude better. Thirteen of them were solved by GENDI and COMBI but not solved by SIMDI within two hours. The geometric mean of the solution times is 228 seconds by GENDI and 301 seconds by COMBI. Some cases of extreme reduction of the solution time are: `bell5` solved in 9.9 s and 2055 processed nodes by GENDI; `fiber` solved in 18 s and 203 nodes by GENDI, and in 11.3 s and 172 nodes by COMBI; `gt2` solved in 1.1 s and 120 nodes by GENDI, and in 0.5 s and 41 nodes by COMBI. All these instances could not be solved in two hours and hundreds of thousands of nodes when branching on variables. Another example is `10teams` where SIMDI closed only 28.6% of the gap in two hours, while GENDI solved the problem in 75 minutes and COMBI solved it in four minutes. A significant difference in the gap closed is observed for `rout`, `p0282`, and `fiber`, as well.

One instance, `bell4`, was solved only by GENDI. It was solved in 51.2 s and 3490 nodes while the other methods could not solve it in two hours and more than 250,000 nodes. SIMDI managed to close 96.6% of the gap and COMBI closed 99.6%. The last instance from this group, `stein45`, was solved by SIMDI in 103 minutes and by COMBI in 80 minutes but not solved by GENDI.

These observations clearly reveal the properties of branching on general disjunctions to close larger amounts of the integrality gap and to decrease the amount of enumeration.

Table 2.5: Geometric mean of the ratios of gap closed over 23 instances not solved by any method

Ratio	GENDI / SIMDI	COMBI / SIMDI	COMBI / GENDI
Absolute gap closed	1.30	1.25	0.97
Absolute gap closed by branching	1.59	1.43	0.90

Instances not solved by any method

Twenty eight instances were not solved by any method within the time limit. We compare the gap closed by the three algorithms in the two-hour interval of time. None of the algorithms closed any gap by branching for three instances: `liu`, `markshare1`, and `2`. For two instances, `opt1217` and `p2756`, SIMDI closed no gap while GENDI and COMBI closed positive amounts of gap. These five instances are excluded from the summary statistics presented in Table 2.5. For the remaining 23 instances, GENDI and COMBI closed 30% and 25% more gap than SIMDI, respectively. If we consider only the gap closed in the branching phase of cut and branch, the improvement in the gap is 59% and 43%, respectively. These results show that branching on general disjunctions closes significantly more gap than branching on variables for the same time interval, on average.

2.6 Conclusion

In this paper, we propose a procedure for branching on general disjunctions as part of a branch-and-cut algorithm for solving Mixed Integer Linear Programming problems. The procedure is independent of the instance characteristics and can be applied to any MILP.

We discuss the relation between branching disjunctions and intersection cuts and show that branching on general disjunctions can close more gap than adding the corresponding intersection cut, implying that branching cannot be substituted by cutting planes. We propose to use the distance cut off by the intersection cut as a measure of quality of the disjunction. This measure is used for pre-selection of the most promising disjunctions before

strong branching.

We test these ideas in experiments with 80 test instances from the literature, comparing branching on variables with branching on general disjunctions. We observe that on average the effect of branching on general disjunctions is: (i) smaller tree size and solution time for the instances solved to optimality, and (ii) larger amount of gap closed for the other instances. The test results clearly indicate that the proposed procedure outperforms regular branching on variables for most instances.

We also observe that, in addition to the larger amount of gap closed, branching on general disjunctions results in only one feasible child more often than branching on variables does (with the same branching rules applied). This is an interesting side effect that decreases the tree size further. In our implementation, we select ten disjunctions for strong branching and sometimes more than one produces an infeasible child. One could add all these disjunctive inequalities, which are valid for P_I, as cuts instead of adding just one through branching. This will decrease the search space at the child node. We have not implemented this idea.

We obtain an efficient algorithm by considering only a specific class of disjunctions — those defining mixed integer Gomory cuts — instead of searching the whole set of split disjunctions. This approach can be extended to disjunctions defining other classes of split cuts, such as lift-and-project, reduce-and-split, and mixed integer rounding cuts. More work can be done on criteria for evaluating disjunctions as well.

Chapter 3

The effect of angle on the quality of a family of cutting planes

3.1 Introduction

In a branch-and-cut algorithm, cutting planes are added with the aim of removing parts of the relaxed set that contain no feasible solution. The idea of generating cuts within branch and bound is one of the cornerstones of the recent major improvements in the Mixed Integer Linear Programming (MILP) optimization software. It lead to a considerable decrease of the solution time for hard instances.

The development of various combinatorial and general cutting planes, together with the increased computational power, allow us to generate a large number of cuts. Adding all of them to the formulation can be detrimental to the speed of the solution and to the numerical stability. There are indications that the leading commercial software packages for MILP apply cut selection. Unfortunately, their procedures are not revealed. In addition, there has not been much research devoted to this topic.

Adding cuts affects the performance of the algorithms in various ways. Cuts strengthen the formulation of the problem. This leads to an improvement of the bound used for pruning and, therefore, to a decrease in the amount of enumeration. On the other hand, increasing the size of the formulation slows down the solution of the LP relaxation. This effect propagates to the children of the node where cuts are added, and to their children. If the added cuts are dense (with a large fraction of non-zero coefficients), the slowdown

in reoptimization is even larger. In addition, cuts may affect the numerical stability of the coefficient matrix.

In this study, we concentrate on some geometric properties of cutting planes in an attempt to evaluate their quality. The novelty in our approach is that we look at the quality of a family of cuts as a group, and not only at the properties of the individual cuts. Cuts are added in rounds and the effect of a round of cuts is not the sum of the effects that individual cuts cause. Furthermore, some of the negative effects, like numerical instability, are often caused by the interaction of cuts rather than their individual properties.

Most of the widely used classes of general cutting planes are intersection cuts. (E.g., mixed integer Gomory cuts [35], mixed integer rounding cuts [47], lift-and-project cuts [12], and reduce-and-split cuts [5].) Adding many of these cuts simultaneously causes a significant decrease in the angles between the active constraints — an effect we call *flattening*. As a result, future rounds of cuts are not as efficient. Flattening may affect the branching phase too.

We study the effects of adding cuts on the performance of branch and cut, and test the efficiency of several cut selection algorithms. We propose a cuts selection procedure that incorporates distance cut off and angles between cuts as measures of quality of a set of cuts. We also propose a termination criterion for cut generation.

3.2 Motivation and algorithms

3.2.1 Reasons for cut selection

Cut selection aims at alleviating the negative effects of adding a large number of cuts while preserving the cutting power of the set of selected cuts. In order to distinguish "good" from "bad" cuts, various cut properties are considered. Usually, cuts are evaluated on an individual basis. In this study, we emphasize the importance of evaluating cuts as a group. Cuts are generated and added in rounds rather than one by one. Therefore, the collective influence of a set of cuts on the algorithm is what we need to evaluate.

The importance of cut selection is well understood. There is evidence that contemporary state-of-the-art commercial MILP solvers implement some kind of cuts selection procedures. Nevertheless, to our knowledge, there is no published systematic research on cut selection

algorithms.

The main drawbacks of adding a large number of cutting planes are potential numerical problems and increased total solution time caused by a slowdown in the subproblems reoptimization. The typical reasons for numerical instability are (i) the large difference in the magnitude of cut coefficients, and (ii) the presence of two almost parallel cuts in the pool of added cuts. The first issue is addressed by discarding the cut. To deal with the second issue, a new cut is added to the pool of selected cuts only if its angle with those already selected is above some threshold. In the literature, a common rule is "the cosine of the angles should be no greater than 0.999." [13, 31]

The efforts to minimize the slowdown caused by adding many cuts usually materialize in discarding part of the generated cuts. Cut density is known to have a strong negative impact on the speed of reoptimization. As a consequence, dense cuts are often discarded. Further cut selection is done by applying some measure of cut quality. A commonly used measure is the *distance cut off*. It equals the Euclidean distance between the current basic solution and the cut hyperplane. It is also called *depth* or *steepness* by some authors. We will use the terms distance cut off and depth interchangeably.

Better distance cut off does not guarantee a better new LP bound but, clearly, there is a relation between the two. One reason why distance cut off is not a perfect measure of the improvement in the lower bound is that the depth does not incorporate the direction of the objective. Another criticism is that the distance cut off is computed in the full-dimensional space while the polyhedron may not be full dimensional. A solution would be to compute the depth in the affine subspace spanned by the polyhedron. Unfortunately, this subspace is often unknown.

Cook, et al. [24] studied these two variations of distance cut off and other distance-related measures empirically, applied to a set of traveling salesman problem instances. They concluded that all measures based on distance perform similarly. We adopt distance cut off (in the full-dimensional space) as a quality measure of an individual cut.

An important property characterizing a group of cuts is the size of angles between them. As we noted, angle has been used in the literature but mainly for discarding parallel or close-to-parallel cuts. The threshold for a minimum acceptable angle is typically set to a very small angle. (The only exception, to our knowledge, is the work of Anreello, et al. [6] on $\{0, \frac{1}{2}\}$-cuts, where a threshold of $\cos = 0.1$ is tested with this particular class of

combinatorial cuts.) We suggest applying a more restrictive angle threshold to selecting general cutting planes. The goal is to control the effect of cuts on the polyhedral structure close to the optimal basis. We justify the importance of angles between cuts in the next section.

3.2.2 The importance of angle between cuts

We argue that adding many cuts to the formulation is not good for the facial structure of the polyhedron close to the optimal basis. One consequence is that the large number of cuts produces a large number of (relatively) small facets. This could affect the computational efficiency by increasing the number of pivots necessary to solve the subproblems when branching.

Another effect is that the angles between the active constraints after adding a round of cuts and reoptimizing become smaller. We call this *flattening*. We prove it below.

Consider the Mixed Integer Linear Program:

$$\text{(MILP)} \qquad \min\{c^T x : Ax \geq b,\ x_j \text{ integer for } j \in N_I\}, \tag{3.1}$$

where $c, x, \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and $N_I \subseteq N := \{1, 2, \ldots, n\}$. Non-negativity constraints on all variables are included in $Ax \geq b$, therefore, A is of full column rank. The Linear Programming relaxation, denoted by LP, is obtained from MILP by dropping the integrality constraint on x_j for $j \in N_I$. Let P_I and P denote the sets of feasible solutions to MILP and LP, respectively. Let $\bar{x}$ be an optimal solution of $\min\{c^T x : x \in P\}$.

Let $\mathcal{C}$ be the constraint set of LP indexed by $C := \{1, 2, \ldots, m\}$. Let $a_i \in \mathbb{R}^n$ for $i \in C$ be the normal vectors of the constraints in $\mathcal{C}$. Let $\mathcal{C}(\bar{x}) \subseteq \mathcal{C}$ be the set of constraints with nonbasic slack at $\bar{x}$ and let they be indexed by $C(\bar{x}) \subseteq C$. A further relaxation of the set P with respect to a basic solution $\bar{x}$ is obtained by removing the constraints that are not in $C(\bar{x})$. We denote it by $P(\bar{x})$:

$$P(\bar{x}) := \{x \in \mathbb{R}^n : a_i^T x \geq b_i,\ \forall i \in C(\bar{x})\}. \tag{3.2}$$

This set is a translate of a polyhedral cone: $P(\bar{x}) = K + \bar{x}$, where $K = \{x \in \mathbb{R}^n : a_i^T x \geq 0,\ \forall i \in C(\bar{x})\}$ and $\bar{x}$ solves $\{x \in \mathbb{R}^n : a_i^T x = b_i,\ \forall i \in C(\bar{x})\}$. Let the extreme rays of K, r^j for $j \in C(\bar{x})$, be defined as $r^j := \{x \in \mathbb{R}^n : a_i^T x = b_i,\ \forall i \in C(\bar{x}) \setminus \{j\}$ and $a_j^T x \geq b_j\}$.

Now, if $Cone(\{r^j\})$ denotes the convex polyhedral cone generated by vectors $\{r^j\}$, $P(\bar{x}) = Cone(\{r^j\}_{j \in C(\bar{x})}) + \bar{x}$.

Consider the following procedure: we start with a basic solution $\bar{x}$, generate intersection cuts and add them to the constraint matrix, and reoptimize to obtain a new optimal solution $\bar{x}'$. (For a description of intersection cuts, refer to Section 2.3 and to Balas [11].) Let $C'(\bar{x}')$ be the index set of active constraints at $\bar{x}'$. We can state the following proposition.

Proposition 1. $Cone(\{a_i\}_{i \in C'(\bar{x}')}) \subseteq Cone(\{a_i\}_{i \in C(\bar{x})})$.

This proposition follows directly from the following lemma.

Lemma 1. *Let $\gamma^T x \leq \gamma_0$ be an intersection cut valid for $P(\bar{x})$ and cutting off the current basic solution $\bar{x}$. Then, $\gamma \in Cone(\{a_i\}_{i \in C(\bar{x})})$.*

Proof. Vectors a_i for $i \in C(\bar{x})$ form a basis in $\mathbb{R}^n$, therefore γ can be uniquely expressed as $\sum_{i \in C(\bar{x})} \lambda_i a_i$, for $\lambda_i \in \mathbb{R}$, $\forall i \in C(\bar{x})$. (Note that the basis $\{a_i\}_{i \in C(\bar{x})}$ is non-degenerate by definition.) Then, $\gamma \in Cone(\{a_i\}_{i \in C(\bar{x})})$ is equivalent to $\lambda_i \geq 0$, $\forall i$. Assume the converse, i.e. $\lambda_j < 0$ for some $j \in C(\bar{x})$. By the definition of the extreme rays of $P(\bar{x})$, $a_i^T r^j = 0$ for $i \neq j$ and $a_j^T r^j > 0$. Therefore, $\gamma^T r^j = \lambda_j a_j^T r^j < 0$. But this cannot hold for an intersection cut, where, by definition, $\gamma^T r^i \geq 0$ for all i. □

For two vectors $\alpha, \beta \in \mathbb{R}^n$, let $\sphericalangle(\alpha, \beta)$ denote the angle between them. Proposition 1 implies that $\max\{\sphericalangle(a_i, a_j) : i, j \in C'(\bar{x}')\} \leq \max\{\sphericalangle(a_i, a_j) : i, j \in C(\bar{x})\}$, In words, after adding cuts, the maximum angle between the normal vectors of the active constraints cannot increase. This is shown in Figure 3.1 as well, where the thick lines are the active original constraints, the thin lines are the cuts, and the direction of minimization is up.

After several rounds of cuts, the surface of the polyhedron around the optimal solution becomes much "flatter" than in the beginning. If the maximum angle between the active constraints becomes very small, future rounds of cuts will cut very small parts of the polyhedron and yield only minimal improvement of the integrality gap, and cutting plane generation will eventually be abandoned. We propose to avoid this by adding such a subset of the generated cuts that keeps the angle between active constraints above some threshold. As a result, this will yield a solution $\bar{x}''$ worse than $\bar{x}'$ (see Figure 3.1) but can allow us generate more rounds of efficient cuts. The overall result could be better than adding all

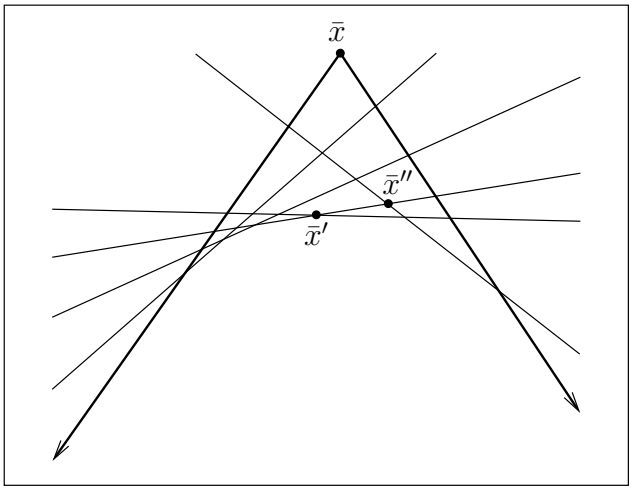

Figure 3.1: A basic solution and cuts

cuts. We show this empirically. Detailed results of the experiments are presented in Section 3.3.

3.2.3 Cut selection algorithms

Various cut selection algorithms are used in the empirical tests in order to show the importance of cut selection and the role of angle between cuts. We present a description of the algorithms in Figure 3.2. We give each algorithm a short name that will be used in the reports of the computational experiments.

In the described algorithms, the selected cuts are accumulated in a cut pool called $\mathcal{P}$. Each algorithm takes as input a round of violated cuts and a numeric parameter that specify the degree of selectiveness of the algorithm.

A trivial procedure for cut selection is one that approves all cuts. We use it for comparison and call it "Add-all."

We define two algorithms that consider only the distance cut off as a criterion for selection. The first one, called "Depth-fix," uses a fixed threshold for the depth, δ. All cuts with depth no less than δ are accepted; the rest are discarded. A drawback of this procedure is that we cannot hope to find a constant δ that would work well for all problem instances because of the large variance of the average depth of generated cuts among problem instances. Furthermore, the estimation of an appropriate threshold for a particular instance

Algorithm "Add-all"

Input: Set S of valid cuts.

1. Pool $\mathcal{P} := S$.

Algorithm "Depth-fix"

Input: Set S of valid cuts. Depth threshold, δ.

1. Let $t := |S|$. Label the cuts in S by $s_1, s_2, \ldots, s_t$.
2. For $i := 1$ to t

   If $\mathrm{dco}(s_i) \geq \delta$, $\mathcal{P} := \mathcal{P} \cup \{s_i\}$.

Algorithm "Depth-dyn"

Input: Set S of valid cuts. Fraction of the cuts to be kept, k.

1. Let $t := |S|$. Label the cuts in S by $s_1, s_2, \ldots, s_t$,
   s.t. $\mathrm{dco}(s_1) \geq \mathrm{dco}(s_2) \geq \cdots \geq \mathrm{dco}(s_t)$.
2. $\mathcal{P} := \{s_1, ..., s_m\}$, where $m = kt$ rounded to the nearest integer.

Algorithm "DA-fix"

Input: Set S of valid cuts. Angle threshold, φ.

1. Let $t := |S|$. Label the cuts in S by $s_1, s_2, \ldots, s_t$,
   s.t. $\mathrm{dco}(s_1) \geq \mathrm{dco}(s_2) \geq \cdots \geq \mathrm{dco}(s_t)$.
2. $\mathcal{P} := \{s_1\}$.
3. For $i := 2$ to t

   If $\cos(\sphericalangle(s_i, p)) \leq \varphi$ for all $p \in \mathcal{P}$,

   $\mathcal{P} := \mathcal{P} \cup \{s_i\}$.

Algorithm "DA-dyn"

Input: Set S of valid cuts. Fraction of cuts to be kept, k.

1. Let $t := |S|$. Label the cuts in S by $s_1, s_2, \ldots, s_t$,
   s.t. $\mathrm{dco}(s_1) \geq \mathrm{dco}(s_2) \geq \cdots \geq \mathrm{dco}(s_t)$.
2. Find φ, such that the number of selected cuts by Algorithm "DA-fix" would be kt rounded to the nearest integer. Use binary search.
3. $\mathcal{P} := \{s_1\}$.
4. For $i := 2$ to t

   If $\cos(\sphericalangle(s_i, p)) \leq \varphi$ for all $p \in \mathcal{P}$,

   $\mathcal{P} := \mathcal{P} \cup \{s_i\}$.

Figure 3.2: Algorithms for cut selection

is a non-trivial task. Even further, a given fixed threshold may not work equally well for all rounds of cuts at the root and throughout the branching tree. The reason is that cuts generated in the first two-three rounds at the root usually have much greater average depth compared to cuts generated later. As a solution, we suggest a more dynamic way of setting the value of δ. For each round of cuts, δ equals the value that keeps a particular fraction k of the cuts. We call this procedure "Depth-dyn."

A variation of a cut selection heuristic that incorporates both distance cut off and angle is called "DA-fix." In this procedure, cuts with larger distance cut off are preferred but they are added to the pool only if they pass the *angle test*, i.e. if the cosines of angles between the candidate and the cuts already in the pool are below some threshold, φ. This algorithm directly controls the minimum angle between two cuts in the pool. As with depth, a fixed value of the angle threshold may not work equally well for all rounds of cuts and for all problem instances. We develop a dynamic version of this procedure, where the threshold is set specifically for every round of cuts at a value that would keep a fixed fraction k of the generated cuts. We call this algorithm "DA-dyn."

During the course of this study, we tested cut selection algorithms that relied solely on the angle between cuts. Those showed inferior results. We conclude that the distance cut off is an important property of cuts and it should be considered.

3.2.4 When to stop generating cuts

Above, we discussed the selection of a set of cuts out of a large pool of generated cuts. Another important question is whether we should generate a next round of cuts or proceed to branching. The answer to this question is usually given by analyzing the improvements in the integrality gap caused by the previous rounds of cuts. We argue that another measure is more relevant.

A major goal of adding cutting planes is to improve the lower, cut-off bound. Therefore, the change in the lower bound is a natural measure of the efficiency of a round of cuts. That change is usually larger for the first rounds and diminishes as we proceed with cut generation. When the change becomes "small," this is interpreted as an indication that further cut generation will be fruitless.

This measure is widely accepted and we believe this is the measure used by the strategies

Parameters:	r: round of cuts to be used as a benchmark
	p: fraction (of the average distance cut off by cuts in round r)
	l: lag, number of previous rounds to consider.
Rule:	If $(i > r)$
	If ($\overline{\text{dco}}(i) < p\,\overline{\text{dco}}(r)$
	and $\overline{\text{dco}}(i-1) < p\,\overline{\text{dco}}(r)$
	$\ldots$
	and $\overline{\text{dco}}(i-l) < p\,\overline{\text{dco}}(r)$)
	Terminate cut generation and proceed to branching.

Figure 3.3: Termination criterion for cut generation

for actively managing cut generation in some current MILP solvers. (A passive strategy would prescribe a fixed number of rounds as long as cuts can be generated. E.g., ten rounds at the root node and one round at specific other nodes.) But practice shows that the improvement in the lower bound does not always tell all the truth. There are cases when the improvement stalls for several rounds in a row, suggesting that cuts have probably exhausted their potency, and then, suddenly, there is a large improvement. For example, in a specific experiment with instance `10teams`, there was no improvement in the gap in the first three rounds but then rounds four and five closed 100% of the gap. Cases like this occur when the optimal face of the LP relaxation is not a vertex and many cuts are needed until all of the points of that face are cut off.

We suggest to use an alternative measure of the progress made by a round of cuts: the average distance cut off by the cuts. (Clearly, other functions of the cuts' depth can be used as well.) The presence of deep cuts suggests that significant parts of the feasible set are cut off and cuts are not out of steam yet. On the contrary, when all cuts in a round are shallow (which could be due to flattening), chances are that future rounds will not be productive.

Usually, rounds of cuts with small average distance cut off cause small improvement in the lower bound. The converse is not true. In the aforementioned example (`10teams`), the average distance cut off in the first three rounds clearly showed that cuts were doing work.

A termination criterion for cut generation may look like the one in Figure 3.3. The

rule is applied after a round of cuts has been generated and before proceeding to a next round. Let $\overline{\text{dco}}(i)$ denote the average distance cut off by the cuts in round i. A particular implementation could be "If the average depth of the current round is less than $p = 30\%$ of the average depth of the first round ($r = 1$), stop cut generation."

As we pointed out, first-round cuts are usually significantly deeper than cuts generated later. Therefore, the average depth of the first round may not be a good benchmark. To overcome this, round $r > 1$ can be chosen as a benchmark.

In addition, computational experience shows that the sequence of average depth per round is not strictly decreasing as a function of the round number. A round of shallow cuts may be followed by a round of deep ones. One may want to observe several consecutive rounds of week cuts before pronouncing future cut generation useless. Parameter l captures this. If the average depth of the current round and the depths of the previous l rounds are all smaller than the benchmark $p\,\overline{\text{dco}}(r)$, cut generation is halted. Based on practical experience, we suggest $r = 3$, $p = 0.5$, and $l = 1$ or 2.

3.3 Experimental observations

In our computational experiments, we study the effects of adding cuts and those of cut selection on the efficiency of the solution algorithm.

The solution algorithms used are branch-and-cut and branch-and-bound (used for comparison). In branch-and-cut, up to 30 rounds of cuts are applied at the root node and three rounds of cuts are applied at the nodes at level l, such that $l \equiv 0 (\text{mod } 4)$ and $0 < l \leq 40$. One round of cuts consists of all generated mixed integer Gomory cuts, mixed integer rounding cuts, and reduce-and-split cuts. After each round, the modified subproblem is reoptimized and inactive cuts are deleted. In the branching phase, we apply full strong branching on the ten most fractional variables, with quality measure $\psi(z_1, z_2) = \lambda \min(z_1, z_2) + (1 - \lambda) \max(z_1, z_2)$ and $\lambda = 5/6$. The node selection rule is best-dive with lowest-LP-bound child selection in diving. (Refer to Section 1.2 for detailed description of the rules.)

As a measure of algorithm efficiency, we often use the progress in the lower bound or, equivalently, the gap closed. The *integrality gap* of a MILP problem is the difference between the optimal objective value and the objective value of the LP relaxation, $z^* - z(P_{\text{LP}})$. Let

the lower bound at a particular moment be $z_{\min}$. The *gap closed* by the algorithm is $z_{\min} - z(P_{\mathrm{LP}})$. A related measure is the *relative gap closed*: $\frac{z_{\min}-z(P_{\mathrm{LP}})}{z^*-z(P_{\mathrm{LP}})}$, the gap closed as a fraction of the initial integrality gap. It is also known as *percentage gap closed*. We use gap closed in our analysis, because the optimal objective value of some instances is not known. Relative gap closed is used in some figures.

The test set consists of 30 problem instances: 28 from MIPLIB 2003 [4] and 2 from MIPLIB 3.0 [20]. We discarded all instances that can be solved in less than 200 nodes, as well as instances that require long time to solve a single subproblem and, as a result, less than 200 nodes are processed in the allotted one-hour limit. The selected instances are: `a1c1s1`, `aflow30a`, `aflow40b`, `air04`, `air05`, `bell4`, `bell5`, `danoint`, `fiber`, `gesa2_o`, `mas74`, `mas76`, `misc07`, `mkc`, `mod011`, `modglob`, `nsrand-ipx`, `p0201`, `pk1`, `qiu`, `rgn`, `roll3000`, `rout`, `set1ch`, `stein45`, `swath`, `timtab1`, `timtab2`, `tr12-30`, and `vpm2`. We denote the set of test instances by $\mathcal{I}$.

The experiments were conducted on a Sysbuilder Intel computer with Intel Xeon 3.20GHz processors (32-bit). The software packages used are BCP [57, 48], an MILP solver, and CGL, a cut generation library – both part of the COIN-OR [33] open-source project. The LP solver is ILOG CPLEX 9.0 [27].

In most of the experiments, we concentrate our attention at the root node. In current state-of-the-art MILP solvers, aggressive cut generation in many rounds is done only at the root. There are good reasons for this. Root cuts are globally valid. In addition, improving the lower bound at the root affects pruning throughout the search tree. As a result, cutting decisions taken at the root node are of much greater importance than those taken at the other nodes.

We start by observing flattening and its effect on cut depth and on the improvement of the lower bound. (Section 3.3.1.) In Section 3.3.2, in a series of experiments, we compare adding all cuts versus adding only 10% of the cuts. We study the effect of this very restrictive cut selection on the gap closed and on the quality of future rounds of cuts. We compare the solution time in the cutting phase, as well. In Section 3.3.3, we conduct another series of experiments with a range of cut selection algorithms. We analyze the effect of adding cuts on the branching time and on the number of fractional variables. In Section 3.3.4, we continue our study on the effect of cut selection. Finally, in Section 3.3.5, we empirically

test the importance of angle as a cut selection criterion when solving the test instances to completion.

Based on the experimental results, we reach the following conclusions.

- Adding cuts leads to:
  - larger number of pivots to reoptimize children;
  - larger number of fractional variables.
- Benefits from cuts selection:
  - closes similar amount of gap, on average;
  - leads to faster reoptimization after a round of cuts are added;
  - leads to faster reoptimization in strong branching, when only a small number of cuts are selected;
  - helps maintain better average depth in future rounds of cuts.
- Maintaining good angle between cuts is beneficial for the efficiency of the branch-and-cut algorithm.

3.3.1 Effect of flattening on cuts

One of the main motivations for applying cut selection is the negative effect of flattening on the improvement of the lower bound. For practically all instances, the first two rounds make the largest improvement in the bound, and the improvement sharply decreases in later rounds. We show this in an experiment where 30 rounds of cuts are generated at the root and all cuts are added.

Our observations are best summarized by Figure 3.4.A. It shows the dynamics of relative gap closed over the 30 rounds of cuts for two instances: `air05` (solid line) and `rout` (dashed line). The first five rounds provide a significant improvement in the gap. The next five rounds still add some improvement but at a decreasing rate. The lower bound practically freezes after round ten. In general, we observe that cuts lose most of their power by the seventh round, approximately, in case all generated cuts are added. In later experiments, we show that cut selection can increase the number of rounds that make a significant improvement in the LP bound.

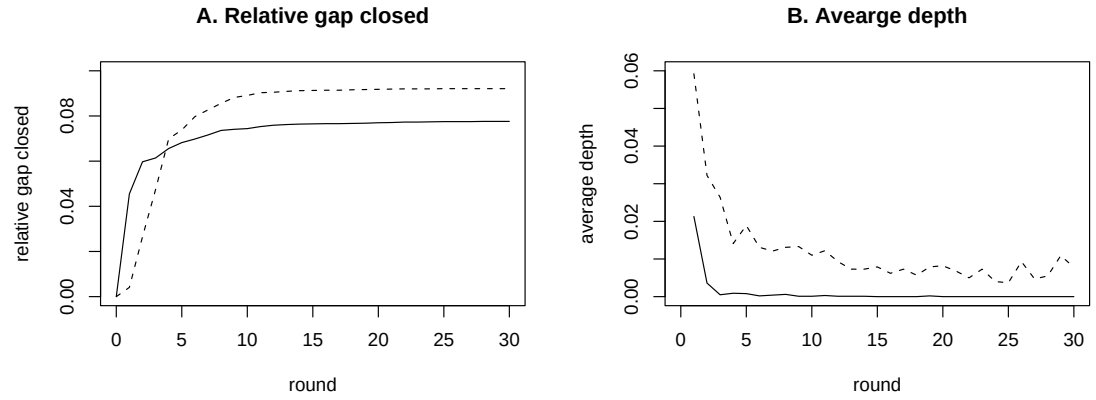

Figure 3.4: Change of (A) relative gap closed and (B) average distance cut off per round. Instances: `air05` (solid line) and `rout` (dashed line).

Decreased improvement in the lower bound is usually associated with a decrease in the depth of generated cuts. We compute the average distance cut off by the cuts in each round, and observe that it decreases in later rounds, coinciding with the decrease of efficiency of cuts. This can be seen in Figure 3.4.B. The average depth drops significantly by the fourth round and stays low after that. Clearly, the decrease of the average depth in these two examples is not due to closing most of the integrality gap and approaching a feasible solution. Gap closed stalls at levels less than 10% (see Figure 3.4.A) because cuts lose their efficiency. We conclude that the change in the average depth of a round of cuts can be used as an indicator of the decreased quality of cuts.

We hypothesize that flattening is a reason for cuts exhausting their power while closing only a small fraction of the gap. Flattening can be detected by observing the number of cuts from different rounds that would be selected by a procedure using a fixed angle threshold, in the spirit of algorithm DA-fix(φ). (Refer to Figure 3.2.3 for a description of the algorithm.) In the first round, a large fraction of the cuts pass the angle test of the algorithm. This fraction drops rapidly in the following rounds. This is an indication that the angles between the cuts become smaller. Table 3.1 contains experimental data showing this. Algorithms DA-fix(φ) for $\varphi \in \{0.866, 0.966, 0.996, 0.999\}$, corresponding to angles of approximately 30, 15, 5, and 2.5 degrees, are run with instance `air05`. For each of the algorithms and for

Table 3.1: Number of generated and selected cuts by algorithm DA-fix(φ) for $\varphi \in \{0.866, 0.966, 0.996, 0.999\}$. Instance: `air04`.

	$\varphi = 0.999$		$\varphi = 0.996$		$\varphi = 0.966$		$\varphi = 0.866$	
Round	Selected cuts	Generated cuts	Selected cuts	Generated cuts	Selected cuts	Generated cuts	Selected cuts	Generated cuts
1	139	217	139	217	139	217	129	217
2	160	232	156	232	25	232	3	232
3	124	207	36	207	1	207	3	225
4	58	228	9	228	4	229	3	225
5	25	233	2	230	2	238	1	233
6	4	228	10	224	1	229	1	205
7	3	231	40	228	2	223	1	230
8	1	228	10	222	1	225	1	231
9	20	222	9	216	1	233	1	219
10	11	224	1	234	1	227	1	235
11	11	231	2	230	1	239	1	224
12	19	207	1	229	1	235	1	234
13	2	231	1	232	1	234	1	228
14	7	228	1	235	3	223	1	231
15	2	226	3	232	1	232	1	229
16	1	214	1	227	1	213	1	235
17	2	227	1	235	1	239	1	238
18	4	234	1	233	1	230	1	231
19	1	228	1	234	1	234	1	231
20	1	231	1	236	1	232	1	232
21	1	215	1	240	1	233	1	241
22	1	215	1	228	1	226	1	233
23	1	225	1	239	1	228	1	237
24	1	239	1	239	1	238	1	234
25	1	232	1	239	1	229	1	237
26	1	234	1	223	1	231	1	235
27	1	237	1	237	1	234	1	236
28	1	239	1	232	1	238	1	241
29	1	234	1	223	1	231	1	241
30	1	228	1	220	1	240	1	240

each of the 30 rounds of cuts, the number of generated and the number of selected cuts are reported. Consider the least restrictive cut selection, DA-fix(0.999), shown in the second and third columns in the table. We observe that the number of generated cuts stays in the narrow interval $[207, 239]$ in the different rounds, while the number of selected cuts decreases from roughly two thirds of the cuts in round one, to just one cut in the late rounds. (In the latter case, the algorithm selects the deepest cut and then discovers that all other cuts are within unsatisfactorily small angle with the selected cut.) As we increase the angle threshold, the drop in the number of selected cuts is faster. We notice that the cuts in the first round are so distinct from each other that even with a 30-degree threshold, 129 out of 217 are approved. (cf. the last pair of columns.) But in the second round, only three cuts pass the 30-degree angle test. These observations clearly indicate the presence of flattening.

3.3.2 Effects of cut selection

Effect of cut selection on gap closed

We conduct the following experiment. We execute three versions of cut selection: Add-all, Depth-dyn(0.1), and DA-dyn(0.1), and compare the gap closed after 30 rounds of cuts generated at the root. We show that the gap closed by the cut selection algorithms, where only 10% of the cuts are used, is roughly the same as that closed by Add-all. This proves empirically that discarding a large fraction of the generated cuts does not hurt the amount of gap closed, provided that a sufficient number of rounds are generated.

We compute the ratio of gap closed by Depth-dyn(0.1) and Add-all for each test instance. The mean of the ratios is 0.971, showing that keeping only the 10% deepest cuts closes on average 97% of the gap that would be closed if all cuts were kept. It is expected that discarding cuts would affect the amount of gap closed — we show that this effect is limited. Based on the experimental results, we claim that Depth-dyn(0.1) closes at least 90% of the gap closed by Add-all. We perform a one-sided statistical test of the hypothesis that the ratio of gap closed is not greater than 0.9. The null hypothesis is rejected with more than 95% confidence (p-value = 0.029).

We repeat the above analysis for DA-dyn(0.1), where 10% of the generated cuts are selected based on depth and angle. We find out that this algorithm performs even better. The mean of the ratios of gap closed by DA-dyn(0.1) and Add-all is 0.995. The amount of

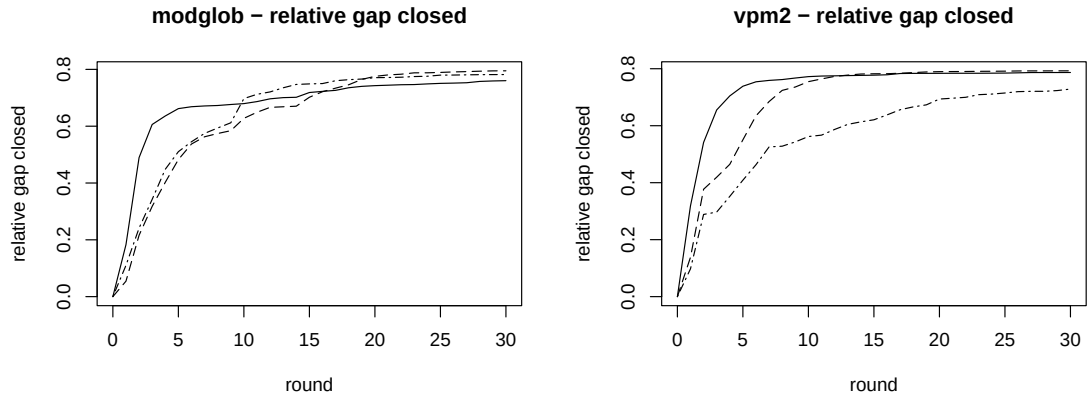

Figure 3.5: Dynamics of relative gap closed by Add-all, Depth-dyn(0.1), and DA-dyn(0.1). Instances: `modglob` (left) and `vpm2` (right).

gap closed by the two algorithms is practically the same. Based on statistical hypothesis tests, we claim that:

(i) DA-dyn(0.1) closes more than 95% of the gap closed by Add-all, with 95% certainty (p-value = 0.049), and

(ii) DA-dyn(0.1) closes more than 90% of the gap closed by Add-all, with 99.9% certainty (p-value = 6.6e-4).

We conclude that applying even very restrictive cuts selection (and discarding 90% of the cuts) does not affect the amount of gap closed significantly.

Clearly, if we generated only three or five rounds of cuts, Add-all would outperform the other algorithms. The cut selection algorithms improve the lower bound at a slower rate but they maintain good rate of improvement for many rounds. As a result, after 30 rounds, all tested algorithms perform similarly. This is evident from Figure 3.5, where the dynamics of the gap closed by the three algorithms are shown for instances `modglob` and `vpm2`. The solid line shows the progress of Add-all, the dash-dot line shows the progress of Depth-dyn(0.1), and the dashed line shows the progress of DA-dyn(0.1). We can see that, given a sufficient number of rounds, the cut selection algorithms close roughly the same gap as Add-all. They can even close more gap, as with instance `modglob`.

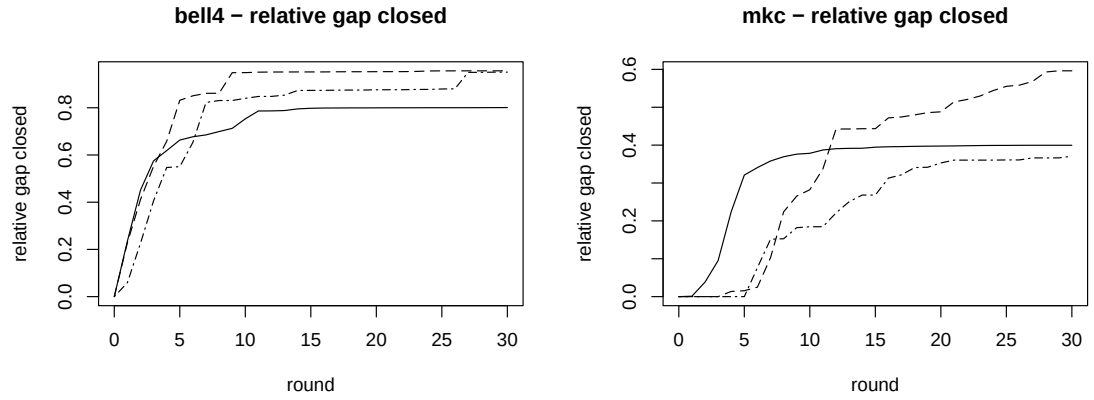

Figure 3.6: Dynamics of relative gap closed by Add-all, Depth-dyn(0.1), and DA-dyn(0.1). Instances: `bell4` (left) and `mkc` (right).

Two cases where cut selection outperforms Add-all significantly are shown in Figure 3.6. With instance `bell4`, cut selection algorithms perform very well even in the first rounds. Algorithm DA-dyn(0.1), in particular, closes as much gap as Add-all in the first three rounds. But while Add-all slows pace in round four, DA-dyn(0.1) continues good improvement. As a result, DA-dyn(0.1) closes 95% of the gap in nine rounds while Add-all needs fourteen rounds to reach its final state of 80% gap closed.

We observe a completely different scenario with instance `mkc`. Discarding 90% of the cuts leads to inability of the cut selection algorithms to close any gap in the first three to five rounds. In five rounds, Add-all closes about 32% of the gap. But it loses power soon after that and stalls at level 40%, while DA-dyn(0.1) makes a steady progress to close 60% of gap in 30 rounds without showing signs of slowing down. These examples show that cut selection can increase the efficiency of branch-and-cut by avoiding flattening.

Effect of cut selection on solution time in the cutting phase

In the previous experiment, we observed that adding just 10% of the cuts generated in 30 rounds of cuts closes roughly the same amount of gap as 30 rounds of Add-all. We also observed that in Add-all, most of the progress is done in the first 6-7 rounds. A reasonable question to ask is: If 7 rounds of cuts (with Add-all) can provide the same lower bound,

isn't it a waste of time to generate 30 rounds and apply cut selection? The answer is: No. We observe that the solution time is much smaller when cut selection is applied. This compensates for the larger number of rounds necessary to obtain the same amount of gap closed.

We measure the average solution time per round, for each instance, denoted by t_r. (The solution time includes time for cut generation and reoptimization, and time for cuts selection, if applied.) We form the ratio of time per round of the two algorithms Depth-dyn(0.1) and Add-all: t_r(Depth-dyn(0.1)) / t_r(Add-all). The average of this ratio over all instances is 0.281. (In addition, there is a sufficient evidence to claim that it is smaller than 0.33, with 95% confidence.) Similarly, the average of t_r(DA-dyn(0.1)) / t_r(Add-all) is 0.349. (It is smaller than 0.4 with 95% confidence.)

We conclude that the two cut selection algorithms are significantly faster than Add-all — roughly three to four times. This allows them to generate three to four times more rounds of cuts than Add-all, for the same time.

Effect of cut selection on future rounds of cuts

An important observation is that cut selection helps maintain a better average depth of the future rounds of cuts.

In the same experimental setting as above, we run Add-all, Depth-dyn(0.1), and DA-dyn(0.1) algorithms, generating 30 rounds of cuts at the root. In each round, Add-all adds all of the generated cuts, while the other two algorithms select only 10% of the cuts. For each algorithm, we compute the average of the distance cut off by all cuts generated in the 30 rounds. (Note that the average is taken over all generated cut, and not over the added cuts.) We denote the average by dco_{avg}. The mean of the ratio dco_{avg}(Depth-dyn(0.1)) / dco_{avg}(Add-all), is 1.81. Similarly, the mean of the ratio dco_{avg}(DA-dyn(0.1)) / dco_{avg}(Add-all), is 1.35. (Both ratios are statistically significantly larger than one, with confidence 99.9% and p-values 2.1e-7 and 1.9e-4, resp.)

In other words, cuts generated by Depth-dyn(0.1) are 81% deeper than those generated by Add-all, on average. Cuts generated by DA-dyn(0.1) are 35% deeper than those generated by Add-all, on average. This is due to selecting only a small fraction of the generated cuts to be added to the formulation, which results in less flattening.

It is important to note that cuts generated by Depth-dyn(0.1) are 36% deeper than those generated by DA-dyn(0.1), on average. In the same time, DA-dyn(0.1) makes much better progress in the early rounds. In the first three rounds, DA-dyn(0.1) closes 50% more gap than Depth-dyn(0.1), on average. In the first ten rounds, DA-dyn(0.1) closes 25% more gap than Depth-dyn(0.1). (Note that both algorithms add roughly the same number of cuts.) The fact that DA-dyn(0.1) is more efficient despite the smaller average depth of the cuts indicates that depth is not the only cut quality one should be concerned about. Incorporating angle as a cut selection criterion improves the performance of branch-and-cut.

3.3.3 Effects of adding cuts

In this section, we study the effect of adding cuts on the reoptimization time of the children nodes. Not surprisingly, increasing the LP size by adding rows affects the solution time and the number of simplex pivots in reoptimization. What we find surprising is that the number of fractional variables increases as a result of adding cuts.

Effect of cuts on the time to reoptimize children nodes

We make two types of comparisons. First, we run various versions of branch-and-cut where different cut selection is applied, including Add-all. The base case for comparison is pure branch-and-bound. This experiment shows that regardless of the intensity of cut generation, adding cuts leads to a significant increase in the number of pivots necessary to reoptimize the children nodes.

The set of algorithms used in this experiment is $\mathcal{A}$ = {Add-all, Depth-dyn(0.2), Depth-dyn(0.5), Depth-dyn(0.8), DA-fix(0), DA-fix(0.5), DA-fix(0.866), DA-fix(0.966), DA-fix(0.996)}. The cosine bounds of algorithm DA-fix (0, 0.5, 0.866, 0.966, and 0.996) correspond to angles of 90, 60, 30, 15, and 5 degrees, resp. In addition, we run pure branch-and-bound for comparison.

As an estimate of the number of pivots to solve a child node, we use the average number of pivots over all possible children. For each tested algorithm $a \in \mathcal{A}$ and each test instance $i \in \mathcal{I}$, we proceed as follows. After completing the cut generation phase at the root, we perform full strong branching on all fractional variables. Let $J \subseteq N_I$ be the set of fractional variables in the solution of the current subproblem. We solve the $2|J|$ nodes created by

Table 3.2: Ratio of Number of pivots in children reoptimization. P-values of the hypothesis test: "Ratio is not greater than 1."

Algorithm	Mean ratio	p-value
DA-fix(0)	2.42	6.51e-07
DA-fix(0.5)	4.22	7.07e-05
DA-fix(0.866)	4.78	4.79e-05
DA-fix(0.966)	4.96	2.49e-05
DA-fix(0.996)	5.09	3.47e-06
Add-all	4.55	1.39e-06
Depth-dyn(0.8)	5.58	1.70e-05
Depth-dyn(0.5)	4.75	4.34e-05
Depth-dyn(0.2)	3.47	3.96e-05

branching on any of the variables in J. We record the number of performed simplex pivots and compute the average over all $2|J|$ solved subproblems. We denote the average by $p_i(a)$. This average is an estimate of the number of pivots necessary to solve a child node. Similarly, for every instance $i \in \mathcal{I}$, we compute the average number of simplex pivots needed to reoptimize a child of the root node in pure branch and bound. We denote it by $p_i(\text{BnB})$. For each tested algorithm $a \in \mathcal{A}$, we compute the mean of the ratio of $p_i(a)$ and $p_i(\text{BnB})$:

$$|\mathcal{I}|^{-1} \sum_{i \in \mathcal{I}} \frac{p_i(a)}{p_i(\text{BnB})}.$$

The results are shown in the second column of Table 3.2. E.g., on average, algorithm DA-fix(0) needs 2.42 times more simplex pivots to reoptimize the children of the root node compared to pure branch and bound.

We observe that the mean ratios are much larger than 1: they are between 2.4 and 5.6, and usually around 5. We test the statistical hypotheses that the mean ratios are smaller than 1 (one-sided t-test). The null hypotheses are rejected at a very low significance level. (The maximum p-value is of order 1e-5. The p-values of the hypothesis tests are shown in the third column of the table.) This is a very strong indication that adding cuts increases the reoptimization time. Adding even a small number of cuts causes a large increase in the

Table 3.3: Summary statistics

Algorithm	Fraction of cuts selected	Average gap closed by cuts	Average pivots after cuts	Average pivots in branching
DA-fix(0)	0.03	41.5%	79.7	85.6
DA-fix(0.5)	0.08	42.1%	111.6	105.0
DA-fix(0.866)	0.15	44.9%	135.7	110.3
DA-fix(0.966)	0.22	43.3%	129.7	116.0
DA-fix(0.996)	0.33	44.0%	139.3	123.9
Add-all	1.00	43.0%	176.0	122.1
Depth-dyn(0.8)	0.80	42.5%	167.7	128.7
Depth-dyn(0.5)	0.50	42.6%	128.6	115.2
Depth-dyn(0.2)	0.20	43.6%	123.1	102.9
Branch-and-bound	N/A	N/A	N/A	48.7

number of pivots. For example, DA-fix(0) adds only 3% of the generated cuts, on average, and this causes an increase of 2.42 times. Algorithm DA-fix(0.5) adds 8% of the cuts and this leads to 4.22 times more pivots.

The fraction of cuts selected by a particular algorithm, averaged over all instances, is shown in the second column of Table 3.3. The last column of the table shows the average number of pivots performed in strong branching, per subproblem. In addition, the table shows the average number of pivots in reoptimization after each round of cuts and the average relative gap closed by cuts. These data are analyzed in the following section.

In a second experiment, we study whether cut selection alleviates the negative effect on reoptimization speed. The test cases are the various cut selection routines, $\mathcal{A}\backslash\{\text{Add-all}\}$. The base case for comparison is Add-all. The second column of Table 3.4 contains the values of

$$|\mathcal{I}|^{-1} \sum_{i \in \mathcal{I}} \frac{p_i(a)}{p_i(\text{Add-all})},$$

for $a \in \mathcal{A}\backslash\{\text{Add-all}\}$. E.g., algorithm DA-fix(0) requires only 62% of the simplex pivots

Table 3.4: Ratio of Number of pivots in children reoptimization. P-values of the hypothesis test: "Ratio is not less than 1."

Algorithm	Mean ratio	p-value
DA-fix(0)	0.62	6.7e-10
DA-fix(0.5)	0.94	0.23
DA-fix(0.866)	1.04	0.66
DA-fix(0.966)	1.15	0.80
DA-fix(0.996)	1.13	0.93
Depth-dyn(0.8)	1.23	0.99
Depth-dyn(0.5)	1.03	0.69
Depth-dyn(0.2)	0.79	9.3e-04

that Add-all needs to reoptimize the children of the root node.

We observe that the ratios are smaller than one only for the most restrictive cut selection rules. The ratios are statistically significantly smaller than one for DA-fix(0) and Depth-dyn(0.2), with 99.9% confidence. We conclude that a considerable decrease in the number of added cuts is necessary in order to get a 20%–40% decrease in the number of pivots.

Effect of cut selection on the number of fractional variables

We compare the number of fractional variables in the optimal basis of the LP relaxation and that number after adding 30 rounds of cuts. We compute the ratio of the latter and the former for each instance $i \in \mathcal{I}$ and each algorithm $a \in \mathcal{A}$. The mean ratios are shown in Table 3.5. We observe that adding cuts leads to an increase of roughly 40% in the number of fractional variables. The ratios we report are statistically significantly greater than one, with confidence greater than 99%. We conclude that adding cuts causes a significant increase in the number of fractional variables, regardless of the number of added cuts.

We can see that the mean ratios reported above are very close. A formal test shows that the difference between the cut selection algorithms and Add-all is not statistically significant. The increase in the number of fractional variables cannot be minimized by cut selection.

Table 3.5: Mean ratio of the number of fractional variables, after vs. before cuts. P-values of the hypothesis test: "Mean ratio is not greater than 1."

Algorithm	Mean ratio	p-value
DA-fix(0)	1.33	0.0014
DA-fix(0.5)	1.41	0.0016
DA-fix(0.866)	1.40	0.0008
DA-fix(0.966)	1.42	0.0005
DA-fix(0.996)	1.39	0.0011
Add-all	1.36	0.0027
Depth-dyn(0.8)	1.40	0.0016
Depth-dyn(0.5)	1.43	0.0011
Depth-dyn(0.2)	1.39	0.0019

3.3.4 More on the effects of cut selection

Effect of cut selection on the time to reoptimize after each round of cuts

When adding cuts, time is consumed for cut generation as well as for reoptimization after each round of cuts. Significant savings in the latter can be achieved through cut selection.

For each tested algorithm $a \in \mathcal{A}$ and each test instance $i \in \mathcal{I}$, we generate 30 rounds of cuts at the root. Let the number of pivots performed in the reoptimization after round r be $q_i^r(a)$. Let $q_i(a) = \sum_{r=1}^{30} q_i^r(a)/30$ be the average over the 30 rounds. We compare the cut selection algorithms in $\mathcal{A}$ versus Add-all. For each algorithm $a \in \mathcal{A}\backslash\{\text{Add-all}\}$, we compute the mean of the ratio of $q_i(a)$ and $q_i(\text{Add-all})$:

$$|\mathcal{I}|^{-1} \sum_{i \in \mathcal{I}} \frac{p_i(a)}{p_i(\text{Add-all})}.$$

The results are shown in the second column of Table 3.6.

For all tested methods for cut selection, statistical hypothesis tests show that the number of pivots is reduced significantly compared to the base case where all generated cuts are added. We can claim this with more than 95% certainty, except for Depth-dyn(0.8), where the certainty is 90%). We observe that the fewer the cuts added, the fewer the pivots needed

Table 3.6: Mean ratio of the number of pivots in reoptimization after rounds of cuts. P-values of the hypothesis test: "Mean ratio is not less than 1."

Algorithm	Mean ratio	p-value
DA-fix(0)	0.495	1.85e-13
DA-fix(0.5)	0.686	3.54e-08
DA-fix(0.866)	0.824	0.0011
DA-fix(0.966)	0.862	0.0253
DA-fix(0.996)	0.913	0.0315
Depth-dyn(0.8)	0.968	0.0923
Depth-dyn(0.5)	0.846	0.0006
Depth-dyn(0.2)	0.696	3.01e-06

to reoptimize, on average. The decrease can be as much as 50% (DA-fix(0)).

Effect of cut selection on gap closed

The above experiments show that we can gain speed by applying cut selection. But there is a trade-off: discarding cuts may decrease the cutting power of the round of cuts and may lead to a smaller improvement in the lower bound. In Section 3.3.2, we showed that algorithms Depth-dyn(0.1) and DA-dyn(0.1) close roughly the same amount of gap as Add-all. Here, we confirm this observation for the larger set of algorithms $\mathcal{A}\backslash\{\text{Add-all}\}$.

For each algorithm $a \in \mathcal{A}$ and for each instance $i \in \mathcal{I}$, let $g_i(a)$ be the gap closed after 30 rounds of cuts. We are interested in the ratio of gap closed by $a \in \mathcal{A}\backslash\{\text{Add-all}\}$ and gap closed by Add-all. The second column of Table 3.7 contains the averages of these ratios:

$$|\mathcal{I}|^{-1} \sum_{i \in \mathcal{I}} \frac{g_i(a)}{g_i(\text{Add-all})},$$

for $a \in \mathcal{A}\backslash\{\text{Add-all}\}$. Although most of them are larger than one, the difference from one is not statistically significant. We cannot claim that cut selection methods perform better than adding all cuts but there is a sufficient evidence that for most of them, the gap closed is at least 95% of the gap closed by Add-all. We can claim this for DA-fix(0.866), DA-fix(0.966), DA-fix(0.996), Depth-dyn(0.8), and Depth-dyn(0.5), with 95% confidence. In

Table 3.7: Mean ratio of gap closed with cut selection vs. Add-all. P-values of hypothesis tests "Mean ratio ≤ 0.95" and "Mean ratio ≤ 0.9."

Algorithm	Mean ratio	p-value Ratio≤ 0.95	p-value Ratio≤ 0.9
DA-fix(0)	0.971	0.247	1.35e-02
DA-fix(0.5)	1.007	0.087	7.13e-03
DA-fix(0.866)	1.042	0.016	8.13e-04
DA-fix(0.966)	1.003	0.001	2.44e-07
DA-fix(0.996)	1.010	0.033	7.99e-04
Depth-dyn(0.8)	1.009	0.001	7.52e-07
Depth-dyn(0.5)	0.986	0.037	5.46e-05
Depth-dyn(0.2)	1.010	0.092	9.20e-03

addition, all tested cut selection methods close at least 90% of the gap closed by Add-all, with 99% confidence. (The p-values are shown in the third and fourth columns of Table 3.7.) We conclude that there is a significant empirical evidence that selecting a small subset of the generated cuts does not hurt the improvement in the lower bound significantly.

3.3.5 Effect of angle

In a practical cut selection algorithm, depth and angle thresholds should be established dynamically, since no fixed threshold can be expected to perform well on all problem instances and at all nodes of the search tree. We proposed two cut selection algorithms that do this: Depth-dyn(k) and DA-dyn(k), which keep a specified fraction of the generated cuts. (Refer to Section 3.2.3.) In this section, we test the effect of incorporating the angle between cuts in a cut selection routine by comparing the performance of these algorithms for fixed k. In addition, we test our proposed rule for cut generation termination with parameters $r = 3$, $p = 0.5$, and $l = 2$. (Section 3.2.4.)

We run branch-and-cut with a time limit of one hour. Strong branching on the ten most fractional variables is applied at each node. We apply the following cut selection rules:

Depth-dyn(0.5), DA-dyn(0.5), and DA-dyn(0.5) with the termination rule embedded. (We call the last algorithm "DA-dyn(0.5)+.") Our goal is to compare algorithms that select comparable number of cuts. In addition, we run Depth-dyn(0.25), DA-dyn(0.25), and DA-dyn(0.25)+, and compare their results separately.

The ratio of gap closed in one hour by DA-dyn(0.5) and Depth-dyn(0.5) is 1.14. Similarly, the ratio of gap closed by DA-dyn(0.25) and Depth-dyn(0.25) is 1.03. Both of them are significantly greater than 1, with confidence 95%. (P-values 0.021 and 0.050, resp.) These ratios are averages over all test instances, part of which are solved in the alloted time. The ratio for the solved instances is one, which may lower the actual ratio of efficiency of the two algorithms. If we compute the ratios for the interrupted instances only, we obtain 1.21 and 1.06, for $k = 0.5$ and 0.25, resp. Both are significantly larger than one, with confidence 95%. (P-values 0.024 and 0.036, resp.) These results show that incorporating the angle between cuts in the cut selection procedure leads to an improvement in the gap closed for the same amount of time. Adding the termination rule to DA-dyn(k) does not bring a significant improvement. The gap closed by DA-dyn(k)+ is very close to that of DA-dyn(k), for the tested k.

In the same experiment, we compare the time spent in strong branching per node. We find out that Depth-dyn(k) and DA-dyn(k) do not differ significantly according to this criterion. On the other hand, adding the termination rule in DA-dyn(k)+ brings on an improvement of 21% to 29% in the branching time over the former two algorithms. According to statistical t-tests, the improvement is statistically significant with 99.9% confidence.

Adding cut generation termination criterion to the algorithm speeds up the solution. It can clearly reduce the time for the cutting phase by reducing the number of rounds. But it also reduces the reoptimization time in branching, as we just showed. One reason may be the smaller size of the subproblems. On the other hand, terminating cut generation when future rounds do not look promising helps prevent flattening. The result of this experiment may be an evidence of a negative effect of flattening on branching.

3.4 Conclusion

We studied empirically the effect of adding cuts to branch-and-cut algorithms. We observed that adding even a small number of cuts leads to:

(i) significant increase in the reoptimization time;

(ii) significant increase in the number of fractional variables in the optimal basis.

We introduced the notion of flattening and studied its effect on the quality of cuts. We justified the importance of angle between cuts and developed a cut selection algorithm that evaluates the quality of a family of cuts as a group. The benefits of this algorithm are:

(i) decreased reoptimization time;

(ii) improved numerical stability;

(iii) improved polyhedral properties that increase the chance of generating good cuts in future rounds.

We observed that the gap closed does not decrease significantly even when most of the cuts are discarded.

These results show the important role of proper selection of cuts for the efficiency of branch and cut.

We studied the large impact that cut selection has on cut generation. But the shape in which we leave the formulation after cuts (in particular, the angles between the active constraints) is likely to have an effect on branching, as well. The interaction of cuts and branching is a very interesting and practically unexplored area. Better understanding of this interaction, as well as developing good measures of cut quality will guide us towards the answer of the question of great practical importance: Shall we generate another round of cuts or proceed to branching, at a particular node of the search tree. We hope to see significant future development in this direction.

Chapter 4

Early estimates of the size of branch-and-bound trees

4.1 Introduction

The effectiveness of the branch-and-bound procedure for solving Mixed Integer Linear Programming (MILP) problems has made it a method of choice in commercial software for several decades. Its applicability to large instances has increased in the last ten years with the increased computational power of computers as well as substantial improvements in algorithms. Although current software packages are able to solve many large instances by branch and bound and its modifications, there are also many other instances where they fail due to the excessive size of the enumeration tree.

The branch-and-bound algorithm is a divide-and-conquer approach that dynamically constructs a search tree, each node of which represents a subproblem. Upper and lower bounds can be obtained from feasible solutions and from solving the linear programming relaxation of these subproblems. These bounds are used to prune the tree. In addition to the bounds, the search strategies determine the size and shape of the search tree.

The application of the branch-and-bound algorithm can be limited by both the computing time and the storage space required (even when storing nodes on a hard disk). The solution process may take hours or days and there is very little a priori indication of how difficult a model will be to solve. Unfortunately, there is no known method to extract this information from the problem formulation. Practice shows that even small modifications

in a model can increase or decrease the solution time by an order of magnitude. On the other hand, many present-day applications require a solution of MIP problems within minutes. Some specialized commercial software products for solving MIP problems apply only heuristics because speed is more important than obtaining an optimal solution. Therefore, from a practical point of view, even a rough estimate of the computing time required by branch and bound would be useful. It can help decide whether to continue with branch and bound or switch to heuristics.

Memory requirements are also critical. To store the tree may require enormous space and it is possible for the branch-and-bound algorithm to terminate prematurely after many hours of work without providing a satisfactory solution due to a lack of memory. For example, some instances from MIPLIB, a standard library of test problems, require many gigabytes for node storage.

The CPU time required for a branch-and-bound solution depends roughly linearly on the number of nodes in the branch-and-bound tree (for simplicity, we will call it *the tree*). In the present paper, we attempt to devise a method for estimating the total size of the tree at an early stage of the solution process. We define the following requirements for the method:

- It should function as part of a general-purpose MILP solver. It should provide predictions without controlling or directing the solution process.
- It should be able to output a prediction with satisfactory precision after a short period of time (e.g., five seconds for medium-size problems). It should be able to update the prediction as time elapses.
- The additional computations for these predictions should consume a negligible amount of time compared to the branch-and-bound algorithm. They should not slow down the solution process. They should rely as much as possible on the data obtained from the MILP solver.

"Satisfactory precision" can be defined in different ways. We propose to measure the precision by the *error factor*—the factor by which the prediction under- or overestimates the actual tree size. We consider that a prediction within an error factor of five provided after five seconds of solution time is satisfactory. Such a prediction will allow us to conclude

whether the solution will take minutes, hours, or days. For example, an estimated solution time of one hour would be interpreted as saying that the instance can be solved between 12 minutes and five hours. If we had set a time limit of ten hours and the actual solution time of an instance exceeded ten hours, the one-hour prediction would not be considered satisfactory, whereas a four-hour prediction would be.

We introduce a notion related to the shape and size of a tree called the γ-*sequence*. The nodes at distance i from the root node are said to occupy *level* i. Let the *width* of a level be the number of nodes at that level. We define γ_i as the ratio between the width of level $i+1$ and that of level i. The γ-sequence of a tree is the sequence of γ_i for all levels i with positive width. Given the γ-sequence of a tree, we can reproduce the number of nodes at each level. Our main goal is to obtain a satisfactory approximation to the γ-sequence. After running the branch-and-bound algorithm for a short period of time, we obtain a subtree of the whole branch-and-bound tree. One approach is to use the γ-sequence of the partial tree as a basis for the estimation. Our tests showed that this does not lead to good results. Instead, we will use the partial tree to estimate three key parameters of the complete tree: the depth, the last full level, and the waist level. We will use these parameters for modeling the γ-sequence. We describe and analyze our approach and present our computational results in Section 4.3.

Knuth [40] proposed a procedure for estimating the size of branch-and-bound trees based on sampling by random paths. Discussion of this method and its application is included in Section 4.2.

The test-bed for the experiments described in this paper includes 28 instances from MIPLIB 3.0 [20]. The solution of most of them requires building a branch-and-bound tree of more than 1000 nodes. We also included some of the "smaller" instances. The choice is aimed at obtaining diversity while concentrating on the nontrivial instances. We also tested our prediction algorithm on additional instances from the literature. The computations were made on a Sun Ultra 60 (360MHz UltraSPARC-II processor) with ILOG CPLEX 8.0 [27]. Because our goal in this paper focuses on the branch-and-bound algorithm, we did not apply heuristics and cuts at nodes other than the root node, except at the very end of Section 4.3 where we briefly report on our experience with branch and cut.

4.2 Earlier work

Knuth [40] was the first to discuss how to estimate the size of a general backtrack tree. The method that he proposed is a random exploration of the tree based on a Monte Carlo approach. The algorithm repeatedly traverses random paths from the root node to the leaves, without backtracking. At each node, one of its successors is chosen at random according to a uniform probability. The estimate of the number of nodes in the tree is the average over several runs of $1 + d_1 + d_1 d_2 + \cdots + \prod_{i=1}^{k} d_i$, where d_i is the number of successors to the chosen node at level i and k is the depth reached. Furthermore, Knuth generalized the above simple, unbiased method to allow the selection of random paths under non-uniform probabilities. He proves that the expected value of both estimates, unbiased and biased, is the size of the search tree, and he provides upper bounds on the variance of the estimates.

Knuth's algorithm has been improved in various ways by Purdom [56] and Chen [22]. The modified algorithm by Purdom attempts to reduce the variance of the estimate by allowing more than one branch out of a node to be further investigated. Chen adopted a stratified-sampling approach, based on a "heuristic function" (stratifier) supplied by the algorithm designer. Chen proved that, by exploiting the tree structure reflected by the stratifier, the heuristic sampling method reduces the variance relative to Knuth's algorithm.

Although he did not present many test results, Knuth provided a good insight into the potential problems that may arise when applying his procedure. He emphasized the large variance of the estimator, as well as the tendency to get underestimations when the deep levels are visited with very low probability. Another important remark was that "the estimation procedure does not apply directly to branch-and-bound algorithms," unless the optimal objective value is given a priori as a bound. Thus, the procedure can be used to estimate the amount of work to prove any given bound for optimality.

Nevertheless, Knuth's method has been employed for estimating the size of a branch-and-bound tree. Lobjois and Lemaitre [45] proposed a method to select, for each instance of the maximal-constraint-satisfaction problem, the most appropriate branch-and-bound algorithm from among several candidates. They compared the running times of the algorithms predicted by Knuth's procedure and concluded that, despite the great variability of the estimates, it selects the best algorithm in most cases. One conclusion was that Knuth's

estimator can be used for comparison purposes even when it outputs imprecise predictions.

Brüngger et al. [21] set up such an estimator in their solver to predict the running time when tackling large-scale quadratic assignment problems (QAP) by parallel computation. Anstreicher et al. [7] also used Knuth's procedure for estimating the solution time of a specialized branch-and-bound algorithm for QAP. They reported excellent results of the basic, unbiased method for instances of size less than 24 but pointed out that the quality of the estimation rapidly deteriorates as the size of the problems increased. To fix this, they applied "importance sampling," identical to the biased sampling proposed by Knuth. They suggested the use of non-uniform probabilities that depend on the inherited relative gap at a node. In addition, they proposed a way of reducing the variance of the estimates at deeper levels in the tree and avoiding wasteful duplication of computations at low levels. Rather than start the random dives at the root node, they first ran the branch-and-bound algorithm in breadth-first mode to obtain all nodes at a predetermined level, and then initialized Knuth's algorithm from a node at that level. The modified procedure output very good estimations for QAP problems of size up to 30.

We performed tests with our set of MIPLIB instances but could not observe the good estimation properties of Knuth's procedure reported in the aforementioned papers. We applied unbiased random sampling with 1000 iterations. Even when the optimal objective value was provided as a cutoff bound, the error factor of the prediction was greater than five in 11 of the 23 instances solved to optimality (cf. Table B.1 in Appendix B). Five instances were not solved within ten hours of computing time and 1GB of storage space. For these instances, a correct prediction should exceed the number of nodes at interruption. Knuth's method provided such a prediction in three cases and produced significant underestimations in the other two cases. The large number of errors can be attributed to the large variance of the estimator and to the insufficient number of iterations (1000 while Anstreicher et al. proposed 10,000). However, even with this relatively small number of samples, the estimation time was significant. It was greater than one minute for all problems but one, and it was greater than five minutes in ten out of 28 cases. For many of the instances, this is an unreasonably long period of time devoted solely to time estimation without contributing to the solution. Moreover, in 11 cases, the number of nodes visited during the estimation procedure exceeded the size of the branch-and-bound tree, i.e., the estimation procedure took longer to execute than the solution algorithm itself. Even with this abundant infor-

mation, the prediction error factor was greater than five in five of these 11 cases. When we applied 10,000 samples in order to obtain a more precise estimate, the estimation time became longer than the solution time for 20 out of 28 instances. This made the price for the increased precision too high.

Furthermore, starting with information about the optimal objective value is not realistic. The above experiment, repeated with no cutoff bound, lead to huge overestimations (by factors of 10^2 to 10^{31}) for almost all problems while the estimation time was even longer than that reported in Table B.1.

We can see three main reasons for the observed inaccuracy. First, the lack of a good cutoff bound results in a very small amount of pruning in the second experiment. Second, due to the exponential growth of the estimate with node depth, the error tends to be small when the tree is shallow, as those studied by Anstreicher et al., but when the tree is deep, e.g., more than 100 levels, even one or two sample paths that go to the deepest levels can cause a huge overestimation. Third, it is possible that Knuth's procedure works much better in some classes of problems and with some types of branch-and-bound algorithms than with others. (The algorithm employed by Anstreicher et al. is specialized for QAP.)

Our conclusion is that in most cases Knuth's method is not practical for early prediction of the solution time of general MILP problems. We would like to have a much faster routine with acceptable precision that does not assume prior knowledge of the optimal objective value.

Our exploration in this paper is distinct from Knuth's work in the following four respects. First, it does not rely on an initial bound, although having such a bound would be advantageous. Second, our procedure employs a standard branch-and-bound algorithm, which does backtracking and updates the bound. Third, our method is based on estimating parameters of the enumeration tree and extrapolating its γ-sequence from these parameters, rather than estimating the γ-sequence directly by the number of descendants. Finally, our estimation procedure analyzes the partial tree produced by the branch-and-bound algorithm and then continues the search. Therefore, the work done in the estimation phase is essentially the beginning of the solution process. While the random sampling can find a good solution by chance, the time spent by this sampling procedure is usually lost for the solution of the problem.

Some other means of estimating the termination time of a branch-and-bound algorithm

have been considered as well. One idea is to estimate upper and lower bounds on the objective value as a function of time and then apply simple regression. A ballpark estimate of running time can be obtained by extrapolating those curves and predicting when the gap will be zero. One could also consider the number of active nodes in the queue as a function of time and, again, extrapolate the curve. Both approaches require a significant amount of solution time in order to capture the trend. The gap closes in large steps at the beginning and in much smaller ones later on. The behavior of the set of active nodes is very problem-specific. Our experience shows that the gap closed in the first 5–10 seconds and the dynamics of the set of active nodes in the same period hardly provide sufficient information to make a sensible prediction. However, combining all these methods could lead to a more precise estimator. Further investigation in this direction should be fruitful.

4.3 Our method

4.3.1 General Description

In what follows, we assume that the maximum number of descendants of a node is two. By redefining the last full level, our estimation procedure can accommodate a branching scheme with any number of descendants.

Definition 1. *In a branch-and-bound tree T, let $w_T(i)$ be the* width *of level i, i.e., the number of nodes at that level. Let $d_T = max\{i : w_T(i) > 0\}$ be the* depth *of the tree. Level $l_T = \min\{i : \frac{w_T(i+1)}{w_T(i)} < 2,\ 0 \le i \le d_T\}$ is called the* last full level *of the tree (assuming that each node has at most two successors). Up to this level, the tree is a complete binary tree. Let the* waist *of the tree be the level with maximum width, $b_T = \arg\max\{w_T(i) : 0 \le i \le d_T\}$. When this level is not unique, define $b_T = \left\lceil \frac{b_1+b_2}{2} \right\rceil$, where $b_1 = \min\{i : w_T(i) = t\}$, $b_2 = \max\{i : w_T(i) = t\}$, and $t = \max\{w_T(i) : 0 \le i \le d_T\}$, i.e., b_T is the center of the smallest interval containing all the levels with maximum width. Let $n(T) = \sum_{i=0}^{d_T} w_T(i)$ be the number of nodes in T. The sequence $\{w_T(i) : 0 \le i \le d_T\}$ is called the* profile *of tree T.*

A framework for estimating the size of a branching tree T is given in Figure 4.1. The branch-and-bound algorithm is paused at a given point in time. The resulting tree of visited nodes, t, also called the *partial tree*, is used to estimate the parameters of the complete branch-and-bound tree. In Step 3 we find the last full level, the waist, and the

Input: An MILP problem

Initialization: Set counters for the width of the levels, $w(i) = 0$, for $i = 0, \ldots, \widetilde{D}$, and $\widetilde{D}$ large enough.

Step 0. Run the branch-and-bound algorithm.

Step 1. At each node of the branching tree, increment the corresponding counter by 1.

Step 2. Stop when a pre-specified event occurs (e.g., a time or node bound is reached), and let t be the resulting subtree of T. Set $w_t(i) = w(i)$, $\forall i$.

Step 3. Find l_t, b_t, and d_t.

Step 4. Construct a measurement tree M.

Step 5. If the problem is not solved, output $n(M)$ (an estimate of $n(T)$), continue the branch-and-bound algorithm, and go to Step 1. Otherwise, terminate.

Figure 4.1: Framework of the Algorithm

depth of the partial tree, which serve as estimates of the parameters of the complete tree. The *measurement tree* constructed in Step 4 is not a real tree but a profile of a tree that is designed to replicate, as much as possible, the profile of the estimated tree. It is built according to a model to be discussed in the next section. The number of nodes in the measurement tree is used as an estimate of the total number of nodes in the branch-and-bound tree.

We apply this procedure repeatedly in order to output periodic estimations until the branch-and bound algorithm terminates. We formally divide the solution process into two phases. Phase I ends with the output of the first prediction. In Phase II, we output periodic predictions. As a termination criterion for Phase I, we require that both of the following conditions be satisfied: the solution time is at least five seconds and the number of nodes in the partial tree is at least 20 times the depth of the partial tree ($n(t) \geq 20d_t$). The second condition is important because a reasonable level width is necessary in order to obtain a sensible approximation of the parameters of the complete tree. The factor of 20 is established empirically and can be changed to reflect tradeoffs between speed and accuracy of the first prediction.

In the above procedure, branch and cut can be used instead of branch and bound. It is important to note that cuts added after branching has started can change the structure of the branching tree and affect the validity of the predictions.

4.3.2 The Linear Model for Estimating the γ-Sequence

In this section, we describe a model for the measurement tree needed in Step 4 of the above procedure. We propose to model the profile of the complete tree using three parameters only, namely l_t, b_t, and d_t.

A characteristic that uniquely defines a tree profile is its γ-sequence—the ratios that describe the change of width from one level to the next.

Definition 2. *Consider a branch-and-bound tree T and let d_T be its depth. The sequence $\gamma_0, \gamma_1, \ldots, \gamma_{d_T}$ is called the γ-sequence of this branch-and-bound tree, where $\gamma_i = \frac{w_T(i+1)}{w_T(i)}$, for $0 \leq i \leq d_T$.*

Given the γ-sequence of a tree T, the width of a particular level i is $w_i = \prod_{j=0}^{i-1} \gamma_j$. The size of the tree is $n(T) = 1 + \sum_{i=1}^{d_T} \prod_{j=0}^{i-1} \gamma_j$. A tree model is essentially a model for building the γ-sequence.

We analyzed the profiles of branch-and-bound trees obtained using the CPLEX default solution settings [27]. The solution algorithm was cut and branch, where cuts are applied only at the root node. Our tests show that, for almost all of the problems in MIPLIB, the profile of the tree looks like a bell-shaped curve. The same observation is made by Knuth [40] for backtrack algorithms in general. The γ-sequence is generally decreasing for i greater than the last full level and the value of γ_i is approximately 1 at the waist and 0 at the deepest level. This observation justifies the use of a linear model for the change of γ, defined by the formula:

$$\gamma_i = \begin{cases} 2, & \text{for } 0 \leq i \leq l_T - 1, \\ 2 - \frac{i - l_T + 1}{b_T - l_T + 1}, & \text{for } l_T \leq i \leq b_T - 1, \\ 1 - \frac{i - b_T + 1}{d_T - b_T + 1}, & \text{for } b_T \leq i \leq d_T. \end{cases}$$

This simple model outputs satisfactory estimations in the majority of the cases. Figure 4.2 shows a typical tree profile (the solid line) and the measurement tree obtained by the

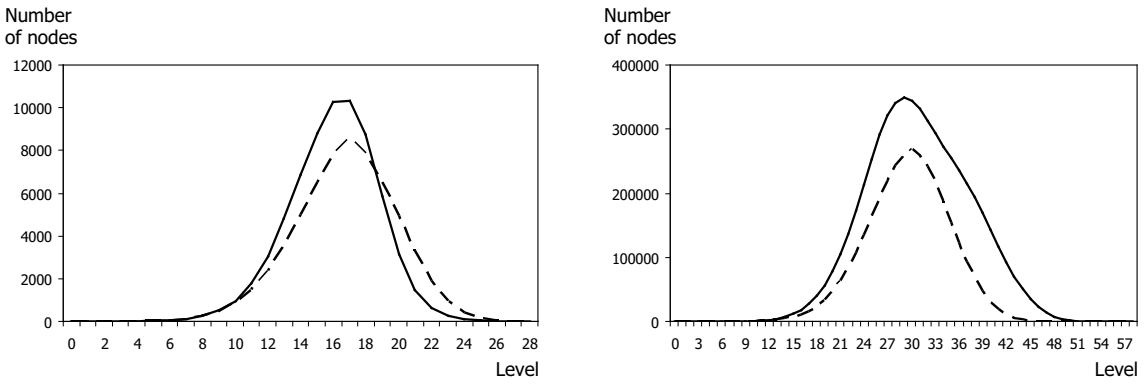

Figure 4.2: Profiles of the Actual Tree and Linear-Model Measurement Tree. Instances: `stein45` and `noswot`.

linear tree model (the dashed line). The proximity of the two lines is common for most problems with a bell-shaped tree profile.

The estimation properties of the linear model are tested in experiments with the set of 28 problems from MIPLIB. In this part, we assume that the exact values of l_T, b_T, and d_T for the complete trees are available and we study the accuracy of the tree model.

Table B.2 compares the size of the measurement tree obtained by the linear model with the actual number of nodes in T. The last column shows the ratio between the two.

The solution of the problems marked with an asterisk has been interrupted after ten hours and, therefore, the figure in the second column is the size of the branching tree at interruption. We analyze this group of problems separately. A desirable output of the tree model for these instances is an estimation greater than the number of nodes processed before interruption. This is observed in four out of five cases. In the fifth case, the ratio between the predicted number of nodes and the number of nodes after ten hours of computation is 0.4. In this paper, we consider this satisfactory (i.e., within a factor of five) although we do not know the true ratio between the predicted and the actual number of nodes.

Twenty-three problems were solved to completion. For 15 problems, the error is within a factor of five. For 11 of them the error is within a factor of two. Our conclusion is that the linear model provides a satisfactory estimation of the actual tree profile for most instances and, therefore, can be used to estimate the number of nodes in the tree.

4.3.3 Computational Experience (MIPLIB Instances)

We performed computational experiments with the procedure described above and our test-bed of 28 MIPLIB problems. We ran the branch-and-cut algorithm of CPLEX 8.0 with its default branching and node-selection rules [27] but with the restriction that cuts and heuristics were applied only at the root node. The results are presented in Table B.3. The solution process was interrupted when solution time exceeded ten hours or when the branching-tree size exceeded 1GB. These instances are marked with an asterisk. We applied the linear tree model for the construction of the measurement tree based on the parameters l_t, b_t, and d_t of the partial tree obtained at the end of Phase I. The predicted tree size is reported in the second column of Table B.3. The third column contains the actual tree size and the ratio between the prediction and the true value is shown in the fourth column. The fifth column contains the time to obtain the prediction.

We compute a time estimate for solving the instance. This time estimate θ equals the size of the measurement tree times the average solution time of a node in the partial tree, based on the assumption that the average running time at a node is relatively constant during the solution. Instead of the point estimate θ, we output a range $[\alpha, \beta]$ for the solution time, shown in column six. The width of this range corresponds to an estimation error of five. Specifically, $\alpha = \max\{\text{Phase I time}, 0.2\theta\}$, $\beta = 5\theta$ (or $+\infty$ when $5\theta > 10$ hours), and we round seconds and minutes to the nearest multiple of five, and hours (or minutes smaller than five) to the nearest integer. The true solution time or the time until interruption is given in the last column. Incorrect predictions are marked by a dagger. For instances solved to completion, a dagger marks the cases for which our prediction interval does not contain the actual solution time. For the instances that are not solved to completion, a dagger marks the cases where we predicted that the instance could be solved in less than ten hours. Note that, when an instance was interrupted because of space limitation, it could happen that our time prediction is within a factor of five of the actual unknown solution time but we still consider these to be incorrect predictions. For example, if the predicted solution time is between 15 minutes and six hours and the solution process is interrupted after two hours because of space limitation, we consider the prediction incorrect.

Five instances, `lseu`, `mod008`, `modglob`, `rgn`, and `stein27`, were solved during Phase I. They are not present in the table. Eighteen of the remaining instances were solved to

optimality. For ten of them the prediction is correct. For two instances, `bell3a` and `gesa2_o`, the error is small. There are four cases, `mas74`, `misc07`, `noswot`, and `rout`, with a considerable error. The solution of five instances was interrupted after exceeding the time or space limit, and this was predicted correctly. Overall, the results are satisfactory considering the great diversity of the instances.

4.3.4 Computational Experience (Additional Instances)

Our procedure for estimating the number of nodes in a branch-and-bound tree was designed based on observations from a diverse sample of 28 instances from the MIPLIB. In order to validate the procedure, we applied it to an independent test set. We used MILP benchmarks from the literature representing several different problem types. The results are reported in Tables B.4, B.5, and B.6.

The first group of instances are multidimensional knapsack problems from Beasley [18] and Chu and Beasley [23]. Due to a significant amount of pruning, the branch-and-bound trees tend to be slim and deep. Our procedure deals relatively well with the lower-dimensional instances (`mknapcb1`, `mknapcb4`, `mknapcb7`) but not as well with higher-dimensional problems. Overall, 13 out of 27 instances are solved to completion. Correct predictions are obtained for seven of them, for two instances the error in prediction is small, and in four cases the error is significant. The solution of 14 instances was interrupted, in most cases because of the space limit. For 11 of them, this interruption was predicted correctly.

The second group consists of set covering instances from Beasley [17, 18]. For these instances, the estimation requires typically more than one minute, in some cases more than ten minutes. The reason is the long solution time at a node (on average, 40 times longer than for the group of multidimensional knapsack problems). The estimation procedure performs well for these instances. There is only one incorrect prediction out of 18.

The third group of instances are bin-packing problems from Falkenauer [30] and Beasley [18]. The solution of all these instances takes more than ten hours and our method provides correct estimations in all cases. Due to the long solution time of a single subproblem, the Phase I time is much longer than five seconds, reaching more than two hours in two cases. This may seem too long for a prediction but this is the time to make only about 20–30

dives in the tree. If Knuth's estimation method were applied with 1000 or 10,000 dives, the prediction procedure would hardly be practical.

The fourth type of instances we tested are capacitated facility-location problems from Beasley [16, 18]. A huge solution tree is typical, which leads to exceeding the space limit after less than three hours in all of the cases. (The space limit applied to this group of instances was 3GB.) Interruption was predicted correctly for all these instances.

Finally, we tested seven other MILP benchmark instances from Argonne National Laboratory and Mittelmann [49, 51]. The prediction is correct for four of them.

In this section, 76 additional instances were tested. Twenty-seven of them were solved to completion. For these 27 instances, the number of correct predictions was 19. In two cases, the prediction is close to the actual solution time and in six cases the error of prediction is significant. The solution of 49 instances was interrupted due to the time or space limit. For them, there are only five cases of incorrect prediction.

Overall, we tested 99 MILP instances. The predicted time range was correct for 78 instances. (In this summary, we exclude the five MIPLIB problems that were solved during Phase I.) In particular, there were 54 instances that required excessive time (more than ten hours) or space (more than 1GB) and this was predicted correctly in 49 cases. For the 45 instances that could be solved within the ten hour and 1GB limits, this fact was predicted correctly for 39 of them. In other words, given the time and space limitations that we set for these experiments, the estimation procedure estimated correctly whether an instance could be solved in 88 out of 99 cases. We conclude that, although not precise, this method often provides a reasonable early estimate of the computing time of a branch-and-bound algorithm.

4.3.5 Analysis and Refinements

In this section we identify several sources of imprecision in our estimation procedure and we discuss possible remedies. We also discuss our experience with branch and cut.

The Linear Model

The results reported in Section 4.3.2 show that for most problems, the linear model has satisfactory precision. However, it does not perform well when the tree is deep and slim.

The problem stems from the fact that the linear model uses only three parameters, the last full level, the waist, and the depth. The model does not incorporate an estimate of the maximum width of the tree, which we might call the *waistline*, i.e., the width at the waist. If a slim tree and a fat tree have identical last full level, waist, and depth, the model will output the same estimation. The linear model assumes that γ decreases linearly from two to one, which reproduces satisfactorily the profiles of most of the branching trees we tested. But if the actual decrease is faster in the beginning, as is the case with deep and slim trees like those of `bell5`, `misc07`, and `mod008`, the waistline will be much less in the real tree than in the measurement tree. This causes significant overestimations by the linear model.

One approach to better model the behavior of γ observed in the slim trees is to use a nonlinear model of the γ-sequence. For example, one could use a convex combination of the linear model γ and its cubic perturbation

$$\tilde{\gamma}_i = \lambda(\bar{\gamma}_i - 1)^3 + (1 - \lambda)(\bar{\gamma}_i - 1) + 1, \qquad \text{for } 0 \leq i \leq d_T$$

where $\lambda \in [0, 1]$ and $\bar{\gamma}_i$ is the γ-sequence obtained by the linear model. This model has the linear model as a special case, when $\lambda = 0$. Increasing λ, it can be tuned to output good estimations to γ-sequences of slim trees, but it does not provide good results for the most common tree profiles when λ is far from 0. The linear model performs better for general MILP problems but if we deal with a special class of problems, it might be worth analyzing the tree profile and tuning the model.

To illustrate the above, we performed tests with the 30 multidimensional knapsack problem instances of the group `mknapcb1` (100 variables, 5 constraints) from Beasley [18] and Chu and Beasley [23]. As we observed in Section 4.3.3, the branch-and-bound trees of this type of problems are usually slim and the linear model overestimates their size. We tried the cubic model with $\lambda = 0.5$. For this choice of λ, in 28 of the 30 cases, the prediction by the cubic model is closer to the actual number of nodes than that of the linear model. The mean error factor of the estimations by the linear model is 1.84, while that of the cubic model estimation is 1.28. (A value of one means exact estimation.)

The cubic model is only one example of improvement. Different models based on different sets of parameters could also prove useful.

Estimating the Waist

Another concern is the quality of the estimation of the tree parameters. Tests of the sensitivity of the linear model to changes in the parameters show that the waist is the most important one. Even small changes in it cause large variations in the number of nodes in the measurement tree, while the variations caused by changes in the depth and the last full level are less significant. On the other hand, our experiments show that d_t and l_t of a small (with respect to the complete tree T) partial tree t are better estimates of d_T and l_T, respectively, than b_t is an estimate of b_T.

State-of-the-art branch-and-bound algorithms employ node-selection rules that are a combination of depth-first search and best-bound search. For example, the default branch-and-bound algorithm of CPLEX 8.0 dives along a path until a node gets pruned and then continues from a best-bound node. As a consequence, shortly after the start of the algorithm, the top levels are well studied and the depth is estimated with good precision. The sampling error present in the partial tree affects mainly the determination of the waist.

The variability of b_t is shown in Figure 4.3. The left figure depicts the tree profiles after 320, 640, 1280, 2560, 5120, and 10715 seconds of solution time of the problem called rout. It can be seen that the waist gradually increases with time up to its final value of 33. This is shown also in the figure on the right, where the thin horizontal line is the waist of the complete tree, b_T, and the thick solid line represents the waist of the partial trees as a function of solution time. As time elapses, the waist approaches that of the complete tree. Greater fluctuations are typical at the beginning of the solution procedure. Tests show that usually $b_t < b_T$ for a small partial tree t, and sometimes the difference is significant. Therefore, often b_t is not a good estimate of b_T.

In some cases, the error in waist estimation can be reduced by considering the levels with large width and taking the estimate of the waist to be the midpoint of these levels. We call this the *average waist* and define it as follows:

Definition 3. *The* average waist *of a tree T is the level $\bar{b}_T = \left\lceil \frac{b_1+b_2}{2} \right\rceil$, where $b_1 = \min\{i : w_T(i) \geq 0.5t\}$, $b_2 = \max\{i : w_T(i) \geq 0.5t\}$, and $t = \max\{w_T(i) : 0 \leq i \leq d_T\}$, i.e., $\bar{b}_T$ is the center of the smallest interval containing the levels with width at least 50% of the maximum width.*

It is not uncommon that $\bar{b}_T \neq b_T$ for the complete tree T, but our tests indicate that

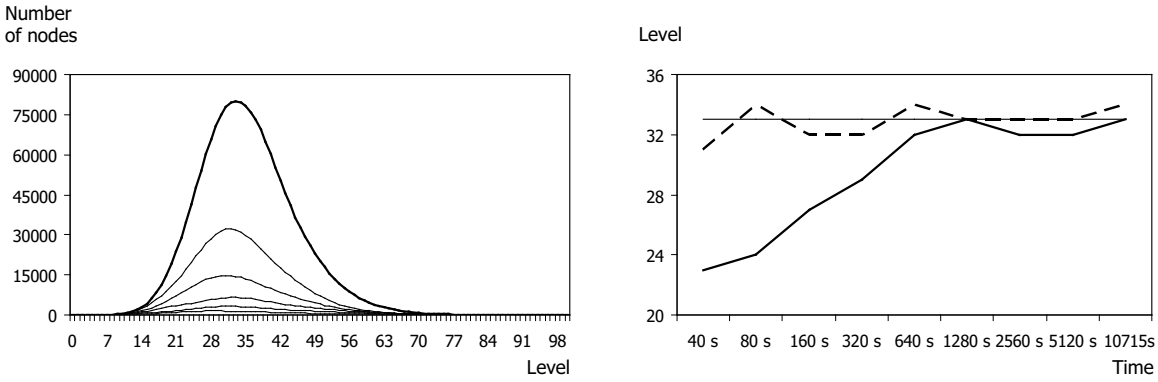

Figure 4.3: Evolution of the Profile, the Waist, and the Average Waist. Plotted Profiles After 320, 640, 1280, 2560, 5120, and 10715 Seconds of Solution Time. Problem: `rout`.

both values are close. Early on in the solution process, the average waist is often a better estimate of the waist of the complete tree. Additionally, compared to b_t, the average waist $\bar{b}_t$ shows less variation during the solution process. Therefore, the average waist can be used to improve the prediction when there is a large variation in the waist. The average waist of the partial tree of problem `rout` is plotted with a dashed line in the right graph of Figure 4.3.

Repeating the experiments on the same 99 test instances by using the average waist instead of the waist, the number of correct predictions increased from 78 to 85. This is a reduction of the incorrect predictions by one third. Further research in this direction would be worthwhile.

Effect of the Bound from the Best Feasible Solution Found

If a good upper bound (for a minimization problem) is not found early in the solution process, the method can produce very poor and erratic estimates. Even after a good upper bound is found, the method could still produce poor estimates if it is biased by early "deep dives." Depending on the diving strategy, the first few dives into the tree can be very deep compared to what they would have been with a good a priori upper bound. This biases the estimate of the depth. One simple idea to avoid this problem is to eliminate (post factum) any node whose lower bound exceeds the current upper bound, even though such nodes are technically part of the search tree. This method will produce the same estimate that would

have been produced had the bound been known a priori and should reduce the bias.

Experiments with Branch and Cut

We repeated the experiment from Section 4.3.3 with the default settings of the CPLEX branch-and-cut algorithm. The results are reported in Table B.7.

Six instances, `lseu`, `mod008`, `modglob`, `pp08a`, `rgn`, and `stein27`, were solved during Phase I. They are not present in the table. Eighteen of the remaining instances were solved to optimality. For eleven of them the prediction is correct. There are six cases, `arki001`, `blend2`, `mas74`, `misc07`, `pk1`, and `pp08aCUTS`, with considerable error. The solution of four instances was interrupted after exceeding the time or space limit, and this was predicted correctly for three of them.

Compared with Table B.3, there is a deterioration in the prediction in some cases but there are some improvements too (like with problem rout). Overall, the quality of the estimate is not significantly different.

4.4 Conclusion

We have shown empirically that the branch-and-bound solution time of an MILP solver can be roughly estimated in the early stages of the solution process. We proposed a procedure for this estimation based on parameters of a small subtree. Our experiments showed that, in a relatively short time, we can obtain sufficient information to predict the total running time with an error within a factor of five. This procedure can easily be built into an MILP solver. It is fast and does not interfere with the branch-and-bound algorithm.

It might be worth exploring γ-sequence models that are contingent on particular types of integer-programming problems. One might also be able to obtain relevant information on the whole tree profile using other parameters of the tree. Our attempts to use the amount of pruning in the subtree were fruitless but more research in this direction would be interesting.

Chapter 5

Conclusion

In this thesis, we presented ideas for modifications in the branch-and-cut algorithm, and tested their performance empirically.

We proposed a practical algorithm for branching on general split disjunctions. It incorporates a heuristic measure of disjunction quality, based on the relation between branching disjunctions and intersection cuts. In our algorithm, we considered the class of disjunctions defining the mixed integer Gomory cuts at an optimal basis of the LP relaxation. One of the reasons for our choice was the aim at efficiency and speed. An important extension of our work would be the development of efficient algorithms for other classes of disjunctions. Theoretically, branching on split disjunctions is more powerful than branching on single variables. Efficient implementation of branching on general disjunctions is the key factor for its success.

Computational experiments showed that our procedure for branching on split disjunctions is more efficient than branching on variables. In addition, a procedure combining the strengths of both methods showed much promise. However, all of our tests used strong branching for branching-object selection. Efficient branching rules, such as pseudocost branching and its variations, are widely used but cannot be implemented for general split disjunctions. Developing an algorithm combining pseudocost branching on variables and strong branching on split disjunctions would be an important advance toward practical application of our procedure.

We developed an algorithm for selecting a set of good cuts out of a large pool of generated cuts. In this study, we made several important observations. We witnessed the effect of

flattening: adding all of the generated cuts causes the angles between the active constraints after reoptimization to decrease, which leads to poorer quality cuts in the future rounds. We showed that this effect can be avoided by clever cut selection. Other consequences of cut selection are decreased reoptimization time and prevented numerical problems. This is achieved without sacrificing the amount of gap closed. A key feature of our cut selection procedure as that it evaluates the quality of cuts as a group. The quality measure that we use is the angle between cuts. It would be beneficial to find other criteria for evaluating groups of cuts.

We saw indications that branching may be affected by flattening as well. In addition, improving the amount of gap closed by cuts does not always lead to faster solution of the problem. More research is needed for better understanding the interaction between cuts and branching. As a future step, a challenging research direction would be to develop a decision rule for branching versus cutting. Whether to add one more round of cuts or proceed to branching is a very important practical question that awaits its answer.

The running time of branch-and-bound algorithms on a particular instance is difficult to predict. We showed that a rough estimate can, nevertheless, be obtained early in the solution process. Our estimation procedure is based on modeling the profile of the complete branching tree. The model relies on the observation that the profile exhibits a bell-like shape. An improvement in the prediction quality of the procedure is contingent on better understanding the reasons for this bell-like shape and the parameters that influence it. In modeling the tree profile, we use three parameters but they can not capture the shape completely. We envision the evolution of our idea to a sampling procedure which applies a directed search during the initial, prediction phase. The search would aim at collecting the information needed for a good prediction. This search should be used to collect other information useful for branch and cut as well, e.g. to initialize pseudocosts.

Appendix A

Experimental results on Chapter 2: Branching on general disjunctions

Table A.1: Comparison of the gap closed at the root node by branching on a single variable and on a MIG disjunction.

	Simple disjunction		MIG disjunction	
Instance	Absolute gap closed	Relative gap closed [%]	Absolute gap closed	Relative gap closed [%]
10teams	0	0	4	57.14
a1c1s1	42.45	-	9.87	-
aflow30a	6.09	3.49	11.22	6.42
aflow40b	6.77	4.17	4.49	2.76
air01	53	100	53	100
air02	170	100	170	100
air03	1295.75	100	1295.75	100
air04	43.67	7.26	53.73	8.93
air05	72.54	14.61	31.11	6.27
air06	21.97	67.32	25.78	78.99
arki001	42.2	-	0	-
atlanta-ip	0	-	0.18	-
bell3a	3174.32	20.03	1857.8	11.72
bell3b	316638.07	82.89	13855.95	3.63
bell4	360675.73	64.79	122948.47	22.08
bell5	298008.96	83.25	298008.96	83.25
blend2	0.01	0.84	0.13	19.72
bm23	0.91	6.8	0.29	2.13
cap6000	69.83	44.38	85.59	54.4
dano3mip	0.09	-	0.02	-
danoint	0.03	1	0.03	1.01
dcmulti	921.07	21.9	647.89	15.4
ds	0.12	-	0.13	-
egout	5.77	1.38	16	3.82
fast0507	0.06	3.03	0.05	2.43
fiber	2648.8	1.06	40055.4	16.03
fixnet3	84.62	0.75	365.02	3.24
fixnet4	29.43	0.63	95.5	2.04
fixnet6	22.44	0.81	23.79	0.85
flugpl	689.44	2.01	1874.28	5.46
gen	9.23	5.03	65.21	35.57

continued on next page

Table A.1: *continued*

Instance	Simple disjunction		MIG disjunction	
	Absolute gap closed	Relative gap closed [%]	Absolute gap closed	Relative gap closed [%]
gesa2	3636.77	1.2	8160.05	2.69
gesa2_o	3636.77	1.2	8160.05	2.69
gesa3	5348.27	3.4	10051.01	6.39
gesa3_o	5348.27	3.4	10051.01	6.39
glass4	0	-	0	-
gt2	359.47	4.66	4816.57	62.51
harp2	4413.48	0.97	21196.43	4.67
khb05250	1670000	15.15	670982	6.09
l152lav	5.83	8.89	9.79	14.91
liu	214	-	214	-
lp4l	9.11	37.16	20.44	83.41
lseu	12	4.21	98.06	34.37
manna81	0.5	0.38	0.5	0.38
markshare1	0	0	0	0
markshare2	0	0	0	0
mas74	40.84	3.1	75.5	5.73
mas76	20.59	1.85	62.27	5.6
misc01	0	0	13.24	2.61
misc02	0	0	25	3.68
misc03	0	0	92.5	6.38
misc04	5.65	54.92	5.65	54.92
misc05	1.2	2.24	14.4	26.87
misc06	2.08	22.73	2.08	22.73
misc07	0	0	0	0
mkc	0	0	0	0
mod008	0.24	1.48	0.02	0.14
mod010	3.59	22.53	13.52	84.92
mod011	415328.41	5.49	709329.48	9.38
mod013	1.36	5.45	6.13	24.59
modglob	8969.42	2.9	8969.42	2.9
momentum1	10.41	-	3199.99	-
momentum2	0.16	-	690.39	-

continued on next page

Table A.1: *continued*

Instance	Simple disjunction		MIG disjunction	
	Absolute gap closed	Relative gap closed [%]	Absolute gap closed	Relative gap closed [%]
momentum3	0.26	-	4.9	-
msc98-ip	0	-	27360	-
mzzv11	0	0	8.49	0.69
mzzv42z	48.88	4.51	0	0
net12	3.65	1.85	1.87	0.95
nsrand-ipx	0	0	110	4.74
nw04	22	3.99	259.33	47.04
opt1217	0	0	0.27	6.78
p0033	29.93	5.27	205.76	36.2
p0040	10.49	4.55	160.91	69.83
p0201	0	0	310	41.89
p0282	181.18	0.22	32843.91	40.28
p0291	1031.33	29.31	1031.33	29.31
p0548	12.68	0.15	12.68	0.15
p2756	0	0	0	0
pipex	0.07	0.48	4.08	28.1
pk1	0	0	0	0
pp08a	142.14	3.09	80	1.74
pp08aCUTS	68.03	3.64	81.44	4.36
protfold	0.36	-	0.64	-
qiu	0	0	0	0
qnet1	42.99	2.45	213.47	12.16
qnet1_o	474.01	12.05	474.01	12.05
rd-rplusc-21	0	-	0	-
rgn	0	0	0.8	2.4
roll3000	0.94	-	0	-
rout	2.34	2.44	2.34	2.44
sample2	15	11.72	15	11.72
sentoy	7.44	11.06	10.79	16.04
set1al	15.64	0.33	37.93	0.8
set1ch	95.87	0.43	722.86	3.21
set1cl	15.64	0.33	37.93	0.79

continued on next page

Table A.1: *continued*

Instance	Simple disjunction		MIG disjunction	
	Absolute gap closed	Relative gap closed [%]	Absolute gap closed	Relative gap closed [%]
seymour	0.36	1.86	0.49	2.57
sp97ar	54195.09	-	249175.81	-
stein9	0	0	0	0
stein15	0	0	0	0
stein27	0	0	0	0
stein45	0	0	0	0
swath	0.13	-	4.12	-
t1717	146.93	-	86.64	-
timtab1	24290	3.3	24290	3.3
timtab2	23443	-	5462	-
tr12-30	367.03	0.32	300	0.26
vpm1	0.25	5.39	0.66	14.48
vpm2	0.06	1.68	0.1	2.7

Table A.2: Comparison of the gap closed and the number of active nodes after five levels of branching.

	Simple disjunctions			MIG disjunctions		
Instance	Nodes at level 5	Absolute gap closed	Relative gap closed [%]	Nodes at level 5	Absolute gap closed	Relative gap closed [%]
10teams	32	0.0	0.0	10	4.0	57.1
a1c1s1	32	336.3	-	32	57.3	-
aflow30a	28	26.1	15.0	4	32.8	18.7
aflow40b	19	12.3	7.6	16	32.8	20.2
air04	23	212.4	35.3	31	230.1	38.3
air05	32	246.9	49.7	32	192.9	38.9
arki001	32	148.6	-	32	0.0	-
bell3a	14	9645.2	60.8	8	9128.8	57.6
bell3b	2	338929.2	88.7	8	258394.2	67.6
bell4	5	494769.1	88.9	17	416779.6	74.9
bell5	4	299455.8	83.6	1	307175.6	85.8
blend2	16	0.2	28.2	27	0.2	36.5
bm23	32	3.5	25.7	16	6.2	46.5
cap6000	32	106.8	67.9	32	106.8	67.9
danoint	32	0.0	1.7	32	0.0	1.7
dcmulti	31	2048.0	48.7	15	821.9	19.5
egout	1	29.3	7.0	1	61.0	14.6
fast0507	12	0.3	17.3	6	0.2	11.5
fiber	32	6974.0	2.8	31	72697.2	29.1
fixnet3	32	370.5	3.3	32	1545.9	13.7
fixnet4	32	145.0	3.1	32	415.0	8.9
fixnet6	32	60.5	2.2	32	95.5	3.4
flugpl	14	5097.6	14.9	11	6615.9	19.3
gen	6	33.0	18.0	1	117.4	64.0
gesa2	32	14251.6	4.7	21	39091.6	12.9
gesa2_o	32	12133.5	4.0	19	39091.6	12.9
gesa3	32	32480.0	20.6	13	43313.2	27.5
gesa3_o	32	32480.0	20.6	13	43313.2	27.5
glass4	12	0.0	-	10	0.0	-
gt2	32	6182.9	80.2	12	4940.9	64.1
harp2	32	24867.7	5.5	2	32823.5	7.2

continued on next page

Table A.2: *continued*

Instance	Simple disjunctions			MIG disjunctions		
	Nodes at level 5	Absolute gap closed	Relative gap closed [%]	Nodes at level 5	Absolute gap closed	Relative gap closed [%]
khb05250	32	5183898.0	47.0	32	5097557.0	46.3
l152lav	28	30.7	46.7	28	30.7	46.7
liu	32	214.0	-	32	214.0	-
lseu	28	25.0	8.7	5	160.9	56.4
manna81	32	1.0	0.8	1	2.5	1.9
markshare1	32	0.0	0.0	32	0.0	0.0
markshare2	32	0.0	0.0	32	0.0	0.0
mas74	32	147.2	11.2	32	159.4	12.1
mas76	32	99.6	9.0	32	111.9	10.1
misc01	7	36.5	7.2	6	40.0	7.9
misc03	8	1450.0	100.0	16	67.5	4.7
misc05	20	17.4	32.5	26	16.5	30.8
misc07	24	15.3	1.1	18	11.3	0.8
mkc	28	1.7	3.5	1	0.0	0.0
mod008	25	16.1	100.0	25	16.1	100.0
mod011	32	1230186.8	16.3	32	2746460.5	36.3
mod013	32	6.9	27.5	16	13.6	54.4
modglob	32	25488.7	8.2	32	32258.2	10.4
net12	28	9.4	4.8	15	15.0	7.6
nsrand-ipx	32	0.0	0.0	28	415.0	17.9
nw04	8	138.0	25.0	8	138.0	25.0
opt1217	32	0.0	0.0	32	1.0	25.4
p0033	12	179.4	31.6	1	388.9	68.4
p0201	30	120.0	16.2	31	480.0	64.9
p0282	26	934.3	1.1	27	77433.0	95.0
p0548	1	64.3	0.8	1	123.8	1.5
p2756	7	0.0	0.0	1	1.0	0.2
pipex	32	1.5	10.1	23	5.6	38.5
pk1	32	0.0	0.0	32	0.0	0.0
pp08a	32	673.9	14.6	32	380.0	8.3
pp08aCUTS	32	240.9	12.9	32	374.7	20.0
qiu	31	272.7	34.1	29	290.7	36.4

continued on next page

Table A.2: *continued*

Instance	Simple disjunctions			MIG disjunctions		
	Nodes at level 5	Absolute gap closed	Relative gap closed [%]	Nodes at level 5	Absolute gap closed	Relative gap closed [%]
qnet1	32	665.7	37.9	28	683.8	38.9
qnet1_o	24	1905.4	48.4	24	1884.7	47.9
rd-rplusc-21	1	0.0	-	1	0.0	-
rgn	24	33.4	100.0	32	8.8	26.3
roll3000	20	3.4	-	3	226.4	-
rout	32	14.9	15.6	23	20.7	21.7
sample2	26	86.7	67.7	32	63.0	49.2
sentoy	32	27.0	40.2	32	25.2	37.4
set1al	32	76.0	1.6	1	173.5	3.7
set1ch	32	502.1	2.2	2	2740.4	12.2
set1cl	32	76.0	1.6	1	173.5	3.6
seymour	32	1.0	5.2	1	1.9	10.2
sp97ar	32	424694.5	-	2	827703.8	-
stein15	29	2.0	100.0	2	2.0	100.0
stein27	32	0.0	0.0	6	0.0	0.0
stein45	32	0.0	0.0	5	0.0	0.0
swath	24	1.4	-	1	13.7	-
timtab1	24	70664.0	9.6	13	65088.0	8.8
timtab2	21	52373.0	-	11	62952.0	-
tr12-30	32	1833.2	1.6	32	1063.7	0.9
vpm1	32	0.5	10.5	5	1.9	40.9
vpm2	32	0.5	12.7	6	0.5	13.7

Table A.3: Branching on simple disjunctions, MIG disjunctions, and combined in a cut-and-branch framework with time limit of two hours.

Instance	Relative gap closed by cuts [%]	Simple disjunctions			MIG disjunctions			Combined		
		Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]
10teams	0	7200	4550	28.6	4526.8	1255	100	240.5	99	100
a1c1s1	2518.2 *	7200	13437	3516.8 *	7200	14710	3033.4 *	7200	12329	2863.7 *
aflow30a	5.8	7200	60379	85	7200	33485	82.7	7200	31054	86.1
aflow40b	0	7200	11795	43	7200	5180	50.4	7200	5679	48.5
air04	0	7200	631	96.4	7200	432	82.9	7200	410	82.5
air05	0	1029.9	274	100	5081.2	1089	100	1981.1	413	100
bell3a	58.6	129	28754	100	123.8	28643	100	126.2	28643	100
bell3b	79.3	7200	260898	96.6	105.3	8811	100	1687.6	60447	100
bell4	45	7200	268780	96.6	51.2	3490	100	7200	288096	99.6
bell5	89.1	7200	1517347	99.5	9.9	2055	100	4964.6	400625	100
blend2	0	41.7	1915	100	11.2	519	100	22	747	100.1
bm23	28.7	0.8	210	100	0.4	123	100	0.6	126	100
cap6000	0	7200	34572	99.9	6918.6	34719	100	6931.4	34718	100
danoint	0	7200	7614	23.8	7200	4800	16.9	7200	6335	22.5
dcmulti	10.4	273.2	5475	100	4.9	113	100	8.2	187	100
egout	51.8	3.7	422	100	2.9	223	100	1.6	120	100
fast0507	0	7200	98	25.4	7200	72	15.7	7200	55	18.4
fiber	35.4	7200	122832	66.5	18	203	100	11.3	172	100
fixnet3	2.1	7200	173254	20.8	7200	160591	49.7	7200	164895	46.4
fixnet4	2.5	7200	168267	19.6	7200	142080	32.6	7200	147986	32.8
fixnet6	1.1	7200	154719	13.7	7200	130749	22.9	7200	137584	19.4

continued on next page

Table A.3: *continued*

Instance	Relative gap closed by cuts [%]	Simple disjunctions			MIG disjunctions			Combined		
		Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]
flugpl	14.5	2.6	1050	100	0.1	38	100	0.1	34	100
gen	50.4	20.1	246	100	17.3	119	100	13	107	100
gesa2	83.4	7200	52912	92.6	270.2	1867	100	413.4	2904	100
gesa2_o	82.8	7200	56691	93.8	475.8	3268	100	128	949	100
gesa3	48.1	127.5	843	100	19.3	140	100	19.2	142	100
gesa3_o	57.4	579.5	4246	100	22.8	168	100	15.7	121	100
glass4	0 *	7200	274535	2200 *	7200	251180	1863.1 *	7200	260218	1967.3 *
gt2	67.2	7200	624148	83.9	1.1	120	100	0.5	41	100
harp2	0	7200	80904	38.5	7200	63573	39.7	7200	56013	43.8
khb05250	0	39.6	1175	100	35.7	1071	100	39.6	1172	100
l152lav	0	32.6	196	100	34.9	217	100	12.9	87	100
liu	214 *	7200	39865	214 *	7200	37436	214 *	7200	44918	214 *
lseu	47.8	79.2	12297	100	14.3	2546	100	8.7	1483	100
manna81	100	0.4	1	100	0.3	1	100	0.3	1	100
markshare1	0	7200	2150514	0	7200	1349880	0	7200	461430	0
markshare2	0	7200	1633304	0	7200	1071762	0	7200	358188	0
mas74	0	7200	510347	75.8	7200	521762	78.3	7200	356091	75.1
mas76	0	4078	347759	100	3146.7	263187	100	5640.9	253624	100
misc01	6.5	3.5	207	100	2.8	180	100	3.2	163	100
misc03	6.4	7.3	279	100	17.4	707	100	10.5	369	100
misc05	23.7	8.2	207	100	4.8	141	100	2.3	75	100
misc07	0	634.4	9768	100	926.9	16161	100	528.2	8864	100

continued on next page

Table A.3: *continued*

Instance	Relative gap closed by cuts [%]	Simple disjunctions			MIG disjunctions			Combined		
		Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]
mkc	31.1	7200	3314	31.1	7200	201	60.3	7200	786	55.7
mod008	0	78.4	12498	100	80.3	12213	100	7.8	1131	100
mod011	0	7200	3701	65.3	7200	2424	83.2	7200	3272	72.8
mod013	35	0.8	140	100	1.2	167	100	0.6	72	100
modglob	18.9	7200	193834	59.1	7200	164859	77.4	7200	186402	76
net12	7.2	7200	64	7.8	7200	5	12.3	7200	8	8.3
nsrand-ipx	8.2	7200	7426	15.2	7200	6264	55.5	7200	6740	50.1
nw04	0	3584.8	474	100	6795.9	699	100	48.3	6	100
opt1217	0	7200	181560	0	7200	128001	25.4	7200	153013	43.3
p0033	12.6	0.7	349	100	0.1	22	100	0.1	40	100
p0201	45.6	4.5	126	100	3.8	131	100	3	82	100
p0282	5.4	7200	271116	12.5	3.6	140	100	6.1	186	100
p0548	4.5	7200	263866	86.1	7200	232534	99	7200	249489	97.4
p2756	0	7200	71574	0	7200	80020	2.4	7200	83052	2.4
pipex	46	2	422	100	1.6	273	100	1.4	219	100
pk1	0	3483.6	229371	100	4675.1	297376	100	5320.1	262141	100
pp08a	86.5	7200	161212	98.8	1068.4	18022	100	1297.8	24570	100
pp08aCUTS	49.7	7200	149337	93.6	4797.4	83338	100	7165	134868	100
qiu	0	3394.4	9786	100	4194.6	10741	100	2838.3	7185	100
qnet1	0	138.3	1219	100	309.1	1140	100	126.9	574	100
qnet1_o	20.6	35.4	214	100	204.5	826	100	234.1	828	100
rgn	26.3	9.2	1383	100	8.3	1367	100	6.7	1125	100

continued on next page

Table A.3: *continued*

Instance	Relative gap closed by cuts [%]	Simple disjunctions			MIG disjunctions			Combined		
		Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]	Solution time [s]	Nodes processed	Rel. gap closed [%]
rout	0	7200	105270	53.5	2775.3	35932	100	655.3	10152	100
sample2	26.7	0.5	100	100	0.7	117	100	0.8	107	100
sentoy	0	1.2	188	100	1.4	211	100	1.2	180	100
set1al	96.2	0.4	19	100	0.5	19	100	0.6	19	100
set1ch	85.3	7200	49392	87.7	7200	37705	91.7	7200	44872	91.2
set1cl	94.3	0.2	11	100	0.3	11	100	0.3	11	100
seymour	12.1	7200	589	35.8	7200	267	34.5	7200	417	35.7
sp97ar	234315 *	7200	1385	1397498 *	7200	808	3461368 *	7200	701	3344473 *
stein15	0	1.3	91	100	1.4	110	100	1.4	86	100
stein27	0	126.4	2771	100	180	3983	100	143.3	2889	100
stein45	0	6216.9	44577	100	7200	42340	93.8	4824.9	30572	100
swath	0 *	7200	22623	28.9 *	7200	16514	55.8 *	7200	15119	55.2 *
tr12-30	73.5	7200	41266	74.5	7200	34603	79.1	7200	33826	78.9
vpm1	39.7	7200	334508	96.3	1071.5	27509	100	1064.7	29306	100
vpm2	41.7	7200	197220	81.9	7200	121610	94	7200	144094	94.6

* Absolute gap closed reported.

Appendix B

Experimental results on Chapter 4: Early estimates of the size of branch-and-bound trees

Table B.1: Tree-Size Estimation by Knuth's Method with Sample Size 1000

Problem	Predicted number of nodes	Actual number of nodes	Ratio	Estimation Time [seconds]	Estimation Nodes visited
air05	754	1221	0.61	31200	5690
arki001 *	262	1124575	2.3E-4	2838	4695
bell3a	14110	18512	0.76	121	13736
bell4	1.3E+06	13654	95	78	9261
bell5	1362	301146	0.004	61	4451
blend2	365	5750	0.063	201	6683
gesa2_o	6011	1136	5.2	809	11187
harp2 *	22775	786616	0.028	925	9563
lseu	4738	1614	2.9	41	9578
markshare1 *	1.2E+10	52464676	228	94	30855
markshare2 *	5.0E+12	45059758	1.1E+5	129	38200
mas74	8.7E+06	10159496	0.85	182	20251
mas76	1.0E+06	637057	1.6	144	16655
misc07	21697	111784	0.19	1289	10580
mod008	1373	2161	0.63	92	13377
mod011	25945	21788	1.2	12180	9683
modglob	2.0E+06	304	6480	209	11804
noswot	4.1E+07	5614491	7.3	112	13266
pk1	163508	337940	0.48	155	14529
pp08a	9.5E+09	933	1.0E+7	204	28249
pp08aCUTS	4.6E+06	1687	2744	360	19624
qiu	40163	9358	4.3	6081	11556
rgn	1261	3025	0.42	73	9665
rout	34871	1797969	0.019	1643	11024
seymour *	4.4E+15	54713	8.0E+10	128520	42574
stein27	8429	3706	2.3	200	12573
stein45	167466	68093	2.5	889	15508
vpm2	337552	25255	13	216	14180

* Solution procedure interrupted. Column "Actual number of nodes" contains the number of nodes at time of interruption.

Table B.2: Linear Model Estimation

Problem	Actual number of nodes	Linear model estimation	Ratio
air05	1221	3017	2.47
arki001 *	1124575	1910566720	1698.92
bell3a	18512	3217	0.17
bell4	13654	20639	1.51
bell5	301146	47977699	159.32
blend2	5750	13401	2.33
gesa2_o	1136	6325	5.57
harp2 *	786616	261277428	332.15
lseu	1614	3023	1.87
markshare1 *	52464676	20885162	0.40
markshare2 *	45059758	925206188	20.53
mas74	10159496	6257366	0.62
mas76	637057	703329	1.10
misc07	111784	6657013	59.55
mod008	2161	39000	18.05
mod011	21788	41535	1.91
modglob	304	472	1.55
noswot	5614491	3336543	0.59
pk1	337940	1532758	4.54
pp08a	933	1171	1.26
pp08aCUTS	1687	3472	2.06
qiu	9358	60458	6.46
rgn	3025	19660	6.50
rout	1797969	27316720	15.19
seymour *	54713	72319299	1321.79
stein27	3706	3996	1.08
stein45	68093	63255	0.93
vpm2	25255	31299	1.24

* Instance not solved to completion.

Table B.3: Predictions by Our Estimation Procedure

Problem	Predicted number of nodes	Actual number of nodes	Ratio	Phase I time [s]	Predicted time range	Actual solution time	
air05	2043	1221	1.7	291	5 m – 40 m	5 m	
arki001 *	26808093	1124575	23	61	> 10 h	> 10 h	
bell3a	3217	18512	0.17	5	5 s – 15 s	20.7 s	†
bell4	13980	13654	1	5	5 s – 1 m	12.2 s	
bell5	987629	301146	3.2	5	2 m – 1 h	4.2 m	
blend2	59796	5750	10	7	40 s – 15 m	15.5 s	†
gesa2_o	6325	1136	5.6	8	10 s – 5 m	8.4 s	†
harp2 *	3644539	786616	4.6	45	> 1 h	> 1.2 h	
markshare1 *	13480899	52464676	0.26	5	> 30 m	> 10 h	
markshare2 *	925206188	45059758	21	5	> 10 h	> 10 h	
mas74	118255	10159496	0.01	5	30 s – 15 m	3.9 h	†
mas76	69963	637057	0.11	5	15 s – 10 m	10.0 m	
misc07	1437454696	111784	13000	14	> 10 h	10.7 m	†
mod011	9576	21788	0.44	429	15 m – 6 h	2.0 h	
noswot	74866	5614491	0.01	5	25 s – 10 m	2.2 h	†
pk1	31044	337940	0.09	5	10 s – 5 m	9.4 m	†
pp08a	786	933	0.84	5	5 s – 25 s	5.6 s	
pp08aCUTS	2355	1687	1.4	5	5 s – 1 m	9.1 s	
qiu	2085	9358	0.22	65	1 m – 20 m	9.5 m	
rout	8758	1797969	0.01	21	20 s – 10 m	2.9 h	†
seymour *	182659036	54713	3300	2171	> 10 h	> 10 h	
stein45	43822	68093	0.64	5	30 s – 10 m	2.2 m	
vpm2	9176	25255	0.36	5	5 s – 1 m	40.6 s	

* Instance not solved to completion.

† Incorrect time prediction.

Table B.4: More Test Results

Instance	Phase I time [s]	Predicted time range	Solution time	
Multidimensional Knapsack Problems				
mknapcb1-1	5	5 s – 2 m	1.3 m	
mknapcb1-11	5	5 s – 20 s	7.7 s	
mknapcb1-21	5	5 s – 50 s	5.3 s	
mknapcb2-1	5	> 1 h	8.2 m	†
mknapcb2-11	5	> 10 h	20.1 m	†
mknapcb2-21	5	> 10 h	15.3 m	†
mknapcb3-1 *	15	> 10 h	> 2.6 h	
mknapcb3-11 *	15	> 10 h	> 2 h	
mknapcb3-21	16	> 10 h	6.4 m	†
mknapcb4-1	5	15 s – 10 m	13.4 m	†
mknapcb4-11	5	35 s – 15 m	5.9 m	
mknapcb4-21	5	5 s – 50 s	17.9 s	
mknapcb5-1 *	7	15 m – 6 h	> 2.5 h	†
mknapcb5-11 *	7	> 2 h	> 2.3 h	
mknapcb5-21 *	7	10 m – 4 h	> 2.6 h	†
mknapcb6-1 *	22	> 10 h	> 2.8 h	
mknapcb6-11 *	25	25 m – 10 h	> 2.9 h	†
mknapcb6-21 *	23	> 10 h	> 2.1 h	
mknapcb7-1	5	5 m – 1 h	23.2 m	
mknapcb7-11	5	10 m – 3 h	3.3 h	†
mknapcb7-21	5	5 m – 1 h	7.9 m	
mknapcb8-1 *	18	> 10 h	> 10 h	
mknapcb8-11 *	14	> 1 h	> 10 h	
mknapcb8-21 *	15	> 4 h	> 8.8 h	
mknapcb9-1 *	74	> 10 h	> 5.6 h	
mknapcb9-11 *	63	> 10 h	> 4.8 h	
mknapcb9-21 *	50	> 8 h	> 4.3 h	

* Instance not solved to completion.

† Incorrect time prediction.

Table B.5: More Test Results

Instance	Phase I time [s]	Predicted time range	Solution time	
Set-Covering Problems				
scpnre1	581	10 m – 3 h	58.3 m	
scpnre2	654	> 30 m	7.6 h	
scpnre3	707	15 m – 7 h	1.1 h	
scpnre4	188	5 m – 1 h	37.1 m	
scpnre5	87	2 m – 40 m	21.7 m	
scpnrf1	1311	20 m – 7 h	29.7 m	
scpnrf2	860	15 m – 3 h	19.9 m	
scpnrf3	625	10 m – 3 h	13.6 m	
scpnrf4	635	15 m – 6 h	1.4 h	
scpnrf5	407	15 m – 7 h	2.1 h	
scpnrg1 *	1625	> 1 h	> 10 h	
scpnrg2 *	558	15 m – 5 h	> 10 h	†
scpnrg3 *	846	> 4 h	> 10 h	
scpnrg4 *	1166	> 10 h	> 10 h	
scpnrh1 *	3429	> 10 h	> 10 h	
scpclr10 *	106	> 10 h	> 10 h	
scpclr11 *	2203	> 10 h	> 10 h	
scpclr12 *	12501	> 10 h	> 10 h	
MILP Benchmarks				
bc *	1315	> 10 h	> 10 h	
binkar10_1 *	21	5 m – 3 h	> 6.4 h	†
eilD76	469	> 10 h	30.1 m	†
mas284	10	10 s – 5 m	1.2 m	
mkc1	159	> 10 h	4.3 h	†
prod1	5	1 m – 25 m	11.2 m	
ran14x18_1 *	24	> 10 h	> 10 h	

* Instance not solved to completion.

† Incorrect time prediction.

Table B.6: More Test Results

Instance	Phase I time [s]	Predicted time range	Solution time
Bin-Packing Problems			
t60_00 *	174	> 10 h	> 10 h
t60_05 *	93	> 10 h	> 10 h
t60_10 *	238	> 10 h	> 10 h
t60_15 *	432	> 10 h	> 10 h
t120_00 *	1487	> 10 h	> 10 h
t120_05 *	7329	> 10 h	> 10 h
t120_10 *	8125	> 10 h	> 10 h
t120_15 *	1681	> 10 h	> 10 h
u120_00 *	663	> 10 h	> 10 h
u120_05 *	1010	> 10 h	> 10 h
u120_10 *	1229	> 10 h	> 10 h
u120_15 *	951	> 10 h	> 10 h
Capacitated Facility-Location Problems			
capa1 *	242	> 10 h	> 2.8 h
capa2 *	241	> 10 h	> 2.8 h
capa3 *	238	> 10 h	> 2.8 h
capa4 *	237	> 10 h	> 2.8 h
capb1 *	209	> 10 h	> 2.5 h
capb2 *	209	> 10 h	> 2.5 h
capb3 *	209	> 10 h	> 2.5 h
capb4 *	209	> 10 h	> 2.5 h
capc1 *	200	> 10 h	> 2.5 h
capc2 *	201	> 10 h	> 2.5 h
capc3 *	222	> 10 h	> 2.6 h
capc4 *	219	> 10 h	> 2.6 h

* Instance not solved to completion.

† Incorrect time prediction.

Table B.7: Predictions by Our Estimation Procedure for a Branch-and-Cut Algorithm

Problem	Predicted number of nodes	Actual number of nodes	Ratio	Phase I time [s]	Predicted time range	Actual solution time	
air05	1576	720	2.2	169	1 m – 35 m	2.8 m	
arki001	396870541	347635	1100	74	> 10 h	3.2 h	†
bell3a	4818	19001	0.25	5	5 s – 35 s	29.6 s	
bell4	11403	19599	0.58	5	5 s – 1 m	20.6 s	
bell5	302597	582887	0.52	5	1 m – 25 m	9.6 m	
blend2	1434224	3973	360	8	20 m – 7 h	13 s	†
gesa2_o	1511	670	2.3	5	5 s – 1 m	6 s	
harp2 *	168082	320884	0.52	41	5 m – 3 h	> 54 m	†
markshare1 *	54252828	54563771	1	5	> 2 h	> 10 h	
markshare2 *	789981661	41189904	19	5	> 10 h	> 8.4 h	
mas74	56129	6518567	0.01	5	20 s – 10 m	3.1 h	†
mas76	35890	559528	0.06	5	10 s – 4 m	10.5 m	†
misc07	1.07e+11	79952	1.3e+6	17	> 10 h	9.5 m	†
mod011	4190	9635	0.43	985	15 m – 6 h	2.1 h	
noswot	83316	8308673	0.01	5	30 s – 10 m	4.8 h	†
pk1	9251	540710	0.02	5	5 s – 2 m	18.1 m	†
pp08aCUTS	1715	1910	0.9	6	5 s – 1 m	14 s	
qiu	3355	10098	0.33	118	2 m – 45 m	14.5 m	
rout	93362	99119	0.94	34	5 m – 2 h	42.8 m	
seymour *	119836809	57420	2087	2504	> 10 h	> 10 h	
stein45	26468	60717	0.44	5	20 s – 5 m	2 m	
vpm2	2418	4328	0.56	5	5 s – 30 s	10 s	

* Instance not solved to completion.

† Incorrect time prediction.

Bibliography

[1] K. Aardal, R. E. Bixby, C. A. J. Hurkens, A. K. Lenstra, and J. W. Smeltink. Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances. *Informs Journal on Computing*, 12(3):192–202, 2000.

[2] K. Aardal, C. A. J. Hurkens, and A. K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, 2000.

[3] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operation Research Letters*, 33:42–54, 2005.

[4] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. Technical Report 05-28, Zuse Institute Berlin, Takustr. 7, Berlin, 2005.

[5] K. Andersen, G. Cornuéjols, and Y. Li. Reduce-and-split cuts: Improving the performance of mixed integer Gomory cuts. *to appear in Management Science*, 2005.

[6] G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch and cut framework: a computational study with $\{0, 1/2\}$-cuts. To appear in INFORMS Journal on Computing, 2006.

[7] K. Anstreicher, N. Brixius, J-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91:563–588, 2002.

[8] D. Applegate, R. Bixby, V. Chvatal, and B. Cook. Finding cuts in the tsp. Technical Report 95-05, Certer for Discrete Mathematics & Theoretical Computer Science, March 1995.

[9] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. The traveling salesman problem. Manuscript, 1995.

[10] E. Balas. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13:517–546, 1965.

[11] E. Balas. Intersection cuts - a new type of cutting planes for integer programming. *Operations Research*, 19:19–39, 1971.

[12] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[13] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996.

[14] E. Balas, S. Ceria, G. Cornuejols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, July 1996.

[15] E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *OR 69: Proc. Fifth Int. Conf. Oper. Res.*, pages 447–454, London, 1970. Tavistock Publications.

[16] J. E. Beasley. An algorithm for solving large capacitated warehouse location problems. *European Journal of Operational Research*, 33:314–325, 1988.

[17] J. E. Beasley. A lagrangian heuristic for set-covering problems. *Naval Research Logistics*, 37:151–164, 1990.

[18] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[19] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer programming. *Mathematical Programming*, 1:76–94, 1971.

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

[21] A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard. Solving large-scale QAP problems in parallel with the search library ZRAM. *Journal of Parallel and Distributed Computing*, 50:157–169, 1998.

[22] P. G. Chen. Heuristic sampling: a method for predicting the performance of tree search programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[23] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[24] W. Cook, R. Fukasawa, and M. Goycoolea. Choosing the best cuts. MIP 2006 Workshop, June 2006.

[25] W. Cook, R. Kannan, and A Schrijver. Chvátal closures for mixed integer programs. *Mathematical Programming*, 47:155–174, 1990.

[26] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. In R. E. Bixby, E. A. Boyd, and R. Z. Rios-Mercado, editors, *Integer Programming and Combinatorial Optimization, 6th International IPCO Conference*, Lecture notes in Computer Science 1412, pages 284–293. Springer-Verlag, Berlin, 1998.

[27] ILOG CPLEX. Reference manual. URL: ftp://www.ilog.com/products/cplex, 2003.

[28] R. J. Dakin. A tree search algorithm for mixed programming problems. *Computer Journal*, 1965.

[29] G. Danzig, D. Fulkerson, and S. Johnson. Solution of a large scale travelling salesman problem. *Operations Research*, 2:393–410, 1954.

[30] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. Technical report, CRIF Industrial Management and Automation, CP 106 - P4, 50 av. F.D.Roosevelt, B-1050 Brussels, Belgium, 1994.

[31] M. C. Ferris, G. Pataki, and S. Schmieta. Solving the seymour problem. *Optima*, (66):1 – 7, 2001.

[32] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2003.

[33] Computational Infrastructure for Operations Research (COIN-OR). URL: http://www.coin-or.org/index.html, 2006.

[34] R. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[35] R. Gomory. An algorithm for the mixed integer problem. Technical Report RM-2597, The Rand Corporation, 1960.

[36] R. Gomory. *Combinatorial Analysis*, R. E. Bellman and M. Hall, Jr., eds., chapter Solving linear programming problems in integers, pages 211–216. American Mathematical Society, 1960.

[37] R. Gomory. *Recent Advances in Mathematical Programming*, R. Graves and P. Wolfs, eds., chapter An algorithm for integer solutions to linear programs, pages 269–302. McGraw-Hill, 1963.

[38] M. Grötschel, L. Lovász, and A. Schrijver. *Progress in Combinatorial Optimizarion*, chapter Geometric methods in combinatorial optimization, pages 167–183. Academic Press, Toronto, 1984.

[39] Jr. H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[40] D. E. Knuth. Estimating the efficiency of backtracking programs. *Mathematics of Computing*, 29:121–136, 1975.

[41] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 1960.

[42] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[43] J. Linderoth and M. Savelsbergh. A computational study of search strategies for mixed integer programming. Report LEC-97-12, Georgia Institute of Technology, 1997.

[44] J. D. C. Little, K. G. Murthy, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 21:972–989, 1963.

[45] L. Lobjois and M. Lemaitre. Branch-and-bound algorithm selection by performance prediction. Technical report, American Association for Artificial Intelligence, 1998.

[46] L. Lovász and H. E. Scarf. The generalized basis reduction algorithm. *Mathematics of Operations Research*, 17:751–764, 1992.

[47] H. Marchand and L.A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49:363–371, 2001.

[48] F. Margot. BAC: A BCP based branch-and-cut example.
URL: http://www.coin-or.org/Papers/bac.ps, 2003.

[49] Argonne National Laboratory Mathematics and Computer Science Division. NEOS guide test problems: Mip benchmarks. URL: ftp://ftp.mcs.anl.gov/neos/mip-bench, 2003.

[50] S. Mehrotra and Z. Li. On generalized branching methods for mixed integer programming. Technical report, Northwestern University, Evanston, Illinois 60208, 2004.

[51] H. Mittelmann. Benchmarks for optimization software.
URL: ftp://plato.la.asu.edu/pub/milp, 2003.

[52] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[53] Website of Zuse Institute Berlin. Miplib 2.0.
URL: http://miplib.zib.de/miplib3/miplib_prev.html, 2003.

[54] J. Owen and S. Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear program. *Computational Optimization and Applications*, 20:159–170, 2001.

[55] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric travelling salesman problems. *SIAM Review*, 33:60–100, 1991.

[56] P. W. Purdom. Tree size by partial backtracking. *SIAM Journal on Computing*, 7:481–491, 1978.

[57] T. K. Ralphs and L. Ladányi. COIN/BCP user's manual.
URL: http://www.coin-or.org/Presentations/bcp-man.pdf, 2001.

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