SYMPHONY:

Single or Multi-Purpose Optimization over Networks

Authors:

Márta Esö
Laci Ladányi
Ted Ralphs
Les Trotter
Outline of Talk

• Introduction to Parallel Branch, Cut, and Price
• Description of SYMPHONY
• Exercises
**Generic Branch and Cut**

**Input:** \((E, F), c \in \mathbb{Z}^E, \alpha \in \mathbb{Z}\) and \(\bar{s} \in F\) such that 
\[c(\bar{s}) = \alpha.\]

**Output:** A least cost member \(s^*\) of \(F\).

1. Create an LP relaxation \(R^0\) consisting of inequalities valid for the polytope \(P = \text{conv}(F)\).

2. Set the candidate list \(C = \{R^0\}\).

3. **REPEAT UNTIL** \(C = \emptyset\)

   - Select a subproblem \(S^i\) defined by incidence vectors in \(F\) that are feasible solutions to the corresponding LP relaxation \(R^i\) from \(C\). Set \(C \rightarrow C \setminus R^i\).
   
   - Iteratively solve and augment \(R^i\) with additional violated inequalities valid for \(P\) until no more can be found.
   
   - If \(R^i\) becomes infeasible or its optimal value exceeds \(\alpha - 1\), then prune the subproblem.
   
   - Otherwise, if the optimal solution vector \(\hat{x}\) is integral check it for membership in \(F\). Update \(\alpha\) and \(\bar{s}\) if \(\hat{x} \in F\) and \(c(\hat{x}) < \alpha\).
   
   - If \(\hat{x}\) is infeasible, branch by partitioning \(\text{conv}(S^i)\) using 1 or more hyperplanes and add the new subproblems to \(C\).
SYMPHONY

SYMPHONY is a generic framework for implementing parallel branch, cut, and price.

• It was designed specifically to run in a parallel environment.
  - The same source code can be compiled to run in a serial, fully distributed (using PVM), or shared-memory (using threads) environment.
  - The user doesn’t need to have any knowledge of parallelism to implement.
  - It runs on multiple platforms and can even run slave processes simultaneously on different platforms.

• User supplies:
  - separation subroutines,
  - the initial LP relaxation,
  - feasibility checker, and
  - other optional subroutines.

• SYMPHONY takes care of all other aspects of algorithm execution, including communication.

• The source code and documentation are available free from www.BranchAndCut.org
Parallelizing Branch, Cut, and Price

There are several obvious ways to parallelize branch and cut:

- Process **multiple subproblems** in parallel.
  
  **Advantage:** Faster enumeration.
  
  **Disadvantage:** Can enlarge the search tree.

- Within a single subproblem, solve LP relaxations and generate cuts in parallel.
  
  **Advantage:** LP reoptimized sooner and more often.
  
  **Disadvantage:** Cut generation can “lag behind.”

- A further possibility is to process **multiple search trees** in parallel.
  
  **Advantage:** Trees share upper bounds, cuts, and can use different branching rules, etc.
  
  **Disadvantage:** Wasted computation.
Implementing Parallel Branch, Cut, and Price

In SYMPHONY, there are six module types that work together to perform the algorithm:

**Master** Maintains problem instance data, spawns other processes, performs I/O.

**Tree Manager** Controls overall execution by tracking growth of the tree and dispatching subproblems to the LP solvers.

**LP Solvers** Perform processing and branching operations on subproblems.

**Cut Generators** Take LP solutions and generate valid inequalities.

**Cut Pools** Act as auxiliary cut generators by maintaining a list of the “most effective” inequalities found so far.

**GUI** Allows graphical display of fractional and integer solutions as well as real-time addition of cuts by user.
The Processes of Parallel Branch and Cut

Figure 1: SYMPHONY modules
Current Ports

- **Platforms**
  - Pentium PC running Linux or Solaris
  - Sun Sparc running Solaris or SunOS
  - IBM RS6000 running AIX
  - DEC Alpha running OSF

- **Applications**
  - Vehicle Routing Problem
  - Traveling Salesman Problem
  - Airline Crew Scheduling Problem (Set Partitioning)

- **LP Solvers**
  - XMP
  - CPLEX

Process communication for distributed version is currently accomplished using the Parallel Virtual Machine message-passing protocol.
Measuring Parallel Speed-up

- The goal is to perform the same or less total work in parallel as in serial ⇒ **linear speed-up**.

- Speed-up is \( \frac{\text{parallel running time}}{\text{serial running time}} \).

- In our case, we would like the parallel running time to be less than \( \frac{\text{serial running time}}{\text{number of LP modules used}} \).

- Alternatively, we would like the size of the search tree to be constant.

- Factors affecting speed-up
  - Size of tree
  - Quality of initial upper bound
  - Algorithm for tree search

- To try to overcome the speed-up problem, there are two parameters available:
  - load_balance_level
  - load_balance_iter
Solving Large Scale Integer Programs

- Solving large scale IPs requires consideration of additional factors.

- Of primary concern is controlling the number of variables and constraints in each subproblem.

- This creates efficiencies in both memory use and solution speed.

- Simple pre-processing does not suffice because of the dynamic nature of the solution process.

- In addition, the cut pool and branch and cut tree must be stored as efficiently as possible without sacrificing solution speed.

- This creates tradeoffs that are not easy to make.
Handling of Cuts

- Currently, each LP has its own dedicated cut generator.

- Violated cuts are received and processed by the LP solver
  - Each LP solver maintains a small “local cut pool.”
  - A limited number of cuts are added to the LP in each iteration. This prevents “saturation.”
  - Cuts are added and/or removed from the LP dynamically based on their effectiveness.
  - Cuts are only sent to the global cut pool if they prove effective locally.

- One or more cut pools maintain a list of the most “effective” cuts found so far.
  - Each pool services a subtree – pools are dynamically allocated.
  - The use of multiple pools allows locally valid cuts to be generated if desired.
  - With multiple cut pools, pools are smaller and contain cuts that were generated “closer” in the tree ⇒ more likely to be violated.
  - The size of each pool is controlled through the purging of “ineffective” cuts.
Handling of Variables

• **Reduced cost and logical fixing** are used to remove variables (if allowed by user).

• **Column generation** is needed for very large problems.
  - The user supplies the base set of variables and a column generation subroutine.
  - Column generation can be done at various times.

• **A two-phase algorithm** is also available.
  - The algorithm is run to completion using the base set of variables before generating additional columns.
  - Using the upper bound and cuts from the first phase, all variables are priced out in the root node and are then propagated down into the leaves as required.
  - The tree is also trimmed by aggregating children back into their parent as appropriate.
  - Afterwards, each leaf is processed again.
Branching

- Can branch on cuts or variables.

- Multi-way branching is supported.
  - Any number of children is allowed.
  - Branch on several left hand values for a constraint.

- Strong branching is used by default.
  - Select several branching candidates (can be cuts, variables, or both)
  - “Presolve” each candidate.
  - Choose the “best” for branching.

- Fractional branching is also a built-in option.
Tree Manager

- **Data Storage**
  - Efficient data storage is essential.
  - The current state of the entire tree is stored, including the current basis.
  - The description of each node is stored explicitly or with respect to its parent, whichever is smaller.

- **Tree Management**
  - The search algorithm is a combination of depth-first and best-first search.
  - Depth-first avoids node set-up costs.
  - Best-first allows selection of “best” node.
Solving the Traveling Salesman Problem

The TSP is one of the most well-known combinatorial optimization problems.

Feasible solutions are those incidence vectors satisfying:

\[ \sum_{j=1}^{n} x_{ij} = 2 \quad \forall i \in V \]
\[ \sum_{i \in S \atop j \notin S} x_{ij} \geq 2 \quad \forall S \subset V, \, |S| > 1. \]

We have implemented a basic TSP solver using SYMPHONY with subroutines for separation from CONCORDE, a TSP solver developed by Applegate, Bixby, Chvatal, and Cook.

The following classes are separated:

- Subtour elimination constraints
- Blossom inequalities
- Comb inequalities
Exercises

- Everything needed is in

\donet\software\symphony\tools

  - cshrc.stub, lines which must be added to your .cshrc file in order to run the software.
  - testing, a shell script which runs the test set.
  - test.par, a template parameter file.
  - pickres, a shell script which parses the output file and writes a results summary in LaTex format.

- Groups
  - Parallel Speed-up
  - Strong Branching 1
  - Strong Branching 2
  - Search Algorithm
  - Cutting Planes
  - Constraint Management