Computational Integer Programming
Universidad de los Andes

Lecture 9

Dr. Ted Ralphs

Reading for This Lecture

- Branch-and-Price: Integer Programming with Column Generation. Savelsbergh (2001).
- Branch-and-Price: Column Generation for Huge Integer Programs. Barnhart, Johnson, Nemhauser, Savelsbergh, and Vance (1998).

Branch and Price

- Branch and cut is a method for solving integer programs whose LP relaxations have a *HUGE* number of potential constraints.
- *Branch and price*, on the other hand, is a method for dealing with problems that have a huge number of potential variables.
- The idea is to use *column generation* to solve the LP relaxations.
- We solve the LP relaxation to optimality with only a subset of the columns.
- We then ask whether any column that has been left out has negative reduced cost—if so, that column is added and we reoptimize.
- The problem of determining the column with most negative reduced cost is an optimization problem.
- Note that this can be seen as cut generation with respect to the dual of the LP relaxation.

Example: The Cutting Stock Problem

- We are selling rolls of paper in specified widths w_i, $i = 1, \ldots, m$.
- For each width i, we have a given demand d_i that must be satisfied.
- There are large rolls from which the smaller rolls are cut with width W.
- We want to minimize the total number of larger rolls we need to use.
- An IP formulation of this problem is

$$
\begin{aligned}
min \quad & \sum_{i=1}^{n} \lambda_i \\
s.t. \quad & \sum_{i=1}^{n} \lambda_i a^i \geq d \\
& \lambda_i \geq 0, i = 1, \ldots, n, \\
& \lambda_i \quad \text{integer}, i = 1, \ldots, n
\end{aligned}
$$

where the columns a^i represent the *feasible patterns*.

The Column Generation Subproblem

- The potential columns correspond to feasible *patterns*.
- A given column vector a corresponds to a feasible pattern if and only if

$$\sum_{i=1}^{m} a_i w_i \leq W$$

  and a contains only nonnegative integers.
- The objective function coefficient of every pattern (column) is 1.
- Finding the column with the smallest reduced cost is a knapsack problem:

$$\begin{aligned} \max & \sum_{i=1}^{m} p_i a_i \\ s.t. & \sum_{i=1}^{m} w_i a_i \leq W \\ & a_i \geq 0 \\ & a_i \quad \text{integer} \end{aligned}$$

Reformulating Using Dantzig-Wolfe

- Many (most?) problems that can be solved with column generation have formulations with a much smaller number of variables.
- The cutting stock problem, for example, could be formulated with a smaller number of variables, but typically is not.
- There are a number of reasons for using column generation in these cases
  - In principle, these methods are used primarily to strengthen the LP relaxation.
  - However, they may also be applied when the relaxation of certain linking constraints allows the problem to decompose into blocks.
  - This may then help to eliminate *symmetry* from the model.
  - A final possible reason is simply because solving the subproblem using a combinatorial algorithm can lead to a more efficient way of solving the LP relaxation.
- *Extended formulations* can be generated systematically using the method of Dantzig-Wolfe decomposition discussed earlier.

Example: The Generalized Assignment Problem

- The problem is to assign m tasks to n machines subject to capacity constraints.
- An IP formulation of this problem is

$$\max \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} z_{ij}$$

$$s.t. \qquad \sum_{j=1}^{n} z_{ij} = 1, \qquad i = 1, \ldots, m,$$

$$\sum_{i=1}^{m} w_{ij} z_{ij} \leq d_j, \qquad j = 1, \ldots, n,$$

$$z_{ij} \in \{0, 1\}, i = 1, \ldots, m, j = 1, \ldots, n,$$

Reformulating the Generalized Assignment Problem

- Let's rewrite the GAP using a Dantzig-Wolfe reformulation by relaxing the the first set of constraints.
- The subproblem then becomes a set of independent knapsack problems.

$$\max \sum_{j=1}^{n} \sum_{i=1}^{m} p_{ij} \left(\sum_{k=1}^{K_j} \lambda_k^j a_{ik}^j \right)$$

$$s.t. \quad \sum_{j=1}^{n} \sum_{k=1}^{K_j} \lambda_k^j a_{ik}^j = 1, \qquad i = 1, \ldots, m,$$

$$\sum_{k=1}^{K_j} \lambda_k^j = 1, \qquad j = 1, \ldots, n,$$

$$\lambda_k^j \in \{0, 1\}, j = 1, \ldots, n, k = 1, \ldots, K_j,$$

- Note that this a set partitioning problem.

Using the Dantzig-Wolfe Reformulation

- We can solve the LP relaxation of this reformulation using column generation to obtain a bound.
- The column generation subproblem is a set of independent knapsack problems (easily parallelized).
- Note that if the machines are identical, this collapses down to a single, much smaller subproblem.
- This corresponds to eliminating the symmetry that was present in the original problem.
- Embedding this bounding scheme into a branch and bound algorithm, we get branch and price.

Contrast with Branch and Cut

- Beware that the bound obtained by solving the LP relaxation is not a valid upper bound unless no columns can be generated!
- Note that the same phenomena can occur in branch and cut, but we are able to stop generating cuts anytime and still have a valid bound.
- It is possible to obtain a "true" upper bound, even when column generation has not been completed.
- However, this still requires exact solution of the subproblem.
- Generally speaking, most of the general framework of branch and cut can be transfered to branch and price.

Solving the Subproblem

- In practice, we do not need to solve the subproblem to optimality in every iteration.
- We only need to find *some* column with negative reduced cost.
- The only time we actually need to solve to optimality is to obtain a valid bound.
- It is also inefficient to generate just one single column in each iteration.
- As in branch and cut, we generally want to generate a number of columns that can be added at once.
- This may require adjustment of the solution algorithm.

Incomplete Methods

- Branch and price can be easily used as a heuristic method (and often is).
- If we use a heuristic method to generate columns, then the resulting solution solution will not necessarily be optimal.
- However, such a solution is often good enough.
- In many cases, an exact algorithm for the subproblem simply is not possible.
- Another heuristic version of the method is to generate columns only in the root node.
- We call this *price and branch*.

Branching with Dantzig-Wolfe Decomposition

- Unfortunately, branching on the variables of the reformulation doesn't work well.
- This is because it's generally difficult to keep a variable from being generated again after it's been fixed to zero.
- Branching must be done in a way that does not destroy the structure of the column generation subproblem.
- We can do this by branching on the *original* variables, i.e., before the reformulation.
- In a 0-1 problem, branching on the j^{th} original variable is equivalent to fixing the value of some element of the columns to be generated.
- This can usually be incorporated into the column generation subproblem.
- By limiting column generation in this way, we can implement a much wider array of branching rules.
- We may branch on disjunctions other than variables disjunctions.

Convergence and Tailing Off

- In practice, column generation methods are sometimes slow to converge.
- This can be due to instability of the dual solution, among other things.
- After a while, the bound may not change much after adding each new column.
- At this point, it may be better to branch than to continue to generate columns.
- Convergence can be accelerated using a number of different stablization techniques.

Branch and Cut and Price

- One of the limitations of branch and price is that the strength of the bound cannot really be improved arbitrarily as it can in branch and cut.
- In principle, however, it is possible to combine branch and cut with branch and price to obtain a powerful hybrid class of methods.
- Cut generation can be done in the space of the compact formulation, while bounding is done in the extended formulation space.
- This leads to a method that retains the advantages of both column and cut generation.

Conclusions

- In this course, we have seen an overview of the basic theory and practice of integer programming.
- What we have covered here only scratches the surface of what you must understand in order to implement a real solver.
- In practice, there are many, many more details involved.
- For example, we did not discuss
  - Numerical issues
  - Projection and lifting
  - Primal heuristics
- In practice, these issues are of crucial importance.
- With the knowledge of the tools described here, you should be able to understand how solvers work.
- In practice, it is often necessary to tweak the parameters of a solver in order to achieve desired performance for difficult classes of problems.
- This will be the focus of the exercises to come.
- Thanks for your attention!