The COIN-OR Open Solver Interface

Matthew Saltzman

Clemson University

DIMACS COIN-OR Workshop 7/17/2006

Outline

What is OSI?

The COIN-OR Open Solver Interface (OSI) attempts to provide a uniform API for math programming solvers embedded in applications.

History

- First released in 2000.
- Supported OSL, Volume, and XPRESS.
- Designed to embed LP solver in BCP, but BCP didn't work with it at first.
- Quickly became popular development target.
  - CPLEX, SoPlex, CLP, DyLP, GLPK followed quickly.
  - CBC, FortMP, Mosek, SYMPHONY more recent.
  - All except OSL, Volume, CLP contributed by non-IBM developers.

Capabilities

- Read and write LP or MIP from MPS or CPLEX LP file or construct in memory (cf CoinUtils).
- Invoke presolver.
- Load problem in embedded solver.
- Set solver parameters.
- Call embedded solver on LP (relaxation).
- Modify problem representation stored in solver.
- Interact with CGL to generate cutting planes that cut off given solution.
- Resolve LP using hot start.
- Call embedded MIP solver using LP solution at root node.
- Extract solution data.
- Extract raw problem pointer to bypass OSI.
- Manage multiple problem instances.

Input Data

The following are the input data needed to describe an instance of the uncapacitated facility location problem (UFL):

Data

- a set of depots $N = \{1, ..., n\}$, a set of clients $M = \{1, ..., m\}$,
- the transportation cost c_{ij} to service client i from j,
- the fixed cost f_j for using depot j

Variables

- x_{ij} is the amount of the demand for client i satisfied from depot j
- y_j is 1 if the depot is used, 0 otherwise

Mathematical Programming Formulation

The following is a mathematical programming formulation of the UFL

UFL Formulation

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} + \sum_{j \in N} f_j y_j && (1) \\
\text{subject to} \quad & \sum_{j \in N} x_{ij} = d_i \quad \forall i \in M, && (2) \\
& \sum_{i \in M} x_{ij} \leq \Big(\sum_{i \in M} d_i\Big) y_j \quad \forall j \in N, && (3) \\
& y_j \in \{0, 1\} \quad \forall j \in N && (4) \\
& 0 \leq x_{ij} \leq d_i \quad \forall i \in M, j \in N && (5)
\end{aligned}
$$

Dynamically Generated Valid Inequalities

- Given the current LP solution, x^*, y^*, this method searches for violated logical constraints of the form

$$x_{ij} - d_j y_j \leq 0.$$

- To generate such inequalities dynamically, get the current solution.
- Then check if

$$x_{ij}^* - d_j y_j^* > \epsilon, \forall i \in M, j \in N.$$

- Also generate inequalities valid for generic MILPs.
- If a violation is found, add the constraint to the current LP relaxation.

Tightening the Initial Formulation

Here is the basic loop for tightening the initial formulation using the dynamically generated inequalities from the previous slide.

Solving the LP relaxation

1. Form the initial LP relaxation and solve it to obtain $(\hat{x}, \hat{y})$.
2. Iterate
   1. Try to generate a valid inequality violated by $(\hat{x}, \hat{y})$.
   2. Optionally, try to generate an improved feasible solution by rounding $\hat{y}$.
   3. Solve the current LP relaxation of the initial formulation to obtain $(\hat{x}, \hat{y})$.
   4. If $(\hat{x}, \hat{y})$ is feasible, STOP. Otherwise, go to Step 1.

Data Members

C++ Class

```
class UFL {
private:
  OsiSolverInterface * si;
  double * trans_cost; //c[i][j] -> c[xindex(i,j)]
  double * fixed_cost; //f[j]
  double * demand;     //d[j]
  int M;               //number of clients (index on i)
  int N;               //number of depots  (index in j)
  double total_demand; //sum{j in N} d[j]
  int *integer_vars;

  int xindex(int i, int j) {return i*N + j;}
  int yindex(int j)        {return M*N + j;}
};
```

Methods

C++ Class

```
class UFL {
public:
  UFL(const char* datafile);
  ~UFL();
  void create_initial_model();
  double tighten_initial_model(ostream *os = &cout);
  void solve_model(ostream *os = &cout);
};
```

Cut Generator Library

- A collection of cutting-plane generators and management utilities.
- Interacts with OSI to inspect problem instance and solution information and get cuts added to the problem.
- Cuts include:
  - Combinatorial cuts: AllDifferent, Clique, KnapsackCover, OddHole
  - Flow cover cuts
  - Lift-and-project cuts
  - Mixed integer rounding cuts
  - General strengthening: DuplicateRows, Preprocessing, Probing, SimpleRounding

COIN LP Solver

- High-quality, efficient LP solver.
- Simplex and barrier algorithms. QP with barrier algorithm.
- Fine control through OSI or direct calls.
- Tight integration with CBC (COIN-OR Branch and Cut MIP solver).

COIN Branch and Cut Solver

- High-quality, efficient branch-and-cut solver.
- LP-based relaxations.
- Calls LP solver via OSI or uses CLP directly.
- Uses CGL to generate cuts.

The initialize_solver() Method

Intializing the LP solver

```
#if defined(COIN_USE_CBC)
#include "OsiCbcSolverInterface.hpp"
typedef OsiCbcSolverInterface OsiXxxSolverInterface;
% #include "CbcModel.hpp"
#elif defined(COIN_USE_CPX)
#include "OsiCpxSolverInterface.hpp"
typedef OsiCpxSolverInterface OsiXxxSolverInterface;
#endif

OsiSolverInterface* UFL::initialize_solver() {
  OsiXxxSolverInterface* si =
    new OsiXxxSolverInterface();

  return si;
}
```

The `create_initial_model()` Method

Creating Rim Vectors

```
CoinIotaN(integer_vars, N, M * N);
CoinFillN(col_lb, n_cols, 0.0);

int i, j, index;

for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    index              = xindex(i,j);
    objective[index] = trans_cost[index];
    col_ub[index]     = demand[i];
  }
}
CoinFillN(col_ub + (M*N), N, 1.0);
CoinDisjointCopyN(fixed_cost, N, objective + (M * N));
```

The create_initial_model() Method

Creating the Constraint Matrix

```
CoinPackedMatrix * matrix =
   new CoinPackedMatrix(false,0,0);

matrix->setDimensions(0, n_cols);
for (i = 0; i < M; i++) {  //demand constraints:
  CoinPackedVector row;
  for (j = 0; j < N; j++) row.insert(xindex(i,j), 1.0);
  matrix->appendRow(row);
}

for (j = 0; j < N; j++) {  //linking constraints:
  CoinPackedVector row;
  row.insert(yindex(j), -1.0 * total_demand);
  for (i = 0; i < M; i++) row.insert(xindex(i,j), 1.0);
  matrix->appendRow(row);
}
```

The create_initial_model() Method

Loading the Problem and Solving the LP Relaxation

```
si->loadProblem(*matrix, col_lb, col_ub,
                objective, row_lb, row_ub);
si->initialSolve();
```

The `tighten_initial_model()` Method

Tightening the Relaxation—Custom Cuts

```
const double* sol = si->getColSolution();
int newcuts = 0, i, j, xind, yind;
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    xind = xindex(i,j);  yind = yindex(j);

    if (sol[xind] - (demand[i] * sol[yind]) >
        tolerance) { // violated constraint
      CoinPackedVector cut;
      cut.insert(xind,  1.0);
      cut.insert(yind, -1.0 * demand[i]);
      si->addRow(cut,  -1.0 * si->getInfinity(), 0.0);
      newcuts++;
    }
  }
}
```

The tighten_initial_model() Method

Tightening the Relaxation—CGL Cuts

```
OsiCuts cutlist;
si->setInteger(integer_vars, N);
CglGomory * gomory = new CglGomory;
gomory->setLimit(100);
gomory->generateCuts(*si, cutlist);
CglKnapsackCover * knapsack = new CglKnapsackCover;
knapsack->generateCuts(*si, cutlist);
CglSimpleRounding * rounding = new CglSimpleRounding;
rounding->generateCuts(*si, cutlist);
CglOddHole * oddhole = new CglOddHole;
oddhole->generateCuts(*si, cutlist);
CglProbing * probe = new CglProbing;
probe->generateCuts(*si, cutlist);
si->applyCuts(cutlist);
```

The solve_model() Method

Calling the MIP Solver

```
si->setInteger(integer_vars, N);

si->branchAndBound();
if (si->isProvenOptimal()) {
  const double * solution = si->getColSolution();
  const double * objCoeff = si->getObjCoefficients();
  print_solution(solution, objCoeff, os);
}
else
  cerr << "B&B failed to find optimal" << endl;
return;
```

Current OSI

Does basic things well, but. . .

- Doing many incremental updates can be inefficient.
- More complex operations require non-portable direct solver interaction.
- Feature creep has caused several SIs to lose synch with base class.
  - CLP interface is most feature compete, but even there, direct access is sometimes needed.
  - CLP—CBC interaction.
  - Parameter setting and messaging.
- Model representation is tied to solver.
- SI layer is responsible for efficiency (e.g., caching).

What to do?

OSI Version 2

- Lots of design work, but
- not much code yet.

One Challenge for Open Source

Developers with day jobs. . .

Some Ideas for OSI2

- Separate model maintenance from solver configuration and direction.
- C++ “best practices”
- C-callable layer (autogenerated?).
- Keep most of the computational load in the base class.

Possible Partial Solution

The Optimization Services project proposes to provide some of what is needed:

- XML representation.
- Model construction/modification API and internal data structures.
- Solver management API.
- Solution extraction API.

These features may form the basis of the “user-facing” side of the API.

Where to go for More Information

`<project>` is one of `Osi`, `Cgl`, `Clp`, `Cbc`, etc.

- Project home pages: `https://projects.coin-or.org/<project>` (Trac pages).
- Documentation: `http://www.coin-or.org/Doxygen/<project>` (Doxygen), `http://www.coin-or.org/Clp/userguide/`, `http://www.coin-or.org/Cbc/userguide/`
- Mailing lists: `http://list.coin-or.org` (see `coin-discuss`, `coin-osi-devel`, `cgl`, `coin-lpsolver`—note lists will be reorganized soon).

Thanks to Matt Galati (SAS) and Ted Ralphs (Lehigh) for the example code.