The SYMPHONY Callable Library For Mixed Integer Programming

T. K. Ralphs
Menal Guzelsoy
Lehigh University

Report No. 04T-011

The SYMPHONY Callable Library for Mixed Integer Programming

Ted Ralphs[*] Menal Guzelsoy[†]

May 13, 2004

Abstract

SYMPHONY is a customizable, open-source library for solving mixed-integer linear programs (MILP) by branch, cut, and price. With its large assortment of parameter settings, user callback functions, and compile-time options, SYMPHONY can be configured as a generic MILP solver or an engine for solving difficult MILPs by means of a fully customized branch and cut algorithm. SYMPHONY can also be configured for a wide variety of architectures, including single-processor, distributed-memory parallel, and shared-memory parallel architectures under MS Windows, Linux, and other Unix operating systems. SYMPHONY 5.0 is implemented as a callable library that can be accessed either through calls to the native C subroutines of the application program interface or through a C++ class derived from the COIN-OR Open Solver Interface. Among its new features are the ability to solve bicriteria MILPs, the ability to stop and warm start MILP computations after modifying parameters or problem data, the ability to create persistent cut pools, and the ability to perform basic sensitivity analysis on MILPs.

1 Introduction

As recently as a decade ago, the software available for solving generic mixed-integer linear programs (MILPs) was relatively limited. In the last 10 years, that has changed dramatically, as the market has opened significantly. There are now more than a dozen solvers available, many of which are open source. Among the academic and research codes available for solving generic MILPs are MINTO [20], MIPO [1], bc-opt [7], SBB [13], bonsaiG [15], PARINO [18] and FATCOP [4, 5]. Commercial offerings include ILOG's CPLEX, IBM's OSL (soon to be discontinued), and Dash's XPRESS. In addition, there are a number of other customizable frameworks, including COIN/BCP [28], ABACUS [16, 17], ALPS [24], and PICO [8].

SYMPHONY is a callable library for solving MILPs that was originally developed as a framework for implementing custom solvers for hard combinatorial problems. It has recently been integrated with the Computational Infrastructure for Operations Research (COIN-OR) libraries [13] and outfitted as a generic MILP solver. SYMPHONY's core solution methodology is a branch, cut, and price algorithm that incorporates most of the advanced solution management features available in other codes. Its main shortcoming is that it lacks both an integer presolver and a primal heuristic, which hurts its performance as a generic MILP code. However, it is otherwise very competitive, especially as a parallel solver. SYMPHONY depends on several other open source libraries for functionality, including the Cut Generation Library maintained by the COIN-OR project for cut generation, COIN-OR's MPS file parser, COIN-OR's Open Solver interface for accessing LP solvers, and GLPK's parser for GMPL files (GMPL is a subset of AMPL) [19].

[*]Dept. of Industrial and Systems Engineering, Lehigh University, Bethlehem PA, `tkr2@lehigh.edu`
[†]Dept. of Industrial and Systems Engineering, Lehigh University, Bethlehem PA, `megb@lehigh.edu`

2 The Application Program Interface

SYMPHONY 5.0 is the first version of SYMPHONY to be implemented as a callable library with a new interface derived from the COIN-OR Open Solver Interface. This change markedly improves SYMPHONY's usability and flexibility. SYMPHONY and solvers built using SYMPHONY have been the subject of a number of papers, most recently [23], [22], and [25]. SYMPHONY's legacy features are well-detailed in the SYMPHONY 4.0 User's Manual [21], so we focus here on new features, such as the application program interface (API), the bicriteria solver, the ability to warm start MILP computations, and the ability to perform sensitivity analysis. To our knowledge, these features are not available in any other MILP code and should of interest to users of the technology. Below, we briefly describe the new API, the C++ interface, and the use of the user callback functions.

2.1 The Callable Library

SYMPHONY's callable library consists of a complete set of subroutines for loading and modifying problem data, setting parameters, and invoking solution algorithms. The user invokes these subroutines through the API specified in the header file `sym_api.h`. Some of the basic commands are described below. For the sake of brevity, the arguments have been left out.

`sym_open_environment()` Opens a new environment, and returns a pointer to it. This pointer then has to be passed as an argument to all other API subroutines (in the C++ interface, this pointer is maintained for the user).

`sym_parse_command_line()` Invokes the built-in parser for setting commonly used parameters, such as the file name which to read the problem data, via command-line switches. A call to this subroutine instructs SYMPHONY to parse the command line and set the appropriate parameters. This subroutine also sets all other parameter values to their defaults, so it should only called when this is desired.

`sym_load_problem()` Reads the problem data and sets up the root subproblem. This includes specifying which cuts and variables are in the *core* (those that are initially present in every subproblem during the search process) and the additional cuts and variables to be initially active in the root subproblem. By default, SYMPHONY reads an MPS or GMPL file specified by the user, but the user can override this default by implementing a user callback that reads the data from a file in a customized format (see Section 2.3).

`sym_find_initial_bounds()` Invokes the user callback to find initial bounds using a custom heuristic.

`sym_solve()` Solves the currently loaded problem from scratch. This method is described in more detail in Section 3.1.

`sym_resolve()` Solves the currently loaded problem from a warm start. This method is described in more detail in Section 3.2.

```
int main(int argc, char **argv)
{
   sym_environment *p = sym_open_environment();
   sym_parse_command_line(p, argc, argv);
   sym_load_problem(p);
   sym_solve(p);
   sym_close_environment(p);
}
```

Figure 1: Implementation of a generic MILP solver with the SYMPHONY C callable library.

`sym_mc_solve()` Solves the currently loaded problem as a multicriteria problem. This method is described in more detail in Section 3.3.

`sym_close_environment()` Frees all problem data and deletes the environment.

As an example of the use of the library functions, Figure 1 shows the code for implementing a generic MILP solver with default parameter settings. To read in an MPS file called `model.mps` and solve it using this program, the following command would be issued:

```
symphony -F model.mps
```

The user does not have to invoke a command to read the MPS file. During the call to `sym_parse_command_line()`, SYMPHONY determines that the user wants to read in an MPS file. During the subsequent call to `sym_load_problem()`, the file is read and the problem data stored. To read an GMPL file, the user would issue the command

```
symphony -F model.mod -D model.dat
```

Although the same command-line switch is used to specify the model file, the additional presence of the `-D` option indicates to SYMPHONY that the model file is in GMPL format and GLPK's GMPL parser is invoked [19]. Note that the interface and the code of Figure 1 is the same for both sequential and parallel computations. The choice between sequential and parallel execution modes is made at compile-time through modification of the makefile or the project settings, depending on the operating system.

In addition to the parts of the API we have just described, there are a number of standard subroutines for accessing and modifying problem data and parameters. These can be used between calls to the solver to change the behavior of the algorithm or to modify the instance being solved. These modifications are discussed in more detail in Section 3.2.

2.2 The OSI Interface

The Open Solver Interface (OSI) is a C++ class that provides a standard API for accessing a variety of solvers for mathematical programs. It is provided as part of the COIN-OR repository [13], along with a collection of solver-specific derived classes that translate OSI call into calls to the underlying libraries of the solvers. A code implemented using calls to the methods in the OSI base class can easily be linked with any solver for which there is an OSI interface. This allows development

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.branchAndBound();
}
```

Figure 2: Implementation of a generic MILP solver with the SYMPHONY OSI interface.

of solver-independent codes and eliminates many portability issues. The current incarnation of OSI supports only solvers for linear and mixed-integer linear programs, although a new version supporting a wider variety of solvers is currently under development.

We have implemented an OSI interface for SYMPHONY 5.0 that allows any solver built with SYMPHONY to be accessed through the OSI, including customized solvers and those configured to run on parallel architectures. To ease code maintenance, for each method in the OSI base class, there is a corresponding method in the callable library. The OSI methods are implemented simply as wrapped calls to the SYMPHONY callable library. When an instance of the OSI interface class is constructed, a call is made to `sym_open_environment()` and a pointer to the environment is stored in the class. Most subsequent calls within the class can then be made without any arguments. When the OSI object is destroyed, `sym_close_environment` is called and the environment is destroyed.

To fully support SYMPHONY's capabilities, we have extended the OSI interface to include some methods not in the base class. For example, we added calls equivalent to our `sym_parse_command_line()` and `sym_find_initial_bounds()`. Figure 2 shows the program of Figure 1 implemented using the OSI interface. Note that the code would be exactly the same for accessing any customized SYMPHONY solver, sequential or parallel.

Although we are using the OSI to access a MILP solver, the current version of the OSI is geared primarily toward support of solvers for linear programming (LP) problems. This is because LP solvers employing some version of the simplex algorithm support much richer functionality and a wider range of interface functions, due to their support of warm starting from previously saved checkpoints. This functionality is difficult to provide for MILP solvers. In SYMPHONY 5.0, we have implemented for MILPs some of the same functionality that has long been available for LP solvers. As such, our OSI interface supports warm starting and sensitivity analysis. The implementations of this functionality is straightforward at the moment, but will be improved in the future, as discussed in Sections 3 and 4.

2.3 User Callback Functions

The user's main avenues for customization of SYMPHONY are the tuning of parameters and the implementation of one or more of over 50 user callback functions. The callback functions allow the user to override SYMPHONY's default behavior for many of the functions performed as part of its algorithm. The user has complete control over branching, cutting plane generation, management of the cut pool and the LP relaxation, search and diving strategies, and limited column generation. The callback functions are grouped by module according to their functionality. The names of the callback functions begin with the prefix `user_`. For instance, the `user_find_cuts()` subroutine is used to implement subroutines for finding problem-specific cutting planes and is part of the cut generation module. A full list of callbacks is contained in the SYMPHONY user's manual [21].

Callbacks in SYMPHONY are implemented slightly differently than in other popular libraries. Each user function is called from a SYMPHONY *wrapper function* that interprets the user's return value and determines what action should be taken. If the user performs the required function, the wrapper function normally exits without further action. If the user requests that SYMPHONY perform a certain default action, then this is done. Files containing default function stubs for all callbacks are provided along with the SYMPHONY source code and must be compiled and linked with SYMPHONY's internal library functions to obtain an executable. Makefiles and Microsoft Visual C++ project files are provided for automatic compilation.

3 Implementation

Because SYMPHONY is designed to allow parallel execution, both the internal library and the set of user callback functions are divided along functional lines into five separate modules. This modularization makes the parallel implementation more transparent and eases code maintenance. The five modules are the *master module*, the *tree manager module*, the *cut generation module*, the *cut pool module*, and the *node processing module*. Only the master module is persistent and the environment pointer described earlier is a pointer to the master module. All other modules are transient and exist only while a solve call is active. The master module's primary functions are

- initialization of the environment,
- setting and maintaining of parameter values,
- I/O,
- storage of static problem data,
- storage of warm start information between calls to the solver,
- distribution of data to other modules, and
- tracking of the status of associated processes during parallel execution.

Other modules encapsulate the specific functionality needed to execute the algorithms described below and can function as independent remote processes for parallel execution. A more complete description of the modular design of SYMPHONY can be found in the user's manual [21] or in [23].

For linear programming problems, the OSI has two function calls for solving the loaded model, `initialSolve()` and `resolve()`. The first call is used when solving a problem from scratch and the second is used when resolving after having modified the problem in some way. SYMPHONY's OSI implementation extends this idea to MILPs. We have also implemented a third solve call `multiCriteriaBranchAndBound()` for solving bicriteria MILPs. In the next few sections, we describe some of the details of how these methods are implemented.

3.1 Initial Solve

Calling the initial solve method solves a given MILP from scratch, as described above. The first action taken is to create an instance of the tree manager module that will control execution of the algorithm. If the algorithm is to be executed in parallel on a distributed architecture, the master module spawns a separate tree manager process that will autonomously control the solution process. The tree manager in turn creates the modules for processing the nodes of the search tree, generating cuts, and maintaining cut pools. These modules work in concert to execute the solution process. When it makes sense, sets of two or more modules, such as a node processing module and a cut generation module may be combined to yield a single process in which the combined modules work in concert and communicate with each other through shared memory instead of across the network. When running as separate process, the modules communicate with each other using a standard communications protocol. Currently, the only option supported is PVM, but it would be relatively easy to add an MPI implementation.

The overall flow of the algorithm is similar to other branch and bound implementations and has been described in detail in [23]. A priority queue of candidate subproblems available for processing is maintained at all times and the candidates are processed in an order determined by the search strategy. The algorithm terminates when the queue is empty or when another specified condition is satisfied. A new feature in SYMPHONY 5.0 is the ability to stop the computation based on exceeding a given time limit, exceeding a given limit on the number of processed nodes, achieving a target percentage gap between the upper and lower bounds, or finding the first feasible solution. After halting prematurely, the computation can be restarted after modifying parameters or problem data. This enables the implementation of a wide range of dynamic and on-line solution algorithms, as we describe next.

3.2 Solve from Warm Start

Among the utility classes in the COIN-OR repository is a base class for describing the data needed to warm start the solution process for a particular solver or class of solvers. To support this option for SYMPHONY, we have implemented such a warm start class for MILPs. The main content of the class is a compact description of the search tree at the time the computation was halted. This description contains complete information about the subproblem corresponding to each node in the search tree, including the branching decisions that lead to the creation of the node, the list of active variables and constraints, and warm start information for the subproblem itself (which is a linear program). All information is stored compactly using SYMPHONY's native data structures, which store only the differences between a child and its parent, rather than an explicit description of every node. This approach reduces the tree's description to a fraction of the size it would otherwise be. In addition to the tree itself, other relevant information regarding the status of the computation is recorded, such as the current bounds and best feasible solution found so far. Using the warm start class, the user can save a warm start to disk, read one from disk, or restart the computation at any point after modifying parameters or the problem data itself. This allows the user to easily implement periodic checkpointing, to design dynamic algorithms in which the parameters are modified after the gap reaches a certain threshold, or to modify problem data during the solution process if needed.

3.2.1 Modifying Parameters

The most straightforward use of the warm start class is to restart the solver after modifying problem parameters. The master module automatically records the warm start information resulting from the last solve call and restarts from that checkpoint if a call to `resolve()` is made, unless external warm start information is loaded manually. To start the computation from a given warm start when the problem data has not been modified, the tree manager simply traverses the tree and adds those nodes marked as candidates for processing to the node queue. Once the queue has been reformed, the algorithm is then able to pick up exactly where it left off. Figure 3 illustrates this concept by showing the code for implementing a solver that changes from depth first search to best first search after the first feasible solution is found. The situation is more challenging if the user

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.setSymParam(OsiSymFindFirstFeasible, true);
   si.setSymParam(OsiSymSearchStrategy, DEPTH_FIRST_SEARCH);
   si.initialSolve();
   si.setSymParam(OsiSymFindFirstFeasible, false);
   si.setSymParam(OsiSymSearchStrategy, BEST_FIRST_SEARCH);
   si.resolve();
}
```

Figure 3: Implementation of a dynamic MILP solver with SYMPHONY.

modifies problem data in between calls to the solver. We address this situation next.

3.2.2 Modifying Problem Data

If the user modifies problem data in between calls to the solver, SYMPHONY must make corresponding modifications to the leaf nodes of the current search tree to allow execution of the algorithm to continue. In principle, any change to the original data that does not invalidate the subproblem warm start data, i.e., the basis information for the LP relaxation, can be accommodated. Currently, SYMPHONY can only handle modifications to the rim vectors of the original MILP. Methods for handling other modifications, such as the addition of columns or the modification of the constraint matrix itself, will be added in the future. To initialize the algorithm, each leaf node, regardless of its status after termination of the previous solve call, must be inserted into the queue of candidate nodes and reprocessed with the changed rim vectors. After this reprocessing, the computation can continue as usual. Optionally, the user can "trim the tree" before resolving. This consists of locating nodes whose descendants are all likely to be pruned in the resolve and eliminating those descendants in favor of processing the parent node itself. This ability could be extended to allow changes that invalidate the warm start data of some leaf nodes.

The ability to resolve after modifying problem data has a wide range of applications in practice. One obvious use is to allow dynamic modification of problem data during the solve procedure, or even after the procedure has been completed. Implementing such a solver is simply a matter of periodically stopping to check for user input describing a change to the problem. Another obvious application is in situations where it is known a priori that the user will be solving a sequence of very similar MILPs. This occurs, for instance, when implementing algorithms for multicriteria optimization, as we describe in Section 3.3. One approach to this is to solve a given "base problem" (possibly limiting the size of the warm start tree), save the warm start information from the base problem and then start each subsequent call from this same checkpoint. Code for implementing this is shown in Figure 4. In this example, the solver is allowed to process 100 nodes and then save the warm start information. Afterward, the original problem is solved to optimality, then is modified and resolved from the saved checkpoint. Results from a sample run of this code are discussed in Section 5.

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   CoinWarmStart* ws;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.setSymParam(OsiSymNodeLimit, 100);
   si.initialSolve();
   ws = si.getWarmStart();
   si.resolve();
   si.setObjCoeff(0, 100);
   si.setObjCoeff(200, 150);
   si.setWarmStart(ws);
   si.setSymParam(OsiSymNodeLimit, -1);
   si.resolve();
}
```

Figure 4: Use of SYMPHONY's warm start capability.

3.3 Bicriteria Solve

For those readers not familiar with bicriteria integer programming, we briefly review the basic notions here. For clarity, we restrict the discussion here to pure integer programs (ILPs), but the principles are easily generalized. A bicriteria ILP is a generalization of a standard ILP presented earlier that includes a second objective function, yielding an optimization problem of the form

$$\begin{aligned} &\text{vmin } [cx, dx], \\ &\text{s.t.} \qquad Ax \le b, \\ &\qquad\qquad x \in \mathbb{Z}^n. \end{aligned} \tag{2}$$

The operator *vmin* is understood to mean that solving this program is the problem of generating *efficient* solutions, which are these feasible solutions p to (2) for which there does not exist a second distinct feasible solution q such that $cq \le cp$ and $dq \le dp$ and at least one inequality is strict. Note that (2) does not have a unique optimal solution value, but a set of pairs of solution values called *outcomes*. The pairs of solution values corresponding to efficient solutions are called *Pareto outcomes*. Surveys of methodology for for enumerating the Pareto outcomes of multicriteria integer programs are provided by Climaco et al. [6] and more recently by Ehrgott and Gandibleux [9, 10] and Ehrgott and Wiecek [11].

The bicriteria ILP (2) can be converted to a standard ILP by taking a nonnegative linear combination of the objective functions [14]. Without loss of generality, the weights can be scaled so they sum to one, resulting in a family of ILPs parameterized by a scalar $0 \le \alpha \le 1$, with the bicriteria objective function replaced by the *weighted sum objective*

$$(\alpha c + (1 - \alpha)d)x. \tag{3}$$

Each selection of weight α produces a different single-objective problem. Solving the resulting ILP produces a Pareto outcome called a *supported outcome*, since it is an extreme point on the convex lower envelope of the set of Pareto outcomes. Unfortunately, not all efficient outcomes are supported, so it is not possible to enumerate the set of Pareto outcomes by solving a sequence of ILPs from this parameterized family. To obtain all Pareto outcomes, one must replace the weighted sum objective (3) with an objective based on the *weighted Chebyshev norm* studied by Eswaran et al. [12] and Solanki [27]. If x^c is a solution to a weighted sum problem with $\alpha = 1$ and x^d is the solution with $\alpha = 0$, then the weighted Chebyshev norm of a feasible solution p is

$$\max\{\alpha(cp - cx^c), (1 - \alpha)(dp - dx^d)\}. \tag{4}$$

Although this objective function is not linear, it can easily be linearized by adding an artificial variable, resulting in a second parameterized family of ILPs. Under the assumption of *uniform dominance*, Bowman showed that an outcome is Pareto if and only if it can be obtained by solving some ILP in this family [3]. In [25], the authors presented a method for enumerating all Pareto outcomes by solving a sequence of ILPs in this parameterized family. By slightly perturbing the objective function, they also showed how to relax the uniform dominance assumption. Note that the set of all supported outcomes, which can be thought of as an approximation of the set of Pareto outcomes, can be similarly obtained by solving a sequence of ILPs with weighted sum objectives.

SYMPHONY 5.0 contains a generic implementation of the algorithm described in [25], along with a number of methods for approximating the set of Pareto outcomes. To support these capabilities, we have extended the OSI interface so that it allows the user to define a second objective function. Of course, we have also added a method for invoking this bicriteria solver called `multiCriteriaBranchAndBound()`. Relaxing the uniform dominance requirement requires the underlying ILP solver to have the ability to generate, among all optimal solutions to a ILP with a primary objective, a solution minimizing a given secondary objective. We added this capability to SYMPHONY through the use of optimality cuts, as described in [25].

Because implementing the algorithm requires the solution of a sequence of ILPs that vary only in their objective functions, it is possible to use warm starting to our advantage. Although the linearization of (4) requires modifying the constraint matrix from iteration to iteration, it is easy to show that these modifications cannot invalidate the basis. In the case of enumerating all supported outcomes, only the objective function is modified from one iteration to the next. In both cases, we save warm start information from the solution of the first ILP in the sequence and use it for each subsequent computation.

The applications of the bicriteria solver are numerous. Besides yielding the ability to closely examine the tradeoffs between competing objectives, the method can be used to perform detailed sensitivity analysis in a manner analogous to that which can be done with simplex based solvers for linear programs. As an example, suppose we would like to know exactly how the optimal objective function value for a given pure integer program depends on the value of a given objective function coefficient (see [2] for a discussion of this in the case of linear programming models). Consider increasing the objective function coefficient of variable i from its current value. Taking the first objective function to be the original one and taking the second objective function to be the i^{th} unit vector, we can derive the desired sensitivity function by using the bicriteria solution algorithm to enumerate all supported solutions and breakpoints. This information can easily be used to obtain the desired function. Figure 5 shows the code for performing this analysis on variable 0. For an example of the application of this code, see Section 5.

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.setObj2Coeff(0, 1);
   si.multiCriteriaBranchAndBound();
}
```

Figure 5: Performing sensitivity analysis with SYMPHONY's bicriteria solver.

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.branchAndBound;
   int cnt = 2;
   int * ind = new int[cnt];
   double * val = new double[cnt];
   ind[0] = 4;   val[0] = 7000;
   ind[1] = 7;   val[1] = 6000;
   lb = si.getLbForNewRhs(cnt, ind, val);
}
```

Figure 6: Performing sensitivity analysis with SYMPHONY

4 Other New Features

4.1 Sensitivity Analysis

In addition to the sensitivity analysis that can be undertaken by using SYMPHONY's bicriteria solver, we have also implemented the method suggested by Schrage and Wolsey in [26] for performing approximate sensitivity analysis on the right hand side vector. The method of Schrage and Wolsey is based on constructing an approximate dual price function from the dual solutions obtained while solving the LP relaxations in each search tree node. Figure 6 shows an example of a program that uses this sensitivity analysis function. This code will give a lower bound for a modified problem with new right hand side values of 7000 and 6000 in the 4th and 7th rows. The price function does not have a simple closed form, and must be computed for each change in the right hand side. This price function can be used to obtain approximate sensitivity information quickly when there is not enough time for a complete resolve. We discuss this example and present some sample computational results with this method in Section 5.

	CPU Time	Search Tree Nodes
Generate warm start	28	100
Solve original problem (from warm start)	3	118
Solve modified problem (from scratch)	24	122
Solve modified problem (from warm start)	6	198

Table 1: Warm starting a computation with p0201

4.2 Persistent Cut Pools

To complement the ability to checkpoint the search tree, the user can also checkpoint and save the global cut pool. When checkpointing the search tree, only the cuts that are currently active in some leaf node and are needed to restart the search process are saved. At times, however, it may be advantageous to save the entire global cut pool, including cuts that were generated, but are not currently active. If this is desirable, the user can direct SYMPHONY to maintain one or more persistent cut pools. Such pools exist as part of the master module and are attached to the tree manager whenever a solve call is made.

5 Examples

We now present some brief examples of applying the capabilities of the new library. These are not meant to be indicative of the solver's performance—computational results will be presented in a full-length companion paper to follow this abstract. Rather, we aim here simply to present some a few numerical examples that illustrate the concepts.

To illustrate the use of the warm start and resolving capabilities of the library, we refer to the code of Figure 4 described earlier. We ran this simple code on the MIPLIB 3 example `p0201`. The results are shown in Table 1. For this simple example, the warm start does its job. Interestingly, the number of search tree nodes generated is actually increased over solving the problem from scratch, but because most of the work takes place in cut generation near the top of the tree, the resolve is much quicker than the initial solve.

As an illustration of the use of the bicriteria solver, we applied the code of Figure 5 to the following simple MILP.

$$\begin{aligned}
\max\ & 8x_1 + \theta x_2, \\
\text{s.t.}\quad & 7x_1 + x_2 \le 56, \\
& 28x_1 + 9x_2 \le 252, \\
& 3x_1 + 7x_2 \le 105, and \\
& x_1, x_2 \ge 0,\ \text{integral.}
\end{aligned}$$

Applying the bicriteria solver of Figure 5 results in the price function $p(\theta)$ shown in Table 2. Extensive computational results obtained applying the solver to a class of network routing problems can be found in [25].

Finally, we illustrate the use of the approximate sensitivity analysis function illustrated in Figure 6. We tested the code shown there with the instance `flugpl` from MIPLIB 3, varying the right hand side values of the 4th and 7th rows. The results are shown in Table 3. In this table, for each pair of right hand side values, the smaller of the two numbers (shown above) is the lower

θ range	$p(\theta)$	x_1^*	x_2^*
$(-\infty, 1.333)$	64	8	0
$(1.333, 2.667)$	$56 + 6\theta$	7	6
$(2.667, 8.000)$	$40 + 12\theta$	5	12
$(8.000, 16.000)$	$32 + 13\theta$	4	13
$(16.000, \infty)$	15θ	0	15

Table 2: Price function for a simple MILP

bound obtained by SYMPHONY, whereas the larger number is the optimal solution value for the problem with the given right hand side values.

6 Conclusions

We have described the main features of the SYMPHONY 5.0 callable library. SYMPHONY includes implementations of a number of techniques useful for performing sensitivity analysis, resolving MILPs from a warm start, and analyzing bicriteria MILPs. To our knowledge, these techniques are not available in any other solver. These capabilities are still being refined and new techniques developed, so they will undoubtedly be improved in future versions of the library. This is an area of active research that we believe has a great deal of potential and has received relatively little attention in the literature. It remains to be seen how well these methods will work in practice. In future work, we plan to extend and generalize the methods presented here to allow greater flexibility on the type of problem modifications and sensitivity analyses that can be performed and to further improve the power of the bicriteria solver.

Acknowledgments. This research was partially supported through NSF grant ACI-0102687 and the IBM Faculty Partnership Program.

References

[1] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229–1246, 1996.

[2] D. Bertsimas and J.N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, Belmont, MA, USA, 1997.

[3] V. J. Bowman. On the relationship of the Tchebycheff norm and the efficient frontier of multiple-criteria objectives. In H. Thieriez, editor, *Multiple Criteria Decision Making*, pages 248–258. Springer, Berlin, 1976.

[4] Q. Chen and M. C. Ferris. FATCOP: A fault tolerant condor-pvm mixed integer program solver. Technical Report Mathematical Programming Technical Report 99-05, Computer Sciences Department, University of Wisconsin at Madison, 1999. Submitted.

[5] Q. Chen, M. C. Ferris, and J. T. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. Technical Report Data Mining Institute Technical Report 99-11, Computer Sciences Department, University of Wisconsin at Madison, 1999.

	6000	6500	7000	7500	8000	8500	9000	9500	10000
7000	1109789.89	1118201.35	1126612.81	1135024.27	1146369.28	1151847.18	1160258.64	1168670.10	1177081.56
	1165500.00	1165500.00	1165500.00	1168500.00	1183500.00	1194000.00	1194000.00	1194000.00	1200000.00
7500	1115727.39	1124138.85	1132550.31	1140961.77	1154383.33	1157784.68	1166196.14	1174607.60	1183019.06
	1165500.00	1165500.00	1165500.00	1168500.00	1183500.00	1197000.00	1197000.00	1197000.00	1203000.00
8000	1121664.89	1130076.35	1139383.33	1154383.33	1169383.33	1169450.00	1172133.64	1180545.10	1188956.56
	1165500.00	1165500.00	1165500.00	1168500.00	1183500.00	1198500.00	1212000.00	1212000.00	1218000.00
8500	1127602.39	1136013.85	1150764.07	1165764.07	1184383.33	1184470.59	1184470.59	1186482.60	1194894.06
	1168500.00	1168500.00	1168500.00	1171500.00	1186500.00	1201500.00	1227000.00	1227000.00	1233000.00
9000	1133539.89	1143960.00	1158960.00	1173960.00	1201500.00	1201500.00	1201500.00	1201500.00	1201500.00
	1183500.00	1183500.00	1183500.00	1186500.00	1201500.00	1216500.00	1240500.00	1240500.00	1246500.00
9500	1139477.39	1147888.85	1162245.00	1177245.00	1201517.78	1201527.78	1201527.78	1201527.78	1206769.06
	1198500.00	1198500.00	1198500.00	1201500.00	1216500.00	1231500.00	1240500.00	1240500.00	1246500.00
10000	1145414.89	1153826.35	1162690.00	1177690.00	1201517.78	1201527.78	1201527.78	1204295.10	1212706.56
	1246500.00	1246500.00	1246500.00	1246500.00	1246500.00	1246500.00	1246500.00	1246500.00	1252500.00
10500	1151352.39	1159763.85	1168175.31	1182987.50	1201517.78	1201527.78	1201821.14	1210232.60	1210232.60
	1261500.00	1261500.00	1261500.00	1261500.00	1261500.00	1261500.00	1261500.00	1261500.00	1267500.00
11000	1157289.89	1165701.35	1174112.81	1186517.78	1201517.78	1201527.78	1207758.64	1216170.10	1224581.56
	1276500.00	1276500.00	1276500.00	1276500.00	1276500.00	1276500.00	1276500.00	1276500.00	1282500.00

Table 3: Sample results with SYMPHONY's sensitivity analysis function

[6] J. Climaco, C. Ferreira, and M. E. Captivo. Multicriteria integer programming: an overview of different algorithmic approaches. In J. Climaco, editor, *Multicriteria Analysis*, pages 248–258. Springer, Berlin, 1997.

[7] C. Cordier, H. Marchand, R. Laundy, and L. A. Wolsey. *bc-opt*: A branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86:335, 1997.

[8] J. Eckstein, C.A. Phillips, and W.E. Hart. Pico: An object-oriented framework for parallel branch and bound. Technical Report RRR 40-2000, Rutgers University, 2000.

[9] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 22:425–460, 2000.

[10] M. Ehrgott and X. Gandibleux. Multiobjective combinatorial optimization—theory, methodology and applications. In M. Ehrgott and X. Gandibleux, editors, *Multiple Criteria Optimization—State of the Art Annotated Bibliographic Surveys*, pages 369–444. Kluwer Academic Publishers, Boston, MA, 2002.

[11] M. Ehrgott and M. M. Wiecek. Multiobjective programming. In M. Ehrgott, J. Figueira, and S. Greco, editors, *State of the Art of Multiple Criteria Decision Analysis*. Kluwer Academic Publishers, Boston, MA, 2004. in print.

[12] P. K. Eswaran, A. Ravindran, and H. Moskowitz. Algorithms for nonlinear integer bicriterion problems. *Journal of Optimization Theory and Applications*, 63(2):261–279, 1989.

[13] Computational Infrastructure for Operations Research. `http://www.coin-or.org`.

[14] A. M. Geoffrion. Proper efficiency and the theory of vector maximization. *Journal of Mathematical Analysis and Applications*, 22:618–630, 1968.

[15] L. Hafer. bonsaig: Algorithms and design. Technical Report SFU-CMPTTR 1999-06, Simon Frazer University Computer Science, 1999.

[16] M. Jünger and S. Thienel. Introduction to abacus—a branch-and-cut system. *Operations Research Letters*, 22:83–?, 1998.

[17] M. Jünger and S. Thienel. The abacus system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience*, 30:1325–1352, 2001.

[18] J. Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 1998.

[19] A. Makhorin. Introduction to glpk. http://www.gnu.org/software/glpk/glpk.html.

[20] G. L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi. Minto, a mixed integer optimizer. *Operations Research Letters*, 15:47–58, 1994.

[21] T. K. Ralphs. SYMPHONY Version 4.0 User's Manual. Technical Report 03T-006, Lehigh University Industrial and Systems Engineering, 2003.

[22] T.K. Ralphs. Parallel branch and cut for capacitated vehicle routing. *Parallel Computing*, 29:607–629, 2003.

[23] T.K. Ralphs, L. Ladnyi, and M.J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98:253–280, 2003.

[24] T.K. Ralphs, L. Ladnyi, and M.J. Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *Journal of Supercomputing*, 28:215–234, 2004.

[25] T.K. Ralphs, M.J. Saltzman, and M.M. Wiecek. An improved algorithm for biobjective integer programming and its application to network routing problems. Technical Report 04T-001, Lehigh University Industrial and Systems Engineering, 2004.

[26] Linus Schrage and Laurence A. Wolsey. Sensitivity analysis for branch and bound linear programming. *Operations Research*, 33:1008–1023, 1985.

[27] R. Solanki. Generating the noninferior set in mixed integer biobjective linear programs: an application to a location problem. *Computers and Operations Research*, 18:1–15, 1991.

[28] T.K. Ralphs. *COIN/BCP User's Manual*, 2001.