COIN-OR: Software Tools for Implementing Custom Solvers

Ted Ralphs
Lehigh University

László Ladányi
IBM T. J. Watson Research Center

Matthew Saltzman
Clemson University

Institute for Operations Research and Management Science Annual Conference, October 19, 2003

Agenda

- **Overview of COIN-OR**
- Overview of COIN-OR branch, cut, and price toolbox
  - BCP
  - OSI
  - CGL
  - CLP
  - VOL
- Developing an application
  - Basic concepts
  - Design of BCP
  - User API
- Example

What is COIN-OR?

- The COIN-OR Project
  - A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
  - An initiative promoting the development and use of interoperable, open-source software for operations research.
  - Soon to become a non-profit corporation known as the COIN-OR Foundation
- The COIN-OR Repository
  - A library of interoperable software tools for building optimization codes, as well as a few stand alone packages.
  - A venue for peer review of OR software tools.
  - A development platform for open source projects, including a CVS repository.

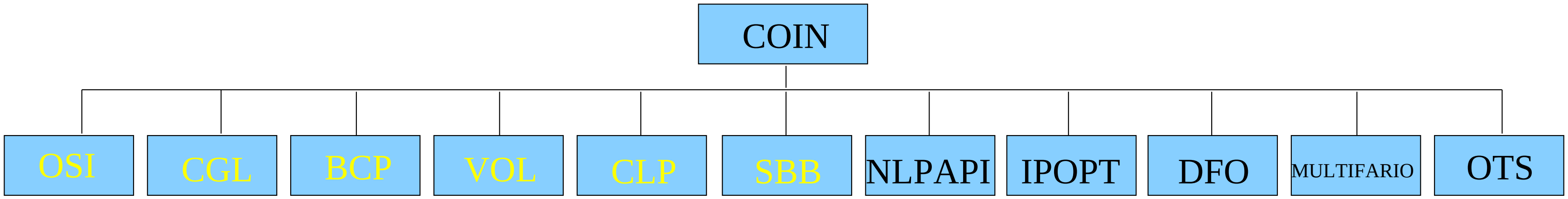
What is Open Source Development?

- *Open source development* is a coding paradigm in which development is done in a cooperative and distributed fashion.
- Strictly speaking, an open source license must satisfy the requirements of the *Open Source Definition*.
- A license cannot call itself "open source" until it is approved by the Open Source Initiative.
- Basic properties of an open source license
  - Access to source code.
  - The right to redistribute.
  - The right to modify.
- The license may require that modifications also be kept open.

Our Agenda

- Accelerate the pace of research in computational OR.
  - Reuse instead of reinvent.
  - Reduce development time and increase robustness.
  - Increase interoperability (standards and interfaces).
- Provide for software what the open literature provides for theory.
  - Peer review of software.
  - Free distribution of ideas.
  - Adherence to the principles of good scientific research.
- Define standards and interfaces that allow software components to interoperate.
- Increase synergy between various development projects.
- Provide robust, open-source tools for practitioners.

Components of the COIN-OR Library



- Branch, cut, price toolbox
  - OSI: Open Solver Interface
  - CGL: Cut Generator Library
  - BCP: Branch, Cut, and Price Library
  - VOL: Volume Algorithm
  - CLP: COIN-OR LP Solver
  - SBB: Simple Branch and Bound
  - COIN: COIN-OR Utility Library
- Stand-alone components
  - IPOPT: Interior Point Optimization
  - NLPAPI: Nonlinear Solver interface
  - DFO: Derivative Free Optimization
  - MULTIFARIO: Solution Manifolds
  - OTS: Open Tabu Search

Agenda

- Overview of COIN-OR
- **Overview of COIN-OR branch, cut, and price toolbox**
  - BCP
  - OSI
  - CGL
  - CLP
  - VOL
- Developing an application
  - Basic concepts
  - Design of BCP
  - User API
- Example

BCP Overview

- Concept: Provide a *framework* in which the user has only to define the core relaxation, along with classes of dynamically generated variables and constraints.
  - Branch and bound ⇒ core relaxation only
  - Branch and cut ⇒ core relaxation plus constraints
  - Branch and price ⇒ core relaxation plus variables
  - Branch, cut, and price ⇒ the whole caboodle
- Existing frameworks
  - SYMPHONY (parallel, C)
  - COIN/BCP (parallel, C++)
  - ABACUS (sequential, C++)
- Components
  - Framework (BCP)
  - LP Solver (OSI)
  - Cut Generator (CGL)
  - Utilities (COIN)

OSI Overview

Uniform interface to LP solvers, including:

- CLP (COIN-OR)
- CPLEX (ILOG)
- DyLP (BonsaiG LP Solver)
- GLPK (GNU LP Kit)
- OSL (IBM)
- SoPlex (Konrad-Zuse-Zentrum für Informationstechnik Berlin)
- Volume (COIN-OR)
- XPRESS (Dash Optimization)
- MOSEK (under construction)

CGL Overview

- Collection of cut generation routines integrated with OSI.
- Intended to provide robust implementations, including computational tricks not usually published.
- Currently includes:
  - Simple rounding cut
  - Gomory cut
  - Knapsack cover cut
  - Rudimentary lift-and-project cut
  - Odd hole cut
  - Probing cut

VOL Overview

- Generalized subgradient optimization algorithm.
- Compatible with branch, cut, and price:
  - provides approximate primal and dual solutions,
  - provides a valid lower bound (feasible dual solution), and
  - provides a method for warm starting.

CLP Overview

- A full-featured, open source LP solver.
- Has interfaces for primal, dual, and network simplex.
- Can be accessed through the OSI.
- Reasonably robust and fast.

SBB Overview

- A lightweight generic MIP solver.
- Uses OSI to solve the LP relaxations.
- Uses CGL to generate cuts.
- Optimized for CLP.

COIN Utility Library Overview

- Contains classes for
  - Storage and manipulation of sparse vectors and matrices.
  - Factorization of sparse matrices.
  - Storage of solver warm start information.
  - Message handling.
  - Reading/writing of MPS files.
  - Other utilities (simultaneous sorting, timing, . . .).
- These are the classes common to many of the other algorithms.

Agenda

- Overview of COIN-OR
- Overview of COIN-OR branch, cut, and price toolbox
  - BCP
  - OSI
  - CGL
  - CLP
  - VOL
- **Developing an application**
  - Basic concepts
  - Design of BCP
  - User API
- Example

Basic Concepts

- We consider problem P:

$$\begin{array}{llll} \min & c^T x & & \\ \text{s.t.} & Ax & \leq & b \\ & x_i & \in & \mathbb{Z} \quad \forall\, i \in I \end{array}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$.

- Let $\mathcal{P} = \mathrm{conv}\{x \in \mathbb{R}^n : Ax \leq b,\ x_i \in \mathbb{Z}\ \forall\, i \in I\}$.
- Basic Algorithmic Approach
  - Use *LP relaxations* to produce *lower bounds*.
  - *Branch* using hyperplanes.
  - The LP relaxations are built up from a core relaxation with dynamically generated *objects* (variables and constraints).

Object Generation

- The efficiency of BCP depends heavily on the size (number of rows and columns) and tightness of the LP relaxations.
- Tradeoff
  - Small LP relaxations ⇒ faster LP solution.
  - Big LP relaxations ⇒ better bounds.
- The goal is to keep relaxations small while not sacrificing bound quality.
- We must be able to easily move constraints and variables in and out of the *active* set.
- This means dynamic generation and deletion.
- 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`.

BCP Modules

- The 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

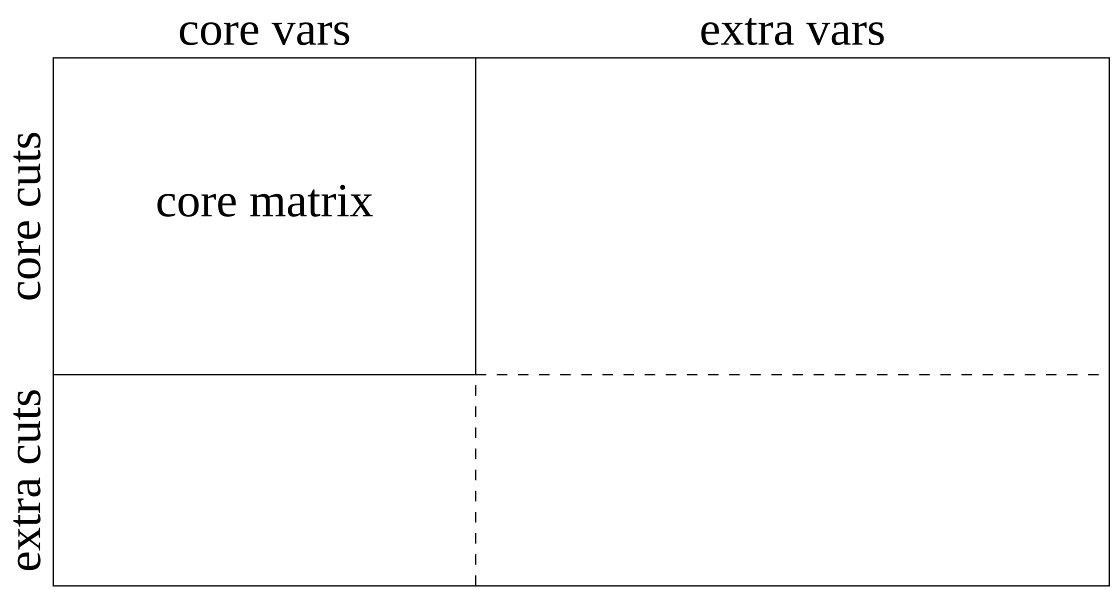
- The user API is implemented via a C++ class hierarchy.
- To develop an application, the user must derive the appropriate classes override the appropriate methods.
- Classes for customizing the behavior of the modules
  - BCP_tm_user
  - BCP_lp_user
  - BCP_cg_user
  - BCP_vg_user
- Classes for defining user objects
  - BCP_cut
  - BCP_var
  - BCP_solution
- Allowing 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 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).

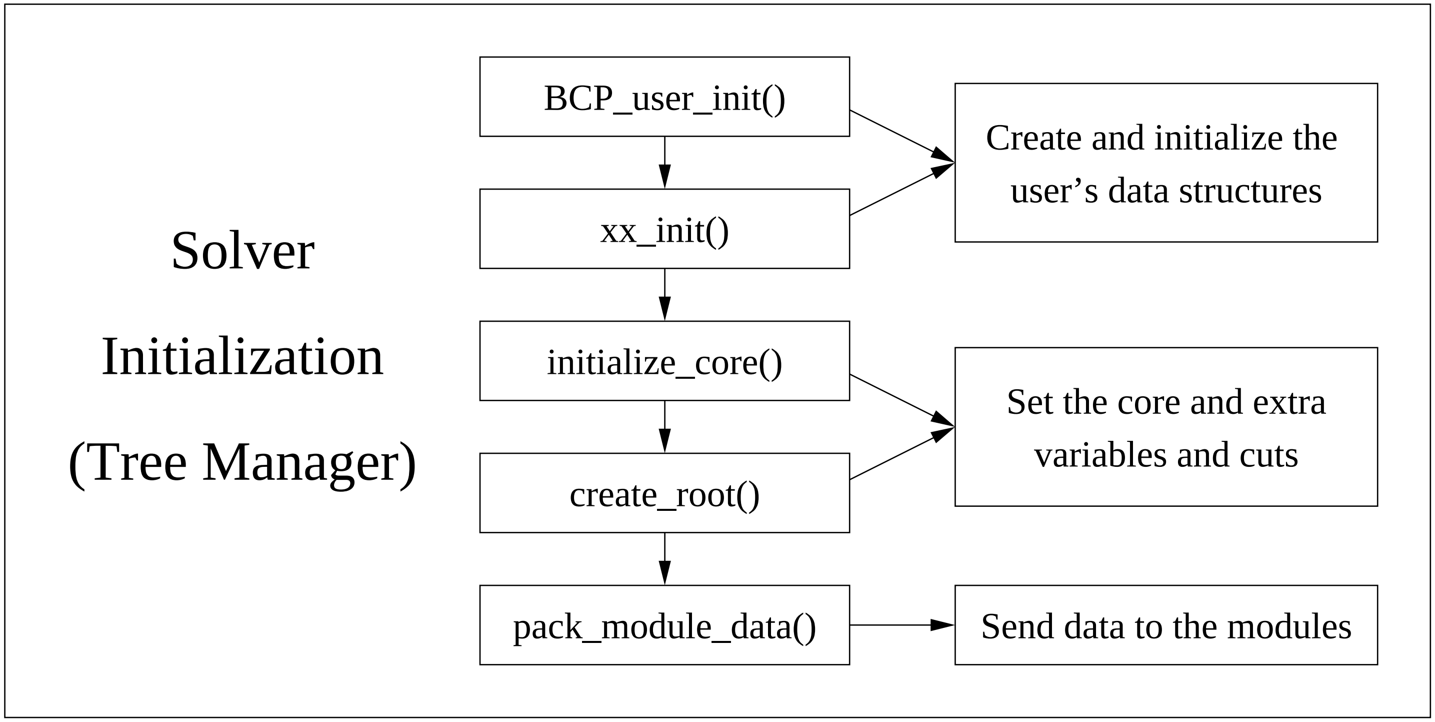
Forming the LP Relaxations in BCP

The current LP relaxation looks like this:

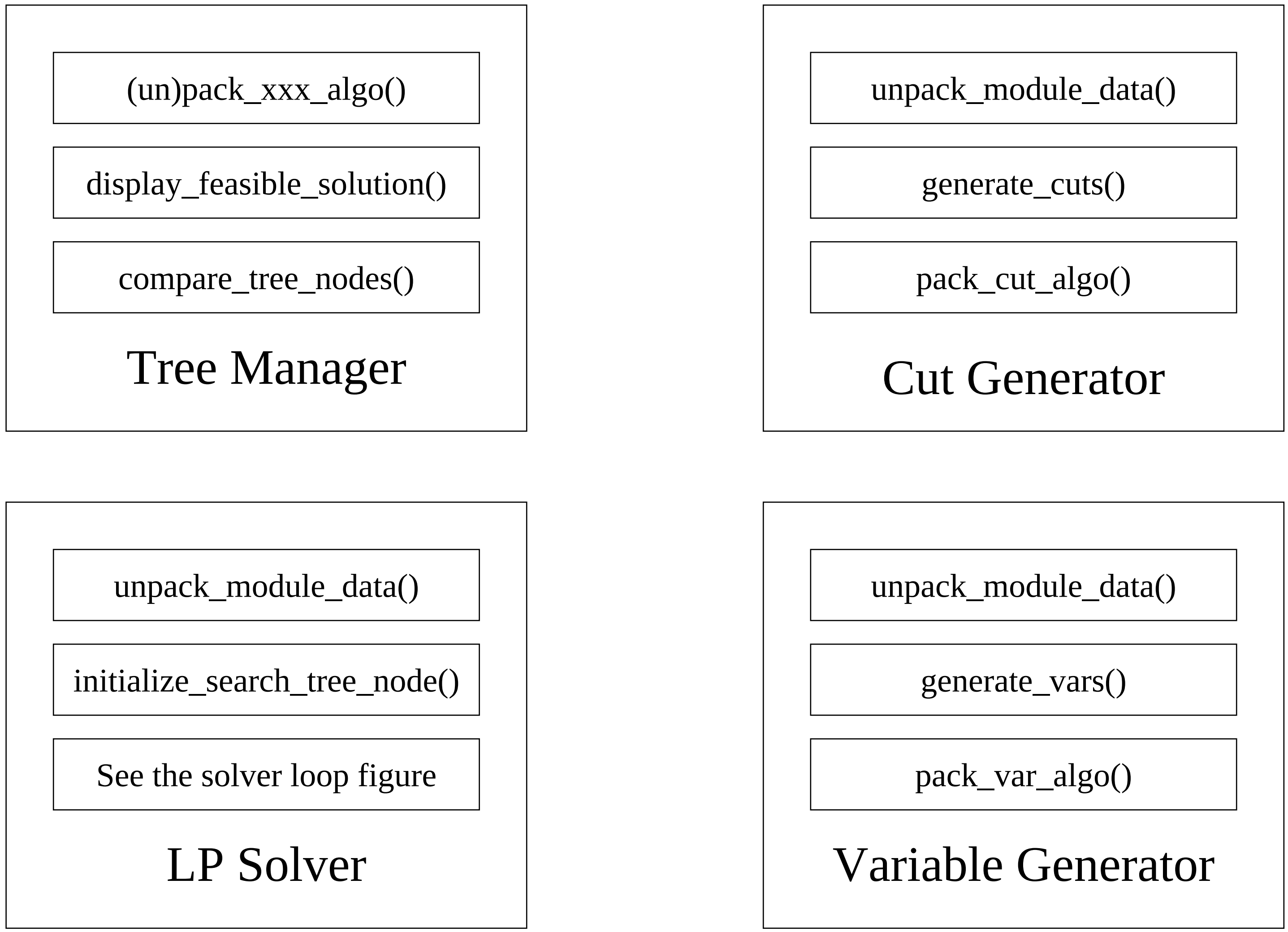


Reason for this split: efficiency.

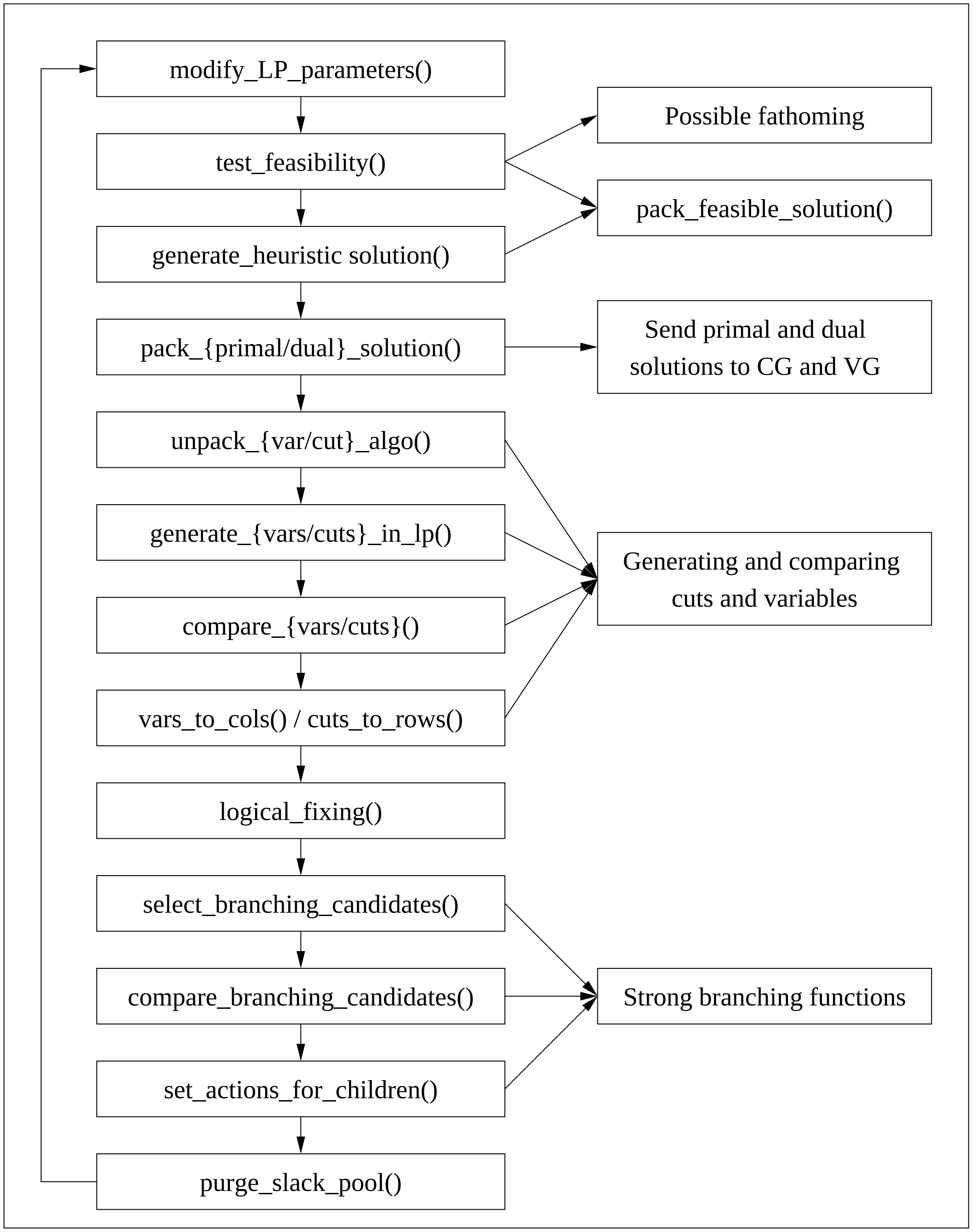
BCP Methods: Initialization



BCP Methods: Steady State



BCP Methods: Node Processing Loop



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 BCP's parameters
- Defined and documented in BCP_tm_par, BCP_lp_par, etc.
- Helper class for creating your parameters.
- Output controlled by verbosity parameters.

Agenda

- Overview of COIN-OR
- Overview of COIN-OR branch, cut, and price toolbox
  - BCP
  - OSI
  - CGL
  - CLP
  - VOL
- Developing an application
  - Basic concepts
  - Design of BCP
  - User API
- **Example**

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

$$
\begin{aligned}
\min \sum_{i \in M} \sum_{j \in N} \frac{c_{ij}}{d_i} x_{ij} + \sum_{j \in N} f_j y_j & & \\
s.t. \quad \sum_{j \in N} x_{ij} = d_i & & \forall i \in M, \\
\sum_{i \in M} x_{ij} \leq \left(\sum_{i \in M} d_i\right) y_j & & \forall j \in N, \\
y_j \in \{0, 1\} & & \forall j \in N \\
0 \leq x_{ij} \leq d_i & & \forall i \in M, j \in N
\end{aligned}
$$

UFL: Solution Approach

- We use a simple branch and cut scheme.
- We dynamically generate the following class disaggregated logical cuts

$$x_{ij} <= d_j y_j, \; \forall i \in M, j \in N \tag{1}$$

- 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;
}
```

BCP Buffers

- One construct that is pervasive in 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:

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

buf.pack(i).pack(j);
buf.unpack(i).unpack(j);
```

UFL: Module Data

- Because BCP is a parallel code, there is no shared 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.

```
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 BCP know what solver we want to use.
- Here is what that looks like:

```
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.

```
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++));
      }
   }
}
```

UFL: Constructing Cuts

- Next, we pass the most violated cuts back to BCP.

```
//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.

```
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 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 Monday at 12:00 in International Ballroom A.
- There are three other sessions revolving around COIN software, including a tutorial on OSI.

Final advice

Use the source, Luke...

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