COIN/BCP User's Manual

T.K. Ralphs[1] L. Ladányi[2]

January 2001

[1]Department of Industrial and Manufacturing Systems Engineering, Lehigh University, Bethlehem, PA 18015, tkralphs@lehigh.edu, http://www.lehigh.edu/~tkr2
[2]Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

©2001 International Business Machines Corporation, Ted Ralphs and others. All right reserved.

Contents

1 Introduction **7**

1.1 A Brief History 7

1.2 Related Work 8

1.3 Organization of the Manual 9

1.4 Introduction to Branch, Cut and Price 9

1.4.1 Branch and Bound 9

1.4.2 Branch and Cut 11

1.4.3 Branch and Price 13

1.4.4 Branch, Cut and Price 14

1.5 Design of COIN/BCP 14

1.5.1 An Object-oriented Approach 14

1.5.2 Data Structures and Storage 15

1.5.3 Modular Implementation 18

1.5.4 COIN/BCP Overview 20

1.6 Details of the Implementation 21

1.6.1 The Tree Manager Module 21
1.6.2 The LP Module 23
1.6.3 The Cut Generator Module 25
1.6.4 The Variable Generator Module 25
1.7 Parallelizing COIN/BCP 26
1.7.1 Parallel Execution and Inter-process Communication 26
1.7.2 Fault Tolerance 27

2 Getting Started: Sample Compiling **28**

2.1 System Requirements 28
2.2 Obtaining the Source Code 29
2.2.1 Using CVS 29
2.2.2 Downloading a `tar` File 30
2.3 Initial compilation and testing 30
2.3.1 Compiling for serial execution 31
2.3.2 Compiling for distributed networks 32

3 Developing Applications with COIN/BCP **34**

3.1 Directory Layout (location of the source files) 34
3.2 Overview of the Class Hierarchy 35
3.3 The Flow of the Algorithm 36
3.3.1 Fathoming procedure 37
3.4 Details of the Interface 40

3.4.1 The `USER_initialize` class 40
3.4.2 The `BCP_tm_user` class 41
3.4.3 The `BCP_lp_user` class 42
3.4.4 The `BCP_cg_user` class 46
3.4.5 The `BCP_vg_user` class 47
3.5 Deriving Problem-specific Classes 48
3.5.1 Generating cuts 48
3.5.2 Generating variables 49
3.5.3 Setting the Core and Extra Object Lists 50
3.5.4 Branching 51
3.5.5 Summary and Optional Methodss 52
3.6 Internal Data Structures 53
3.7 Inter-process Communication 54
3.8 Debugging Your Application 54
3.8.1 The First Rule 54
3.8.2 Debugging with PVM 55
3.8.3 Using `Electric Fence` 55
3.8.4 Using `Purify` 55

4 Sample Application: The MKC Problem **56**
4.1 The MKC Problem 56
4.2 Natural formulation for MKC 57
4.3 A formulation suitable for column generation 59

4.3.1 Generating columns with positive reduced costs 60
4.3.2 Upper bounding 60
4.3.3 Finding integral feasible solutions 61
4.4 Implementation details 61
4.4.1 Cuts, variables and solutions 61
4.4.2 Branching 62
4.4.3 Packing and unpacking 63
4.4.4 MKC_init 63
4.4.5 MKC_tm 63
4.4.6 MKC_lp 63

5 Sample Application: The Maximum Cut Problem **64**
5.1 The Max Cut Problem 64
5.2 Implementation 65
5.2.1 `MC_tm` 66
5.2.2 `MC_lp` 67
5.2.3 `MC_solution` 68
5.2.4 `MC_cycle_cut` 68
5.2.5 Other Classes 68

Chapter 1

Introduction

1.1 A Brief History

Since the inception of optimization as a mathematical discipline, researchers have been intrigued and stymied by the difficulty of solving discrete optimization problems. Even problems with natural and concise formulations remain challenging to solve in practice. The most significant advance in general methodologies occured in 1991 when Padberg and Rinaldi [15] merged the enumeration approach of branch and bound algorithms with the polyhedral approach of cutting planes to create the technique we call *branch, cut and price* or simply *BCP*. Integrating the contributions of many in the field , their paper launched a new era in discrete optimization techniques.

In the last two decades, we have seen tremendous progress in our ability to solve specially structured large-scale discrete optimization problems. Indeed, in 1998, Applegate, Bixby, Cook, and Chvátal [7] solved a *Traveling Salesman Problem* (TSP) instance with 13,509 cities; a full order of magnitude larger than what had been possible just a decade earlier and two orders of magnitude larger than the largest problem that had been solved up until 1978. This progress becomes even more impressive when one realizes that the number of variables in the standard formulation for this problem is approximately the *square* of the problem size. Hence, we are talking about solving a problem with somewhere in excess of *100 million variables.*

This fantastic progress can be attributed to several factors. The increase in available computing power over the last decade, both in terms of processor speed and memory, has been

nothing short of amazing. This hardware improvement made it possible to tackle larger problems thus accumulating knowledge about how the large problems "behave". In turn, this led to increasingly sophisticated software for optimization, and to a wealth of theoretical results. Also, many theoretical results, which had no computational importance for small problems, were "re-discovered" as larger problems became the target. Finally, the use of parallel computing has allowed researchers to further leverage their gains.

As computational research in optimization becomes more widespread and the sophistication of computational techniques increases, one of the main difficulties faced by researchers in the area is the level of effort required to develop an efficient implementation. The need for incorporating problem-dependent methods (most notably dynamic generation of variables and cutting planes) typically required time-consuming development of custom implementations. In the early 1990's a research group was formed with the goal of creating a generic software framework which users could easily customize for their particular problem class. This group eventually produced what was then known as COMPSys (Combinatorial Optimization Multi-processing System) [9]. After several revisions which broadened the framework's functionality, COMPSys became SYMPHONY (Single- or Multi-Process Optimization over Networks) [21]. Starting in 1998 a total reimplementation in C++ was undertaken at IBM research to enable greater flexibility. The result of this effort was opened up as an open-source project in 2000 under the auspices of the Common Optimization INterface for Operations Research (COIN-OR) [6] and is codenamed COIN/BCP.

1.2 Related Work

In the 1990's, there was a virtual explosion of software for discrete optimization. Almost without exception, these new software packages were based on the basic techniques of branch, cut and price. The general-purpose packages fall into two main categories – those based on algorithms not exploing special structures for solving general mixed integer programs (MIPs) and those facilitating the use of special structure via user-supplied, problem-specific subroutines. We will call packages in this second category *frameworks*. There have also been numerous special-purpose codes developed specifically for use in a particular problem setting.

Of the two categories, general-purpose MIP solvers are the most common. Among the dozens of offerings in this category, the most notable are MINTO [12], MIPO [13], bc-opt [4] and several commercial packages.

Generic frameworks, on the other hand, are far less numerous. The most full-featured

packages available are the two frameworks we have already mentioned (SYMPHONY and COIN/BCP) and a commercial product, ABACUS [1]. Some general-purpose MIP solver, such as MINTO [12] and several commercial packages, now have a limited capability of utilizing problem-specific subroutines.

Related software includes general MIP solvers implementing parallel branch and bound (PARINO [16] and FATCOP [10]); frameworks for general parallel branch and bound (PUBB [19], BoB [5], PPBB-Lib [18], and PICO [17]) and special-purpose codes, like CONCORDE, a package for solving the *Traveling Salesman Problem* (TSP). This latter code is the most sophisticated special-purpose code developed to date.

1.3 Organization of the Manual

The manual is divided into three parts. Part I is a gentle introduction to branch, cut and price algorithms, including an overview of the design of COIN/BCP. Anyone interested in learning about or using the framework should spend time with Part I; those already familiar with BCP algorithms will probably want to skim the introductory sections. The reader merely interested in only a high-level description of the framework may wish to stop after reading Part I. Part II is intended to provide the specific details needed to actually develop applications using COIN/BCP. This includes "how-to" descriptions of customizing the Makefiles, compiling the sample code, deciding wich built-in methods to override, and performing other development tasks such as debugging. Part III illustrates these principles with a concrete example. A reference manual of every class structure is available in HTML format and is part of the standard distribution.

1.4 Introduction to Branch, Cut and Price

1.4.1 Branch and Bound

Branch and bound is the broad class of algorithms from which branch, cut and price is descended. A branch and bound algorithm uses a divide and conquer strategy; it partitions the solution space into *subproblems* and then optimizes over each subproblem individually. For instance, let S be the set of solutions to a given problem, and let $c \in \mathbf{R}^S$ be a vector of costs associated with members of S. Suppose we wish to determine a least cost member of S and we are given $\hat{s} \in S$, a "good" solution determined heuristically. Whenever a new, better solution is found we will replace $\hat{s}$ with the new solution. Thus the value of $\hat{s}$ is

always a *global upper bound* on th eoptimal value. In the branch and bound algorithm we maintain a list of *candidate subproblems* each of which contain a subset of the original feasible solutions. This list is initialized by placing S on it. In the *processing* or *bounding* phase of the algorithm we take an entry, say S'. from the candidate list and remove it from the list. We *relax* S', that is, we admit solutions that are not in S' and solve the relaxd problem. There are four possible results.

- The relaxed problem is found to be infeasible. In this case we obviously cannot find a new feasible solution that is feasible and is in S', thus we can *prune* (or *fathom*) the subproblem, that is, we discard it.
- The optimal solution to the relaxed problem is not better (not lower) than the global upper bound. In this case the value of any feasible solution in S' must also not be better than the global upper bound thus we cannot find a feasible solution in s' that would be better than the currently known best solution. Therefore we can fathom the subproblem in this case, too.
- The optimal solution to the relaxed problem is better than the global upper bound *and* it is in S', i.e., it is feasible. In this case we replace $\hat{s}$ with this new solution. Also, we cannot find any even better solution in this subproblem thus we can fathom it.
- The optimal solution to the relaxed problem is better than the global upper bound but is not in S'. In this case we *branch*, i.e., we identify n subsets of S', $S'_1, \dots, S'_n$, such that $\cup_{i=1}^n S'_i = S'$. Each of these subsets, the *children* of S', is a new candidate subproblem and is added to the candidate list.

After a subproblem is processed (and either pruned or branched) a new subproblem is selected for processing as long as the candidate list is not empty, at which point our current best solution is the optimal one.

The sequence of subproblems generated can be displayed as a rooted directed graph; the original problem being the root and there are edges from each subproblem to its children. This graph is the *search tree* and the expression *search (tree) node* is used interchangeably with *subproblem.*

In many applications, the bounding operation is accomplished using the tools of linear programming (LP), a technique first described in full generality by Hoffman and Padberg [**?**]. This general class of algorithms is known as *LP-based branch and bound.* Typically, the integrality constraints of an integer programming formulation of the problem are relaxed to obtain a *LP relaxation*, which is then solved to obtain a lower bound for the problem.

1.4.2 Branch and Cut

Padberg and Rinaldi [15] improved on the basic idea of LP-based branch and bound by describing a method of using globally valid inequalities (i.e., inequalities valid for the convex hull of integer solutions) to strengthen the LP relaxation. They called this technique *branch and cut*. Since then, many implementations (including ours) have been fashioned around the framework they described for solving the Traveling Salesman Problem.

As an example, let a combinatorial optimization problem $CP = (E, \mathcal{F})$ with *ground set* E and *feasible set* $\mathcal{F} \subseteq 2^E$ be given along with a cost function $c \in \mathbf{R}^E$. Now let $\mathcal{P}$ be the convex hull of incidence vectors of members of $\mathcal{F}$. Then we know by Weyl's Theorem (see [14]) that there exists a finite set of inequalities $\mathcal{L}$ which are valid for $\mathcal{P}$ such that

$$\mathcal{P} = \{x \in \mathbf{R}^n : ax \leq \beta \ \ \forall \, (a, \beta) \in \mathcal{L}\}. \tag{1.1}$$

Unfortunately, it is usually difficult, if not impossible, to enumerate all inequalities in $\mathcal{L}$ (or even just those that describe the convex hull near the optimal corner) or we could simply solve the problem using linear programming.

The set of incidence vectors corresponding to the members of $\mathcal{F}$ is sometimes approximated as the set of all incidence vectors obeying a (relatively) small set of inequalities (these inequalities are typically the ones used in the initial LP relaxation). Then in each node of the search tree globally valid inequalities are generated (the best such inequalities are in $\mathcal{L}$) using separation algorithms and heuristics. One could say that the inequalities describing $\mathcal{P}$ are defined implicitely and generated as they are needed.

This way the relaxation is tightened thus the bounding step has a better chance to fathom the node based on the objective value. In Figure 1.1, we describe more precisely how cut generation is employed within the bounding operation of the branch and cut technique.

Once we have failed to generate cuts and the subproblem still cannot be pruned based on the objective value, we are forced to branch. The branching operation is accomplished by specifying a set of hyperplanes which divide the current subproblem in such a way that the current solution is not feasible for the LP relaxation of any of the new subproblems. For example, in a combinatorial optimization problem, branching could be accomplished simply by fixing a variable whose current value is fractional to 0 in one branch and 1 in the other. The procedure is described more formally in Figure 1.2. Figure 1.3 gives a high level description of the generic branch and cut algorithm.

Note that adding cutting planes only increases the lower bound at a search tree node hence as soon as the lower bound on a node exceeds the value of the best known solution the

Bounding Operation
Input: A subproblem $\mathcal{S}$, described in terms of a "small" set of inequalities $\mathcal{L}'$ such that $\mathcal{S} = \{x^s : s \in \mathcal{F} \text{ and } ax^s \leq \beta \ \forall\ (a, \beta) \in \mathcal{L}'\}$ and α, an upper bound on the global optimal value.
Output: Either (1) an optimal solution $s^* \in \mathcal{S}$ to the subproblem, (2) a lower bound on the optimal value of the subproblem, or (3) a message `pruned` indicating that the subproblem should not be considered further.
Step 1. Set $\mathcal{C} \leftarrow \mathcal{L}'$.
Step 2. Solve the LP $\min\{cx : ax \leq \beta \ \forall\ (a, \beta) \in \mathcal{C}\}$.
Step 3. If the LP has a feasible solution $\hat{x}$, then go to Step 4. Otherwise, STOP and output `pruned`. This subproblem has no feasible solutions.
Step 4. If $c\hat{x} < \alpha$, then go to Step 5. Otherwise, STOP and output `pruned`. This subproblem cannot produce a solution of value better than α.
Step 5. If $\hat{x}$ is the incidence vector of some $\hat{s} \in \mathcal{S}$, then $\hat{s}$ is the optimal solution to this subproblem. STOP and output $\hat{s}$ as s^*. Otherwise, apply separation algorithms and heuristics to $\hat{x}$ to get a set of violated inequalities $\mathcal{C}'$. If $\mathcal{C}' = \emptyset$, then $c\hat{x}$ is a lower bound on the value of an optimal element of $\mathcal{S}$. STOP and return $\hat{x}$ and the lower bound $c\hat{x}$. Otherwise, set $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$ and go to Step 2.

Figure 1.1: Bounding in the branch and cut algorithm

Branching Operation
Input: A subproblem $\mathcal{S}$ and $\hat{x}$, the LP solution yielding the lower bound.
Output: $S_1, \ldots, S_p$ such that $\mathcal{S} = \cup_{i=1}^{p} S_i$.
Step 1. Determine sets $\mathcal{L}_1, \ldots, \mathcal{L}_p$ of inequalities such that $\mathcal{S} = \cup_{i=1}^{n} \{x \in \mathcal{S} : ax \leq \beta \ \forall\ (a, \beta) \in \mathcal{L}_i\}$ and $\hat{x} \notin \cup_{i=1}^{n} S_i$.
Step 2. Set $S_i = \{x \in \mathcal{S} : ax \leq \beta \ \ \forall\ (a, \beta) \in \mathcal{L}_i \cup \mathcal{L}'\}$ where $\mathcal{L}'$ is the set of inequalities used to describe $\mathcal{S}$.

Figure 1.2: Branching in the branch and cut algorithm

Generic Branch and Cut Algorithm
Input: A data array specifying the problem instance.
Output: The global optimal solution s^* to the problem instance.
Step 1. Generate a "good" feasible solution $\hat{s}$ using heuristics. Set $\alpha \leftarrow c(\hat{s})$.
Step 2. Generate the first subproblem $\mathcal{S}^I$ by constructing a small set $\mathcal{L}'$ of inequalities valid for $\mathcal{P}$. Set $A \leftarrow \{\mathcal{S}^I\}$.
Step 3. If $A = \emptyset$, STOP and output $\hat{s}$ as the global optimum s^*. Otherwise, choose some $\mathcal{S} \in A$. Set $A \leftarrow A \setminus \{\mathcal{S}\}$. Process $\mathcal{S}$.
Step 4. If the result of Step 3 is a feasible solution $\overline{s}$, then $c\overline{s} < c\hat{s}$. Set $\hat{s} \leftarrow \overline{s}$ and $\alpha \leftarrow c(\overline{s})$ and go to Step 3. If the subproblem was pruned, go to Step 3. Otherwise, go to Step 5.
Step 5. Perform the branching operation. Add the set of subproblems generated to A and go to Step 3.

Figure 1.3: Description of the generic branch and cut algorithm

search tree node can be fathomed.

1.4.3 Branch and Price

As with cutting planes, the columns of A can also be defined implicitly if n is large. If column i is not present in the current matrix, then variable x_i is implicitly taken to have value zero. The process of dynamically generating variables is called *pricing* in the jargon of linear programming. The term originates from computing the *reduced cost* of the column and adding the column to the formulation if it has a negative reduced cost. This procedure can also be viewed as that of generating cutting planes for the dual of the current LP relaxation. Hence, LP-based branch and bound algorithms in which the variables are generated dynamically when needed are known as *branch and price* algorithms. Savelsbergh, et al. [**?**] provide a thorough review of these methods.

Although "branch and cut" and "branch and price" look very symmetric there is a very significant difference. When cuts are introduced the lower bound on the relaxation can only increase while introducing new columns can lower the lower bound. Therefore a search tree node cannot be fathomed until all variables are priced out *and* the lower bound is higher than the best known solution value. For this reason frequently the *price and branch* algorithm is executed, that is first we price, then we do plain branch and bound. Obviously, this will most likely result in a suboptimal solution, but for practical purposes this is usually sufficient.

1.4.4 Branch, Cut and Price

Finally, when both variables and cutting planes are generated dynamically during LP-based branch and bound, the technique becomes known as *branch, cut and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of cuts and variables. However, it is important to note that while branch, cut and price does combine ideas from both branch and cut and branch and price (which are very similar to each other anyway), combining the two techniques requires much more sophisticated methods than either one requires on its own. The effects of this observation are noticable throughout the design of COIN/BCP (see, in particular, Section 1.5.2).

1.5 Design of COIN/BCP

COIN/BCP was designed with three major goals in mind – portability, efficiency and ease of use. With respect to ease of use, we aimed for a "black box" design, whereby the user would not be required to know anything about the implementation of the library, but only about the user interface. With respect to portability, we aimed not only for it to be *possible* to use the framework in a wide variety of settings and on a wide variety of hardware, but also for it to perform *effectively* in all these settings. Our primary measure of effectiveness is how well the framework performs in comparison to problem-specific (or hardware-specific) implementation written "from scratch."

It is important to point out that achieving such design goals involves a number of very difficult tradeoffs, which we will highlight throughout the rest of the manual. For instance, ease of use is quite often at odds with efficiency. In several instances, we had to give up some efficiency to make the code easy to work with and to maintain a true black box implementation. Maintaining portability across a wide variety of hardware, both sequential and parallel, also required some difficult choices. For example, solving large-scale problems on sequential platforms requires extremely memory-efficient data structures in order to maintain the very large search trees that can be generated. However, these storage schemes are highly centralized and do not scale well to large numbers of processors.

1.5.1 An Object-oriented Approach

Applying BCP to large-scale problems presents several difficult challenges. First and foremost is designing methods and data structures capable of handling the potentially huge

number of global cuts and variables that need to be accounted for during the solution process. A second challenge, which is closely related to the first, is effectively dealing with the very large search trees that can be generated for difficult problem instances. A third challenge is to deal with these issues using a problem-independent approach.

Describing a node in the search tree consists of, among other things, specifying which cuts and variables are initially *active* in the subproblem. Hence, the central "objects" in our framework are the cuts and variables. From the user's perspective, implementing a BCP algorithm using COIN/BCP consists primarily of specifying various properties of objects, such as how they are generated, how they are represented, and how they should be realized within the context of a particular subproblem. This is achieved using class derivation and virtual methods. There are a few base classes defined in COIN/BCP that the user can derive classes from and then she can override the virtual methods hence modifying the behavior of the code. Some methods she must override, and for some she can just let the method in the base class executed thus selecting the default behavior. In Sections 3.2 and 3.4 a more detailed description is given about the classes the user can derive new object types from.

1.5.2 Data Structures and Storage

Both the memory required to store the search tree and the time required to process a node are largely dependent on the number of objects (cuts and variables) that are active in each subproblem. Keeping this active set as small as possible is one of the keys to efficiently implementing BCP. For this reason, we chose data structures that enhance our ability to efficiently move objects in and out of the active set. Allowing sets of cuts and variables to move in and out of the linear programs simultaneously is one of the most significant challenges of BCP. We do this by maintaining an abstract *representation* of each global object that contains information about how to add it to a particular LP relaxation.

In the literature on linear and integer programming, the terms *cut* and *row* are typically used interchangeably. Similarly, *variable* and *column* are often used with similar meanings. In many situations, this is appropriate and does not cause confusion. However, in object-oriented BCP frameworks, such as COIN/BCP or ABACUS, a *cut* and a *row* are **fundamentally different objects**. A *cut* (also referred to as a *constraint*) is a user-defined representation of an abstract object which can only be realized as a row in an LP matrix **with respect to a particular set of active variables**. Similarly, a *variable* is a representation which can only be realized as a column of an LP matrix **with respect to a particular set of cuts**. This distinction between the *representation* and the *realization* of objects is a crucial design element and is what allows us to effectively address some of the

challenges inherent in BCP. In the remainder of this section, we will further discuss this distinction and the details of how it is implemented.

Variables and Cuts

Although their algorithmic roles are different, variables and cuts as objects are treated identically in COIN/BCP. We will describe the various types of variables.

The variables are divided into two main groups, core variables and extra variables. The core variables are active in all subproblems, whereas the extra variables can be added and removed. There is no theoretical difference between core and extra variables; however, designating a well-chosen set of core variables can significantly increase efficiency. Because they can move in and out of the problem, maintaining extra variables requires additional bookkeeping and computation. If the user has reason to believe a priori that a variable is "good" or has a high probability of having a non-zero value in some optimal solution to the problem, then that variable should be designated as a core variable. It is up to the user to designate which variables should be active in the root subproblem. Typically, only core variables are active, but there are times when extra variables are also included. Note that using extra variables is only necessary if the user wishes to take advantage of the column-generation features of the framework. For problems with a moderate number of variables, it is probably more efficient to designate every variable as core variable.

The extra variables are also subdivided into two category. First, there are the indexed variables which are represented by a unique global user index which is (as the name suggests) assigned to these variables by the user. This index represents each variable's position in a "virtual" global list known only to the user. The main requirement of this indexing scheme is that, given an index and a list of active cuts, the user must be able to generate the corresponding column to be added to the matrix. For example, in problems where the variables correspond to the edges of an underlying graph, the index could be derived from a lexicographic ordering of the edges.

The indexing scheme provides a very compact representation, as well as a simple and effective means of moving variables in and out of the active set. However, it means that the user must have a priori knowledge of all problem variables and a method for indexing them. For combinatorial models such as the Traveling Salesman Problem, this usually does not present a problem. However, for airline schedule planning models, for instance, the number of columns (each one corresponds to a possible plane route) is not known in advance. In such cases the user may use algorithmic variables. For algorithmic variables there must exists an algorithm that, given the set of active constraints, can create the column corresponding

to the variable. Using the schedule planning example, the compact representation may be the information which flight legs a particular plane is going to fly. From this information it's easy to derive when the plane is on the ground and hence it is easy to compute the coefficients of the column for constraints that, say, specify that at a given time at a given airport only so many planes can be on the ground.

To summarize the advantages and disadvantages of the various variable types:

- core variable: always stay in the formulation which is both good (no bookkeeping required and decreases communication) and bad (the LP relaxation will always have at least those variables in the formulation);
- indexed variables: they can leave the formulation (good) but there are some bookkeeping involved and all must be accounted for in advance (bad); and
- algorithmic variables: gives absolute freedom, there can be as many as the user wants (good) but there is fair amount of bookkeeping involved (bad).

Now we give examples for *indexed* and *algorithmic cuts*. The already mentioned "gate constraints" for the schedule planning problem are typical indexed cuts. There is one for every airport for every minute during the day. Most likely very few of them would ever be violated, but they must be satisfied. Including all as core cuts would increase the problem size enormously. Therefore they better be generated on the fly, and since we can enumerate them in advance, i.e., we can assign an index to each of them, they can be indexed cuts. An algorithmic cut is, for example, a subtour elimination constraint for the TSP. These constraint express that a tour must cross every cut at least twice. There are so many of them ($2^n - 2$ for a problem with n nodes) that we cannot enumerate all of them and assign indeces to them. Thus we need a representation and an algorithm that computes the coefficients for every active variable. A compact representation could be the list of graph nodes on the smaller side of the cut. From this it is easy to deduce which variables (edges) are in the cut, i.e., we can compute the coefficients.

Search Tree

Having described the basics of how objects are represented, we now describe the representation of search tree nodes. Since the core constraints and variables are present in every subproblem, only the indices of the extra constraints and variables are stored in each node's description. Also warm starting information is maintained at the search tree nodes to speed up solving the LP relaxation when the processing of a search tree node begins. This warm

start information is either inherited from the parent or comes from earlier partial processing of the node itself (see Section 1.6.1). Along with the set of active objects, we must also store the branching information that was used to generate the node. The branching operation is described in Section 1.6.2.

Because the set of active objects and the LP solution (hence the warm starting information) do not tend to change much from parent to child, all of these data are stored as differences with respect to the parent when that description is smaller than the explicit one. This method of storing the entire tree is highly memory-efficient. The list of nodes that are candidates for processing is stored in a heap ordered by a comparison function defined by the search strategy (see 1.6.1). This allows efficient generation of the next node to be processed.

1.5.3 Modular Implementation

COIN/BCP's functions are currently grouped into four independent computational modules. This modular implementation not only facilitates code maintenance, but also allows easy and highly configurable parallelization. All modules are present in the executable file but depending on the computational setting, either they run alternating thus executing the algorithm as a serial process or in each process only one module will be running thus executing the algorithm in parallel over a network. The modules pass data through a message-passing protocol defined in a separate communications API. In the remainder of the section, we describe the modularization scheme and the implementation of each module in a sequential environment. We will defer serious discussion of the issues involved in parallel execution of the code until Section **??**.

The Tree Manager Module

The *tree manager module* (TM) first performs problem initialization and I/O and then becomes the master process controlling the overall execution of the algorithm. It tracks the status of all processes, as well as that of the search tree, and distributes the subproblems to be processed to the LP module(s). Specific functions performed by the tree manager module are:

- Read in the parameters from a data file.
- Read in the data for the problem instance.

- Compute an initial upper bound using heuristics.
- Perform problem preprocessing.
- Initialize the BCP algorithm by constructing the root node.
- Initialize output devices and act as a central repository for output.
- Process requests for problem data.
- Receive new solutions and store the best one.
- Receive data for subproblems to be held for later processing.
- Handle requests from linear programming modules to release a subproblem for processing.
- Receive branching object information, set up data structures for the children, and add them to the list of candidate subproblems.
- Keep track of the global upper bound and notify all LP processes when it changes.
- Write current state information out to disk periodically to allow a warm restart in the event of a system crash.
- Receive the message that the algorithm is finished and print out run data.

The Linear Programming Module

The *linear programming* (LP) module is the most complex and computationally intensive of the four processes. Its job is to perform the bounding and branching operations. These operations are, of course, central to the performance of the algorithm. Functions performed by the LP module are:

- Inform the tree manager when a new subproblem is needed.
- Receive a subproblem. Process the subproblem in conjunction with the cut generator.
- If necessary, choose a branching object and send its description back to the tree manager.

The Cut Generator Module

The *cut generator* performs only one function – generating valid inequalities violated by the current fractional solution and sending them back to the requesting LP process. Here are the functions performed by the cut generator module:

- Receive an LP solution and attempt to separate it from the convex hull of all solutions.
- Send generated valid inequalities back to the LP solver.
- When finished processing a solution vector, inform the LP not to expect any more cuts in case it is still waiting.

The Variable Generator Module

The function of the *variable generator* (VG) is dual to that of the cut generator. Given a dual solution vector, the variable generator attempts to generate variables with negative reduced cost and sends them back to the requesting LP process if any are found. Here are the functions performed by the variable generator module:

- Receive a set of dual values and attempt to generate variables with negative reduced cost.
- Send generated variables back to the LP solver.
- When finished processing a dual solution vector, inform the LP not to expect any more variables in case it is still waiting.

1.5.4 COIN/BCP Overview

Currently, COIN/BCP is what is known as a single-pool BCP algorithm. The term *single-pool* refers to the fact that there is a single central list of candidate subproblems to be processed, which is maintained by the tree manager. Most sequential implementations use such a single-pool scheme. However, other schemes may be used in parallel implementations. For a description of various types of parallel branch and bound, see [11].

The tree manager module begins by reading in the parameters and problem data. After initial I/O is completed, subroutines for finding an initial upper bound and constructing

the root node are executed. During construction of the root node, the user must designate the initial set of active cuts and variables, after which the data for the root node are used to initialize the list of candidate nodes. The tree manager then sets up the cut pool module(s), the linear programming module(s), and the cut generator module(s). All LP modules are marked as idle. The algorithm is now ready for execution.

In the steady state, the tree manager controls the execution by maintaining the list of candidate subproblems and sending them to the LP modules as they become idle. The LP modules receive nodes from the tree manager, process them, branch (if required), and send back the identity of the chosen branching object to the tree manager, which in turn generates the children and places them on the list of candidates to be processed.

The preference ordering for processing nodes is a run-time parameter (or can be controlled by a user-defined method). Typically, the node with the smallest lower bound is chosen to be processed next since this strategy minimizes the overall size of the search tree. However, at times, it will be advantageous to *dive* down in the tree. The concepts of *diving* and *search chains*, introduced in Section 1.6.1, extend the basic "best-first" approach.

1.6 Details of the Implementation

1.6.1 The Tree Manager Module

The primary functions performed by the tree manager were listed in Section 1.5.3. During initialization, the user can provide a routine to read problem-specific parameters in from the parameter file. She can also provide a subroutine for upper bounding if desired, though upper bounds can also be provided explicitly. A good initial upper bound can dramatically decrease the solution time by allowing more variable-fixing and earlier pruning of search tree nodes. If no upper bounding subroutine is available, then the two-phase algorithm, in which a good upper bound is found quickly in the first phase using a reduced set of variables can be advantageous. See Section 1.6.1 for details. The user's only unavoidable obligation during preprocessing is to specify the core of the problem, that is, the list of core variables and cuts as well as the corresponding matrix. If desired, a list of extra variables and cuts that are to be active in the root node can be specified. Again, we point out that selecting a good set of core variables can make a marked difference in solution speed, especially using the two-phase algorithm.

Search Chains and Diving

Once execution of the algorithm begins, the tree manager's primary job is to guide the search by deciding which candidate node should be chosen as the next to be processed. This is done using either one of the several built-in rules or a user-specified method. As mentioned earlier, we typically choose the node with the smallest lower bound because this rule minimizes the size of the search tree. However, there are several reasons why we might want to deviate from this rule.

One reason for not strictly enforcing the search order is because it is somewhat expensive to construct a search node, send it to the LP solver and set it up for processing. If, after branching, we choose to continue processing one of the children of the current subproblem, we avoid the set-up cost, as well as the cost of communicating the node description of the retained child subproblem back to the tree manager. This is called *diving* and the resulting chain of nodes is called a *search chain*. There are a number of rules for deciding when an LP process should be allowed to dive. One such rule is to look at the number of variables in the current LP solution that have fractional values (i.e., are causing infeasibility). When this number is low, there is a good chance of finding a feasible integer solution quickly by diving. This rule has the advantage of not requiring any global information.

We also dive if one of the children is "close" to being the best node, where "close" is defined by a chosen parameter. In addition to the time saved by avoiding reconstruction of the LP in the child, diving has the apparent advantage of quickly leading to the discovery of feasible solutions and hence better upper bounds. Since every feasible solution lies at the end of a search chain, it is reasonable to dive periodically if there is reason to believe the current upper bound is not very good. Therefore, random diving also takes place according to a specified parameter.

The Two-Phase Algorithm

As in branch and bound, finding good feasible solutions quickly is critical to the efficiency of the algorithm. Good feasible solutions provide upper bounds, which in turn allow us to prune unpromising nodes and process other nodes more efficiently. There are several ways in which feasible solutions can be found. First, the user can provide a subroutine that derives an upper bound through heuristic techniques that will be run before beginning to explore the branch and cut tree. Providing a good initial upper bound can dramatically decrease the overall solution time. Another way of finding feasible solutions is to discover them while exploring the search tree, as discussed above.

If no upper bounding subroutine is available, then a unique two-phase algorithm can also be invoked. In the two-phase method, the algorithm is first run to completion on the specified set of core variables. Any node that would have been pruned in the first phase is sent to a pool of candidates for the second phase instead. If the set of core variables is small, but well-chosen, this first phase should be quick and should result in a near-optimal solution, and hence a good upper bound. In addition, the first phase will produce a list of useful cuts. Using the upper bound and the list of cuts from the first phase, the root node is *repriced* – that is, it is reprocessed with the full set of variables and cuts, the hope being that most or all of the variables not included in the first phase will be priced out of the problem in the new root node. Any variable so priced out can be eliminated from the problem globally. If we are successful at pricing out all the inactive variables, we have shown that the solution from the first phase was, in fact, optimal. If not, we must go back and price out the (reduced) set of extra variables in each one of the leaves of the search tree produced during the first phase. We then continue processing any node in which we fail to price out all the variables.

In order to avoid pricing variables in every leaf of the tree, we can *trim the tree* before the start of the second phase. Trimming the tree consists of eliminating the children of any node whose aforementioned children all have lower bounds above the current upper bound. We then reprocess the parent node itself. This is typically more efficient since there is a high probability that, given the new upper bound and cuts, we will be able to prune the parent node and save the work of processing each child individually.

1.6.2 The LP Module

The LP Engine

COIN/BCP requires the use of a third-party callable library (referred to as the *LP engine* or *LP library*) to solve the LP relaxations once they are formulated. COIN/BCP communicates with the LP engine through the *Open Solver Interface* (OSI) an API that provides a uniform API to various LP solvers. Therefore COIN/BCP is able to work with any LP engine that has an OSI interface (currently OSL, CPLEX, XPRESS-MP and the Volume Algorithm has OSI interfaces). OSI is also part of the COIN [6] project.

Managing the LP Relaxation

The majority of the computational effort of branch and cut is spent solving LPs and hence a major emphasis in the development was to make this process as efficient as possible.

Besides using a good LP engine, the primary way in which this is done is by controlling the size of each relaxation, both in terms of number of active variables and number of active constraints.

The number of constraints is controlled through use of a local cut pool and through purging of ineffective constraints. When a cut is generated by the cut generator, it is first sent to the local cut pool. In each iteration, up to a specified number of the strongest cuts (measured by degree of violation) from the local pool are added to the problem. Cuts that are not strong enough to be added to the relaxation are eventually purged from the list. In addition, cuts are purged from the LP itself when they have been deemed ineffective for more than a specified number of iterations and the lower bound on the LP relaxation has increased since the cut was added to the formulation. The meaning of the term "ineffective" can be specified by a parameter and it is defined as either (1) the corresponding slack variable is positive or (3) the dual value corresponding to the row is zero. The second condition for purging a cut is necessary to avoid cycling.

The number of variables (columns) in the relaxation is controlled through *reduced cost fixing* and *dynamic column generation*. Periodically, each active variable is *priced* to see if it can be tightened based on their reduced cost. Note that a variable can be tightened by reduced cost fixing only if *every* extra variable is known to price out (has non-negative reduced cost). Also, in every iteration the user is given the opportunity to tighten bounds on any variable.

FIXME: ... Must write some more about dynamic column generation ...

Branching

Branching takes place whenever either (1) both cut generation and column generation (if it is performed) have failed; (2) "tailing off" in the objective function value has been detected (if this option is selected); or (3) the user chooses to force branching. A general *branching object* specifies new bounds for a set of variables and/or cuts in every children. The well-known *branching variable* is a special case: a fractional variable is selected and two children are created as usual. Selecting a branching object can be fully automated (in which case branching variables are selected) or fully controlled by the user, as desired. Branching can result in as many children as the user desires though two is typical. Once it is decided that branching will occur, the user must either select the list of candidates for *strong branching* (see below for the procedure) or allow COIN/BCP to do so automatically by using one of several built-in strategies, such as branching on the variable whose value is farthest from integrality. The number of candidates can depend on the level of the current node in the tree. For instance, it is usually best to expend more effort on branching near the top of the

tree where it is more "important".

After the list of candidates is selected, each candidate is *presolved*, i.e., a quick near-optimization is done (like performing a specified number of iterations of the dual simplex algorithm) in each of the resulting subproblems. Based on the objective function values obtained in each of the potential children, the final branching object is selected, again either by the user or by built-in rule. When the branching object has been selected, the LP process sends a description of that object to the tree manager, which then creates the children and adds them to the list of candidate nodes. It is then up to the tree manager to specify which node the now-idle LP process should process next. That issue will be addressed in Section 1.6.1 below.

1.6.3 The Cut Generator Module

To implement the cut generator process, the user must provide a method that accepts an LP solution and returns cuts violated by that solution to the LP module. In parallel configurations, each cut is returned immediately to the LP module, rather than being passed back as a group once the function exits. This allows the LP to begin adding cuts and solving the current relaxation before the cut generator is finished if desired. Parameters controlling if and when the LP should begin solving the relaxation before the cut generator is finished can be set by the user.

1.6.4 The Variable Generator Module

The variable generator functions very similarly to the cut generator. To implement the variable generator process, the user must provide a method that accepts a dual solution vector and returns variables with negative reduced cost. As with the cut generator, in parallel configurations, each variable is returned immediately to the LP module, rather than being passed back as a group once the function exits. This allows the LP to begin adding variables and solving the current relaxation before the variable generator is finished if desired. Parameters controlling if and when the LP should begin solving the relaxation before the variable generator is finished can be set by the user.

1.7 Parallelizing COIN/BCP

Because of the clear partitioning of work that occurs when the branching operation generates new subproblems, branch and bound algorithms lend themselves well to parallelization. As a result, there is already a significant body of research on performing branch and bound in parallel environments. We again point the reader to the survey of parallel branch and bound algorithms by Gendron and Crainic [11].

In parallel BCP, as in general branch and bound, there are two major sources of parallelism. First, it is clear that any number of subproblems on the current candidate list can be processed simultaneously. Once a subproblem has been added to the list, it can be properly processed before, during, or after the processing of any other subproblem. This is not to say that processing a particular node at a different point in the algorithm won't produce different results – it most certainly will – but the algorithm will terminate correctly in any case. The second major source of parallelism is to parallelize the processing of individual subproblems. By allowing separation to be performed in parallel with the solution of the linear programs, we can theoretically process a node in little more than the amount of time it takes to solve the sequence of LP relaxations. Both of these sources of parallelism can be easily exploited using the COIN/BCP framework.

The most straightforward parallel implementation, which is the one we currently employ, is a master-slave model, in which there is a central manager responsible for partitioning the work and parceling it out to the various slave processes that perform the actual computation. The reason we chose this approach is because it allows memory-efficient data structures for sequential computation and yet is conceptually easy to parallelized. This approach has limited scalability, but given the tradeoffs, we decided to accept that for the time-being. In future versions of the software, we hope to "decentralize" the implementation in order to allow better scalability. see [17] for an idea of how this could be done.

1.7.1 Parallel Execution and Inter-process Communication

COIN/BCP supports both a sequential and a parallel, distributed execution. All the user has to specify is the message passing protocol to be used. At the moment COIN/BCP has interfaces to the Parallel Virtual Machine (PVM) [20] protocol and to a single process protocol that is used to execute the algorithm sequentially. This latter protocol emulates being parallel thus there is no need to change anything in COIN/BCP or in the user code to do serial execution. (Granted, because of the emulation the serial code at the moment has significant overheads.) Theoretically COIN/BCP can utilize any third-party communication

protocol supporting dynamic spawning of processes and basic message-passing functions. All communication subroutines interface with COIN/BCP through a separate communications API. As mentioned above, currently PVM is the only message-passing protocol supported, but interfacing with another protocol is a straightforward exercise.

1.7.2 Fault Tolerance

Fault tolerance is an important consideration for solving large problems on networks whose nodes may fail unpredictably. The tree manager tracks the status of all processes and can restart them as necessary. It doesn't matter (too much) if a slave process is killed, the most that can be lost is the work that had been completed on that particular search tree node. Furthermore, new processors can be added to the parallel configuration on the fly and the TM process can spawn new slaves on those processes.

Chapter 2

Getting Started: Sample Compiling

Having familiarized yourself with the overall design of COIN/BCP in Chapter 1, you are now ready to get started with using COIN/BCP to develop applications of your own. The remainder of this manual is contains the technical details you need to successfully undertake this task. This chapter provides a description of how to get started with COIN/BCP. This is basically the same information contained in the README file that comes with the distribution.

Although COIN/BCP is inherently intended to be compiled and run on multiple architectures and across distributed networks, for now we do not use GNU's autoconf. The make files are designed to conveniently allow builds for multiple architectures within a single directory tree. This means that there may be a little hand configuring to do and you might need to know a little about your computing environment in order to make COIN/BCP compile. This should be limited to editing the make files and providing some path names. You may also have to live with some complaints from the compiler because of missing function prototypes, etc.

2.1 System Requirements

Currently, to obtain and compile COIN/BCP, you need to be running some version of Unix, preferably with the gcc/g++ compiler installed. Before you try to compile COIN/BCP,

you should first ensure that the version of gcc/g++ you are using is at least 2.95.1 by typing `gcc -v` on the command line. If you have an earlier version, COIN/BCP may not compile correctly – ask your sysadmin for an upgrade. If you're lucky enough to be your own sysadmin, then you're probably running Linux. The default version that comes with some Linux distributions, such as Redhat 6.2, is earlier than 2.95.1 so you should download and install a later version. This may also involve upgrading some other packages on which gcc depends.

There are a few auxiliary applications that you might also want to have installed. If you want to use CVS to download the code and update it automatically (highly recommended), then you should also install or ask for CVS to be installed. If you are unsure of what CVS is, please visit `www.cvs.org`. If you choose not to use CVS, then you can download the code as a tar file. The applications `tkcvs` and `tkdiff` provide a nice graphical interface to `cvs`. Finally, if you download `doxygen` (`www.doxygen.org`) and `qt` (`www.trolltech.com`), you will be able to automatically generate very nice documentation from the Bcp source code in an HTML format.

2.2 Obtaining the Source Code

2.2.1 Using CVS

To obtain the source code using CVS:

- Set the `CVSROOT` environment variable to be
  `:pserver:anonymous@oss.software.ibm.com:/usr/cvs/coin`
- Issue `cvs login` command with password `anonymous`
- Issue `cvs checkout MODULE` where `MODULE` is one of
  - `Bcp`: branch, cut, and price framework,
  - `Bcp-all`: BCP framework plus Osi and Vol
  - `Mkc`: multiple knapsack with color constraints (application),
  - `MaxCut`: Maximum weighted cut (application),
  - `mkc7`: large sample mps file for Mkc,
  - `Vol`: volume algorithm,
  - `Cgl`: cut generator library,

  - `Osi`: open solver interface,
  - `Dfo`: derivative free optimization,
  - `COIN`: to get all modules,

If you are just starting with Bcp, get the module `Bcp-all`. It will automatically get the two sample applications (`Mkc` and `MaxCut`), as well as the other necessary modules (`Osi` and `Vol`). Note that the directory `COIN/` will be installed as a subdirectory of whatever directory you issue the CVS commands from. The simplest thing to do is to issue the commands from your home directory and then enter the subdirectory `COIN/` to work with the source files.

2.2.2 Downloading a `tar` File

The tar files can be obtained from `www.coin-or.org`. Simply download the latest files and unpack them in your home directory. This will create a `COIN/` subdirectory, if one does not already exist, and place the source files in subdirectories of this directory. These files follow the same naming scheme as the CVS modules.

Note that the files in the repository have `.tar.gz` extension indicating that the archive file has been compressed using `gzip`. On U*ix systems use gunzip to uncompress them first, on Windows Winzip can uncompress and unpack the archive.

2.3 Initial compilation and testing

Since the whole project lives in the `COIN` directory in the rest of this section every path specified is relative to this directory.

- First, edit the file `Makefile.coin`. This is the global make file with settings used by all the software in the repository. It contains primarily pathnames and compiler settings. Uncomment the appropriate lines for whichever LP solvers you want to link in, and set the following variables as needed:
  - Uncomment whichever LP solver you want to use
  - `COINDIR`: the path to the `COIN/` directory

  - `OSLDIR`: the directory where OSL library is installed (if using OSL). Later on the Makefile makes the assumption that standard installation procedure was followed for OSL, i.e., the include files of OSL are in `$(OSLDIR)/includes` and the library is in `$(OSLDIR)/program`.
  - `XPRDIR`: same for XPRESS-MP.
  - `CPXDIR`: same for CPLEX

- You should not need to make any modifications to `Bcp/Makefile`.
- Now, edit the application specific make file. This is `Bcp/MaxCut/Makefile` for the MaxCut example. Here are the variables that you can set:
  - `USER_OPT`: options the compiler should use for compiling the user's code.
  - `COMM_PROTOCOL`: which message-passing protocol to use. Currently the options are `PVM` and `NONE` (compile as a serial application).
  - `USER_SRC_PATH`: a list of directories (besides `TM`, `LP`, etc. – see Section **??** for the directory structure) the user has source codes in.
- In addition, you will see that there are some other variables to set once you have written your own application. These include paths to your source files and the names of your source files. We will discuss how to construct this make file in more detail later in the manual.

2.3.1 Compiling for serial execution

As mentioned above, `COMM_PROTOCOL` must be set to `NONE` in the user application Makefile.

- First you have to create the volume library (if you plan to use the volume algorithm as your LP engine):
  - Change directory into `Faa/Vol` and type `make`
- Create the Osi library:
  - Change directory into `Osi` and type `make`
- Type "`make`" in the application directory. To start out, we recommend that you try to compile the `MaxCut` application in the directory `Bcp/MaxCut` application module. Dependency files will be created in `Bcp/Bcp-common/dep` and in `Bcp/MaxCut/dep`.

Object files will be placed in `Bcp/Bcp-common/$(ARCH)` and in `Bcp/MaxCut/$(ARCH)`, where $(ARCH) is the platform you run on combined with the optimization level (e.g, Linux-g or AIX-4.3-O). This latter directory also contains the executable named `bcps` (the "s" stands for being serial).

- To test the sample program type `$(ARCH)/bcps sample.par` on the command line. The sample parameter file and sample data file are included in the distribution.

2.3.2 Compiling for distributed networks

To use COIN/BCP on a network of computers fist a message passing library must be installed. At the moment COIN/BCP has an interface to the *Parallel Virtual Machine* (PVM) software only (an MPI interface is planned). The current version of PVM can be obtained at `www.ccs.ornl.gov/pvm`. It should compile and install without any problem. The user will have to set a few environment variable (such as `PVM_ROOT`), but this is all explained clearly in the PVM documentation. Note that there must be a link to the compiled executable from the `$PVM_ROOT/bin/$PVM_ARCH` directory in order for parallel processes to be spawned correctly.

The compilation procedure for COIN/BCP and the location of the files are almost identical to what's described in the previous subsection with the following differences:

- Set `COMM_PROTOCOL` to `PVM` in the user application Makefile.
- The executable name will be `bcpp`.
- Make a symbolic link from the `$(PVM_ROOT)/bin/$(PVM_ARCH)/` directory to the executable. This is required by PVM unless you override the default directory in your PVM hostfile.
- Start the PVM daemon by typing "`pvm`" on the command line and then typing "`quit`".
- To test the sample program first start up the pvm daemon on each of the machines you plan to use in the computation (how to do this is also explained in the PVM documentation) and then start COIN/BCP by issuing the `$(ARCH)/bcpp sample.par` command. The sample parameter file and sample data file are included in the distribution.

Now you should have successfully compile and execute the sample application. Once you

have accomplished this much, you are well on your way to having an application of your own.

Chapter 3

Developing Applications with COIN/BCP

3.1 Directory Layout (location of the source files)

The easiest way to get oriented is to examine the organization of the source files. Once you install the COIN/BCP distribution and enter the `Bcp` directory, you will see several subdirectories, including one called `Bcp-common/`. `Bcp-common/` contains the source files for the internal framework. Users should not have to modify these files, but should be familiar with their organization. The other directories contain the source code for applications. There are two samples available, a Max Cut solver (`MaxCut/`) and a solver for the Multiple Knapsack Problem with Color Constraints (`Mkc/`). First let's explore the Bcp-common directory.

Within `Bcp-common/`, there are directories associated with each of the modules, a directory called `Member/` and a directory (`include/`) for the header files. The source files in `Member/` contain the implementation of methods of classes that are used in multiple modules (like cuts and variables), while the module directories contain module specific class methods and module specific functions.

The same directory hierarchy is used in the directories of the sample applications and it is recommended that the user adheres to this convention simply because the `Bcp/Makefile` automatically looks for user source files in these directories. If the user places her files in different directories then she must specify those directories in `USER_SRC_PATH`.

3.2 Overview of the Class Hierarchy

We now briefly describe the class hierarchy from the user's point of view. Our aim here is not to describe the full class structure, but just those parts that the user needs to be familiar with in order to derive new user classes and override the appropriate methods. As we discussed in Chapter 1, we have taken an object-oriented approach with the main objects being the cuts and variables. As such, there are three main "object" classes from which the user may want to derive new problem-specific descendents. These are:

- `BCP_cut`: This class is used for describing cuts. There are three child classes derived from this one which implement the three different types of cuts—core, indexed, and algorithmic.
- `BCP_var`: This class is used for describing variables. Again, there are three derived classes which implement the three different types of variables—core, indexed, and algorithmic.
- `BCP_solution`: This class is used for describing feasible solutions to the full problem. This description may be implemented simply as a list of the nonzero variables and their corresponding values or can be a user-defined representation of a more combinatorial nature. For instance, in the Traveling Salesman Problem, the user may wish to store feasible solutions directly as permutations of the nodes instead of just as a list of edge variable indices and values.

Ideally, these would be the only classes the user would need to worry about. However, in order to modularize the code and support parallelism, we have deviated from an idealized, object-oriented design and defined some module-oriented "interface" classes as well. These classes do not contain any data elements, but instead contain methods which are unique to a particular module and cannot be contained in one of our primary object classes. They also contain methods which work on sets of objects—these cannot be implemented in the standard, object-oriented fashion. As an example, all the subroutines for communicating data between the modules. The interface classes are:

- `BCP_xx_user`: Here, "xx" is the name of a particular module. The user must derive a new class for each module in which she wants to override a default method. Note that although these base classes exist primarily as containers for module-specific methods, the user can also use his derived classes to store the problem data needed for performing the user-defined methods in that module. Alternatively, these data structures

could be defined in a separate base class and then, using multiple inheritance, derived into a common child class.

- `USER_initialize`: This class contains subroutines for generating objects from the derived interface classes. This is also where the messaging environment and LP solver environment objects are created.

In addition to deriving from these classes as appropriate, the user must also provide the definition of the function called `BCP_user_init()` which returns an object of the class derived from `USER_initialize`.

3.3 The Flow of the Algorithm

Keeping in mind this basic class structure, we now describe again, as in Chapter 1, the overall flow of the BCP algorithm, but this time with more detail and an emphasis on how the interface routines defined in the `BCP_xx_user` classes fit into this flow. In some cases, it may be important to know the specific order in which the interface routines are called since procedures performed in one subroutine could depend on data generated in a previous subroutine. This applies mainly to the LP module, which has the largest number of associated interface routines.

In Figures 3.1 and 3.2, the arrows between boxes indicate the flow of the algorithm. Keep in mind that these charts merely give a high-level description of the algorithm. Section 3.4 and the HTML documentation contain a full and detailed description of the API. Figure 3.1 indicates how the solver is initialized and lists the interface routines used by the tree manager, the cut generator, and the variable generator. First, the solver environment is initialized (see Section 3.4.1), then the module-specific user classes are created. Afterwards, the user specifies the initial set of core and extra variables and packs the problem data to be sent to the other modules. Initialization of each of the slave processes begins with the unpacking of these data at their destinations.

Once the solver is initialized, the TM module utilizes only three interface methods. The first one initializes a new phase (see Section 1.6.1), the second receives a new feasible solution, while the third compares two search tree nodes. This comparison function is used to insert new candidate nodes into a priority queue. The top element of the queue will be selected for processing when an LP process becomes available. Similarly, the CG and VG modules also call relatively few interface routines. Their only job is to wait for a primal (dual) solution from which they try to generate violated valid inequalities (improving columns).

The generated objects are sent back to the LP process.Figure (3.2) describes the flow of the LP solver loop, from where the majority of the interface routines are called. These routines will be described below.

3.3.1 Fathoming procedure

Fathoming, which is very simple in a regular branch and cut algorithm is much more involved when pricing is present. The reason is that introducing new columns can push the lower bound below the global upper bound or can restore feasibility if the LP relaxation was found infeasible. If a true optimal solution is desired in a BCP algorithm, then a search tree node can be fathomed if and only if there are no columns that can restore feasibility and there is no column with negative reduced cost. Of these two conditions the first one is the "worse". Almost always there are variables that can be introduced to restore feasibility, but usually that pushes the lower bound too high. However, we *must* restore feasibility, because there can be columns with negative reduced cost afterwards that could bring down the objective value. On the other hand, when fathoming would happen because of too high lower bound, all we got to look for are columns with negative reduced cost. Frequently none are found (especially if the global upper bound is really good) so the node can really be fathomed. For this reason it is recommended that users wanting to generate variables on the fly set up their model in a way that ensures primal feasibility at all times. (Not to mention that then she doesn't have to override the feasibility restoration methods.)

For each search tree node COIN/BCP maintains a state, that is, whether there are indexed or algorithmic variables not in the formulation. Furthermore, for indexed variables COIN/BCP can maintain a list about which ones have been permanently priced out (excluded from any solution in the subtree). To utilize this ability of COIN/BCP, only two very simple methods must be overridden: `next_indexed_var()` and `create_indexed_var()`. The first method is used to enumerate the indices of the indexed variables one by one, the second method is used to actually create the indexed variables.

Now if fathoming would happen because the lower bound exceeds the upper bound then, if COIN/BCP is instructed to maintain the above mentioned list, first the not-yet-priced-out indexed variables are tested then the `generate_vars_in_lp()` method is invoked for finding algorithmic variables with negative reduced cost.

If fathoming would happen because of infeasibility then again first the indexed variables tested whether any of them destroys the proof of infeasibility (i.e., whether it has a negative inner product with the specified dual ray) then the `restore_feasibility()` method is invoked so that the user can test the algorithmic variables.

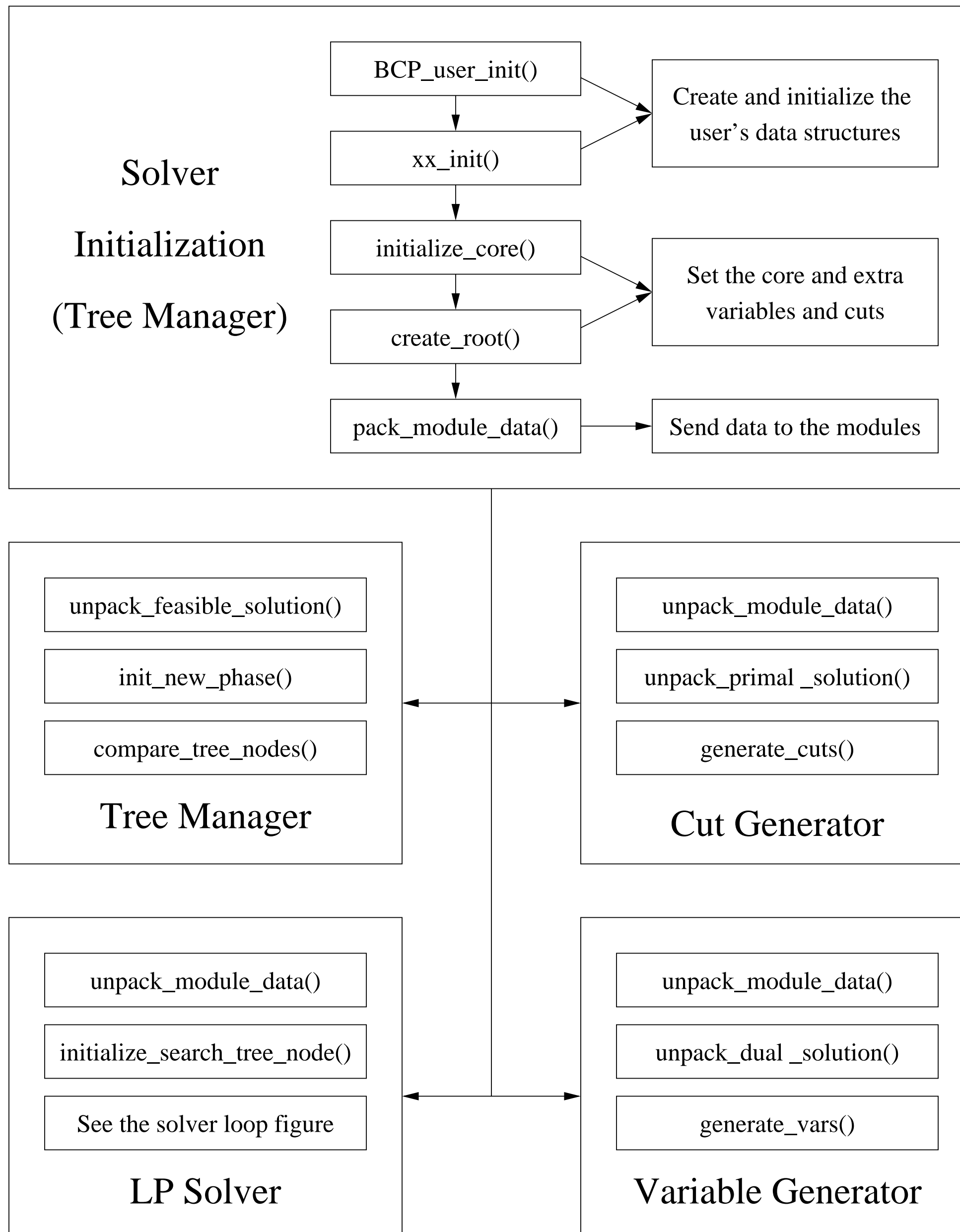

Figure 3.1: Solver initialization and algorithm overview

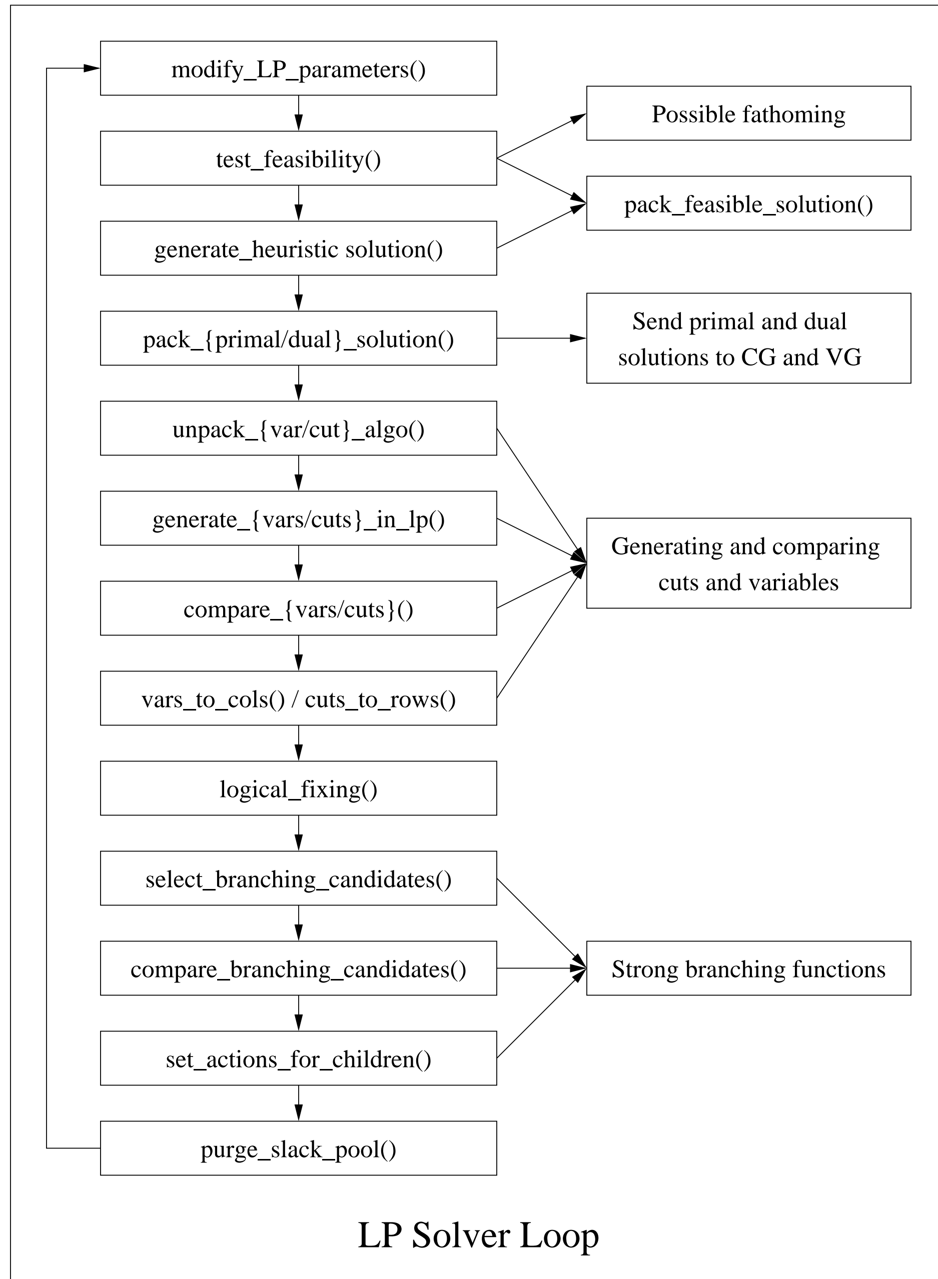

Figure 3.2: LP solver loop

3.4 Details of the Interface

As mentioned earlier in our overview of the class hierarchy (Section 3.2), the user can modify the behavior of the framework by overriding the default methods. To override the methods in a particular module, she simply derives a new child class from the corresponding `BCP_xx_user` base class and overrides the appropriate methods. If not overridden, the default method will be invoked by the framework. Whenever possible, methods have default which will work for the most common problem settings. In some cases, there are several default implementations from which the user can choose by setting a parameter. Alternatively, these methods can be invoked directly by the user as desired, allowing for the use of different methods in different situations. In the remainder of this section, we describe in more detail the virtual methods of the interface classes. These descriptions are at a high level—for the exact specification, see the HTML manual pages included with the distribution [?].

3.4.1 The `USER_initialize` class

The user must communicate the existence of the objects she designed to COIN/BCP. For example, for all processes she intends to use she must have derived something from the `BCP_xx_user` classes. Since COIN/BCP contains the `main()` function there are two ways to achieve this. The "C" style solution is to have a functions declared in COIN/BCP but not defined. These functions must be defined by the user and return pointers to objects defined by her. The disadvantage of this solution is that the user has to define all of these functions, even if she doesn't intend to create some type of objects. Furthermore, if there are possible defaults she must indicate somehow to the calling function that a default should be executed.

COIN/BCP employs a C++ style coding here. Only one function is declared that the user must define, `BCP_user_init()`. This function must return an object of a type she derived from `USER_initialize` and in which she overrode some methods. This way the user is forced to define only one function, and she can choose the default behavior (like initializing the LP engine class) by simply not overriding a method.

When COIN/BCP will be converted into a library there will be no need for this class.

- `msgenv_init()`: return the message passing environment to be used. The user probably doesn't want to override this method, as the default COIN/BCP uses will be determined by the value of `COMM_PROTOCOL` in the application Makefile.

- `tm_init()`: return the object the user has derived from `BCP_tm_user`. Note that this method should also take care of reading the parameter file and the problem data and whatever initialization the user wants to do. The user *must* override this method, there *must* be a user derived tree manager class.

- `lp_init()`: return the object the user has derived from `BCP_lp_user`. The user *must* override this method, there *must* be a user derived LP class.

- `cg_init()` and `vg_init()`: return the object the user has derived from the classes `BCP_cg_user` and `BCP_vg_user`. The user must override these if and only if she wants to generate cuts / variables.

3.4.2 The `BCP_tm_user` class

- `pack_module_data()`: in this method the user must pack the data that will be needed to perform computations in other modules. By default this method is empty and it is very likely that the user wants to override it.

  Note that this method should be overridden if and only if the `unpack_module_data()` methods of any other process is overridden.

- `unpack_feasible_solution()`: unpack a solution that is feasible to the problem. Not really necessary to override, it should only be done if the default generic solution (`BCP_solution_generic`) format is not good for the user for some reason. That format contains the description and value of all variables that are at nonzero level in the solution. The user may have a much more compact and intuitive representation of the solution, in which case she would override this method.

  By default a `BCP_solution_generic` object is unpacked.

  Note that this method should be overridden if and only if the corresponding method, `pack_feasible_solution()` in the class `BCP_lp_user` is overridden.

- `(un)pack_warmstart()`: (un)pack warmstarting information for a search tree node. The user probably doesn't want to override this method, as it should correspond to the LP solver selected. The default, just like for the LP engine, will be determined by the defined `COIN_USE_XXX` value.

- `(un)pack_var_algo()`: (un)pack an algorithmic variable. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic variable in which case she must override this method.

- `(un)pack_cut_algo()`: (un)pack an algorithmic cut. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic cut in which case she must override this method.
- `initialize_core()` and `create_root()`: the first of these two methods sets up the core of the problem (see Section 1.5.2) while the second specifies what extra cuts and variables should be present in the root node. By default the first method creates an empty core and the second method does not list any extra objects. Therefore to get something into the root node the user must override at least one of them.
- `init_new_phase()`: perform any necessary initialization before a new phase starts in the algorithm. Nothing is done as default. If the user does not do any pricing then the it is probably fine. Otherwise (since the column generation strategy must be specified in this method, too) the user must override it.
- `compare_tree_nodes()`: compare two search tree nodes. Return true if the first node should be processed before the second one. The default behavior is controlled by the `TreeSearchStrategy` parameter which is set to `BCP_BestFirstSearch` by default.

3.4.3 The `BCP_lp_user` class

This is by far the most complex class.

- `unpack_module_data()`: unpack the data packed for this process in the TM by the `pack_module_data()` method. By default this method is empty and it is very likely that the user wants to override it.

  Note that if this method is overridden then the TM's `pack_module_data()` method must be overridden, too.
- `(un)pack_warmstart()`: (un)pack warmstarting information for a search tree node. The user probably doesn't want to override this method, as it should correspond to the LP solver selected. The default, just like for the LP engine, will be determined by the defined `COIN_USE_XXX` value.
- `(un)pack_var_algo()`: (un)pack an algorithmic variable. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic variable in which case she must override this method.
- `(un)pack_cut_algo()`: (un)pack an algorithmic cut. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic cut in which case she must override this method.

- `initialize_solver_interface()`: return the LP engine. The user probably doesn't want to override this method as the default COIN/BCP uses will be determined by the defined `COIN_USE_XXX` value. However, it's possible that the user wants to define more than one LP solver and choose one based on a parameter. In this case she needs to override this method. In this case she should also override the `(un)pack_warmstart()` methods in the TM and LP user classes as well.

- `initialize_new_search_tree_node()`: do some preprocessing (e.g., logical tightening of bounds on variables and/or constraints) on the search tree node before it gets processed. By default this method is empty. A good candidate for overriding in column generation methods, since there branching information usually encodes some logic, thus implying significant tightening.

- `modify_lp_parameters()`: a chance to modify the parameters of the LP engine. By default this method is empty. Those experimenting with using different parameters in "regular" LP optimization and LP optimization in strong branching will want to override it.

- `test_feasibility()`: test whether the LP solution is feasible for the whole problem. If it is so then return a `BCP_solution` object. The default just tests whether all integrality requirements are met. (Actually there are several default options but they differ in their speed only by exploiting special knowledge, e.g., knowing that all variable must be binary.) If the user has her own representation of the solution she definitely wants to override it (the default method creates a `BCP_solution_generic` object). Also, she must override it if cuts are being generated as COIN/BCP has no way of knowing whether the not yet added cuts are all satisfied.

- `generate_heuristic_solution()`: try to come up with a good solution from the given LP solution. By default this method is empty. If the user has a quick heuristic it's worth to add it here since a good solution can drastically cut the size of the search tree.

- `pack_feasible_solution()`: the pair of `unpack_feasible_solution()` in the tree manager. Override neither or both. The default tries to treat and pack the solution argument as a `BCP_solution_generic` object and throws an exception if it is not such a solution.

- `pack_primal_solution()`: pack the information to be sent to the cut generator (this is usually the primal solution). The default method packs a selected set of variables along with their values. The selection is parameter driven, it can be everything, the nonzeros, the fractional values, etc. Override neither or both of this method and its

pair, `unpack_primal_solution()` in the cut generator. There is no reason to override it if no cut generator processes are started.

- `pack_dual_solution()`: pack the information to be sent to the variable generator (this is usually the dual solution). The default method packs a selected set of cuts along with their dual values. The selection is parameter driven, it can be everything, the nonzeros, the fractional values, etc. Override neither or both of this method and its pair, `unpack_dual_solution()` in the variable generator. There is no reason to override it if no variable generator processes are started.

- `display_lp_solution()`: display the result of most recent LP optimization. This method is invoked every time an LP relaxation is optimized and the user can display (or not display) it. By default the solution is displayed if the verbosity of COIN/BCP is high enough.

  This method exists mainly for debugging purposes. Few people would ever want to see all LP solutions. It's unlikely anyone would override this method.

- `next_indexed_var()` and `create_indexed_var()`: methods used if COIN/BCP is to maintain the list of indexed variables that are permanently priced out. The first method returns the user index of the variable whose index is the next one after the argument while the second method creates an indexed variable (and the corresponding column) given the index. By default they all throw exceptions. The user must override them if they are to be used.

- `restore_feasibility()`: These methods are invoked before fathoming a search tree node that has been found infeasible. If COIN/BCP maintains the list of indexed variables that are permanently priced out then by the time this method is invoked every indexed variable is tested whether it can destroy the proof of infeasibility and the user should look only for algorithmic variables. Otherwise (i.e., if COIN/BCP does not maintain the list) it is up to the user to check both indexed and algorithmic variables whether they "cut off" the dual rays.

- `cuts_to_rows()`: create the corresponding rows for a set of cuts with respect to the currently active variables. By default this method throws an exception (should not be called if not written). It must be overridden if cuts are generated.

- `vars_to_cols()`: create the corresponding columns for a set of variables with respect to the currently active cuts. By default this method throws an exception (should not be called if not written). It must be overridden if variables are generated.

- `generate_cuts_in_lp()`: generate cuts within the LP process. Sometimes too much information would need to be transmitted for cut generation (e.g., the full tableau for

Gomory cuts) or the cut generation is so fast that transmitting the info would take longer than generating the cuts. In such cases it might better to generate the cuts locally. This routine provides the opportunity. By default this method is empty (will be interfaced with Cgl).

- `generate_vars_in_lp()`: generate variables within the LP process. Sometimes too much information would need to be transmitted for variable generation or the variable generation is so fast that transmitting the info would take longer than generating the variables. In such cases it might be better to generate the variables locally. This routine provides the opportunity. By default this method is empty.

- `compare_cuts()`: compare two generated cuts. Cuts are generated in different iterations, they come from the Cut Pool, etc. There is a very real possibility that the LP process receives several cuts that are either identical or one of them is better then another (cuts off everything the other cuts off). This routine is used to decide which one to keep if not both. By default both cuts are kept. The user should override this method only if there is a significant chance that cuts will be regenerated.

- `compare_vars()`: compare two generated variables. Variables are generated in different iterations, they come from the Variable Pool, etc. There is a very real possibility that the LP process receives several variables that are either identical or one of them is better then another (e.g., almost identical but has much lower reduced cost). This routine is used to decide which one to keep if not both. By default both variables are kept. The user should override this method only if there is a significant chance that variables will be regenerated.

- `logical_fixing()`: this method provides an opportunity for the user to tighten the bounds of variables. The method is invoked after reduced cost fixing. By default this method is empty. For many problems there are possibilities for tightening the bounds based on logical inferences. The user should explore this.

- `select_branching_candidates()`: decide whether to branch or not and select a set of branching candidates if branching is decided upon. The return value of the method indicates what should be done: branching, continuing with the same node or abandoning the node completely. The default implementation branches if there are no cuts or variables waiting to be added to the formulation. In that case it selects variables for strong branching. A good branching rule can really speed up computation. It's probably worth to override this method and experiment.

- `compare_branching_objects()`: decide which one of two candidates should be selected for actual branching. The default implementation looks at the presolved ob-

jective values in the children and makes a decision based on those (the decision is parameter controlled). Probably the user is best off leaving this method alone.

- `set_actions_for_children()`: decide what to do with the children of the selected branching object. By default the possibility of diving is explored and then all or all but one (in case of diving) children are sent back to the tree manager. Probably the user is best off leaving this method alone.

- `purge_slack_pool()`: selectively purge the list of slack cuts. When a cut becomes ineffective and is eventually purged from the LP formulation it is moved into a slack pool. The user might consider these cuts later for branching. This function enables the user to purge any cut from the slack pool (those she wouldn't branch on anyway). Of course, the user is not restricted to these cuts when branching, this is only there to help to collect slack cuts. There are several default. The user probably doesn't want to override this method.

3.4.4 The `BCP_cg_user` class

This class is extremely simple. All it does is that it receives primal solutions and generates cuts from them. If there is no separate cut generator process the user doesn't need to derive a class from this one.

- `unpack_module_data()`: unpack the data packed for this process in the TM by the `pack_module_data()` method. By default this method is empty and it is very likely that the user wants to override it.

  Note that if this method is overridden then the TM's `pack_module_data()` method must be overridden, too.

- `unpack_primal_solution()`: unpack the information sent from the LP (this is usually the primal solution). The default method unpacks a set of variables along with their values. See the `pack_primal_solution()` of the LP process. Override neither or both of this and that method.

- `generate_cuts()`: do the actual cut generation. By default this method is empty. The user better override it otherwise why have a separate CG process?

- `unpack_var_algo()`: unpack an algorithmic variable. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic variable in which case she must override this method. Note that in the cut generator

there is no need to pack algorithmic variables. They are only received with the primal solution.

- `pack_cut_algo()`: pack an algorithmic cut. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic cut in which case she must override this method. Note that in the cut generator there is no need to unpack algorithmic cuts. They are only sent out to the LP process.

3.4.5 The `BCP_vg_user` class

This class is extremely simple. All it does is that it receives dual solutions and generates variables from them. If there is no separate variable generator process the user doesn't need to derive a class from this one.

- `unpack_module_data()`: unpack the data packed for this process in the TM by the `pack_module_data()` method. By default this method is empty and it is very likely that the user wants to override it.

  Note that if this method is overridden then the TM's `pack_module_data()` method must be overridden, too.

- `unpack_primal_solution()`: unpack the information sent from the LP (this is usually the dual solution). The default method unpacks a set of cuts along with their values. See the `pack_dual_solution()` of the LP process. Override neither or both of this and that method.

- `generate_vars()`: do the actual variable generation. By default this method is empty. The user better override it otherwise why have a separate VG process?

- `pack_var_algo()`: pack an algorithmic variable. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic variable in which case she must override this method. Note that in the variable generator there is no need to unpack algorithmic variables. They are only sent out to the LP process.

- `unpack_cut_algo()`: unpack an algorithmic cut. By default this method throws an exception since if it is invoked then the user must have generated an algorithmic cut in which case she must override this method. Note that in the variable generator there is no need to pack algorithmic cut. They are only received with the dual solution.

3.5 Deriving Problem-specific Classes

In this section, we give a rough explanation of the design decisions that have to be made and under what conditions the user needs to derive certain types of classes and override certain methods.

3.5.1 Generating cuts

In some cases, such as in pure branch and bound or branch and price, the user will not need to generate cutting planes dynamically, but for most applications, dynamic cut generation is critical to the efficiency of the algorithm. Assuming that the user has chosen to perform dynamic cut generation, he must decide between the two different types of cuts that can be dynamically generated—indexed, and algorithmic. As we have already discussed, there is no theoretical difference between these two types, but indexed cuts are more memory efficient since they do not have to be represented by a (possibly) bulky, abstract data structure. If it is possible to implement a particular class of cuts using an indexing scheme, this should generally be done. However, keep in mind that most classes of cuts cannot be implemented using indexing simply because they are too large to accomodate a workable indexing scheme.

For each class of cuts that the user wants to implement as an algorithmic class, it will be necessary to derive a new C++ class from `BCP_cut_algo` as a container for the data needed to construct the cut. In addition, the user needs to modify the `pack_cut_algo()` and `unpack_cut_algo()` methods in the appropriate `BCP_xx_user` classes. For indexed and core cuts, it is not necessary to derive a new class or implement packing and unpacking algorithms since all these cuts have a common representation.

With either algorithmic or indexed cuts, the user must also override the `cuts_to_rows()` and `compare_cuts()` methods in the `BCP_lp_user class`. The former specifies how to realize a given set of cuts as matrix rows with respect to the current set of variables while the latter is a function which determines if two cut objects actually represent the same cut. Of course, in addition, the user must also override either the `generate_cuts()` method of the `BCP_cg_user` class or the `generate_cuts_in_lp()` method of the `BCP_lp_user` class. The choice of whether to generate cuts in a separate cut generator or simply as part of the LP loop depends on the problem setting. In problems where generating cuts is relatively quick and the LP solver will be sitting idle waiting for the cut generator to return the cuts, it is easiest to simply generate them in the LP module itself. If cut generation is lengthy or requires large amounts of memory, then it is better to generate them in a separate generator.

3.5.2 Generating variables

Generally speaking, dynamic variable generation (often called column generation) is used less frequently than dynamic cut generation. If it is possible to efficiently generate all variables explicitly in the root node and there is enough memory to store them, this is generally the best thing to do. This allows variables to be fixed by reduced cost and nodes to be fathomed without expensive pricing (see the last paragraph). However, sometimes this is either not possible or not efficient because (1) there is not enough memory to store all of the variables in the matrix at once, (2) it is expensive to generate the variables, or (3) there is an efficient method of pricing large subsets of variables at once. There may also be other scenarios requiring variable generation.

In most ways, variable generation is similar to cut generation. However, there are some significant differences. While generating cuts helps tighten the formulation and increase the lower bound, generating variables has the opposite effect. Therefore, one must be somewhat careful about when variable generation takes place as it destroys monotonicity of the objective function, upon which algorithmic performance sometimes depends. In the last paragraph of this section, we also address the issue of variable generation prior to fathoming a search nodes, another important consideration.

As with cuts, the user must choose between the two different types of variables—algorithmic, and indexed. Again, there is no theoretical difference between these two types, but indexed variables are more memory efficient than algorithmic variables. To utilize algorithmic variables, the user should should derive a class or classes from `BCP_var_algo`, as with cuts. Also, the corresponding packing and unpacking methods need to be modified appropriately. For indexed variables, it is not necessary to derive a new class—the `BCP_var_indexed` class is provided for this purpose. In either case, the user must also override the `vars_to_cols()` and `compare_vars()` methods in the `BCP_lp_user class`. The former specifies how to realize a given set of variable as matrix columns with respect to the current set of cuts while the latter is a function which determines if two variable objects actually represent the same variable. As before, the user must also override either the `generate_vars()` method of the `BCP_vg_user` class or the `generate_vars_in_lp()` method of the `BCP_lp_user` class.

Our final consideration is that of fathoming. Before a node can be properly fathomed in BCP, it is necessary to ensure that there are no columns whose addition to the problem could reverse the conditions necessary for fathoming the node in question, i.e., by either lowering the objective function value back below the current upper bound or by restoring feasibility. For indexed variables, the framework can automatically keep track of which variables need to be priced out before the search tree node can be fathomed. In order for this option to be utilized, the user must provide the methods `next_indexed_var()` and

`create_indexed_var()`. If this scheme is not used, or the user is generating algorithmic variables, then the user's variable generation method should expend whatever effort is necessary to test whether there is a variable whose addition to the problem would lower the objective function value, i.e., a variable with negative reduced cost. Any such variable should be added to the problem before fathoming. In addition, the user should either ensure that all LP relaxation encountered are feasible (strongly encouraged) or implement the `restore_feasibility()` method. This method is when a node would be fathomed because of infeasibility, and the user is supposed to return new variables whose corresponding columns destroy the proof of infeasibility (i.e., have negative inner product with the known dual rays).

3.5.3 Setting the Core and Extra Object Lists

Recall that the core cuts and variables are those that are never removed from the problem. In some cases, significant savings can be achieved by properly choosing the list of core and extra objects well. To set the list of core objects, the user is required to override the `initialize_core()` methods in the `BCP_tm_user` class. There are important differences between the strategy for setting the list of core variables and that for setting the list of core cuts so we address each of these topics separately in what follows.

In the current implementation, the main advantage of putting a variable into the core is lower communication overhead and lower overhead for node creation in the tree manager and node setup in the LP module. Since variables in the core are present in every relaxation, information about them does not have to be communicated and stored along with each node description. Therefore, it is best to put into the core any variable that has a high probability of having a positive value in an optimal solution to the problem.

On the other hand, putting variables into the core that turn out not to be important can cause the size of the matrices for the subproblems to be bigger than necessary and can slow down the calaculation in other ways. It is important to realize that, although putting variables into the core does not prevent them from being fixed to zero by reduced cost (and in essence removed from the calculation), they must still be maintained as part of the matrix. In particular, when cuts are put in row form to be added to the matrix, the coefficients for these columns will have to be calculated, even though they are not part of the calculation.

For cuts, some of the same factors are at work, but there is more at stake, at least for simplex-based LP solvers. Although ineffective cuts can similarly be removed from the problem by changing the right hand side to $+\infty$ or $-\infty$, the number of rows that are

actually present in the matrix determines the size of the basis for the simplex method. The size of the basis contributes significantly to the overall running time of the simplex method. Hence, it is prudent to allow removal of ineffective rows as soon as possible. One reason for not allowing such removal is that it might be prohibitively difficult or expensive to regenerate the row if it was ever needed again. It might also be the case that some of the user's separation algorithms depend on the fact that the solution already satisfies some subset of inequalities. In this latter case, the most efficient way to guarantee this might be to simply leave those cuts in the problem at all times.

We have now seen the rationale for constructing the set of core objects. The user can also optionally specify that a designated subset of the extra cuts and variables (user indexed and/or algorithmic) should be initially present in the root, but not maintained as core objects. These variables and cuts are specified in the `create_root()` method of the `BCP_tm_user` class. The primary reason for designating these is that they are not important enough to become core variables, but would be too expensive to generate later, potentially over and over in various parts of the tree. With respect to variables, it is usually best to include as many of them as feasible in the root node. Provided that a good upper bound exists, they will get priced out of the problem quickly if they are not important. Also, their presence should not significantly slow down simplex-based LP solvers. The same does not apply to cuts, however. It is important to consider carefully the cuts that go into the base since these will determine the starting size of the basis for simplex-based LP solvers.

3.5.4 Branching

Next to effective cut and variables generation, strong branching is the function most critical to the efficiency of BCP. Fortunately, the framework takes care of most of the details. Furthermore, the defaults should work fine in most cases. For instance, one of the built-in defaults is to branch on the variable furthest from being integral (closest to .5 for 0-1 problems. This is an often-used method that will work fine for starting out. To implement his own branching scheme, the user has only to implement two functions in the BCP_lp_user class—`select_branching_candidates()` and `compare_branching_candidates()`. Based on knowledge of the problem's structure, the user must decide which objects (cuts and/or variables) to branch on. Unfortunately, there are not many rules of thumb here. The only way to find out what works best in a particular problem setting is trial and error.

3.5.5 Summary and Optional Methodss

In this subsection a summarized reference is provided for the classes and subroutines that need to be considered based on various design decisions. For each decision the methods to be implemented is listed. Optional methods not discussed in this chapter are also included. For more on those methods, please see the HTML documentation.

Perform cut generation

- Derive a class for each cut type from `BCP_cut_algo`.
- Override `generate_cuts_in_lp()` in `BCP_lp_user` class to generate cuts directly in the LP module.
- Override `generate_cuts()` in `BCP_cg_user` to generate cuts in a separate cut generation module.
- Override `cuts_to_rows()` in `BCP_lp_user`.
- Override `compare_cuts()` in `BCP_lp_user` class.
- Override `(un)pack_cut_algo()` in the appropriate `BCP_xx_user` classes.

Perform column generation

- Derive a class for each variable type from `BCP_var_algo`.
- Override `generate_vars_in_lp()` in `BCP_lp_user` to generate variables directly in the LP module.
- Override `generate_vars()` in `BCP_vg_user` to generate variables in a separate variable generation module.
- Override `vars_to_cols()` in `BCP_lp_user`.
- Override `compare_vars()` in `BCP_lp_user`.
- Override `(un)pack_var_algo()` in the appropriate `BCP_xx_user` classes.
- To use the built-in mechanism for tracking which indexed variables have been priced out, override `next_indexed_var()` and `create_indexed_var()` in `BCP_lp_user`.
- Ensure that all LP relaxations remain feasible or override `restore_feasibility()` in `BCP_lp_user`.

Customize strong branching

- Override `select_branching_candidates()`, `compare_branching_candidates()` and `set_actions_for_children()` in `BCP_lp_user`.

Set the problem core.

- Override `initialize_core()` in BCP_tm_user.

Create the root node.

- Override `create_root()` in BCP_tm_user.

Modify the LP solver parameters.

- Override `modify_lp_parameters()` in `BCP_lp_user`.

Define data structure to store and send feasible solutions.

- Derive a new solution class from BCP_solution.
- Override `(un)pack_feasible_solution()` in the classes `BCP_lp_user` (packing) and `BCP_tm_user` (unpacking).

Define data structure to send LP solutions.

- Override `(un)pack_{primal,dual}_solution()` in the classes `BCP_lp_user` (packing) and `BCP_{cg,vg}_user` (unpacking).

Use a primal heuristic to generate feasible solutions.

- Override `generate_heuristic_solution()` in `BCP_lp_user`.

Send problem-specific data to the modules.

- Override `(un)pack_module_data()` in the appropriate `BCP_xx_user` classes.

Display solutions in user-defined format.

- Override `display_xx_solution()` in `BCP_lp_user()` and/or `display_solution()` in `BCP_tm_user`.

Perform logical fixing of variables.

- Override `logical_fixing()` in `BCP_LP_user`.

3.6 Internal Data Structures

With few exceptions, the data structures used internally by COIN/BCP are undocumented and most users will not need to access them directly. However, if such access is desired, a pointer to the main data structure used by each of the modules can be obtained simply by calling the method `getXxProblemPointer()` of the `BCP_xx_user` class where `xx` is the

appropriate module. This method will return a pointer to the data structure for the appropriate module. Casual users are advised against modifying COIN/BCP's internal data structures directly.

3.7 Inter-process Communication

The implementation of COIN/BCP strives to shield the user from having to know anything about communications protocols or the specifics of inter-process communication. This is achieved by creating a `BCP_buffer` object and whenever user data needs to be passed from one process to another the user is asked to pack the data into this buffer on the sending side and to unpack the data from another buffer on the receiving side. Sending the data around and receiving it is entirely internal to COIN/BCP. Note that data must be unpacked in exactly the same order as it was packed, as data is read linearly into and out of the message buffer. The easiest way to ensure this is done properly is to simply copy the pack statements into the unpacking function and change the function names.

3.8 Debugging Your Application

3.8.1 The First Rule

COIN/BCP has many built-in options to make debugging easier. The most important one, however, is the following rule. **It is easier to debug the fully sequential version than the fully distributed version**. Debugging parallel code is not terrible, but it is more difficult to understand what is going on when you have to look at the interaction of several different modules running as separate processes. This means multiple debugging windows which have to be closed and restarted each time the application is re-run. Since the difference between compiling an application for serial and parallel execution is as little as changing a definition in the Makefile it is trivial to first compile a serial code, debug it and then compile for parallel execution. Make sure to set the `USER_OPT` flag to "`-g`" in the application Makefile.

3.8.2 Debugging with PVM

If you wish to venture into debugging your distributed application, then you simply need to set the parameter `DebugXxProcesses`, where `Xx` is the name of the module you wish to debug, to the value "1" (representing true) in the parameter file. This will tell PVM to spawn the particular process or processes in question under a debugger. What PVM actually does in this case is to launch the script `$PVM_ROOT/lib/debugger`. You will undoubtedly want to modify this script to launch your preferred debugger in the manner you deem fit. If you have trouble with this, please send e-mail to the mailing list (see Section **??**).

It's a little tricky to debug interacting parallel processes, but you will quickly get the idea. The main difficulty is in that the order of operations is difficult to control. Random interactions can occur when processes run in parallel due to varying system loads, process priorities, etc. Therefore, it may not always be possible to duplicate errors. To force runs that you should be able to reproduce, make sure to disable timeout during cut generation which is a major source of randomness. Furthermore, run with only one active node allowed at a time. This will keep the tree search from becoming random. These two steps should allow runs to be reproduced. You still have to be careful, but this should make things easier.

3.8.3 Using `Electric Fence`

The make file is already set up for compiling applications using `Electric Fence`. Instead of just typing `make` type `make ebcps`. The executable name is the same as described earlier, but with an "e" in front of it.

3.8.4 Using `Purify`

The make file is already set up for compiling applications using `Purify` on platforms where it is available. Make certain that the `purify` command is in your path and Instead of just typing `make` type `make pbcps`. The executable name is the same as described earlier, but with an "p" in front of it.

Chapter 4

Sample Application: The MKC Problem

In this chapter we describe how the solver for the MKC problem were implemented. This implementation is a sample for a column generation scheme; no cut generation is done. Since this problem is not so well known, first we will describe the problem setting then the implementation details.

4.1 The MKC Problem

MKC stands for *Multiple Knapsack problem with Color constraints* as it is derived by generalizing the multiple knapsack problem along two directions: (i) adding assignment restrictions on items which can be assigned to a knapsack, (ii) adding a new attribute (called "color") to the items and then adding the associated "color" constraints which restrict the number of distinct colors which can be assigned to a knapsack to two.

This problem is motivated by the surplus inventory matching problem in the steel industry ([**?**]): before planning production, an attempt is made to satisfy orders using leftover slabs from surplus inventory. The goal of inventory matching is to maximize the total weight of the orders satisfied from the leftover and to minimize the leftover weight of each slab used in the matching. For each order we can identify a set of applicable slabs from the surplus inventory. These assignment restrictions are based on quality and physical dimension considerations. For any given order only slabs which are of the same quality or better can be applied. In

addition, the thickness and width requirements for each order need to be compatible with those of the slab applicable. These considerations restrict the number of applicable slabs for each order. The color constraints place restrictions on the sets of orders that can be matched to the same slab in the surplus inventory. Because of processing considerations in the finishing line of a steel mill not all orders assignable to a slab can be packed together on the slab. There is a route associated with each order that specifies the set of process operations that need to be applied in the finishing mill. Orders with different routes require different process operations and are referred to as being of different types. Slabs packed with different order types need to be cut before they are processed in the finishing mill. Since cutting slabs is expensive and often the cutting machine is a bottleneck, strong constraints are posed in terms of the number of allowed cuts per slab. The simplest and most commonly used constraint used is to limit the number of required cuts to one; i.e., no more that two order types are allowed on a slab. In order to describe this constraint formally we associate a unique *color* with each route code and restrict the number of colors on a slab to be no more than two. Notice that this implies that we associate a color with each order based on its route code. This restricts the number of different order types on a slab to two and the number of required cuts to be no more than one.

4.2 Natural formulation for MKC

This formulation has three sets of variables and four sets of constraints modeling the various restrictions.

$$\max \sum_{i=1}^{N} \sum_{j \in N^i} w^i x_j^i - \sum_{i=1}^{N} (W_j - \sum_{j \in N^i} w^i x_j^i) z_j$$

$$\sum_{i \in N_j} w^i x_j^i \leq W_j z_j \qquad 1 \leq j \leq M \tag{4.1}$$

$$\sum_{j \in N^i} x_j^i \leq 1 \qquad 1 \leq i \leq N \tag{4.2}$$

$$\sum_{c \in C_j} y_j^c \leq 2 \qquad 1 \leq j \leq M \tag{4.3}$$

$$x_j^i \leq y_j^{c^i} \qquad 1 \leq i \leq N,\ j \in N_i \tag{4.4}$$

$$x_j^i \in \{0,1\} \qquad 1 \leq i \leq N,\ j \in N_i$$

$$y_j^c \in \{0,1\} \qquad \forall c \in C_j,\ 1 \leq j \leq M$$

$$z_j \in \{0,1\} \qquad 1 \leq j \leq M$$

Table 4.1: List of notations

N	: Total number of orders.
M	: Total number of slabs.
N^i	: Set of slabs incident to order i.
N_j	: Set of orders incident to slab j.
w^i	: Weight of order i.
W_j	: Weight of slab j.
C_j	: Set of colors incident on slab j.
c^i	: The color of order i.
x^i_j	: 1 if order i is assigned to slab j; 0 otherwise.
y^c_j	: 1 if orders of color c obtain material from slab j; 0 otherwise.
z_j	: 1 if any order is incident to slab j; 0 otherwise.

The total number of variables in this formulation is

$$\sum_{i=1}^{N} |N^i| + \sum_{j=1}^{M} |C_j| + M \quad = \quad \sum_{j=1}^{M} |N_j| + \sum_{j=1}^{M} |C_j| + M$$

while the total number of constraints is $2\sum_{i=1}^{N} |N^i| + 2M + N$.

Constraints (4.1) specify that if a slab is used then the total weight of the orders assigned to the slab cannot exceed the weight of the slab; Constraint (4.2) describes that each order will be made at most once; while constraints (4.3) and (4.4) enforce the coloring restriction.

Notice that the objective function is non-linear. However, since $z_j = 0$ forces x^i_j to be zero for all $i \in N_j$ and $z_j = 1$ implies $x^i_j z_j = x^i_j$, for all feasible solutions the objective function is equivalent to

$$\sum_{i=1}^{N} \sum_{j \in N^i} w^i x^i_j - \sum_{i=1}^{N} (W_j z_j - \sum_{j \in N^i} w^i x^i_j) = \sum_{i=1}^{N} \sum_{j \in N^i} 2w^i x^i_j - \sum_{i=1}^{N} W_j z_j$$

.

The final observation is that the objective function just combines the two stated goals (maximizing satisfied orders and minimizing wasted parts of slabs) with equal weights. This may or may not be the best composite objective, but this is how the creator of the application specified the problem. Also, all that a different composite weight would change is the multiplier 2 for w^i (the coefficient of x^i_j) and the multiplier 1 for W_j (the coefficient of z_j); nothing in the proposed algorithms would need to be changed.

4.3 A formulation suitable for column generation

This new formulation has significantly more columns than the original formulation, on the other hand it results in a well studied problem, the set packing problem ([**?**]).

There are two types of constraints in this formulation. The first type corresponds to the slabs in the problem, the second type to the orders. The variables represent feasible production patterns, that is, variable u has a 1 in the row corresponding to the slab the production pattern is to be made of and 1's in the rows corresponding to the orders in the production pattern. Each variable is a binary variable indicating whether that production pattern is chosen in the solution or not. Let us introduce the following notation:

- P is the set of feasible production patterns;
- P_j is the set of set of feasible patterns manufacturable from slab j;
- P^i is the set of set of feasible patterns containing order i;
- R_k is the row (constraint) corresponding to the slab the production pattern corresponding to u_k is made of; and
- R^k is the set of rows (constraints) corresponding to the orders in the production pattern corresponding to u_k.

Let the cost of variable u_k be $\bar{c}_k = \sum_{i \in R^k} 2w^i - W_{R_k}$ and create the following set packing problem:

$$\max \sum_{k \in P} \bar{c}_k u_k$$

$$\sum_{k \in P^i} u_k \leq 1 \qquad \forall 1 \leq i \leq N \tag{4.5}$$

$$\sum_{k \in P_j} u_k \leq 1 \qquad \forall 1 \leq j \leq M \tag{4.6}$$

$$u_k \in \{0, 1\} \qquad \forall k \in P$$

It is very easy to see that there is a one to one correspondence between the feasible solutions of this set packing problem and the feasible solutions of the original formulation. Moreover, the construction of the $\bar{c}$ cost vector ensures that the corresponding solutions have identical objective values. Therefore optimizing this problem is the same as optimizing the original formulation.

The obvious problem with this formulation is that the number of feasible production patterns is enormous.

4.3.1 Generating columns with positive reduced costs

To improve the solution evenly, for each slab we generate a production pattern whose corresponding column has the highest reduced cost, i.e., the most positive if there is one with positive reduced cost. Finding these columns is again a set of optimization problems, since for a dual vector π the reduced cost of variable u_k whose production pattern is made of slab j is simply

$$\bar{c}_k - \pi_j - \sum_{i \in R^k} \pi^i = \sum_{i \in R^k} 2w^i - W_j - \pi_j - \sum_{i \in R^k} \pi^i = -(W_j + \pi_j) + \sum_{i \in R^k} (2w^i - \pi^i) \tag{4.7}$$

and we want to maximize this over the set of production patterns that can be manufactured from slab j. For a fixed j the first term is constant. The feasible production patterns from slab j are those that satisfy the capacity and color constraints, thus this problem is equivalent to (using the notation from the original formulation):

$$\max \sum_j (2w^i - \pi_i) x^i_j \tag{4.8}$$

$$\sum_i w^i x^i_j \le W_j \tag{4.9}$$

$$\sum_i y_j^{c(i)} \le 2 \tag{4.10}$$

$$x^i_j \in \{0, 1\} \tag{4.11}$$

which is a knapsack problem with the side constraint that selected objects must have no more than two different colors. Moreover, the constant term in the reduced cost implies that we are only looking for production patterns whose reduced cost exceeds $W_j + \pi_j$. Since solving the LP relaxation of the knapsack problem (even with the side constraint) is rather simple, this required lower bound on the reduced cost can be very helpful in quickly concluding that there is no improving pattern for a particular slab.

4.3.2 Upper bounding

The previous subsection addresses the issue of how to solve the full LP relaxation by iteratively solving smaller LP relaxations and generating columns, but we need something

more. We need to be able to derive an upper bound on the optimal objective value of the full LP relaxation in every iteration. There are two reasons for this. The first is that in a Branch-and-Price algorithm we can fathom a search tree node if the upper bound on the optimal objective value of the LP relaxation at the node is already lower than the value of a currently known feasible solution. Since we may not be able to solve the subproblems that generate the columns (after all, even though the knapsack problem is considered relatively easy, it *is* NP-complete) we still want to have an upper bound for fathoming purposes. The second reason is that without an upper bound on the optimal value of the LP relaxation of the full problem we couldn't tell how close we are to optimality, we wouldn't have a proven gap.

Fortunately, upper bounding is very easy using Dantzig-Wolfe decomposition [**?**]. Since the sum of all variables in P_i is not more than 1, the objective value of the LP relaxation cannot change more than the highest reduced cost (or an upper bound on that value) as a result of changing the values of the variables in P_i. To get an upper bound on the reduced cost we can use the LP relaxation of the subproblem (the side constrained knapsack problem) which is very easy to solve. Now adding all "per slab" upper bounds to the optimal objective value of the current LP relaxation yields an upper bound on the optimum of the full LP relaxation.

4.3.3 Finding integral feasible solutions

Another advantage of the column generation based formulation is that it is very easy to generate feasible solutions. In each iteration we considered the fractional solution and started by including every variable above 0.5 in the solution. From the set of remaining fractional variables we exluded all that intersected the already selected variables. Whatever remained afterwards was always such a small set that we could solve the set packing problem on that set by enumeration.

4.4 Implementation details

4.4.1 Cuts, variables and solutions

First of all, we did not have to worry about anything cut generation related, since we were not generating cuts. Since the number of constraints is not too great (number of orders + number of slabs) we decided to treat all of them as core constraints, thus completely

eliminating the need to bother about cuts.

For the variables first we had to decide which ones are going to be core variables and which ones will be extra variables. Since we had no reason to believe that any one particular pattern was more likely to be in an optimal solution than some other pattern we decided not to have core variables at all (this also simplified coding somewhat). Since the variables are the feasible production patterns, they do not lend themselves to any enumeration scheme, so we decided not to have indexed variables either. Therefore all our variables are algorithmic ones. Actually, we had two kind of algorithmic variables, one for the production patterns and another one for branching, but we will discuss that latter in Section **??**. Both types of variables are derived from `BCP_var_algo` and are defined in `MKC_var.hpp`.

We have defined our solution class for two reasons. First, all our pattern variables are binary variables so there is no reason to include the value of the non-zero ones. Second, there might be branching variables (not pattern variables) that are at nonzero level as we go down in the tree, and we didn't want to include those in a feasible solution. Still, if we wanted to, we could have used a generic solution type. We just thought that using our own solution type makes the code clearer.

4.4.2 Branching

The problem with branching when generating columns is that we must be able to generate columns after branching, too. In other words, every generated column must conform to whatever branching decisions have been made to that point. That means that branching on a regular variable is out of question. On one side (when it is fixed to 1) we'd have great results, it would significantly shrink the search space. However, on the other side (fixing the variable to 0) the restriction is that we cannot regenerate that variable. But that variable will almost always "want to be regenerated" (definitely immediately after branching), since its reduced cost will make it attractive (after all, we have forcibly moved it away from where it ended up in the LP-optimal solution). So for our problem this would mean that after one branching we have to check the optimal solution to the knapsack subproblem and if it is the forbidden variable then we have to find the second best solution. After two branchings we may have to find the third best solution, etc. This is impossible.

Instead, the following logic is introduced. A branching object will specify whether a particular order O is manufactured from slab S or not.

4.4.3 Packing and unpacking

Packing and unpacking of user objects is really straightforward. For example, look at the `MKC_var_(un)pack()` functions. The packing function packs the type of each variable and invokes the pack member of the variables while the unpacking function unpacks the types and invokes the appropriate constructor.

In general, when an object is packed it is simply torn down to built-in types and those are packed. On the other side the date is unpacked in the same order and the appropriate objects are constructed. In the following subsection we will not mention the (un)packing member methods.

4.4.4 MKC_init

This is the implementation of the intializer class. The TM initializer reads in the problem and the parameters. Unfortunately this piece is rather complicated since the problem is specified as an MPS file and we have to extract order and slab information from it. The problem is loaded into the `kss` member of the `MKC_tm` class. See the `MKC_knapsack.hpp` file for data structures. Once the problem is read in a pointer to the `MKC_tm` class is returned. The LP initializer just returns a pointer to an empty `MKC_lp` object.

4.4.5 MKC_tm

There were only four methods (besides the (un)packing ones) we had to deal with. Initializing the core consisted of simply specifying the core cuts as we had no core variables (hence no core matrix). Since we had no core variables we had to add some extra variable in creating the root. There are two options for this, one is to add those variables that were read in from a file (maybe as a result of a previous run), or we could generate columns for the all zero dual solution (which is in some sense the optimal solution if we have no variables...). Displaying the solution has the option to test the solution that it really satisfies the original formulation (we have used this for debugging purposes) and then the solution is printed in two different ways. Finally, there will be only one phase and we will generate columns in it, so we set this in the `init_new_phase()` method.

4.4.6 MKC_lp

Chapter 5

Sample Application: The Maximum Cut Problem

In this cahpter, we describe the implementation of a sample application—a solver for the maximum cut problem. This application is a prototypical example of branch and cut, i.e., BCP with a fixed set of variables. No column generation is used in this implementation. This simplifies many of the basic tasks.

5.1 The Max Cut Problem

Given an undirected graph $G = (N, E)$ with edge weight function $\omega : E \to \mathbf{R}$, the Maximum Cut Problem (MCP) is that of partitioning the nodes into two subsets in such a way that the total weight of the edges in the cut separating the two sets is maximized. This is a well-known problem—several branch and cut algorithms for dense graphs have been presented in [2, 8]. In the following description, we consider complete graphs only. For a complete graph undirected graph with n nodes, a linear relaxation of the integer programming problem is givem by

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & \\ & x_{ij} + x_{jk} + x_{ik} \leq 2 \quad \forall (i,j,k) \in N^3 && (5.1) \\ & x_{ij} - x_{jk} - x_{ik} \leq 0 \quad \forall (i,j,k) \in N^3 && (5.2) \\ & 0 \leq x_{ij} \leq 1 \quad \forall (i,j) \in E && (5.3) \end{aligned}$$

(5.4)

Here x_{ij} takes value 1 if the edge (i, j) appears in the cut, and 0 otherwise. Constraints (5.1) - (5.2) are called the *triangle inequalities* and they define facets of the cut polytope (see [3]).

Another set of inequalities, which is a superset of (5.1) - (5.2), is the following. Let C be a cycle and $F \subseteq C$ with $|F| = 2k + 1$. Then

$$\sum_{e \in F} x_e - \sum_{e \in C \setminus F} x_e \leq |F| - 1 \tag{5.5}$$

is a valid inequality. This follows from the fact that the intersection of a cycle and a cut has even cardinality. Note, that although this set of inequalities include those in (5.1) - (5.2), the polytope defined by these is the same, i.e., these inequalities can be derived from those in (5.1) - (5.2) (see [3]).

A polynomial time separation algorithm for this class of inequalities has been given in [3]. However we use a faster heuristic as follows. Let $\bar{x}$ be the fractional solution we want to separate and define weights

$$w_e = c_e \cdot \max(\bar{x}_e, 1 - \bar{x}_e). \tag{5.6}$$

Then find a maximum weighted spanning tree T with weights w. For an edge $e \in T$, if $\bar{x}_e \geq 1 - \bar{x}_e$ then the end-nodes of e should be on opposite sides of the cut—we give a label "A" to this edge. Otherwise, if $\bar{x}_e < 1 - \bar{x}_e$ then the end-nodes should be on the same side of the cut, and we give the label "B" to this edge. Once every edge in T has been labeled we have a heuristic cut K. For each edge $e \notin T$, we add it to T and look at the cycle C that is created. If $e \in K$, we test the violation of an inequality (5.5), where the set F is given by the A-edges. If $e \notin K$ the set F is given by the A-edges and the edge e. Although, as we noted above, inequalities (5.5) are implied by (5.1) - (5.2), we use (5.5) because our simple separation heuristic is faster than enumerating triangles.

5.2 Implementation

Because the size of the problems we can currently solve is small, we can easily include all the edge variables explicitly. Hence, we do not need to consider dynamiuc column generation. Hence, we do not need to concern ourselves with the `BCP_vg_user` class or the `BCP_var_algo` class. To simplify things further, we decided not to use a separate cut generator either. This is usually a good approach when cut generation is relatively inexpensive. It is also a good

idea during initial development since it makes debugging much easier. Because we are not using a separate cut generator, we do not need to consider the `BCP_cg_user` class either.

As with virtually any BCP implementation, we will need to consider the `BCP_tm_user` and `BCP_lp_user` classes. Also, because we will be dynamically generating algorithmic cuts, we will need to derive a new class to represent the cycle cuts (5.5) from the class `BCP_cut_algo`. Finally, we will need to derive a new class for describing the feasible solutions from `BCP_solution`. In the remainder of the section, we provide a high-level description of each of these classes. The reader is encouraged to look at the source code and the HTML documentation for more detail. See Chapter 2 for information on getting and examining the source code and documentation.

5.2.1 MC_tm

This is the class derived from the `BCP_tm_user` class. This class is derived for the purpose of overriding a variety of functions that we need to customize. These consist mainly of routiines that pass data between the processes during parallel execution and the routines for describing the problem core and root node. Below, we list each function and describe how it was re-implemented.

- `unpack_feasible_solution()`: This subroutine exists to unpack the user-defined solution class described in Section 5.2.3. The corresponding `pack_feasible_solution()` routine will be described in Section 5.2.2. Also, see Section 5.2.3 for a description of how the feasible solutions are represented.
- `pack_module_data()`: Here, we are packing the data that needs to be sent to the LP process. This consists of the number of nodes and a list of the edges in the graph. The corresponding `unpack_module_data()` routines is described in Section 5.2.2.
- `pack_cut_algo()`: Here, we pack the cycle cuts to be sent to the LP solver. The corresponding `unpack_cut_algo()` routines is described in Section 5.2.2.
- `unpack_cut_algo()`: Here, we unpack the cycle cuts that are received from the LP solver. The corresponding `pack_cut_algo()` routines is described in Section 5.2.2.
- `initialize_core()`: Essentially for convenience and ease of implementation, we place all the variables in the core. This is possible since we are not using column generation, but may not be the most efficient method. None of the cuts are placed in the core since we don't have an inherently important subset that we know should never be removed from the problem.

- `create_root()`: To initialize the root node, we use some heuristics to generate an initial set of cycle cuts. However, as noted before, these are "extra" cuts and do not get put into the core. They may be removed later in the calculation.

- `display_feasible_solution()`: This routine is used essentially to display the solutions in a more "user-friendly" way, instead of simply as a list of variable indices and values. See Section 5.2.3 for a description of how the feasible solutions are represented.

5.2.2 `MC_lp`

This is the class derived from the `BCP_lp_user` class. Again, this class is derived for the purpose of overriding a variety of functions that we need to customize. These consist not only of routiines that pass data between the processes, as before, but also routines for generating cuts and performing strong branching. Below, we list each function and describe how it was re-implemented.

- `unpack_module_data()`: Here, we unpack the data sent from the TM. This data is stored in a user-defined class called `MC_problem`.

- `pack_cut_algo()`: Here, we pack the cycle cuts to be sent to the TM. The cuts are represented as a list of edges—first the edges in the set F, then the edges not in F. To contruct the corresponding valid inequality, we need only determine which edges variables are present in the relaxation. See the description of `cuts_to_rows()` below.

- `unpack_cut_algo()`: Here, we unpack the cycle cuts that are received from the TM along with the description of a subproblem.

- `modify_lp_parameters()`: Here, we modify the LP parameters before solution of the relaxation commences.

- `test_feasibility()`: Because integrality of the solution is not enough to imply feasibility, we needed to override this method. If it is found that the solution is integral but not feasible, then cuts proving the infeasibility are easy to derive and are added to the LP relaxation, allowing the solution process to continue.

- `pack_feasible_solution()`: Here, any feasible solutions that are found are packed and sent to the TM for storage. See Section 5.2.3 for a description of how the feasible solutions are represented.

- `cuts_to_rows()`: This subroutine generates the rows of the current LP relaxation corresponding to the cuts to be added. For cycle cuts, this consists simply of determining which of the edge variables that have a positive coefficient in the cycle cut, i.e., the variables corresponding to the edges of the corresponding cycle, are active in the current subproblem. For each variable corresponding to an edge that is in the cycle cut and also active in the subproblem, the corresponding matrix coefficient must be added to the row description.
- `compare_cuts()`: This routine simply compares two cuts and determines if they are the same cut. In the case of cycle cuts, this is straightforward.
- `generate_cut_in_lp()`: This is the subroutine that generates the cuts to be added to the relaxation. A description of the algorithm for generating the cycle cuts was given in Section 5.1.
- `select_branching_candidates`: Here, we select the edges to be branched on. We branch basically on edges that are have high cost and are close to value .5.

5.2.3 `MC_solution`

Although feasible solutions to this problem consist of a set of edges, they can be more compactly represented as simply a list of the nodes contained in either shore of the cut. This user-defined class is used for representing the solutions in this more compact, intuitive fashion.

5.2.4 `MC_cycle_cut`

This class was derived from `BCP_cut_algo` to contain the representation of the cycle cuts (5.5). They are stored simply as a list of edges in the cycle. The list is in two parts—first, the edges in the set F are listed and then those not in the set F. Of course, the cardinality of the set F has to be stored as well.

5.2.5 Other Classes

There are a number of other classes that we have defined to hold data used during the solution process. Please see the HTML documentation and the source code itself for a list of these.

Bibliography

[1] *ABACUS: A Branch and Cut System.*
http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html.

[2] F. Barahona, M. Jünger, and G. Reinelt. Experiments in quadratic 0-1 programming. *Mathematical Programming*, 44:127–137, 1989.

[3] F. Barahona and A.R. Mahjoub. On the cut polytope. *Mathematical Programming*, 36:157–173, 1986.

[4] *BC-OPT.*
http://.

[5] *Bob.*
http://www.prism.uvsq.fr/optimize/softs/bob.en.html.

[6] *COIN-OR: Common Optimization Interface for Operations Research.*
http://www.coin-or.org.

[7] *CONCORDE.*
http://www.keck.caam.rice.edu/concorde.html.

[8] C. De Simone and G. Rinaldi. A cutting plane algorithm for the max-cut problem. *Optimization Methods and Software*, 3:195–214, 1994.

[9] M. Eső, L. Ladányi, T.K. Ralphs, and L.E. Trotter. Fully parallel generic branch-and-cut. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, March 14-17 1997.

[10] *FATCOP.*
http://www.cs.wisc.edu/~ferris/fatcop.html.

[11] B. Gendron and T.G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–??, 1994.

[12] *MINTO*.
http://akula.isye.gatech.edu/~mwps/projects/minto.html.

[13] *MIPO*.
http://www.columbia.edu/~sc244/abstract.html#tmipo.

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

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

[16] *PARINO*.
http://www-unix.mcs.anl.gov/~linderot/projects/pio.html#PARINO.

[17] *PICO*.
http://www.cs.sandia.gov/~caphill/proj/pico.html.

[18] *PPBB-LIB*.
http://www.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/introduction.html.

[19] *PUBB*.
http://al.ei.tuat.ac.jp/~yshinano/pubb/index.html.

[20] *PVM: Parallel Virtual Machine*.
http://www.epm.orml.gov/pvm/pvm_home.html.

[21] *SYMPHONY: Single- or Multi-Process Optimization over Networks*.
http://www.branchandcut.org/SYMPHONY.