Building a Custom Solver with the COIN-OR Branch, Cut, and Price Frameworks

Ted Ralphs and Menal Guzelsoy
Lehigh University

László Ladányi
IBM T. J. Watson Research Center

Matthew Saltzman
Clemson University

CORS/INFORMS Joint Int’l Meeting, Banff, Alberta, Canada, Sunday, May 16, 2004
Agenda

• Introduction to BCP Frameworks

• Introduction to SYMPHONY
  – Callable library API
  – OSI interface
  – User callbacks

• Introduction to COIN/BCP
  – Basic concepts
  – Design of COIN/BCP
  – User API
  – Example
Concept

• **Concept**: Provide a *framework* in which the user has only to define the core relaxation, along with classes of dynamically generated variables and constraints.

• **SYMPHONY** and **COIN/BCP** are two frameworks that can be used to implement solvers for mixed-integer programs.

• They have similar design concepts and state-of-the-art implementations of *branch, cut and price*.

• **SYMPHONY**
  – is a callable library with C and OSI interfaces,
  – works out of the box as a generic MIP solver,
  – employs callbacks for customization,
  – is a bit easier for the novice.

• **COIN/BCP**
  – has more power for implementing column generation and integrating cut and column generation,
  – employs C++ inheritance for customization,
  – is a bit more difficult to learn.
SYMPHONY Overview

• Description
  – A callable library for solving mixed-integer linear programs with a wide variety of customization options.
  – Fully integrated with the Computational Infrastructure for Operations Research (COIN-OR) libraries (soon to be in the repository).
  – Outfitted as a generic MILP solver, with cut generation from the CGL.
  – Extensive documentation available.
  – Source can be downloaded from www.branchandcut.org

• SYMPHONY Solvers
  - Generic MILP
  - Traveling Salesman Problem
  - Vehicle Routing Problem
  - Mixed Postman Problem
  - Set Partitioning Problem
  - Matching Problem
  - Network Routing
Supported Formats and Architectures

- **Input formats**
  - MPS (COIN-OR parser)
  - GMPL/AMPL (GLPK parser)
  - User defined

- **Output/Display formats**
  - Text
  - IGD
  - VbcTool

- **Supported architectures**
  - Single-processor Linux, Unix, or Windows
  - Distributed memory parallel (message-passing)
  - Shared memory parallel (OpenMP)
C Callable Library

- Primary subroutines
  - sym_open_environment()
  - sym_parse_command_line()
  - sym_load_problem()
  - sym_find_initial_bounds()
  - sym_solve()
  - sym_mc_solve()
  - sym_resolve()
  - sym_close_environment()

- Auxiliary subroutines
  - Accessing and modifying problem data
  - Accessing and modifying parameters
  - User callbacks
Implementing a MILP Solver with SYMPHONY

- Using the callable library, we only need a few lines to implement a solver.
- The file name and other parameters are specified on the command line.
- The code is exactly the same for all architectures, even parallel.
- Command line would be

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

  ```
  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);
  }
  ```
• For each method in OSI, SYMPHONY has a corresponding method.
• The OSI interface is implemented as wrapped C calls.
• The constructor calls `sym_open_environment()` and the destructor calls `sym_close_environment()`.
• The OSI `initialSolve()` method calls `sym_solve()`.
• The OSI `resolve()` method calls `sym_resolve()`.
• There is also a multicriteria solve method.

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

• The main avenues for advanced customization are the parameters and the user callback subroutines.

• There are more than 50 callbacks and over 100 parameters.

• The user can override SYMPHONY’s default behavior in a variety of ways.
  – Custom input
  – Custom displays
  – Branching
  – Cut/column generation
  – Cut pool management
  – Search and diving strategies
  – LP management
Resources

• **SYMPhONY 5.0** will be released soon and can be downloaded from

  \[\text{http://www.branchandcut.org/SYMPhONY}\]

• SYMPHONY is well-documented, and includes sample codes and tutorials.

• There is a mailing list, but it is better to just send me e-mail directly to me.

• SYMPHONY will be added to the COIN-OR repository once it moves to INFORMS.
COIN/BCP Overview

- COIN/BCP is focused on the implementation of full-blown branch, cut, and price algorithms.
- The framework centers around the management of classes of dynamically generated cut and variables, generically called objects.
- Subproblems are composed of dynamic lists of these objects.
- The goal is to keep the lists as small as possible, while not sacrificing bound quality.
- Defining a class of objects consists of defining methods for
  - generating new objects, given the primal/dual solution to the current LP relaxation,
  - representing the object (for storage and/or sharing), and
  - adding objects to a given LP relaxation.
Getting Started

- The source can be obtained from www.coin-or.org as “tarball” or using CVS.

- Platforms/Requirements
  - Linux, gcc 2.95.3/2.96RH/3.2/3.3
  - Windows, Visual C++, CygWin make (untested)
  - Sun Solaris, gcc 2.95.3/3.2 or SunWorkshop C++
  - AIX gcc 2.95.3/3.3
  - Mac OS X

- Editing the Makefiles
  - Makefile.location
  - Makefile.<operating system>

- Make the necessary libraries. They’ll be installed in ${CoinDir}/lib.
  - Change to appropriate directory and type make.
COIN/BCP Modules

- The COIN/BCP library is divided into three basic modules:

  - **Tree Manager** Controls overall execution by maintaining the search tree and dispatching subproblems to the node processors.
  - **Node Processor** Perform processing and branching operations.
  - **Object Generation** Generate objects (cuts and/or variables).

- The division into separate modules is what allows the code to be parallelizable.
The User API

- The user API is implemented via a C++ class hierarchy.
- To develop an application, the user must derive the appropriate classes and override the appropriate methods.
- Classes for customizing the behavior of the modules
  - BCP_tm_user
  - BCP_lp_user
  - BCP_cg_user
  - BCP_vg_user
- Classes for defining user objects
  - BCP_cut
  - BCP_var
  - BCP_solution
- Allowing COIN/BCP to create instances of the user classes.
  - The user must derive the class USER_initialize.
  - The function BCP_user_init() returns an instance of the derived initializer class.
Objects in COIN/BCP

- Most application-specific methods are related to handling of objects.
- Since representation is independent of the current LP, the user must define methods to add objects to a given subproblem.
- For parallel execution, the objects need to be packed into (and unpacked from) a buffer.

- Object Types
  - Core objects are objects that are active in every subproblem (BCP_xxx_core).
  - Indexed objects are extra objects that can be uniquely identified by an index (such as the edges of a graph) (BCP_xxx_indexed).
  - Algorithmic objects are extra objects that have an abstract representation (BCP_xxx_algo).
### Forming the LP Relaxations in COIN/BCP

The current LP relaxation looks like this:

<table>
<thead>
<tr>
<th></th>
<th>core vars</th>
<th>extra vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>core cuts</td>
<td>core matrix</td>
<td></td>
</tr>
<tr>
<td>extra cuts</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reason for this split: efficiency.
COIN/BCP Methods: Initialization

Solver

Initialization
(Tree Manager)

- BCP_user_init()
- xx_init()
- initialize_core()
- create_root()
- pack_module_data()

Create and initialize the user’s data structures
Set the core and extra variables and cuts
Send data to the modules
COIN/BCP Methods: Steady State

**Tree Manager**
- (un)pack_xxx_algo()
- display_feasible_solution()
- compare_tree_nodes()

**Cut Generator**
- unpack_module_data()
- generate_cuts()
- pack_cut_algo()

**LP Solver**
- unpack_module_data()
- initialize_search_tree_node()
- See the solver loop figure

**Variable Generator**
- unpack_module_data()
- generate_vars()
- pack_var_algo()
COIN/BCP Methods: Node Processing Loop

- modify_LP_parameters()
  - test_feasibility()
  - generate_heuristic_solution()
  - pack_{primal/dual}_solution()
  - unpack_{var/cut}_algo()
  - generate_{vars/cuts}_in_lp()
  - compare_{vars/cuts}()
  - vars_to_cols() / cuts_to_rows()
  - logical_fixing()
  - select_branching_candidates()
  - compare_branching_candidates()
  - set_actions_for_children()
  - purge_slack_pool()

Possible fathoming
Send primal and dual solutions to CG and VG
Generating and comparing cuts and variables
Strong branching functions
Parameters and using the finished code

- Create a parameter file
- Run your code with the parameter file name as an argument (command line switches will be added).
- BCP_ for COIN/BCP’s parameters
- Defined and documented in BCP_tm_par, BCP_lpar, etc.
- Helper class for creating your parameters.
- Output controlled by verbosity parameters.
Example: Uncapacitated Facility Location

- **Data**
  - a set $N$ of facilities and a set $M$ of clients,
  - transportation cost $c_{ij}$ to service client $i$ from depot $j$,
  - fixed cost $f_j$ for using depot $j$, and
  - the demand of $d_i$ of client $i$.

- **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

$$\begin{align*}
\min & \sum_{i \in M} \sum_{j \in N} \frac{c_{ij}}{d_i} x_{ij} + \sum_{j \in N} f_j y_j \\
\text{s.t.} & \sum_{j \in N} x_{ij} = d_i \quad \forall i \in M, \\
& \sum_{i \in M} x_{ij} \leq (\sum_{i \in M} d_i)y_j \quad \forall j \in N, \\
& y_j \in \{0, 1\} \quad \forall j \in N \\
& 0 \leq x_{ij} \leq d_i \quad \forall i \in M, j \in N
\end{align*}$$
UFL: Solution Approach

• The code for this example is available at

  http://sagan.ie.lehigh.edu/coin/uflBCP.tar.gz

• We use a simple branch and cut scheme.

• We dynamically generate the following class disaggregated logical cuts

  \[ x_{ij} \leq d_j y_j, \forall i \in M, j \in N \]  

  (1)

• These can be generated by complete enumeration.

• The indices \( i \) and \( j \) uniquely identify the cut., so we will use this to create the packed form.

• The core relaxation will consist of the LP relaxation.
UFL: User classes

User classes and methods

- **UFL_init**
  - tm_init()
  - lp_init()

- **UFL_lp**
  - unpack_module_data()
  - pack_cut_algo()
  - unpack_cut_algo()
  - generate_cuts_in_lp()
  - cuts_to_rows()

- **UFL_tm**
  - read_data()
  - initialize_core()
  - pack_module_data()

- **UFL_cut**
UFL: Initialization Methods

USER_initialize * BCP_user_init()
{
    return new UFL_init;
}

BCP_lp_user *
UFL_init::lp_init(BCP_lp_prob& p)
{
    return new UFL_lp;
}

BCP_tm_user * UFL_init::tm_init(BCP_tm_prob& p, const int argnum,
                         const char * const * arglist)
{
    UFL_tm* tm = new UFL_tm;
    tm->tm_par.read_from_file(arglist[1]);
    tm->lp_par.read_from_file(arglist[1]);
    return tm;
}
COIN/BCP Buffers

- One construct that is pervasive in COIN/BCP is the `BCP_buffer`.
- A `BCP_buffer` consists of a character string into which data can be packed for storage or communication (parallel code).
- The usual way of adding data to a buffer is to use the `pack()` method.
- The pack method returns a reference to the buffer, so that multiple calls to `pack()` can be strung together.
- To pack integers `i` and `j` into a buffer and then unpack from the same buffer again, the call would be:

```plaintext
int i = 0, j = 0;
BCP_buffer buf;

buf.pack(i).pack(j);
buf.unpack(i).unpack(j);
```
UFL: Module Data

- Because COIN/BCP is a parallel code, there is no shared memory between modules.

- The `pack_module_data()` and `unpack_module_data()` methods allow instance data to be broadcast to other modules.

- In the UFL, the data to be broadcast consists of the number of facilities \(N\), the number of clients \(N\), and the demands.

- Here is what the pack and unpack methods look like.

```c
void UFL_tm::pack_module_data(BCP_buffer& buf, BCP_process_t pty)
{
    lp_par.pack(buf);
    buf.pack(M).pack(N).pack(demand,M);
}

void UFL_lp::unpack_module_data(BCP_buffer& buf) {
    lp_par.unpack(buf);
    buf.unpack(M).unpack(N).unpack(demand,M).unpack(capacity,N);
}
```
UFL: Initializing the Core

- The core is specified as an instance of the `BCP_lp_relax` class, which can be constructed from
  - either a vector of `BCP_rows` or `BCP_cols`, and
  - a set of rim vectors.
- In the `initialize_core()` method, the user must also construct a vector of `BCP_cut_core` and `BCP_var_core` objects.
UFL: Initializing the Solver Interface

• In the BCP_lp_user class, we must initialize the solver interface to let COIN/BCP know what solver we want to use.

• Here is what that looks like:

```cpp
OsiSolverInterface* UFL_lp::initialize_solver_interface(){
#if COIN_USE_OSL
    OsiOslSolverInterface* si = new OsiOslSolverInterface();
#endif
#if COIN_USE_CPX
    OsiCpxSolverInterface* si = new OsiCpxSolverInterface();
#endif
#if COIN_USE_CLP
    OsiClpSolverInterface* si = new OsiClpSolverInterface();
#endif
    return si;
}
```
**UFL: Cut Class**

class UFL_cut : public BCP_cut_algo{
public:
    int i,j;
public:
    UFL_cut(int ii, int jj):
        BCP_cut_algo(-1 * INF, 0.0), i(ii), j(jj) {
    }
    UFL_cut(BCP_buffer& buf):
        BCP_cut_algo(-1 * INF, 0.0), i(0), j(0) {
            buf.unpack(i).unpack(j);
        }
    void pack(BCP_buffer& buf) const;
};

inline void UFL_cut::pack(BCP_buffer& buf) const{
    buf.pack(i).pack(j);
}
UFL: Generating Cuts

- To find violated cuts, we simply enumerate, as in this code snippet.

```c++
double violation;
vector<pair<int,int>> cut_v;
map<double,int> cut_violation; //map keeps violations sorted
map<double,int>::reverse_iterator it;

for (i = 0; i < M; i++){
    for (j = 0; j < N; j++){
        xind = xindex(i,j);
        yind = yindex(j);
        violation = lpres.x()[xind]-(demand[i]*lpres.x()[yind]);
        if (violation > tolerance){
            cut_v.push_back(make_pair(i,j));
            cut_violation.insert(make_pair(violation,cutindex++));
        }
    }
}
```
Next, we pass the most violated cuts back to COIN/BCP.

```cpp
//Add the xxx most violated ones.
maxcuts = min((int)cut_v.size(),
              lp_par.entry(UFL_lp_par::UFL_maxcuts_iteration));
it = cut_violation.rbegin();
while(newcuts<maxcuts){
    cutindex = it->second;
    violation = it->first;
    new_cuts.push_back(new UFL_cut(cut_v[cutindex].first,
                                   cut_v[cutindex].second));
    newcuts++;
    it++;
}
```
**UFL: Adding Cuts to the LP**

- Here is the `cuts_to_rows` function that actually generates the rows to be added to the LP relaxation.

```cpp
void UFL_lp::cuts_to_rows(const BCP_vec<BCP_var*>& vars,
                          BCP_vec<BCP_cut*>& cuts,
                          BCP_vec<BCP_row*>& rows,
                          const BCP_lp_result& lpres,
                          BCP_object_origin origin, bool allow_multiple){
    const int cutnum = cuts.size();
    rows.reserve(cutnum);
    for (int c = 0; c < cutnum; ++c) {
        UFL_cut* mcut = dynamic_cast<const UFL_cut*>(cuts[c]);
        if (mcut != 0){
            CoinPackedVector cut;
            cut.insert(xindex(mcut->i,mcut->j), 1.0);
            cut.insert(yindex(mcut->j), -1.0 * demand[mcut->i]);
            rows.push_back(new BCP_row(cut,-1.0 * INF, 0.0));
        }
    }
}
```
Resources

• Documentation
  – There is a user’s manual for COIN/BCP, but it is out of date.
  – The most current documentation is in the source code—don’t be afraid to use it.

• Other resources
  – There are several mailing lists on which to post questions and we make an effort to answer quickly.
  – Also, there is a lot of good info at www.coin-or.org.
  – There are some basic tutorials and other information, including the example you saw today at sagan.ie.lehigh.edu/coin/.

• There is a user’s meeting today at 1:00.

• There are also two other sessions revolving around COIN software.
Final advice

Use the source, Luke...

...and feel free to ask questions either by email or on the discussion list.