TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

An Extended Local Branching Framework and its Application to the Multidimensional Knapsack Problem

ausgeführt am

Institut für Computergrafik und Algorithmen

der Technischen Universität Wien

unter der Anleitung von

a.o. Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl

Univ.-Ass. Dipl.-Ing. Jakob Puchinger

durch

Daniel Lichtenberger

Matr. Nr. 9825754

Märzstrasse 80/12

1150 Wien

Datum

Unterschrift

Abstract

This thesis deals with local branching, a local search algorithm applied on top of a Branch and Cut algorithm for mixed integer programming problems. Local branching defines custom sized neighborhoods around given feasible solutions and solves them partially or completely before exploring the rest of the search space. Its goal is to improve the heuristic behavior of a given exact integer programming solver, i.e. to focus on finding good solutions early in the computation.

Local branching is implemented as an extension to the open source Branch and Cut solver COIN/BCP. The framework's main goal is to provide a generic implementation of local branching for integer programming problems. IP problems are optimization problems where some or all variables are integer values and must satisfy one or more (linear) constraints. Several extensions to the standard local branching algorithm were added to the framework. Pseudo-concurrent exploration of multiple local trees, aborting local trees and a variable fixing heuristic allow the user to implement sophisticated search metaheuristics that adjust the local branching parameters adaptively during the computation. A major design goal was to provide a clean encapsulation of the local branching algorithm to facilitate embedding of the framework in other, higher-level search algorithms, for example in evolutionary algorithms.

As an example application, a solver for the multidimensional knapsack problem is implemented. A custom local branching metaheuristic imposes node limits on local subtrees and adaptively tightens the search space by fixing variables and reducing the size of the neighborhood. Test results show that local branching can offer significant advantages to standard Branch and Cut algorithms and eventually proves optimality in shorter time. Especially for large, complex test instances exploring the local neighborhood of a good feasible solution often yields better short-term results than the unguided standard Branch and Cut algorithm. Improving the solutions found early in the computation also helps to remove additional parts of the search tree, potentially leading to better solutions in longer runs.

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit Local Branching, einem lokalen Suchalgorithmus, der auf einem Branch and Cut Algorithmus für ganzzahlige Optimierungsprobleme aufsetzt. Local Branching definiert beliebig große Nachbarschaften um gegebene gültige Lösungen und löst diese teilweise oder komplett, bevor der Rest des Lösungsraums durchsucht wird. Das Ziel ist eine Verbesserung des heuristischen Verhaltens des gegebenen Solvers für ganzzahlige Optimierungsprobleme, d.h. sich auf das möglichst frühe Finden guter Lösungen zu konzentrieren.

Local Branching ist als Erweiterung des Open Source Branch and Cut Solvers COIN/BCP implementiert. Das Hauptziel des Frameworks ist eine generische Implementierung von Local Branching für ganzzahlige Optimierungsprobleme, also Probleme, bei denen alle oder einige Variablen ganzzahlig sein müssen, und zusätzlich eine oder mehrere (lineare) Bedingungen in Form von Ungleichungen erfüllen müssen. Es wurden mehrere Erweiterungen zum Framework hinzugefügt: die pseudo-parallele Abarbeitung mehrerer lokaler Suchbäume, das vorzeitige Terminieren lokaler Suchbäume sowie eine unabhängige Variablen-Fixing-Heuristik. Durch diese Erweiterungen können die Parameter für Local Branching im Laufe der Berechnung beliebig verändert werden. Ein wesentliches Ziel beim Entwurf des Frameworks war eine klare Kapselung des Local Branching Algorithmus, um die Einbettung in andere, höhere Suchalgorithmen zu ermöglichen, etwa in evolutionäre Algorithmen.

Als Beispielapplikation wurde ein Solver für das mehrdimensionale Rucksackproblem implementiert. Eine eigene Local Branching Metaheuristik beschränkt die Größe lokaler Bäume durch Knotenlimits und kann den Suchraum durch Anwendung der Variablen-Fixing-Heuristik weiter einschränken. Die Testergebnisse zeigen signifikante Vorteile für Local Branching im Vergleich zum normalen Branch and Cut Algorithmus. Vor allem für große, komplexe Testinstanzen liefert die Suche in lokalen Bäumen oft bessere Resultate am Anfang der Berechnung. Dadurch wird auch die Zeit zum Finden (und Beweisen) der optimalen Lösung potentiell verringert, da dadurch früher zusätzliche Teile des Suchbaums weggeschnitten werden können.

Contents

Chapter 1

Introduction

Integer programming problems (IPs) are optimization problems that restrict some or all variables to integer values. In contrast to linear programming problems (LPs) without integrality constraints, IPs are NP-hard. Much research has gone into effective search algorithms for integer programs, leading to exact algorithms like Branch and Bound [25], cutting plane algorithms [30], and a large variety of heuristical algorithms that trade optimality for quickly getting "good enough" solutions.

This thesis considers the modification of standard Branch and Cut to follow ideas from local search based heuristics, the so-called *local branching* [13]. *Branch and Bound* is a generic algorithm for solving integer programming problems by partitioning the search space into smaller subproblems (branching), calculating bounds on the best solution that can be found in a subproblem (bounding), and removing those subproblems that are proven to contain only solutions inferior to the best known solution (pruning). The bounding operation is commonly executed by solving the LP relaxation (i.e. the IP problem without the integral constraints). *Branch and Cut* tries to delay the branching operation by adding constraints (cuts) that are violated by the current LP result, leading to a reduction of the search tree size.

Local branching defines subproblems through additional local branching cuts that isolate a neighborhood of a certain size around a given feasible solution. By exploring this smaller subproblem before the rest of the search tree, the intention is to improve good feasible solutions before continuing Branch and Cut in a standard way. Several extensions have been added to local branching: pseudo-concurrent tree exploration, the possibility to abort local trees, and a variable fixing heuristic have been added. Due to its general design, local branching can be used with any IP solver.

A large part of integer programming is concerned with *combinatorial problems*. These include for example the subset sum equality problem, various graph theory problems, and the well-known family of knapsack problems. In this thesis, the *multidimensional knapsack problem* is used to demonstrate the use and the benefits of local branching. Although all types of knapsack problems are NP-hard, some problems can be efficiently solved by enumerative techniques like dynamic programming. For others, like the multidimensional knapsack problem, no such methods are known. These problems supply well suited testcases for fully fledged Branch and Cut solvers, and are often too complex to be solved to optimality in reasonable time.

1.1 Thesis Overview

In chapter 2, an overview of integer programming problems, cutting plane techniques and Branch and Bound algorithms is given to summarize the building blocks of Branch and Cut. Chapter 3 provides an introduction to local branching as proposed by Fischetti and Lodi [13]. Chapter 4 introduces the framework implemented for this thesis, including extensions to the local branching algorithm, and describes the overall design of the interface to the framework. In chapter 5, an overview of the open source COIN/BCP framework used for implementing local branching is given. Chapter 6 contains the implementation details of the local branching framework. An overview of knapsack problems in general and multidimensional knapsack problems in particular is given in chapter 7. The implementation of a sample local branching application for the multidimensional knapsack problem is described in chapter 8. Test results exploring the benefits and drawbacks of local branching based on the sample application are given in chapter 9. Chapter 10 summarizes the results and provides a brief outlook on possible future work. In appendix A, the patches necessary for the COIN/BCP source code are described. Appendix B provides a brief overview of the test scripts used for analyzing the local branching test runs.

Chapter 2

Branch and Cut

Branch and Cut is an exact algorithm for solving integer programming problems. It combines cutting plane methods with Branch and Bound. The following introduction is based on Lee and Mitchell's Branch and Bound tutorial [26], Mitchell's introduction to Branch and Cut [29], the COIN/BCP User's Manual by Ralphs and Ladanyi [36], and the book on integer programming by Laurence Wolsey [42].

2.1 Integer Programming Problems

An *integer programming problem* (IP) is an optimization problem in which some or all variables are restricted to integer values. A given *objective function* has to be maximized or minimized in a solution space constrained by inequalities. A *mixed integer programming problem* (MIP) contains both integer and continuous variables, a *pure integer programming problem* restricts all variables to be integer. *Mixed* or *pure 0-1 integer programming problems* restrict all integer variables to be 0 or 1, thus they are also called *binary integer programming problems*. In this thesis we will concentrate on *linear 0-1 integer programming problems* where all variables are binary and all terms of the objective function and constraints are linear. The objective function should be maximized. A linear 0-1 IP can then be stated as:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \in \{0,1\}^n \end{array} \tag{2.1}$$

with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. We can define the solution space S of a problem as

$$S = \{x \in \{0,1\}^n : Ax \leq b\}. \tag{2.2}$$

2.1.1 Convex Hull of an Integer Program

In algebraic topology, $Ax \leq b$ defines a *convex polyhedron* which contains all feasible solutions of the integer program. H. Weyl proved in 1935 that a convex polyhedron can be defined as the intersection of a finite number of half-spaces or as the *convex hull* combined with the conical hull of a finite number of vectors or points. If the problem is formulated in rational

numbers, Weyl's theorem implies the existence of a finite system of linear inequalities whose solution set coincides with the convex hull of our solution space S, also written as $conv(S)$. This directly leads to *cutting plane algorithms* for solving integer programming problems that will be described in section 2.3.

2.1.2 Relaxations

A key concept of integer programming is that of problem relaxation. A *relaxation* of an optimization problem as stated in equation (2.1) is an optimization problem

$$\max\{c_R^T x : x \in S_R\}, \tag{2.3}$$

where $S \subseteq S_R$ and $c^T x \leq c_R^T x$ for all $x \in S$. The relaxed solution space is a superset of the problem solution space, and the relaxed objective function is equal to or greater than the original function for all feasible solutions of the given problem.

A common relaxation for linear integer programming problems is the *linear programming relaxation* (*LP relaxation*). The integer constraints on all variables are removed and the problem can then be solved with linear programming methods. The most common algorithm for solving linear programs is the *simplex method* invented by George Bernard Dantzig in 1947. There are instances where the simplex method requires an exponential number of steps, but those problems seem to be highly unlikely in practical applications where the simplex method achieves very good performance.

Khachian's *ellipsoid algorithm* [22] proved that linear programming was polynomial in 1979. Karmarkar's interior-point method [20] was both a practical and theoretical improvement over the ellipsoid algorithm.

2.2 Branch and Bound

Branch and Bound is a class of exact algorithms for various optimization problems, especially integer programming problems and combinatorial optimization problems (COP). It partitions the solution space into smaller subproblems that can be solved independently (*branching*). *Bounding* discards subproblems that cannot contain the optimal solution, thus decreasing the size of the solution space. Branch and Bound was first proposed by Land and Doig in 1960 [25] for solving integer programs.

Given a maximization problem as described in equations (2.1) and (2.2), a Branch and Bound algorithm iteratively partitions the solution space S, for example by branching on binary variables - fixing one of them to 0 in one branch and to 1 in the other branch. For each subproblem an *upper bound* on the objective value is calculated. The upper bound is guaranteed to be equal to or greater than the optimal solution for this subproblem. When a feasible solution (i.e., no fractional variables remaining) is found, all subproblems whose upper bounds are lower than this solution's objective value can be discarded. The best known feasible solution represents a *lower bound* for all subproblems, and only subproblems with an upper bound greater than the global lower bound have to be considered. Discarding a subproblem is called *fathoming* or *pruning*. Upper bounds for a subproblem can be obtained by relaxing the subproblem, thus they are often obtained by optimizing the subproblem's LP relaxation.

Figure 2.1 summarizes the above steps using a pseudo-code notation. The sequence of subproblems created by branching can be organized as a rooted directed graph. The original

1. Initialize list of all subproblems $C = \{S\}$
2. Generate a feasible solution and store it in $\hat{s}$. It is not necessary to generate a feasible solution (e.g. by heuristics), but it can help to reduce the search tree size. When no initial solution is provided, the objective value for $\hat{s}$ is set to $-\infty$.
3. Repeat while $C \neq \emptyset$:
   (a) Take a subproblem S' from C
   (b) *Relax* S' and solve the relaxed problem
   (c) Decide to branch or prune as explained in figure 2.2.
4. Return $\hat{s}$

Figure 2.1: Branch and Bound pseudo-code

problem is the root node with edges going to each of its children. This graph is called the *search tree* and its *nodes* represent all generated *subproblems*.

2.3 Cutting Plane Algorithms

As in section 2.1, we will consider a binary integer programming problem, its mathematical formulation is stated in equations (2.1) and (2.2). The fundamental concept used for cutting plane algorithms is that of a *valid inequality*. An inequality

$$\pi x \leq \pi_0 \tag{2.4}$$

is valid if $\pi x \leq \pi_0$ for all $x \in S$, where S contains all feasible solutions of the IP.

The basic idea of *cutting planes* is to describe the convex hull $conv(S)$ of the original problem by adding valid inequalities to the LP relaxation until the LP solution becomes feasible for the original problem.

Mitchell [30] outlines the following structure of a cutting plane algorithm:

1. Solve the LP relaxation using linear programming methods such as the simplex algorithm.
2. If the LP solution is feasible for the integral problem, return the optimal solution.
3. Otherwise add cutting planes to the relaxation that separate the LP solution from the convex hull of feasible integral points.
4. Go to first step.

Cutting planes can be generated with or without problem specific knowledge. One method of obtaining cutting planes is by combining inequalities from the current LP relaxation. This is known as *integer rounding*, and the resulting cutting planes are called *Chvátal-Gomory cutting planes* [15, 16, 7]. The following example is taken from [30]. Consider the integer programming problem

Depending on the solution of the relaxed problem, do one of the following:

1. No solution was found, the relaxed problem is infeasible. Then there is also no feasible solution in S', thus the subproblem is pruned.
2. The optimal solution is not better than $\hat{s}$. The subproblem can be pruned because its upper bound is lower than the global lower bound.
3. The optimal solution is better than $\hat{s}$ and it is in S' (the integer constraints are satisfied). Replace $\hat{s}$ with the new optimal solution. The subtree can be pruned because no better solution can be found.
4. The optimal solution is better than $\hat{s}$ but it is not in S' (at least one integer constraint is violated). In this case S' is partitioned into n smaller subproblems such that: $\bigcup_{i=1}^{n} S'_i = S'$. Each of these *children* of S' is added to C. This is the common case and is usually called *branching*.

Figure 2.2: Deciding to branch or prune

$$\begin{aligned} \text{minimize } & -2x_1 - x_2 \\ & x_1 + 2x_2 \leq 7 \\ & 2x_1 - x_2 \leq 3 \\ & x_1, x_2 \in \mathbb{N}_0. \end{aligned} \tag{2.5}$$

A cutting plane is obtained by a weighted combination of inequalities, e.g.

$$0.2(x_1 + 2x_2 \leq 7) + 0.4(2x_1 - x_2 \leq 3) \tag{2.6}$$

gives the valid inequality

$$x_1 \leq 2.6. \tag{2.7}$$

This inequality is valid for all LP relaxations, but in a feasible solution, the left hand side must be an integer value. This leads to the inequality

$$x_1 \leq 2. \tag{2.8}$$

Gomory's cutting plane algorithm will find the optimal solution by iterating the steps as described above. However, the number of steps to describe the convex hull (called the *Chvátal rank*) is typically very high, leading to very slow convergence [10, 11].

It can be enhanced using techniques like adding many Chvátal-Gomory cuts at once, as shown in [1] and [5]. Another approach is to combine cutting plane methods with Branch and Bound, which leads to a method called *Branch and Cut*.

1. Initialize candidate list $C = \{S\}$
2. Generate a feasible solution and store it in $\hat{s}$
3. Repeat while $C \neq \emptyset$:
   (a) Take a subproblem S' from C
   (b) Relax S' and solve the relaxed problem, store LP result in $\tilde{s}$
   (c) Repeat:
      (1) Try to add cuts to the relaxed problem that are violated by $\tilde{s}$
      (2) Exit loop when no new cuts were generated in step 1
      (3) Solve relaxed problem again, store LP result in $\tilde{s}$. Note that the objective value of $\tilde{s}$ is monotonically decreasing since the added cuts render infeasible the previous LP results.
   (d) Depending on $\tilde{s}$, decide to branch or prune the node as shown in figure 2.2.
4. Return $\hat{s}$

Figure 2.3: Branch and Cut pseudo-code

2.4 Branch and Cut

Branch and Cut methods use Branch and Bound to partition the solution space into smaller subproblems, but also utilize *cutting plane methods* to tighten the relaxation and thus to reduce the size of the search tree. Branch and Cut was first proposed by Padberg and Rinaldi [31] as a framework for solving traveling salesman problems.

The purpose of cutting planes or *cuts* is to reduce the upper bound derived from the optimal solution of the LP relaxation. A smaller upper bound makes pruning the subproblem more likely, thus reducing the search tree size. When the algorithm failed to generate new cuts that are violated by the current LP solution, the subproblem is *branched* as in Branch and Bound.

As cut generation can be very expensive, it is common to generate cuts only for some nodes in the search tree. For example, it might be reasonable to generate cuts for every eighth node or for all nodes at a depth of a multiple of eight. The *cut-and-branch* variant adds cutting planes only at the root node. A pseudo-code formulation of Branch and Cut is given in figure 2.3.

Chapter 3

Local Branching

While there exist sophisticated solvers for integer programming problems, for many hard problems the optimal solution is often hard to find within a reasonable time. Therefore, it becomes increasingly important to find reasonably good solutions early in the computation process.

Local Branching is a local search meta-heuristic for integer programs proposed by Fischetti and Lodi in 2002 [13] that is entirely embedded in a Branch and Cut framework. Its goal is to improve the heuristic behavior of a given MIP solver without losing optimality, that is, to find good feasible solutions as soon as possible while still being able to find the global optimum and prove its optimality.

Local Branching works by partitioning the search tree through so-called *local branching cuts*. Since those local cuts are just specific constraints for integer programming problems, they can be expressed like normal IP constraints using any generic MIP solver.

3.1 Soft vs. Hard Variable Fixing

A common technique for IP heuristics is hard variable fixing. For example, a heuristic might use a LP solver to compute a continuous optimal solution, heuristically fix some variables to integer values (e.g. by rounding the variable with the least fractional value), and then repeating these steps for the resulting subproblem without the fixed variables. This way, relatively good (but probably not optimal) solutions may be found in reasonable time even for hard problems.

The major downside of this approach is that it may be nearly impossible during the early stages to decide which variable should be fixed. This inevitably leads to bad fixings which may not be detected until much later, requiring some kind of backtracking to undo bad choices.

To overcome this limitation of variable fixing, Fischetti and Lodi proposed *soft variable fixing*. It does not select a single variable for fixing, but only specifies that a certain percentage of all variables of a given *feasible solution* should be fixed. This approach is best illustrated using the binary integer programming problem described in section 2.1. Supposing there is a feasible solution and 90% of its nonzero variables should be fixed to 1, Fischetti and Lodi add a *soft fixing constraint*

$$\sum_{j=1}^{n} \bar{x}_j x_j \geq \lceil 0.9 \sum_{j=1}^{n} \bar{x}_j \rceil \tag{3.1}$$

to the current formulation. $\bar{x}_j$ represents the feasible solution around which a local neigh-

borhood is isolated, i.e. in any feasible solution x_j only 10% of those variables set to 1 in $\bar{x}_j$ may be flipped to 0. The idea is that fixing 90% of the variables helps the solver to find good solutions as effectively as when fixing a large number of variables, but with a much larger degree of freedom.

3.2 A Basic Local Branching Framework

Given a binary integer programming problem as stated in section 2.1 and a feasible solution $\bar{x}$, the *binary support* $\bar{S}$ is defined as $\bar{S} := \{j \in [1, n] : \bar{x}_j = 1\}$, i.e. the indices of those variables that are set to 1. A soft fixing constraint in terms of the previous section can then be formulated as

$$\Delta(x, \bar{x}) := \sum_{j \in \bar{S}} (1 - x_j) + \sum_{j \notin \bar{S}} x_j \leq k. \tag{3.2}$$

Fischetti and Lodi call this a *local branching constraint* that counts all binary variables that flipped their value from zero to one or from one to zero compared to $\bar{x}$. $\Delta(x, \bar{x})$ actually represents the *Hamming distance* between x and $\bar{x}$, thus the constraint is also called *Hamming distance constraint*. When the cardinality of $\bar{S}$ is fixed, this constraint is equivalent to

$$\Delta(x, \bar{x}) := \sum_{j \in \bar{S}} (1 - x_j) \leq k' (= \frac{k}{2}), \tag{3.3}$$

because for every variable x_j with $j \in \bar{S}$ that flips from one to zero another variable must flip from zero to one. This definition is consistent with the classical *k'-opt* neighborhood for the Traveling Salesman Problem, where at most k' edges may be replaced.

A local branching constraint partitions the search tree in two disjunct branches

$$\Delta(x, \bar{x}) \leq k \text{ (\textit{local branch})} \quad \text{and} \quad \Delta(x, \bar{x}) > k \text{ (\textit{normal branch})}. \tag{3.4}$$

The local branch is completely solved before continuing with the normal branch. When a new global optimum $\bar{x}^2$ was found in the local branch, local branching can continue with the new solution by adding a new constraint to the remaining "normal" branch, again partitioning the search tree in two disjunct branches

$$\begin{aligned} &\Delta(x, \bar{x}) > k, \quad \Delta(x, \bar{x}^2) \leq k \quad \text{(\textit{local branch})}, \\ &\Delta(x, \bar{x}) > k, \quad \Delta(x, \bar{x}^2) > k \quad \text{(\textit{normal branch})}. \end{aligned} \tag{3.5}$$

This scheme works as long as the local branching trees yield new global optima and is illustrated in figure 3.2. The numbers indicate the sequence in which the subproblems are generated and processed. The actual optimization problems are solved by a generic MIP solver.

The size of the local subtrees at positions 2, 4 and 6 depend on the choice of the k parameter. Small values of k define a relatively small neighborhood that is easier to solve, but may not contain solutions that are significantly better than the current one. Larger values of k offer higher degrees of freedom during the tree search, but drastically increase the size of the local branching trees.

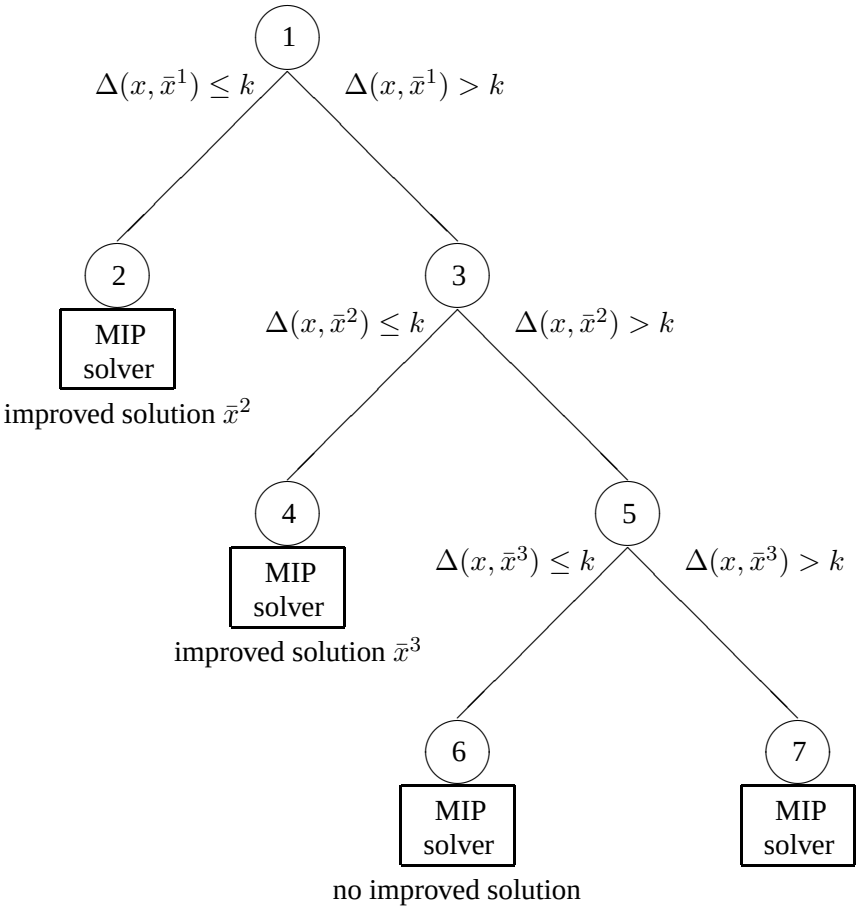

Figure 3.1: Local Branching

3.3 Local Branching Extensions

Fischetti and Lodi [13] proposed several extensions to the standard local branching algorithm described in the previous section.

- *Imposing a time limit on local branching trees* allows to use large values of k without having to explore a local tree completely. When time runs out and a better solution was found in the local tree, the algorithm creates a new local tree at the original root node using the new solution. However, since the previous local tree was not explored completely, this may lead to a duplication of effort as the optimal solution might still be in the first local tree, and its search space can therefore not be excluded. If the time limit is reached without finding a new better solution, k is decreased to speed up the exploration of the local tree.

- *Diversification* may be used when a local tree did not improve the best known solution. Fischetti and Lodi suggest to start with a soft diversification by enlarging the neighborhood, e.g. by $\lceil k/2 \rceil$. When no better solution is found in this larger local tree, they apply a strong diversification by taking another (worse) solution and restarting local branching with this solution.

- *Embedding local branching* in heuristic frameworks like *Tabu Search*, *Variable Neighborhood Search*, *Simulated Annealing* or *Evolutionary Algorithms* can be easily done, since local branching naturally defines a custom sized neighborhood around a given solution. Additional constraints imposed by the heuristic framework can be described as linear cuts which makes them easy to join with local branching constraints.

- *Working with infeasible solutions* is necessary for problems where finding an initial feasible solution is hard, e.g. for hard set partitioning models. In order to use an infeasible solution as initial solution for local branching, one may define additional slack variables for some of the constraints while penalizing them in the objective function.

- *General integer variables* require a new definition of the local branching constraint. Some general integer problems still have a relevant subset of 0-1 variables that can be used for local branching. In case there are no relevant binary variables, introducing weights leads to a viable local branching constraint. In a MIP model that involves the bounds $l_j \leq x_j \leq u_j$ for $j = 1 \ldots n$, a local branching constraint can be defined as

$$\Delta(x, \bar{x}) := \sum_{j:\bar{x}_j = l_j} \mu_j (x_j - l_j) + \sum_{j:\bar{x}_j = u_j} \mu_j (u_j - x_j) + \sum_{j: l_j < \bar{x}_j < u_j} \mu_j (x_j^+ + x_j^-) \leq k. \tag{3.6}$$

  The weights are defined as $\mu_j = 1/(u_j - l_j)$, while x_j^+ and x_j^- define additional slack variables that satisfy the equation

$$x_j = \bar{x}_j + x_j^+ - x_j^-, \qquad x_j^+ \geq 0, x_j^- \geq 0.$$

Of course, there are other possibilities to improve the standard local branching algorithm not proposed by Fischetti and Lodi. The following enhancements have been integrated in our local branching framework as described in chapter 4.

- *Fixing variables* allows to tighten the neighborhood when the original local tree was too large to be explored completely. Variables that share the same value in the incumbent solution and in the solution of the LP relaxation are less likely to change in the global optimum. By fixing some of those variables in the local tree and adding a corresponding cut to the remaining tree local branching can avoid calculating parts of the tree that will probably yield no better results. This approach was proposed by Danna et al. [8] and is known as *RINS* (Relaxation Induced Neighborhood Search).
- *Concurrent exploration of different local trees* provides diversification by creating several local trees from different feasible solutions and exploring them simultaneously.

Chapter 4

An Advanced Local Branching Framework

In this chapter a generic framework for local branching is described. Standard local branching is implemented as described in chapter 3, and several extensions are introduced to improve its performance. The main goal of this framework is to provide a local search algorithm for higher-level metaheuristics, for example evolutionary algorithms. These metaheuristics can use local branching for exploring the neighborhood of certain solutions, and use the generated feasible solutions as input for their own improvement algorithms. The actual implementation of the framework will be described in chapter 6.

4.1 Basic Functionality

A sequential version of the standard local branching algorithm provides the basis for the framework. It is capable of using local branching to completely solve a problem without further user intervention. The main phases of the local branching algorithm are:

1. *Generate an initial solution for the first tree*.
2. *Initialize the first local tree* using the previously generated solution.
3. Repeat:
   (a) *Completely solve the local tree*.
   (b) When the local search terminates:
   - *A better feasible solution was found* in this local tree: create a new local tree using the improved solution from this local tree as initial solution.
   - *No better feasible solution was found*: abort local branching.
4. *Solve the rest of the search tree*.
5. *Return optimal solution*.

The initial solution can be created by a custom heuristic, or it is derived from the optimum of the root node's LP relaxation (e.g. by rounding or truncating the LP solution). The default implementation uses local branching constraints as described in section 3, i.e. Hamming

distance constraints defining a neighborhood around the solution according to the distance parameter k. Other constraints for branching might be implemented by the user as well.

4.1.1 Limitations

The standard local branching algorithm works well for some instances, but has several limitations:

1. Depending on the number of variables and the value of k, the Hamming distance constraint possibly defines a very large neighborhood. Given a binary IP with n variables, a feasible solution has $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ neighbors with a Hamming distance of k (the local tree includes all neighbors with a Hamming distance not larger than k, so the actual search tree is even larger).

2. Depending on the specific problem, there might be more than one reasonable initial solution for local branching. When a given heuristic returns several promising solutions that would create (partially) disjunct local trees (i.e. their Hamming distance is greater than k), only one neighborhood can be explored.

3. While a local branching constraint defines a neighborhood around a feasible solution, it provides no further guidance for exploring this neighborhood besides the standard branch and cut strategies (e.g. best bound first search). Other local search heuristics might help to tighten the search tree.

Preliminary testing with multidimensional knapsack problems confirmed these shortcomings. The test instances contain integer programming problems with 100 to 500 variables and 5 to 30 constraints. Detailed results will be discussed in chapter 9.

The larger test instances ($n \geq 250$, $d \geq 10$) contain too many variables for using k values larger than approximately 5. This allows at most five variables to flip their values, and likely prohibits significant improvements to the initial solution of a local tree. For any value $k > 5$ even the local tree defines a subproblem that is often too complex to be solved completely within the given time limit. Additionally, the first initial solution is either derived by a first fit heuristic or by rounding the first LP solution, and both are unlikely to be in a small neighborhood of the optimal solution.

4.2 Extending the Basic Algorithm

In order to address the shortcomings described in the previous section, three extensions have been added to the standard local branching algorithm. The first one eliminates the restriction of sequential execution by allowing to create new local trees before the previous one(s) are finished. On a related issue, the second extension allows to abort local trees before they are completely solved. The last extension tries to reduce subproblem complexity by fixing variables that are less likely to change in the optimal result than others.

4.2.1 Using Multiple Local Trees

The first major extension to standard local branching is the support for *pseudo-concurrent exploration* of several local trees. It removes the burden of relying on one local tree at a time,

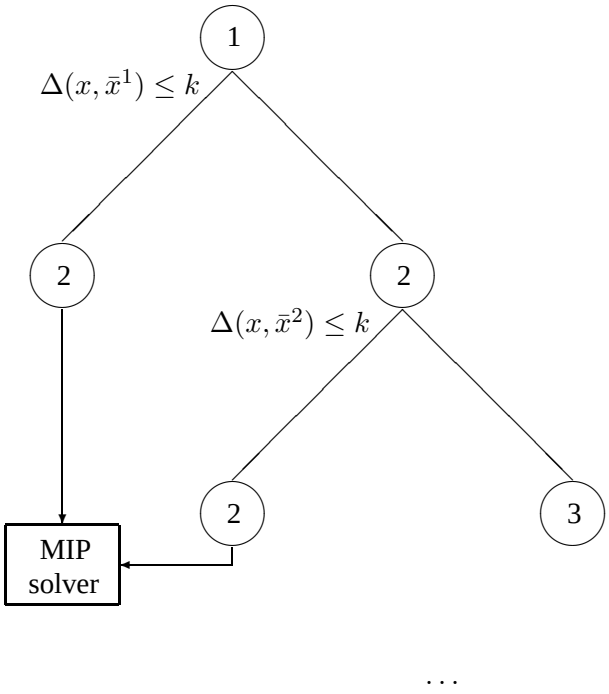

Figure 4.1: Pseudo-concurrent local tree exploration using a single MIP solver instance

probably leading to a more robust search tree exploration less likely to be caught in local optima of the objective function.

The framework provides a method to spawn any number of local branching trees simply by providing an initial solution. However, these trees are not parallelized in the sense of a separate process dedicated to a single local tree. Instead, there is still a single pool for subproblems where all local tree nodes are stored. Depending on the chosen tree search strategy, some trees will probably get more time than others with less promising nodes. For example, when a best bound first tree search is performed, a local tree whose nodes have relatively poor upper bounds will get less time than a tree with more promising nodes.

Unfortunately this kind of pseudo-concurrent exploration likely leads to a duplication of effort in some cases since it is not known a priori which trees will actually be completely solved, and thus, inverse local branching constraints cannot be considered. When a local tree is prematurely terminated, no information about this local tree (except feasible solutions found so far) can be further utilized: the neighborhood defined by this tree cannot be excluded from future local trees because it still may contain the optimal solution.

The framework achieves parallel exploration by a simple modification to the standard local branching algorithm: before a local tree is not completely solved, the inverse local branching constraint for the rest of the search tree remains inactive. When a local tree is prematurely terminated, the inverse constraint is removed from all future local trees. Figure 4.2.1 shows the modified version of the original algorithm previously shown in figure 3.2. All parts of the search tree labeled with the same number are parallelized, thus the inverse constraints on the right hand side are missing. The initial solutions $\bar{x}^1$ and $\bar{x}^2$ are not necessarily related; they are supplied by the higher-level metaheuristic.

4.2.2 Aborting Local Trees

Aborting local trees greatly enhances control over local branching. It becomes possible to impose time or node limits on local trees, abort stagnating local trees in favor of more promising ones, or simply restart local branching with new solutions (possibly generated outside the framework.)

When a local tree is aborted, the inverse local branching constraint (defining the search tree outside the local tree) must be removed. This also applies to the variable fixing constraint introduced in the next section. Besides that, the only issue is to find some criteria for premature local tree termination. Specifying a *time limit* is rather complex due to the distributed design of COIN/BCP. Different machines may have different performance ratings, and it would require non-trivial extensions to COIN to track the time spent for each subproblem, potentially across several LP processes, cut generators, and variable generators.

The local branching framework offers extensive information about the number of nodes processed per local tree instead. Using these facilities, it is easy to impose a node limit on a local tree, or to specify a maximum number of nodes that can be processed in a single local tree before an improvement is found. The downside of this approach is that the number of nodes that can be processed in a given CPU timeslice depends on the complexity of the IP problem. Thus node limits may have to be set heuristically, for example by some constant value multiplied with a weighted sum of the number of variables and the number of constraints.

4.2.3 Tightening the Search Tree by Variable Fixing

The variable fixing extension to local branching is based on *Relaxation Induced Neighborhood Search* (RINS) proposed by Danna et al. [8]. The underlying assumption is that variables having the same integer value in the incumbent solution and in the LP relaxation are likely to be set to their optimal value. By fixing some of those variables to their current value, the local search focuses attention on the fractional variables.

Compared with reducing search tree size by reducing the value of k, variable fixing gives more freedom to the exploration of the more promising fractional variables while ignoring the allegedly less promising integral variables of the current LP optimum.

Choosing the best variables to be fixed is a problem by itself. The framework picks a random selection from the set of all variables having the same integer values in the integral and the LP solution. The number of fixed variables is given relative to the total number of variables in the (sub-)problem. In the following, let F_1 denote the indices of variables fixed to one, and F_0 the indices of variables fixed to zero.

While fixing variables in the local tree can be done directly in the MIP solver, the inverse constraint is a bit more complicated: a node becomes feasible when at least one of the fixed variables changed its value. Ignoring the local branching constraint, this can be achieved through a new row cut of the form

$$\sum_{j \in F_1} (1 - x_j) + \sum_{j \in F_0} x_j > 0. \tag{4.1}$$

When using Hamming distance cuts for local branching both constraints can be combined to a single cut. A solution is feasible outside the local tree when either the Hamming distance is greater than k, or at least one variable flipped. In other words, when a variable flips (and

the above inequality becomes valid), the Hamming distance constraint should be considered irrelevant. The following row cut achieves these goals:

$$\Delta(x, \bar{x}) > k - k \left[\sum_{j \in F_1} (1 - x_j) + \sum_{j \in F_0} x_j \right], \tag{4.2}$$

where $\Delta(x, \bar{x})$ denotes the Hamming distance between the initial solution $\bar{x}$ and the current solution x as defined in equation (3.2). When one of those fixed variables flips, the right hand side will be less or equal to zero. Since the Hamming distance is always greater or equal to zero, the constraint is satisfied (even if the Hamming distance is smaller than k).

4.2.4 Utilizing the Extensions

When developing a local branching metaheuristic, most often a combination of the extensions described above works best. For example, it is apparent that when aborting local trees, one may also change the local branching parameters for the value of k and the number of variables to be fixed. When using multiple trees, it may be reasonable to start different trees with different local branching parameters.

The combination of variable fixing and node limits on local trees is a straight-forward way of tightening the search tree as the global solution improves. For example, the search may be started with rather weak constraints, i.e. a relatively high value of k and no or little variable fixing. When the local tree fails to yield new solutions in a given node limit, it is aborted and the parameters are modified. For example, the value of k is decreased or the number of variables to be fixed is increased. Then a new local tree can be created, using the new parameters and the last improved solution from the previous tree (or even the initial solution of the previous tree, since the tightening might lead to a faster convergence towards an improved solution). See chapter 8 for a sample implementation of this tightening scheme.

Chapter 5

COIN/BCP

The implementation of the local branching framework is based on the *Branch and Cut and Price* framework (BCP) that is part of the *Computational Infrastructure for Operations Research* (COIN) project [28]. By augmenting an existing Branch and Cut framework reimplementation of a MIP solver is avoided. Furthermore, developers familiar with COIN/BCP can easily use the framework.

5.1 COIN Overview

5.1.1 History

As research in combinatorial optimization advanced tremendously over the past decades, developers faced increasing complexity when trying to implement efficient versions of their algorithms. While standard algorithms like Branch and Cut exist, problem-dependent algorithms often required custom implementations due to problem-dependent methods like variable or cut generation. In the early 1990s a research group was founded with the goal of providing developers a generic software framework which could be adapted to specific problems. This led to the release of *COMPSys* (Combinatorial Optimization Multi-processing System) [12]. After several revisions this project became *SYMPHONY* (Single- or Multi-Process Optimization over Networks). In 1998, a reimplementation in C++ was started at IBM research.

As a result, the COIN project was publicly announced in the first half of 2000, including a generic Branch, Cut and Price framework codenamed *COIN/BCP*. IBM guaranteed to support the online infrastructure for the COIN project for three years, including the website at http://www.coin-or.org, several mailing lists and the source code repository. Much of the initial development was done by IBM researchers, but in the past years the spirit of open source has picked up and has led to various contributions by external researchers.

5.1.2 Components

Our framework uses the following components of the COIN project:

- *BCP*: the Branch, Cut and Price framework used for solving MIPs.
- *OSI*: the Open Solver Interface, a standardized API for calling math programming solvers. It is used by BCP to call a simplex solver for solving the LP relaxations. A

wide variety of solvers is supported, most notably the free COIN/CLP and the commercial CPLEX MIP solvers.

- *CLP*: COIN LP, the COIN project's LP solver. This is a free implementation of a simplex solver.
- *CGL*: A Cut Generator Library for generating standard cuts for IP problems like Gomory cuts or knapsack cover cuts.

5.2 Design of COIN/BCP

The following introduction to the design of COIN/BCP is based on the user manual by Ralphs and Ladányi [36]. The major design goals for COIN/BCP are portability, efficiency and ease of use. It provides a black-box design with a clean end-user interface that keeps most of the actual implementation hidden from the user.

COIN/BCP was developed using an object-oriented approach. The central objects are cuts and variables that can be used as base classes for user-defined objects. Additionally, user objects provide methods that can be re-implemented to alter specific aspects of the algorithm, like tightening variables or adding cuts. While this approach enables a straight-forward implementation for many combinatorial optimization tasks, there is still enough flexibility left even for implementing complex meta-algorithms like local branching with little or no changes of the COIN/BCP code.

5.2.1 Variables and Cuts

Since search trees can easily contain hundreds of thousands of nodes, a simple object-oriented approach storing the variables and cuts for each node in objects leads to excessive memory consumption. COIN/BCP tries to reduce memory usage by keeping the number of active variables and cuts (the *active set*) as small as possible by using data structures that make it possible to move objects in and out of the active set efficiently. This is accomplished by maintaining an abstract *representation* of each global object that keeps information about adding or removing it from a particular problem instance (i.e. a particular LP relaxation).

In other words, a *variable* does not represent a specific column of a LP relaxation, but is an abstract object that can be *realized* as a specific column of a LP relaxation. Similarly, a *cut* does not describe a specific row of a LP relaxation, but it contains an abstract cut that can be realized as a row in a LP relaxation.

COIN/BCP distinguishes between two groups of cuts and variables: so-called *core cuts* and *core variables* that are active in all subproblems, and *extra cuts* and *extra variables* that can be added and removed dynamically. Extra cuts help to reduce the active set, but require additional bookkeeping when adding or removing them from the formulation. There are two different types of extra cuts:

- *Indexed cuts* are represented by a unique index value. The user must be able to generate the corresponding row cut when given the index number by using some kind of a virtual global list known only to the user. This is the most efficient way of representing extra cuts in a formulation and is particularly useful when the number of cuts is very high and most likely only few are violated by any feasible solution. Using indexed cuts, only

constraints that are violated by a given LP solution have to be realized as rows in the LP relaxation. The downside is the extra bookkeeping involved for adding and removing those cuts, and the user must find an enumeration scheme when using indexed cuts.

- *Algorithmic cuts* give the user absolute freedom, especially in the case when the number of cuts is not known a priori and the cuts cannot be enumerated. The only requirement is that the user must be able to generate the corresponding row when given a set of active variables by some sort of algorithm. The downside, as with indexed cuts, is the fair amount of bookkeeping involved for creating and removing algorithmic cuts.

Indexed and *algorithmic* variables work in a similar way. Indexed variables are generated by the user when given an index number. They are useful when given a big number of variables while most likely only few of them would be different from zero. Adding all variables to the core matrix would increase the problem complexity enormously. Similar to algorithmic cuts, algorithmic variables can describe any user-defined constraint that is violated by a given LP solution. Indexed and algorithmic variables are also essential for column generation algorithms that generate variables during the computation.

5.3 COIN/BCP modules

COIN/BCP is grouped into four independent modules. They communicate using a message-passing protocol which is defined in a separate API. Thus they are well equipped for parallel execution, even on separate machines connected by a network.

5.3.1 The Tree Manager Module

The tree manager (*TM*) module represents the master process of the computation algorithm. It is responsible for problem initialization and controls the other modules. There is only one tree manager for any computation. Its main functions are:

- Reading parameters and the problem instance from the command line or from a file.
- Constructing the root node of the search tree.
- Beginning the computation by creating LP processes (see below) to solve individual nodes of the search tree.
- Receiving new solutions from the child LP modules and storing the best one.
- Receiving new subproblems and storing them for later processing.
- Pruning subproblems based on the global upper bound.
- Sending stored subproblems to idle LP processes.
- Printing the final result when the computation has finished (i.e. all subprocesses are in an idle state and all subproblems have been solved.)

5.3.2 The Linear Programming Module

The linear programming (*LP*) module performs the actual computation, i.e. the bounding and branching operations. Its main functionality includes:

- Requesting new subproblems from the tree manager.
- Receiving and processing subproblems.
- Choosing branching objects and sending the resulting subproblems back to the tree manager.

5.3.3 The Cut Generator Module

Since cut generation may be computationally expensive, it can be performed inside a separate cut generator module. It receives a LP solution by a LP process, tries to generate valid inequalities violated by this solution, and sends the cuts back to the LP solver. Then it remains in an idle state until a new solution is sent to the cut generator.

5.3.4 The Variable Generator Module

Similar to the cut generator module, the variable generator module's only responsibility is to generate variables for a given LP solution. If any variables are generated, they are sent back to the requesting LP process and the variable generator module keeps waiting for new solutions.

5.4 The Linear Programming Module

For a better understanding of our local branching extensions, a deeper explanation of the linear programming module is required. The LP module uses a LP engine for finding upper bounds, generates cuts and variables when necessary and performs branching operations.

5.4.1 The LP Engine

The LP module uses the *Open Solver Interface* (OSI) in order to communicate with third-party LP libraries or *LP engines*.

5.4.2 Managing the LP Relaxation

The LP module is responsible for managing extra variables and cuts. It does so by maintaining a local cut pool where any generated extra cuts are stored. In each iteration, up to a specified number of the strongest cuts are added to the problem. A cut's strongness corresponds to the degree of violation in the current LP solution. Cuts that proved ineffective over a specified number of iterations are purged from the cut pool. Variables can be tightened by user-defined methods before solving the LP.

5.4.3 Branching

Branching is performed when no new cuts were generated or the user forces branching. A *branching object* describing all of the new cuts and variables and their corresponding bounds is generated and sent back to the tree manager. The branching operation can be based on cuts or on variables. Optionally *strong branching* can be performed. In strong branching, several branching objects are created and then pre-solved, i.e. quickly optimized in a probably non-optimal way. The most promising candidate, based on some internal rule (e.g. best objective value) or on the user's decision, is used for branching.

By default, COIN/BCP uses branching on fractional variables. One (standard branching) or several (strong branching) fractional variables are selected and corresponding branching objects are generated. The selection of variables can be user-defined or based on some internal rule, COIN/BCP allows to specify the number of the most fractional variables (i.e. those nearest to 0.5 in a binary optimization problem) and the number of those variables close to one to be selected for (strong) branching. When the total number of variables is greater than one, strong branching has to be enabled.

5.5 Parallelizing COIN/BCP

Since Branch and Cut methods can heavily benefit from parallelization one design goal of COIN/BCP was that of parallelization. There are two main sources of parallelism: Obviously, each subproblem of the candidate list can be solved independently from the others. This can be accomplished by spawning more than one instance of the LP module, either on one machine (reasonable for multi-processor systems) or on a cluster of computers connected by a network.

The second source lies within the processing of a single subproblem: the individual tasks can be parallelized with the LP solver, which means a node can be completely processed in roughly the time it takes the LP engine to solve the relaxation. This is the reason that the potentially expensive cut and variable generators are placed in separate modules outside the LP module.

In COIN/BCP, the architecture is based on a master/slave model. The tree manager assumes the role of the master process in control of slave processes that execute its orders. The tree manager is responsible for spawning at least one process of each type (LP, cut generation, variable generation) and keeping them busy until all subproblems are solved.

5.5.1 Inter-Process Communication

COIN/BCP allows the user to choose between sequential and parallel execution. It is based on an abstract message passing protocol with parallel and sequential implementations. The former is an interface to the Parallel Virtual Machine (PVM) protocol, the latter emulates a parallel machine that effectively executes the algorithm sequentially inside a single process. Support for other communication frameworks can be added by implementing the abstract base class of COIN/BCP's messaging framework.

One instance of an object in memory can never be shared between different modules since these modules might be executed in different processes or on different machines. Instead, objects can be *packed* and *unpacked* into a COIN-specific buffer class (BCP_buffer). The messaging framework uses these buffers for communication between modules, thus it is possible

to transmit user-defined objects by implementing methods for packing and unpacking objects of these types.

5.5.2 Fault Tolerance

Using distributed computation, fault tolerance becomes important because a single crashed machine should not cause the termination of the whole program. For this purpose, the tree manager tracks all processes and restarts them as necessary. When a process is lost, the subproblems assigned to this process are reassigned to other processes. Additionally, new machines can be added to the distributed network on the fly without having to restart the computation.

5.6 Developing Applications with COIN/BCP

This section gives a brief overview of the basic steps for developing an application with COIN/BCP. It focuses on the parts that will be modified in our local branching framework, a more complete description can be found in the COIN/BCP user manual [36].

Developing an application for a specific problem basically means subclassing some of COIN/BCP's provided classes, implementing some abstract methods and overriding others to diverge from default behavior. The main classes designed for derivation by the user are:

- BCP_lp_user: The user-defined LP module extension. By subclassing it it is possible to modify the LP module's decisions (e.g. whether to branch or generate cuts) and to store problem-specific data. One object of this type is generated for each LP process.
- BCP_tm_user: The user-defined tree manager extension. It is embedded into the tree manager module and is responsible among other things for initializing the problem and deciding on the tree search strategy.
- BCP_vg_user is used for implementing user-defined variable generators.
- BCP_cg_user provides a possibility to generate cuts inside a separate process.
- USER_initialize: This class is used for instantiating objects of the derived BCP_xx_user classes.
- BCP_cut: This abstract base class is used for describing cuts, allowing the user to derive problem-specific cut classes. The subclasses for core, indexed and algorithmic cuts are derived from this class.
- BCP_var: This class can be subclassed for describing user-defined variable types. Subclasses for core, indexed and algorithmic variables are already defined.
- BCP_solution: The abstract base class for describing feasible solutions. A generic implementation exists (BCP_solution_generic).

In the BCP_tm_user and BCP_lp_user classes there are some key methods that are of great importance for the implementation of our local branching framework. For a complete description of these classes, refer to the autogenerated documentation and the user's manual [36].

5.6.1 The BCP_tm_user Class

- pack_module_data(): This method is invoked to pack the data needed to start the computation in other modules. This can be used for sending problem-specific data (e.g. local branching parameters) to the LP module.
- unpack_feasible_solution(): This method is called when the tree manager received a new feasible solution. By overriding this method the user gets *every* feasible solution found by a LP module, which can be used for further enhancements (e.g. crossover between two or more solutions when using a genetic algorithm).
- initialize_core() and create_root() are used for initializing the problem (probably by reading it from a file) and setting up the root node of the search tree.
- compare_tree_nodes(): This method is essential for the tree search strategy. It is invoked by the tree manager when a new tree node was received which will be inserted in the candidate queue. By overriding this method it is possible to use an arbitrary tree search strategy, we use this to calculate local tree nodes before any other tree nodes. The standard implementation can be configured by a parameter to perform either a depth first, breadth first, or best bound first tree search.

The candidate queue

The tree manager keeps a list of all unprocessed subproblems in a single *candidate queue*. This is also known as a *single-pool BCP algorithm*. The candidate queue is implemented in the *BCP_node_queue* class as a heap-based priority queue. When the tree manager receives new subproblems from one of the LP processes, the *priority* of the item is determined indirectly by repeatedly calling the binary compare_tree_nodes() function until the final position of the subproblem has been found. The subproblems remain in the candidate queue until taken out by the tree manager for an idle LP process. Since the lower bound may have increased since the subproblem was inserted into the queue, the subproblem may be pruned before it is actually sent to an LP process. In this case, the next subproblem is taken from the queue.

5.6.2 The BCP_lp_user Class

- unpack_module_data(): This method is the counterpart to the pack_module_data() method of the BCP_tm_user class. It receives the information sent by the tree manager and can be used to initialize problem-specific data in a LP module.
- initialize_solver_interface() can be overridden to use a specific LP engine (like COIN's own CLP solver or the commercial CPLEX solver).
- initialize_new_search_tree_node(): This method is called before a new tree node is solved, providing an opportunity to tighten or fix variables.
- generate_heuristic_solution(): When the problem's structure allows to quickly generate feasible solutions based on the current LP solution (e.g. by clever rounding of fractional values), this method can be overridden to generate feasible solutions. By finding good solutions the search tree size can be drastically reduced.

- select_branching_candidates(): This method decides whether to branch or not, and selects branching candidates. The default implementation uses branching on fractional variables, the local branching framework will override this method to implement local branching cuts.

Chapter 6

Implementation of the Framework

The main intention for creating a local branching framework was to provide a clean, reliable and extendable framework for local branching metaheuristics. The main design goals of the local branching framework are:

- *Problem-independent functionality*: The local branching framework should be usable for any binary IP problem. Therefore, it should not make problem-dependent assumptions and separate local branching logic from problem-dependent functionality.
- *Explicit local branching metaheuristics*: It should be possible to literally write a metaheuristic function without being forced to scatter the algorithm over many COIN/BCP classes.
- *"Transparency"*: The implementation of an algorithm for COIN/BCP should not be much different from the implementation using our local branching framework. Furthermore, existing COIN/BCP advantages such as parallelization should not be affected by local branching.
- *Avoid changes to COIN/BCP sourcecode*: It would have been possible to embed local branching directly into the COIN/BCP source repository. However, this would tie local branching very close to COIN/BCP's internals which are subject to change at will. Additionally, it would be much harder to integrate future bugfixes and enhancements of COIN/BCP.
- *Hide COIN/BCP internals*: The complexity of COIN/BCP should be hidden from the local branching metaheuristic. Instead, service methods for querying the current state of local branching should be provided.

These goals were met by subclassing the predefined user classes of COIN/BCP (mainly BCP_lp_user and BCP_tm_user) and embedding the local branching algorithm in those subclasses. Additionally, a *local tree manager* class provides handlers and parameters for controlling the flow of the local branching algorithm. This way most of the complexity and the COIN/BCP-specific implementation is hidden from the user.

Ideally, enabling local branching for an existing COIN/BCP program would be done by replacing the COIN/BCP user classes with the framework's derived classes. Of course, some additional initialization has to be performed since some classes of the framework need to be subclassed by the user again.

The main classes of the framework are:

- LB_tm: the local branching framework's tree manager implementation. It is responsible for managing local trees, creating and terminating local trees, and controlling the LP modules.
- LB_lp: the local branching framework's LP module. It executes the tree manager's instructions regarding local branching, i.e. creating local cuts in the branching operations according to the given parameters (e.g. value of k).
- LB_MetaHeuristic: the user's control module. It provides methods for the user to create and terminate local trees, provides statistical data about all local trees, and handlers that are repeatedly called and are intended to be used to implement another high-level heuristic like an evolutionary algorithm. When using the framework, this is the main class the user has to care about. Unlike COIN/BCP modules, this module is not executed within its own process, but is attached to the tree manager module.
- LB_init: the local branching framework's implementation of the USER_initialize class. It is used to initialize instances of the tree manager (LB_tm) and LP modules (LB_lp). Currently LB_init does not implement any initialization logic on its own, it merely exists for extension purposes.
- LB_cut: a simple row cut used for the Hamming distance constraint.
- LocalTreeManager: an internal data manager class that is responsible for tracking all local trees, managing the found solutions for local trees, and maintaining a list of all currently active local trees. The user probably does not need to override this class except when additional data about local trees should be stored.
- LocalTreeIndex: used to store information about all existing local trees. An instance of this class is shared by the LB_MetaHeuristic and LocalTreeManager objects, the latter being mostly responsible for updating the information in the LocalTreeIndex, while the metaheuristic object can use the index to determine its actions (e.g. terminating a local tree that appears to be stagnating).
- LocalTree: keeps information about a single local tree. Everything the tree manager (and therefore the other local branching classes) knows about a local tree is kept in a LocalTree object. For example, it contains information about the number of active nodes, the number of created nodes since the last improvement of the objective value or the current best solution found in the tree.

Figure 6.1 shows an UML diagram for the classes attached to the LB_tm class.

There are four classes that must always be subclassed for a working program. The methods to be implemented are defined by the COIN/BCP superclasses and have to be implemented anyway to get a working program (except for the LB_MetaHeuristic subclass).

- LB_tm: pack_cut_algo() and unpack_cut_algo() must be able to pack and unpack LB_cut row cuts (although other custom cut types can be supported too). initialize_core() must be implemented to initialize the core matrix, ending with a call to LB_tm's version of this method to complete initialization. The user must also implement the create_lbh() method to return a new instance of his implementation of the LB_MetaHeuristic subclass.

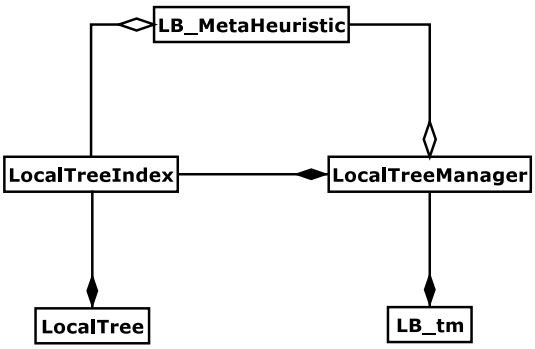

Figure 6.1: UML class diagram for the tree manager extension

- LB_lp: pack_cut_algo() and unpack_cut_algo() have to be implemented similarly to the user's LB_tm implementation. initialize_solver_interface() has to create the LP engine for the LP process. generate_heuristic_solution() *may* be implemented to obtain feasible solutions from LP results (e.g. by clever rounding).
- LB_init: lp_init() and tm_init() are implemented to return new instances of the user's LB_lp and LB_tm subclasses. The tm_init() method is a good place to initialize the problem instance. LB_tm's initialize() method has to be called immediately after the tree manager has been created.
- LB_MetaHeuristic: at least initial_solution() must be implemented to return the initial solution used for the very first local tree. The user probably wants to override some other methods (especially tree_finished()) in order to create more local trees during the computation process.

6.1 Integrating Local Branching into COIN/BCP

The main effort of implementing local branching went into the interaction between the local branching metaheuristic and the COIN Branch and Cut framework. A major goal was to exploit the existing framework as much as possible. Rewriting parts of the COIN/BCP code to implement local branching would leave the user stuck to exactly one version of COIN/BCP, without having the benefit of upcoming enhancements and bugfixes.

Our implementation requires only minimal patches of existing COIN/BCP code where the original user classes did not provide the necessary flexibility. These changes are described in appendix A.

6.1.1 Identifying Local Tree Nodes

COIN/BCP provides a mechanism to attach an *user data object* to nodes for storing individual information about each node. By creating a subclass of BCP_user_data and creating methods to pack and unpack objects of this subclass to a BCP_buffer, the LP modules can attach any user-defined data to a single node of the search tree.

The local branching framework uses this method to assign each node a *LB_user_data* object. Its main components are the *node type*, the unique number of its local tree if it is in a local tree and some internal information. The node type distinguishes three types of nodes:

- UD_LocalRoot: represents a *local root node*, i.e. the root node of a local tree.
- UD_NormalRoot: represent a *normal root node*, i.e. the root node for the search tree outside the local tree.
- UD_LocalNode: represents a node in a local tree.

Note that nodes outside a local tree do not have a type, in fact they have no assigned user data objects at all since there is no extended information that might be of interest. The terms *local* and *normal* root node emerged from the way COIN/BCP handles branching: when a subproblem is branched, appropriate *branching objects* are created (containing the variables or cuts to be branched on), and these are used to create two or more child nodes that represent the root nodes of the new subproblems. When local branching is initiated, branching occurs on a local branching constraint, i.e. on a cut. Two children are created: one with a Hamming distance of $\Delta(x, \bar{x}) \leq k$, the other with a Hamming distance of $\Delta(x, \bar{x}) > k$. These child nodes represent the root nodes of the new subproblems: the first being the *local root node*, the second being the *normal root node*.

The children of the local root node form a local tree and are assigned an unique identification number (ID), represented by the class *LocalTreeId*. In order to guarantee unique tree IDs even when the trees are created in different LP processes, a LocalTreeId consists of the internal COIN index number of its root node and the unique index number of the LP process where it was created (supplied by the tree manager). The other classes do not care about LocalTreeId's internals, all they need are the pre-defined operators ($==$, $!=$, $<$) and the packing and unpacking methods for transferring LocalTreeIds between modules.

6.1.2 The LB_tm Module

The LB_tm module provides our own implementation of COIN/BCP's tree manager module. It is derived from BCP_tm_user and implements some of its methods to integrate local branching into the tree manager. It provides the LP modules with commands concerning local branching, is able to create local trees by transmitting the appropriate root nodes and keeps track of the number of created and pruned nodes for all local trees.

LB_tm implements several methods of the BCP_tm_user class:

- pack_module_data(): sends miscellaneous initialization information to the LP process. This includes the initial solution for local branching, if local branching is enabled at all, and the value of k to be used for the first local tree.
- pack_user_data(): this method is called by COIN/BCP when a node with an user data object is sent to a LP module. Additionally to packing the node's LB_user_data object, the tree manager updates its tree statistics about pruned nodes and sends additional information when a new local tree is started (i.e. the initial solution, value of k, and some other necessary information).
- unpack_user_data(): used to unpack an LB_user_data object sent from a LP process.

- unpack_feasible_solution(): this method is invoked by COIN/BCP when a new feasible solution was sent by a LP process. This method unpacks the feasible solution (of type BCP_solution_generic) and updates the LocalTreeIndex's statistics when it was found in a local tree. When a new global optimum is found, the solution is broadcasted to all active LP processes using COIN/BCP's messaging framework.

- compare_tree_nodes(): this binary comparison function is crucial for the local branching metaheuristic. It is called by COIN/BCP to insert a new tree node in the internal candidate queue. The nodes are ordered by priority, and the first node is the next to be sent to an idle LP process. COIN/BCP implements this method in a way that it represents a certain tree search strategy (i.e. ordering nodes by level for breadth-first or by upper bound for best bound first search). The local branching framework extends this method by always preferring local tree nodes to "normal" nodes. When both nodes are local tree nodes (or both are normal nodes), the COIN/BCP comparison function is called.

6.1.3 The LB_lp Module

The LB_lp module implements local branching for the LP module. It executes the commands sent from the tree manager (LB_tm) module, and is able to create local trees when a normal root node is sent.

The following methods of the BCP_lp_user superclass have been implemented:

- unpack_module_data(): the counterpart to LB_tm::pack_module_data(). Initialization of this LP process is performed, and common parameters like the value of k are set, and the initial feasible solution used for local branching is unpacked.

- pack_user_data(): packs an LB_user_data object to a buffer.

- unpack_user_data(): unpacks an user data object. This method is called by COIN/BCP when a new node with an attached user data object arrived from the tree manager. The additional information sent by LB_tm::pack_user_data() when a new local tree should be created is also stored in the LB_lp object.

- unpack_user_message(): this function was patched into COIN/BCP to allow transmitting user-defined messages between the tree manager and LP modules. By setting a certain message tag number this method gets called when the message arrives at the process.

- pack_feasible_solution(): packs a feasible solution to be sent to the tree manager. Additional to its predefined behavior, the local branching framework adds the current user data object if the feasible solution was generated in a local tree. This makes it possible for the tree manager to assign each received solution to the local tree where it was found.

- initialize_new_search_tree_node(): this function gets called before a node is processed (i.e. before the LP relaxation is computed). This allows the user to tighten variable and cut bounds. The local branching framework performs two major tasks in this method:

  – When the local root node of a new tree is processed, variable fixing might occur. A given percentage of all free variables that is equal in the initial solution of the local tree and the current LP result is fixed to its value.

  - As explained in section 4.2.3, the inverse constraint when fixing variables is expressed as a row cut. Depending on whether the corresponding local tree was completely solved, this row cut has to be (de-)activated when the normal root node is processed.

- select_branching_candidates(): this method is invoked by COIN/BCP when the LP relaxation of a subproblem has been solved. This method can either decide to reprocess the subproblem when cuts have been added, or to return one or more branching objects. When a local tree should be created, an appropriate local branching constraint (a Hamming distance cut by default) is created and a branching object is returned. The local branching cut is created by the virtual method create_local_constraint() that can be re-implemented by the user.

- set_actions_for_children(): when a branching object was chosen, the LP process decides for each child whether to keep it for immediate processing or to return it to the tree manager. At most one child can be kept in a LP process. The local branching framework uses this method to force processing of the local root node when a new tree was created.

- set_user_data_for_children(): the user data information for the child nodes of a branching object is generated after the branching object was chosen. When a new local tree is created, this method sets the local tree identification number and other internal data about the local tree. When an existing local tree is branched, it propagates the information stored in the current user data object to its children.

Creating the Hamming distance cut

LB_lp::create_hamming_constraint() generates a Hamming distance constraint used for local branching. It resembles the local branching constraint as described by Fischetti and Lodi. In its current implementation it is restricted to binary IPs. It can be used as a template for custom local cut generators.

Note that create_hamming_constraint() is not virtual, the framework calls the virtual function create_local_constraint() when creating local cuts which in turn calls the Hamming distance generator. When implementing new cuts, simply override the latter virtual function. Both functions return a local branching object and get information about the current node through their input parameters:

- lpres represents the current LP optimum.

- vars contains all available variables at the current node.

- cuts contains all current cuts. The generated local cut(s) can not be appended to this collection, they must be contained in the returned branching object.

- br_sol is the feasible solution sent from the tree manager to be used for local branching (the *initial solution*).

The creation of the Hamming distance constraint is straight-forward: first, the feasible solution br_sol is unpacked to a local array for easier access. Then the variable coefficients of the cut are generated. Each variable that flips its value in any feasible solution must be

detected, and the sum of all changed variables must not be greater than k. Thus, the coefficients for the cut are:

$$c_j := \begin{cases} 1 & \text{if } j \in S_0 = \{j : \bar{x}_j = 0\}, \\ \text{-1} & \text{if } j \in S_1 = \{j : \bar{x}_j = 1\}, \end{cases} \tag{6.1}$$

where $\bar{x}$ is the initial solution for the local tree. The local cut is then described as

$$0 \leq \sum_{j \in S_0} x_j + \sum_{j \in S_1} (1 - x_j) \leq k \tag{6.2}$$

for any feasible solution x. Simple transformation leads to the row constraint as it will be passed to COIN/BCP:

$$-|S_1| \leq \sum_{j \in S_0} x_j - \sum_{j \in S_1} x_j \leq k - |S_1|. \tag{6.3}$$

After creating a row cut with the bounds for the local tree and the rest of the search tree, some variables are selected for fixing if the tree manager requested variable fixing for this local tree. The indices of all free (non-fixed) variables equal in both the initial and the current LP solution are stored in an array, which in turn is used to randomly pick the requested number of variables. An additional constraint for the normal tree is added as described in section 4.2.3. The variables of the local tree will be fixed when initialize_new_search_tree_node() is called for the local root node. The list of picked variables can be kept inside the LB_lp module since the local root node is processed immediately after branching by the same process.

6.2 Managing Local Trees

In order to provide a clean separation between the low-level local branching implementation represented by the LB_tm and LB_lp classes and global local branching state information, the *LocalTreeManager* class was introduced.

It encapsulates all methods for tracking local trees, such as maintaining active node numbers and assigning found solutions to local trees. A *LocalTreeIndex* object is used for storing the data, which is shared with the *LB_MetaHeuristic* objects that is responsible for controlling the local branching algorithm.

To emphasize the different purposes of these classes, a quick overview of the control distribution follows. All classes below (except LB_lp) are effectively singletons inside the LB_tm process.

- The LB_lp module(s) process individual subproblems.
- The LB_tm module assigns subproblems to LB_lp modules and receives all new subproblems and solutions. When it receives information about a local tree (e.g. a new solution was found), the appropriate LocalTreeManager method is called. It also polls the LB_MetaHeuristic object for commands, e.g. creation or termination of local trees.
- The LocalTreeManager is mostly a passive module that provides data management routines concerning local trees. Its only active part lies in the activation of LB_MetaHeuristic routines on certain events, e.g. calling a method to notify the meta heuristic of a new solution or a terminated tree.

- The LB_MetaHeuristic object is mainly responsible for controlling local branching – that is, creating new local trees or terminating existing ones. Additionally, the initial solution for the first local tree is generated inside this class.
- The LocalTreeIndex serves as shared data pool for the LocalTreeManager and LB_MetaHeuristic classes. The former is responsible for keeping the information up to date, the latter uses it mainly as decision source (e.g. to terminate all trees with more than 50.000 nodes).

6.2.1 The LocalTreeIndex

The LocalTreeIndex gathers information about all local trees and provides miscellaneous statistical information, e.g. the number of active trees or the number of created nodes per tree.

The LocalTree class

A local tree is represented by a *LocalTree* object. This class provides several get and set methods. The latter are called by the LocalTreeManager, but the former can be also used in some LB_MetaHeuristic methods to query miscellaneous information about the given local tree. The most significant properties of a local tree are:

- get_nodes_created() returns the total number of created nodes in this local tree.
- get_nodes_deleted() returns the total number of deleted nodes in this subtree. This includes pruned nodes (because of their upper bound), infeasible nodes, and nodes deleted by the tree manager when a local tree was terminated.
- get_active_nodes() returns the number of active (i.e. not processed and not deleted) nodes, in other words the difference between created and deleted nodes. A local tree with no active nodes is terminated, all active local tree must have at least one node that has not been deleted.
- get_nodes_created_since_improvement() returns the number of created nodes since the last improvement of the best solution *in this local tree*.
- get_nodes_deleted_since_improvement() is the equivalent number for deleted nodes.
- get_terminated() returns true when this local tree has been explicitly terminated. This is a helper function for the user to avoid terminating the same tree several times (since tree termination might not occur immediately, depending on the number of active nodes.)

The LocalTreeIndex class

The LocalTreeIndex class stores all local trees in an associative container (using the previously introduced LocalTreeIds as key and LocalTree as value type). The local tree manager is responsible for creating new entries when necessary, so the LocalTreeIndex actually provides a single service method:

find() accepts a LocalTreeId as parameter and returns the corresponding LocalTree object, or throws an exception when the local tree does not exist.

Additionally, miscellaneous statistical information about all local trees (e.g. the best global solution found so far, or the number of nodes created since the last tree was terminated) is provided through getter methods similar to those in the LocalTree class. For a complete description, refer to the autogenerated class documentation.

6.2.2 The LocalTreeManager

The LocalTreeManager is instantiated by the LB_tm object and assumes control over the LocalTreeIndex data storage. It provides a clean interface for all tasks concerning local tree information, such as assigning solutions to local trees, adding created nodes or removing deleted nodes. Additionally, it maintains the lists for active, terminated and aborted trees in the LocalTreeIndex. The user probably does not want to interfere with the LocalTreeManager's methods, they are automatically called from LB_tm when needed. Instead, the LocalTreeManager delegates control to the LB_MetaHeuristic object, which will be implemented by the user and is responsible for local tree control.

6.3 Controlling Local Branching

The LB_MetaHeuristic class provides a clean and simple encapsulation of the local branching algorithm. Its main goal is to provide an interface to the local branching framework without requiring the user to deal with the internals of COIN and the framework. LB_MetaHeuristic is an abstract class that does not implement any local branching functionality. However, it takes little to implement the standard local branching algorithm.

LB_MetaHeuristic operates asynchronously in the sense that its actions, for example local tree creation, are not immediately executed. Instead, internal flags indicate the action to be taken by the tree manager when it is possible. This limitation is caused by the internal structure of COIN which imposes certain limits on direct control of the computation in exchange for performance, efficiency and parallelism. This should be taken into account when advanced tasks, such as creating multiple local trees at once, do not work as expected.

Local branching control is basically performed by two operations: creating local trees and terminating local trees. Additional parameters influence local branching, such as the value of k, or the amount of variables to be fixed.

LB_MetaHeuristic offers the following methods and data fields for local branching control:

- create_tree() sets internal variables to tell the tree manager that a new tree should be created. The initial solution can be passed as a parameter, or the current incumbent solution will be used. Note that subsequent calls to this function have no effect since the actual tree creation is executed asynchronously by the tree manager. For creating multiple trees at once, the calls to create_tree() have to be synchronized, for example using the tree_created() handler described below.

- terminate_tree() tells the tree manager to terminate the local tree with the given LocalTreeId.

- terminate_active_trees() terminates all active trees. This may be especially useful for terminating local branching.

- lb_k contains the value of k to be used for new local trees.

- lb_fixvars contains the amount of variables to be fixed relative to the total number of variables.

The local branching algorithm is determined by LB_MetaHeuristic's reaction to certain events. These event handlers are called by the LocalTreeManager and offer great degrees of freedom for creating own local branching metaheuristics. The event handlers are:

- initial_solution() is called to obtain the initial solution for the very first local tree. The parameters lb_k and lb_fixvars might also be set in this procedure.
- tree_created() is called when a new tree was generated. The corresponding tree identification and the tree object are passed as references. After creating a tree with create_tree(), this is the first occasion to create another local tree.
- tree_finished() is called when a tree is finished, i.e. it has no active nodes remaining.
- new_node_generated() is called whenever a new node was generated in a local tree. Since this method gets called regularly, time-related tasks (such as terminating local trees above a certain node limit) can be implemented in this method.

6.3.1 Implementing a Basic Local Branching Algorithm

In order to emphasize how the LB_MetaHeuristic object can be used for implementing local branching, consider the following task. Standard local branching should be implemented, i.e. one active tree at any given time, with a node limit of 10.000 nodes per local tree. This is accomplished by implementing LB_MetaHeuristic's event handlers in the following way:

- initial_solution() returns a heuristically generated solution and sets lb_k and lb_fixvars to the desired values.
- tree_terminated() creates a new local tree by calling create_tree(). This single function call implements the standard sequential local branching algorithm and ensures that there is always only one active local tree.
- new_node_generated() uses the LocalTreeIndex object (index) to fetch the number of created nodes for the active tree and calls terminate_tree() when the node limit was exceeded.

Chapter 7

Multidimensional Knapsack Problems

7.1 Introduction

Given a knapsack of fixed capacity c and n items with profits p_j and weights w_j for $j = 1 \dots n$, the task is to find the most valuable subset of items that fits into the knapsack. We assume that $p_1 \dots p_n$ and $w_1 \dots w_n$ are positive integers. The *unbounded knapsack problem* does not limit the number of times each item type can be used. In the *binary* or *0-1 knapsack problem*, the number of items is constrained to be 0 or 1. The *multiple-choice knapsack problem* requires the items to be chosen from disjoint classes. In the *multiple knapsack problem*, several knapsacks are to be filled simultaneously.

The rest of this chapter will be based on binary knapsack problems. Formally, a binary knapsack problem can be stated as:

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} p_j x_j & \text{(objective function)} \\ \text{subject to} & \sum_{j=1}^{n} w_j x_j \leq c & \text{(constraint)} \\ & x_j \in \{0,1\} & j = 1 \dots n \end{array} \tag{7.1}$$

In this formulation, x_j is 1 when item j is included in the knapsack and 0 otherwise. $p_1 \dots p_n$ contains the profit (or value) for each item, and $w_1 \dots w_n$ the weight or resource usage. Note that it is trivial to obtain a (poor) feasible solution $x_j = 0$ for all j.

This section is based on the book on knapsack problems by Pisinger et al. [21] which provides a thorough reference for the family of knapsack problems.

7.1.1 Algorithms for Knapsack Problems

All knapsack problems are NP-hard, therefore it is highly unlikely to find an optimal algorithm with a polynomial worst-case time complexity. Despite this, there are algorithms that achieve reasonable solution times also for large instances. The following overview of exact and approximate algorithms is based on [32]. Note that the multidimensional knapsack problem is more complex and thus usually not covered by highly effective approaches like the dynamic programming approach.

- *Branch and Bound*: A Branch and Bound implementation for knapsack problems was first proposed by P. J. Kolesar in 1967 [23].

- *Dynamic programming*: Basically an enumeration algorithm which can achieve excellent performance on some families of knapsack problems, especially those bounded by relatively low integer capacity. For these it is possible to obtain an optimal solution in $\Theta(nc)$ with n being the number of items and c the knapsack capacity.
- *State space relaxation*: A dynamic programming relaxation where the coefficients are scaled by a fixed value. The complexity of an algorithm may be decreased, but the optimal solution may no longer be found. This is an interesting approach for efficient approximate algorithms.
- *Preprocessing*: Some variables may be fixed at their optimal values by using bounding tests.
- *Fully polynomial time approximation schemes* (*FPTAS*): These are heuristics that can find a solution z with a relative error bounded by any constant value ε, i.e. $\frac{z-z^*}{z^*} \leq \varepsilon$, where z^* is the optimal solution value, in polynomial time bounded by the length of the input and $\frac{1}{\varepsilon}$. A fully polynomial approximation scheme for the binary knapsack problem was presented by Ibarra and Kim in 1975 [19].

7.1.2 Multidimensional Knapsack Problems

The generalization of the knapsack problem to more than a single constraint is the *multidimensional knapsack problem*, also known as *d-dimensional knapsack problem* (*d-KP*) or *multiconstraint knapsack problem*. It is defined as an integer program with the following structure:

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} p_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} w_{ij} x_j \leq c_i & i = 1 \ldots d \\ & x_j \in \{0,1\} & j = 1 \ldots n \end{array} \tag{7.2}$$

An equivalent formulation using vectors is:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Wx \leq c \\ & x \in \{0,1\}^n, c \in \mathbb{N}_0^d, W \in {\mathbb{N}_0}^{d \times n} \end{array} \tag{7.3}$$

There are two main characteristics of integer programs that describe multidimensional knapsack problems: First, they are particular difficult instances of integer programming because the constraint matrix W is unusually dense, while most other relevant classes are defined by sparse constraint matrices. But analogously to other knapsack problems it is also particularly easy to find a feasible solution: $x_j = 0$ for all j, whereas in general integer programming finding feasible solutions might be as hard as finding an optimal solution.

Typically the number of items n exceeds the number of constraints d. A rough bound for computing optimal solutions of multidimensional knapsacks with todays algorithms and computers is $n = 500$ and $d = 10$.

It has been shown by Korte and Schrader in 1979 [24] that the existence of a fully polynomial time approximation scheme for a multidimensional knapsack problem even with only two

constraints ($d = 2$) would imply $P = NP$, i.e. that every NP-hard problem could be solved in polynomial time. However, there exists a polynomial time approximation scheme (*PTAS*) with a running time of $\Theta(n^{\lceil d/\varepsilon \rceil - d})$ [4]. Compared to a FPTAS, a PTAS has the drawback of an exponential increase in running time with respect to the accuracy, i.e. its running time is polynomial only with respect to the input length, but not to the required ε value.

7.2 Heuristic Algorithms

The enormous complexity of multidimensional knapsack problems motivated extensive research in heuristic algorithms.

7.2.1 Greedy Heuristics

Greedy heuristics work by inserting all items that do not violate any resource constraints (*primal greedy heuristics*) or by first putting all items into the knapsack and then removing items until the solution becomes feasible (*dual greedy heuristics*). Since the order in which the items are inserted or removed matters and some items are more valuable than others (i.e. offer a relatively high profit for relatively low resource usage), items are sorted by an arbitrary *efficiency* e_j before inserting them into the knapsack. The most obvious efficiency measure for a one-dimensional binary knapsack problem is the relative profit $e_j = \frac{p_j}{w_j}$.

Since there is more than one resource constraint in multidimensional knapsack problems, there is no such trivial method of determining the efficiency of an item. The nearest counterpart would be the aggregation of all d constraints, i.e.

$$e_j = \frac{p_j}{\sum_{i=1}^{d} w_{ij}}, \tag{7.4}$$

where e_j would be the efficiency for item j. The main drawback is that it does not work well when the resource constraints are of different orders of magnitude. In this case, one constraint may completely dominate all others.

This can be avoided by taking the relative weight for each constraint and define

$$e_j = \frac{p_j}{\sum_{i=1}^{d} \frac{w_{ij}}{c_i}}. \tag{7.5}$$

Senju and Toyoda [38] proposed a different way to incorporate the relative distribution of weights by including the difference between the capacity and the total resource usage of all items for a given constraint.

$$e_j = \frac{p_j}{\sum_{i=1}^{d} w_{ij}(\sum_{j=1}^{n} w_{ij} - c_i)}. \tag{7.6}$$

A generalized formulation of these efficiency measures was proposed by Fox and Scudder [37] by introducing a *relevance value* r_i for every constraint.

$$e_j = \frac{p_j}{\sum_{i=1}^{d} r_i w_{ij}} \tag{7.7}$$

Equation (7.4) can be derived by setting $r_i = 1$ for all i, (7.5) by setting $r_i = \frac{1}{c_i}$, and (7.6) by setting $r_i = \sum_{j=1}^{n} w_{ij} - c_i$.

Advanced adaptive algorithms adjust the relevance values when an item was inserted, an early version of such an algorithm can be found in [38].

7.2.2 Relaxation-Based Heuristics

Heuristics based on the LP relaxations of integer programs can also be used for multidimensional knapsack problems with little or no adaptation. A simple and fast approach was given by Bertsimas and Demir in 2002 [3]. It starts by fixing variables in the LP solution depending on a parameter $\gamma \in [0, 1]$:

$$x_j^H := \begin{cases} 1 & \text{if } x_j^{LP} = 1, \\ 0 & \text{if } x_j^{LP} < \gamma. \end{cases} \tag{7.8}$$

In the second phase the subproblem defined by the undecided variables $\gamma \leq x_j^{LP} < 1$ is solved again, and further variables are fixed:

$$x_j^H := \begin{cases} 1 & \text{if } x_j^{LP} = 1, \\ 0 & \text{if } x_j^{LP} = 0, \\ 0 & \text{for } j = \textbf{argmin}\{x_j^{LP} \mid 0 < x_j^{LP} < 1\}. \end{cases} \tag{7.9}$$

The last assignment fixes the variable with the least fractional value to zero. This second phase is repeated until all variables are fixed to zero or one. Small values for γ lead to better solution values but longer running times, while bigger values offer better performance. Setting $\gamma = 1$ is equivalent to rounding down the first LP solution. The authors proposed setting $\gamma = 0.25$ when performance is more important than solution quality.

7.2.3 Hybrid Algorithms

More advanced algorithms combine different approaches to the multidimensional knapsack problem. They are often more complex and require more running time, but can yield near-optimal solutions in many cases where simpler heuristics fail.

One such approach was proposed by Lee and Guignard in 1988 [27]. Their algorithm starts with a modified version of Toyoda's primal greedy heuristic [40]. Instead of deciding on each item separately, they decide on several items at once before recomputing the relevance values, leading to better performance. Based on this feasible solution, the LP relaxation is solved. A comparison between the feasible solution and the LP solution in combination with the reduced costs of the LP solution is used to fix some variables and reduce the problem size. These steps are iterated, the number of iterations is controlled by a parameter.

A more recent heuristic was given by Vasquez and Hao in 2001 [41]. They combine linear programming and tabu search to search binary areas around continuous solutions. This is facilitated by additional constraints that limit the search space around a solution, like a sphere constraint that geometrically isolates the search space.

7.2.4 Evolutionary Algorithms

Evolutionary algorithms (EAs) were inspired by biological evolution and try to mimic the evolutionary process. An evolutionary algorithm keeps one or more *populations* of solutions for a given problem, and tries to improve these solutions by imitating evolutionary procedures like *selection*, *recombination* and *mutation*.

Several evolutionary algorithms exist for the multidimensional knapsack problem. Major differences between algorithms concern varying operators for recombination and mutation, and also different representations of the solutions themselves. Raidl [33, 34, 35] proposed different approaches for the multidimensional knapsack problem, also combining evolutionary algorithms with local improvement heuristics.

A particularly effective approach is based on an EA by Chu and Beasley [6]. It uses a *direct representation* using bit vectors $\in \{0,1\}^n$ for representing solutions. Recombination is done by uniform crossover, i.e. a child solution is created by randomly picking bits from one of its parents. Bit-wise mutation can be used to increase the diversity of solutions. Both crossover and mutation can produce infeasible solutions, so a repair algorithm is required. A two-phase heuristic is used for repairing and local improvement: the items are ordered by an utility ratio similar to the efficiency measures described in section 7.2.1. For repairing solutions, the least promising items are removed until the solution becomes feasible. The local improvement algorithm processes items not present in the current solution by decreasing utility ratio and inserts them if no constraints are violated.

Decoder-based EAs replace the direct representation of a solution with an encoding scheme. Recombination and mutation operate on the encoded solutions, implicitly exploring the original search space. The choice of the encoding scheme can greatly influence the effectivity of recombination and mutation and the convergence of the overall search algorithm.

A possible encoding scheme for knapsack problems uses *permutations*. Instead of storing the value for each variable, a permutation $\pi : J \rightarrow J$ with $J = \{1, \ldots, n\}$ is used to represent a solution. In order to get a direct representation for a solution, decoding starts with the feasible solution $x = (0, \ldots, 0)$. Then the variables are visited in the sequence described by the permutation, and variables that do not violate constraints are set to 1. Mutation randomly exchanges two different positions in a permutation, recombination is done using *uniform order based crossover* [9] which keeps the ordering (but not necessarily the positions) of the parent solutions. A *permutation based EA* for the knapsack problem has been proposed by Hinterding [18] and it has also been applied to the multidimensional knapsack problem by Gottlieb [17], Raidl [33], Thiel and Voss [39].

Chapter 8

A Sample Application: MD-KP

This chapter guides through a sample application for the local branching framework, a Branch and Cut solver for the multidimensional knapsack problem as described in chapter 7. The goal is to optimize the following integer program:

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} p_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} w_{ij} x_j \leq c_i & i = 1 \ldots d \\ & x_j \in \{0,1\} & j = 1 \ldots n \end{array} \tag{8.1}$$

Since the application actually will be embedded into the COIN/BCP main program, it makes sense to adhere to COIN/BCP's directory structure. COIN/BCP provides makefiles for building custom applications where only the added user files have to be defined.

The application source is grouped into the following directories:

- include/ contains all header files for the application's classes.
- LP/ contains the implementation of this program's LP module.
- TM/ contains the tree manager module.
- Member/ contains the other classes, in our case the initialization class (descendant of LB_init) and the metaheuristic.

First of all, defining a class to hold a problem instance simplifies the further operations. In case of the multidimensional knapsack problem, we basically need some arrays to hold the coefficients of the constraints and the objective function, and the corresponding bounds. This class is based on the BranchAndCut example from the COIN source and can be easily adopted to other integer programming problems. It is defined as follows:

```
class KS_prob {
public:
    int nItems;              ///< Total number of items (= columns)
    int nConstraints;        ///< Total number of constraints (= rows)

    double optimalknown; ///< Known optimal solution, -DBL_MAX if unknown

    double* clb;             ///< Lower bound for each core variable (usually 0.0)
```

```
    double* cub;          ///< Upper bound for each core variable (usually 1.0)
    double* val;          ///< Objective value (profit) for each core variable
    double** res;         ///< Resource demands
    double* rescons;      ///< Resource constraints

    CoinPackedMatrix* core; ///< Core matrix
    double* rlb_core;     ///< Lower bounds in the core matrix (usually 0.0)
    double* rub_core;     ///< Upper bounds in the core matrix (usually rescons[rownum])
};
```

Next, the tree manager class will be implemented.

8.1 KS_tm implementation

The tree manager implementation is responsible for reading the problem data from a file, initializing the core matrix of COIN/BCP, and providing methods to pack and unpack cuts.

8.1.1 Test File Format

Our program will read test instances taken from Chu and Beasley's paper on a genetic algorithm for the multidimensional knapsack problem [6]. These test files contain 270 different instances, ranging from very easy to very complex problems. The problems are described in plain text, each file contains 30 instances of the same dimension (i.e. the same number of variables and constraints). The format of the text file is:

- The number of instances (should be 30).
- For each problem:
  - Number of variables n, number of constraints d, optimal solution (if known).
  - The coefficients of the objective function p_j for $j = 1 \ldots n$.
  - For each constraint i: the coefficients of the constraint w_{ij}.
  - The upper bounds of the constraints c_i for $i = 1 \ldots d$.

KS_tm::read_input() reads the data from the given file name using the given instance number into a KS_prob object which is stored in KS_tm. Since it is just a very simple line parser the implementation is omitted here.

8.1.2 Setting up the Core Matrix

The raw core matrix was set up by KS_tm::read_input() and stored into KS_prob::core, but until now COIN/BCP is not aware of the problem data. To do this, we must override LB_tm::initialize_core() to create all core variables and core cuts (i.e. all IP constraints). The core matrix is put together using the coefficients and upper and lower bounds loaded by read_input(). In the end, we call the implementation of the superclass because the local branching framework might have to do some setup actions on its own.

```
void KS_tm::initialize_core(BCP_vec<BCP_var_core*>& vars,
BCP_vec<BCP_cut_core*>& cuts, BCP_lp_relax*& matrix) {
```

```
    // initialize core variables
    for (int i = 0; i < prob.nItems; ++i)
        if (0.0 == prob.clb[i] && 1.0 == prob.cub[i])
            vars.push_back(new BCP_var_core(BCP_BinaryVar, prob.val[i], 0, 1));

    // initialize core cuts
    for (int i = 0; i < prob.core->getNumRows(); ++i)
        cuts.push_back(new BCP_cut_core(prob.rlb_core[i], prob.rub_core[i]));

    // create LP relaxation
    matrix = new BCP_lp_relax;
    matrix->copyOf(*prob.core, prob.val, prob.clb, prob.cub, prob.rlb_core, prob.rub_core);

    LB_tm::initialize_core(vars, cuts, matrix); // execute LB's initializiation
}
```

8.1.3 Packing and Unpacking of Cuts

We do not create own cuts in our sample application, but we still have to provide means to pack and unpack LB_cut objects. These cuts will be used for local branching cuts and inverse variable fixing constraints. Since LB_cut already provides methods for packing and unpacking those cuts, the corresponding implementations in LB_tm are very compact:

```
void KS_tm::pack_cut_algo(const BCP_cut_algo* cut, BCP_buffer& buf) {
    const LB_cut* lb_cut = dynamic_cast<const LB_cut*>(cut);
    if (!lb_cut)
        throw BCP_fatal_error("pack_cut_algo(): unknown cut type!\n");
    lb_cut->pack(buf);
}

BCP_cut_algo* KS_tm::unpack_cut_algo(BCP_buffer& buf) {
    return new LB_cut(buf);
}
```

8.1.4 Sending the Problem Description to the LP Module

The LP module will need the problem description stored in the KS_prob object for heuristically finding feasible solutions. KS_tm::pack_module_data() allows to send any information to an LP process when it is created, thus we simply append our problem description to the given buffer. The local branching framework appends data on its own, so the implementation of the superclass is called too.

```
void KS_tm::pack_module_data(BCP_buffer& buf, BCP_process_t ptype) {
    if (BCP_ProcessType_LP == ptype)
        buf.pack(&prob);
    LB_tm::pack_module_data(buf, ptype);
}
```

Actually only a pointer to the problem description is passed. This is possible since the focus of the sample application does not lie on parallel execution, but on simplicity. However, it is a rather trivial task to write packing and unpacking routines for the KS_prob class (especially since the core matrix does not need to be transmitted).

8.1.5 Creating a KS_MetaHeuristic Object

The KS_MetaHeuristic implements the abstract LB_MetaHeuristic class and contains the user-defined local branching control methods. Since the tree manager does not know the meta heuristic's type a priori, we instantiate a KS_MetaHeuristic object by overriding LB_tm's abstract method create_lbh():

```
virtual LB_MetaHeuristic* create_lbh() {
        return new KS_MetaHeuristic(ltm->index, &prob);
}
```

ltm is the tree manager's LocalTreeManager object, *ltm→index* is the shared LocalTreeIndex, and *prob* contains the problem definition initialized by KS_tm::read_input().

The implementation of the tree manager is now complete, the next task is to implement the LP module.

8.2 KS_lp Implementation

In the LP module implementation, the main effort goes into cut generation and heuristically finding feasible solutions. First, we need the counterpart to pack_module_data, or the local branching framework will use the wrong buffer values. Additionally, methods for packing and unpacking cuts are also required.

```
void KS_lp::unpack_module_data(BCP_buffer& buf) {
    buf.unpack(pprob);
    LB_lp::unpack_module_data(buf);
}

void KS_lp::pack_cut_algo(const BCP_cut_algo* cut, BCP_buffer& buf) {
    const LB_cut* lb_cut = dynamic_cast<const LB_cut*>(cut);
    if (!lb_cut)
        throw BCP_fatal_error("LB_lp::pack_cut_algo: unknown cut type!\n");
    lb_cut->pack(buf);
}

BCP_cut_algo* KS_lp::unpack_cut_algo(BCP_buffer& buf) {
    return new LB_cut(buf);
}
```

We also need to setup the LP solver. Here we will instantiate COIN's own CLP solver, the complete version of the sample application also supports CPLEX.

```
OsiSolverInterface* KS_lp::initialize_solver_interface() {
    OsiClpSolverInterface *clp = new OsiClpSolverInterface;
    clp->messageHandler()->setLogLevel(0);
```

```
    return clp;
}
```

8.2.1 Generating Feasible Solutions

COIN/BCP provides a method where the user can generate feasible solutions from solved LP relaxations. By generating good feasible solutions early in the computation, the search tree size can be reduced. However, the heuristic should not be too complex since this method is called for every solved LP relaxation.

For the multidimensional knapsack problem, we chose a simple greedy heuristic as described in section 7.2.1. The efficiency measure is not based on the resource usage of each variable, but on its value in the LP result. Thus, after sorting the variables by descending LP value, all variables that do not violate the resource constraints are added to the solution, which is then returned to the LP module.

```
BCP_solution* KS_lp::generate_heuristic_solution(const BCP_lp_result& lpres,
    const BCP_vec<BCP_var*>& vars, const BCP_vec<BCP_cut*>& cuts) {

    const double* x = lpres.x();
    BCP_solution_generic* sol = new BCP_solution_generic(false);

    // sort variables by LP result value, then insert them in reversed order (best first)
    multimap<double, int> sorted;
    for (unsigned int i = 0; i < vars.size(); ++i)
        sorted.insert(pair<double, int>(x[i], i));

    // track current resource usage
    double* myres = new double[pprob->nConstraints];
    for (int j = 0; j < pprob->nConstraints; ++j)
        myres[j] = 0.0;

    multimap<double, int>::reverse_iterator it;
    for (it = sorted.rbegin(); it != sorted.rend(); ++it) {
        int i = (*it).second;          // number of the variable to be inserted
        bool ok = true;
        for (int j = 0; j < pprob->nConstraints; ++j) {
            // check if any resource constraint is violated
            if (myres[j] + pprob->res[j][i] > pprob->rescons[j])
                ok = false;
        }
        if (ok) {       // insert item into knapsack
            for (int j = 0; j < pprob->nConstraints; ++j)
                myres[j] += pprob->res[j][i];
            sol->add_entry(vars[i], 1);
        }
    }

    delete[] myres;
    return sol;
}
```

8.2.2 Generating Cuts

The second main task for the LP module is cut generation. When a LP relaxation was solved, one can either try to generate cuts violated by the LP result, or the subproblem is branched. COIN/CGL offers several generic cut generators. If one chooses to generate cuts, it can be either done for every node, or for nodes meeting a certain condition. In this example, we choose to generate cuts for nodes at every eighth level of the search tree. We try to create generic *knapsack cover cuts* and *Gomory cuts*. The cut generators are first instantiated and stored in a list. We also set the maximal number of items for the knapsack cut generator. Higher numbers lead to higher computational complexity, but also higher chances of finding cuts. Then every cut generator is invoked to generate cuts and store them in *cutlist*. These cuts are then appended to the *new_cuts* output parameter.

```
void KS_lp::generate_cuts_in_lp(const BCP_lp_result& lpres,
     const BCP_vec<BCP_var*>& vars,
     const BCP_vec<BCP_cut*>& cuts,
     BCP_vec<BCP_cut*>& new_cuts,
     BCP_vec<BCP_row*>& new_rows) {

     vector<CglCutGenerator*> cgs;

     // generate nodes at every 8th level
     if (current_level() % 8 == 0) {
          CglKnapsackCover* kc = new CglKnapsackCover;
          kc->setMaxInKnapsack(pprob->nItems);
          cgs.push_back(kc);
          cgs.push_back(new CglGomory);
     }

     if (cgs.size() > 0) {
          OsiSolverInterface* si = getLpProblemPointer()->lp_solver;
          for (int i = vars.size() - 1; i >= 0; --i)
               si->setInteger(i);

          OsiCuts cutlist;
          for (int i = cgs.size() - 1; i >= 0; --i) {
               cgs[i]->setAggressiveness(100);
               cgs[i]->generateCuts(*si, cutlist);
               delete cgs[i];
               cgs[i] = 0;
          }
          for (int i = cutlist.sizeRowCuts() - 1; i >= 0; --i) {
               LocalTreeId id = in_localbranching() ? get_ks_user_data()->id : LocalTreeId();
               new_cuts.push_back(new LB_cut(cutlist.rowCut(i), id));
          }
     }
}
```

When using cut generation, we have to implement COIN/BCP's cuts_to_rows method. It is used to realize the abstract cut representations to actual rows of the LP relaxation.

```
void KS_lp::cuts_to_rows(const BCP_vec<BCP_var*>& vars, BCP_vec<BCP_cut*>& cuts,
     BCP_vec<BCP_row*>& rows, const BCP_lp_result& lpres,
     BCP_object_origin origin, bool allow_multiple) {
```

```
    const int cutnum = cuts.size();
    for (int i = 0; i < cutnum; ++i) {
        const OsiRowCut* bcut = dynamic_cast<const LB_cut*>(cuts[i]);
        if (bcut)
            rows.push_back(new BCP_row(bcut->row(), bcut->lb(), bcut->ub()));
        else
            throw BCP_fatal_error("Unknown cut type in cuts_to_rows.\n");
    }
}
```

8.3 KS_init Implementation

COIN/BCP provides the USER_initialize class to instantiate custom tree manager and LP module implementations. KS_init implements LB_init, which in turn was derived from USER_initialize. KS_init::lp_init() creates a new KS_lp object, and KS_init::tm_init() instantiates a new KS_tm object. While the former is called once for every created LP process, the latter is only called on program startup. Thus it is used to process command line parameters and initialize the tree manager by calling KS_tm::read_input() for the given file name. Additionally, the global function BCP_user_init() has to be implemented to return a new KS_init() object.

With these three classes, the basic COIN/BCP implementation is finished. For the standard COIN/BCP classes, the implemented methods are sufficient for an executable Branch and Cut algorithm for the multidimensional knapsack problem. The local branching framework requires the implementation of a fourth class, the local branching metaheuristic.

8.4 KS_MetaHeuristic Implementation

All local branching related logic is contained in KS_MetaHeuristic, our implementation of the LB_MetaHeuristic class. Our local branching controller accomplishes three main tasks:

- initial_solution() creates a heuristic solution used for the first local branching tree.
- new_node_generated() is called regularly by the framework and monitors local branching progress. When the given node limits are exceeded, it restarts local branching.
- tree_finished() starts a new local tree when the last active tree finished, effectively implementing the sequential local branching algorithm.

8.4.1 Configuring Local Branching

Before the actual local branching implementation is described, we need a way to adjust certain parameters of our heuristic without recompiling the whole program. COIN/BCP offers a convenient, generic parameter parser that can be used to load user-defined parameters from the command line or from a file, perform type checking and sanity checks, and define default values. In order to utilize COIN/BCP's parser, we start by defining a parameter class, KS_parameters. It is not derived from any other class, instead for each parameter type (integer, double, string, ...) it defines enumerations containing the parameters' names. This class is

used as a generic type parameter for COIN/BCP's BCP_parameter_set::create_keyword_list() and set_default_entries. The first method assigns actual string labels to the user-defined parameters, the second methods initializes all parameters with default values.

For example, the following code defines a couple of double parameters and then implements COIN/BCP's methods for using them:

```
class KS_parameters {
public:
    enum dbl_params {
        LB_FixVarsInitial,
        LB_FixVarsIncrement,
        LB_FixVarsMax,
        LB_MultiK,
        end_of_dbl_params };
};

template<> void BCP_parameter_set<KS_parameters>::create_keyword_list() {
    keys.push_back(make_pair(BCP_string("LB_FixVarsInitial"),
                             BCP_parameter(BCP_DoublePar, LB_FixVarsInitial)));
    keys.push_back(make_pair(BCP_string("LB_FixVarsIncrement"),
                             BCP_parameter(BCP_DoublePar, LB_FixVarsIncrement)));
    keys.push_back(make_pair(BCP_string("LB_FixVarsMax"),
                             BCP_parameter(BCP_DoublePar, LB_FixVarsMax)));
    keys.push_back(make_pair(BCP_string("LB_MultiK"),
                             BCP_parameter(BCP_DoublePar, LB_MultiK)));
}

template<> void BCP_parameter_set<KS_parameters>::set_default_entries() {
    set_entry(LB_FixVarsInitial, 0.0);
    set_entry(LB_FixVarsIncrement, 0.0);
    set_entry(LB_FixVarsMax, 0.0);
    set_entry(LB_MultiK, 1.0);
}
```

BCP_parameter_set also provides methods for reading parameter from the command line, from files, or from input streams. It also offers methods for accessing parameter values (get_entry() and set_entry()) and packing or unpacking of parameter sets. Since the parameters are only important for the local tree metaheuristic, we do not need to pass the parameters to other modules.

8.4.2 Setting up Local Branching

Before the local branching heuristic takes over control, we have to tell the framework if local branching should be enabled at all - and which parameters to use. The event handlers depend on an already running local branching algorithm (e.g. a new node was generated, or a tree was terminated). The decision whether to create a local tree or to use standard branching takes place when the first LP process is initialized in LB_tm::pack_module_data(). LB_MetaHeuristic::*lb_maxpasses* sets the maximum number of local trees. If it is zero, no local trees will be generated at all and standard branching will be used. This information is also used in tree creation methods, which will fail when the number of created local trees exceeds lb_maxpasses.

The KS_MetaHeuristic class sets lb_maxpasses in its read_parameters() method, where the command line is parsed using the parameter methods described in the previous section, effectively disabling local branching when either the maximum number of local trees or the value of k has been set to 0. The value of k for the very first local tree is also set in this method, which is called by KS_init::tm_init() after the tree manager and the KS_MetaHeuristic objects were created.

8.4.3 Creating the Initial Solution

The initial solution is retrieved by the local tree manager when the first LP process is initialized. To demonstrate the use of pseudo-concurrent tree exploration, we use three different heuristics to start local branching with three (partially) disjunct local trees. The solution derived from the first LP result is used for the first local tree, and two greedy heuristics with different efficiency measures provide the other two solutions.

Since the metaheuristic class does not know the first LP result (it is local to the LP process), a small workaround forces the local branching framework to actually use the first LP result: initial_solution() returns an empty solution (i.e. with all variables set to 0) and tells the framework to use the feasible solution derived from the first LP result (if it is better). The first LP-derived solution is certainly better than the empty solution for any feasible problem instance, thus it will always be used. The other initial solutions are stored in the metaheuristic object and are sent to the tree manager as soon as the first local tree is created.

To summarize the steps above, the initial local trees are created in the following way:

1. initial_solution() calls greedy() to generate two initial solutions using two different efficiency measures. These solutions are stored in the KS_MetaHeuristic object for later use.

2. initial_solution() returns an empty solution to force the framework to use the solution derived from the first LP result as initial solution for the first tree.

3. tree_created() issues create_tree() to create the next two local branching trees.

The implementation of greedy() is similar to the feasible solution generator described in section 8.2.1. A greedy heuristic inserts the items ordered by an efficiency value determined by one of the following formulas as described in section 7.2.1:

$$e_j = \frac{p_j}{\sum_{i=1}^{d} \frac{w_{ij}}{c_i}}. \tag{8.2}$$

The profit p_j for item j is divided by the relative resource usage, i.e. the sum of all relative weights. The relative weight concerning a given resource i and an item j is the absolute weight w_{ij} divided by the resource limit c_i. This way, weights are scaled to $[0 \dots 1]$ regardless of their absolute value, leading to a fair consideration of all weights.

$$e_j = \frac{p_j}{\sum_{i=1}^{d} w_{ij}(\sum_{j=1}^{n} w_{ij} - c_i)}. \tag{8.3}$$

The second efficiency measure takes the weight distribution into account, i.e. it emphasizes scarce resources.

The code for sorting the items follows below, the greedy heuristic for inserting the items is the same as in section 8.2.1. The parameter *fun* determines which efficiency measure should be used.

```
vector<double> resource_usages(pprob->nConstraints, 0.0);
if (weight_distribution == fun) {
    // calculate total resource usages for all resources
    for (int j = 0; j < pprob->nConstraints; ++j) {
        for (int i = 0; i < pprob->nItems; ++i)
            resource_usages[j] += pprob->res[j][i];
    }
}

multimap<double, int> relval;
for (int i = 0; i < pprob->nItems; ++i) {
    double allres = 0.0;
    if (relative_weight == fun) {
        // relative weight of each item
        for (int j = 0; j < pprob->nConstraints; ++j)
            allres += pprob->res[j][i] / pprob->rescons[j];
    } else if (weight_distribution == fun) {
        // weight distribution according to Senju and Toyoda
        for (int j = 0; j < pprob->nConstraints; ++j)
            allres += pprob->res[j][i] * (resource_usages[j] - pprob->rescons[j]);
    }
    relval.insert(pair<double, int>(pprob->val[i] / allres, i));
}
```

The implementation of initial_solution() is simple. Two solutions are generated, and an empty solution is returned to force the LP process to use the feasible solution derived from the first LP result.

```
BCP_solution_generic* KS_MetaHeuristic::initial_solution(
        BCP_vec<BCP_var_core*>& corevars,
        std::map<BCP_IndexType, BCP_var*>& vars,
        bool& allow_lp_result) {

    solution_relative_weight = greedy(corevars, vars, relative_weight);
    solution_weight_distribution = greedy(corevars, vars, weight_distribution);
    allow_lp_result = true;
    return new BCP_solution_generic(false);
}
```

The tree_created() handler is called when a new tree was created by the tree manager. Here we can create the two other local trees from the initial solutions created in initial_solution().

```
void KS_MetaHeuristic::tree_created(const LocalTreeId& id, LocalTree& tree) {
    if (1 == index.trees.size())
        create_tree(solution_relative_weight);
    else if (2 == index.trees.size()) {
        create_tree(solution_weight_distribution);
}
```

8.4.4 Imposing Node Limits on Local Trees

While the framework provides all necessary information for terminating trees above a certain node count, the criteria for aborting local trees have to be checked in the meta heuristic. In this metaheuristic, we will implement a rather simple node-based tree termination scheme that has the following characteristics:

- Trees will be aborted when their total number of created nodes exceeds a given limit.
- Trees will be aborted when the number of created nodes since the last improvement of the best feasible solution found inside the local tree exceeds a given limit.
- When a tree is aborted and the maximum number of local trees is not reached, a new local tree is created with the current best global solution. Based on the result of the last local tree, the new tree will be eventually modified:
  - When a better solution was found since the last tree was created, local branching is restarted with this new solution and the initial local branching parameters.
  - When no new solutions were found, the new local tree is tightened (if the corresponding parameters are set): the number of variables to be fixed is increased, and/or the value of k gets modified.

Written as a handler using LB_MetaHeuristic::new_node_generated(), the following code implements node limits for all local trees. Some additional safety checks occurs, such as checking if local branching is enabled (*lb_enabled*). Due to COIN/BCP's asynchronous design, nodes may be added to a tree event after the tree was terminated. Thus it is also checked if the current tree has not already been terminated (*ltree.get_terminated()*). The last expressions of the outer *if* clause formulate the node limits described above using the statistical data of the LocalTreeIndex.

```
void KS_MetaHeuristic::new_node_generated(const LocalTreeId& id, LocalTree& tree) {
    int maxnodes_without_improvement =
        params.entry(KS_parameters::LB_MaxNodesWithoutImprovement);
    int maxnodes = params.entry(KS_parameters::LB_MaxNodes);
    if (lb_k > 1 && lb_enabled && !tree.get_terminated() &&
        ((maxnodes_without_improvement > 0 &&
          tree.get_nodes_created_since_improvement() > maxnodes_without_improvement) ||
         (maxnodes > 0 && tree.get_nodes_created() > maxnodes))) {

        terminate_active_trees();
        if (index.nodes_created_since_improvement < index.nodes_created_since_newtree) {
            // an improved solution was found, restart with this solution
            lb_fixvars = params.entry(KS_parameters::LB_FixVarsInitial);
            lb_k = params.entry(KS_parameters::LB_K);
        }
        else if (params.entry(KS_parameters::LB_FixVarsInitial) > 0.0 ||
                 params.entry(KS_parameters::LB_MultiK) != 1.0) {
            // the old tree did not yield a better solution,
            // so fix some variables and/or modify k value.
            lb_fixvars += params.entry(KS_parameters::LB_FixVarsIncrement);
            lb_fixvars = min(lb_fixvars, params.entry(KS_parameters::LB_FixVarsMax));
            lb_k = (int) round(params.entry(KS_parameters::LB_MultiK) * lb_k);
            lb_k = max(lb_k, params.entry(KS_parameters::LB_MinK));
```

```
                lb_k = min(lb_k, params.entry(KS_parameters::LB_MaxK));
                lb_tightening = true;              // do it only once
            }
        }
    }
```

8.4.5 Handling Terminated Local Trees

LB_MetaHeuristic::tree_finished is called when a formerly active tree has no more active nodes remaining. It may be called when the tree was solved completely, or when it was aborted, e.g. by our new_node_generated() implementation. When the tree was aborted, the tree's get_terminated() function returns true. In both cases, a new tree has to be created. LB_MetaHeuristic::create_tree() takes care of the limit on the total number of local trees by setting lb_enabled to false if necessary. Our implementation additionally tracks the number of retries for the current incumbent solution (i.e. the number of local trees spawned with the last incumbent solution), disabling local branching when a given number of retries has been exceeded.

```
void KS_MetaHeuristic::tree_finished(const LocalTreeId& id, LocalTree& tree) {
    LB_MetaHeuristic::tree_finished(id, tree);
    if (0 == index.active_trees.size() && lb_enabled) {
        if (index.nodes_created_since_improvement <
            index.nodes_created_since_newtree) {
            create_tree();
            lb_tightening = false;
            lb_retries = 0;
        }
        else if (lb_tightening && lb_retries <
                 params.entry(KS_parameters::LB_MaxRetries)) {
            create_tree();
            ++lb_retries;
        }
        else
            lb_enabled = false;
    }
}
```

8.5 Finishing Touches

The Branch and Cut solver for multidimensional knapsack problems is now complete. For compiling the application, a properly patched installation of COIN/BCP is needed (see appendix A), and an adapted makefile. A makefile template can be taken from one of the COIN/BCP examples, found under *Examples/BAC* or *Examples/BranchAndCut* in the COIN directory. Note that all of the framework's sources have to be added to the makefile. The related part of the *Makefile.bc* file should look like the following:

```
USER_SRC =
USER_SRC += KS_init.cpp
USER_SRC += KS_lp.cpp
```

```
USER_SRC += KS_tm.cpp
USER_SRC += KS_metaheuristic.cpp

# LB sources
USER_SRC += LB_tm.cpp
USER_SRC += LB_lp.cpp
USER_SRC += LB_cut.cpp
USER_SRC += LB_user_data.cpp
USER_SRC += LB_init.cpp
USER_SRC += localtree.cpp
USER_SRC += localtreeid.cpp
USER_SRC += localtreemanager.cpp
USER_SRC += LB_metaheuristic.cpp
```

Chapter 9

Test Results

The multidimensional knapsack solver described in the previous chapter was used to undertake extensive testing of local branching performance. The test instances were taken from J.E. Beasley's OR Library [2] which provides 270 instances for the multidimensional knapsack problem. They are grouped by problem dimension into nine files, each containing 30 instances. The smallest problem size contains 100 variables and 5 constraints, the largest 500 variables subject to 30 constraints. Each file contains three groups of instances with tightness ratios of $\alpha = 0.25$, 0.5 , and 0.75. The tightness ratio defines how "tight" the resource limits are set, lower ratios define tighter resource limits.

Additional problems were taken from the Hearin Center for Enterprise Science [14]. This dataset contains eleven test instances originally used for the multiple knapsack problem, ranging from two instances with 100 variables and 15 constraints to an extremely large test instance with 2500 variables and 100 constraints.

9.1 Test Environment

All tests were executed on an Intel Pentium 4 with 2.8 GHz and 2 GB of RAM, running Linux with a 2.4.21 kernel. The LP engine used for testing was CPLEX 8.1.

The test application was configured to output status information at regular intervals, allowing a reporting tool described in appendix B to generate tables and plots comparing different configurations.

The running time is always given in CPU time for the COIN/BCP process as recorded by COIN/BCP's own timing statistics. This also includes time spent solving the LPs using the CPLEX solver.

When analyzing the results with respect to time (i.e. determining when a solution was found during the computation), the number of processed nodes is taken instead of the CPU time. As described in chapter 6, it is much easier for COIN/BCP to track the number of nodes during a computation than the CPU time spent so far. This simplification relies on the fact that the number of processed nodes does not vary significantly for a given test instance, even when using different parameters for local branching.

9.2 Test Results Overview

Summarizing the detailed results that will be presented in this chapter, three major observations were made:

- Local branching can find better solutions than standard branching early in the computation especially for large, complex test instances.
- For smaller, easier test instances local branching did not show such advantages or was inferior to standard branching.
- The settings for local branching, especially the value of k, the number of variables to be fixed, and the maximum number of nodes depend on the problem size. It was not possible to find a parameter configuration that delivers good results for all problem sizes (or even a rule of thumb to account for problem complexity).

Since local branching showed its benefits primarily with larger test instances, most of the detailed results will concentrate on larger knapsack problems.

9.2.1 Final Objective Comparison

The first method for comparing two configurations is by comparing the best feasible solution found in a given timespan. When both configurations found the same feasible solution (or at least two solutions with the same objective value), the configuration which processed less nodes to find this solution is considered better.

9.2.2 Online Performance

As an artificial measure for determining the efficiency of the algorithm an online performance rating was introduced. The basic idea is to plot the best objective value over time, and then calculate the sum of the area below:

$$\sum_{x=0}^{nodes_{max}} w(x)objval(x) \tag{9.1}$$

with $objval(x)$ being the interpolated final objective value for the number of processed nodes x and $nodes_{max}$ the total number of processed nodes. By adding a monotonically decreasing weight function, the online performance favors algorithms that find good solutions early in the computation, even if the final result is the same.

We chose a simple inverse exponential weight function,

$$w(x) = e^{\frac{-x}{nodes_{max}}}. \tag{9.2}$$

The online performance rating is calculated by summing up $objval(x)w(x)$ for $x = 1 \ldots nodes_{max}$ and scaling the result by $\frac{1}{nodes_{max}}$. When comparing different configurations, all results have to be processed over the same range. We set $nodes_{max}$ to the maximum number of processed nodes for the last improvement in the considered configurations.

Figure 9.1 shows a plot of the objective value for two different configurations on the left, and the corresponding online performance weight function on the right. While both configurations ultimately find almost equally good solutions, configuration 6 yields better solutions

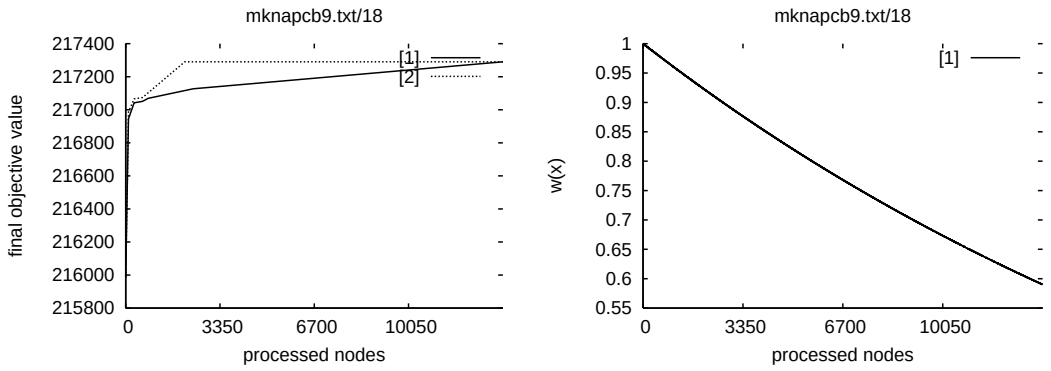

Figure 9.1: Final objective value and the corresponding online performance weights for a test instance.

earlier in the computation, so its online performance score is greater than that of configuration 1 (the former configuration is actually using local branching, while the latter is not.)

9.3 Local Branching Configurations

As explained in the previous chapters, there are several key parameters that influence the performance of local branching. In short, the user has to decide on the following parameters:

- The value of k for the Hamming distance constraint determines how many binary variables can flip inside a single local tree.
- Fixing some variables helps to reduce the problem complexity inside the tree defined by the chosen k value.
- Defining node limits on local trees avoids stagnation.
- By adapting the values of k and the numbers of variables to be fixed at run-time, the search can be narrowed or broadened at will.

The variables used for controlling these parameters are described in chapter 8. In the test results, the following abbreviated notation will be used:

- $k = k_{initial}, k_{scaling}, k_{\{min|max\}}$ defines the initial k value, the factor k is multiplied with when a tree is aborted, and the minimum (when the factor is less than 1) or maximum (when the factor is greater than 1) value for k. When only one value is given, k remains constant throughout the computation.
- *Variable fixing:* $f_{initial}, f_{increment}, f_{max}$ contains the number of variables fixed when a local tree using a new incumbent solution is started, and the increment and maximum values to be used when a tree is aborted and restarted with the same initial solution.
- *Maximum nodes* is the maximum number of nodes for a local tree, no node limit is used when this parameter is omitted.

- *Maximum nodes without improvement* declares the maximum number of nodes to be created in a local tree without finding an improved feasible solution. When this value is omitted, no such node limit is imposed on local trees.

All parameters depend on the size of the test instance, so no reasonable default values can be provided.

9.4 Short-Time Tests

The objective of the following tests is to accelerate the process of finding good solutions early in the computation, while the final objective is not of paramount importance. For these test runs, a time limit of 10 minutes per instance was used. Since there was no single configuration that succeeded in all presented problem instances, the results are separated by problem size. For the initial solution a greedy heuristic generated two solutions, the first using relative weights as in equation (7.5) as efficiency measure and the second using the first LP optimum. The better solution (regarding the objective value) was used as initial solution.

9.4.1 Local Branching and Node Limits

The first set of test runs uses standard local branching and for some instances node limits, but no cut generation. We start with examining the moderately sized mknapcb7 test instance collection from the OR Library [2].

Mknapcb7: 100 variables, 30 constraints

The following configurations have been tested for all 30 test instances of mknapcb7:

(1) Standard Branch and Cut.

(2) $k = 13$, variable fixing: 0.1, no node limit.

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 10000.

For these test instances, the local tree search without a node limit was superior to a local tree search with the same parameters, but with a limit on the total number of nodes per local tree. When comparing final objectives with the conditions described earlier in this chapter, configuration (2) wins against (1) with a clear advantage of $27 : 10$ (the numbers do not add up to 30 because of some ties.) For configuration (3), the direct comparison yields only a slight advantage of $19 : 17$ for local branching.

Introducing the online performance rating, (2) loses some of its advantage, but still showing a distinct advantage of $22 : 14$ when compared to (1). Configuration (3) stays at $19 : 17$, not showing a significant benefit from local branching.

Comparing the number of local trees, (2) mostly used only a single local tree with an average of 1.2 local trees, (3) created an average of 27.1 trees per test instance.

Figure 9.2 shows two sample graphs from this test series.

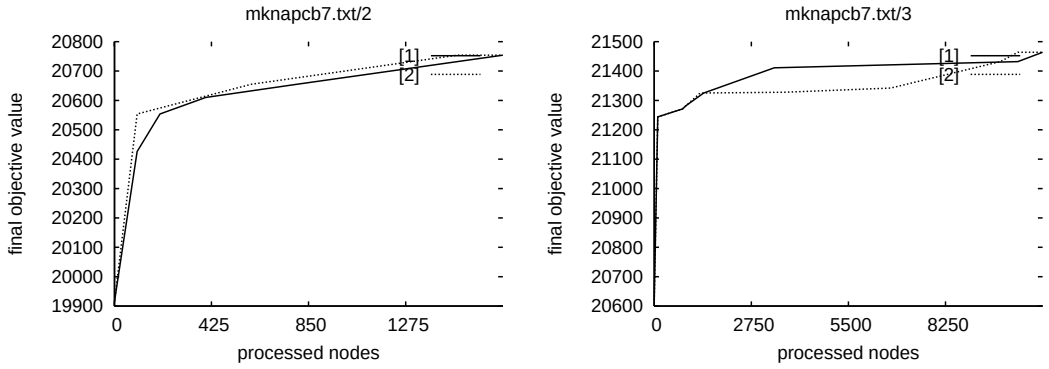

Figure 9.2: Two sample graphs for mknapcb7 showing a benefit for local branching on the left, and an advantage for normal Branch and Cut on the right.

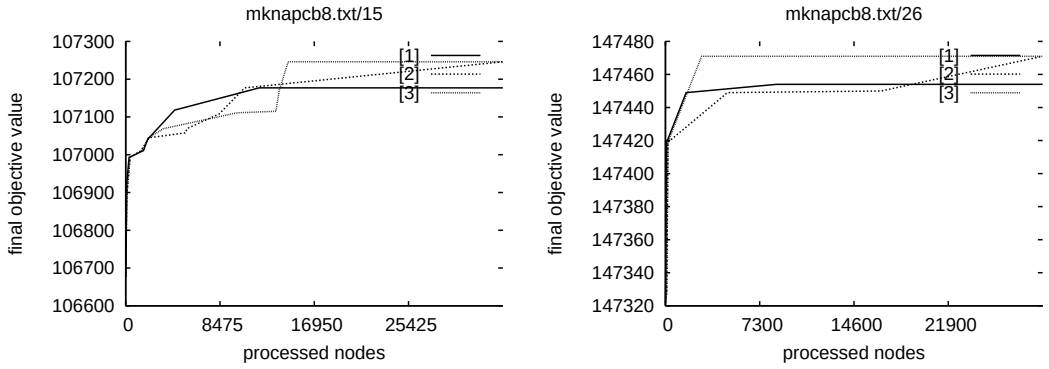

Figure 9.3: Two sample graphs for mknapcb8 showing the benefit of node limits for configuration (3) against the same configuration without node limit (2) and standard Branch and Cut (1).

Mknapcb8: 250 variables, 30 constraints

For the larger test instances of mknapcb8, imposing node limits proved to be more beneficial. A local branching setup similar to the previous configurations proved to be superior to standard Branch and Cut, especially when considering the online performance rating. The following configurations delivered the best results compared to standard Branch and Cut:

(1) Standard Branch and Cut.

(2) $k = 13$, variable fixing: 0.1, no node limit

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.5$, maximum nodes per tree: 5000

Comparing the final objective values, both (2) and (3) showed an advantage of $18 : 14$ against (1). When comparing the online performance rating, (2) exhibited a slight disadvantage of $15 : 17$, while (3) apparently benefited from the node limit and won clearly by $21 : 11$.

Figure 9.3 again shows two sample plots comparing the three parameter settings.

Instance	n	d	best	final objective	online performance	Wilcoxon
mknapcb7	100	30	(2)	27 : 10	22 : 14	$< 0.01\%$
mknapcb8	250	30	(3)	18 : 14	21 : 11	15.4%
mknapcb9	500	30	(3)	19 : 11	17 : 13	2.1%

Table 9.1: Summary table for the tested mknapcb problems. Each row represents 30 different instances with the given dimensions. The ratios given for final objective value and online performance compare the best local branching configuration to standard Branch and Cut.

Mknapcb9: 500 variables, 30 constraints

The largest class of test instances from the OR Library, mknapcb9, exhibited similar behavior regarding local branching. Compared to standard branch and cut, two configurations exhibited similar performance.

(1) Standard Branch and Cut.

(2) $k = 10$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 5000

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 1000

Regarding the final objective values, configuration (2) achieves a ratio of $18 : 12$ against standard Branch and Cut, reducing the node limit leads to a further minor improvement of $19 : 11$. The online performance ratings are a bit less favorable, showing a $17 : 13$ advantage for both local branching configurations. Configuration (2) created an average of 12.2 local trees per computation, while (3) used an average of 67.8 local trees.

Table 9.1 summarizes the results and also contains the result of a *Wilcoxon rank sum test*. It represents the error probability for the assumption that the first method performs on average better than the second. It is generated by creating two columns, one for each configuration, and setting a 1 where a configuration achieved the best result for a given test instance, and 0 otherwise. We do not compare the final objective values, because even when two configurations achieved the same value, we prefer the configuration that reached it with fewer processed nodes.

A Wilcoxon score close to 0 means that the the local branching configuration is very likely to be better than the standard Branch and Cut algorithm, a value close to 0.5 means that no significant difference exists. For mknapcb7 and mknapcb9 the Wilcoxon test clearly indicates that the local branching results are better than the standard Branch and Cut results, while the result for mknapcb8 (0.15) is less clear.

The detailed test results for all test instances are given in tables 9.2, 9.3, and 9.4.

MK-gk11: 2500 variables, 100 constraints

The huge eleventh test instance from the problems taken from the Hearin Center for Enterprise Science [14] shows clear advantages for most local branching configurations. The tested configurations are:

(1) Standard Branch and Cut.

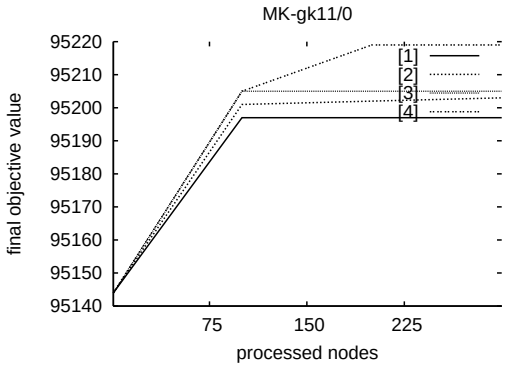

Figure 9.4: Clear advantages for local branching in test instance MK-gk11.

(2) $k = 5$, variable fixing: $0.05, 0.2, 0.5$, maximum nodes per tree: 250

(3) $k = 20$, no variable fixing, no node limit.

(4) $k = 10$, no variable fixing, maximum number of nodes: 1000

All local branching configurations both showed faster convergence and better final objective values. The plot of the objective value is given in figure 9.4.

9.4.2 Cut Generation

The idea of cut generation is to find valid inequalities that are violated by the current LP optimum as described in section 2.3, thus decreasing the LP optimum and increasing the chances to prune a subproblem. The size of the search tree can be reduced significantly at the expense of computationally expensive cut generation.

In our test results, cut generation did not lead to a significant advantage for local branching in comparison to the standard Branch and Cut algorithm, instead it compensated the advantage of local branching. We used COIN/CGL's generic cut generators, the most efficient proved to be the knapsack cover cut generator. Others, like the Gomory cut generator, did not improve the results but slowed down the computation.

Mknapcb7: 100 variables, 30 constraints

The following parameter configurations have been tested:

(1) Standard Branch and Cut.

(2) $k = 13$, variable fixing: 0.1, no node limit.

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 1000.

Without cut generation, configuration (2) beat standard Branch and Cut by $27 : 10$. With cut generation, the standard algorithm reaches a tie result of $18 : 18$ both against (2) and (3). Regarding the online performance rating, local branching gains a slight advantage of $18 : 16$ for configuration (2) and $19 : 15$ for (3).

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**21946**	**21946**	**21946**	**13719.11**	**13719.11**	**13719.11**	0	2	29
1	21716	**21716**	21716	13700.76	**13710.87**	13699.94	0	1	25
2	20754	**20754**	**20754**	13096.68	**13102.44**	**13102.44**	0	1	27
3	21464	**21464**	21464	**13532.09**	13511.85	13524.59	0	1	26
4	21844	**21844**	**21844**	13797.81	**13809.01**	**13809.01**	0	1	24
5	22176	22176	**22176**	14005.53	14005.90	**14006.11**	0	1	26
6	21799	**21799**	**21799**	13718.36	**13723.17**	**13723.17**	0	1	26
7	**21397**	21327	21327	**13493.88**	13472.05	13473.10	0	1	25
8	22471	22475	**22482**	14187.96	14201.18	**14210.38**	0	1	24
9	**20983**	**20983**	20983	**13230.11**	13230.08	13223.38	0	1	27
10	40691	40691	**40767**	25722.46	25723.66	**25742.90**	0	1	25
11	41308	**41308**	41304	26108.12	**26108.20**	26106.94	0	1	25
12	**41630**	**41630**	**41630**	**26304.63**	**26304.63**	**26304.63**	0	1	25
13	**41041**	**41041**	**41041**	**25937.69**	**25937.69**	**25937.69**	0	1	27
14	40889	**40889**	40889	25842.11	**25842.93**	25838.43	0	1	24
15	41028	**41058**	41022	**25934.05**	25921.21	25915.76	0	1	25
16	**41062**	41038	41038	**25957.11**	25936.80	25935.54	0	1	25
17	42719	**42719**	42719	26976.12	**26979.48**	26979.01	0	1	25
18	**42230**	**42230**	**42230**	**42230.00**	**42230.00**	**42230.00**	0	2	49
19	**41700**	**41700**	**41700**	**41700.00**	**41700.00**	**41700.00**	0	1	26
20	57494	57494	**57494**	36319.24	36320.82	**36322.32**	0	1	28
21	60027	**60027**	60026	**37944.30**	37943.28	37943.50	0	1	26
22	58052	58015	**58052**	36677.22	36670.90	**36688.60**	0	1	26
23	60776	**60776**	**60776**	38415.55	**38415.96**	**38415.96**	0	2	34
24	**58884**	**58884**	**58884**	**37214.71**	**37214.71**	**37214.71**	0	2	26
25	60011	**60011**	**60011**	**37919.29**	37910.45	37910.45	0	2	30
26	58132	58132	**58132**	36737.55	36737.63	**36741.17**	0	1	27
27	59064	**59064**	**59064**	37325.45	**37333.21**	**37333.21**	0	2	29
28	58975	**58975**	58975	37277.47	**37278.91**	37276.15	0	1	25
29	60603	**60603**	**60603**	38295.77	**38299.39**	**38299.39**	0	1	27

Table 9.2: All results for mknapcb7 using a single initial solution ($n = 100$, $d = 30$.)

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**56824**	56824	56824	**35914.54**	35910.64	35910.64	0	1	25
1	58310	58332	**58364**	36842.34	36860.57	**36878.71**	0	1	23
2	56498	**56508**	56493	**35706.59**	35706.18	35701.40	0	1	24
3	**56930**	**56930**	**56930**	**35989.96**	**35989.96**	**35989.96**	0	1	24
4	**56629**	56601	56629	**35783.77**	35748.73	35773.20	0	1	23
5	**57146**	**57146**	**57146**	**35983.42**	**35983.42**	**35983.42**	0	1	24
6	56206	56180	**56219**	35513.51	35510.10	**35524.30**	0	1	22
7	56392	56413	**56413**	35636.01	**35650.11**	35644.80	0	1	23
8	**57429**	57413	57429	36271.66	36272.94	**36278.42**	0	1	22
9	56447	**56447**	**56447**	**35659.16**	35658.13	35658.13	0	1	23
10	**107746**	107746	107732	**68108.02**	68106.39	68088.89	0	1	23
11	**108336**	108335	108335	68469.79	68459.16	**68469.89**	0	1	23
12	**106442**	106375	106415	**67257.55**	67244.95	67256.61	0	1	22
13	106780	106766	**106786**	67498.51	67490.14	**67499.44**	0	1	22
14	107349	107414	**107414**	67856.07	**67885.23**	67880.51	0	1	22
15	107177	107246	**107246**	67729.09	67731.84	**67736.79**	0	1	22
16	106294	106241	**106297**	67167.78	67144.31	**67181.55**	0	1	23
17	**103998**	103998	103977	**65727.95**	65727.41	65724.06	0	1	24
18	106736	106751	**106758**	67452.37	67460.47	**67468.22**	0	1	22
19	105675	**105716**	105681	66792.13	**66814.53**	66798.56	0	1	21
20	**150097**	150081	150097	94866.09	94863.58	**94866.84**	0	1	25
21	**149907**	149881	149854	**94742.37**	94729.76	94706.55	0	1	24
22	152971	**152973**	152960	**96697.38**	96692.10	96687.17	0	1	27
23	153177	153177	**153190**	96824.54	**96825.11**	96824.86	0	1	25
24	150287	150287	**150287**	94938.43	94942.16	**94947.21**	0	1	26
25	148520	**148544**	148544	93871.33	**93891.89**	93889.45	0	1	25
26	147454	147471	**147471**	93209.11	93206.87	**93218.07**	0	1	26
27	152817	**152912**	152877	96585.86	**96656.17**	96635.54	0	1	26
28	149570	**149570**	149554	94533.41	**94541.43**	94538.37	0	1	24
29	149586	**149595**	149586	94553.56	94554.05	**94554.53**	0	1	24

Table 9.3: All results for mknapcb8 using a initial solution ($n = 250$, $d = 30$.)

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**115850**	115841	115809	73184.48	**73215.35**	73192.49	0	12	70
1	114623	114667	**114701**	72420.52	72456.21	**72473.47**	0	12	69
2	116541	**116661**	116607	73647.29	**73729.35**	73693.17	0	14	73
3	115096	115128	**115152**	72753.14	72760.71	**72784.41**	0	11	71
4	116266	116316	**116385**	73495.78	73519.30	**73553.33**	0	14	66
5	115563	115584	**115600**	73032.02	73048.46	**73065.10**	0	12	70
6	113928	**113982**	113928	72006.14	**72038.81**	72006.60	0	11	70
7	114190	114137	**114190**	72161.72	72126.01	**72170.82**	0	14	78
8	**115419**	115133	115419	**72943.75**	72771.30	72822.16	0	12	64
9	**116988**	116891	116929	**73908.53**	73881.07	73888.93	0	12	67
10	217925	217995	**217995**	137747.72	137797.39	**137801.07**	0	12	75
11	214517	**214626**	214626	135580.86	**135609.00**	135589.10	0	12	65
12	215835	**215844**	215844	136399.86	**136411.96**	136372.69	0	12	58
13	**217827**	217827	217805	137664.20	**137671.45**	137665.81	0	11	72
14	**215559**	215515	215535	**136263.63**	136212.21	136221.81	0	10	62
15	215697	**215722**	215717	136337.13	**136366.40**	136351.87	0	11	60
16	215772	**215780**	215780	136388.68	**136398.82**	136381.53	0	12	59
17	**216419**	216366	216341	**136784.92**	136763.70	136744.46	0	11	66
18	217290	217196	**217290**	137312.79	137278.83	**137348.63**	0	11	73
19	214624	214592	**214633**	**135652.21**	135632.98	135649.19	0	11	65
20	301643	**301643**	301627	**190667.63**	190656.90	190667.01	0	12	76
21	299957	**299987**	299945	189579.82	**189618.86**	189577.88	0	14	62
22	304985	304985	**304994**	192790.68	192790.07	**192792.68**	0	14	76
23	301854	301891	**301955**	190787.31	**190809.87**	190803.87	0	12	58
24	304411	**304413**	304350	192420.91	**192423.62**	192368.46	0	12	68
25	296891	296891	**296959**	187672.97	187664.53	**187705.63**	0	13	74
26	303261	303262	**303270**	191663.09	**191687.73**	191671.50	0	14	67
27	306890	**306937**	306892	**193995.36**	193976.43	193981.49	0	12	70
28	**303111**	303088	303083	**191592.30**	191588.20	191565.66	0	15	65
29	300479	**300499**	300439	**189930.56**	189896.47	189905.03	0	11	64

Table 9.4: All results for mknapcb9 using a single initial solution ($n = 500$, $d = 30$.)

9.4.3 Multiple Initial Solutions

Creating multiple local trees in the beginning of the computation can help to improve the initial performance of the local branching algorithm. The processor time is distributed over several local trees, preferring those with better nodes (according to the tree search strategy, i.e. those with better bounds.)

As described in chapter 8, three different initial solutions were used:

- The feasible solution generated heuristically from the first LP result.
- A solution returned by a greedy heuristic using the relative weight as an efficiency measure.
- A solution returned by the same heuristic including the weight distribution as an efficiency measure.

Compared to local branching with a single initial solution the results improved considerably. For the mknapcb7 test instances, the three initial local trees contributed equally to the best found solution, that is, the initial trees were of roughly the same size. For the mknapcb8 and mknapcb9 instances, the tree based on the first LP result was most often superior to the trees based on efficiency measures, meaning that the latter two trees were often completed after a few nodes. Apparently the greedy heuristics with efficiency values worked better for the smaller mknapcb7 instances than for the more complex mknapcb8 and mknapcb9 instances.

Mknapcb7: 100 variables, 30 constraints

The following configurations have been tested:

(1) Standard Branch and Cut.

(2) $k = 13$, variable fixing: 0.1, no node limit.

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 10000.

Regarding the final objective values, both local branching configurations showed an $24 : 10$ advantage to standard Branch and Cut. While this result is similar to what local branching with a single initial solution achieved, the pseudo-concurrent tree exploration shows more benefits when looking at the online performance rating. Both local branching configurations achieved a clear advantage of $25 : 9$ compared to local branching, which is considerably better than the results using a single initial solution.

Mknapcb8: 250 variables, 30 constraints

As in section 9.4.1, the following configurations have been tested for mknapcb8:

(1) Standard Branch and Cut.

(2) $k = 13$, variable fixing: 0.1, no node limit

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.5$, maximum nodes per tree: 5000

Instance	n	d	best	final objective	online performance	Wilcoxon
mknapcb7	100	30	(2)	24 : 10	25 : 9	0.02%
mknapcb8	250	30	(2)	22 : 8	24 : 6	0.02%
mknapcb9	500	30	(3)	22 : 8	25 : 5	0.02%

Table 9.5: Summary table for the tests using multiple initial solutions, including a Wilcoxon probability score for the assumption that the final objective values of standard Branch and Cut are better than the given local branching configuration.

Comparing the final objective values, configuration (2) showed an advantage of $22 : 8$ against standard Branch and Cut, (3) had an advantage of $21 : 9$. Regarding online performance, (2) showed a clear advantage of $24 : 6$ and (3) an advantage of $26 : 4$ compared to standard Branch and Cut.

Mknapcb9: 500 variables, 30 constraints

The following configurations were tested:

(1) Standard Branch and Cut.

(2) $k = 10$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 5000.

(3) $k = 13$, variable fixing: $0.1, 0.1, 0.8$, maximum nodes per tree: 1000.

Both (2) and (3) showed clearly superior results to (1), with (2) showing a slight advantage of $17 : 13$ and (3) beating standard Branch and Cut by $22 : 8$. The online performance ratings clearly favor the local branching configurations: (2) beats (1) by $24 : 6$, (3) beats (1) by $25 : 5$. Compared with the same configuration without multiple initial solutions, (2) exhibited an advantage of $19 : 11$ for the final objective value and $27 : 3$ for the online performance rating.

Table 9.5 summarizes the results for these test runs. The detailed results for all test instances of mknapcb7, mknapcb8 and mknapcb9 are given in tables 9.6, 9.7, and 9.8. Bold values indicate the best result for a single instance. Note that it is possible for more than one configuration to achieve the "best" result.

9.5 Long Runs

The test runs described in the last section are useful for testing the short-time heuristical behavior a large variety of local branching configurations. Testing the instances of the OR library [2] with longer running times (up to one hour) did not reveal significantly different behavior. However, the very large eleventh instance of the second set of test instances [14] with 2500 variables and 100 constraints was an interesting target for examining long-run behavior. The huge core matrix dramatically slows down the LP solver, affecting the significance of the results of a 10 minute test run. Increasing the CPU time to 2 hours showed interesting results: local branching extended its lead (with unmodified parameters), standard branch and cut was clearly inferior to *all* tested configurations.

Figure 9.5 shows the final objective plots for the following configurations:

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**21946**	**21946**	**21946**	**13719.11**	**13719.11**	**13719.11**	0	3	27
1	21716	**21716**	**21716**	13698.34	**13720.04**	**13720.04**	0	3	28
2	20754	**20754**	**20754**	13096.68	**13114.77**	**13114.77**	0	3	31
3	21464	**21464**	**21464**	13517.23	**13566.96**	**13566.96**	0	3	25
4	**21844**	21814	21814	**13803.20**	13792.24	13792.50	0	3	20
5	22176	**22176**	**22176**	14005.53	**14018.62**	**14018.62**	0	3	21
6	21799	**21799**	**21799**	13718.36	**13758.35**	**13758.35**	0	3	24
7	21397	21327	**21397**	13493.88	13483.27	**13503.57**	0	3	24
8	22471	22493	**22525**	14187.96	14218.78	**14229.36**	0	3	16
9	20983	**20983**	**20983**	13223.38	**13262.97**	**13262.97**	0	3	29
10	40691	**40767**	40767	25722.66	25737.27	**25745.08**	0	3	19
11	41308	**41308**	41304	26108.12	**26110.50**	26109.40	0	3	23
12	41630	**41630**	**41630**	26304.63	**26313.70**	**26313.70**	0	3	24
13	41041	**41041**	**41041**	25909.87	**25925.38**	**25925.38**	0	3	30
14	**40889**	40889	40872	25830.92	**25842.90**	25838.46	0	3	23
15	41028	**41058**	41058	25934.27	**25937.79**	25931.24	0	3	23
16	**41062**	41038	41062	**25957.11**	25942.97	25956.31	0	3	37
17	42719	**42719**	**42719**	26972.46	**27001.34**	**27001.34**	0	3	21
18	**42230**	**42230**	**42230**	**42230.00**	**42230.00**	**42230.00**	0	4	30
19	**41700**	**41700**	**41700**	**41700.00**	**41700.00**	**41700.00**	0	3	22
20	57494	**57494**	**57494**	36319.24	**36343.09**	**36343.09**	0	3	29
21	**60027**	60026	60026	**37944.30**	37943.18	37943.18	0	3	48
22	**58052**	58052	58025	**36682.74**	36681.73	36671.01	0	3	19
23	60776	**60776**	**60776**	38415.55	**38418.18**	**38418.18**	0	4	25
24	**58884**	**58884**	**58884**	**37214.71**	**37214.71**	**37214.71**	0	4	28
25	60011	**60011**	**60011**	37919.29	**37931.77**	**37931.77**	0	4	27
26	58132	**58132**	**58132**	36737.55	**36748.63**	**36748.63**	0	3	30
27	59064	**59064**	**59064**	37325.45	**37336.68**	**37336.68**	0	3	26
28	58975	**58975**	58975	37277.47	**37280.14**	37279.40	0	3	20
29	60603	**60603**	**60603**	38295.77	**38307.77**	**38307.77**	0	3	24

Table 9.6: All results for mknapcb7 using multiple initial solutions ($n = 100$, $d = 30$.)

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**56824**	56824	56824	**35914.54**	35912.01	35912.54	0	3	16
1	58310	**58520**	**58520**	36842.34	**36990.73**	**36990.73**	0	3	17
2	56498	**56553**	56493	35708.82	**35719.57**	35713.55	0	3	17
3	56930	**56930**	**56930**	35989.96	**35990.17**	**35990.17**	0	3	17
4	**56629**	56601	56629	35783.77	35782.33	**35796.56**	0	3	15
5	57146	**57146**	**57146**	35983.42	**36058.77**	**36058.77**	0	3	16
6	56206	56253	**56253**	35513.51	35543.21	**35553.50**	0	3	15
7	56392	**56457**	56448	35636.01	35672.34	**35674.56**	0	3	16
8	57429	57429	**57433**	36271.66	**36299.70**	36293.59	0	3	14
9	56447	**56447**	**56447**	35659.16	**35683.55**	**35683.55**	0	3	16
10	107746	**107746**	**107746**	68108.02	**68110.74**	**68110.74**	0	3	17
11	108336	108335	**108352**	68469.79	68482.54	**68489.44**	0	3	16
12	**106442**	106440	106415	67257.55	**67280.00**	67265.87	0	3	15
13	106780	106790	**106806**	67498.51	67507.29	**67516.62**	0	3	16
14	107349	107374	**107414**	67856.07	67874.43	**67893.45**	0	3	14
15	107177	**107246**	**107246**	67729.09	**67789.77**	**67789.77**	0	3	16
16	106294	106283	**106305**	67167.78	67181.78	**67189.00**	0	3	14
17	103998	**103998**	103995	65727.95	**65739.76**	65737.10	0	3	15
18	106736	106758	**106800**	67452.37	67480.32	**67501.87**	0	3	15
19	105675	**105742**	105723	66792.13	**66838.93**	66827.20	0	3	15
20	**150097**	150073	150096	94866.09	94865.84	**94866.62**	0	3	17
21	149907	149862	**149907**	94742.37	94720.42	**94748.96**	0	3	17
22	152971	**152973**	152971	**96697.38**	96685.57	96690.15	0	3	18
23	153177	153177	**153190**	96824.54	96830.61	**96838.22**	0	3	18
24	150287	**150287**	**150287**	94938.43	**95000.95**	**95000.95**	0	3	19
25	148520	**148544**	**148544**	93871.33	**93901.04**	**93901.04**	0	3	18
26	147454	**147471**	147454	93209.11	**93216.50**	93204.43	0	3	15
27	152817	**152877**	152877	96585.86	**96629.50**	96623.47	0	3	17
28	**149570**	149568	149565	**94546.38**	94545.42	94542.68	0	3	16
29	149586	**149595**	**149595**	94553.56	**94565.92**	**94565.92**	0	3	18

Table 9.7: All results for mknapcb8 using multiple initial solutions ($n = 250$, $d = 30$.)

	final objective value			online performance			number of local trees		
instance	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
0	**115850**	115824	115838	73176.36	73214.09	**73227.44**	0	8	48
1	114623	114699	**114701**	72420.13	72488.98	**72497.70**	0	8	50
2	116541	116661	**116661**	73647.29	**73744.23**	73737.49	0	7	42
3	115096	115180	**115206**	72752.50	72799.13	**72816.38**	0	7	43
4	116266	116321	**116385**	73495.78	73530.94	**73565.28**	0	8	39
5	115563	115604	**115741**	73026.80	73060.22	**73095.36**	0	7	37
6	113928	**113979**	113928	72006.14	**72044.76**	72019.41	0	8	42
7	**114190**	114168	114174	72161.72	**72166.85**	72163.73	0	8	42
8	**115419**	115198	115419	**72943.75**	72817.93	72918.72	0	8	41
9	**116988**	116952	116886	73908.53	**73921.92**	73891.15	0	8	39
10	217925	217983	**218042**	137742.72	137792.81	**137831.38**	0	8	49
11	214517	**214626**	214626	135580.86	**135671.85**	135641.70	0	9	44
12	215835	**215885**	215854	136383.17	**136454.78**	136439.97	0	8	47
13	217827	217769	**217827**	137664.20	137654.34	**137697.27**	0	8	49
14	**215559**	215557	215548	**136263.27**	136258.66	136252.59	0	8	48
15	215697	**215726**	215718	136337.13	**136370.18**	136364.35	0	8	47
16	215772	215791	**215792**	136386.68	**136407.75**	136398.33	0	9	37
17	216419	216403	**216419**	136777.53	136788.67	**136808.10**	0	8	49
18	217290	217290	**217312**	137312.79	**137350.16**	137340.43	0	8	39
19	214624	214581	**214633**	135652.21	135641.87	**135661.50**	0	8	42
20	**301643**	301627	301583	190667.63	**190668.82**	190643.19	0	9	49
21	299957	**299987**	299984	189558.15	**189630.07**	189583.03	0	9	44
22	304985	**305002**	305002	192790.68	**192804.37**	192803.92	0	9	51
23	301854	302001	**302004**	190787.31	190862.57	**190896.06**	0	8	44
24	**304411**	304380	304398	**192419.64**	192406.11	192397.45	0	9	41
25	296891	296892	**296986**	187672.97	187673.13	**187727.07**	0	8	46
26	303261	303240	**303285**	191663.09	191689.34	**191710.94**	0	8	44
27	306890	**306911**	306910	193995.36	**194005.21**	194001.03	0	9	41
28	**303111**	303092	303090	191583.54	**191600.27**	191594.76	0	9	45
29	300479	300444	**300488**	189930.56	189920.25	**189945.99**	0	8	36

Table 9.8: All results for mknapcb9 using multiple initial solutions ($n = 500$, $d = 30$.)

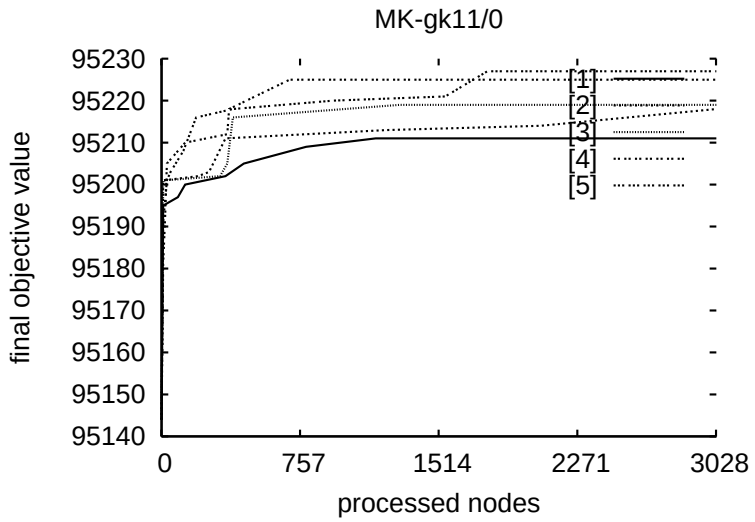

Figure 9.5: The final objective value for MK-gk11, using a time limit of 2 hours.

(1) Standard Branch and Cut.

(2) $k = 5$, variable fixing: $0.05, 0.2, 0.5$, maximum nodes per tree: 250

(3) $k = 5$, variable fixing: $0.05, 0.2, 0.5$, maximum nodes per tree: 500

(4) $k = 10$, variable fixing: $0.05, 0.2, 0.5$, maximum nodes per tree: 500

(5) $k = 20$, variable fixing: $0.05, 0.2, 0.5$, maximum nodes per tree: 1000

Chapter 10

Summary and Outlook

This thesis described the implementation of a generic local branching framework based on the open source COIN/BCP Branch, Cut and Price library. Local branching is a local search heuristic that is well suited for integration in existing integer programming solvers. The framework provides the possibility to augment COIN/BCP programs with local branching search capabilities. Several extensions to the standard local branching algorithm were implemented: pseudo-concurrent exploration of multiple local trees, aborting local trees, and search space tightening through variable fixing.

An encapsulated metaheuristic class offers means for a clean implementation of local branching metaheuristics without touching COIN/BCP's internals. Rich statistical data about the current state of the local branching algorithm is provided by the framework. Methods for creating new trees, terminating existing trees or modifying the local branching search parameters are also provided.

As a sample application, a Branch and Cut solver for the multidimensional knapsack problem was used to demonstrate the application of the local branching framework and to research the effects of local branching.

The results for the multidimensional knapsack problem were promising: local branching showed better convergence, especially in the early stages of the computation, and showed significant benefits for large, complex test instances. By guiding the Branch and Cut solver through neighborhood search and fixing of variables, local branching allows to find better results earlier in the computation, which also leads to a reduction of the search tree complexity in the later stages. However, the benefit for relatively small test instances was less clear.

The local branching framework was designed in a way that facilitates embedding local branching as a local search metaheuristic in another, higher-level search algorithm. This is the main area where future work could be expected, to use the heuristical characteristics of local branching to improve other search algorithms not based on Branch and Cut. In general, any algorithm that involves some kind of local neighborhood search lends itself to the integration of local branching. For example, an evolutionary algorithm could use the framework to generate new, better solutions based on especially promising candidates.

Appendix A

COIN/BCP patches

The local branching framework requires some small patches to the COIN/BCP source. The patches add the following functionality to COIN/BCP:

- Support for user-defined messages between LP and TM modules has been added.
- A special slot for the normal tree root node is provided in the candidate queue. This is necessary because the normal root node must be used whenever a new local tree is started.
- There is no way for the COIN/BCP user classes to catch all pruned nodes. Pruned nodes are now added to a list that is available to the framework's tree manager.
- When a local tree is aborted, all nodes must be removed from the candidate list. Since the nodes are potentially scattered over the candidate list, and the candidate list can contain millions of nodes, explicitly deleting all nodes may be ineffective. Instead, a list of local tree identification numbers is kept and nodes from these trees are pruned immediately instead of being returned to the tree manager.

All filenames in this section are relative to the COIN/BCP root directory.

A.1 Adding User-Defined Messages

In order to support user-defined messages between LP and TM modules, we have to add an unique message tag for user messages and stubs for packing and unpacking routines. We start by adding two new message tags to the BCP_message_tag enumeration in *include/BCP_message_tag.hpp* (written in bold face):

```
(...)
    /** The message contains the description of a variable. */
    BCP_Msg_VarDescription,          // VG / VP -> LP
    /** No more (improving) variables could be found. (Message body is
        empty.) */
    BCP_Msg_NoMoreVars,              // VG / VP -> LP
    BCP_Msg_UserMessageToLp,
    BCP_Msg_UserMessageToTm
};
```

Then we add virtual method declarations of unpack_user_message() to BCP_tm_user and BCP_lp_user by adding the following lines to the corresponding class definitions in *include/BCP_lp_user.hpp* and *include/BCP_tm_user.hpp*:

```
virtual void
unpack_user_message(BCP_lp_prob& prob, BCP_buffer& buf);
```

We also provide a default implementation that throws an exception when called in *LP/BCP_lp_user.cpp* and *TM/BCP_tm_user.cpp*:

```
void
BCP_tm_user::unpack_user_message(BCP_tm_prob& prob, BCP_buffer& buf) {
  throw BCP_fatal_error(
"BCP_tm_user::unpack_user_message() invoked but not overridden!\n");
}
```

The implementation for BCP_lp_user is identical except for the class name. The last step to be taken is to call these handlers from the tree manager and LP module message processing functions. For the tree manager, we add the following block to the switch statement of *BCP_tm_prob::process_message()* in *TM/BCP_tm_msgproc.cpp*:

```
case BCP_Msg_UserMessageToTm:
  user->unpack_user_message(*this, msg_buf);
  msg_buf.clear();
  break;
```

Similarly, we add the following code to the switch statement of *BCP_lp_prob::process_message()* in *LP/BCP_lp_msgproc.cpp*:

```
case BCP_Msg_UserMessageToLp:
  user->unpack_user_message(*this, msg_buf);
  msg_buf.clear();
  break;
```

A.2 Extending the Candidate List

When local trees can be spawned before the previous tree terminated, the normal root node (the sibling of the local tree root) has to be extracted from the candidate list. The easiest and fastest way to achieve this goal is to store the normal root node in an extra variable and modify the methods to insert and retrieve items.

A.2.1 include/BCP_tm_node.hpp

We start with modifying *include/BCP_tm_node.hpp*. We have to add two member variables to *BCP_node_queue* which is used to store the candidate list.

```
/** root node of the "normal" tree, to be used when all local trees
    are processed or a new tree should be opened */
BCP_tm_node* normal_root_node;
```

```
/** use normal_root_node instead of a candidate from the queue the
    next time top() is called */
bool use_normal_root_node;
```

Then we add a new parameter to BCP_node_queue::insert that permits to insert a normal root node without using the extra slot. This is used when the normal root node is the last remaining node and should be returned to the candidate list. By setting a default value, existing calls to this function do not need to be modified.

```
/** Insert a new node into the queue. */
void insert(BCP_tm_node* node, bool replace_normal_root_node = true);
```

We also slightly modify the inline functions *empty()* and *top()* to account for the normal root variable.

```
/** Return whether the queue is empty or not */
inline bool empty() const { return !normal_root_node && _pq.size() == 1; }

/** Return the top member of the queue */
BCP_tm_node* top() const {
        return ((normal_root_node && use_normal_root_node) ||
        (normal_root_node && _pq.size() == 1))
        ? normal_root_node : _pq[1];
}
```

The last modification correctly initializes the new variables in the constructor.

```
BCP_node_queue(BCP_tm_prob& p): _p(q), _pq(),
  normal_root_node(0), use_normal_root_node(false) { _pq.push_back(NULL); }
```

A.2.2 include/BCP_tm_node.cpp

We also have to change the implementations of the *insert* and and *pop* methods of the BCP_node_queue class. Since we have to access user data objects, we have to include the framework's user data header. By using a compile-time flag for applications that use the local branching framework, the COIN source remains usable for other applications.

```
#ifdef COIN_LB
        #include "LB_user_data.hpp"
#endif
```

The *pop()* method removes a node from the head of the priority queue. When the queue is has only one element left, the normal root node is re-inserted to the list. If the normal root node has been used (by setting *use_normal_root_node* to true), it is deleted when pop() is called.

```
void
BCP_node_queue::pop()
{
        if (use_normal_root_node && normal_root_node) {
                normal_root_node = 0;
                return;
```

```
        }
        if (normal_root_node && _pq.size() <= 2) {
                // reinsert normal_root_node when the last element is popped
                insert(normal_root_node, false);
                normal_root_node = 0;
                use_normal_root_node = false;
        }

(...)
```

In the insert() method, we have to detect normal root nodes and store them in the extra variable instead of the normal candidate list. By using the COIN_LB flag again, the LB_user_data cast does not conflict with other COIN/BCP applications.

```
void
BCP_node_queue::insert(BCP_tm_node* node, bool replace_normal_root_node)
{
        #ifdef COIN_LB
        if (node->user_data() && replace_normal_root_node) {
                const LB_user_data* ud =
                        dynamic_cast<const LB_user_data*> (node->user_data());
                if (ud && LB_user_data::UD_NormalRoot == ud->type) {
                        normal_root_node = node;
                        return;
                }
        }
        #endif
(...)
```

A.2.3 TM/BCP_tm_functions.cpp

Another small modification is necessary in the static helper function *BCP_tm_start_one_node()*. When the normal root node should be returned, it is returned without further checking (e.g. if it should be pruned). This way the tree manager can recognize when the normal root node is pruned without further modifications (in this case, the node would be pruned by a LP process). Also, when a node was pruned because of the global upper bound, it is added to the pruned_nodes list that is described in the next section.

```
(...)
        p.ub() * (1 - p.param(BCP_tm_par::TerminationGap_Relative)))
        process_this = false;

if (p.candidates.use_normal_root_node) {
        process_this = true;
        p.candidates.use_normal_root_node = 0;
}
if (process_this)
        break;
if (desc->indexed_pricing.get_status() == BCP_PriceNothing ||
        p.current_phase_colgen == BCP_DoNotGenerateColumns_Fathom) {
        next_node->status = BCP_PrunedNode_OverUB;

        p.pruned_nodes.push_back(next_node);
```

(...)

A.3 Counting Pruned Nodes

We start by adding a new public member to the *BCP_tm_prob* class. It is used to store nodes that have been pruned. Since even pruned nodes are never deleted from memory, the tree manager can access this list without further restrictions. The tree manager can also empty the list when the nodes have been processed.

```
/** Pruned nodes are stored in this list - may be cleared when no longer needed */
BCP_vec<BCP_tm_node*> pruned_nodes;
```

There are two more places where nodes may be pruned inside the tree manager.

A.3.1 TM/BCP_tm_msg_node_rec.cpp

Among other things, the method BCP_tm_unpack_branching_info() prunes child nodes generated from a branching object when necessary. We add those nodes to our pruned_nodes() list.

```
(...)
case BCP_FathomChild:
        child->status = BCP_PrunedNode_Discarded;
        p.pruned_nodes.push_back(child);
        break;
(...)
```

A.3.2 TM/BCP_tm_msgproc.cpp

The tree manager also receives pruned nodes from LP processes. These nodes are also added to pruned_nodes.

```
(...)
case BCP_Msg_NodeDescription_Discarded:
case BCP_Msg_NodeDescription_OverUB_Pruned:
case BCP_Msg_NodeDescription_Infeas_Pruned:
        node = BCP_tm_unpack_node_no_branching_info(*this, msg_buf);
        pruned_nodes.push_back(node);
(...)
```

With these modifications, the tree manager is able to track the number of active nodes for all local trees. It uses the local tree identification number stored in the user data of the pruned nodes to update the node numbers of the corresponding local tree.

A.4 Aborting Local Trees

For aborting local trees, we store a set of local tree identification numbers in the candidate list. In the tree manager method responsible for finding a new subproblem for a LP process, we simply discard nodes that are in this set of terminated trees.

A.4.1 include/BCP_tm_node.hpp

We have to include two additional headers, again wrapped in a precompiler conditional.

```
#ifdef COIN_LB
        #include "localtreeid.hpp"
        #include <set>
#endif
```

Then we add a new public member to BCP_node_queue:

```
#ifdef COIN_LB
        /** LocalTreeId values of trees to be terminated
        (= to be pruned by BCP_node_queue::pop and BCP_node_queue::top) */
        std::set<LocalTreeId> terminate_ids;
#endif
```

A.4.2 TM/BCP_tm_functions

In *BCP_tm_start_one*, we modify the head of the main loop, the updates marked with bold face.

```
(...)
while (true){
        if (p.candidates.empty()) return BCP_NodeStart_NoNode;
        next_node = p.candidates.top();
        p.candidates.pop();
        desc = next_node->_desc;

        bool process_this = true;
#ifdef COIN_LB
        const LB_user_data* lb_ud =
                dynamic_cast<const LB_user_data*> (next_node->user_data());
        if (lb_ud && p.candidates.terminate_ids.end() !=
            p.candidates.terminate_ids.find(lb_ud->id))
                process_this = false;
        else
#endif
        if (! p.has_ub())           // if no UB yet or lb is lower than UB then go ahead
                break;
(...)
```

Appendix B

Test Scripts

The results of chapter 9 were retrieved using test scripts written in Bash and Python code. The *printstats.py* script expects a file containing the output of a set of test runs, usually covering more than one instance and testing several configurations. The results are grouped by filename and configuration, and miscellaneous statistical data can be extracted. For example, tables containing the final objective values or the online performance rating. Additionally, plots of the final objective value can be created. The *gnuplot* program is used to generate these cuts which can be viewed on screen or written to a postscript file. Since the test logs are usually rather large and take some seconds for processing, a simple interactive command line interface was implemented to shorten user response times.

B.1 Generating Log Files

To simplify testing different configurations on many different files, a short Bash script is available. The configurations to be tested are entered as an array, which is then applied to every file supplied. Since the instances of the OR Library [2] contain 30 test instances per file, the instance numbers to be tested can be specified in an array.

```
#!/bin/bash
outfile=testall.log
rm $outfile
instances="`seq 0 29`"
testcases=("LB_K 0" "LB_K 10 LB_MaxNodes 5000" "LB_K 20")

for file in $*
do
  echo -e "Processing" $file "...\n"
    for inst in $instances
    do
      for opts in "${testcases[@]}"
      do
        date >> $outfile
        echo -e "Processing" $file ", instance" $inst ", params =" $opts "...\n" >> $outfile
        nice Linux-O/bcps $opts ${file}:${inst} >> $outfile
        echo >> $outfile
      done
    done
done
```

The script has to be executed from the main knapsack application directory and stores the results in the file specified by *outfile*. The instance numbers are stored in *instances* (in this case $\{0 \dots 29\}$), the configurations are stored in *testcases*. The test files are supplied on the command line, possibly using wildcards.

B.2 Analyzing Log Files

The *printstats.py* script parses the log file given on the command line and offers a simple line-based interactive interface to query the results. The most important commands are:

- *help* returns a list of all commands.
- *help [command]* returns a short description and possible parameters of the given command.
- *table [configuration]** prints a table containing all tested instances as rows and the given configurations (or all, if none are supplied) as columns. The index numbers of the configurations correspond with the log file and are also displayed below the table.
- *columns [parameter]* sets the displayed values. Possible parameters are:
  - *finalobjective*: the final objective value.
  - *finalobjective delta*: the final objective value, and the number of processed nodes relative to the best configuration in a row (when two configurations found the same result.)
  - *onlineperformance*: the online performance rating.
  - *localtrees*: the number of (created) local trees.
  - *localtime*: time spent in local branching relative to the total computation time.
  - *finalbinary*: a binary comparison function for the final objective value, useful for executing Wilcoxon rank sum tests.
- *showfilename [true/false]* enables or disables the file name column in the table view.
- *plot [filename] [configurations]** executes gnuplot to plot the final objective values of the given configurations (or all if none are given).
- *outputformat [screen/postscript]* sets the output format of the plots generated by the plot commands. *screen* uses gnuplot to display the diagram on the screen, *postscript* writes the output to a postscript file.
- *outputdir [directory]* sets the directory where the postscript files are stored (default: current working directory.)

Bibliography

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

[2] J. E. Beasley. Operation research library. http://www.brunel.ac.uk/depts/ma/research/jeb/info.html.

[3] D. Bertsimas and R. Demir. An approximate dynamic programming approach to multidimensional knapsack problems. *Management Science*, 48(4):550–565, 2002.

[4] A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123:333–345, 2000.

[5] S. Ceria, G. Cornuejols, and M. Dawande. Combining and strengthening gomory cuts. In E. Balas and J. Clausen, editors, *Integer Programming and Combinatorial Optimization: Proc. of the 4th International IPCO Conference*, pages 438–451. Springer, Berlin, Heidelberg, 1995.

[6] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.

[7] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, (4):305–337, 1973.

[8] E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 2004.

[9] L. Davis. A genetic algorithm tutorial. In *Handbook of Genetic Algorithms*, pages 1–101, New York, 1991.

[10] F. Eisenbrand. On the chvátal rank of polytopes in the 0/1 cube. *Discrete Applied Mathematics*, 98:21–27, 1999.

[11] F. Eisenbrand. *Gomory-Chvátal cutting planes and the elementary closure of polyhedra*. PhD thesis, 2000.

[12] M. Esö, L. Ladányi, T. K. Ralphs, and L. Trotter. Fully parallel generic branch-and-cut. *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[13] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2002.

[14] H. C. for Enterprise Science. Benchmarks for the multiple knapsack problem. http://hces.bus.olemiss.edu/tools.html.

[15] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulleting of the American Mathematical Society*, (64):275–278, 1958.

[16] R. E. Gomory. An algorithm for integer solutions to linear programs. *Recent Advances in Mathematical Programming*, pages 269–302, 1963.

[17] J. Gottlieb. Permutation-based evolutionary algorithms for multidimensional knapsack problems. *Proceedings of 2000 ACM Symposium on Applied Computing*, 2000.

[18] R. Hinterding. Mapping, order-independent genes and the knapsack problem. *Proceedings of the 1st IEEE International Conference on Evolutionary Computation*, pages 13–17, 1994.

[19] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[20] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[21] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[22] L. G. Khachian. A polynomial algorithm for linear programming. *Doklady Akad. Nauk USSR*, 224:1093–1096, 1979.

[23] P. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13:723–735, 1967.

[24] B. Korte and R. Schrader. On the existence of fast approximation schemes. In O. L. Mangasarian, R. R. Meyer, and S. Robinson, editors, *Nonlinear Programming 4*, pages 415–437. Academic Press, 1981.

[25] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[26] E. K. Lee and J. E. Mitchell. Branch-and-bound methods for integer programming. In *Encyclopedia of Optimization*, volume 2, pages 509–519. Kluwer Academic Publishers, 2001.

[27] J. S. Lee and M. Guignard. An approximate algorithm for multidimensional zero-one knapsack problems. *Management Science*, 34(3):402–410, 1988.

[28] R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47:57–66, 2003.

[29] J. E. Mitchell. Branch-and-cut algorithms for integer programming. In *Encyclopedia of Optimization*, volume 2, pages 519–525. Kluwer Academic Publishers, 2001.

[30] J. E. Mitchell. Cutting plane algorithms for integer programming. In *Encyclopedia of Optimization*, volume 2, pages 525–533. Kluwer Academic Publishers, 2001.

[31] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, 1991.

[32] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, Dept. of Computer Science, Feb. 1995.

[33] G. Raidl. An improved genetic algorithm for the multiconstrainted 0-1 knapsack problem. *Proceedings of the 5th IEEE International Conference on Evolutionary Computation*, pages 207–211, 1998.

[34] G. R. Raidl. Weight-codings in a genetic algorithm for the multiconstraint knapsack problem. *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*, pages 596–603, 1999.

[35] G. R. Raidl and J. Gottlieb. Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem. *Accepted for publication in the Evolutionary Computation Journal*, 2004.

[36] T. K. Ralphs and L. Ladányi. *COIN/BCP User's Manual*. http://www.coin-or.org/Presentations/bcp-man.pdf, 2001.

[37] G. D. Scudder and G. Fox. A heuristic with tie breaking for certain 0-1 integer programming models. *Naval Research Logistics Quarterly*, 32:613–623, 1985.

[38] S. Senju and Y. Toyoda. An approach to linear programming with 0-1 variables. *Management Science*, 11:B196–B207, 1967.

[39] J. Thiel and S. Voss. Some experiences on solving multiconstraint zero-one knapsack problems with genetic algorithms. *INFOR 32*, pages 226–242, 1994.

[40] Y. Toyoda. A simplified algorithm for obtaining approximate solution to zero-one programming problems. *Management Science*, 21:1417–1427, 1975.

[41] M. Vasquez and J.-K. Hao. A hybrid approach for the 0-1 multidimensional knapsack problem. *Proceedings of IJCAI 01*, 2001.

[42] L. A. Wolsey. *Integer Programming*. John Wiley and Sons, 1998.