Branch, Cut, and Price: Sequential and Parallel [*]

T.K. Ralphs[1], L. Ladányi[2], and L.E. Trotter, Jr.[3]

[1] Department of Industrial and Manufacturing Systems Engineering, Lehigh University, Bethlehem, PA 18017, `tkralphs@lehigh.edu`, `www.lehigh.edu/~tkr2`
[2] Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, `ladanyi@us.ibm.com`
[3] School of OR&IE, Cornell University, Ithaca, NY 14853, `ltrotter@cs.cornell.edu`

Abstract. Branch, cut, and price (BCP) is an LP-based branch and bound technique for solving large-scale discrete optimization problems (DOPs). In BCP, both cuts and variables can be generated dynamically throughout the search tree. The ability to handle constantly changing sets of cuts and variables allows these algorithms to undertake the solution of very large-scale DOPs; however, it also leads to interesting implementational challenges. These lecture notes, based on our experience over the last six years with implementing a generic framework for BCP called SYMPHONY (Single- or Multi-Process Optimization over Networks), address these challenges. They are an attempt to summarize some of what we and others have learned about implementing BCP, both sequential and parallel, and to provide a useful reference for those who wish to use BCP techniques in their own research.

SYMPHONY, the software from which we have drawn most of our experience, is a powerful, state-of-the-art library that implements the generic framework of a BCP algorithm. The library's modular design makes it easy to use in a variety of problem settings and on a variety of hardware platforms. All library subroutines are generic—their implementation does not depend on the problem-setting. To develop a full-scale BCP algorithm, the user has only to specify a few problem-specific methods such as cut generation. The vast majority of the computation takes place within a "black box," of which the user need have no knowledge. Within the black box, SYMPHONY performs all the normal functions of branch and cut—tree management, LP solution, cut pool management, as well as inter-process communication (if parallelism is used). Source code and documentation for SYMPHONY are available at `http://branchandcut.org/SYMPHONY`.

1 Introduction

Since the inception of optimization as a recognized field of study in mathematics, researchers have been both intrigued and stymied by the difficulty of solving many of the most interesting classes of discrete optimization problems. Even combinatorial problems, though conceptually easy to model as integer programs, have long remained challenging to solve in practice. The last two decades have seen tremendous progress in our ability to solve large-scale discrete optimization problems. These advances have culminated in the approach that we now call *branch and cut*, a technique (see [45, 72, 50]) which brings the computational tools of branch and bound algorithms together with the theoretical tools of polyhedral combinatorics. In 1998, Applegate, Bixby, Chvátal, and Cook used this technique to solve a *Traveling Salesman Problem* instance with 13,509 cities, a full order of magnitude larger than what had been possible just a decade earlier [2] and two orders of magnitude larger than the largest problem that had been solved up until 1978. This feat 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 number of cities. Hence, we are talking about solving a problem with roughly *100 million variables.*

There are several reasons for this impressive progress. Perhaps the most important is the dramatic increase in available computing power over the last decade, both in terms of processor speed and memory. This increase in the power of hardware has subsequently facilitated the development of increasingly sophisticated software for optimization, built on a wealth of theoretical results. As software development has become a central theme of optimization research efforts, many theoretical results have been "re-discovered" in light of their new-found computational importance. Finally, the use of parallel computing has allowed researchers to further leverage their gains.

Because of the rapidly increasing sophistication of computational techniques, one of the main difficulties faced by researchers who wish to apply these techniques is the level of effort required to develop an efficient implementation. The inherent need for incorporating problem-dependent methods (most notably for dynamic generation of variables and constraints) has typically required the time-consuming development of custom implementations. Around 1993, this led to the development by two independent research groups of software libraries aimed at providing a generic framework that users could easily customize for use in a particular problem setting. One of these groups, headed by Jünger and Thienel, eventually produced ABACUS (A Branch And CUt System) [52], while the other, headed by the authors and Ladányi, produced what was then known as COMPSys (Combinatorial Optimization Multi-processing System). After several revisions to enable more broad functionality, COMPSys became SYMPHONY (Single- or Multi-Process Optimization over Networks) [78, 76]. A version of SYMPHONY, which we will call COIN/BCP, has also been produced at IBM under the COIN-OR project [27]. The COIN/BCP package takes substantially the same approach and has the same functionality as SYMPHONY, but has extended SYMPHONY's capabilities in some areas, as we will point out.

These lecture notes are based on our experience over the last six years with implementing the SYMPHONY framework and using it to solve several classical combinatorial optimization problems. At times, we will also draw on our experience with the COIN/BCP framework mentioned earlier. What follows is intended to summarize some of what we and others have learned about implementing BCP algorithms and to provide a concise reference for those who wish to use branch and cut in their own research.

2 Related Work

The past decade has witnessed development of numerous software packages for discrete optimization, most of them based on the techniques of branch, cut, and price. The packages fell into two main categories—those based on general-purpose algorithms for solving mixed integer programs (MIPs) without the use of special structure *crossreference Martin* and those facilitating the use of special structure by interfacing with user-supplied, problem-specific subroutines. We will call packages in this second category *frameworks*. There have also been numerous special-purpose codes developed for use in particular problem settings.

Of the two categories, MIP solvers are the most common. Among the many offerings in this category are MINTO [70], MIPO [10], bc-opt [26], and SIP [67]. Generic frameworks, on the other hand, are far less numerous. The three frameworks we have already mentioned (SYMPHONY, ABACUS, and COIN/BCP) are the most full-featured packages available. Several others, such as MINTO, originated as MIP solvers but have the capability of utilizing problem-specific subroutines. CONCORDE [2, 3], a package for solving the *Traveling Salesman Problem* (TSP), also deserves mention as the most sophisticated special-purpose code developed to date.

Other related software includes several frameworks for implementing parallel branch and bound. Frameworks for general parallel branch and bound include PUBB [85], BoB [16], PPBB-Lib [87], and PICO [35]. PARINO [65] and FAT-COP [21] are parallel MIP solvers.

3 Organization of the Manuscript

In Sect. 4, we briefly describe branch, cut, and price for those readers requiring a review of the basic methodology. In Sect. 5, we describe the overall design of SYMPHONY without reference to implementational details and with only passing reference to parallelism. In Sect. 6, we then move on to discuss details of the implementation. In Sect. 7, we touch on issues involved in parallelizing BCP. Finally, in Sect. 8 and Sect. 9, we discuss our computational experience, with both sequential and parallel versions of the code. In these sections, we describe the implementation of solvers for two combinatorial optimization models, the *Vehicle Routing Problem* and the *Set Partitioning Problem*. We point out and explain those features and parameters that have been the most important. We also address the effectiveness of parallelism.

4 Introduction to Branch, Cut, and Price

In the remainder of this document, we discuss the application of BCP algorithms to the solution of *discrete optimization problems*. A discrete optimization problem (DOP) can be broadly defined as that of choosing from a finite set S an *optimal* element s^* that minimizes some given *objective function* $f : S \to \mathbf{R}$ ($\mathbf{R}$ will denote the set of all real numbers, and $\mathbf{Z}$ the set of all integers). DOPs arise in many important applications such as planning, scheduling, logistics, telecommunications, bioengineering, robotics, and design of intelligent agents, among others. Most DOPs are in the complexity class $\mathcal{NP}$-complete, so there is little hope of finding provably efficient algorithms [40]. Nevertheless, intelligent search algorithms, such as LP-based branch and bound (to be described below), have been tremendously successful at tackling these difficult problems [50].

4.1 Branch and Bound

Branch and bound is the broad class of algorithms from which branch, cut, and price has evolved. A branch and bound algorithm uses a divide and conquer strategy to partition the solution space into *subproblems* and then optimizes individually over each subproblem. For instance, let S be the set of solutions to a given DOP, 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. Using branch and bound, we initially examine the entire solution space S. In the *processing* or *bounding* phase, we relax the problem. In so doing, we admit solutions that are not in the feasible set S. Solving this relaxation yields a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of S or has cost equal to that of $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify n subsets $S_1, \ldots, S_n$ of S, such that $\cup_{i=1}^n S_i = S$. Each of these subsets is called a *subproblem*; $S_1, \ldots, S_n$ are sometimes called the *children* of S. We add the children of S to the list of *candidate subproblems* (those which await processing). This is called *branching*.

To continue the algorithm, we select one of the candidate subproblems and process it. There are four possible results. If we find a feasible solution better than $\hat{s}$, then we replace $\hat{s}$ with the new solution and continue. We may also find that the subproblem has no solutions, in which case we discard (*prune*) it. Otherwise, we compare the lower bound for the subproblem to our global upper bound, given by the value of the best feasible solution encountered thus far. If it is greater than or equal to our current upper bound, then we may again prune the subproblem. Finally, if we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of active candidates. We continue in this way until the list of candidate subproblems is empty, at which point our current best solution is, in fact, optimal.

4.2 Branch, Cut, and Price

In many applications, the bounding operation is accomplished using the tools of linear programming (LP), a technique described in full generality, e.g., by Hoffman and Padberg [50]. 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 an *LP relaxation*, which is then solved to obtain a lower bound for the problem. In [72], Padberg and Rinaldi improved on this basic idea 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. This technique was called *branch and cut*. Since then, many implementations (including ours) have been fashioned after the ideas 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 $c \in \mathbf{R}^E$. The incidence vectors corresponding to the members of $\mathcal{F}$ are sometimes specified 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. Now let $\mathcal{P}$ be the convex hull of incidence vectors of members of $\mathcal{F}$. Then we know by Weyl's Theorem (see [71]) that there exists a finite set $\mathcal{L}$ of inequalities valid for $\mathcal{P}$ such that

$$\mathcal{P} = \{x \in \mathbf{R}^n : ax \leq \beta \ \forall \ (a, \beta) \in \mathcal{L}\} = \{x \in \mathbf{R}^n : Ax \leq b\}. \tag{1}$$

The inequalities in $\mathcal{L}$ are the potential *constraints*, or *cutting planes*, to be added to the relaxation as needed. Unfortunately, it is usually difficult, if not impossible, to enumerate all of the inequalities in $\mathcal{L}$, else we could simply solve the problem using linear programming. Instead, they are defined implicitly and we use separation algorithms and heuristics to generate these inequalities when they are violated. In Fig. 1, we describe more precisely how the bounding operation is carried out in a branch and cut algorithm for combinatorial optimization.

Once we have failed to either prune the current subproblem or separate the current *relaxed solution* from $\mathcal{P}$, we are forced to branch. The branching operation is usually 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 Fig. 2. Figure 3 gives a high level description of the generic branch and cut algorithm.

As with constraints, 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, but can also be viewed as that of generating constraints for the dual of the current LP relaxation. Hence, LP-based branch and bound algorithms in which the variables are generated

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 and the corresponding relaxed solution $\hat{x}$, or (3) a message `pruned` indicating that the subproblem should not be considered further.

Step 1. Set $\mathcal{C} \leftarrow \mathcal{L}'$.

Step 2. If the LP $\min\{cx : ax \leq \beta \ \forall \ (a, \beta) \in \mathcal{C}\}$ is infeasible, then STOP and output `pruned`. This subproblem has no feasible solutions.

Step 3. Otherwise, consider the LP solution $\hat{x}$. If $c\hat{x} < \alpha$, then go to Step 4. Otherwise, STOP and output `pruned`. This subproblem cannot produce a solution of value better than α.

Step 4. 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^*.

Step 5. Otherwise, apply separation algorithms and heuristics to $\hat{x}$ to obtain 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 $c\hat{x}$.

Step 6. Otherwise, set $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$ and go to Step 2.

Fig. 1. Bounding in the branch and cut algorithm for combinatorial optimization

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 $\mathcal{S}$.

Fig. 2. Branching in the branch and cut algorithm

Generic Branch and Cut Algorithm

Input: A data array specifying the problem instance.

Output: A 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 $B \leftarrow \{\mathcal{S}^I\}$.

Step 3. If $B = \emptyset$, STOP and output $\hat{s}$ as the global optimum s^*. Otherwise, choose some $\mathcal{S} \in B$. Set $B \leftarrow B \setminus \{\mathcal{S}\}$. Apply the bounding procedure to $\mathcal{S}$ (see Fig. 1).

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.

Fig. 3. Description of the generic branch and cut algorithm

dynamically are known as *branch and price* algorithms. In [15], Barnhart et al. provide a thorough review of these methods.

When both variables and constraints are generated dynamically during LP-based branch and bound, the technique is known as *branch, cut, and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of constraints and that of variables. We further examine this symmetry later in these notes. For now, 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 themselves have many commonalities), combining the two techniques requires much more sophisticated methods than either requires alone. This is an important theme in what follows.

In our descriptions, we will often use the term *search tree*. This term derives from the representation of the list of subproblems as the nodes of a graph in which each subproblem is connected only to its parent and its children. Storing the subproblems in such a form is an important aspect of our global data structures. Since the subproblems correspond to the nodes of this graph, they will sometimes be referred to as *nodes in the search tree* or simply as *nodes*. The *root node* or *root* of the tree is the node representing the initial subproblem.

5 Design of SYMPHONY

In the remainder of these notes, we will illustrate general principles applicable to implementing BCP by drawing on our experience with SYMPHONY. We thus begin with a high-level description of the framework. SYMPHONY was designed with two major goals in mind—ease of use and portability. 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 the framework to perform *effectively* in all these settings. Our primary measure of effectiveness was how well the framework would perform in comparison with a problem-specific (or hardware-specific) implementation written "from scratch."

The reader should be mindful of the fact that achieving such design goals involves a number of difficult tradeoffs, which we highlight throughout the rest of this text. For instance, ease of use is quite often at odds with efficiency. In many instances, we had to sacrifice some efficiency in order 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. Sequential and shared-memory platforms demand memory-efficient data structures in order to maintain the very large search trees that can be generated. When moving to distributed platforms, these storage schemes do not scale well to large numbers of processors. This is further discussed in Sect. 7.1.

5.1 An Object-oriented Approach

As we have already remarked, 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 numbers of constraints and variables that need to be accounted for during the solution process. The dynamic nature of the algorithm requires that we must also be able to efficiently move constraints and variables in and out of the *active set* of each search node at any time. A second, closely-related challenge is that of effectively dealing with the very large search trees that can be generated for difficult problem instances. This involves not only the important question of how to store the data, but also how to move it between modules during parallel execution. A final challenge in developing a generic framework, such as SYMPHONY, 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 constraints and variables are initially *active* in the subproblem. In fact, the vast majority of the methods in BCP that depend on the model are related to generating, manipulating, and storing the constraints and variables. Hence, SYMPHONY can be considered an object-oriented framework with the central "objects" being the constraints and variables. From the user's perspective, implementing a BCP algorithm using SYMPHONY 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.

With this approach, we achieved the "black box" structure by separating these problem-specific functions from the rest of the implementation. The internal library interfaces with the user's subroutines through a well-defined Application Program Interface (API) and independently performs all the normal functions of BCP—tree management, LP solution, and cut pool management, as

well as inter-process communication (when parallelism is employed). Although there are default options for many of the operations, the user can also assert control over the behavior of the algorithm by overriding the default methods or by manipulating the parameters.

Although we have described our approach as being "object-oriented," we would like to point out that SYMPHONY is implemented in C, not C++. To avoid inefficiencies and enhance the modularity of the code (allowing for easy parallelization), we used a more "function-oriented" approach for the implementation of certain aspects of the framework. For instance, methods used for communicating data between modules are not naturally "object-oriented" because the type of data being communicated is usually not known by the message-passing interface. It is also common that efficiency considerations require that a particular method be performed on a whole set of objects at once rather than on just a single object. Simply invoking the same method sequentially on each of the members of the set can be inefficient. In these cases, it is far better to define a method which operates on the whole set at once. In order to overcome these problems, we have also defined a set of *interface functions*, which are associated with the computational modules (Sect. 5.3).

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 (constraints and variables) 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 constraints 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 *constraint* and *row* are often used interchangeably. Similarly, *variable* and *column* are often used with the same meaning. In many situations, this is appropriate and does not cause confusion. However, in object-oriented BCP frameworks, such as SYM-PHONY or ABACUS [52], a *constraint* and a *row* are *fundamentally different objects*. A *constraint* (also referred to as a *cut*) 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 constraints*. This distinction between the *representation* and the *realization* of objects is a crucial design element that allows us to effectively address some of the challenges inherent in BCP. In the remainder of this section, we further discuss this distinction and its implications.

Variables. In SYMPHONY, problem variables are *represented* by a unique global index assigned to each variable by the user. This index indicates 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 constraints, the user must be able to generate the corresponding column to be added to the matrix. As an 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 (when viewed as ordered pairs of nodes).

This 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 does not present a problem. However, for other models such as airline crew scheduling (discussed below), for instance, the number of columns may not be known in advance. Even if the number of columns is known in advance, a viable indexing scheme may not be evident. Eliminating the indexing requirement by allowing variables to have abstract, user-defined representations (such as we do for constraints, as described in the next section), would allow for more generality, but would also sacrifice some efficiency. A hybrid scheme, allowing the user to have both indexed and *algorithmic* variables (variables with user-defined representations) has been implemented in COIN/BCP and is also planned for a future version of SYMPHONY.

For efficiency, the problem variables can be divided into two sets, the *core variables* and the *extra variables*. The core variables are active in all subproblems, whereas the extra variables can be freely added and removed. There is no theoretical difference between core variables 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 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. Core variables selection in the case of the Vehicle Routing Problem will be illustrated in Sect. 8.1. For a detailed description of core variable selection in the case of the Traveling Salesman Problem, see *crossreference Elf et al.* In addition to the core variables, the user can also designate other variables that should be active in the root subproblem. Often, it is useful to activate these variables in the root, as it is likely they will be priced out quickly anyway. When not using column generation, all variables must be active in the root node.

Constraints. Because the global list of potential constraints is not usually known a priori or is extremely large, constraints cannot generally be represented simply by a user-assigned index. Instead, each constraint is assigned a global index only after it becomes active in some subproblem. It is up to the user, if desired, to designate a compact *representation* for each class of constraints that is to be generated and to implement subroutines for converting from this compact representation to a matrix row, given the list of active variables. For instance, suppose that the set of variables with nonzero coefficients in a particular class

of constraints corresponds to the set of edges across a cut in a graph. Instead of storing the index of each variable and its corresponding coefficient explicitly, one can simply store the set of nodes on one side ("shore") of the cut as a bit array. The constraint can then be constructed easily for any particular set of active variables (see Sect. 8.1 for more on this example).

Just as with variables, the constraints are divided into *core constraints* and *extra constraints*. The core constraints are those that are active in every subproblem, whereas the extra constraints can be generated dynamically and are free to enter and leave as appropriate. Obviously, the set of core constraints must be known and constructed explicitly by the user. Extra constraints, on the other hand, are generated dynamically by the cut generator as they are violated. As with variables, a good set of core constraints can have a significant effect on efficiency.

Note that the user is not *required* to designate a compact representation scheme. Constraints can simply be represented explicitly as matrix rows with respect to the global set of variables. However, designating a compact form can result in large reductions in memory use if the number of variables in the problem is large.

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. A critical aspect of implementing BCP is the maintenance of a complete description of the current basis (assuming a simplex-based LP solver) for each node to allow a warm start to the computation. This basis is either inherited from the parent, computed during strong branching (see Sect. 6.2), or comes from earlier partial processing of the node itself (see Sect. 6.3). Along with the set of active objects, we must also store the identity of the object which was branched upon to generate the node. The branching operation is described in Sect. 6.2.

Because the set of active objects and the status of the basis 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 6.3). This allows efficient generation of the next node to be processed.

5.3 Modular Implementation

SYMPHONY's functions are grouped into five independent computational modules. This modular implementation not only facilitates code maintenance, but also allows easy and highly configurable parallelization. Depending on the computational setting, the modules can be compiled as either (1) a single sequential code, (2) a multi-threaded shared-memory parallel code, or (3) separate processes running over a distributed network. The modules pass data to each other

either through shared memory (in the case of sequential computation or shared-memory parallelism) or through a message-passing protocol defined in a separate communications API (in the case of distributed execution). A schematic overview of the modules is presented in Fig. 4. In the remainder of the section, we describe the modularization scheme and the implementation of each module in a sequential environment. We defer serious discussion of issues involved in parallel execution of the code until Sect. 7.

The Master Module. The *master module* includes functions that perform problem initialization and I/O. These functions implement the following tasks:

- 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 sending data for the root node to the *tree manager*.
- 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 the message that the algorithm has finished and print out data.
- Ensure that all modules are still functioning.

The Tree Manager Module. The *tree manager* controls the overall execution of the algorithm. It tracks the status of all modules, as well as that of the search tree, and distributes the subproblems to be processed to the LP module(s). Functions performed by the tree manager module are:

- Receive data for the root node and place it on the list of candidates for processing.
- 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 modules when it changes.
- Write current state information out to disk periodically to allow a restart in the event of a system crash.
- Keep track of run data and send it to the master program at termination.
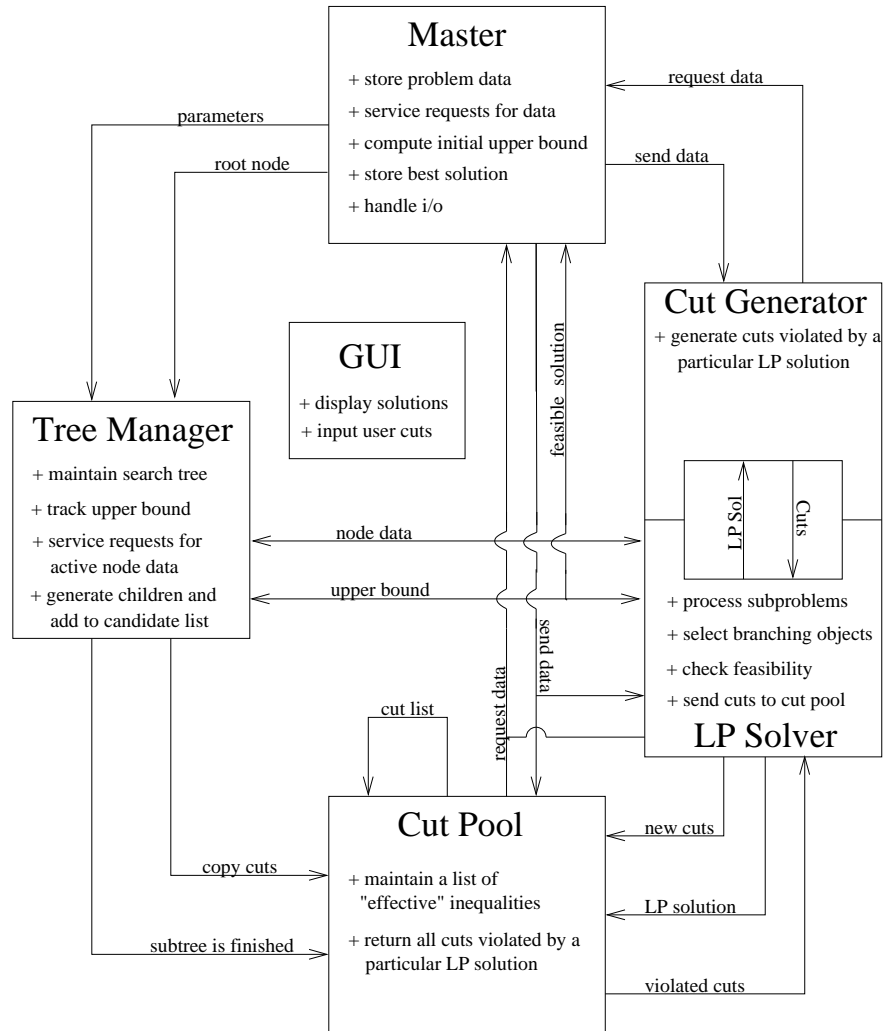
The Modules of Branch, Cut, and Price

Fig. 4. Schematic overview of the branch, cut, and price algorithm

The Linear Programming Module. The *linear programming* (LP) module is the most complex and computationally intensive of the five modules. Its job is to use linear programming 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 and process it in conjunction with the cut generator and the cut pool.
- Decide which cuts should be sent to the global pool to be made available to other LP modules.
- If necessary, choose a branching object and send its description back to the tree manager.
- Perform the fathoming operation, including generating variables.

The Cut Generator Module. The *cut generator* performs only one function— generating valid inequalities violated by the current LP solution and sending them back to the requesting LP module. 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 Cut Pool Module. The concept of a *cut pool* was first suggested by Padberg and Rinaldi [72], and is based on the observation that in BCP, the inequalities which are generated while processing a particular node in the search tree are generally globally valid and potentially useful at other nodes. Since generating these cuts is sometimes a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree thus far for use in processing future subproblems. Hence, the cut pool functions as an auxiliary cut generator. More explicitly, the functions of the cut pool module are:

- Receive cuts generated by other modules and store them.
- Receive an LP solution and return a set of cuts which this solution violates.
- Periodically purge "ineffective" and duplicate cuts to control its size.

5.4 SYMPHONY Overview

Currently, SYMPHONY is 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 [43].

The master 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 sent to the tree manager to initialize the list of candidate nodes. The tree manager in turn 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 (see Sect. 6.2 for a description of the branching operation). The preference ordering for processing nodes is a run-time parameter. Typically, the node with the smallest lower bound is chosen to be processed next, since this "best-first" 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 Sect. 6.3, extend the basic best-first approach.

We mentioned earlier that cuts and variables can be treated in a somewhat symmetric fashion. However, it should be clear by now that our current implementation favors branch and cut algorithms, where the computational effort spent generating cuts dominates that of generating variables. Our methods of representation also clearly favor such problems. In a future version of the software, we plan to eliminate this bias by adding additional functionality for handling variable generation and storage. This is the approach already taken in COIN/BCP [27]. For more discussion of the reasons for this bias and the differences between the treatment of cuts and variables, see Sect. 6.2.

6 Details of the Implementation

6.1 The Master Module

The primary functions performed by the master module were listed in Sect. 5.3. If needed, the user must provide a routine to read problem-specific parameters in from a parameter file. Also suggested is a subroutine for upper bounding, though upper bounds can also be provided explicitly. A good initial upper bound can dramatically decrease the solution time by allowing more variable-fixing (see Sect. 6.2 and also *crossreference Elf et al*) 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 useful (see Sect. 6.3 for details). The user's

only unavoidable obligation during preprocessing is to specify the list of core variables and, if desired, the list of extra variables that are to be active in the root node. Again, we point out that selecting a good set of core variables can make a marked difference in solution speed, especially when using the two-phase algorithm.

6.2 The Linear Programming Module

The LP module is at the core of the algorithm, as it performs the computationally intensive bounding operations for each subproblem. A schematic diagram of the LP solver loop is presented in Fig. 5. The details of the implementation are discussed in the following sections.
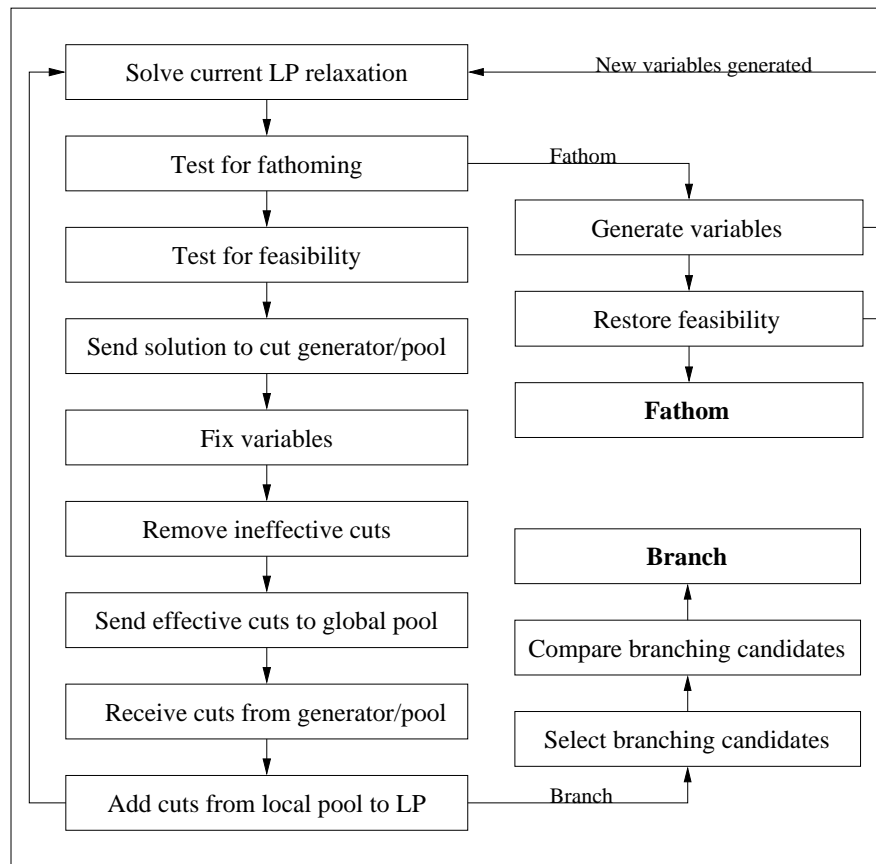

Fig. 5. Overview of the LP solver loop

The Linear Programming Engine. SYMPHONY 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. As with the user functions, SYMPHONY communicates with the LP engine through an API that converts SYMPHONY's internal data structures into those of the LP engine. Currently, the framework will only work with advanced, simplex-based LP engines, such as CPLEX [49], since the LP engine must be able to accept an advanced basis, and provide a variety of data to the framework during the solution process. The internal data structures used for maintaining the LP relaxations are similar to those of CPLEX and matrices are stored in the standard column-ordered format.

Managing the LP Relaxation. The majority of the computational effort of BCP 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 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, where ineffective is defined as either (1) the corresponding slack variable is positive, (2) the corresponding slack variable is basic, or (3) the dual value corresponding to the row is zero (or very small). Cuts that have remained effective in the LP for a specified number of iterations are sent to the global pool where they can be used in later search nodes. Cuts that have been purged from the LP can be made active again if they later become violated.

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 fixed by reduced cost. That is, the LP reduced cost is examined in an effort to determine whether fixing that variable at one of its bounds would remove improving solutions; if not, the variable is fixed and removed from consideration. For a more detailed description of the conditions for fixing and setting variables by reduced cost, see *crossreference Elf et al.* If the matrix is *full* at the time of the fixing, meaning that all unfixed variables are active, then the fixing is permanent for that subtree. Otherwise, it is temporary and only remains in force until the next time that columns are dynamically generated.

Because SYMPHONY was originally designed for combinatorial problems with relatively small numbers of variables, techniques for performing dynamic column generation are somewhat unrefined. Currently, variables are priced out sequentially by index, which can be costly. To improve the process of pricing

variables, we plan to increase the symmetry between our methods for handling variables and those for handling cuts. This includes (1) allowing user-defined, abstract representations for variables, (2) allowing the use of "variable generators" analogous to cut generators, (3) implementing both global and local pools for variables, (4) implementing heuristics that help determine the order in which the indexed variables should be priced, and (5) allowing for methods of simultaneously pricing out large groups of variables. Much of this is already implemented in COIN/BCP.

Because pricing is computationally burdensome, it currently takes place only either (1) before branching (optional), or (2) when a node is about to be pruned (depending on the phase—see the description of the two-phase algorithm in Sect. 6.3). To use dynamic column generation, the user must supply a subroutine which generates the column corresponding to a particular user index, given the list of active constraints in the current relaxation. When column generation occurs, each column not currently active that has not been previously fixed by reduced cost is either priced out immediately, or becomes active in the current relaxation. Only a specified number of columns may enter the problem at a time, so when that limit is reached, column generation ceases. For further discussion of column generation, see Sect. 6.3, where the two-phase algorithm is described.

Since the matrix is stored in compressed form, considerable computation may be needed to add and remove rows and columns. Hence, rows and columns are only physically removed from the problem when there are sufficiently many to make it "worthwhile." Otherwise, deleted rows and columns remain in the matrix but are simply ignored by the computation. Note that because ineffective rows left in the matrix increase the size of the basis unnecessarily, it is usually advisable to adopt an aggressive strategy for row removal.

Branching. Branching takes place whenever either (1) both cut generation and column generation (if performed) have failed; (2) "tailing off" in the objective function value has been detected (see *crossreference Elf et al* for a description of tailing off); or (3) the user chooses to force branching. Branching can take place on cuts or variables and can be fully automated 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 SYMPHONY to do so automatically by using one of several built-in strategies, such as branching on the variable whose value is farthest from being integral. The number of candidates may depend on the level of the current node in the tree—it is usually best to expend more effort on branching near the top of the tree.

After the list of candidates is selected, each candidate is *pre-solved*, by 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. This procedure of using exploratory LP informa-

tion in this manner to select a branching candidate is commonly referred to as *strong branching*. When the branching object has been selected, the LP module 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 module should process next. This issue is further discussed below.

6.3 The Tree Manager Module

The tree manager's primary job is to control the execution of the algorithm by deciding which candidate node should be chosen as the next to be processed. This is done using either one of several built-in rules or a user-defined rule. Usually, the goal of the search strategy is to minimize overall running time, but it is sometimes also important to find good feasible solutions early in the search process. In general, there are two ways to decrease running time—either by decreasing the size of the search tree or by decreasing the time needed to process each search tree node.

To minimize the size of the search tree, the strategy is to select consistently that candidate node with the smallest associated lower bound. In theory, this strategy, sometimes called *best-first*, will lead the smallest possible search tree. However, we need to consider the time required to process each search tree node as well. This is affected by both the quality of the current upper bound and by such factors as communication overhead and node set-up costs. When considering these additional factors, it will sometimes be more effective to deviate from the best-first search order. We discuss the importance of such strategies below.

Search Chains and Diving. 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 module 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. When this number is low, there may be 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 advantage of often leading quickly to the discovery of feasible solutions, as discussed above. Good upper bounds not only allow earlier pruning of unpromising search chains, but also should decrease the time needed to process each search tree node by allowing variables to be fixed by reduced cost.

The Two-Phase Algorithm. If no heuristic subroutine is available for generating feasible solutions quickly, then a unique two-phase algorithm can also be invoked. In the two-phase method, the algorithm is first run to completion on a specified set of core variables. Any node that would have been pruned in the first phase is instead sent to a pool of candidates for the second phase. If the set of core variables is small, but well-chosen, this first phase should be finished quickly and should result in a near-optimal solution. 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 is 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 thus priced out can be eliminated from the problem globally. If we are successful at pricing out all of 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 leaf 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 for which each child has lower bound 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 avoid the task of processing each child individually.

6.4   The Cut Generator Module

To implement the cut generator module, the user must provide a function 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 returned within a group of cuts when the function terminates. This allows the LP to begin adding cuts and re-solving the current relaxation before the cut generator is finished, if desired. Parameters controlling if and when the LP should begin solving the new relaxation before the cut generator is finished can be set by the user.

6.5   The Cut Pool Module

Maintaining and Scanning the Pool. The cut pool's primary job is to receive a solution from an LP module and return cuts from the pool that are violated by it. The cuts are stored along with two pieces of information—the level of the tree on which the cut was generated, known simply as the *level* of the cut, and the number of times it has been checked for violation since the last time it was actually found to be violated, known as the number of *touches*. The number of touches can be used as a simplistic measure of its effectiveness. Since the pool can get quite large, the user can choose to scan only cuts whose number of

touches is below a specified threshold and/or cuts that were generated on a level at or above the current one in the tree. The idea behind this second criterion is to try to avoid checking cuts that were not generated "nearby" in the tree, as they are less likely to be effective. Any cut generated at a level in the tree below the level of the current node must have been generated in a different part of the tree. Although this is admittedly a naive method, it has proven reasonably effective in practice.

On the other hand, the user may define a specific measure of quality for each cut to be used instead. For example, the degree of violation is an obvious candidate. This measure of quality must be computed by the user, since the cut pool module has no knowledge of the cut data structures. The quality is recomputed every time the user checks the cut for violation and a running average is used as the global quality measure. The cuts in the pool are periodically sorted by this measure and only the highest quality cuts are checked each time. All duplicate cuts, as well as all cuts whose number of touches exceeds or whose quality falls below specified thresholds, are periodically purged from the pool in order to limit computational effort.

Using Multiple Pools. For several reasons, it may be desirable to have multiple cut pools. When there are multiple cut pools, each pool is initially assigned to a particular node in the search tree. After being assigned to that node, the pool services requests for cuts from that node and all of its descendants until such time as one of its descendants is assigned to another cut pool. After that, it continues to serve all the descendants of its assigned node that are not assigned to other pools.

Initially, the first pool is assigned to the root node. All other pools are unassigned. During execution, when a new node is selected for processing, the tree manager must determine which pool will service the node. The default is to assign the same pool as that of its parent. However, if there is currently an idle pool (either it has never been assigned to any node or all the descendants of its assigned node have been processed or reassigned), then that cut pool can be assigned to the new node. The new pool is initialized with all the cuts currently in the cut pool of the parent node, after which the two pools operate independently on their respective subtrees. When generating cuts, the LP module sends the new cuts to the cut pool assigned to service the node during whose processing the cuts were generated.

The primary motivation behind the idea of multiple cut pools is as follows. First, we want simply to limit the size of each pool as much as possible. By limiting the number of nodes that a cut pool has to service, the number of cuts in the pool will be similarly limited. This not only allows cut storage to be spread over multiple machines, and hence increases the available memory, but at the same time, the efficiency with which the cut pool can be scanned for violated cuts is also increased. A secondary reason for maintaining multiple cut pools is that it allows us to limit the scanning of cuts to only those that were generated in the same subtree. As described above, this helps focus the search and should

increase the efficiency and effectiveness of the search. This idea also allows us to generate locally valid cuts, such as the classical Gomory cuts (see [71]).

7 Parallelizing 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 refer the reader to the survey of parallel branch and bound algorithms by Gendron and Crainic [43], as well as other references such as [35, 46, 80, 57].

In parallel BCP, as in general branch and bound, there are two major sources of parallelism. First, it is clear that any group 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. For instance, 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 perform the more expensive of these two operations. Alternatively, it is also possible to separate over several classes of cuts simultaneously. However, computational experience has shown that savings from parallelizing cut generation are difficult to achieve at best. Nonetheless, both of these sources of parallelism can be easily exploited using the SYMPHONY framework.

The most straightforward parallel implementation, 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. This approach was adopted because it allows memory-efficient data structures for sequential computation and yet is conceptually easy to parallelize. Unfortunately, this approach has limited scalability. We discuss design tradeoffs involving scalability in the next section.

7.1 Scalability

Overview of Scalability. We now digress slightly to discuss the importance of *scalability* in parallel algorithm development. Generally speaking, the scalability of a parallel system (the combination of a parallel algorithm and a parallel architecture) is the degree to which it is capable of efficiently utilizing increased computing resources (usually additional processors). To assess this capability, we compare the speed with which we can solve a particular problem instance in parallel to that with which we could solve it on a single processor. The *sequential running time* (T_0) is used as the basis for comparison and is usually taken to be

the running time of the best available sequential algorithm. The *parallel running time* (T_p) is the running time of the parallel algorithm in question and depends on p, the number of processors available. The *speedup* (S_p) is the simply the ratio T_0/T_p and hence also depends on p. Finally, the efficiency (E_p) is the ratio S_p/p of speedup to number of processors.

In general, if the problem size is kept constant, efficiency drops as the number of processors increases—this is a product of the fact that there is a fixed fraction of work that is inherently sequential in nature (reading in the problem data, for example). This *sequential fraction* limits the theoretical maximum speedup (see [1]). However, if the number of processors is kept constant, then efficiency generally *increases* as problem size increases [58, 46, 47]. This is because the sequential fraction becomes smaller as problem size increases. Thus, we generally define scalability in terms of the rate at which the problem size must be increased with respect to the number of processors in order to maintain a fixed efficiency. For more details, see [57].

Scalability for BCP. In order to maintain high parallel efficiency, it is critical not only to keep each processor busy, but to keep each processor busy with *useful work*. Hence, as in [46], we differentiate between two different notions of load balancing—quantitative load balancing and qualitative load balancing. Quantitative load balancing consists of ensuring that the amount of work allocated to each processor is approximately equal. Qualitative load balancing, on the other hand, consists of ensuring not only that each processor has enough work to do, but also that each processor has *high-quality* work to do.

The use of a single central tree manager has the advantage of making load balancing easy. Whenever a processor runs out of work, the tree manager can simply issue more. Furthermore, it can easily issue the "best" work that is available at that time, usually the subproblem with the least lower bound. Unfortunately, the central tree manager becomes a computational bottleneck when large numbers of slave processes are employed. The degree to which this occurs is highly dependent on the problem setting. If each search tree node requires significant processing time (and hence the tree is not growing too quickly), then scalability may not be much of an issue. For problems in which quick enumeration of a large search tree is the primary computational approach, scalability will suffer.

This problem has been studied extensively for general branch and bound and various approaches to "decentralization" have been suggested to relieve the bottleneck at the tree manager. However, while these approaches are more scalable, they appear to be inefficient when the numbers of processors is small, at least for our purposes. Moreover, they do not allow the use of our differencing scheme for storing the entire tree efficiently at a single processor. The straightforward implementation of such a scheme may, therefore, sacrifice our ability to solve large problems sequentially. Furthermore, fault tolerance could also be decreased (see Sect. 7.2). It's in view of these considerations that we employ the master-slave model. See Sect. 10.1 for a discussion of future improvements to scalability.

7.2   Details of the Parallel Implementation

Parallel Configurations. SYMPHONY supports numerous configurations, ranging from completely sequential to fully parallel, allowing efficient execution in many different computational settings. As described in the previous section, there are five modules in the standard distributed configuration. Various subsets of these modules can be combined to form separate executables capable of communicating with each other across a network. When two or more modules are combined, they simply communicate through shared-memory instead of through message-passing. However, they are also forced to run in sequential fashion in this case, unless the user chooses to enable threading using an OpenMP compliant compiler (see next section).

As an example, the default distributed configuration includes a separate executable for each module type, allowing full parallelism. However, if cut generation is fast and not memory-intensive, it may not be worthwhile to have the LP solver and its associated cut generator work independently, as this increases communication overhead without much potential benefit. In this case, the cut generator functions can be called directly from the LP solver, creating a single, more efficient executable.

Inter-process Communication. SYMPHONY can utilize any third-party communication protocol supporting basic message-passing functions. All communication subroutines interface with SYMPHONY through a separate communications API. Currently, PVM is the only message-passing protocol supported, but interfacing with another protocol is a straightforward exercise.

Additionally, it is possible to configure the code to run in parallel using threading to process multiple search tree nodes simultaneously. Currently, this is implemented using OpenMP compiler directives to specify the parallel regions of the code and perform memory locking functions. Compiling the code with an OpenMP compliant compiler will result in a shared-memory parallel executable. For a list of OpenMP compliant compilers and other resources, visit `http://www.openmp.org`.

Fault Tolerance. Fault tolerance is an important consideration for solving large problems on computing networks whose nodes may fail unpredictably. The tree manager tracks the status of all processes and can restart them as necessary. Since the state of the entire tree is known at all times, the most that will be lost if an LP process or cut generator process is killed is the work that had been completed on that particular search node. To protect against the tree manager itself or a cut pool being killed, full logging capabilities have been implemented. If desired, the tree manager can write out the entire state of the tree to disk periodically, allowing a restart if a fault occurs. Similarly, the cut pool process can be restarted from a log file. This not only allows for fault tolerance but also for full reconfiguration in the middle of solving a long-running problem. Such reconfiguration could consist of anything from adding more processors to moving the entire solution process to another network.

8 Applications

To make the ideas discussed thus far more concrete, we now introduce two practical applications from combinatorial optimization with which many readers will already be familiar. Graph-based problems, especially those involving *packing* and *routing* constraints, lend themselves particularly well to implementation in this type of framework. This is because many of the constraints, such as those dealing with connectivity of the solution, can be represented compactly using bit vectors, as described previously. Also, the one-to-one correspondence between variables and edges in the underlying graph yields a simple variable indexing scheme based on a lexicographic ordering of the edges. We therefore begin by describing the use of SYMPHONY to implement a basic solver for the Vehicle Routing Problem [75, 74], and then move on to describe a Set Partitioning solver [36]. Summary computational results will be given later in Sect. 9.

8.1 The Vehicle Routing Problem

The *Vehicle Routing Problem* (VRP) was introduced by Dantzig and Ramser [32] in 1959. In this graph-based problem, a central depot $\{0\}$ uses k independent delivery vehicles, each of identical capacity C, to service integral demands d_i for a single commodity from customers $i \in N = \{1, \cdots, n\}$. Delivery is to be accomplished at minimum total cost, with c_{ij} denoting the transit cost from i to j, for $0 \leq i, j \leq n$. The cost structure is assumed *symmetric*, i.e., $c_{ij} = c_{ji}$ and $c_{ii} = 0$.

A solution for this problem consists of a partition $\{R_1, \ldots, R_k\}$ of N into k *routes*, each satisfying $\sum_{j \in R_i} d_j \leq C$, and a corresponding permutation, or *tour*, σ_i, of each route specifying the service ordering. This problem is naturally associated with the complete undirected graph consisting of nodes $N \cup \{0\}$, edges E, and edge-traversal costs $c_{ij}, \{i, j\} \in E$. A solution is a cloverleaf pattern whose k petals correspond to the routes serviced by the k vehicles. An integer programming formulation can be given as follows:

$$min \quad \sum_{e \in E} c_e x_e$$

$$\sum_{e=\{0,j\} \in E} x_e = 2k \tag{2}$$

$$\sum_{e=\{i,j\} \in E} x_e = 2 \quad \forall i \in N \tag{3}$$

$$\sum_{\substack{e=\{i,j\} \in E \\ i \in S, j \notin S}} x_e \geq 2b(S) \quad \forall S \subset N, \ |S| > 1 \tag{4}$$

$$0 \leq x_e \leq 1 \quad \forall e = \{i, j\} \in E, \ i, j \neq 0 \tag{5}$$

$$0 \leq x_e \leq 2 \quad \forall e = \{0, j\} \in E \tag{6}$$

$$x_e \quad integral \quad \forall e \in E. \tag{7}$$

Here, $b(S)$ can be any lower bound on the number of trucks needed to service the customers in set S, but for ease of computation, we define $b(S) = \lceil (\sum_{i \in S} d_i)/C \rceil$. The constraint (2) stipulates that there should be exactly k routes, while the constraints (3) require that each customer be visited by exactly one vehicle. The constraints (4) ensure connectivity of the solution while also implicitly ensuring that no route services total demand in excess of the capacity C.

Solver Implementation. Implementing a BCP algorithm based on the above formulation is straightforward using the framework. As discussed in Sect. 5.2, our main concern is with the treatment of constraints and variables. The number of variables is small enough for practical instances of this problem that we don't need to concern ourselves with column generation. We tried using column generation but did not find it to be advantageous. Our indexing scheme for the variables is based on a lexicographic ordering of the edges in the complete graph. This enables an easily calculable one-to-one mapping of edges to indices. To construct the core of the problem, we select the variables corresponding to the k cheapest edges adjacent to each node in the graph, as these are the variables most likely to have a positive value in some optimal solution. The remainder of the variables are added as *extra* variables in the root node. The hope is that most of these will be priced out of the problem quickly.

Constraints present different challenges, however. The number of constraints in the LP relaxation is exponential. Furthermore, the separation problem for the constraints (4) is known to be $\mathcal{NP}$-complete [9]. We therefore rely on heuristic separation routines to generate the constraints (4) dynamically during the search process. Because the number of degree constraints (2) and (3) is small and we want them to appear in all relaxations (our heuristics depend on this fact), they are placed in the core. Initially, these are the only active constraints in the root node. In the basic solver presented here, dynamic constraint generation takes place only for the capacity constraints (4). Since the sets of variables with nonzero indices in these constraints correspond to the sets of edges across cuts in the graph, these constraints can be represented as a bit array indicating the nodes included in one shore of the cut (the one not containing the depot). To construct the row corresponding to a particular constraint, it suffices to check the edge corresponding to each active variable in the current relaxation and determine if its endpoints are on opposite shores of the cut. If so, the variable has a coefficient of one in the row. Otherwise, its coefficient is zero.

Besides cut generation, only a few other problem-specific routines are needed to implement the basic solver. Strong branching candidates are selected using a built-in default rule—select those variables nearest to .5 in value. Some logical fixing can also be done. For instance, if two edges adjacent to a particular node are already fixed to one, then all other edges adjacent to that node can be fixed to zero.

8.2 Set Partitioning Problem

In [36], Eső used an early version of SYMPHONY to implement a solver for the Set Partitioning Problem (SPP). Here, we review her work. Combinatorially, the Set Partitioning Problem can be stated as follows. We are given a ground set S of m objects and a collection C of subsets $S_1, \cdots, S_n$ of S, each with a given cost $c_j = c(S_j)$. We wish to select the minimum weight subfamily of C that forms a partition of S. This problem is well-studied and describes many important applications, including airline crew scheduling, vehicle routing, and political districting (see [41, 11, 51, 19]).

To describe an integer programming formulation of the SPP, we construct matrix A, whose rows correspond to the members of S and whose columns correspond to the members of C. Entry a_{ij} is 1 if the ith element of S is included in subset S_j; otherwise, we set a_{ij} to zero. Then the problem can simply be stated as

$$min \sum_{j=1}^{n} c_j x_j \tag{8}$$

$$s.t. \sum_{j=1}^{n} a_{ij} x_j = 1 , \quad 1 \le i \le m \tag{9}$$

$$x_j \in \{0, 1\} , \quad 1 \le j \le n. \tag{10}$$

Each row of this formulation expresses the constraint that there must be exactly one member of the partition containing each element of the set S.

Solver Implementation. In crew scheduling applications, the matrix A can be extremely large and generating it can be difficult. Furthermore, solving the LP relaxation itself can also be very difficult. For the work in [36], the matrix A was assumed given. Even so, preprocessing A, in order to reduce its size without eliminating any optimal solutions, was found to be vitally important. Since finding feasible solutions to this problem within the search tree can prove quite difficult, heuristic solution procedures were used to find good upper bounds. Both the preprocessing of the matrix and the execution of heuristic procedures were performed before the branch and cut procedure was invoked. It is important to note that performing intensive computation prior to beginning the branch and cut procedure can potentially decrease parallel speedup by increasing the computation's sequential fraction (see 7.1).

Unlike many combinatorial problems, for crew scheduling models, it was difficult to judge a priori the relative importance of each column and its likelihood of participating in some optimal solution. This is because the magnitude of the objective function coefficient corresponding to a particular column is not necessarily a good indicator of its usefulness. Large objective function coefficients may simply correspond to columns representing large subsets of the set S. Because of this, the set of core variables was taken to be empty in order to allow removal by reduced cost and logical implications in the lower levels of the search tree.

On the other hand, all constraints remaining after preprocessing were taken to be core constraints.

Part of the difficulty inherent in crew scheduling models stems from the extensive computation often required for solving the LP relaxations. In particular, the simplex algorithm sometimes has trouble optimizing these linear programs. Therefore, the barrier method with dual crossover was used to solve the initial LP relaxation and derive a feasible basis for the dual simplex algorithm, which was then used for the remaining calculations. The same problem reduction procedures that were so important during preprocessing were also employed throughout the tree to further reduce the matrix after branching or otherwise fixing variables. In addition, a primal heuristic was employed to derive new feasible solutions and hence improve the current upper bound. Candidates for branching were taken from among the variables, existing cuts that had become slack, and cuts produced specifically for branching. The algorithm employed detection of tailing off and forced branching whenever such a condition was detected.

There are many known classes of cutting planes that can be used to strengthen the LP relaxations for this problem. Examples are clique inequalities, odd holes, packing and cover odd holes, odd antiholes, and other lifted versions of these classes. Because it was not clear which of these classes would produce the strongest cuts for a particular instance, the cut generator was itself parallelized in order to find cuts in several classes simultaneously.

9 Computational Experience

In this section, we describe our experience using the framework to solve the two classical combinatorial optimization problems we have already described (VRP and SPP). Although we report some running times in Sect. 9.2 below, what follows should be considered anecdotal evidence based on our observations and experience with the development of SYMPHONY.

9.1 Sequential Performance

Performance of our code when running sequentially has improved dramatically since the first published results in [75]. Although the same fundamental design has been maintained since the inception of the project, the implementation has been streamlined and improved to the point that the running time for a standard set of Vehicle Routing Problems, *even after adjusting for increased processing speed*, has improved by more than two orders of magnitude since 1995. This underscores the fact that, first and foremost, *implementational details can produce a marked difference in efficiency for BCP algorithms*. In the following sections, we summarize a few of the "details" that have proven to be important. However, we emphasize that the most effective way to learn about the implementation of these algorithms is to examine the documentation and source code itself [76].

It is well-known that the vast majority of the computing time for BCP is spent in two activities—solving LP relaxations and generating new cuts and variables.

In SYMPHONY, both of these activities are performed by external libraries. The result is that, in practice, very little time is spent executing instructions that actually reside within the framework itself. Hence, improvements to running times must come not through reducing the time spent within the framework, but rather through *reducing the time spent in code outside the framework*. Although we have no control over the efficiency of these external codes, we *can* control not only the input to these external subroutines, but also the number of times they need to be called. To achieve real improvements in efficiency, one must guide the solution process with this in mind.

Linear Program Management. To reduce time spent solving linear programs, we emphasize once again that the most important concept is the ability to limit the size of the LP relaxations by *allowing cuts and variables to be fluidly generated, activated, and deactivated* at various points during the (iterative) solution process. This entire process must be managed in such a way as to reduce the size of the matrix while not reducing the quality of the solutions produced. *It is also critical to maintain LP warm-start information (i.e., a description of the current basis) throughout the tree to allow efficient processing of each search tree node.*

Constraints. The most effective approach to managing the constraints in the LP relaxation has been to *be conservative with adding constraints to the relaxation while being liberal with removing them.* We have found that by deleting ineffective constraints quickly, we can significantly reduce LP solution time. Of course, it would be better not to add these ineffective constraints in the first place. The local cut pools, which allow only the "best" constraints to be added in each iteration, have been instrumental in reducing the number of cuts that eventually do become ineffective. This combination approach has worked extremely well.

With respect to reducing the time spent *generating* constraints, *the global cut pool is effective for constraints whose generation is relatively expensive.* For constraints that are inexpensive to generate in comparison to the time spent solving LPs, the cost of maintaining the cut pool does not always pay off. Our approach to management of the cut pool has been similar to that of managing the constraints of the linear programs, but here it is less clear how to effectively keep its size under control. Our conservative policy with respect to adding constraints to the pool has produced good results. However, the question of how to determine which constraints should be removed from the cut pool needs further consideration.

Variables. Although cuts and variables can be handled symmetrically in many respects, there are some major 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 is invoked*, as it destroys monotonicity of the objective function, upon

which algorithmic performance sometimes depends. Furthermore, 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 eliminate the conditions necessary to fathom the node in question, i.e., by either lowering the objective function value back below the current upper bound or by restoring feasibility. Thus, the user must be mindful of whether the node is about to be fathomed before performing column generation.

In many problem settings, particularly those involving combinatorial optimization, it is much easier to judge *a priori* the importance of a particular variable (based on the problem structure and the structure of the objective function) than it is to judge the importance of a constraint. *It is important to take advantage of this information.* We have mentioned two different ways in which we can do this. By declaring some "unimportant" variables inactive in the root node, the user can delay including them in any LP relaxation until the column generation step. In the two-phase method with repricing in the root node (see Sect. 6.3), it is possible, even probable, that these variables would simply be priced out immediately in the second phase. In theory, this should allow much faster processing of subproblems and less time spent solving LPs.

In our experience with combinatorial problem solving, however, generating columns, unless done very efficiently, can be an expensive operation whose cost is not usually justified. This is especially true in the presence of a good initial upper bound. In this case, most "unimportant" columns end up being priced out either in the root node or relatively soon thereafter. *Therefore, if efficient explicit generation of all variables is possible in the root node and there is sufficient memory to store them, this is generally the best option.* This allows variables to be fixed by reduced cost and nodes to be fathomed without expensive pricing. However, if either (1) there is not enough memory to store all of the problem's columns 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, then column generation might be an effective technique.

Search Tree Management. The primary way in which solution time can be reduced is by *reducing the size of the search tree.* Effective branching, on both variables and constraints, is an important tool for this reduction. *One of the most effective methods available for branching is* strong branching, discussed in Sect. 6.2. In a recent test on a set of Vehicle Routing Problems, it was found that the number of search nodes explored was reduced (in comparison with standard branching) by over 90% using strong branching with just seven candidate branching objects. For this reason, use of strong branching is highly recommended. Nonetheless, this branching should be delayed as long as possible. As long as violated cuts can be found and the relaxation solution value has not tailed off, processing of the current search tree node should continue.

Another way to affect the size of the search tree is through an effective search strategy, as we discussed in 6.3. We have found that *a hybrid of "best-first," along with controlled diving produces the best results.* Diving leads to improved upper

bounds and reductions in node set-up costs while minimizing communication with the tree manager. This can be significant when employing parallelism.

9.2 Parallel Performance

Given the modular design of SYMPHONY, the transition from sequential to parallel processing is straightforward. However, the centralized management of the search process and centralized storage of the search tree, while highly effective for sequential processing, does not lead to a scalable parallel algorithm. That said, *the parallel efficiency of our approach is very high for small numbers of processors.* For many practical settings, this is all that is needed. Furthermore, as discussed earlier, this parallelism is achieved without sacrificing any of the efficiencies to be gained in the more typical sequential setting.

Parallel efficiency in BCP is achieved mainly through effective qualitative and quantitative load balancing. For small numbers of processors, our approach handles load balancing with ease. The search trees produced in parallel runs are approximately the same size as those produced sequentially, leading to *linear speedup.* See the sample computational results in the next section for an example of this.

As the number of processors increases, the tree manager eventually becomes a computational bottleneck. As indicated in Sect. 7.1, the point at which this happens is highly problem-dependent. When this point is reached, parallel efficiency can be increased by limiting communication between the tree manager and the LP solver as much as possible. For instance, increased diving can significantly enhance large-scale parallel efficiency. For more ideas on improving the scalability of these algorithms, see the discussion in Sect. 10.1.

9.3 Computational Results for Vehicle Routing

Our experience with the VRP has reinforced many of the lessons already discussed. We experimented with column generation, but found it to be more effective to simply include all variables in the root node. However, placing only the k shortest edges adjacent to each customer node in the problem core did lead to significant gains. We found strong branching to be an extremely effective tool for reducing the size of the search tree. However, we experienced diminishing returns when examining more than 7-10 candidates. We used a hybrid diving strategy that allowed us to find feasible solutions early in the search process. In many cases, this led to running times almost equaling those achieved by computing a heuristic upper bound prior to beginning the branch and cut phase of the algorithm. Although effective constraint generation is absolutely critical for this problem, use of the cut pool did not have a dramatic effect. This is probably because the cuts we applied are relatively easy to generate.

Table 1 presents summary computational results for recent testing of the basic VRP solver discussed earlier. These results are not intended to be comprehensive, but are presented for illustrative purposes. For more detailed discussion

of using SYMPHONY to solve the VRP and more in-depth computational results, see [74]. Tests here were performed on a network of 3 workstations powered by 4 DEC Alpha processors each using CPLEX as the LP engine. The problems are easy- to medium-difficulty problems from VRPLIB [81] and other sources [79].

¿From Table 1, it is evident that the number of nodes in the search tree is largely independent of the number of LP processes being utilized. This essentially ensures linear speedup as long as parallel overhead remains low and there are no computational bottlenecks. Predictably, as the number of processors increases, the idle time also increases, indicating that the tree manager is becoming saturated with requests for data. Because not many strong cuts are known for this model, we tend to rely on quick enumeration to solve these problems. This is possible because the LP relaxations are relatively easy to solve. It is therefore common to develop large search trees in a short period of time. Our compact storage scheme allows us to deal with these large search trees. However, scalability suffers in this situation.

Table 1. Summary Computational Results for VRP instances

	Number of LP processes used			
	1	2	4	8
Number of search tree nodes	6593	6691	6504	6423
Wallclock solution time (sec)	2493	1281	666	404
Wallclock solution time per node	0.38	0.38	0.41	0.50
Idle time per node	0.00	0.01	0.03	0.08

9.4 Computational Results for Set Partitioning

In contrast to the Vehicle Routing Problem, the Set Partitioning Problem produces relatively small search trees but search nodes can be much more difficult to process. In particular, the simplex algorithm had difficulty in solving some of the LP relaxations encountered in this problem. However, as mentioned earlier, Eső does report success with using the barrier method with dual crossover to solve the initial LP relaxations. Problem size reduction techniques help to control the size of the LP relaxations, leading to reduced node processing times.

In this problem, tailing off can be a problem, so branching was invoked whenever the lower bound did not show significant improvement over a sequence of iterations. As in the VRP, strong branching was found to be an important tool in reducing the size of the search tree. However, for the SPP, choosing specially constructed cuts, in addition to variables, as branching candidates was found to be important. Dynamic cut generation was also critical to efficiency. Extensive computational results on a variety of problems (assembled from the crew scheduling literature) are reported in [36]. The implementation solved many of these problems to optimality in the root node of the enumeration tree, so there

was no need in such instances for parallel processing. Of the remaining problems, some proved too difficult to solve, mainly due to difficulties in solving LP relaxations, as indicated above. Several of the more difficult models did yield solutions, however, with significant multi-processing effort. The principles observed here were similar to those for the VRP: The number of search tree nodes was essentially independent of the number of LP processes, resulting in linear speed-up for as many as 8 LP processes. For example, with model *aa04*, a well-known difficult problem taken from [50], the following results shown in Table 2 were obtained (see also Table 5.12 of [36]). The computing platform is an IBM SP2 with LP solver CPLEX.

Table 2. Sample computational results for the crew scheduling model *aa04*

	Number of LP processes used			
	1	2	4	8
Number of search tree nodes	283	268	188	234
Depth of search tree	25	22	16	17
Wallclock solution time (sec)	2405	1111	350	240

10 Future Development

Although the theory underlying BCP algorithms is well-developed, our knowledge of how to implement these algorithms continues to improve and grow. To some extent, effective implementation of these algorithms will continue to depend on problem-specific techniques, especially cut generation. However, we have already learned a great deal about how to remove certain burdens from the user by implementing generic defaults that work well across a wide variety of problems. In this section, we offer a few ideas about where future growth will occur.

10.1 Improving Parallel Scalability

With the state of technology driving an increasing interest in parallel computation, it is likely that parallel algorithms will continue to play an important role in the field of optimization. In these notes, we have touched on some of the central issues surrounding the parallelization of BCP algorithms, but much remains to be learned. In particular, more scalable approaches to BCP need to be developed. As we have already pointed out, this clearly involves some degree of decentralization. However, the schemes that have appeared in the literature (mostly applied to parallel branch and bound) appear inadequate for the more complex challenges of BCP.

The most straightforward approach to improving scalability is simply to increase the task granularity and thereby reduce the number of decisions the tree manager has to make, as well as the amount of data it has to send and receive.

To achieve this, we could simply allow each LP process to examine an entire subtree or portion of a subtree before checking back for additional work. This approach would be relatively easy to implement, but has some potentially serious drawbacks. The most serious of these is that the subtree being examined could easily turn out to contain mostly unpromising nodes that would not have been examined otherwise. Hence, this scheme seems unlikely to produce positive results in its most naive form.

Another approach is to attempt to relieve the bottleneck at the central tree manager by only storing the information required to make good load-balancing decisions (most notably, the lower bound in each search tree node) centrally. The data necessary to generate each search node could be stored either at one of a set of "subsidiary tree managers" or within the LP modules themselves. This is similar to a scheme implemented by Eckstein [33] for parallel branch and bound. Such a scheme would maintain the advantages of global decision-making while moving some of the computational burden from the tree manager to other processes. However, effective load balancing would necessarily involve an expensive increase in the amount of data being shuttled between processes. Furthermore, the differencing scheme we use for storing the search tree will not extend easily to a decentralized environment.

10.2 Other Directions

The vast majority of research on BCP has concentrated on the now well-studied technique of branch and cut. Branch and price, on the other hand, has received relatively little attention and the integration of these two methods even less. In particular, issues related to if, when, and how to effectively generate new variables, independent of the problem setting, need further investigation. Effective management of pools for both cuts and variables is another important computational issue which deserves attention. As we pointed out several times, SYMPHONY currently has a bias towards the implementation of branch and cut algorithms. We intend to improve and generalize our implementation of variable generation in order to make the framework more flexible and efficient for branch and price.

Outside of branching on fractional variables, few generic branching rules have been developed. Most BCP implementations still rely on variable branching because it is easy to implement and relatively effective. However, there are situations in which it can be ineffective when compared to branching on a well-selected cut or on a set of objects. Automatic methods of determining which cuts will make effective branching objects have yet to be examined.

Until recently, almost all BCP algorithms have utilized simplex-based LP solvers to perform lower bounding. Currently, these solvers still offer the best performance across a wide range of problems. However, new solution techniques, such as the volume algorithm (see [14]) are showing promise in helping to solve those problems on which the simplex algorithm falters. As discussed in 8.2, we have already seen in [36] that the barrier method successfully provided an alternative to the simplex method in solving large-scale LP problems arising in

airline crew scheduling models. Relaxation schemes that use techniques other than linear programming, e.g., semi-definite programming, are also coming into prominence. Interfacing with these new solvers should provide fruitful avenues for further improvement in BCP methodology.

11   Conclusion

In these notes, we have given the reader a summary of many important challenges of implementing branch, cut, and price algorithms. However, there are many more details to be explored below the surface. We encourage the interested reader to visit `http://BranchAndCut.org` for more extensive documentation and source code for SYMPHONY.

References

1. Amdahl, G.M.: Validity of the Single-processor Approach to Achieving Large-scale Computing Capabilities. In AFIPS Conference Proceedings **30** (Atlantic City, N.J., April 18-20), AFIPS Press (1967), 483
2. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: On the solution of traveling salesman problems, Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians (1998), 645
3. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: CONCORDE TSP Solver. Available at `www.keck.caam.rice.edu/concorde.html`
4. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: Finding Cuts in the TSP. DIMACS Technical Report 95-05, 1995
5. Agarwal, Y., Mathur, K., and Salkin, H.M.: Set Partitioning Approach to Vehicle Routing. Networks **7**, 731, 1989
6. Araque, J.R., Kudva, G., Morin, T.L., and Pekny, J.F.: A Branch-and-Cut Algorithm for Vehicle Routing Problems. Annals of Operations Research **50**, 37, 1994
7. Araque, J.R., Hall, L., and Magnanti, T.: Capacitated Trees, Capacitated Routing and Associated Polyhedra. Discussion paper 9061, CORE, Louvain La Nueve, 1990
8. Augerat, P., Belenguer, J.M., Benavent, E., Corberán, A., and Naddef, D.: Separating Capacity Constraints in the CVRP Using Tabu Search. European Journal of Operations Research **106**, 546, 1998
9. Augerat, P., Belenguer, J.M., Benavent, E., Corberán, A., Naddef, D., and Rinaldi, G.: Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem. Research Report 949-M, Université Joseph Fourier, Grenoble, France, 1995
10. Balas, E., Ceria, S., and Cornuéjols, G.: Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework. Management Science **42**, 9, 1996
11. Balas, E., and Padberg, M.W.: Set Partitioning: A Survey. SIAM Review **18**, 710, 1976
12. Balas, E., and Toth, P.: Branch and Bound Methods. In Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G, and Shmoys, D.B., eds., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, New York, 361, 1985.

13. Balinski, M.L., and Quandt, R.E.: On an Integer Program for a Delivery Problem. Operations Research **12**, 300, 1964
14. Barahona, F., and Anbil, R.: The Volume Algorithm: Producing Primal Solutions with a Subgradient Method. Mathematical Programming **87**, 385, 2000
15. Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P., and Vance, P.H.: Branch-and-Price: Column Generation for Huge Integer Programs. Operations Research **46**, 316, 1998
16. Benchouche, M., Cung, V.-D., Dowaji, S., Le Cun, B., Mautor, T., and Roucairol, C.: Building a Parallel Branch and Bound Library. In Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science **1054**, Springer, Berlin, 201, 1996
17. Blasum, U., and Hochstättler, W.: Application of the Branch and Cut Method to the Vehicle Routing Problem. Zentrum für Angewandte Informatik Köln Technical Report zpr2000-386, 2000
18. Boehning, R.L., Butler, R.M., and Gillet, B.E.: A Parallel Integer Linear Programming Algorithm. European Journal of Operations Research **34**, 393, 1988
19. Borndörfer, R.: Aspects of Set Packing, Patitioning, and Covering. PhD. Dissertation, Technischen Universität Berlin, 1997
20. Campos, V., Corberán, A., and Mota, E.: Polyhedral Results for a Vehicle Routing Problem. European Journal of Operations Research **52**, 75, 1991
21. Chen, Q., and Ferris, M.C.: FATCOP: A Fault Tolerant Condor-PVM Mixed Integer Programming Solver. University of Wisconsin CS Department Technical Report 99-05, Madison, WI, 1999
22. Christofides, N., and Eilon, S.: An Algorithm for the Vehicle Dispatching Problem, Operational Research Quarterly **20**, 309, 1969
23. Christofides, N., Mingozzi, A., and Toth, P.: Exact Algorithms for Solving the Vehicle Routing Problem Based on Spanning Trees and Shortest Path Relaxations. Mathematical Programming **20**, 255, 1981
24. Chvátal, V.: Linear Programming. W.H. Freeman and Company, San Francisco, 1983
25. Cornuéjols, G., and Harche, F., Polyhedral Study of the Capacitated Vehicle Routing Problem. Mathematical Programming **60**, 21, 1993
26. Cordier, C., Marchand, H., Laundy, R., and Wolsey, L.A.: bc-opt: A Branch-and-Cut Code for Mixed Integer Programs. Mathematical Programming **86**, 335, 1999
27. Common Optimization INterface for Operations Research. http://www.coin-or.org
28. Crowder, H., and Padberg, M.: Solving Large Scale Symmetric Traveling Salesman Problems to Optimality. Management Science **26**, 495, 1980
29. Crowder, H., Johnson, E.L., and Padberg, M.: Solving Large-Scale Zero-One Linear Programming Problems. Operations Research **31** 803, 1983
30. Cullen, F.H., Jarvis, J.J., and Ratliff, H.D.: Set Partitioning Based Heuristic for Interactive Routing. Networks **11**, 125, 1981
31. Cung, V.-D., Dowaji, S., Le Cun, B., Mauthor, T., and Roucairol, C.: Concurrent Data Structures and Load Balancing Strategies for Parallel Branch and Bound/A* Algorithms. DIMACS Series in Discrete Optimization and Theoretical Computer Science **30**, 141, 1997
32. Dantzig, G.B, and Ramser, J.H., The Truck Dispatching Problem. Management Science **6**, 80, 1959
33. Eckstein, J.: Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM-5. SIAM Journal on Optimization **4**, 794, 1994

34. Eckstein, J.: How Much Communication Does Parallel Branch and Bound Need? INFORMS Journal on Computing **9**, 15, 1997

35. Eckstein, J., Phillips, C.A., and Hart, W.E.: PICO: An Object-Oriented Framework for Parallel Branch and Bound. RUTCOR Research Report 40-2000, Rutgers University, Piscataway, NJ, 2000

36. Eső, M.: Parallel Branch and Cut for Set Partitioning. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1999

37. Fisher, M.L.: Optimal Solution of Vehicle Routine Problems Using Minimum k-Trees. Operations Research **42**, 141, 1988

38. Fisher, M.L., and Jaikumar, R.: A Generalized Assignment Heuristic for Solving the VRP. Networks **11**, 109, 1981

39. Foster, B.A., and Ryan, D.M.: An Integer Programming Approach to the Vehicle Scheduling Problem, Operational Research Quarterly **27**, 367, 1976

40. Garey, M.R., and Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, 1979

41. Garfinkel, R.S., and Nemhauser, G.L.: Integer Programming. Wiley, New York, 1972

42. Geist, A., et al.: PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994

43. Gendron, B., and Crainic, T.G.: Parallel Branch and Bound Algorithms: Survey and Synthesis. Operations Research **42**, 1042, 1994

44. Golub, G.H., and Van Loan, C.F.: Matrix Computations. The Johns Hopkins University Press, Baltimore, 1989

45. Grötschel, M., Jünger, M., and Reinelt, G.: A Cutting Plane Algorithm for the Linear Ordering Problem. Operations Research **32**, 1155, 1984

46. Grama, A., and Kumar, V.: Parallel Search Algorithms for Discrete Optimization Problems. ORSA Journal on Computing **7**, 365, 1995

47. Gustafson, J.L.: Re-evaluating Amdahl's Law. Communications of the ACM **31**, 532, 1988

48. Held, M., and Karp, R.M.: The Traveling Salesman Problem and Minimal Spanning Trees. Operations Research **18**, 1138, 1969

49. ILOG CPLEX 6.5 Reference Manual, ILOG, 1994

50. Hoffman, K., and Padberg, M.: LP-Based Combinatorial Problem Solving. Annals of Operations Research **4**, 145, 1985/6

51. Hoffman, K., and Padberg, M.: Solving Airline Crew Scheduling Problems by Branch-and-cut. Management Science **39**, 657, 1993

52. Jünger, M., and Thienel, S.: The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization. Software Practice and Experience **30**, 1325, 2000

53. Jünger, M., and Thienel, S.: Introduction to ABACUS—a branch-and-cut system. Operations Research Letters **22**, 83, 1998

54. Jünger, M., Reinelt, G., and Thienel, S.: Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 111, 1995

55. Kozen, D.C.: The Design and Analysis of Algorithms, Springer-Verlag, New York, 1992

56. Kopman, L.: A New Generic Separation Algorithm and Its Application to the Vehicle Routing Problem. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1999

57. Kumar, V., and Rao, V.N.: Parallel Depth-first Search. Part II. Analysis. International Journal of Parallel Programming **16**, 501, 1987

58. Kumar, V., and Gupta, A.: Analyzing Scalability of Parallel Algorithms and Architectures. Journal of Parallel and Distributed Computing **22**, 379, 1994

59. Ladányi, L.: Parallel Branch and Cut and Its Application to the Traveling Salesman Problem. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1996

60. Laporte, G., and Nobert, Y.: Comb Inequalities for the Vehicle Routing Problem. Methods of Operations Research **51**, 271, 1981

61. Laporte, G., Nobert, Y., and Desrouchers, M.: Optimal Routing with Capacity and Distance Restrictions. Operations Research **33**, 1050, 1985

62. Laursen, P.: Can Parallel Branch and Bound without Communication Be Effective? SIAM Journal of Optimization **4**, 288, 1994

63. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, New York, 1985

64. Letchford, A.N., Eglese, R.W., and Lysgaard, J.: Multi-star Inequalities for the Vehicle Routing Problem. Technical Report available at `http://www.lancs.ac.uk/staff/letchfoa/pubs.htm`

65. Linderoth, J.: Topics in Parallel Integer Optimization. Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 1998

66. Marsten, R.: The Design of The XMP Linear Programming Library. ACM Transactions on Mathematical Software **7**, 481, 1981

67. Martin, A.: Integer Programs with Block Structure. Habilitation Thesis, Technischen Universität Berlin, 1998

68. Naddef, D., and Rinaldi, G.: Branch and Cut. In Toth, P., and Vigo, D., eds., Vehicle Routing, SIAM, 2000.

69. Nagamochi, H., and Ibaraki, T.: Computing Edge Connectivity in Multigraphs and Capacitated Graphs. SIAM Journal of Discrete Mathematics **5**, 54, 1992

70. Nemhauser, G.L., Savelsbergh, M.W.P., and Sigismondi, G.S.: MINTO, a Mixed INTeger Optimizer. Operations Research Letters **15**, 47, 1994

71. Nemhauser, G.L., and Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley, New York, 1988

72. Padberg, M., and Rinaldi, G.: A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems. SIAM Review **33**, 60, 1991

73. Radaramanan, R., and Choi, K.: A Branch and Bound Algorithm for Solving the Vehicle Routing Problem. Proceedings of the 8th Annual Conference on Computers and Industrial Engineering, 236.

74. Ralphs, T.K., Kopman, L., Pulleyblank, W.R., and Trotter Jr., L.E.: On the Capacitated Vehicle Routing Problem. Submitted for publication

75. Ralphs, T.K.: Parallel Branch and Cut for Vehicle Routing. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1995

76. Ralphs, T.K., SYMPHONY Version 2.8 User's Guide. Available at `www.branchandcut.org/SYMPHONY`

77. Ralphs, T.K., and Ladányi, L. Computational Experience with Branch, Cut, and Price. To be submitted.

78. Ralphs, T.K., and Ladányi, L.: SYMPHONY: A Parallel Framework for Branch and Cut. White paper, Rice University, 1999.

79. Ralphs, T.K.: Repository of Vehicle Routing Problem Instances. Available at `http://www.BranchAndCut.org/VRP`

80. Rao, V.N., and Kumar, V.: Parallel Depth-first Search. Part I. Implementation. International Journal of Parallel Programming **16**, 479, 1987

81. Reinelt, G.: TSPLIB—A traveling salesman problem library. ORSA Journal on Computing **3**, 376, 1991. Update available at `http://www.crpc.rice.edu/softlib/tsplib/`

82. Rushmeier, R., and Nemhauser, G.L.: Experiments with Parallel Branch and Bound Algorithms for the Set Covering Problem. Operations Research Letters **13**, 277, 1993

83. Savelsbergh, M.W.P.: A Branch-and-Price Algorithm for the Generalized Assignment Problem. Operations Research **45**, 831, 1997

84. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York, 1986

85. Shinano, Y., Higaki, M., and Hirabayashi, R.: Generalized Utility for Parallel Branch and Bound Algorithms. Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 392, 1995

86. Shinano, Y., Harada, K., and Hirabayashi, R.: Control Schemes in a Generalized Utility for Parallel Branch and Bound. Proceedings of the 1997 Eleventh International Parallel Processing Symposium, IEEE Computer Society Press, Los Alamitos, CA, 621, 1997

87. Tschöke, S., and Polzer, T.: Portable Parallel Branch and Bound Library User Manual, Library Version 2.0. Department of Computer Science, University of Paderborn