The SYMPHONY Framework for Mixed-Integer Linear Programming: Advanced Features

TED RALPHS

ISE Department
COR@L Lab
Lehigh University
tkralphs@lehigh.edu

COR@L
COMPUTATIONAL OPTIMIZATION
RESEARCH AT LEHIGH

DIMACS Workshop on COIN-OR
July 19, 2006

Outline

API

- Advanced customization is performed using the user callback subroutines.
- There are more than 50 callbacks that can be implemented to develop a custom solver.
- The user can override SYMPHONY's default behavior in a variety of ways.
- The callbacks are invoked in SYMPHONY from a wrapper function that checks the results of the user's action.

Callback return status codes

- `USER_SUCCESS`: User executed action successfully.
- `USER_ERROR`: Something went wrong.
- `USER_DEFAULT`: SYMPHONY should perform this action.

- Each callback passes a pointer to the user's previously created data structure.

Callback Overview

Commonly used callback routines

- `user_initialize_root_node()`
- `user_display_solution()`
- `user_create_subproblem()`
- `user_find_cuts()`
- `user_is_feasible()`
- `user_select_candidates()`
- `user_compare_candidates()`
- `user_logical_fixing()`

Using the SYMPHONY Callbacks

- Function stubs for the callbacks are in the `Applications/USER` subdirectory.
- They are in files divided by functional module:

User Callback Files

- `USER/src/Master/user_master.c`
- `USER/src/LP/user_lp.c`
- `USER/src/CutGen/user_cg.c`
- `USER/src/CutPool/user_cp.c`

- The applications in the `SYMPHONY/Applications/` directory can be used as templates.

Building an Application

- To use the callbacks, a new library must be built that includes hooks for the callbacks.
- This is done automatically by the provided Makefile or MSVC++ project file.

Unix/Linux/CYGWIN/MinGW

- Type `make` in the `USER` subdirectory.
- Executable will be `USER/bin/$(ARCH)/$(LP_SOLVER)/symphony`

MSVC++

- Open the `USER\WIN32\user.dsw` file.
- Modify settings as before.
- Build the `user` project.

Customizing the Output

This code shows a solution display callback for the matching solver.

```
int user_display_solution(void *user, double lpetol, int varnum,
                          int *indices, double *values,
                          double objval)
{
   user_problem *prob = (user_problem *) user;
   int index;

   for (index = 0; index < varnum; index++){
      if (values[index] > lpetol) {
         printf("%2d matched with %2d at cost %6d\n",
                prob->match1[indices[index]],
                prob->match2[indices[index]],
                prob->cost[prob->match1[indices[index]]]
                [prob->match2[indices[index]]]);
      }
   }
   return(USER_SUCCESS);
}
```

Generating Cutting Planes

This code shows a cut generator for the matching solver.

```
int user_find_cuts(void *user, int varnum, int iter_num, int level,
                   int index, double objval, int *indices, double *values,
                   double ub, double etol, int *num_cuts, int *alloc_cuts,
                   cut_data ***cuts)
{
   /* There’s some code for declaring variables and allocating memory left out */
   for (i = 0; i < varnum; i++){ /* forming a dense solution vector */
      edge_val[prob->node1[indices[i]]][prob->node2[indices[i]]] = values[i];
   }
   for (i = 0; i < prob->nnodes; i++){
      for (j = i+1; j < prob->nnodes; j++){
         for (k = j+1; k < prob->nnodes; k++){
            if (edge_val[i][j]+edge_val[j][k]+edge_val[i][k] > 1.0 + etol){
               cutind[0] = prob->index[i][j];
               cutind[1] = prob->index[j][k];
               cutind[2] = prob->index[i][k];
               cutval[0] = cutval[1] = cutval[2] = 1.0;
               cg_add_explicit_cut(3, cutind, cutval, 1.0, 0, ’L’,
                                   TRUE, num_cuts, alloc_cuts, cuts);
               cutnum++;

            }
         }
      }
   }
   return(USER_SUCCESS);
}
```

Warm Starts for MILP

- To allow resolving from a warm start, we have defined a SYMPHONY warm start structure, based on the `CoinWarmStart` class.
- The class stores a snapshot of the search tree, with node descriptions including:
  - lists of active cuts and variables,
  - branching information,
  - warm start information, and
  - current status (candidate, fathomed, etc.).
- The tree is stored in a compact form by storing the node descriptions as differences from the parent.
- Other auxiliary information is also stored, such as the current incumbent.
- A warm start can be saved at any time and then reloaded later.
- The warm starts can also be written to and read from disk.
- Has the same look and feel as warm starting for LP.

Warm Starting Procedure

After modifying parameters

- If only parameters have been modified, then the candidate list is recreated and the algorithm proceeds as if left off.
- This allows parameters to be tuned as the algorithm progresses if desired.

After modifying problem data

- We limit modifications to those that do not invalidate the node warm start information.
- Currently, we only allow modification of rim vectors.
- After modification, all leaf nodes must be added to the candidate list.
- After constructing the candidate list, we can continue the algorithm as before.

Warm Starting Code (Parameter Modification)

- The following example shows a simple use of warm starting to create a dynamic algorithm.
- Here, the warm start is automatically save and reloaded.

Warm Starting Example

```
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();
}
```

Warm Starting Code (Problem Modification)

- The following example shows how to warm start after problem modification.

Warm Starting Example

```
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, 1);
   si.setObjCoeff(200, 150);
   si.setWarmStart(ws);
   si.resolve();
}
```

Warm Starting Example

Applying the code from the previous slide to the MIPLIB 3 problem `p0201`, we obtain the results below.

Results

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

Note that the warm start doesn't reduce the number of nodes generated, but does reduce the solve time significantly.

Introduction

- The general form of a (pure) bicriteria ILP is

$$\begin{array}{rrcl} \text{vmax} & [cx, dx], & & \\ \text{subject to} & Ax & \leq & b, \\ & x & \in & \mathbb{Z}^n. \end{array} \tag{1}$$

- Solutions don't have single objective function values, but pairs of values called *outcomes*.
- A feasible $\hat{x}$ is called *efficient* if there is no feasible $\bar{x}$ such that $c\bar{x} \geq c\hat{x}$ and $d\bar{x} \geq d\hat{x}$, with at least one inequality strict.
- The outcome corresponding to an efficient solution is called *Pareto*.
- The goal of a bicriteria ILP is to enumerate Pareto outcomes.

Example

Consider the following bicriteria ILP:

$$\begin{array}{rrcl} \text{vmax} & [8x_1, x_2], & & \\ \text{subject to} & 7x_1 + x_2 & \leq & 56, \\ & 28x_1 + 9x_2 & \leq & 252, \\ & 3x_1 + 7x_2 & \leq & 105, \\ & x_1, x_2 & \geq & 0 \end{array}$$

Analyzing this bicriteria ILP, we can determine the price function $p(\theta)$ returning the optimal solution to the single-objective ILP with paramterized objective $8x_1 + \theta x_2$.

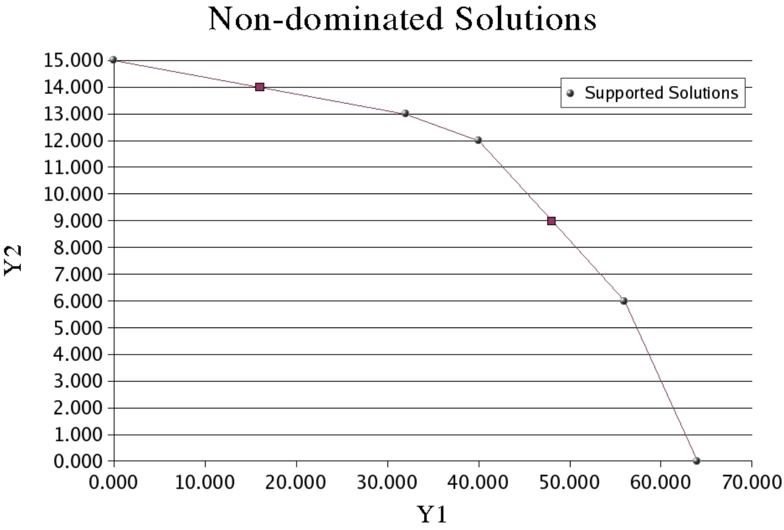
Code for Example

The following code solves the model on the previous slide.

Simple Bicriteria Solver

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

Pareto Outcomes for Example

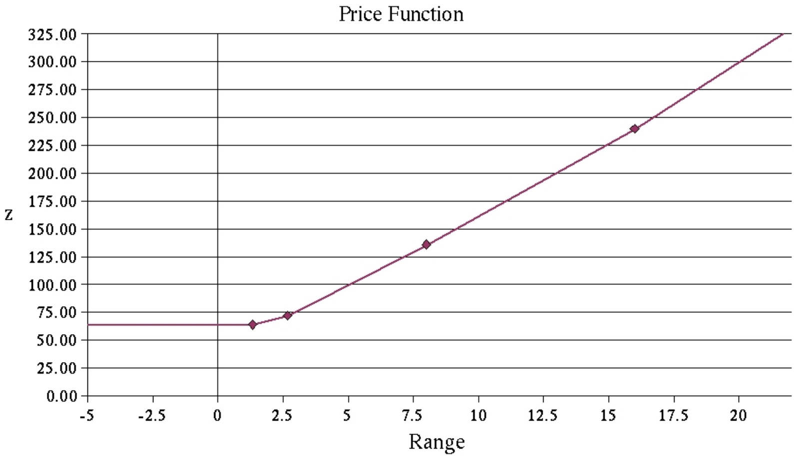


Price Function for Example

By examining the supported solutions and break points, we can easily determine $p(\theta)$, the optimal solution to the ILP with objective $8x_1 + \theta$.

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

Graph of Price Function



Other Sensitivity Analysis

SYMPHONY will calculate bounds after changing the objective or right-hand side vectors.

Code for Sensitivity Analysis

```
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);
}
```

Building and Executing for SMP

- Warning: the shared-memory configuration has not been tested in quite some time, but should still work.
- Building for shared memory is exactly the same as for sequential, except an OpenMP-enabled compiler must be used.
- The number of threads/processors must be specified on the command line, as follows

```
symphony -p 4 -F sample.mps
```

- At run time, multiple threads will be created, one for the Master/TM, and one for each of the node processors.
- All modules will communicate through shared memory.

Building for Distributed Memory

- To run the distributed memory version, you must first install PVM.
- Once PVM is installed and you are able to start the console, you need to make a few modifications to the config file.
- For the “standard” configuration, set

```
COMM_PROTOCOL = PVM
SYM_COMPILE_IN_CG = TRUE
SYM_COMPILE_IN_CP = TRUE
SYM_COMPILE_IN_LP = FALSE
SYM_COMPILE_IN_TM = TRUE
```

- This will result in two executables and two callable libraries.

Parallel Executables

symphony_lp_cg
symphony_m_tm_cp

Running on Distributed Memory

- To run the distributed memory version, you must first start PVM.
- Then run the master executable as in the shared memory case.

```
symphony_m_tm_cp -p 4 -F sample.mps
```

- As with any PVM application, the executables must be in PVM's path.
- The easiest way to accomplish this is to create a soft link from `$HOME/pvm3/bin/$PVM_ARCH`

```
cd ~/pvm3/bin/$PVM_ARCH
ln -s ~/COIN-SYMPHONY/SYMPHONY/bin/$ARCH/$LP_SOLVER/sympho
```

Building Applications in Parallel

- To build an application in parallel, you must set some additional variables in the application makefile

```
COMPILE_IN_CG = TRUE
COMPILE_IN_CP = TRUE
COMPILE_IN_LP = FALSE
COMPILE_IN_TM = TRUE
```

- As in the previous case, this will result in two executables.
- Start PVM and run the master executable as before.
- In some cases, additional coding is needed to allow applications to run in parallel.
- Take a look at the VRP example for ideas.

Where to Get Help

The following Web sites have manuals, tutorial material, and more.

`https://projects.coin-or.org/SYMPHONY`
`http://www.branchandcut.org/SYMPHONY`
`coin-symphony@list.coin-or.org`

My personal home page has papers about SYMPHONY and other materials that may be of interest.

`http://www.lehigh.edu/~tkr2`