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

SYMPHONY Developed By

T.K. Ralphs[†]
L. Ladányi[‡]

Interactive Graph Drawing
Software By

M. Esö[§]

September 8, 2000

[*]This research was partially supported by Texas ATP Grant 97-3604-010
[†]Department of Industrial and Manufacturing Systems Engineering, Lehigh University, Bethlehem, PA 18015
[‡]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

Contents

1 Introduction

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 manual is concerned with the detailed specifications needed to develop an application using SYMPHONY. It is assumed that the user has already read the white paper *SYMPHONY: A Parallel Framework for Branch, Cut, and Price*, which provides a high-level introduction to parallel branch, cut, and price and the overall design and use of SYMPHONY. Reading and understanding of the white paper should be undertaken before trying to develop an application.

1.1 New in Version 2.8

If you are new to SYMPHONY, you can skip to Section 1.3. 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 3.4).
- **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 2.4).
- **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 3.5).

1.2 Changes to the User Interface

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()` (Section2.1) 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 3.4) 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()` (Section2.4) 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()` (Section2.2) 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.

1.3 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 1.8 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 1.12 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 1.12 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.

1.4 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 interprocess 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.

1.5 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.

1.6 Data Structures

1.6.1 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.

1.6.2 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.

1.7 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.

1.8 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.

1.9 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`.

1.10 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.

1.11 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.

1.12 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 1.9.

1.13 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 3.4 for information on setting parameters in the parameter file.

1.14 Debugging Your Application

1.14.1 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.

1.14.2 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 1.16).

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.

1.14.3 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 1.13, 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 3.4 for information on setting parameters in the parameter file.

1.14.4 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 2.3). 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 2.1). 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.

1.14.5 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.

1.14.6 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 3.4 for the appropriate parameter settings to use logging.

1.15 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 3.

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 3.4 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.

1.16 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`.

2 User Written Functions

2.1 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 3.4))

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 2.2 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 1.7. 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 2.3 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 1.7. 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 2.4 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 1.7. 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 1.14.4 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.`

2.2 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 2.1). 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 1.7 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 1.7 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 2.2).
`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 2.2).
`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 3.5).

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 $\mathtt{i}^{\mathtt{th}}$ 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.

2.3 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 2.1) 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 2.2 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 1.14.4 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.

2.4 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 2.1) 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 2.2 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.

2.5 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.

3 Parameter file

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.

3.1 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.

3.2 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 1.14.2 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 1.14.5 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.)

3.3 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.

3.4 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 1.15), 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 1.15 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.

3.5 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.`

3.6 Cut Generator Parameters

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

3.7 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 Section2.4).

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.