SYMPHONY **2.8 User's Manual** [*]

SYMPHONY Developed By

T.K. Ralphs[†]
L. Ladányi[‡]

Interactive Graph Drawing
Software By

M. Esö[§]

December 2, 2001

[*]This research was partially supported by NSF Grant DMS-9527124 and Texas ATP Grant 97-3604-010

[†]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, `tkralphs@lehigh.edu`, `http://www.lehigh.edu/~tkr2`

[‡]Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

[§]Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

[1]A revised version of Sections 1-7 of this manual now appears in the Springer-Verlag book *Computational Combinatorial Optimization* edited by M. Jünger and D. Naddef, see http://link.springer.de/link/service/series/0558/tocs/t2241.htm

Contents

1 A Brief History

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 [15, 27, 17]) which brings the computational tools of branch and bound algorithms together with the theoretical tools of polyhedral combinatorics. Indeed, 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 [1] 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 cutting planes) 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) [18], while the other, headed by the authors, 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). A version of SYMPHONY written in C++, which we call COIN/BCP has also been produced at IBM under the COIN-OR project [8]. 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.

2 Related Work

The 1990's witnessed a broad development of software for discrete optimization. Almost without exception, these new software packages were 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) 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 dozens of offerings in this category are MINTO [25], MIPO [3], bc-opt [9], and SIP [24]. 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 [1, 2], 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 [32], BoB [6], PPBB-Lib [34], and PICO [11]. PARINO [23] and FATCOP [7] are parallel MIP solvers.

3 Organization of the Manual

In Sections 4, we briefly describe branch, cut, and price for those readers wishing to have a review of the basic methodology. In Section 5, we will describe the overall design of SYMPHONY without reference to the implementational details and with only passing reference to parallelism. In Section 6, we will then move on to discuss the details of the implementation. In Section 7, we briefly discuss issues involved in parallel execution of SYMPHONY. In Section 8, we discuss how to develop applications using SYMPHONY. Section 9 contains a description of the API, and finally, SYMPHONY's parameters are described in Section 10.

4 Introduction to Branch, Cut, and Price

4.1 Branch and Bound

Branch and bound is the broad class of algorithms from which branch, cut, and price is descended. A branch and bound algorithm uses a divide and conquer strategy 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 problem, and let $c \in \mathbf{R}^S$ be a vector of costs associated with members of S. Suppose we wish to determine a least cost member of S and we are given $\hat{s} \in S$, a "good" solution determined heuristically. 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 $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify n subsets of S, $S_1, \ldots, S_n$, 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 need 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, or *prune* it. Otherwise, we compare the lower bound to our global upper bound. 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

Bounding Operation

Input: A subproblem $\mathcal{S}$, described in terms of a "small" set of inequalities $\mathcal{L}'$ such that $\mathcal{S} = \{x^s : s \in \mathcal{F} \text{ and } ax^s \leq \beta \ \forall \ (a, \beta) \in \mathcal{L}'\}$ and α, an upper bound on the global optimal value.

Output: Either (1) an optimal solution $s^* \in \mathcal{S}$ to the subproblem, (2) a lower bound on the optimal value of the subproblem, or (3) a message `pruned` indicating that the subproblem should not be considered further.

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

Step 2. Solve the LP $\min\{cx : ax \leq \beta \ \forall \ (a, \beta) \in \mathcal{C}\}$.

Step 3. If the LP has a feasible solution $\hat{x}$, then go to Step 4. Otherwise, STOP and output `pruned`. This subproblem has no feasible solutions.

Step 4. If $c\hat{x} < \alpha$, then go to Step 5. Otherwise, STOP and output `pruned`. This subproblem cannot produce a solution of value better than α.

Step 5. If $\hat{x}$ is the incidence vector of some $\hat{s} \in \mathcal{S}$, then $\hat{s}$ is the optimal solution to this subproblem. STOP and output $\hat{s}$ as s^*. Otherwise, apply separation algorithms and heuristics to $\hat{x}$ to get a set of violated inequalities $\mathcal{C}'$. If $\mathcal{C}' = \emptyset$, then $c\hat{x}$ is a lower bound on the value of an optimal element of $\mathcal{S}$. STOP and return $\hat{x}$ and the lower bound $c\hat{x}$. Otherwise, set $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$ and go to Step 2.

Figure 1: Bounding in the branch and cut algorithm

this subproblem to the list of active candidates. We continue in this way until the list of active subproblems is empty, at which point our current best solution is the optimal one.

4.2 Branch, Cut, and Price

In many applications, the bounding operation is accomplished using the tools of linear programming (LP), a technique first described in full generality by Hoffman and Padberg [17]. This general class of algorithms is known as *LP-based branch and bound.* Typically, the integrality constraints of an integer programming formulation of the problem are relaxed to obtain a *LP relaxation*, which is then solved to obtain a lower bound for the problem. In [27], 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. They called this technique *branch and cut.* Since then, many implementations (including ours) have been fashioned around the framework they described for solving the Traveling Salesman Problem.

As an example, let a combinatorial optimization problem $CP = (E, \mathcal{F})$ with *ground set* E and *feasible set* $\mathcal{F} \subseteq 2^E$ be given along with a cost function $c \in \mathbf{R}^E$. The incidence vectors corresponding to the members of $\mathcal{F}$ are sometimes specified as the 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 [26]) that there exists a finite set $\mathcal{L}$ of inequalities valid for $\mathcal{P}$ such that

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

The inequalities in $\mathcal{L}$ are the potential cutting planes to be added to the relaxation as needed. Unfortunately, it is usually difficult, if not impossible, to enumerate all of inequalities in $\mathcal{L}$ or we

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

Figure 2: Branching in the branch and cut algorithm

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

Figure 3: Description of the generic branch and cut algorithm

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 Figure 1, we describe more precisely how the bounding operation is carried out in branch and cut.

Once we have failed to either prune the current subproblem or separate the current fractional solution from $\mathcal{P}$, we are forced to branch. The branching operation is accomplished by specifying a set of hyperplanes which divide the current subproblem in such a way that the current solution is not feasible for the LP relaxation of any of the new subproblems. For example, in a combinatorial optimization problem, branching could be accomplished simply by fixing a variable whose current value is fractional to 0 in one branch and 1 in the other. The procedure is described more formally in Figure 2. Figure 3 gives a high level description of the generic branch and cut algorithm.

As with cutting planes, the columns of A can also be defined implicitly if n is large. If column i is not present in the current matrix, then variable x_i is implicitly taken to have value zero. The process of dynamically generating variables is called *pricing* in the jargon of linear programming, but can also be viewed as that of generating cutting planes for the dual of the current LP relaxation. Hence, LP-based branch and bound algorithms in which the variables are generated dynamically when needed are known as *branch and price* algorithms. In [5], Barnhart, et al. provide a thorough review of these methods.

When both variables and cutting planes are generated dynamically during LP-based branch and bound, the technique becomes known as *branch, cut, and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of cuts and that of variables. We will further examine this symmetry later in the manual. 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 are very similar to each other anyway), combining the two techniques requires much more sophisticated methods than either one requires on its own. This is an important idea that is at the core of our design.

In the remainder of the manual, we will often use the term *search tree.* This term derives from the common 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

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

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

5.1 An Object-oriented Approach

As we have already alluded to, 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 cuts 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 cuts 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 cuts 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 cuts and variables. Hence, SYMPHONY can be considered an object-oriented framework with the central "objects" being the cuts 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) (see Section 9) 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 parameter setting.

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 extremely 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. These function will be described in detail in Section 9.

5.2 Data Structures and Storage

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

In the literature on linear and integer programming, the terms *cut* and *row* are typically used interchangeably. Similarly, *variable* and *column* are often used with similar meanings. In many situations, this is appropriate and does not cause confusion. However, in object-oriented BCP frameworks, such as SYMPHONY or ABACUS [19, 18], a *cut* and a *row* are *fundamentally different objects*. A *cut* (also referred to as a *constraint*) is a user-defined representation of an abstract object which can only be realized as a row in an LP matrix *with respect to a particular set of active variables*. Similarly, a *variable* is a representation which can only be realized as a column of an LP matrix with respect to a *particular set of cuts*. This distinction between the *representation* and the *realization* of objects is a crucial design element and is what allows us to effectively address some of the challenges inherent in BCP. In the remainder of this section, we further discuss this distinction and its implications.

5.2.1 Variables

In SYMPHONY, problem variables are *represented* by a unique global index assigned to each variable by the user. This index represents each variable's position in a "virtual" global list known only to the user. The main requirement of this indexing scheme is that, given an index and a list of active cuts, the user must be able to generate the corresponding column to be added to the matrix. 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 some set partitioning models, 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 cuts), 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) is planned for a future version of SYMPHONY.

For efficiency, the problem variables can be divided into two sets, the *base variables* and the *extra variables*. The base variables are active in all subproblems, whereas the extra variables can be added and removed. There is no theoretical difference between base variables and extra variables; however, designating a well-chosen set of base variables can significantly increase efficiency. Because they can move in and out of the problem, maintaining extra variables requires additional bookkeeping and computation. If the user has reason to believe a priori that a variable is "good" or has a high probability of having a non-zero value in some optimal solution to the problem, then that variable should be designated as a base variable. It is up to the user to designate which variables should be active in the root subproblem. Typically, when column generation is used, only base variables are active. Otherwise, all variables must be active in the root node.

5.2.2 Constraints

Because the global list of potential constraints (also called cuts) 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 nonzero variables in a particular class of constraints corresponds to the set of edges across a cut in a graph. Instead of storing the indices of each variable explicitly, one could simply store the set of nodes on one side ("shore") of the cut as a bit array. The constraint could then be constructed easily for any particular set of active variables (edges).

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.

5.2.3 Search Tree

Having described the basics of how objects are represented, we now describe the representation of search tree nodes. Since the base 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 complete description of the current basis is maintained to allow a warm start to the computation in each search node. This basis is either inherited from the parent, computed during strong branching (see Section 6.2.3), or comes from earlier partial processing of the node itself (see Section 6.3.3). Along with the set of active objects, we must also store the identity of the object(s) which were branched upon to generate the node. The branching operation is described in Section 6.2.3.

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 in distributed fashion over a 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). an schematic overview of the modules is presented in Figure 4. In the remainder of the section, we describe the modularization scheme and the implementation of each module in a sequential environment.

5.3.1 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.

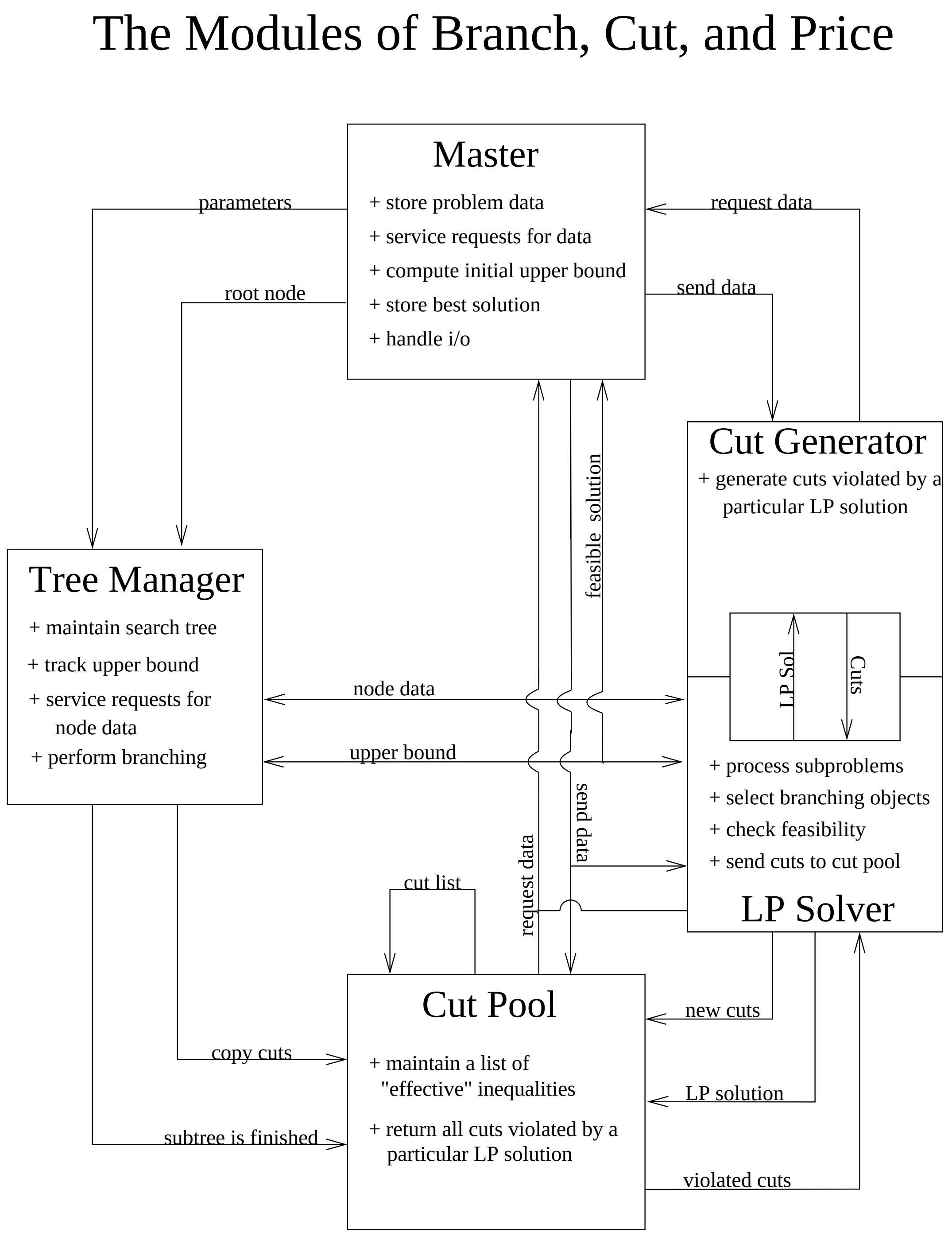

Figure 4: Schematic overview of the branch, cut, and price algorithm

- 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 is finished and print out data.
- Ensure that all modules are still functioning.

5.3.2 The Tree Manager Module

The *tree manager* controls the overall execution of the algorithm. It tracks the status of all processes, as well as that of the search tree, and distributes the subproblems to be processed to the LP module(s). 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.

5.3.3 The Linear Programming Module

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

- Inform the tree manager when a new subproblem is needed.
- Receive a subproblem 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.

5.3.4 The Cut Generator Module

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

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

5.3.5 The Cut Pool Module

The concept of a *cut pool* was first suggested by Padberg and Rinaldi [27], and is based on the observation that in BCP, the inequalities which are generated while processing a particular node in the search tree are also generally valid and potentially useful at other nodes. Since generating these cuts is usually a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree so far for use in processing future subproblems. Hence, the cut pool functions as an auxiliary cut generator. More explicitly, here are the functions of the cut pool module:

- 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 what is known as a single-pool BCP algorithm. The term *single-pool* refers to the fact that there is a single central list of candidate subproblems to be processed, which is maintained by the tree manager. Most sequential implementations use such a single-pool scheme. However, other schemes may be used in parallel implementations. For a description of various types of parallel branch and bound, see [14].

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 functions control 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 Section 6.2.3 for a description of the branching operation). A schematic summary of the algorithm is shown in Figure 4.

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 strategy minimizes the overall size of the search tree. However, at times, it will be advantageous to *dive* down in the tree. The concepts of *diving* and *search chains*, introduced in Section 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 the implementation of 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 erase this bias by adding additional functionality for handling variable generation and storage. This is the approach already taken by of COIN/BCP [8]. For more discussion of the reasons for this bias and the differences between the treatment of cuts and variables, see Section 6.2.2.

6 Details of the Implementation

6.1 The Master Module

The primary functions performed by the master module were listed in Section 5.3.1. If needed, the user must provide a routine to read problem-specific parameters in from the parameter file. She must also provide a subroutine for upper bounding if desired, though upper bounds can also be provided explicitly. A good initial upper bound can dramatically decrease the solution time by allowing more variable-fixing and earlier pruning of search tree nodes. If no upper bounding subroutine is available, then the two-phase algorithm, in which a good upper bound is found quickly in the first phase using a reduced set of variables can be advantageous. See Section 6.3.3 for details. The user's only unavoidable obligation during pre-processing is to specify the list of base 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 base variables can make a marked difference in solution speed, especially 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 processing and 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.

6.2.1 The LP 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 [10], 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.

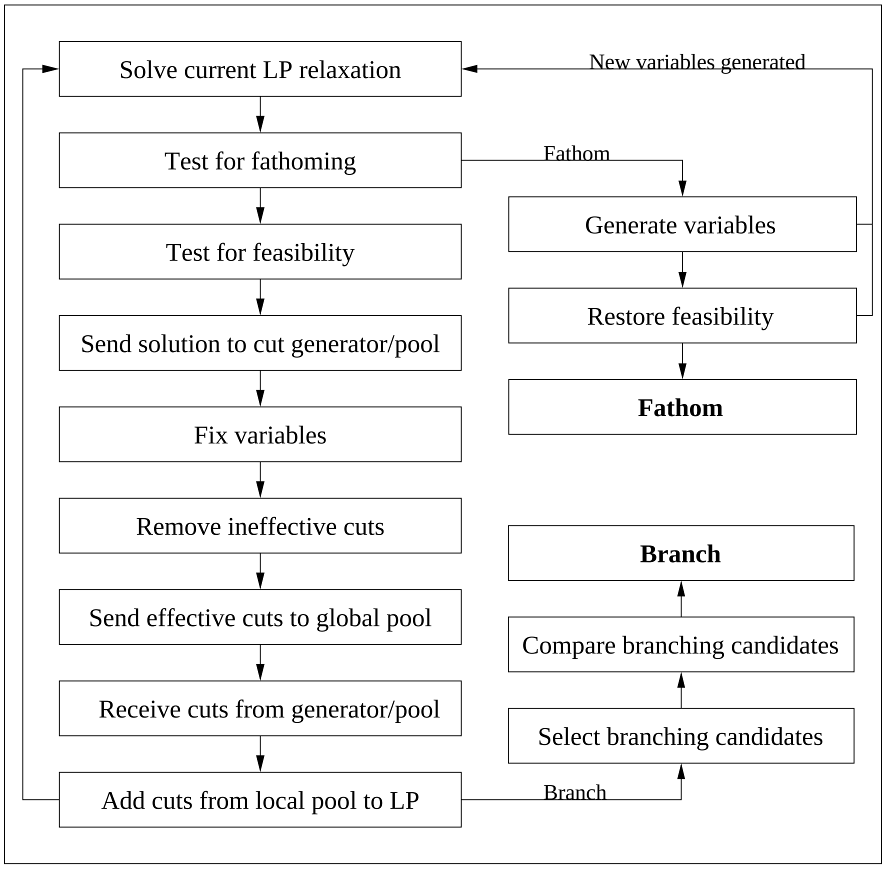
Solve current LP relaxation
Test for fathoming
Test for feasibility
Send solution to cut generator/pool
Fix variables
Remove ineffective cuts
Send effective cuts to global pool
Receive cuts from generator/pool
Add cuts from local pool to LP
New variables generated
Fathom
Generate variables
Restore feasibility
Fathom
Branch
Compare branching candidates
Select branching candidates
Branch

Figure 5: Overview of the LP solver loop

6.2.2 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. 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. 6.3.3). 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. 6.3.3, 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.

6.2.3 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; 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 information 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

6.3.1 Managing the Search Tree

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.

6.3.2 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.

6.3.3 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 process, 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 passed back as a group once the function exits. This allows the LP to begin adding cuts and solving the current relaxation before the cut generator is finished if desired. Parameters controlling if and when the LP should begin solving the relaxation before the cut generator is finished can be set by the user.

6.5 The Cut Pool Module

6.5.1 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 does seem to work reasonably well.

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 to keep it as small as possible.

6.5.2 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 gets assigned to another cut pool. After that, it continues to serve all the descendants of its assigned node that are not assigned to other cut pools.

Initially, the first cut pool is assigned to the root node. All other cut pools are unassigned. During execution, when a new node is sent to be processed, the tree manager must determine which cut pool the node should be serviced by. The default is to use the same cut pool as its parent. However, if there is currently an idle cut pool process (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 is assigned to this new node. All the cuts currently in the cut pool of its parent node are copied to the new pool to initialize it, after which the two pools operate independently on their respective subtrees. When generating cuts, the LP process 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 two-fold. 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 spread over multiple processors, 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 the current search node. 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 [26]).

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 point the reader to the survey of parallel branch and bound algorithms by Gendron and Crainic [14], as well as other references such as [11, 16, 31, 20].

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

The most straightforward parallel implementation, which is the one we currently employ, is a master-slave model, in which there is a central manager responsible for partitioning the work and parceling it out to the various slave processes that perform the actual computation. The reason we chose this approach is because it allows memory-efficient data structures for sequential computation and yet is conceptually easy to parallelize. Unfortunately, this approach does have limited scalability. For further discussions on the scalibility of BCP algorithms and approaches to improving it, see [21] and [22].

7.1 Details of the Parallel Implementation

7.1.1 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.

7.1.2 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 [13] 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`.

7.1.3 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 warm restart if a fault occurs. Similarly, the cut pool process can be warm-started 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 Developing Applications with SYMPHONY

SYMPHONY (Single- or Multi-Process Optimization over Networks) Version 2.8 is a powerful environment for implementing branch, cut, and price algorithms. The subroutines in the SYMPHONY library comprise a state-of-the-art solver which is designed to be completely modular and easy to port to various problem settings. All library subroutines are generic—their implementation does not depend on the the problem-setting. To develop a full-scale, parallel branch and cut algorithm, the user has only to specify a few problem-specific functions such as preprocessing and separation. The vast majority of the computation takes place within a "black box," of which the user need have no knowledge. SYMPHONY communicates with the user's routines through well-defined interfaces and performs all the normal functions of branch and cut—tree management, LP solution, cut pool management, as well as inter-process or inter-thread communication. Although there are default options, the user can also assert control over the behavior of SYMPHONY through a myriad of parameters and optional subroutines. SYMPHONY can be built in a variety of configurations, ranging from fully parallel to completely sequential, depending on the user's needs. The library runs serially on almost any platform, and can also run in parallel in either a fully distributed environment (network of workstations) or shared-memory environment simply by changing a few options in the make file. To run in a distributed environment, the user must have installed *Parallel Virtual Machine* (PVM) software, available for free from Oak Ridge National Laboratories at `http://www.ccs.ornl.gov/pvm/` . To run in a shared memory environment, the user must have installed an OpenMP compliant compiler. A cross-platform compiler called *Omni*, which uses `cc` or `gcc` as a back end, is available for free download at `http://pdplab.trc.rwcp.or.jp/Omni` . This section of the manual is concerned with the detailed specifications needed to develop an application using SYMPHONY. It is assumed that the user has already read the first part of the manual, which provides a high-level introduction to parallel branch, cut, and price and the overall design and use of SYMPHONY.

8.1 New in Version 2.8

If you are new to SYMPHONY, you can skip to Section 8.3. Here is a list the new features available in SYMPHONY 2.8:

- **New search rules**. There are new search rules available in the tree manager. These rules enable better control of diving (see Section 10.4).

- **More accurate timing information**. Reported timing information is now more accurate.

- **Idle Time Reporting**. Measures of processor idle time are now reported in the run statistics.

- **More efficient cut pool management**. Cuts are now optionally ranked and purged according to a user-defined measure of quality. See the description of `user_check_cut()` (Section 9.4).

- **Easier use of built-in branching functions**. Built-in branching functions can now be more easily called directly by the user if desired. Previously, these functions required the passing of internal data structures, making them difficult for the user to call directly. See the functions branch_* in the file `LP/lp_branch.c` for usage.

- **Better control of strong branching**. A new strong branching strategy allows the user to specify that more strong branching candidates should be used near the top of the tree where branching decisions are more critical. See the description of the relevant parameters (Section 10.5).

8.2 Changes to the User API

There are some minor changes to the user interface in order to allow the use of the new features. If you have code written for an older version, you will have to make some very minor modifications before compiling with version 2.8.

- `user_start_heurs()` (Section9.1) now includes as an optional return value a user-calculated estimate of the optimal upper bound. This estimate is used to control diving. See the description of the new diving rules (see Section 10.4) for more information. Since this return value is optional, you need only add the extra argument to your function definition to upgrade to the 2.8 interface. No changes to your code are required.

- `user_check_cut()` (Section9.4) now includes as an optional return value a user-defined assessment of the current quality of the cut. Since this return value is optional, you need only add the extra argument to your function definition to upgrade to the 2.8 interface. No changes to your code are required.

- `user_select_candidates()` (Section9.2) now passes in the value of the current level in the tree in case the user wants to use this information to make branching decisions. Again, the new argument just needs to be added to the function definition. No changes to your code are required.

8.3 Getting Started

Here is a sketch outline of how to get started with SYMPHONY. This is basically the same information contained in the README file that comes with the distribution.

Because SYMPHONY is inherently intended to be compiled and run on multiple architectures and in multiple configurations, I have chosen not to use the automatic configuration scripts provided by GNU. With the make files provided, compilation for multiple architectures and configurations can be done in a single directory without reconfiguring or "cleaning". This is very convenient, but it means that there is some hand configuring to do and you might need to know a little about your computing environment in order to make SYMPHONY compile. For the most part, this is limited to editing the make file and providing some path names. Also, for this reason, you may have to live with some complaints from the compiler because of missing function prototypes, etc.

Note that if you choose not to install PVM, you will need to edit the make file and provide an environment variable which makes it possible for "make" to determine the current architecture. This environment variable also allows the path to the binaries for each architecture to be set appropriately. This should all be done automatically if PVM is installed correctly.

Preparing for compilation

- First unpack the distribution by typing "`tar -xzf SYMPHONY-2.8.tgz`".

- Edit the various path variables in the make file (`SYPHONY-2.8/Makefile`) to match where you installed the source code and where the LP libraries and header files reside for each architecture on your network. Other architecture-dependent variables should also be set as required. Be sure to read the comments in the make file to understand what variables have to be set.

Compiling the sequential version

- Type "`make`" in the SYMPHONY root directory. This will first make the SYMPHONY library (sequential version). After this step is completed, you are free to type "`make clean`" and/or delete the `$ROOT/obj.*` and `$ROOT/dep.*` directories if you want to save disk space. You should only have to remake the library if you change something in SYMPHONY's internal files.

- After making the libraries, SYMPHONY will compile the user code and then make the executable for the sample application, a vehicle routing and traveling salesman problem solver. The name of the executable will be "`master_tm_lp_cg_cp`", indicating that all modules are contained in a single executable.

- To test the sample program, you can get some problem files from `http://branchandcut.org/VRP/data/` `or the TSPLIB (http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/) . The file format is that specified for the TSPLIB. There is also one sample file included with the distribution. Make sure the executable directory is in your path and type ‘‘master_tm_lp_cg_cp -F sample.vrp -N 5’’, where sample.vrp is the sample problem file. The -N argument gives the number of routes, which must be specified in advance. TSP instances can also be solved, but in this case, the number of routes does not need to be specified.`

Compiling for shared memory

- To compile a shared memory version, obtain an OpenMP compliant compiler, such as Omni (free from `http://pdplab.trc.rwcp.or.jp/Omni`) . Other options are listed at the OpenMP Web site (`http://www.openmp.org`) .

- Set the variable `CC` to the compiler name in the make file and compile as above.

- Voila, you have a shared memory parallel solver.

- Note that if you have previously compiled the sequential version, then you should first type "`make clean_all`", as this version uses the same compilation directories as the sequential version. With one active subproblem allowed, it should run exactly the same as the sequential version so there is no need to compile both.

Compiling for distributed networks

- You must first obtain and install the *Parallel Virtual Machine* (PVM) software, available for free from Oak Ridge National Laboratories at `http://www.ccs.ornl.gov/pvm/` . See Section 8.8 for more notes on using PVM.

- In the Makefile, be sure to set the `COMM_PROTOCOL` to `PVM`. Also, change one or more of `COMPILE_IN_TM`, `COMPILE_IN_LP`, `COMPILE_IN_CG`, and `COMPILE_IN_CP`, to `FALSE`, or you will end up with the sequential version. Various combinations of these variables will give you different configurations and different executables. See Section 8.12 for more info on setting them. Also, be sure to set the path variables in the make file appropriately so that make can find the PVM library.

- Type "`make`" in the SYMPHONY root directory to make the distributed libraries. As in Step 1 of the sequential version, you may type "`make clean`" after making the library. It should not have to remade again unless you modify SYMPHONY's internal files.

- After the libraries, all executables requested will be made.

- Make sure there are links from your $PVM_ROOT/bin/$PVM_ARCH/ directory to each of the executables in the Vrp/bin.$REV directory. This is required by PVM.

- Start the PVM daemon by typing "`pvm`" on the command line and then typing "`quit`".

- To test the sample program, you can get some problem files from `http://branchandcut.org/VRP/data/` `or the TSPLIB (http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/) . The file format is that specified for the TSPLIB. There is also one sample file included with the distribution. Make sure the executable directory is in your path and type ‘‘master -F sample.vrp -N 5’’, where sample.vrp is the sample problem file. The -N argument gives the number of routes, which must be specified in advance. TSP instances can also be solved, but in this case, the number of routes does not need to be specified. Note that the actual executable name may not be ‘‘master’’ if COMPILE_IN_TM is set to TRUE in the make file. See Section 8.12 for more information on executable names.`

This should result in the successful compilation of the sample application. Once you have accomplished this much, you are well on your way to having an application of your own. Don't be daunted by the seemingly endless list of user function that you are about to encounter. Most of them are optional or have default options. If you get lost, consult the source code for the sample application to see how it's done.

8.4 Source Files

The easiest way to get oriented is to examine the organization of the source files. When you unpack the SYMPHONY distribution, you will notice that the source files are organized along the lines of the modules. There is a separate directory for each module—master (`Master`), tree manager (`TreeManager`), cut generator (`CutGen`), cut pool (`CutPool`), and LP solver (`LP`). In addition, there is a directory called `DrawGraph` and a directory called `Common` that also contain source files. The `DrawGraph` directory provides an interface from SYMPHONY to the *Interactive Graph Drawing* software package developed by Marta Esö. This is an excellent utility for graphical display and debugging. The `Common` directory contains source code for functions used by multiple modules.

Within each module's directory, there is a primary source file containing the function `main()` (named `*.c` where * is the module name), a source file containing functions related to inter-process communication (named `*_proccomm.c`) and a file containing general subroutines used by the module (named `*_func.c`). The master is the exception and is organized slightly differently. The LP process source code is further subdivided due to the sheer number of functions.

The `include` directory contains the header files. Corresponding to each module, there are three header files, one containing internal data structures and function prototypes associated with the module (named `*.h` where * is the module name), one containing the data structures for storing the parameters (these are also used by the master process), and the third containing the function prototypes for the user functions (name `*_u.h`). By looking at the header files, you should get a general idea of how things are laid out.

In addition to the subdirectories corresponding to each module, there are subdirectories corresponding to applications. The sample application is contained in the directory `Vrp/`. The files containing function stubs that can be filled in to create a new application are contained in the directory `User/`. There is one file for each module, initially called `User/*/*_user.c`. The primary thing that you, as the user, need to understand to build an application is how to fill in these stubs. That is what the second section of this manual is about.

8.5 User-written Functions

The majority of the user functions are called from either the master process or the LP process. For these two modules, user functions are invoked from so-called *wrapper functions* that provide the interface. Each wrapper function is named `*_u()` , where `*` is the name of the corresponding user function, and is defined in a file called `*_wrapper.c`. The wrapper function first collects the necessary data and hands it to the user by calling the user function. Based on the return value from the user, the wrapper then performs any necessary post-processing. Most user functions are designed so that the user can do as little or as much as she likes. Where it is feasible, there are default options that allow the user to do nothing if the default behavior is acceptable. This is not possible in all cases and the user must provide certain functions, such as separation.

In the next section, the user functions will be described in detail. The name of every user written function starts with `user_`. There are three kinds of arguments:

IN: An argument containing information that the user might need to perform the function.

OUT: A pointer to an argument in which the user should return a result (requested data, decision, etc.) of the function.

INOUT: An argument which contains information the user might need, but also for which the user can change the value.

The return values from each function are as follows:

Return values:

`ERROR`	Error in the user function. Printing an error message is the user's responsibility. Depending on the work the user function was supposed to do, the error might be ignored (and some default option used), or the process aborts.
`USER_AND_PP`	The user implemented both the user function and post-processing (post-processing by SYMPHONY will be skipped).
`USER_NO_PP`	The user implemented the user function only.
`DEFAULT`	The default option is going to be used (the default is one of the built-in options, SYMPHONY decides which one to use based on initial parameter settings and the execution of the algorithm).
`built_in_option1`	
`built_in_option2` ...	The specified built-in option will be used.

Notes:

- Sometimes an output is optional. This will always be noted in the function descriptions.
- If an array has to be returned (i.e., the argument is `type **array`) then (unless otherwise noted) the user has to allocate space for the array itself and set `*array` to be the array allocated. If an output array is optional then the user *must not* set `*array` for the array she is not going to fill up because this is how SYMPHONY decides which optional arrays are filled up.
- Some built-in options are implemented so that the user can invoke them directly from the user function. This might be useful if, for example, the user wants to use different built-in options at different stages of the algorithm, or if he wants to do the post-processing himself but does not want to implement the option itself.

8.6 Data Structures

8.6.1 Internal Data Structures

With few exceptions, the data structures used internally by SYMPHONY are undocumented and most users will not need to access them directly. However, if such access is desired, a pointer to the main data structure used by each of the modules can be obtained simply by calling the function `get_*_ptr()` where * is the appropriate module (see the header files). This function will return a pointer to the data structure for the appropriate module. Casual users are advised against modifying SYMPHONY's internal data structures directly.

8.6.2 User-defined Data Structures

The user can define her own data structure for each module to maintain problem-specific data and any other information the user needs access to. A pointer to this data structure is maintained by SYMPHONY and is passed to the user as an argument to each user function. Since SYMPHONY knows nothing about this data structure, it is up to the user to allocate it, maintain it, and free it as required.

8.7 Inter-process Communication for Distributed Computing

While the implementation of SYMPHONY strives to shield the user from having to know anything about communications protocols or the specifics of inter-process communication, it may be necessary for the user to pass information from one module to another in some cases—for instance, if the user must pass problem-specific data to the LP process after reading them in from a data file. In cases where this might be appropriate, user functions are supplied in pairs—a *send* function and a *receive* function. All data are sent in the form of arrays of either type `char`, `int`, or `double`, or as strings. To send an array, the user has simply to invoke the function `send_?_array(? *array, int length)` where ? is one of the previously listed types. To receive that array, there is a corresponding function called `receive_?_array(? *array, int length)`. When receiving an array, the user must first allocate the appropriate amount of memory. In cases where variable length arrays need to be passed, the user must first pass the length of the array (as a separate array of length one) and then the array itself. In the receive function, this allows the length to be received first so that the proper amount of space can be allocated before receiving the array itself. Note that data must be received in exactly the same order as it was passed, as data is read linearly into and out of the message buffer. The easiest way to ensure this is done properly is to simply copy the send statements into the receive function and change the function names. It may then be necessary to add some allocation statements in between the receive function calls.

8.8 Working with PVM

To compile a distributed application, it is necessary to install PVM. The current version of PVM can be obtained at `http://www.ccs.ornl.gov/pvm/`. It should compile and install without any problem. You will have to make a few modifications to your `.cshrc` file, such as defining the `PVM_ROOT` environment variable, but this is all explained clearly in the PVM documentation. Note that all executables (or at least a link to them) must reside in the `$PVM_ROOT/bin/$PVM_ARCH` directory in order for parallel processes to be spawned correctly. The environment variable `PVM_ARCH` is set in your `.cshrc` file and contains a string representing the current architecture type. To run a parallel application, you must first start up the daemon on each of the machines you plan to use in the computation. How to do this is also explained in the PVM documentation.

8.9 Communication with Shared Memory

In the shared memory configuration, it is not necessary to use message passing to move information from one module to another since memory is globally accessible. In the few cases where the user would ordinarily have to pass information using message passing, it is easiest and most efficient to simply copy the information to the new location. This copying gets done in the *send* function and hence the *receive* function is never actually called. This means that the user must perform all necessary initialization, etc. in the send function. This makes it a little confusing to write source code which will work for all configurations. However, the confusion should be cleared up by looking at the sample application, especially the file `Vrp/Master/vrp.c`.

8.10 The LP Engine

SYMPHONY requires the use of a third-party callable library to solve the LP relaxations once they are formulated. Currently, CPLEX© is the only available option. Any LP solver with the appropriate capabilities can be interfaced with SYMPHONY by writing a set of interface routines contained in the file `LP/lp_solver.c`. Once the interface routines are written, the make file must be modified to link with the new LP solver.

8.11 Developing an Application

Once the user functions are filled in, all that remains is to compile the application. The distribution comes with two make files that facilitate this process. The primary make file resides in the root directory. The user make file resides in the user's subdirectory, initially called `User/`. There are a number of variables that must be set in the primary make file. Read the comments in the file `SYMPHONY-2.8/Makefile` to ensure that everything is set properly. The user make file shouldn't require much modification unless you add source files other than the ones included in the distribution or change their names.

When you are ready, type "`make`" to make the executables. SYMPHONY will create three subdirectories—`User/obj.*`, `User/bin.*`, and `User/dep.*` where * is a number corresponding the current architecture (determined by the `PVM_ARCH` environment variable). Note that if you don't have PVM installed, you should either modify the make file appropriately (read the make file to see how to do this) or set the `PVM_ARCH` environment variable by hand. If your architecture is not be listed in the make file, edit it by following the example set by the architectures already included. Make sure to set the corresponding path variables properly. Be sure to also set the proper links from the `$PVM_ROOT/bin/$PVM_ARCH` as explained in the previous section if you are compiling a distributed version.

8.12 Configuring the Modules

In the `make` file, there are four variables that control which modules run as separate executables and which are called directly in serial fashion. The variables are as follows:

COMPILE_IN_CG: If set to `TRUE`, then the cut generator function will be called directly from the LP in serial fashion, instead of running as a separate executable. This is desirable if cut generation is quick and running it in parallel is not worth the price of the communication overhead.

COMPILE_IN_CP: If set to `TRUE`, then the cut pool(s) will be maintained as a data structure auxiliary to the tree manager.

COMPILE_IN_LP: If set to `TRUE`, then the LP functions will be called directly from the tree manager. When running the distributed version, this necessarily implies that there will only be one active subproblem at a time, and hence the code will essentially be running serially. IN the shared-memory version, however, the tree manager will be threaded in order to execute subproblems in parallel.

COMPILE_IN_TM: If set to `TRUE`, then the tree will be managed directly from the master process. This is only recommended if a single executable is desired (i.e. the three other variables are also set to true). A single executable is extremely useful for debugging purposes.

These variables can be set in virtually any combination, though some don't really make much sense. Note that in a few user functions that involve process communication, there will be different versions for serial and parallel computation. This is accomplished through the use of `#ifdef` statements in the source code. This is well documented in the function descriptions and the in the source files containing the function stubs. See also Section 8.9.

8.13 Executable Names

In order to keep track of the various possible configurations, executable and their corresponding libraries are named as follows. For the fully distributed version, the names are `master`, `tm`, `lp`, `cg`, and `cp`. For other configurations, the executable name is a combination of all the modules that were compiled together joined by underscores. In other words, if the LP and the cut generator modules were compiled together (i.e. `COMPILE_IN_CG` set to `TRUE`), then the executable name would be "`lp_cg`" and the corresponding library file would be called "`liblp_cg.a`." You can rename the executables as you like. However, if you are using PVM to spawn the modules, as in the fully distributed version, you must set the parameters `*_exe` in the parameter file to the new executable names. See Section 10.4 for information on setting parameters in the parameter file.

8.14 Debugging Your Application

8.14.1 The First Rule

SYMPHONY has many built-in options to make debugging easier. The most important one, however, is the following rule. **It is easier to debug the fully sequential version than the fully distributed version**. Debugging parallel code is not terrible, but it is more difficult to understand what is going on when you have to look at the interaction of several different modules running as separate processes. This means multiple debugging windows which have to be closed and restarted each time the application is re-run. For this reason, it is highly recommended to develop code that can be compiled serially even if you eventually intend to run in a fully distributed environment. This does make the coding marginally more complex, but believe me, it's worth the effort. The vast majority of your code will be the same for either case. Make sure to set the compile flag to "`-g`" in the make file.

8.14.2 Debugging with PVM

If you wish to venture into debugging your distributed application, then you simply need to set the parameter `*_debug`, where * is the name of the module you wish to debug, to the value "4" in the parameter file (the number "4" is chosen by PVM). This will tell PVM to spawn the particular process or processes in question under a debugger. What PVM actually does in this case is to launch the script `$PVM_ROOT/lib/debugger`. You will undoubtedly want to modify this script to launch your preferred debugger in the manner you deem fit. If you have trouble with this, please send e-mail to the list serve (see Section 8.16).

It's a little tricky to debug interacting parallel processes, but you will quickly get the idea. The main difficulty is in that the order of operations is difficult to control. Random interactions can occur when processes run in parallel due to varying system loads, process priorities, etc. Therefore, it may not always be possible to duplicate errors. To force runs that you should be able to reproduce, make sure the parameter `no_cut_timeout` appears in the parameter file or start SYMPHONY with the "`-a`" option. This will keep the cut generator from timing out, a major source of randomness. Furthermore, run with only one active node allowed at a time (set `max_active_nodes` to "1"). This will keep the tree search from becoming random. These two steps should allow runs to be reproduced. You still have to be careful, but this should make things easier.

8.14.3 Using `Purify` and `Quantify`

The make file is already set up for compiling applications using `purify` and `quantify`. Simply set the paths to the executables and type "`make pall`" or "`p*`" where * is the module you want to purify. The executable name is the same as described in Section 8.13, but with a "p" in front of it. To tell PVM to launch the purified version of the executables, you must set the parameters `*_exe` in the parameter file to the purified executable names. See Section 10.4 for information on setting parameters in the parameter file.

8.14.4 Checking the Validity of Cuts and Tracing the Optimal Path

Sometimes the only evidence of a bug is the fact that the optimal solution to a particular problem is never found. This is usually caused by either (1) adding an invalid cut, or (2) performing an invalid branching. There are two options available for discovering such errors. The first is for checking the validity of added cuts. This checking must, of course, be done by the user, but SYMPHONY can facilitate such checking. To do this, the user must fill in the function `user_check_validity_of_cut()` (see Section 9.3). THIS function is called every time a cut is passed from the cut generator to the LP and can function as an independent verifier. To do this, the user must pass (through her own data structures) a known feasible solution. Then for each cut passed into the function, the user can check whether the cut is satisfied by the feasible solution. If not, then there is a problem! Of course, the problem could also be with the checking routine. To see how this is done, check out the sample application file `Vrp/cg_user.c`. After filling in this function, the user must recompile everything (including the libraries) after uncommenting the line in the make file that contains "`BB_DEFINES += -DCHECK_CUT_VALIDITY`." Type "`make clean_all`" and then "`make`."

Tracing the optimal path can alert the user when the subproblem which admits a particular known feasible solution (at least according to the branching restrictions that have been imposed so far) is pruned. This could be due to an invalid branching. Note that this option currently only works for branching on binary variables. To use this facility, the user must fill in the function `user_send_feas_sol()` (see Section 9.1). All that is required is to pass out an array of user indices that are in the feasible solution that you want to trace. Each time the subproblem which admits this feasible solution is branched on, the branch that continues to admit the solution is marked. When one of these marked subproblems is pruned, the user is notified.

8.14.5 Using the `Interactive Graph Drawing` Software

The Interactive Graph Drawing (IGD) software package is included with SYMPHONY and SYMPHONY facilitates its use through interfaces with the package. The package, which is a Tcl/Tk application, is extremely useful for developing and debugging applications involving graph-based problems. Given display coordinates for each node in the graph, IGD can display support graphs corresponding to fractional solutions with or without edge weights and node labels and weights, as well as other information. Furthermore, the user can interactively modify the graph by, for instance, moving the nodes apart to "disentangle" the edges. The user can also interactively enter violated cuts through the IGD interface.

To use IGD, you must have installed PVM since the drawing window runs as a separate application and communicates with the user's routines through message passing. To compile the graph drawing application, type "`make dglib dg`" in the SYMPHONY root directory. The user routines in the file `dg_user.c` can be filled in, but it is not necessary to fill anything in for basic applications.

After compiling `dg`, the user must write some subroutines that communicate with `dg` and cause the graph to be drawn. Regrettably, this is currently a little more complicated than it needs to be and is not well documented. However, by looking at the sample application, it is relatively easy to see how it should be done. To enable graph drawing, put the line `do_draw_graph 1` into the parameter file or use the `-d` command line option.

8.14.6 Other Debugging Techniques

Another useful built-in function is MakeMPS, which will write the current LP relaxation to a file in MPS format. This file can then be read into the LP solver interactively or examined by hand for errors. Many times, CPLEX gives much more explicit error messages interactively than through the callable library. The form of the function is

```
void MakeMPS(LPData *lp_data, int bc_index, int iter_num)
```

The matrix is written to the file "`matrix.[bc_index].[iter_num].mps`" where *bc_index* is the usually passed as the index of the current subproblem and *iter_num* is the current iteration number. These can, however, be any numbers the user chooses. If SYMPHONY is forced to abandon solution of an LP because the LP solver returns an error code, the current LP relaxation is automatically written to the file "`matrix.[bc_index].[iter_num].mps`" where *bc_index* is the index of the current subproblem and *iter_num* is the current iteration number. MakeMPS can be called using breakpoint code to examine the status of the matrix at any point during execution.

Logging is another useful feature. Logging the state of the search tree can help isolate some problems more easily. See Section 10.4 for the appropriate parameter settings to use logging.

8.15 Controlling Execution and Output

Calling SYMPHONY with no arguments simply lists all command-line options. Most of the common parameters can be set on the command line. Usually it is easier to use a parameter file. To invoke SYMPHONY with a parameter file type "`master -f filename ...`" where filename is the name of the parameter file. The format of the file is explained in Section 10.

The output level can be controlled through the use of the verbosity parameter. Setting this parameter at different levels will cause different progress messages to be printed out. Level 0 only prints out the introductory and solution summary messages, along with status messages every 10 minutes. Level 1 prints out a message every time a new node is created. Level 3 prints out messages describing each iteration of the solution process. Levels beyond 3 print out even more detailed information.

There are also two possible graphical interfaces. For graph-based problems, the Interactive Graph Drawing Software allows visual display of fractional solutions, as well as feasible and optimal solutions discovered during the solution process. For all types of problems, VBCTOOL creates a visual picture of the branch and cut tree, either in real time as the solution process evolves or as an emulation from a file created by SYMPHONY. See Section 10.4 for information on how to use VBCTOOL with SYMPHONY. Binaries for VBCTOOL can be obtained at
`http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html`.

8.16 Other Resources

There is a SYMPHONY user's list serve for posting questions/comments. To subscribe, send "`subscribe symphony-users`" to `majordomo@branchandcut.org`. There is also a Web site for SYMPHONY at `http://branchandcut.org/SYMPHONY` . Bug reports can be sent to `symphony-bugs@branchandcut.org`.

9 The User API Specification

9.1 User-written functions of the Master process

▷ user_usage

```
void user_usage()
```

Description:

The user can use any capitol letter (except 'H') for command line switches to control user-defined parameter settings without the use of a parameter file. The function `user_usage()` can optionally print out usage information for the user-defined command line switches. The command line switch `-H` automatically calls the user's usage subroutine. The switch `-h` prints SYMPHONY's own usage information.

▷ user_initialize

```
int user_initialize(void **user)
```

Description:

The user allocates space for and initializes the user-defined data structures for the master process.

Arguments:

`void **user`	OUT	Pointer to the user-defined data structure.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	Initialization is done.

▷ user_free_master

```
int user_free_master(void **user)
```

Description:

The user frees all the data structures within `*user`, and also free `*user` itself. This can be done using the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on return).

Return values:

`ERROR`	Ignored. This is probably not a fatal error.
`USER_NO_PP`	Everything was freed successfully.

▷ user_readparams

`int user_readparams(void *user, char *filename, int argc, char **argv)`

Description:

The user reads in parameters from the file named `filename`. The file `filename` is a file containing both built-in parameters and user parameters. The filename is given as a command line argument when starting the application and is then passed to the user. The user must open the file for reading, scan the file for lines that contain user parameters and then read the parameters in as appropriate. See the file `Master/master_io.c` to see how SYMPHONY does this.

Optionally, the user can also parse the command line arguments. All capital letters are reserved for user-defined command line switches. The switch `-H` is reserved for help and calls the user's usage subroutine (see `user_send_lp_data()`).

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`char *filename`	IN	The name of the parameter file.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	User parameters were read successfully.

▷ user_io

`int user_io(void *user)`

Description:

The user prepares all information needed to specify the problem instance (e.g., reads in data from a data file, etc.).

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	User I/O was completed successfully.

▷ user_init_draw_graph

`int user_init_draw_graph(void *user, int dg_id)`

Description:

This function is invoked only if the `do_draw_graph` parameter is set. The user can initialize the graph drawing process by sending some initial information (e.g., the location of the nodes of a graph, like in the TSP.)

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int dg_id`	IN	The process id of the graph drawing process.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	The user completed initialization successfully.

▷ user_start_heurs

```
int user_start_heurs(void *user, double *ub, double *ub_estimate)
```

Description:

The user invokes heuristics and generates the initial global upper bound and also perhaps an upper bound estimate. This is the last place where the user can do things before the branch and cut algorithm starts. She might do some preprocessing, in addition to generating the upper bound.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`double *ub`	OUT	Pointer to the global upper bound. Initially, the upper bound is set to either `-MAXDOUBLE` or the bound read in from the parameter file, and should be changed by the user only if a better valid upper bound is found.
`double *ub_estimate`	OUT	Pointer to an estimate of the global upper bound. This is useful if the `BEST_ESTIMATE` diving strategy is used (see the treemanager parameter `diving_strategy` (Section 10.4))

Return values:

`ERROR`	Error. This error is probably not fatal.
`USER_NO_PP`	User executed function successfully.

▷ user_set_base

```
int user_set_base(void *user, int *basevarnum, int **basevars, double **lb,
                  double **ub, int *basecutnum, int *colgen_strat)
```

Description:

The user must specify the set of base variables and the number of base constraints. The base constraints themselves need not be specified since they are never stored explicitly.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int *varnum`	OUT	Pointer to the number of base variables.
`int **userind`	OUT	Pointer to an array containing the user indices of the base variables.
`int **lb`	OUT	Pointer to an array containing the lower bounds for the base variables.
`int **ub`	OUT	Pointer to an array containing the upper bounds for the base variables.
`int *cutnum`	OUT	The number of base constraints.
`int *colgen_strat`	INOUT	The default strategy or one that has been read in from the parameter file is passed in, but the user is free to change it. See `colgen_strat` in the description of parameters for details on how to set it.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	The required data are filled in, but no post-processing done.
`USER_AND_PP`	All required post-processing done.

Post-processing:

The array of user indices is sorted if the user has not already done so.

▷ user_create_root

```
int user_create_root(void *user, int *extravarnum, int **extravars)
```

Description:

The user must specify which extra variables are to be active in the root node in addition to the base variables.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int *extravarnum`	OUT	Pointer to the number of extra active variables in the root.
`int *extravars`	OUT	Pointer to an array containing a list of user indices of the extra variables to be active in the root.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	All required data filled out, but no post-processing done.
`USER_AND_PP`	All required post-processing done.

Post-processing:

The array of extra indices is sorted if the user has not already done so.

▷ user_receive_feasible_solution

```
int user_receive_feasible_solution(void *user, int msgtag, double cost,
                                   int numvars, int *indices, double *values)
```

Description:

Feasible solutions can be sent and/or stored in a user-defined packed form if desired. For instance, the TSP, a tour can be specified simply as a permutation, rather than as a list of variable indices. In the LP process, a feasible solution is packed either by the user or by a default packing routine. If the default packing routine was used, the `msgtag` will be `FEASIBLE_SOLUTION_NONZEROS`. In this case, `cost`, `numvars`, `indices` and `values` will contain the solution value, the number of nonzeros in the feasible solution, and their user indices and values. The user has only to interpret and store the solution. Otherwise, when `msgtag` is `FEASIBLE_SOLUTION_USER`, SYMPHONY will send and receive the solution value only and the user has to unpack exactly what she has packed in the LP process. In this case the contents of the last three arguments are undefined.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int msgtag`	IN	`FEASIBLE_SOLUTION_NONZEROS` or `FEASIBLE_SOLUTION_USER`
`double cost`	IN	The cost of the feasible solution.
`int numvars`	IN	The number of variables whose user indices and values were sent (length of `indices` and `values`).
`int *indices`	IN	The user indices of the nonzero variables.
`double *values`	IN	The corresponding values.

Return values:

`ERROR`	Ignored. This is probably not a fatal error.
`USER_NO_PP`	The solution has been unpacked and stored.

▷ user_send_lp_data

```
int user_send_lp_data(void *user, void **user_lp)
```

Description:

The user has to send all problem-specific data that will be needed in the LP process to set up the initial LP relaxation and perform later computations. This could include instance data, as well as user parameter settings. This is one of the few places where the user will need to worry about the configuration of the modules. If either the tree manager or the LP are running as a separate process (either `COMPILE_IN_LP` or `COMPILE_IN_TM` are `FALSE` in the `make` file), then the data will be sent and received through message-passing. See `user_receive_lp_data()` in Section 9.2 for more discussion. Otherwise, it can be copied over directly to the user-defined data structure for the LP. In the latter case, `*user_lp` is a pointer to the user-defined data structure for the LP that must be allocated and initialized. For a discussion of message-passing in SYMPHONY, see Section 8.7. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_lp`	OUT	Pointer to the user-defined data structure for the LP process.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	Packing is done.

▷ user_send_cg_data

```
int user_pack_cg_data(void *user, void **user_cg)
```

Description:

The user has to send all problem-specific data that will be needed by the cut generator for separation. This is one of the few places where the user will need to worry about the configuration of the modules. If either the tree manager, the LP, or the cut generator are running as a separate process (either `COMPILE_IN_LP`, `COMPILE_IN_TM`, or `COMPILE_IN_CG` are `FALSE` in the make file), then the data will be sent and received through message-passing. See `user_receive_cg_data` in Section 9.3 for more discussion. Otherwise, it can be copied over directly to the user-defined data structure for the CG. In the latter case, `*user_cg` is a pointer to the user-defined data structure for the CG that must be allocated and initialized. For a discussion of message-passing in SYMPHONY, see Section 8.7. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_cg`	OUT	Pointer to the user-defined data structure for the cut generator process.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	Packing is done.

▷ user_send_cp_data

```
int user_pack_cp_data(void *user, void **user_cp)
```

Description:

The user has to send all problem-specific data that will be needed by the cut pool in order to store and check cuts. This is one of the few places where the user will need to worry about the configuration of the modules. If either the tree manager, the LP, or the cut pool are running as a separate process(either `COMPILE_IN_LP`, `COMPILE_IN_TM`, or `COMPILE_IN_CP` are `FALSE` in the `make` file), then the data will be sent and received through message-passing. See `user_receive_cp_data()` in Section 9.4 for more discussion. Otherwise, it can be copied over directly to the user-defined data structure for the CP. In the latter case, `*user_cp` is a pointer to the user-defined data structure for the CP that must be allocated and initialized. For a discussion of message passing in SYMPHONY, see Section 8.7. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_cp`	OUT	Pointer to the user-defined data structure for the cut pool process.

Return values:

`ERROR`	Error. SYMPHONY stops.
`USER_NO_PP`	Packing is done.

▷ user_display_solution

```
int user_display_solution(void *user)
```

Description:

This function is invoked when the best solution found so far is to be displayed (after heuristics, after the end of the first phase, or the end of the whole algorithm). This can be done using either a text-based format or using the `drawgraph` process.

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	Displaying is done.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

▷ user_send_feas_sol

```
int user_process_own_messages(void *user, int *feas_sol_size, int **feas_sol)
```

Description:

This function is useful for debugging purposes. It passes a known feasible solution to the tree manager. The tree manager then tracks which current subproblem admits this feasible solution and notifies the user when it gets pruned. It is useful for finding out why a known optimal solution never gets discovered. Usually, this is due to either an invalid cut of an invalid branching. Note that this feature only works when branching on binary variables. See Section 8.14.4 for more on how to use this feature.

Return values:

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int *feas_sol_size`	INOUT	Pointer to size of the feasible solution passed by the user.
`int **feas_sol`	INOUT	Pointer to the array of user indices containing the feasible solution. This array is simply copied by the tree manager and must be freed by the user.

`ERROR`	Solution tracing is not enabled.
`USER_NO_PP`	Tracing of the given solution is enabled.

▷ user_process_own_messages

```
int user_process_own_messages(void *user, int msgtag)
```

Description:

The user must receive any message he sends to the master process (independently of SYMPHONY's own messages). An example for such a message is sending feasible solutions from separate heuristics processes fired up in `user_start_heurs()`.

Arguments:

`void *user`	`IN`	`Pointer to the user-defined data structure.`
`int msgtag`	`IN`	`The message tag of the message.`

Return values:

`ERROR`	`Ignored.`
`USER_NO_PP`	`Message is processed.`

9.2 User-written functions of the LP process

Data Structures

We first describe a few structures that are used to pass data into and out of the user functions of the LP process.

▷ cut_data

One of the few internally defined data structures that the user has to deal with frequently is the `cut_data` data structure, used to store the packed form of cuts. This structure has 8 fields listed below.

`int size` – The size of the `coef` array.

`char *coef` – An array containing the packed form of the cut, which is defined and constructed by the user. Given this packed form and a list of the variables active in the current relaxation, the user must be able to construct the corresponding constraint.

`double rhs` – The right hand side of the constraint.

`double range` – The range of the constraint. It is zero for a standard form constraint. Otherwise, the row activity level is limited to between `rhs` and `rhs` + `range`.

`char type` – A user-defined type identifier that represents the general class that the cut belongs to.

`char sense` – The sense of the constraint. Can be either 'L' ($\leq$), 'E' ($=$), 'G' ($\geq$) or 'R' (ranged). This may be evident from the `type`.

`char branch` – Determines whether the cut can be branched on or not. Possible initial values are `DO_NOT_BRANCH_ON_THIS_ROW` and `ALLOWED_TO_BRANCH_ON`.

`int name` – Identifier used by SYMPHONY. The user should not set this.

▷ waiting_row

A closely related data structure is the `waiting_row`, essentially the "unpacked" form of a cut. There are six fields.

`source_pid` – Used internally by SYMPHONY.

`cut_data *cut` – Pointer to the cut from which the row was generated.

`int nzcnt, *matind, *matval` – Fields describing the row. `nzcnt` is the number of nonzeros in the row, i.e., the length of the `matind` and `matval` arrays, which are the variable indices (wrt. the current LP relaxation) and nonzero coefficients in the row.

`double violation` – If the constraint corresponding to the cut is violated, this value contains the degree of violation (the absolute value of the difference between the row activity level (i.e., lhs) and the right hand side). This value does not have to be set by the user.

▷ var_desc

The `var_desc` structure is used list the variables in the current relaxation. There are four fields.

`int userind` – The user index of the variables, `int colind` – The column index of the variables (in the current relaxation),

`double lb` – The lower bound of the variable,

`double ub` – The upper bound of the variable.

Function Descriptions

Now we describe the functions themselves.

▷ user_receive_lp_data

```
int user_receive_lp_data (void **user)
```

Description:

The user has to receive here all problem-specific information sent from the master, set up necessary data structures, etc. Note that the data need only be actively received and the user data structure allocated if either the TM or LP modules are configured as separate processes. Otherwise, data will have been copied into appropriate locations in the master function `user_send_lp_data()` (see Section 9.1). The two cases can be handled by means of `#ifdef` statements. See comments in the source code stubs for more details. Note that the data must be received in exactly the same order as it was sent from the master. See Section 8.7 for more notes on receiving data.

Arguments:

`void **user`	OUT	Pointer to the user-defined LP data structure.

Return values:

`ERROR`	Error. SYMPHONY aborts this LP process.
`USER_NO_PP`	User received the data.

Wrapper invoked from: `lp_initialize()` at process start.

▷ user_free_lp

```
int user_free_lp(void **user)
```

Description:

The user has to free all the data structures within `*user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined LP data structure.

Return values:

`ERROR`	Error. SYMPHONY ignores error message.
`USER_NO_PP`	User freed everything in the user space.

Wrapper invoked from: `lp_close()` at process shutdown.

▷ user_create_lp

```
int user_create_lp(void *user, int varnum, var_desc **vars, int
                   numrows, int cutnum, cut_data **cuts, int *nz,
                   int **matbeg, int **matind, double **matval,
                   double **obj, double **rhs, char **sense,
                   double **rngval, int *maxn, int *maxm,
                   int *maxnz, int *allocn, int *allocm, int *allocnz)
```

Description:

Based on the instance data contained in the user data structure and the list of cuts and variables that are active in the current subproblem, the user has to create the initial LP relaxation for the search node. The matrix of the LP problem must contain the variables whose user indices are listed in `vars` (in the same order) and at least the base constraints.

An LP is defined by a matrix of constraints, an objective function, and bounds on both the right hand side values of the constraints and on the variables. If the problem has n variables and m constraints, the constraints are given by a constraint coefficient matrix of size mxn (described in the next paragraph). The sense of each constraint, the right hand side values and bounds on the right hand side (called *range*) are vectors are of size m. The objective function coefficients and the lower and upper bounds on the variables are vectors of length n. The sense of each constraint can be either 'L' ($\leq$), 'E' ($=$), 'G' ($\geq$) or 'R' (ranged). For non-ranged rows the range value is `0`, for a ranged row the range value must be non-negative and the constraint means that the row activity level has to be between the right hand side value and the right hand side increased by the range value.

Since the coefficient matrix is very often sparse, only the nonzero entries are stored. Each entry of the matrix has a column index, a row index and a coefficient value associated with it. An LP matrix is specified in the form of the three arrays `*matval`, `*matind`, and `*matbeg`. The array `*matval` contains the values of the nonzero entries of the matrix in *column order*; that is, all the entries for the 0^th column come first, then the entries for the 1^st column, etc. The row index corresponding to each entry of `*matval` is listed in `*matind` (both of them are of length nz, the number of nonzero entries in the matrix). Finally, `*matbeg` contains the starting positions of each of the columns in `*matval` and `*matind`. Thus, `(*matbeg)[i]` is the position of the first entry of column i in both `*matval` and `*matind`). By convention `*matbeg` is allocated to be of length $n+1$, with `(*matbeg)[n]` containing the position after the very last entry in `*matval` and `*matind` (so it is very conveniently equal to nz). This representation of a matrix is known as a *column ordered* or *column major* representation.

The arrays that are passed in can be overwritten and have already been previously allocated for the lengths indicated (see the description of arguments below). Therefore, if they are big enough, the user need not reallocate them. If the max lengths are not big enough then she has to free the corresponding arrays and allocate them again. In this case she *must* return the allocated size of the array to avoid further reallocation. If the user plans to utilize dynamic column and/or cut generation, arrays should be allocated large enough to allow for reasonable growth of the matrix or unnecessary reallocations will result. In order to accommodate `*maxn` variables, arrays must be allocated to size `*allocn = *maxn + *maxm +1` and `*allocnz = *maxnz + *maxm` because of the extra space required by the LP solver for slack and artificial variables.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	Number of variables in the relaxation (base and extra).
`var_desc **vars`	IN	An array of length `n` containing the user indices of the active variables (base and extra).
`int rownum`	IN	Number of constraints in the relaxation (base and extra).
`int cutnum`	IN	Number of extra constraints.
`cut_data **cuts`	IN	Packed description of extra constraints.
`int *nz`	OUT	Pointer to the number of nonzeros in the LP.
`int **matbeg`	INOUT	Pointers to the arrays that describe the LP problem (see description above.
`int **matind`	INOUT	
`double **matval`	INOUT	
`double **obj`	INOUT	
`double **rhs`	INOUT	
`char **sense`	INOUT	
`double **rngval`	INOUT	
`int *maxn`	INOUT	The maximum number of variables.
`int *maxm`	INOUT	The maximum number of constraints.
`int *maxnz`	INOUT	The maximum number of nonzeros.
`int *allocn`	INOUT	The length of the `*matbeg` and `*obj` arrays (should be `*maxm + *maxn +1`).
`int *allocm`	INOUT	The length of the `*rhs`, `*sense` and `*rngval` arrays.
`int *allocnz`	INOUT	The length of the `*matval` and `*matind` arrays (should be `*maxnz + *maxm`.

Return values:

`ERROR`	Error. The LP process is aborted.
`USER_AND_PP`	Post-processing will be skipped, the user added the constraints corresponding to the cuts.
`USER_NO_PP`	User created the matrix with only the base constraints.

Post-processing:

The extra constraints are added to the matrix by calling the `user_unpack_cuts()` subroutine and then adding the corresponding rows to the matrix. This is easier for the user to implement, but less efficient than adding the cuts at the time the original matrix was being constructed.

Wrapper invoked from: `process_chain()` which is invoked when setting up a the initial search node in a chain.

▷ user_get_upper_bounds

```
int user_get_upper_bounds(void *user, int varnum, int *indices, double *ub)
```

Description:

The user has to return the upper bounds of the variables whose user indices are given. Note that space for `ub` is already allocated when this function is invoked. There is no post-processing. The default is to set all the upper bounds to 1.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	Length of `vars`.
`int *vars`	IN	Array containing the user indices of the variables.
`double *ub`	OUT	Array of upper bounds (to be filled out by the user).

Return values:

`ERROR`	Error. The LP process is aborted.
`DEFAULT`	Upper bounds are set to one.
`USER_NO_PP`	The user filled up the upper bound array.

Wrapper invoked from: `add_col_set()` (when SYMPHONY adds columns after pricing out) and from `create_lp_u()` (when SYMPHONY has to get the bounds on the extra variables in the new active node).

Note:

Only the upper bounds for extra variables are ever asked for since the array of bounds for the base variables is always maintained. Lower bounds for the extra variables must be zero and hence there is no corresponding function for lower bounds.

▷ user_is_feasible

```
int user_is_feasible(void *user, double lpetol, int varnum, int
                     *indices, double *values, int *feasible)
```

Description:

User tests the feasibility of the solution to the current LP relaxation.

There is no post-processing. Possible defaults are testing integrality (`TEST_INTEGRALITY`) and testing whether the solution is binary (`TEST_ZERO_ONE`).

Arguments:

`void *user`	INOUT	Pointer to the user-defined LP data structure.
`double lpetol`	IN	The ϵ tolerance of the LP solver.
`int varnum`	IN	The length of the `indices` and `values` arrays.
`int *indices`	IN	User indices of variables at nonzero level in the current solution.
`double *values`	IN	Values of the variables listed in `indices`.
`int *feasible`	OUT	Feasibility status of the solution (`NOT_FEASIBLE`, or `FEASIBLE`).

Return values:

`ERROR`	Error. Solution is considered to be not feasible.
`USER_NO_PP`	User checked IP feasibility.
`DEFAULT`	Regulated by the parameter `is_feasible_default`, but set to `TEST_INTEGRALITY` unless over-ridden by the user.
`TEST_INTEGRALITY`	Test integrality of the given solution.
`TEST_ZERO_ONE`	Tests whether the solution is binary.

Wrapper invoked from: `select_branching_object()` after pre-solving the LP relaxation of a child corresponding to a candidate and from `fathom_branch()` after solving an LP relaxation.

▷ user_send_feasible_solution

```
int user_send_feasible_solution(void *user, double lpetol,
                                int varnum, int *indices, double *values)
```

Description:

Send a feasible solution to the master process. The solution is sent using the communication functions described in Section 8.7 in whatever logical format the user wants to use. The default is to pack the user indices and values of variables at non-zero level. If the user packs the solution herself then the same data must be packed here that will be received in the `user_receive_feasible_solution()` function in the master process. See the description of that function for details. This function will only be called when either the LP or tree manager are running as a separate executable. Otherwise, the solution gets stored within the LP user data structure.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`double lpetol`	IN	The ϵ tolerance of the LP solver.
`int varnum`	IN	The length of the `indices` and `values` arrays.
`int *indices`	IN	User indices of variables at nonzero level in the current solution.
`double *values`	IN	Values of the variables listed in `indices`.

Return values:

ERROR	Error. Do the default.
USER_NO_PP	User packed the solution.
DEFAULT	Regulated by the parameter `pack_feasible_solution_default`, but set to SEND_NONZEROS unless over-ridden by the user.
SEND_NONZEROS	Pack the nonzero values and their indices.

Wrapper invoked: as soon as feasibility is detected anywhere.

▷ user_display_solution

```
int user_display_solution(void *user, int which_sol,
                          int varnum, int *indices, double *values)
```

Description:

Given a solution to an LP relaxation (the indices and values of the nonzero variables) the user can (graphically) display it. The `which_sol` argument shows what kind of solution is passed to the function: DISP_FEAS_SOLUTION indicates a solution feasible to the original IP problem, DISP_RELAXED_SOLUTION indicates the solution to any LP relaxation and DISP_FINAL_RELAXED_SOLUTION indicates the solution to an LP relaxation when no cut has been found. There is no post-processing. Default options print out user indices and values of nonzero or fractional variables on the standard output.

Arguments:

void *user	IN	Pointer to the user-defined LP data structure.
int which_sol	IN	The type of solution passed on to the displaying function. Possible values are DISP_FEAS_SOLUTION, DISP_RELAXED_SOLUTION and DISP_FINAL_RELAXED_SOLUTION.
int varnum	IN	The number of variables in the current solution at nonzero level (the length of the `indices` and `values` arrays).
int *indices	IN	User indices of variables at nonzero level in the current solution.
double *values	IN	Values of the nonzero variables.

Return values:

ERROR	Error. SYMPHONY ignores error message.
USER_NO_PP	User displayed whatever she wanted to.
DEFAULT	Regulated by the parameter `display_solution_default`.
DISP_NOTHING	Display nothing.
DISP_NZ_INT	Display user indices (as integers) and values of nonzero variables.
DISP_NZ_HEXA	Display user indices (as hexadecimals) and values of nonzero variables.
DISP_FRAC_INT	Display user indices (as integers) and values of variables not at their lower or upper bounds.
DISP_FRAC_HEXA	Display user indices (as hexadecimals) and values of variables not at their lower and upper bounds.

Wrapper invoked from: `fathom_branch()` with `DISP_FEAS_SOLUTION` or `DISP_RELAXED_SOLUTION` after solving an LP relaxation and checking its feasibility status. If it was not feasible and no cut could be added either then the wrapper is invoked once more, now with `DISP_FINAL_RELAXED_SOLUTION`.

▷ user_shall_we_branch

```
int user_shall_we_branch(void *user, double lpetol, int cutnum,
                         int slacks_in_matrix_num,
                         cut_data **slacks_in_matrix,
                         int slack_cut_num, cut_data **slack_cuts,
                         int varnum, var_desc **vars, double *x,
                         char *status, int *cand_num,
                         branch_obj ***candidates, int *action)
```

Description:

There are two user-written functions invoked from `select_candidates_u`. The first one (`user_shall_we_branch()`) decides whether to branch at all, the second one (`user_select_candidates()`) chooses the branching objects. The argument lists of the two functions are the same, and if branching occurs (see discussion below) then the contents of `*cand_num` and `*candidates` will not change between the calls to the two functions.

The first of these two functions is invoked in each iteration after solving the LP relaxation and (possibly) generating cuts. Therefore, by the time it is called, some violated cuts might be known. Still, the user might decide to branch anyway. The second function is invoked only when branching is decided on.

Given (1) the number of known violated cuts that can be added to the problem when this function is invoked, (2) the constraints that are slack in the LP relaxation, (3) the slack cuts not in the matrix that could be branched on (more on this later), and (4) the solution to the current LP relaxation, the user must decide whether to branch or not. Branching can be done either on variables or slack cuts. A pool of slack cuts which has been removed from the problem and kept for possible branching is passed to the user. If any of these happen to actually be violated (it is up to the user to determine this), they can be passed back as branching candidate type `VIOLATED_SLACK` and will be added into the current relaxation. In this case, branching does not have to occur (the structure of the `*candidates` array is described below in `user_select_candidates()`).

This function has two outputs. The first output is `*action` which can take four values: `USER__DO_BRANCH` if the user wants to branch, `USER__DO_NOT_BRANCH` if he doesn't want to branch, `USER__BRANCH_IF_MUST` if he wants to branch only if there are no known violated cuts, or finally `USER__BRANCH_IF_TAILOFF` if he wants to branch in case tailing off is detected. The second output is the number of candidates and their description. In this function the only sensible "candidates" are `VIOLATED_SLACK`s.

There is no post processing, but in case branching is selected, the `col_gen_before_branch()` function is invoked before the branching would take place. If that function finds dual infeasible variables then (instead of branching) they are added to the LP relaxation and the problem is resolved. (Note that the behavior of the `col_gen_before_branch()` is governed by the `colgen_strat[]` TM parameters.)

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`double lpetol`	IN	The ϵ tolerance of the LP solver.
`int cutnum`	IN	The number of violated cuts (known before invoking this function) that could be added to the problem (instead of branching).
`int slacks_in_matrix_num`	IN	Number of slack constraints in the matrix.
`cut_data **slacks_in_matrix`	IN	The description of the cuts corresponding to these constraints (see Section 9.2).
`int slack_cut_num`	IN	The number of slack cuts not in the matrix.
`cut_data **slack_cuts`	IN	Array of pointers to these cuts (see Section 9.2).
`int varnum`	IN	The number of variables in the current lp relaxation (the length of the following three arrays).
`var_desc **vars`	IN	Description of the variables in the relaxation.
`double *x`	IN	The corresponding solution values (in the optimal solution to the relaxation).
`char *status`	IN	The stati of the variables. There are five possible status values: `NOT_FIXED`, `TEMP_FIXED_TO_UB`, `PERM_FIXED_TO_UB`, `TEMP_FIXED_TO_LB` and `PERM_FIXED_TO_LB`.
`int *cand_num`	OUT	Pointer to the number of candidates returned (the length of `*candidates`).
`candidate ***candidates`	OUT	Pointer to the array of candidates generated (see description below).
`int *action`	OUT	What to do. Must be one of the four above described values.

Return values:

`ERROR`	Error. `DEFAULT` is used.
`USER_NO_PP`	The user filled out `*action` (and possibly `*cand_num` and `*candidates`).
`DEFAULT`	action is set to the value of the parameter `shall_we_branch_default`, which is initially `USER__BRANCH_IF_MUST` unless over-ridden by the user.

Notes:

- The user has to allocate the pointer array for the candidates and place the pointer for the array into `***candidates` (if candidates are returned).
- Candidates of type `VIOLATED_SLACK` are always added to the LP relaxation regardless of what `action` is chosen and whether branching will be carried out or not.
- Also note that the user can change his mind in `user_select_candidates()` and not branch after all, even if she chose to branch in this function. A possible scenario: `cut_num` is zero when this function is invoked and the user asks for `USER__BRANCH_IF_MUST` without checking the slack constraints and slack cuts. Afterwards no columns are generated (no dual infeasible variables found) and thus SYMPHONY decides branching is called for and invokes `user_select_candidates()`. However, in that function the user checks the slack cuts, finds that some are violated, cancels the branching request and adds the violated cuts to the relaxation instead.

Warning: The cuts the user unpacks and wants to be added to the problem (either because they are of type `VIOLATED_SLACK` or type `CANDIDATE_CUT_NOT_IN_MATRIX`) will be deleted from the list of slack cuts after this routine returns. Therefore the same warning applies here as in the function `user_unpack_cuts()`.

Wrapper invoked from: `select_branching_object()`.

▷ user_select_candidates

```
int user_select_candidates(void *user, double lpetol, int cutnum,
                           int slacks_in_matrix_num,
                           cut_data **slacks_in_matrix,
                           int slack_cut_num, cut_data **slack_cuts,
                           int varnum, var_desc **vars, double *x,
                           char *status, int *cand_num,
                           branch_obj ***candidates, int *action,
                           int bc_level)
```

Description:

The purpose of this function is to generate branching candidates. Note that `*action` from `user_shall_we_branch()` is passed on to this function (but its value can be changed here, see notes at the previous function), as well as the candidates in `**candidates` and their number in `*cand_num` if there were any.

Violated cuts found among the slack cuts (not in the matrix) can be added to the candidate list. These violated cuts will be added to the LP relaxation regardless of the value of `*action`.

The `branch_obj` structure contains fields similar to the `cut_data` data structure. Branching is accomplished by imposing inequalities which divide the current subproblem while cutting off the corresponding fractional solution. Branching on cuts and variables is treated symmetrically and branching on a variable can be thought of as imposing a constraint with a single unit entry in the appropriate column. Following is a list of the fields of the `branch_obj` data structure which must be set by the user.

`char type` Can take five values:

- `CANDIDATE_VARIABLE` The object is a variable.
- `CANDIDATE_CUT_IN_MATRIX` The object is a cut (it must be slack) which is in the current formulation.
- `CANDIDATE_CUT_NOT_IN_MATRIX` The object is a cut (it must be slack) which has been deleted from the formulation and is listed among the slack cuts.
- `VIOLATED_SLACK` The object is not offered as a candidate for branching, but rather it is selected because it was among the slack cuts but became violated again.
- `SLACK_TO_BE_DISCARDED` The object is not selected as a candidate for branching rather it is selected because it is a slack cut which should be discarded even from the list of slack cuts.

`int position` The position of the object in the appropriate array (which is one of `vars`, `slacks_in_matrix`, or `slack_cuts`.

`waiting_row *row` Used only if the type is `CANDIDATE_CUT_NOT_IN_MATRIX` or `VIOLATED_SLACK`. In these cases this field holds the row extension corresponding to the cut. This structure can be filled out easily using a call to `user_unpack_cuts()`.

`int child_num`
The number of children of this branching object.

`char *sense, double *rhs, double *range, int *branch`
The description of the children. These arrays determine the sense, rhs, etc. for the cut to be imposed in each of the children. These are defined and used exactly as in the `cut_data` data structure. **Note:** If a limit is defined on the number of children by defining the `MAX_CHILDREN_NUM` macro to be a number (it is pre-defined to be 4 as a default), then these arrays will be statically defined to be the correct length and don't have to be allocated. This option is highly recommended. Otherwise, the user must allocate them to be of length `child_num`.

`double lhs` The activity level for the row (for branching cuts). This field is purely for the user's convenience. SYMPHONY doesn't use it so it need not be filled out.

`double *objval, int *termcode, int *iterd, int *feasible`
The objective values, termination codes, number of iterations and feasibility stati of the children after pre-solving them. These are all filed out by SYMPHONY during strong branching. The user may access them in `user_compare_candidates()` (see below).

There are three default options (see below), each chooses a few variables (the number is determined by the strong branching parameters (see Section 10.5).

Arguments:

Same as for `user_shall_we_branch(), except that *action must be either USER__DO_BRANCH or USER__DO_NOT_BRANCH, and if branching is asked for, there must be a real candidate in the candidate list (not only VIOLATED_SLACKs and SLACK_TO_BE_DISCARDEDs). Also, the argument bc_level is the level in the tree. This could be used in deciding how many strong branching candidates to use.`

Return values:

`ERROR`	`Error. DEFAULT is used.`
`USER_NO_PP`	`User generated branching candidates.`
`DEFAULT`	`Regulated by the select_candidates_default parameter (one of the following three options).`
`USER__CLOSE_TO_HALF`	`Choose variables with values closest to half.`
`USER__CLOSE_TO_HALF_AND_EXPENSIVE`	`Choose variables with values close to half and with high objective function coefficients.`
`USER__CLOSE_TO_ONE_AND_CHEAP`	`Choose variables with values close to one and with low objective function coefficients.`

Wrapper invoked from: `select_branching_object()`.

Notes: `See the notes at user_shall_we_branch().`

▷ user_compare_candidates

```
int user_compare_candidates(void *user, branch_obj *can1, branch_obj *can2,
                            int *which_is_better)
```

Description:

By the time this function is invoked, the children of the current search tree node corresponding to each branching candidate have been pre-solved, i.e., the `objval`, `termcode`, `iterd`, and `feasible` fields of the `can1` and `can2` structures are filled out. Note that if the termination code for a child is `D_UNBOUNDED` or `D_OBJLIM`, i.e., the dual problem is unbounded or the objective limit is reached, then the objective value of that child is set to `MAXDOUBLE / 2`. Similarly, if the termination code is one of `D_ITLIM` (iteration limit reached), `D_INFEASIBLE` (dual infeasible) or `ABANDONED` (because of numerical difficulties) then the objective value of that child is set to that of the parent's.

Based on this information the user must choose which candidate he considers better and whether to branch on this better one immediately without checking the remaining candidates. As such, there are four possible answers: `FIRST_CANDIDATE_BETTER`, `SECOND_CANDIDATE_BETTER`, `FIRST_CANDIDATE_BETTER_AND_BRANCH_ON_IT` and `SECOND_CANDIDATE_BETTER_AND_BRANCH_ON_IT`. An answer ending with `_AND_BRANCH_ON_IT` indicates that the user wants to terminate the strong branching process and select that particular candidate for branching.

There are several default options. In each of them, objective values of the pre-solved LP relaxations are compared.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`branch_obj *can1`	IN	One of the candidates to be compared.
`branch_obj *can2`	IN	The other candidate to be compared.
`int *which_is_better`	OUT	The user's choice. See the description above.

Return values:

`ERROR`	Error. `DEFAULT` is used.
`USER_NO_PP`	User filled out `*which_is_better`.
`DEFAULT`	Regulated by the `compare_candidates_default` parameter, initially set to `LOWEST_LOW_OBJ` unless over-ridden by the user.
`BIGGEST_DIFFERENCE`	Prefer the candidate with the biggest difference between highest and lowest objective function values.
`LOWEST_LOW`	Prefer the candidate with the lowest minimum objective function value. The minimum is taken over the objective function values of all the children.
`HIGHEST_LOW`	Prefer the candidate with the highest minimum objective function value.
`LOWEST_HIGH`	Prefer the candidate with the lowest maximum objective function value.
`HIGHEST_HIGH`	Prefer the candidate with the highest maximum objective function value .

Wrapper invoked from: `select_branching_object()` after the LP relaxations of the children have been pre-solved.

▷ user_select_child

```
int user_select_child(void *user, double ub, branch_obj *can, char *action)
```

Description:

By the time this function is invoked, the candidate for branching has been chosen. Based on this information and the current best upper bound, the user has to decide what to do with each child. Possible actions for a child are `KEEP_THIS_CHILD` (the child will be kept at this LP for further processing, i.e., the process *dives* into that child), `PRUNE_THIS_CHILD` (the child will be pruned based on some problem specific property—no questions asked...), `PRUNE_THIS_CHILD_FATHOMABLE` (the child will be pruned based on its pre-solved LP relaxation) and `RETURN_THIS_CHILD` (the child will be sent back to tree manager). Note that at most one child can be kept at the current LP process.

There are two default options—in both of them, objective values of the pre-solved LP relaxations are compared (for those children whose pre-solve did not terminate with primal infeasibility or high cost). One rule prefers the child with the lowest objective function value and the other prefers the child with the higher objective function value.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int ub`	IN	The current best upper bound.
`double etol`	IN	Epsilon tolerance.
`branch_obj *can`	IN	The branching candidate.
`char *action`	OUT	Array of actions for the children. The array is already allocated to length `can->number`.

Return values:

`ERROR`	Error. `DEFAULT` is used.
`USER_NO_PP`	User filled out `*action`.
`USER_AND_PP`	User filled out `*action` and did an equivalent of the post-processing.
`DEFAULT`	Regulated by the `select_child_default` parameter, which is initially set to `PREFER_LOWER_OBJ_VALUE`, unless over-ridden by the user.
`PREFER_HIGHER_OBJ_VALUE`	Choose child with the highest objective value.
`PREFER_LOWER_OBJ_VALUE`	Choose child with the lowest objective value.

Post-processing:

Checks which children can be fathomed based on the objective value of their pre-solved LP relaxation.

Wrapper invoked from: `branch()`.

▷ user_print_branch_stat

```
int user_print_branch_stat(void *user, branch_obj *can, cut_data *cut,
                           char *action)
```

Description:

Print out information about branching candidate `can`, such as a more explicit problem-specific description than SYMPHONY can provide (for instance, end points of an edge). If `verbosity` is set high enough, the identity of the branching object and the children (with objective values and termination codes for the pre-solved LPs) is printed out to the standard output by SYMPHONY.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`branch_obj *can`	IN	The branching candidate.
`cut_data *cut`	IN	The description of the cut if the branching object is a cut.
`char *action`	IN	Array of actions for the children.

Return values:

`ERROR`	Error. Ignored by SYMPHONY.
`USER_NO_PP`	The user printed out whatever she wanted to.

Wrapper invoked from: `branch()` after the best candidate has been selected, pre-solved, and the action is decided on for the children.

▷ user_add_to_desc

```
int user_add_to_desc(void *user, int *desc_size, char **desc)
```

Description:

Before a node description is sent to the TM, the user can provide a pointer to a data structure that will be appended to the description for later use by the user in reconstruction of the node. This information must be placed into `*desc`. Its size should be returned in `*desc_size`.

There is only one default option: the description to be added is considered to be of zero length, i.e., there is no additional description.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int *desc_size`	OUT	The size of the additional information, the length of `*desc` in bytes.
`char **desc`	OUT	Pointer to the additional information (space must be allocated by the user).

Return values:

`ERROR`	Error. `DEFAULT` is used.
`USER_NO_PP`	User filled out `*desc_size` and `*desc`.
`DEFAULT`	No description is appended.

Wrapper invoked from: `create_explicit_node_desc()` before a node is sent to the tree manager.

▷ user_same_cuts

```
int user_same_cuts (void *user, cut_data *cut1, cut_data *cut2,
                    int *same_cuts)
```

Description:

Determine whether the two cuts are comparable (the normals of the half-spaces corresponding to the cuts point in the same direction) and if yes, which one is stronger. The default is to declare the cuts comparable only if the `type`, `sense` and `coef` fields of the two cuts are the same byte by byte; and if this is the case to compare the right hand sides to decide which cut is stronger.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`cut_data *cut1`	IN	The first cut.
`cut_data *cut2`	IN	The second cut.
`int *same_cuts`	OUT	Possible values: `SAME`, `FIRST_CUT_BETTER`, `SECOND_CUT_BETTER` and `DIFFERENT` (i.e., not comparable).

Return values:

`ERROR`	Error. `DEFAULT` is used.
`USER_NO_PP`	User did the comparison, filled out `*same_cuts`.
`DEFAULT`	Compare byte by byte (see above).

Wrapper invoked from: `process_message()` when a `PACKED_CUT` arrives.

Note:

This function is used to check whether a newly arrived cut is already in the local pool. If so, or if it is weaker than a cut in the local pool, then the new cut is discarded; if it is stronger then a cut in the local pool, then the new cut replaces the old one and if the new is different from all the old ones, then it is added to the local pool.

▷ user_unpack_cuts

```
int user_unpack_cuts(void *user, int from, int one_row_only, int varnum,
                     var_desc **vars, int cutnum, cut_data **cuts,
                     int *new_row_num, waiting_row ***new_rows)
```

Description:

The user has to interpret the given cuts as constraints for the current LP relaxation, i.e., he must decode the compact representation of the cuts (see the `cut_data` structure) into rows for the matrix. A pointer to the array of generated rows must be returned in `***new_rows` (the user has to allocate this array) and their number in `*new_row_num`.

There is no post processing. There are no built-in default options.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int from`	IN	See below in "Notes".
`int one_row_only`	IN	`UNPACK_CUTS_SINGLE` or `UNPACK_CUTS_MULTIPLE` (see notes below).
`int varnum`	IN	The number of variables.
`var_desc **vars`	IN	The variables currently in the problem.
`int cutnum`	IN	The number of cuts to be decoded.
`cut_data **cuts`	IN	Cuts that need to be converted to rows for the current LP. See "Warning" below.
`int *new_row_num`	OUT	Pointer to the number of rows in `**new_rows`.
`waiting_row ***new_rows`	OUT	Pointer to the array of pointers to the new rows.

Return values:

`ERROR`	Error. The cuts are discarded.
`USER_NO_PP`	User unpacked the cuts.

Wrapper invoked from: Wherever a cut needs to be unpacked (multiple places).

Notes:

- When decoding the cuts, the expanded constraints have to be adjusted to the current LP, i.e., coefficients corresponding to variables currently not in the LP have to be left out.

- If the one_row_only flag is set to UNPACK_CUTS_MULTIPLE, then the user can generate as many constraints (even zero!) from a cut as she wants (this way she can lift the cuts, thus adjusting them for the current LP). However, if the flag is set to UNPACK_CUTS_SINGLE, then for each cut the user must generate a unique row, the same one that had been generated from the cut before. (The flag is set to this value only when regenerating a search tree node.)
- The from argument can take on six different values: CUT_FROM_CG, CUT_FROM_CP, CUT_FROM_TM, CUT_LEFTOVER (these are cuts from a previous LP relaxation that are still in the local pool), CUT_NOT_IN_MATRIX_SLACK and CUT_VIOLATED_SLACK indicating where the cut came from. This might be useful in deciding whether to lift the cut or not.
- The matind fields of the rows must be filled with indices with respect to the position of the variables in **vars.
- **Warning:** For each row, the user must make sure that the cut the row was generated from (and can be uniquely regenerated from if needed later) is safely stored in the waiting_row structure. SYMPHONY will free the entries in cuts after this function returns. If a row is generated from a cut in cuts (and not from a lifted cut), the user has the option of physically copying the cut into the corresponding part of the waiting_row structure, or copying the pointer to the cut into the waiting_row structure and erasing the pointer in cuts. If a row is generated from a lifted cut, the user should store a copy of the lifted cut in the corresponding part of waiting_row.

▷ user_send_lp_solution

```
int user_send_lp_solution(void *user, int varnum, var_desc **vars,
                          double *x, int where)
```

Description:

The user has the option to send the LP solution to either the cut pool or the cut generator in some user-defined form if desired. There are two default options—sending the indices and values for all nonzero variables (SEND_NONZEROS) and sending the indices and values for all fractional variables (SEND_FRACTIONS).

Arguments:

void *user	IN	Pointer to the user-defined LP data structure.
int varnum	IN	The number of variables currently in the LP relaxation. (The length of the *vars and x arrays.)
var_desc **vars	IN	The variables currently in the LP relaxation.
double *x	IN	Values of the above variables.
int where	IN	Where the solution is to be sent—LP_SOL_TO_CG or LP_SOL_TO_CP.

Return values:

`ERROR`	Error. No message will be sent.
`USER_NO_PP`	User packed and sent the message.
`DEFAULT`	Regulated by the `pack_lp_solution_default` parameter, initially set to `SEND_NOZEROS`.
`SEND_NONZEROS`	Send user indices and values of variables at nonzero level.
`SEND_FRACTIONS`	Send user indices and values of variables at fractional level.

Wrapper invoked from: `fathom_branch()` after an LP relaxation has been solved. The message is always sent to the cut generator (if there is one). The message is sent to the cut pool if a search tree node at the top of a chain is being processed (except at the root in the first phase), or if a given number (`cut_pool_check_freq`) of LP relaxations have been solved since the last check.

Note:

The wrapper automatically packs the level, index, and iteration number corresponding to the current LP solution within the current search tree node, as well as the objective value and upper bound in case the solution is sent to a cut generator. This data will be unpacked by SYMPHONY on the receiving end, the user will have to unpack there exactly what he has packed here.

▷ user_logical_fixing

```
int user_logical_fixing(void *user, int varnum, var_desc **vars,
                        double *x, char *status)
```

Description:

Logical fixing is modifying the stati of variables based on logical implications derived from problem-specific information. In this function the user can modify the status of any variable. Valid stati are: `NOT_FIXED`, `TEMP_FIXED_TO_LB`, `PERM_FIXED_TO_LB`, `TEMP_FIXED_TO_UB` and `PERM_FIXED_TO_UB`. Be forewarned that fallaciously fixing a variable in this function can cause the algorithm to terminate improperly. Generally, a variable can only be fixed permanently if the matrix is *full* at the time of the fixing (i.e. all variables that are not fixed are in the matrix). There are no default options.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	The number of variables currently in the LP relaxation. (The length of the `*vars` and `x` arrays.)
`var_desc **vars`	IN	The variables currently in the LP relaxation.
`double *x`	IN	Values of the above variables.
`char *status`	INOUT	Stati of variables currently in the LP relaxation.

Return values:

`ERROR`	Error. Ignored by SYMPHONY.
`USER_NO_PP`	User changed the stati of the variables she wanted.

Wrapper invoked from: `fix_variables()` after doing reduced cost fixing, but only when a specified number of variables have been fixed by reduced cost (see LP parameter settings).

▷ user_generate_column

```
int user_generate_column(void *user, int generate_what, int cutnum,
                         cut_data **cuts, int prevind, int nextind,
                         int *real_nextind, double *colval,
                         int *colind, int *collen, double *obj)
```

Description:

This function is called when pricing out the columns that are not already fixed and are not explicitly represented in the matrix. Only the user knows the explicit description of these columns. When a missing variable need to be priced, the user is asked to provide the corresponding column. SYMPHONY scans through the known variables in the order of their user indices. After testing a variable in the matrix (`prevind`), SYMPHONY asks the user if there are any missing variables to be priced before the next variable in the matrix (`nextind`). If there are missing variables before `nextind`, the user has to supply the user index of the real next variable (`real_nextind`) along with the corresponding column. Occasionally SYMPHONY asks the user to simply supply the column corresponding to `nextind`. The `generate_what` flag is used for making a distinction between the two cases: in the former case it is set to `GENERATE_REAL_NEXTIND` and in the latter it is set to `GENERATE_NEXTIND`.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int generate_what`	IN	`GENERATE_NEXTIND` or `GENERATE_REAL_NEXTIND` (see description above).
`int cutnum`	IN	The number of added rows in the LP formulation (i.e., the total number of rows less the number of base constraints). This is the length of the `**cuts` array.
`cut_data **cuts`	IN	Description of the cuts corresponding to the added rows of the current LP formulation. The user is supposed to know about the cuts corresponding to the base constraints.
`int prevind`	IN	The last variable processed (−1 if there was none) by SYMPHONY.
`int nextind`	IN	The next variable (−1 if there are none) known to SYMPHONY.
`int *real_nextind`	OUT	Pointer to the user index of the next variable (−1 if there is none).
`double *colval`	OUT	Values of the nonzero entries in the column of the next variable. (Sufficient space is already allocated for this array.)
`int *colind`	OUT	Row indices of the nonzero entries in the column. (Sufficient space is already allocated for this array.)
`int *collen`	OUT	The length of the `colval` and `colind` arrays.
`double *obj`	OUT	Objective coefficient corresponding to the next variable.

Return values:

`ERROR`	Error. The LP process is aborted.
`USER_NO_PP`	User filled out `*real_nextind` and generated its column if needed.

Wrapper invoked from: `price_all_vars()` and `restore_lp_feasibility()`.

Note:

`colval`, `colind`, `collen` and `obj` do not need to be filled out if `real_nextind` is the same as `nextind` and `generate_what` is `GENERATE_REAL_NEXTIND`.

▷ user_generate_cuts_in_lp

```
int user_generate_cuts_in_lp(void *user, int varnum, var_desc **vars,
                             double *x, int *new_row_num,
                             waiting_row ***new_rows)
```

Description:

The user might decide to generate cuts directly within the LP process instead of using the cut generator. This can be accomplished either through a call to this function or simply by configuring SYMPHONY such that the cut generator is called directly from the LP solver. One example of when this might be done is when generating Gomory cuts (this is planned to be part of SYMPHONY later) or something else that requires knowledge of the current LP tableau. The IN arguments are the same as in `user_send_lp_solution()` (except that there is no `where` argument). Not only the generated cuts but the corresponding rows must be returned (the cuts are in the `waiting_row` structures) because the `user_unpack_cuts()` function will not be invoked for the generated cuts. Also, the user must fill out the `violation` field for every row. The reason for this is that any cut generated here will definitely correspond to the current LP solution so the user must have already computed the violation when generating the cut.

Post-processing consists of checking if any of the new cuts are already in the local pool (or dominated by a cut in the local pool). Since the user will probably use this function to generate tableau-dependent cuts, it is highly unlikely that any of the new cuts would already be in the pool. Therefore the user will probably return `USER_AND_PP` to force SYMPHONY to skip post-processing.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	The number of variables currently in the LP relaxation. (The length of the `*vars` and `x` arrays.)
`var_desc **vars`	IN	The variables currently in the LP relaxation.
`double *x`	IN	Values of the above variables.
`int *new_row_num`	OUT	The number of cuts generated.
`waiting_row ***new_rows`	OUT	The cuts and the corresponding rows.

Return values:

`ERROR`	Error. Interpreted as if no cuts were generated.
`USER_NO_PP`	Cuts were generated but SYMPHONY must compare them to those in the local pool.
`USER_AND_PP`	Cuts were generated and SYMPHONY should not compare them to those in the local pool.
`DEFAULT`	No cuts are generated. (At least for now. We might add Gomory cuts for default later.)

Post-processing:

SYMPHONY checks if any of the newly generated rows are already in the local pool.

Wrapper invoked from: `receive_cuts()` before the cuts from the CG process are received. Since the user will probably use this function to generate tableau-dependent cuts, it is highly unlikely that any of the new cuts would already be in the pool. Therefore the user will probably return `USER_AND_PP` to force SYMPHONY to skip post-processing.

Notes:

- Just like in `user_unpack_cuts()`, the user has to allocate space for the rows.
- Unless the `name` field of a cut is explicitly set to `CUT__SEND_TO_CP`, SYMPHONY will assume that the cut is locally valid only and set that field to `CUT__DO_NOT_SEND_TO_CP`.

▷ user_print_stat_on_cuts_added

```
int user_print_stat_on_cuts_added(void *user, int rownum, waiting_row **rows)
```

Description:

The user can print out some information (if he wishes to) on the cuts that will be added to the LP formulation. The default is to print out the number of cuts added.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int rownum`	IN	The number of cuts added.
`waiting_row **rows`	IN	Array of waiting rows containing the cuts added.

Return values:

`ERROR`	Revert to default.
`USER_AND_PP`	User printed whatever he wanted.
`DEFAULT`	Print out the number of cuts added.

Wrapper invoked from: `add_best_waiting_rows()` after it has been decided how many cuts to add and after the cuts have been selected from the local pool.

▷ user_purge_waiting_rows

```
int user_purge_waiting_rows(void *user, int rownum,
                            waiting_row **rows, char *delete)
```

Description:

The local pool is purged from time to time to control its size. In this function the user has the power to decide which cuts to purge from this pool if desired. To mark the `i`$^{\texttt{th}}$ waiting row (an element of the pre-pool) for removal she has to set `delete[i]` to be `TRUE` (`delete` is allocated before the function is called and its elements are set to `FALSE` by default).

Post-processing consists of actually deleting those entries from the waiting row list and compressing the list. The default is to discard the least violated waiting rows and keep no more than what can be added in the next iteration (this is determined by the `max_cut_num_per_iter` parameter).

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int rownum`	IN	The number of waiting rows.
`waiting_row **rows`	IN	The array of waiting rows.
`char *delete`	OUT	An array of indicators (each of them is one `char`) showing which waiting rows are to be deleted.

Return values:

`ERROR`	Purge every single waiting row.
`USER_AND_PP`	The user removed the unwanted waiting rows and compressed the remaining list.
`USER_NO_PP`	The user marked in `delete` the rows to be deleted.
`DEFAULT`	Described above.

Post-processing:

Delete the appropriate rows.

Wrapper invoked from: `receive_cuts()` after cuts have been added.

9.3 User-written functions of the CG process

Due to the relative simplicity of the cut generator, there are no wrapper functions implemented for CG. Consequently, there are no default options and no post-processing.

▷ user_receive_cg_data

```
int user_receive_cg_data (void **user)
```

Description:

The user has to receive here all problem-specific information that is known to the master and will be needed for computation in the CG process later on. The same data must be received here that was sent in the `user_send_cg_data()` (see Section 9.1) function in the master process. The user has to allocate space for all the data structures, including `user` itself. Note that some or all of this may be done in the function `user_send_cg_data()` if the Tree Manager, LP, and CG are all compiled together. See that function for more information.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.

Return values:

`ERROR`	Error. CG exits.
`USER_NO_PP`	The user received the data properly.

Invoked from: `cg_initialize()` at process start.

▷ user_receive_lp_solution_cg

```
int user_receive_lp_solution_cg(void *user)
```

Description:

This function is invoked only if in the `user_send_lp_solution()` function of the LP process the user opted for packing the current LP solution himself. Here he must unpack the very same data he packed there.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Invoked from: Whenever an LP solution is received.

Return values:

`ERROR`	Error. This LP solution is not processed.
`USER_NO_PP`	The user received the LP solution.

Note:

SYMPHONY automatically unpacks the level, index and iteration number corresponding to the current LP solution within the current search tree node as well as the objective value and upper bound.

▷ user_free_cg

```
int user_free_cg(void **user)
```

Description:

The user has to free all the data structures within `user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on exit from this function).

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	The user freed all data structures.

Invoked from: `cg_close()` at process shutdown.

▷ user_find_cuts

```
int user_find_cuts(void *user, int varnum, int iter_num, int level,
    int index, double objval, int *indices, double *values,
    double ub, double lpetol, int *cutnum)
```

Description:

The user can generate cuts based on the current LP solution stored in `soln`. Cuts found need to be sent back to the LP by calling the `cg_send_cut(cut_data *new_cut)` function. The argument of this function is a pointer to the cut to be sent. See Section 9.2 for a description of this data structure. If the user wants the cut to be added to the cut pool in case it proves to be effective in the LP, then `new_cut->name` should be set to `CUT__SEND_TO_CP`. Otherwise, it should be set to `CUT__DO_NOT_SEND_TO_CP`.

The only output of this function is the number of cuts generated and this value is returned in the last argument.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int iter_num`	IN	The iteration number of the current LP solution.
`int level`	IN	The level in the tree on which the current LP solution was generated.
`index`	IN	The index of the node in which LP solution was generated.
`objval`	IN	The objective function value of the current LP solution.
`int varnum`	IN	The number of nonzeros in the current LP solution.
`indices`	IN	The column indices of the nonzero variables in the current LP solution.
`values`	IN	The values of the nonzero variables listed in `indices`.
`double ub`	IN	The current global upper bound.
`double lpetol`	IN	The current error tolerance in the LP.
`int *cutnum`	OUT	Pointer to the number of cuts generated and sent to the LP.

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	The user function exited properly.

Invoked from: Whenever an LP solution is received.

▷ user_check_validity_of_cut

```
int user_check_validity_of_cut(void *user, cut_data *new_cut)
```

Description:

This function is provided as a debugging tool. Every cut that is to be sent to the LP solver is first passed to this function where the user can independently verify that the cut is valid by testing it against a known feasible solution (usually an optimal one). This is useful for determining why a particular known feasible (optimal) solution was never found. Usually, this is due to an invalid cut being added. See Section 8.14.4 for more on this feature.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`cut_data *new_cut`	IN	Pointer to the cut that must be checked.

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	The user is done checking the cut.

Invoked from: Whenever a cut is being sent to the LP.

9.4 User-written functions of the CP process

Due to the relative simplicity of the cut pool, there are no wrapper functions implemented for CP. Consequently, there are no default options and no post-processing.

▷ user_receive_cp_data

```
int user_receive_cp_data(void **user)
```

Description:

The user has to receive here all problem-specific information sent from `user_send_cp_data()` (see Section 9.1) function in the master process. The user has to allocate space for all the data structures, including `user` itself. Note that this function is only called if the either the Tree Manager, LP, or CP are running as a separate process (i.e. either `COMPILE_IN_TM`, `COMPILE_IN_LP`, or `COMPILE_IN_CP` are set to `FALSE` in the `make` file). Otherwise, this is done in `user_send_cp_data()`. See the description of that function for more details.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.

Return values:

`ERROR`	Error. Cut Pool exits.
`USER_NO_PP`	The user received data successfully.

Invoked from: `cp_initialize` at process start.

▷ user_free_cp

```
int user_free_cp(void **user)
```

Description:

The user has to free all the data structures within `user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on exit).

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	The user freed all data structures.

Invoked from: cp_close() at process shutdown.

▷ user_receive_lp_solution_cp

```
void user_receive_lp_solution_cp(void *user)
```

Description:

This function is invoked only if in the `user_send_lp_solution()` function of the LP process the user opted for packing the current LP solution herself. Here she must receive the very same data she sent there.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`ERROR`	Cuts are not checked for this LP solution.
`USER_NO_PP`	The user function exited properly.

Invoked from: Whenever an LP solution is received.

Note:

SYMPHONY automatically unpacks the level, index and iteration number corresponding to the current LP solution within the current search tree node.

▷ user_prepare_to_check_cuts

```
int user_prepare_to_check_cuts(void *user, int varnum, int *indices,
                               double *values)
```

Description:

This function is invoked after an LP solution is received but before any cuts are tested. Here the user can build up data structures (e.g., a graph representation of the solution) that can make the testing of cuts easier in the `user_check_cuts` function.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int varnum`	IN	The number of nonzero/fractional variables described in `indices` and `values`.
`int *indices`	IN	The user indices of the nonzero/fractional variables.
`double *values`	IN	The nonzero/fractional values.

Return values:

`ERROR`	Cuts are not checked for this LP solution.
`USER_NO_PP`	The user is prepared to check cuts.

Invoked from: Whenever an LP solution is received.

▷ user_check_cut

```
int user_check_cut(void *user, double lpetol, int varnum,
                   int *indices, double *values, cut_data *cut,
                   int *is_violated, double *quality)
```

Description:

The user has to determine whether a given cut is violated by the given LP solution (see Section 9.2 for a description of the `cut_data data` data structure). Also, the user can assign a number to the cut called the *quality*. This number is used in deciding which cuts to check and purge. See the section on Cut Pool Parameters for more information.

Arguments:

`void *user`	INOUT	The user defined part of p.
`double lpetol`	IN	The ϵ tolerance in the LP process.
`int varnum`	IN	Same as the previous function.
`int *indices`	IN	Same as the previous function.
`double *values`	IN	Same as the previous function.
`cut_data *cut`	IN	Pointer to the cut to be tested.
`int *is_violated`	OUT	TRUE/FALSE based on whether the cut is violated or not.
`double *quality`	OUT	a number representing the relative strength of the cut.

Return values:

`ERROR`	Cut is not sent to the LP, regardless of the value of `*is_violated`.
`USER_NO_PP`	The user function exited properly.

Invoked from: Whenever a cut needs to be checked.

Note:

The same note applies to `number`, `indices` and `values` as in the previous function.

▷ user_finished_checking_cuts

```
int user_finished_checking_cuts(void *user)
```

Description:

When this function is invoked there are no more cuts to be checked, so the user can dismantle data structures he created in `user_prepare_to_check_cuts`. Also, if he received and stored the LP solution himself he can delete it now.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`ERROR`	Ignored.
`USER_NO_PP`	The user function exited properly.

Invoked from: After all cuts have been checked.

9.5 User-written functions of the Draw Graph process

Due to the relative simplicity of the cut pool, there are no wrapper functions implemented for DG. Consequently, there are no default options and no post-processing.

▷ user_dg_process_message

```
void user_dg_process_message(void *user, window *win, FILE *write_to)
```

Description:

The user has to process whatever user-defined messages are sent to the process. A write-to pipe to the wish process is provided so that the user can directly issue commands there.

Arguments:

`void *user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	The window that received the message.
`FILE *write_to`	IN	Pipe to the wish process.

Return values:

`ERROR`	Error. Message ignored.
`USER_NO_PP`	The user processed the message.

▷ user_dg_init_window

```
void user_dg_init_window(void **user, window *win)
```

Description:

The user must perform whatever initialization is necessary for processing later commands. This usually includes setting up the user's data structure for receiving and storing display data.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	

Return values:

`ERROR`	Error. Ignored.
`USER_NO_PP`	The user successfully performed initialization.

▷ user_dg_free_window

```
void user_dg_free_window(void **user, window *win)
```

Description:

The user must free any data structures allocated.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	

Return values:

`ERROR`	Error. Ignored.
`USER_NO_PP`	The user successfully freed the data structures.

▷ user_interpret_text

```
void user_interpret_text(void *user, int text_length,
 char *text, int owner_tid)
```

Description:

The user can interpret text input from the window.

Arguments:

`void *user`	INOUT	Pointer to the user-defined data structure.
`int text_length`	IN	The length of `text`.
`char *text`	IN	
`int owner_tid`	IN	The tid of the process that initiated this window.

Return values:

`ERROR`	Error. Ignored.
`USER_NO_PP`	The user successfully interpreted the text.

10 SYMPHONY Parameters

The parameter file name is passed to SYMPHONY as the only command line argument to the master process which is started by the user. Each line of the parameter file contains either a comment or two words – a keyword and a value, separated by white space. If the first word (sequence of non-white-space characters) on a line is not a keyword, then the line is considered a comment line. Otherwise the parameter corresponding to the keyword is set to the listed value. Usually the keyword is the same as the parameter name in the source code. Here we list the keywords, the type of value that should be given with the keywords and the default value. A parameter corresponding to keyword "K" in process "P" can also be set by using the keyword "P_K".

To make this list shorter, occasionally a comma separated list of parameters is given if the meanings of those parameters are strongly connected. For clarity, the constant name is sometimes given instead of the numerical value for default settings and options. The corresponding value is given in curly braces for convenience.

10.1 Global parameters

`verbosity` – **integer (0).** Sets the verbosity of all processes to the given value. In general, the greater this number the more verbose each process is. Experiment to find out what this means.

`random_seed` – **integer (17).** A random seed.

`granularity` – **double (1e-6).** should be set to "the minimum difference between two distinct objective function values" less the epsilon tolerance. E.g., if every variable is integral and the objective coefficients are integral then for any feasible solution the objective value is integer, so `granularity` could be correctly set to .99999.

`upper_bound` – **double (none)** . The value of the best known upper bound.

10.2 Master Process parameters

`M_verbosity` – **integer (0).**

`M_random_seed` – **integer (17).** A random seed just for the Master Process.

`upper_bound` – **double (no upper bound).** This parameter is used if the user wants to artificially impose an upper bound (for instance if a solution of that value is already known).

`upper_bound_estimate` – **double (no estimate).** This parameter is used if the user wants to provide an estimate of the optimal value which will help guide the search. This is used in conjunction with the diving strategy `BEST_ESTIMATE`.

`tm_exe, dg_exe` – **strings ("tm", "dg").** `The name of the executable files of the TM and DG processes. Note that the TM executable name may have extensions that depend on the configuration of the modules, but the default is always set to the file name produced by the make file. If you change the name of the treemanager executable from the default, you must set this parameter to the new name.`

`tm_debug, dg_debug` – **boolean (both `FALSE`).** `Whether these processes should be started under a debugger or not (see 8.14.2 for more details on this).`

`tm_machine` – **string (empty string).** `On which processor of the virtual machine the TM should be run. Leaving this parameter as an empty string means arbitrary selection.`

`do_draw_graph` – **boolean (`FALSE`).** `Whether to start up the DG process or not (see Section 8.14.5 for an introduction to this).`

`do_branch_and_cut` – **boolean (`TRUE`).** `Whether to run the branch and cut algorithm or not. (Set this to FALSE to run the user's heuristics only.)`

10.3 Draw Graph parameters

`source_path` – **string (".").** The directory where the DG tcl/tk scripts reside.

`echo_commands` – **boolean (`FALSE`).** Whether to echo the tcl/tk commands on the screen or not.

`canvas_width, canvas_height` – **integers (1000, 700).** The default width and height of the drawing canvas in pixels.

`viewable_width, viewable_height` – **integers (600, 400).** The default viewable width and height of the drawing canvas in pixels.

`interactive_mode` – **integer (`TRUE`).** Whether it is allowable to change things interactively on the canvas or not.

`node_radius` – **integer (8).** The default radius of a displayed graph node.

`disp_nodelabels, disp_nodeweights, disp_edgeweights` – **integers (all `TRUE`).** Whether to display node labels, node weights, and edge weights or not.

`nodelabel_font, nodeweight_font, edgeweight_font` – **strings (all "-adobe-helvetica-...").** The default character font for displaying node labels, node weights and edge weights.

`node_dash, edge_dash` – **strings (both empty string).** The dash pattern of the circles drawn around dashed nodes and that of dashed edges.

10.4 Tree Manager parameters

`TM_verbosity` – **integer (0).** The verbosity of the TM process.

`lp_exe, cg_exe, cp_exe` – **strings ("lp", "cg", "cp").** The name of the LP, CG, and CP process binaries. Note: when running in parallel using PVM, these executables (or links to them) must reside in the `PVM_ROOT/bin/PVM_ARCH/` directory. Also, be sure to note that the executable names may have extensions that depend on the configuration of the modules, but the defaults will always be set to the name that the make file produce.

`lp_debug, cg_debug, cp_debug` – **boolean (all `FALSE`).** Whether the processes should be started under a debugger or not.

`max_active_nodes` – **integer (1).** The maximum number of active search tree nodes—equal to the number of LP and CG tandems to be started up.

`max_cp_num` – **integer (0).** The maximum number of cut pools to be used.

`lp_mach_num, cg_mach_num, cp_mach_num` – **integers (all 0).** The number of processors in the virtual machine to run LP (CG, CP) processes. If this value is 0 then the processes will be assigned to processors in round-robin order. Otherwise the next `xx_mach_num` lines describe the processors where the LP (CG, CP) processes must run. The keyword – value pairs on these lines must be **TM_xx_machine** and the name or IP address of a processor (the processor names need not be distinct). In this case the actual processes are assigned in a round robin fashion to the processors on this list.

This feature is useful if a specific software package is needed for some process, but that software is not licensed for every node of the virtual machine or if a certain process must run on a certain type of machine due to resource requirements.

`use_cg` – **boolean (`FALSE`).** Whether to use a cut generator or not.

`TM_random_seed` – **integer (17).** The random seed used in the TM.

`unconditional_dive_frac` – **double (0.1).** The fraction of the nodes on which SYMPHONY randomly dives unconditionally into one of the children.

`diving_strategy` – **integer (`BEST_ESTIMATE`{0}).** The strategy employed when deciding whether to dive or not.

The `BEST_ESTIMATE`{0} strategy continues to dive until the lower bound in the child to be dived into exceeds the parameter `upper_bound_estimate, which is given by the user.`

`The COMP_BEST_K{1} strategy computes the average lower bound on the best diving_k search tree nodes and decides to dive if the lower bound of the child to be dived into does not exceed this average by more than the fraction diving_threshold.`

`The COMP_BEST_K_GAP{2} strategy takes the size of the gap into account when` deciding whether to dive. After the average lower bound of the best diving_k nodes is computed, the gap between this average lower bound and the current upper bound is computed. Diving only occurs if the difference between the computed average lower bound and the lower bound of the child to be dived into is at most the fraction diving_threshold of the gap.

Note that fractional diving settings can override these strategies. See below.

diving_k, diving_threshold – **integer, double (1, 0.0).** See above.

fractional_diving_ratio, fractional_diving_num – **integer (0.02, 0).** Diving occurs automatically if the number of fractional variables in the child to be dived into is less than fractional_diving_num or the fraction of total variables that are fractional is less than fractional_diving_ratio. This overrides the other diving rules. Note that in order for this option to work, the code must be compiled with FRACTIONAL_BRANCHING defined. This is the default. See the Makefile for more details.

node_selection_rule – **integer (**LOWEST_LP_FIRST**{0}).** The rule for selecting the next search tree node to be processed. This rule selects the one with lowest lower bound. Other possible values are: HIGHEST_LP_FIRST{1}, BREADTH_FIRST_SEARCH{2} and DEPTH_FIRST_SEARCH{3}.

load_balance_level -- integer (-1).] A naive attempt at load balancing on problems where significant time is spent in the root node, contributing to a lack of parallel speed-up. Only a prescribed number of iterations (load_balance_iter) are performed in the root node (and in each subsequent node on a level less than or equal to load_balance_level) before branching is forced in order to provide additional subproblems for the idle processors to work on. This doesn't work well in general.

load_balance_iter -- integer (-1).] Works in tandem with the load_balance_level to attempt some simple load balancing. See the above description.

keep_description_of_pruned – **integer (**DISCARD**{0}).** Whether to keep the description of pruned search tree nodes or not. The reasons to do this are (1) if the user wants to write out a proof of optimality using the logging function, (2) for debugging, or (3) to get a visual picture of the tree using the software VBCTOOL. Otherwise, keeping the pruned nodes around just takes up memory.

There are three options if it is desired to keep some description of the pruned nodes around. First, their full description can be written out to disk and freed from memory (KEEP_ON_DISK_FULL{1}). There is not really too much you can do with this kind of file, but theoretically, it contains a full record of the solution process and could be used to provide a certificate of optimality (if we were using exact arithmetic) using an independent verifier. In this case, the line following keep_description_of_pruned should be a line containing the keyword pruned_node_file_name with its corresponding value being the name of a file to which a description of the pruned nodes can be written. The file does not need to exist and will be over-written if it does exist.

If you have the software VBCTOOL (see Section 8.15), then you can alternatively just write out the information VBCTOOL needs to display the tree (KEEP_ON_DISK_VBC_TOOL{2}).

Finally, the user can set the value to of this parameter to KEEP_IN_MEMORY{2}, in which case all pruned nodes will be kept in memory and written out to the regular log file if that option is chosen. This is really only useful for debugging. Otherwise, pruned nodes should be flushed.

logging – **integer (**NO_LOGGING**{0}).** Whether or not to write out the state of the search tree and all other necessary data to disk periodically in order to allow a warm start in the case of a system crash or to allow periodic viewing with VBCTOOL.

If the value of this parameter is set to FULL_LOGGING{1}, then all information needed to warm start the calculation will written out periodically. The next two lines of the parameter file following should contain the keywords tree_log_file_name and cut_log_file_name along with corresponding file names as values. These will be the files used to record the search tree and related data and the list of cuts needed to reconstruct the tree.

If the value of the parameter is set to VBC_TOOL{2}, then only the information VBCTOOL needs to display the tree will be logged. This is not really a very useful option since a ''live'' picture of the tree can be obtained using the vbc_emulation parameter described below (see Section 8.15 for more on this).

logging_interval – **integer (1800).** Interval (in seconds) between writing out the above log files.

warm_start – **boolean (0).** Used to allow the tree manager to make a warm start by reading in previously written log files. If this option is set, then the two line following must start with the keywords warm_start_tree_file_name and warm_start_cut_file_name and include the appropriate file names as the corresponding values.

vbc_emulation -- integer (NO_VBC_EMULATION{0}).] Determines whether or not to employ the VBCTOOL emulation mode. If one of these modes is chosen, then the tree will be displayed in ''real time'' using the VBCTOOL Software. When using the option VBC_EMULATION_LIVE{2} and piping the output directly to VBCTOOL, the tree will be displayed as it is constructed, with color coding indicating the status of each node. With VBC_EMULATION_FILE{1} selected, a log file will be produced which can later be read into VBCTOOL to produce an emulation of the solution process at any desired speed. If VBC_EMULATION_FILE is selected, the the following line should contain the keyword vbc_emulation_file_name along with the corresponding file name for a value.

price_in_root – **boolean** (FALSE). Whether to price out variables in the root node before the second phase starts (called *repricing the root*).

trim_search_tree – **boolean** (FALSE). Whether to trim the search tree before the second phase starts or not. Useful only if there are two phases. (It is very useful then.)

colgen_in_first_phase, colgen_in_second_phase – **integers (both 4).** These parameters determine if and when to do column generation in the first and second phase of the algorithm. The value of each parameter is obtained by setting the last four bits. The last two bits refer to what to do when attempting to prune a node. If neither of the last two bits are set, then we don't do anything---we just prune it. If only the last bit is set, then we simply save the node for the second phase without doing any column generation (yet). If only the second to last bit is set, then we do column generation immediately and resolve if any new columns are found. The next two higher bits determine whether or not to do column generation before branching. If only the third lowest bit is set, then no column generation occurs before branching. If only the fourth lowest bit is set, then column generation is attempted before branching. The default is not to generate columns before branching or fathoming, which corresponds to only the third lowest bit being set, resulting in a default value of 4.

time_limit – **integer (0).** Number of seconds of wall-clock time allowed for solution. When this time limit is reached, the solution process will stop and the best solution found to that point, along with other relevant data, will be output. A time limit of zero means there is no limit.

10.5 LP parameters

LP_verbosity – **integer (0).** Verbosity level of the LP process.

set_obj_upper_lim – **boolean** (FALSE). Whether to stop solving the LP relaxation when it's optimal value is provably higher than the global upper bound. There are some advantages to continuing the solution process anyway. For instance, this results in the highest possible lower bound. On the other hand, if the matrix is full, this node will be pruned anyway and the rest of the computation is pointless. This option should be set at FALSE for column generation since the LP dual values may not be reliable otherwise.

try_to_recover_from_error – **boolean** (TRUE). Indicates what should be done in case the LP solver is unable to solve a particular LP relaxation because of numerical problems. It is possible to recover from this situation but further results may be suspect. On the other hand, the entire solution process can be abandoned.

`problem_type` – **integer (`ZERO_ONE_PROBLEM`{0}).** The type of problem being solved. Other values are `INTEGER_PROBLEM`{1} or `MIXED_INTEGER_PROBLEM`{2}. (Caution: The mixed-integer option is not well tested.)

`cut_pool_check_frequency` – **integer (10).** The number of iterations between sending LP solutions to the cut pool to find violated cuts. It is not advisable to check the cut pool too frequently as the cut pool process can get bogged down and the LP solution generally do not change that drastically from one iteration to the next anyway.

`not_fixed_storage_size` – **integer (2048).** The *not fixed list* is a partial list of indices of variables not in the matrix that have not been fixed by reduced cost. Keeping this list allows SYMPHONY to avoid repricing variables (an expensive operation) that are not in the matrix because they have already been permanently fixed. When this array reaches its maximum size, no more variable indices can be stored. It is therefore advisable to keep the maximum size of this array as large as possible, given memory limitations.

`max_non_dual_feas_to_add_min`, `max_non_dual_feas_to_add_max`, `max_non_dual_feas_to_add_frac` – integer, integer, double (20, 200, .05). These three parameters determine the maximum number of non-dual-feasible columns that can be added in any one iteration after pricing. This maximum is set to the indicated fraction of the current number of active columns unless this numbers exceeds the given maximum or is less than the given minimum, in which case, it is set to the max or min, respectively.

`max_not_fixable_to_add_min`, `max_not_fixable_to_add_max`, `max_not_fixable_to_add_frac` – integer, integer, double (100, 500, .1). As above, these three parameters determine the maximum number of new columns to be added to the problem because they cannot be priced out. These variables are only added when trying to restore infeasibility and usually, this does not require many variables anyway.

`mat_col_compress_num`, `mat_col_compress_ratio` – **integer, double (50, .05).** Determines when the matrix should be physically compressed. This only happens when the number of columns is high enough to make it "worthwhile." The matrix is physically compressed when the number of deleted columns exceeds either an absolute number *and* a specified fraction of the current number of active columns.

`mat_row_compress_num`, `mat_row_compress_ratio` – **integer, double (20, .05).** Same as above except for rows.

`tailoff_gap_backsteps`, `tailoff_gap_frac` – **integer, double (2, .99).** Determines when tailoff is detected in the LP process. Tailoff is reported if the average ratio of the current gap to the previous iteration's gap over the last `tailoff_gap_backsteps` iterations wasn't at least `tailoff_gap_frac`.

`tailoff_obj_backsteps`, `tailoff_obj_frac` – **integer, double (2, .99).** Same as above, only the ratio is taken with respect to the change in objective function values instead of the change in the gap.

`ineff_cnt_to_delete` – **integer (0).** Determines after how many iterations of being deemed ineffective a constraint is removed from the current relaxation.

`eff_cnt_before_cutpool` – **integer (3).** Determines after how many iterations of being deemed effective each cut will be sent to the global pool.

`ineffective_constraints` – **integer (`BASIC_SLACKS_ARE_INEFFECTIVE`{2}).** Determines under what condition a constraint is deemed ineffective in the current relaxation. Other possible values are `NO_CONSTRAINT_IS_INEFFECTIVE`{0}, `NONZERO_SLACKS_ARE_INEFFECTIVE`{1}, and `ZERO_DUAL_VALUES_ARE_INEFFECTIVE`{3}.

`base_constraints_always_effective` – **boolean (`TRUE`).** Determines whether the base constraints can ever be removed from the relaxation. In some case, removing the base constraints from the problem can be disastrous depending on the assumptions made by the cut generator.

`branch_on_cuts` – **boolean (`FALSE`).** This informs the framework whether the user plans on branching on cuts or not. If so, there is additional bookkeeping to be done, such as maintaining a pool of slack cuts to be used for branching. Therefore, the user should not set this flag unless he actually plans on using this feature.

`discard_slack_cuts` – **integer (`DISCARD_SLACKS_BEFORE_NEW_ITERATION`{0}).**
Determines when the pool of slack cuts is discarded. The other option is `DISCARD_SLACKS_WHEN_STARTING_NEW_NODE`{1}.

`first_lp_first_cut_time_out, first_lp_all_cuts_time_out, later_lp_first_cut_time_out, later_lp_all_cuts_time_out` – double (0, 0, 5, 1). The next group of parameters determines when the LP should give up waiting for cuts from the cut generator and start to solve the relaxation in its current form or possibly branch if necessary. There are two factors that contribute to determining this timeout. First is whether this is the first LP in the search node of whether it is a later LP. Second is whether any cuts have been added already in this iteration. The four timeout parameters correspond to the four possible combinations of these two variables.

`no_cut_timeout` – This keyword does not have an associated value. If this keyword appears on a line by itself or with a value, this tells the framework not to time out while waiting for cuts. This is useful for debugging since it enables runs with a single LP process to be duplicated.

`all_cut_timeout` – **double (no default).** This keyword tells the framework to set all of the above timeout parameters to the value indicated.

`max_cut_num_per_iter` – **integer (20).** The maximum number of cuts that can be added to the LP in an iteration. The remaining cuts stay in the local pool to be added in subsequent iterations, if they are strong enough.

`do_reduced_cost_fixing` – **boolean (`FALSE`).** Whether or not to attempt to fix variables by reduced cost. This option is highly recommended

`gap_as_ub_frac, gap_as_last_gap_frac` – **double (.1, .7).** Determines when reduced cost fixing should be attempted. It is only done when the gap is within the fraction `gap_as_ub_frac` of the upper bound or when the gap has decreased by the fraction `gap_as_last_gap_frac` since the last time variables were fixed.

`do_logical_fixing` – **boolean (`FALSE`).** Determines whether the user's logical fixing routine should be used.

`fixed_to_ub_before_logical_fixing`, `fixed_to_ub_frac_before_logical_fixing` – **integer, double (1, .01)**. Determines when logical fixing should be attempted. It will be called only when a certain absolute number *and* a certain number of variables have been fixed to their upper bounds by reduced cost. This is because it is typically only after fixing variables to their upper bound that other variables can be logically fixed.

`max_presolve_iter` – **integer (10).** Number of simplex iterations to be performed in the presolve for strong branching.

`strong_branching_cand_num_max`, `strong_branching_cand_num_min`, `strong_branching_red_ratio` – **integer (25, 5, 1)**. These three parameters together determine the number of strong branching candidates to be used by default. In the root node, `strong_branching_cand_num_max` candidates are used. On each succeeding level, this number is reduced by the number `strong_branching_red_ratio` multiplied by the square of the level. This continues until the number of candidates is reduced to `strong_branching_cand_num_min` and then that number of candidates is used in all lower levels of the tree.

`is_feasible_default` – **integer (**`TEST_INTEGRALITY`**{1}).** Determines the default test to be used to determine feasibility. This parameter is provided so that the user can change the default behavior without recompiling. The only other option is `TEST_ZERO_ONE`{0}.

`send_feasible_solution_default` – **integer (**`SEND_NONZEROS`**{0}).** Determines the form in which to send the feasible solution. This parameter is provided so that the user can change the default behavior without recompiling. This is currently the only option.

`send_lp_solution_default` – **integer (**`SEND_NONZEROS`**{0}).** Determines the default form in which to send the LP solution to the cut generator and cut pool. This parameter is provided so that the user can change the default behavior without recompiling. The other option is `SEND_FRACTIONS`{1}.

`display_solution_default` – **integer (**`DISP_NOTHING`**{0}).** Determines how to display the current LP solution if desired. See the description of `user_display_solution() for other possible values. This parameter is provided so that the user can change the default behavior without recompiling.`

`shall_we_branch_default` – **integer (**`USER__BRANCH_IF_MUST`**{2}).** `Determines the default branching behavior. Other values are USER__DO_NOT_BRANCH{0} (not recommended as a default), USER__DO_BRANCH{1} (also not recommended as a default), and USER__BRANCH_IF_TAILOFF{3}. This parameter is provided so that the user can change the default behavior without recompiling.`

`select_candidates_default` – **integer (**`USER__CLOSE_TO_HALF_AND_EXPENSIVE`**{11}).** `Determines the default rule for selecting strong branching candidates. Other values are USER__CLOSE_TO_HALF{10} and USER__CLOSE_TO_ONE_AND_CHEAP{12}. This parameter is provided so that the user can change the default behavior without recompiling.`

`compare_candidates_default` – **integer (**`LOWEST_LOW_OBJ`**{1}).** `Determines the default rule for comparing candidates. See the description of` `user_compare_candidates() for other values. This parameter is provided so that the user can change the default behavior without recompiling.`

`select_child_default` – **integer (**`PREFER_LOWER_OBJ_VALUE`**{0}).** `Determines the default rule for selecting the child to be processed next. For other possible values, see the description user_select_child(). This parameter is provided so that the user can change the default behavior without recompiling.`

10.6 Cut Generator Parameters

`CG_verbosity` – **integer (0).** Verbosity level for the cut generator process.

10.7 Cut Pool Parameters

`CP_verbosity` – **integer (0).** Verbosity of the cut pool process.

`cp_logging` – **boolean (0).** Determines whether the logging option is enabled. In this case, the entire contents of the cut pool are written out periodically to disk (at the same interval as the tree manager log files are written). If this option is set, then the line following must start with the keyword `cp_log_file_name` and include the appropriate file name as the value.

`cp_warm_start` – **boolean (0).** Used to allow the cut pool to make a warm start by reading in a previously written log file. If this option is set, then the line following must start with the keyword `cp_warm_start_file_name` and include the appropriate file name as the value.

`block_size` – **integer (5000).** Indicates the size of the blocks to allocate when more space is needed in the cut list.

`max_size` – **integer (2000000).** Indicates the maximum size of the cut pool in bytes. This is the total memory taken up by the cut list, including all data structures and the array of pointers itself.

`max_number_of_cuts` – **integer (10000).** Indicates the maximum number of cuts allowed to be stored. When this max is reached, cuts are forceably purged, starting with duplicates and then those indicated by the parameter delete_which (see below), until the list is below the allowable size.

`min_to_delete` – **integer (1000).** Indicates the number of cuts required to be deleted when the pool reaches it's maximum size.

`touches_until_deletion` – **integer (10).** When using the number of touches a cut has as a measure of its quality, this parameter indicates the number of touches a cut can have before being deleted from the pool. The number of touches is the number of times in a row that a cut has been checked without being found to be violated. It is a measure of a cut's relevance or effectiveness.

`delete_which` – **integer (**`DELETE_BY_TOUCHES`**{2}).** Indicates which cuts to delete when purging the pool. `DELETE_BY_TOUCHES` indicates that cuts whose number of touches is above the threshold (see `touches_until_deletion` above) should be purged if the pool gets too large. `DELETE_BY_QUALITY`{1} indicates that a user-defined measure of quality should be used (see the function `user_check_cuts` in Section9.4).

`check_which` – **integer (`CHECK_ALL_CUTS`{0}).** Indicates which cuts should be checked for violation. The choices are to check all cuts (`CHECK_ALL_CUTS`{0}); only those that have number of touches below the threshold (`CHECK_TOUCHES`{2}); only those that were generated at a level higher in the tree than the current one (`CHECK_LEVEL`{1}); or both (`CHECK_LEVEL_AND_TOUCHES`{3}). Note that with `CHECK_ALL_CUTS` set, SYMPHONY will still only check the first `cuts_to_check cuts in the list ordered by quality (see the function user_check_cut).`

`cuts_to_check` – **integer (1000).** `Indicates how many cuts in the pool to actually check. The list is ordered by quality and the first cuts_to_check cuts are checked for violation.`

11 Bibliography

References

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

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *CONCORDE TSP Solver*, available at `www.keck.caam.rice.edu/concorde.html`.

[3] E. Balas, S. Ceria, and G. Cornuéjols, *Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework*, Management Science **42** (1996), 9.

[4] E. Balas and P. Toth, *Branch and Bound Methods*, in E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, eds., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, New York (1985), 361.

[5] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance, *Branch-and-Price: Column Generation for Huge Integer Programs*, Operations Research **46** (1998), 316.

[6] M. Benchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol, *Building a Parallel Branch and Bound Library*, in Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science **1054**, Springer, Berlin (1996), 201.

[7] Q. Chen and M.C. Ferris, *FATCOP: A Fault Tolerant Condor-PVM Mixed Integer Programming Solver*, University of Wisconsin CS Department Technical Report 99-05, Madison, WI (1999).

[8] *Common Optimization INterface for Operations Research*, `http://www.coin-or.org`.

[9] C. Cordier, H. Marchand, R. Laundy, and L.A. Wolsey, *bc-opt: A Branch-and-Cut Code for Mixed Integer Programs*, Mathematical Programming **86** (1999), 335.

[10] *ILOG CPLEX 6.5 Reference Manual*, ILOG (1994).

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

[12] M. Eső, *Parallel Branch and Cut for Set Partitioning*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY (1999).

[13] A. Geist et al., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA (1994).

[14] B. Gendron and T.G. Crainic, *Parallel Branch and Bound Algorithms: Survey and Synthesis*, Operations Research **42** (1994), 1042.

[15] M. Grötschel, M. Jünger, and G. Reinelt, *A Cutting Plane Algorithm for the Linear Ordering Problem*, Operations Research **32** (1984), 1155.

[16] A. Grama and V. Kumar, *Parallel Search Algorithms for Discrete Optimization Problems*, ORSA Journal on Computing **7** (1995), 365.

[17] K. Hoffman and M. Padberg, *LP-Based Combinatorial Problem Solving*, Annals of Operations Research **4** (1985/86), 145.

[18] M. Jünger and S. Thienel, *The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization*, Software Practice and Experience **30** (2000), 1325.

[19] M. Jünger and S. Thienel, *Introduction to ABACUS—a branch-and-cut system*, Operations Research Letters **22** (1998), 83.

[20] V. Kumar and V.N. Rao, *Parallel Depth-first Search. Part II. Analysis.*, International Journal of Parallel Programming **16** (1987), 501.

[21] L. Ladányi, T.K. Ralphs, and L.E. Trotter, *Branch, Cut, and Price: Sequential and Parallel*, Computational Combinatorial Optimization, D. Naddef and M. Jünger, eds., Springer, Berlin (2001), 223.

[22] L. Ladányi, T.K. Ralphs, and M.J. Saltzman, *A Library Hierarchy for Implementing Scalable Parallel Search Algorithms*, available at `http://www.lehigh.edu/~tkr2/pubs.html`

[23] J. Linderoth, *Topics in Parallel Integer Optimization*, Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1998).

[24] A. Martin, *Integer Programs with Block Structure*, Habilitation Thesis, Technical University of Berlin, Berlin, Germany (1998).

[25] G.L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi, *MINTO, a Mixed INTeger Optimizer*, Operations Research Letters **15** (1994), 47.

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

[27] M. Padberg and G. Rinaldi, *A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems*, SIAM Review **33** (1991), 60.

[28] T.K. Ralphs, *Parallel Branch and Cut for Vehicle Routing*, Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY (1995).

[29] T.K. Ralphs and L. Ladányi, *Computational Experience with Branch, Cut, and Price: Sequential and Parallel*, in preparation.

[30] T.K. Ralphs and L. Ladányi, *SYMPHONY: A Parallel Framework for Branch and Cut*, White paper, Rice University (1999).

[31] V.N. Rao and V. Kumar, *Parallel Depth-first Search. Part I. Implementation.*, International Journal of Parallel Programming **16** (1987), 479.

[32] Y. Shinano, M. Higaki, and R. Hirabayashi, *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 (1995), 392.

[33] Y. Shinano, K. Harada, and R. Hirabayashi, *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 (1997), 621.

[34] S. Tschöke and T. Polzer, *Portable Parallel Branch and Bound Library User Manual, Library Version 2.0*, Department of Computer Science, University of Paderborn.