The SYMPHONY Callable Library for Mixed Integer Programming

Ted Ralphs* Menal Guzelsoy†

August 19, 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 algorithm. SYMPHONY can run on a variety of architectures, including single-processor, distributed-memory parallel, and shared-memory parallel architectures under MS Windows, Linux, and other Unix operating systems. The latest version is implemented as a callable library that can be accessed either through calls to the native C application program interface, or through a C++ interface 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 rudimentary 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, this has changed dramatically. 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 [24], MIPO [2], bc-opt [7], SBB [13], GLPK [23] bonsaiG [15], PARINO [20] and FATCOP [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 frameworks available, including BCP [19], ABACUS [18], ALPS [28], and PICO [8].

The Computational Infrastructure for Operations Research (COIN-OR) Foundation is a recently formed non-profit foundation that evolved from an initiative launched by IBM in 2001 [22]. The primary goal of COIN-OR is to promote the development of open source software for operations research. The COIN-OR software repository currently hosts a dozen open source projects, all available for free download. SYMPHONY is an open-source callable library for solving MILPs that originated as a framework authored by Ralphs and Ladányi for solving difficult combinatorial problems. The original has since spawned two derivative frameworks, SYMPHONY and BCP [19]. BCP is a C++ framework that is more general than SYMPHONY, but has a steeper learning curve and cannot be used "out of the box." SYMPHONY has recently been integrated with the COIN-OR libraries and outfitted as a generic MILP solver. The source code is available for download from `www.BranchAndCut.org/SYMPHONY`.

*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`

The core solution methodology of SYMPHONY is a branch, cut, and price algorithm that incorporates most of the solution management features available in other codes. Features not yet included, but under development, include an integer presolver, a primal heuristic, and better support for column generation. The absence of the first two features hurt SYMPHONY's performance as a generic MILP solver, but it is otherwise full-featured and well-suited for implementing the customized algorithms required for solving very difficult classes of problems. It also performs well in parallel [27]. SYMPHONY depends on several other open source libraries for specific functionality, including the Cut Generation Library, the Open Solver Interface, and the MPS file parser maintained by COIN-OR, GLPK's GMPL file parser, and a third-party solver for linear-programming problems (LPs), such as the one maintained by COIN-OR (CLP).

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 (OSI). 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 [27], [25], and [29]. SYMPHONY's legacy features are well-detailed in the SYMPHONY User's Manual [26], 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 rudimentary sensitivity analysis. To our knowledge, these features are not yet available in other MILP codes and should be of interest to potential users. Below, we briefly describe the new C API, the C++ interface, and the use of the user callback functions. We assume the reader is familiar with the fundamentals of mixed-integer linear programming.

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 `symphony_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 command-line parser for setting commonly used parameters.

`sym_load_problem()`: Reads the problem data and sets up the root subproblem (see Section 2.3).

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

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

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

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

Figure 1: A generic MILP solver with implemented with SYMPHONY in C.

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

By default, SYMPHONY reads an MPS or GMPL file specified by the user, although this behavior can be overridden by implementing a user callback that reads the data from a file in a customized format (see Section 2.3). SYMPHONY can also be used easily with FLOPC++ [17], an open-source modeling system that accesses solvers through the OSI. 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. Note that 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 MPS file called `sample.mps` and solve it using this program, the following command would be issued:

```
symphony -F sample.mps
```

The code of Figure 1 is identical for both sequential and parallel computations. The choice between sequential and parallel execution modes is made at compile-time. In addition to the parts of the API 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.

2.2 The OSI Interface

The OSI is a C++ interface class maintained by COIN-OR that provides a standard API for accessing a variety of solvers for mathematical programs. A code implemented using calls to the methods in the OSI base class can be linked with any solver for which there is an OSI implementation. This allows development of solver-independent codes and eliminates many portability issues. The current incarnation of OSI supports only solvers for linear and mixed-integer linear programs. 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. For each method in the OSI base class, there is a corresponding method in the C API. The OSI methods are implemented simply as wrapped calls to the C library. When an OSI object is constructed, `sym_open_environment()` is called and a pointer to the environment is stored. When the OSI object is destroyed, `sym_close_environment()` is called and the environment is deleted. To fully support SYMPHONY's capabilities, we have extended the OSI interface to include some methods not in the base class, such as a `parseCommandLine()`

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

Figure 2: A generic MILP solver implemented with SYMPHONY using OSI.

method. Figure 2 shows the program of Figure 1 implemented using the OSI interface. The code would be exactly the same for accessing any customized SYMPHONY solver, sequential or parallel.

The current version of the OSI is geared primarily toward support of LP solvers. One reason for this is that LP solvers based on the simplex algorithm support much richer functionality than do typical MILP solvers. In SYMPHONY 5.0, we have begun to extend some of this functionality to the realm of MILP solvers. For example, our OSI implementation supports warm starting and some basic sensitivity analysis. The implementation of this functionality is rudimentary at the moment, but will be improved in future versions.

2.3 User Callback Functions

The user's main avenues for customization are the tuning of parameters and the implementation of one or more of over 50 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, including branching, cutting-plane generation, management of the cut pool, management of the LP relaxation, search and diving strategies, program output, etc. Callbacks in SYMPHONY are implemented slightly differently than in other popular libraries. Each callback 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 exits without further action. If the user requests that SYMPHONY perform a certain default action, then this is done. All callback functions have names that begin with the prefix "`user`." Files containing default function stubs for the callbacks are provided along with the SYMPHONY source code. These can then be modified by the user as desired. Makefiles and Microsoft Visual C++ project files are provided for automatic compilation. Below is a sampling of commonly used callback functions.

`user_initialize_root_node()`: The user can specify a *core relaxation* consisting of cuts and variables that are to be present in every subproblem. These cuts and variables are never considered for removal and need not be included in the description of each search-tree node, so specifying a core can potentially save memory and increase efficiency.

`user_display_solution()`: The user can specify a custom output format for feasible solutions. This is useful for combinatorial problems where a simple list of variables and values is not interpretable by a human.

`user_create_subproblem()`: Rather than specifying the model directly using an MPS or GMPL file, the user can write a function that creates the initial LP relaxations at each node "on the fly."

`user_find_cuts()`: The user can generate custom classes of cutting planes by separating the current relaxed solution.

`user_is_feasible()`: The user can determine whether a given solution is feasible or not. This is needed in cases where integrality does not necessarily imply feasibility.

`user_select_candidates()`: The user can select candidates for strong branching.

`user_compare_candidates()`: After presolving, the user can choose a candidate to be used for branching.

`user_generate_column()`: The user can generate columns using this function.

`user_logical_fixing()`: The user can tighten bounds or fix variables based on implicit problem structure.

A full list of callbacks is contained in the SYMPHONY User's Manual [26].

3 Solution Procedures

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 facilitates the parallel implementation and eases code maintenance. The five modules are the *master*, *tree manager*, *cut generator*, *cut pool*, and *node processor* modules. Only the master module is persistent and the environment pointer described earlier is a pointer to the master module. Other modules encapsulate the specific functionality needed to execute the algorithms and exist only while a solve call is active. Each module can function as an independent remote process for parallel execution. A more complete description of the modular design of SYMPHONY can be found in [27].

For LPs, 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 re-solving 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 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 `initialSolve()` 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, communicating either through shared memory or through a message-passing protocol, such as PVM [14].

The overall flow of the algorithm is similar to other branch-and-bound implementations and is described in detail in [27]. 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 solution algorithms, as we describe next.

3.2 Solve from Warm Start

Among the utility classes maintained by COIN-OR is a base class for describing the data needed to warm start the solution process for a particular algorithm. To support this option in 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 that created 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. 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 build in fault tolerance by periodically backing up warm-start information to disk, 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.

The ability to re-solve after modifying problem data has a wide range of applications in practice. One obvious application is to allow modification of problem data after the solution procedure has already been initiated. Another obvious application arises when the solution of a family of related MILPs is required, as occurs, for instance, in decomposition algorithms, in parametric and stochastic programming algorithms, in multicriteria optimization algorithms, and in algorithms for analyzing infeasible mathematical models.

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 point 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 shows 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 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

```
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.setSymParam(OsiSymKeepWarmStart, true);
   si.initialSolve();
   si.setSymParam(OsiSymFindFirstFeasible, false);
   si.setSymParam(OsiSymSearchStrategy, BEST_FIRST_SEARCH);
   si.resolve();
}
```

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

algorithm to continue. Changes to the original data that do not invalidate the subproblem warm-start data, i.e., the basis information for the LP relaxation, are the easiest to accommodate. Our current procedures only handle modifications to the right-hand side and objective function vectors of the original MILP. Note that modifications may invalidate valid inequalities that have been previously generated. Currently, we discard such cuts. Methods for handling other modifications, such as the addition and deletion of columns and rows 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 modified input data. After this reprocessing, the computation can continue as usual.

Code illustrating the use of the warm start facility 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 re-solved from the saved warm start. As an illustration of the use of warm starting procedures in practice, Table 1 shows the results of solving a set of 2-stage stochastic integer programming instances modified from [16, 12, 1] with the dual decomposition algorithm of [4]. We used a straightforward implementation of the subgradient algorithm to solve the Lagrangian duals and SYMPHONY to solve the subproblems, with and without warm starting from one iteration to the next. SUTIL [21] was used to read in the instances. The presence of a gap indicates that the problem was not solved to within the gap tolerance in the time limit. Although the running times are not competitive overall because of the slow convergence of our subgradient algorithm, one can clearly see the improvement arising from the use of warm starting.

3.2.3 Persistent Cut Pools

To complement the ability to save the search tree, the user can also save and reuse the global cut pool. When saving 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.

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

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

Such pools exist as part of the master module and are attached to the tree manager whenever a solve call is made.

3.3 Bicriteria Solve

A bicriteria MILP is a generalization of a standard MILP in which one considers a second objective function. One notion of solving a bicriteria MILP consists of generating all *Pareto outcomes*. An *outcome* is the pair of objective function values corresponding to a given feasible solution. The Pareto outcomes are those for which there is no other outcome for which both components are at least as small, and at least one is strictly smaller. In some cases, we are only be interested in the *supported outcomes*, which are those corresponding to solutions to a MILP with a single objective function formed by a convex combination of the two original objectives. For those readers not familiar with bicriteria integer programming, surveys of methodology are provided in [6] and more recently in [9, 10] and [11].

In [29], we describe an asymptotically optimal algorithm for solving bicriteria MILPs. SYMPHONY 5.0 contains a generic implementation of this algorithm, 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 and have also added a method for invoking the bicriteria solver called `multiCriteriaBranchAndBound()`. Implementing this algorithm requires the underlying solver to have the ability to generate, among all optimal solutions to a MILP 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 [29].

The algorithm itself consists of the solution of a sequence of MILPs with identical feasible region, but differing objective functions. Thus, it is possible in principle to use warm starting to improve efficiency. Although the objective function is nonlinear in the case of generating Pareto outcomes, it can be linearized through a standard reformulation. This reformulation does require

Problem	Tree Size Without WS	Tree Size With WS	% Gap Without WS	% Gap With WS	CPU min Without WS	CPU min With WS
storm8	1	1	-	-	14.75	8.71
storm27	5	5	-	-	69.48	48.99
storm125	3	3	-	-	322.58	176.88
LandS27	71	69	-	-	6.50	4.99
LandS125	37	29	-	-	15.72	12.72
LandS216	39	35	-	-	30.59	24.80
dcap233_200	39	61	-	-	256.19	120.86
dcap233_300	111	89	0.387	-	1672.48	498.14
dcap233_500	21	36	24.701	14.831	1003	1004
dcap243_200	37	53	0.622	0.485	1244.17	1202.75
dcap243_300	64	220	0.0691	0.0461	1140.12	1150.35
dcap243_500	29	113	0.357	0.186	1219.17	1200.57
sizes3	225	165	-	-	789.71	219.92
sizes5	345	241	-	-	964.60	691.98
sizes10	241	429	0.104	0.0436	1671.25	1666.75

Table 1: Results of using warm starting to solve stochastic integer programs.

modification of the constraint matrix from iteration to iteration, but it is easy to show that these modifications do not invalidate the basis, allowing the warm start to be loaded very efficiently.

In [29], we report on our experience using the bicriteria solver to analyze the tradeoff between fixed and variable costs for a class of network routing problems. Applying our rudimentary version of warm starting to this problem over the same test set, we have achieved promising results, improving solution times in almost all cases. A summary of results is shown in Figure 5, with the dark bars representing running times with warm starting and the light bars representing running times without warm starting over two data sets described in [29]. The effect is evident, although it is also clear that further refinements to our procedures are still needed.

4 Sensitivity Analysis

Besides yielding the ability to closely examine the tradeoffs between competing objectives, the bicriteria solver can be used to solve *parametric MILPs*, which are families of MILPs parameterized by a single scalar. Typically, parametric MILPs are obtained by parameterizing either the objective function or the right-hand side, replacing the usual single vector with a combination of two vectors. The goal is to determine the complete set of optimal values that occur as the parameter is varied over a given interval. This set characterizes how the optimal value varies as a function of change in either the objective function or the right-hand side in one dimension and is an elementary form of global sensitivity analysis (see [3] for a discussion of this in the case of linear-programming models).

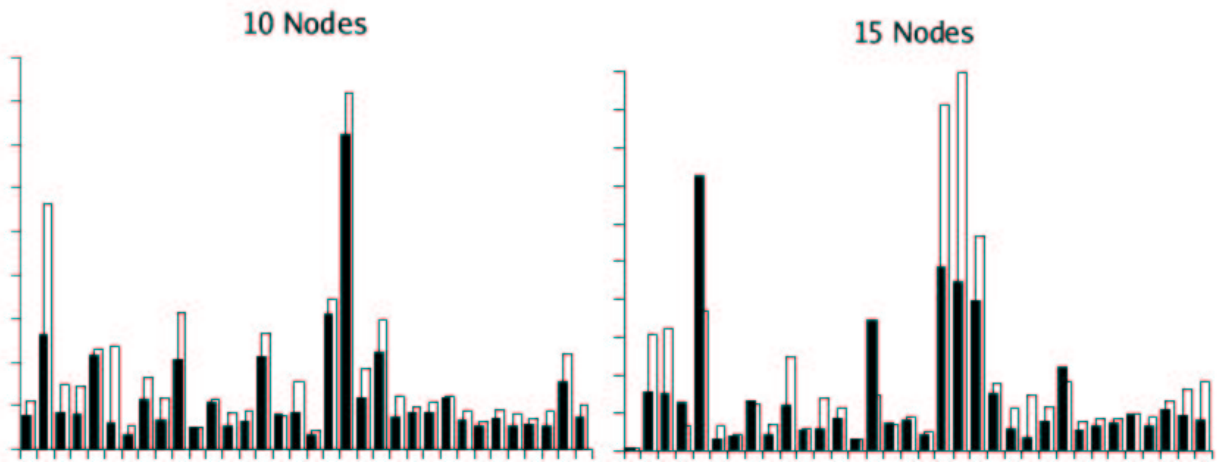

Figure 5: Results of using warm starting to solve multicriteria optimization problems.

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

Figure 6: Performing sensitivity analysis with SYMPHONY's bicriteria solver.

As an example, consider the following simple parametric MILP.

$$\begin{aligned} \max\ & 8x_1 + \theta x_2, \\ \text{s.t.}\quad & 7x_1 + x_2 \leq 56, \\ & 28x_1 + 9x_2 \leq 252, \\ & 3x_1 + 7x_2 \leq 105, \text{and} \\ & x_1, x_2 \geq 0, \text{ integral.} \end{aligned} \tag{1}$$

Taking the first objective function to be $(8, 1)$ and the second objective function $(0, 1)$, we can determine how the optimal value of the MILP varies as a function $p(\theta)$ of the second objective coefficient simply by invoking the bicriteria solution algorithm to enumerate all supported solutions. Figure 6 shows the code for performing this analysis. Applying the bicriteria solver of Figure 6 results in the function $p(\theta)$ shown in Table 2.

In addition to the sensitivity analysis that can be undertaken by using SYMPHONY's bicriteria solver, we have also implemented the method suggested in [30] for performing approximate sensitivity analysis on the right-hand side vector and some related procedures. The method in [30] is based on constructing an approximate dual price function from the dual solutions obtained while

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 example MILP (1).

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

Figure 7: Performing sensitivity analysis with SYMPHONY

solving the LP relaxations in each search-tree node. 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 re-solve. Figure 7 shows a program that uses this sensitivity analysis function. This code will produce a lower bound for a modified problem with new right-hand side values of 7000 and 6000 in the 4^{th} and 7^{th} rows. Similar functions are provided for obtaining quick upper and lower bounds after changing either the right-hand side or objective function vectors.

5 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, re-solving MILPs from a warm start, and analyzing bicriteria MILPs. To our knowledge, these techniques are not available in other solvers. The computational results presented here are very preliminary, but show promise. These capabilities are still being refined and new techniques developed, and we hope to improve them 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. However, 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] S. Ahmed. SIPLIB, 2004. Available from http://www.isye.gatech.edu/ sahmed/siplib.

[2] 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.

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

[4] C.C. Caroe and R. Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24:37–45, 1999.

[5] Q. Chen and M. C. Ferris. FATCOP: A fault tolerant Condor-PVM mixed integer program solver. *SIAM Journal on Optimization*, 11:1019–1036, 2001.

[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*, Boston, MA, 2004. Kluwer Academic Publishers.

[12] A. Felt. Stochastic linear programming data sets, 2004. Available from http://www.uwsp.edu/math/afelt/slptestset.html.

[13] J. Forrest. Simple branch and bound, 2004. Available from http://www.coin-or.org.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.

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

[16] D. Holmes. Stochastic linear programming data sets, 2004. Available from http ://users.iems.nwu.edu/ jrbirge/html/dholmes/post.html.

[17] T.H. Hultberg. FlopC++, 2004. Available from http://www.mat.ua.pt/thh/flopc/.

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

[19] L. Ladányi and T.K. Ralphs. *COIN/BCP User's Manual*, 2001. Available from http://www.coin-or.org.

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

[21] J.T. Linderoth. SUTIL, 2004.

[22] R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47:57–66, 2003.

[23] A. Makhorin. Introduction to GLPK, 2004. Available from http://www.gnu.org/software/glpk/glpk.html.

[24] G. L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

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

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

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

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

[29] T.K. Ralphs, M.J. Saltzman, and M.M. Wiecek. An improved algorithm for biobjective integer programming and its application to network routing problems. To appear in Annals of Operations Research, 2004.

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