Simple Walkthrough for Using SYMPHONY

Michael Trick*and Menal Guzelsoy†

June 24, 2003

SYMPHONY is a callable library including a set of user callback routines to allow it to solve generic MIPs, as well as easily create custom branch-cut-price solvers. Having been fully integrated with COIN, SYMPHONY is capable to use CPLEX, OSL, CLP, GLPK, DYLP, SOPLEX, VOL and XPRESS-MP through the COIN/OSI interface (first two can also be used through the built-in APIs without using COIN/OSI). The SYMPHONY system includes numerous applications: Vehicle Routing Problem (VRP), Capacitated Node Routing Problem (CNRP), Multi-Criteria Knapsack Problem (MCKP), Mixed Postman Problem (MPP), Set Partitioning Problems (SPP-basic and advanced). These applications are extremely well done, but, in generality, are difficult to understand.

Here is a walkthrough for a very simple application that uses SYMPHONY. Rather than presenting the code in its final version, I will go through the steps that I went through. Note that some of the code is lifted from the vehicle routing application. This code is designed to be a sequential code. The MATCH application itself is available for download at `http://www.branchandcut.org/MATCH`.

Our goal is to create a minimum one-matching code on a complete graph. Initially we will just formulate this as an integer program. Then we will include a set of constraints that can be added by cut generation.

I begin with the template file in the `USER` subdirectory included in SYMPHONY. This gives stubs for each user routine. First I need to define the data needed for one-matching. This data will be included in the structure `USER_PROBLEM` in the file `user.h`. Initially, the data will be the number of nodes and the cost matrix, so change `USER_PROBLEM` in `user.h` to be

```
typedef struct USER_PROBLEM{
   int              colnum;         /* Number of rows in base matrix */
   int              rownum;         /* Number of columns in base matrix */
   user_parameters  par;            /* Parameters */
   int              nnodes;         /* Number of nodes */
   int              cost[200][200]; /* Cost of assigning i to j */
```

*Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213 `trick@cmu.edu`, `http://mat.gsia.cmu.edu/trick`

†Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, `megb@lehigh.edu`

```
   int              node1[20000];   /* First index of each variable */
   int              node2[20000];   /* Second index of each variable */
}user_problem;
```

A "real programmer" would not hard-code problem sizes like that, but I am trying to get a minimal code. The fields `node1` and `node2` will be used later in the code in order to map constraints back to the corresponding nodes. Additionally, add the declarations of two functions which will be needed later:

```
int match_read_data PROTO((sym_environment *env, void *user, char *infile));
int match_load_problem PROTO((sym_environment *env, void *user));
```

Next, read in the data. We could easily use the `user_io()` user-callback for this (see this routine in `user_master.c` for an illustration). However, in order to show how it can be done explicitly, we will define our own function `match_read_data()` in `user_main.c` to fill in the user data structure and then use `sym_set_user_data()` to pass this structure to SYMPHONY. The template already has command-line options set up for the user. The "-F" flag defines the data file, so we will use that to put in the data. The datafile contains first the number of nodes in the graph (`nnodes`) followed by the pairwise cost matrix (nnode by nnode). Read this in with the `match_read_data()` routine in `user_main.c`:

```
int match_read_data(sym_environment *env, void *user, char *infile)
{
   int i, j;
   FILE *f = NULL;
   /* This gives you access to the user data structure. */
   user_problem *prob = (user_problem *) user;

   if ((f = fopen(infile, "r")) == NULL){
      printf("main(): user file %s can't be opened\n", infile);
      return(ERROR__USER); /*error check for existence of parameter file*/
   }

   /* Read in the costs */
   fscanf(f,"%d",&(prob->nnodes));
   for (i = 0; i < prob->nnodes; i++)
      for (j = 0; j < prob->nnodes; j++)
          fscanf(f, "%d", &(prob->cost[i][j]));

   prob->colnum = (prob->nnodes)*(prob->nnodes-1)/2;
   prob->rownum = prob->nnodes;

   /* This will pass the user data in to SYMPHONY*/
   sym_set_user_data(env, (void *)prob);
```

```
   return (FUNCTION_TERMINATED_NORMALLY);
}
```

Note that we set the number of rows and columns in this routine. We can now define the integer program. We will have a variable for each edge (i, j) with $i < j$. We have a constraint for each node i forcing one edge to be incident to i in the matching.

We define the IP in our other helper function `match_load_problem()` in `user_main.c`. In the first part of this routine, we will represent the IP with a set of arrays, and then in the second part, will load this representation to SYMPHONY through `sym_explicit_load_problem()`. Note that, we could also create the same IP model in `user_create_subproblem()` callback (see this routine in `user_lp.c` for an illustration).

```
int match_load_problem(sym_environment *env, void *user){

   int i, j, index, n, m, nz, *matbeg, *matind;
   double *matval, *lb, *ub, *obj, *rhs, *rngval;
   char *sense, *is_int;
   user_problem *prob = (user_problem *) user;

   /* set up the inital LP data */
   n = prob->colnum;
   m = prob->rownum;
   nz = 2 * n;

   /* Allocate the arrays */
   matbeg  = (int *) malloc((n + 1) * ISIZE);
   matind  = (int *) malloc((nz) * ISIZE);
   matval  = (double *) malloc((nz) * DSIZE);
   obj     = (double *) malloc(n * DSIZE);
   lb      = (double *) calloc(n, DSIZE);
   ub      = (double *) malloc(n * DSIZE);
   rhs     = (double *) malloc(m * DSIZE);
   sense   = (char *) malloc(m * CSIZE);
   rngval  = (double *) calloc(m, DSIZE);
   is_int  = (char *) malloc(n * CSIZE);

   /* Fill out the appropriate data structures -- each column has
      exactly two entries */
   index = 0;
   for (i = 0; i < prob->nnodes; i++) {
      for (j = i+1; j < prob->nnodes; j++) {
         prob->node1[index] = i; /* The first node of assignment 'index' */
         prob->node2[index] = j; /* The second node of assignment 'index' */
```

```
         obj[index] = prob->cost[i][j]; /* Cost of assignment (i, j) */
         is_int[index] = TRUE;
         matbeg[index] = 2*index;
         matval[2*index] = 1;
         matval[2*index+1] = 1;
         matind[2*index] = i;
         matind[2*index+1] = j;
         ub[index] = 1.0;
         index++;
      }
   }
   matbeg[n] = 2 * n;

   /* set the initial right hand side */
   for (i = 0; i < prob->nnodes; i++) {
      rhs[i] = 1;
      sense[i] = 'E';
   }

   /* Load the problem to SYMPHONY */
   sym_explicit_load_problem(env, n, m, matbeg, matind, matval, lb, ub,
                             is_int, obj, 0, sense, rhs, rngval, true);

   FREE(matbeg);
   FREE(matind);
   FREE(matval);
   FREE(lb);
   FREE(ub);
   FREE(obj);
   FREE(sense);
   FREE(rhs);
   FREE(rngval);

   return (FUNCTION_TERMINATED_NORMALLY);
}
```

Now, we are ready to gather everything in the `main()` routine in `user_main()`. This will involve to create a SYMPHONY environment and a user data structure, read in the data, create the corresponding IP, load it to the environment and ask SYMPHONY to solve it (`CALL_FUNCTION` is just a macro to take care of the return values):

```
int main(int argc, char **argv)
{
   int termcode;
   char * infile;
```

```
   /* Create a SYMPHONY environment */
   sym_environment *env = sym_open_environment();

   /* Create a user problem structure to read in the data and then pass it to
      SYMPHONY.
   */
   user_problem *prob = (user_problem *)calloc(1, sizeof(user_problem));

   CALL_FUNCTION(sym_parse_command_line(env, argc, argv) );

   /* Get the data file name which was read in by '-F' flag. */
   CALL_FUNCTION(sym_get_str_param(env, "infile_name", &infile));

   CALL_FUNCTION(match_read_data(env, (void *) prob, infile));

   CALL_FUNCTION(match_load_problem(env, (void *) prob ));

   CALL_FUNCTION(sym_solve(env) );

   CALL_FUNCTION(sym_close_environment(env) );

   return(0);
}
```

OK, that's it. That defines an integer program, and if you compile and optimize it, the rest of the system will come together to solve this problem. Here is a data file to use:

```
6
0 1 1 3 3 3
1 0 1 3 3 3
1 1 0 3 3 3
3 3 3 0 1 1
3 3 3 1 0 1
3 3 3 1 1 0
```

The optimal value is 5. To display the solution, we need to be able to map back from variables to the nodes. That was the use of the `node1` and `node2` parts of the `USER_PROBLEM`. We can now use `user_display_solution()` in `user_master.c` to print out the solution:

```
int user_display_solution(void *user, double lpetol, int varnum, int *indices,
                          double *values, double objval)
{
   /* This gives you access to the user data structure. */
```

```
   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->node1[indices[index]],
                 prob->node2[indices[index]],
                 prob->cost[prob->node1[indices[index]]]
                 [prob->node2[indices[index]]]);
      }
   }

   return(USER_SUCCESS);
}
```

We will now update the code to include a crude cut generation. Of course, I am eventually aiming for a Gomory-Hu type odd-set separation (ala Groetschel and Padberg) but for the moment, let's just check for sets of size three with more than value 1 among them (such a set defines a cut that requires at least one edge out of any odd set). We can do this by brute force checking of triples.

This is done in two steps: first, we find cuts and store them as we wish. Then we "unpack" the cuts and create the violated inequalities. Finding the cuts is in the routine `user_find_cuts()` in `user_cg.c`. In the following, "new_cuts" is an array which is zero except for `new_cuts[i]`, `new_cuts[j]` and `new_cuts[k]` (where `i`, `j`, and `k` represents the violating triple) which are "1."

```
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)
{
   user_problem *prob = (user_problem *) user;
   double edge_val[200][200]; /* Matrix of edge values */
   int i, j, k;
   int *new_cuts;
   cut_data cut;

   new_cuts = (int *) malloc(prob->nnodes * ISIZE);

   /* Allocate the edge_val matrix to zero (we could also just calloc it) */
   memset((char *)edge_val, 0, 200*200*ISIZE);

   for (i = 0; i < varnum; i++) {
```

```
      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) {
               memset(new_cuts, 0, prob->nnodes * ISIZE);
               new_cuts[i] = 1;
               new_cuts[j] = 1;
               new_cuts[k] = 1;
               cut.size = (prob->nnodes)*ISIZE;
               cut.coef = (char *) new_cuts;
               cut.rhs = 1.0;
               cut.range = 0.0;
               cut.type = TRIANGLE;
               cut.sense = 'L';
               cut.deletable = TRUE;
               cut.branch = ALLOWED_TO_BRANCH_ON;
               cg_send_cut(&cut, num_cuts, alloc_cuts, cuts);
            }
         }
      }
   }

   FREE(new_cuts);

   return(USER_SUCCESS);
}
```

Note the call of `cg_send_cut()`, which tells the system about any cuts found.

The final step is to give a routine that creates cuts from the structure defined in `user_find_cuts()`. This is the routine `user_unpack_cuts()` in `user_lp.c`. The levels of indirection here are somewhat confusing (I don't think I have seen a *** variable before), but the mallocs in the following create things in the right order:

```
int user_unpack_cuts(void *user, int from, int type, int varnum,
                     var_desc **vars, int cutnum, cut_data **cuts,
                     int *new_row_num, waiting_row ***new_rows)
{
   user_problem *prob = (user_problem *) user;

   int i, j, nzcnt;
```

```
   int *cutval;
   waiting_row **row_list;

   *new_row_num = cutnum;
   if (cutnum > 0)
      *new_rows =
          row_list = (waiting_row **) calloc (cutnum, sizeof(waiting_row *));

   for (j = 0; j < cutnum; j++){
      row_list[j] = (waiting_row *) malloc(sizeof(waiting_row));
      switch (cuts[j]->type){

      case TRIANGLE:
         cutval = (int *) (cuts[j]->coef);
         row_list[j]->matind = (int *) malloc(varnum * ISIZE);
         row_list[j]->matval = (double *) malloc(varnum * DSIZE);
         row_list[j]->nzcnt = 0;
         for (nzcnt = 0, i = 0; i < varnum; i++){
            if (cutval[prob->node1[vars[i]->userind]] &&
                cutval[prob->node2[vars[i]->userind]]){
               row_list[j]->matval[nzcnt] = 1.0;
               row_list[j]->matind[nzcnt++] = vars[i]->userind;
            }
         }
         row_list[j]->nzcnt = nzcnt;
         break;

       default:
           printf("Unrecognized cut type!\n");
      }
   }

   return(USER_SUCCESS);
}
```

If you now solve the matching problem on the sample data set, the number of nodes in the branch and bound tree should just be 1 (rather than 3 without cut generation).