Computational Integer Programming

Lecture 12: Branch and Cut

Dr. Ted Ralphs

Reading for This Lecture

- Wolsey Section 9.6
- Nemhauser and Wolsey Section II.6
- Martin "Computational Issues for Branch-and-Cut Algorithms" (2001)
- Linderoth and Ralphs "Noncommercial Software for Mixed-Integer Linear Programming"
- M.W.P. Savelsbergh "Preprocessing and Probing for Mixed Integer Programming Problems."
- T. Berthold "Primal Heuristics for Mixed Integer Programs."
- K. Wolter "Implementations of Cutting Plane separators for Mixed Integer Programs."
- A. Atamturk, G. Nemhauser, and M.W.P. Savelsburgh, "Conflict Graphs in Solving Integer Programming Problems."

Branch and Cut

- *Branch and cut* is an LP-based branch-and-bound scheme in which the linear programming relaxations are augmented by valid inequalities.
- The valid inequalities are generated dynamically using separation procedures.
- We iteratively try to improve the current bound by adding valid inequalities.
- In practice, branch and cut is the method typically used for solving difficult mixed-integer linear programs.
- It is a very complex amalgamation of techniques whose application must be balanced very carefully.

Computational Components of Branch and Cut

- Modular algorithmic components
  - Initial preprocessing and root node processing
  - Bounding
  - Cut generation
  - Primal heuristics
  - Node pre/post-processing (bound improvement, conflict analysis)
  - Node pre-bounding
- Overall algorithmic strategy
  - Search strategy
  - Bounding strategy
    - What cuts to generate and when
    - What primal heuristics to run and when
    - Management of the LP relaxation
  - Branching strategy
    - When to branch
    - How to branch (which disjunctions)
    - Relative amount of effort spent on choosing branch

Tradeoffs

- Control of branch and cut is about *tradeoffs*.
- We are combining many techniques and must adjust levels of effort of each to accomplish an end goal.
- Algorithmic control is an optimization problem in itself!
- Many algorithmic choices can be formally cast as optimization problems.
- What is the objective?
  - Time to optimality
  - Time to first "good" solution
  - Balance of both?

Preprocessing and Probing

- Often, it is possible to simplify a model using logical arguments.
- Most commercial IP solvers have a built-in preprocessor.
- Effective preprocessing can pay large dividends.
- Let the upper and lower bounds on x_j be u_j and l_j.
- The most basic type of preprocessing is calculating *implied bounds*.
- Let (π, π_0) be a valid inequality.
- If $\pi_1 > 0$, then

$$x_1 \leq (\pi_0 - \sum_{j:\pi_j>0} \pi_j l_j - \sum_{j:\pi_j<0} \pi_j u_j)/\pi_1$$

- If $\pi_1 < 0$, then

$$x_1 \geq (\pi_0 - \sum_{j:\pi_j>0} \pi_j l_j - \sum_{j:\pi_j<0} \pi_j u_j)/\pi_1$$

Basic Preprocessing

- Again, let (π, π_0) be any valid inequality for $\mathcal{S}$.
- The constraint $\pi x \leq \pi_0$ is redundant if

$$\sum_{j:\pi_j>0} \pi_j u_j + \sum_{j:\pi_j<0} \pi_j l_j \leq \pi_0.$$

- $\mathcal{S}$ is empty (IP is infeasible) if

$$\sum_{j:\pi_j>0} \pi_j l_j + \sum_{j:\pi_j<0} \pi_j u_j > \pi_0.$$

- For any IP of the form $\max\{cx | Ax \leq b, l \leq x \leq u\}, x \in \mathbf{Z}^n$,
  - If $a_{ij} \geq 0 \forall i \in [1..m]$ and $c_j < 0$, then $x_j = l_j$ in any optimal solution.
  - If $a_{ij} \leq 0 \forall i \in [1..m]$ and $c_j > 0$, then $x_j = u_j$ in any optimal solution.

Probing for Integer Programs

- It is also possible in many cases to fix variables or generate new valid inequalities based on logical implications.
- Consider (π, π_0), a valid inequality for 0-1 integer program.
- If $\pi_k > 0$ and $\pi_k + \sum_{j:\pi_j<0} \pi_j > \pi_0$, then we can fix x_k to zero.
- Similarly, if $\pi_k < 0$ and $\sum_{j:\pi_j<0, j\neq k} \pi_j > \pi_0$, then we can fix x_k to one.
- Example: Generating logical inequalities

Improving Coefficients

- Suppose again that (π, π_0) is a valid inequality for a 0-1 integer program.
- Suppose that $\pi_k > 0$ and $\sum_{j:\pi_j>0, j\neq k} \pi_j < \pi_0$.
- If $\pi_k > \pi_0 - \sum_{j:\pi_j>0, j\neq k} \pi_j$, then we can set
  - $\pi_k \leftarrow \pi_k - (\pi_0 - \sum_{j:\pi_j>0, j\neq k} \pi_j)$, and
  - $\pi_0 \leftarrow \sum_{j:\pi_j>0, j\neq k} \pi_j)$.
- Similarly, suppose that $\pi_k < 0$ and $\pi_k + \sum_{j:\pi_j>0, j\neq k} \pi_j < \pi_0$.
- Then we can again set $\pi_k \leftarrow \pi_k - (\pi_0 - \pi_j - \sum_{j:\pi_j>0, j\neq k} \pi_j)$

Preprocessing and Probing in Branch and Bound

- In practice, these rules are applied iteratively until none applies.
- Applying one of the rules may cause a new rule to apply.
- Bound improvement by reduced cost can be reapplied whenever a new bound is computed.
- Furthermore, all rules can be reapplied after branching.
- These techniques can make a very big difference.

Root Node Processing

- Typically, more effort is put into processing the root node than other nodes in the tree.
- Work done in the root node will impact the processing of every subsequent node.
- Dual bounding
  - Cut generation in the root node can be thought of as an additional pre-processing step ro the formulation before enumeration.
  - Cut generation in the root node can also be used to predict effectiveness of such techniques elsewhere in the tree.
- Primal bounding
  - Primal bounds found in the root node can have a big impact on the search.
  - They help to improvement variable bounds by reduced cost and can also lead to more effective/efficient search strategies.
  - As with cut generation, we use performance in the root node as an indicator of efficacy throughout the tree.

Node Pre/Post-Processing: Bound Improvement by Reduced Cost

- Consider an integer program $\max_{x \in \mathbb{Z}^n}\{cx \mid Ax \leq b, 0 \leq x \leq u\}$.
- Suppose the linear programming relaxation has been solved to optimality and row zero of the tableau looks like

$$z = \bar{a}_{00} + \sum_{j \in NB_1} \bar{a}_{0j} x_j + \sum_{j \in NB_2} \bar{a}_{0j}(x_j - u_j)$$

  where NB_1 are the nonbasic variables at 0 and NB_2 are the nonbasic variables at their upper bounds u_j.
- In addition, suppose that a lower bound $\underline{z}$ on the optimal solution value for IP is known.
- Then in any optimal solution

$$\begin{aligned} x_j &\leq \left\lfloor \frac{\bar{a}_{00} - \underline{z}}{-\bar{a}_{0j}} \right\rfloor \text{ for } j \in NB_1, \text{ and} \\ x_j &\geq u_j - \left\lceil \frac{\bar{a}_{00} - \underline{z}}{\bar{a}_{0j}} \right\rceil \text{ for } j \in NB_2. \end{aligned}$$

Node Pre/Post-Processing: Other Techniques

- Bound improvement in the root node
  - Whenever a new lower bound is found by a heuristic or otherwise, we can apply bound improvement in the root node.
  - To do so, we save the reduced costs of the variables in the root node.
  - We can do this for multiple bases obtained during the processing of the root node.
  - The bound improvements found in this way can be immediately applied to all candidate and active nodes.
- Techniques similar to those applied in the root node can also be applied during the processing of individual nodes.
- New implications may be available once branching constraints are applied.

Node Pre/Post-Processing: Conflict Analysis

- Whenever a node is found to be infeasible, we derive a *conflict*.
- The branching constraints imposed to arrive at that node cannot all be imposed simultaneously.
- These conflicts can be used to derive cuts and may also contribute to enhanement of the conflict graph.

Bounding

- For now, we focus on the use of cutting plane methods for bounding.
- We will discuss decomposition-based bounding in a later lecture.
- The bounding loop is essentially a cutting plen method for solving the subproblem, but with some kind of early termination criteria.
- After termination, branching is performed to continue the algorithm.
- The bounding loop consists of several steps applied iteratively (not necessarily in this order).
  - Solve the current LP relaxation.
  - Decide whether node can be fathomed (by infeasibility or bound).
  - Generate inequalities violated by the solution to the LP relaxation.
  - Perform primal heuristics.
  - Apply node pre/post-processing.
  - Manage/improve LP relaxation (add/remove cuts, change bounds)
  - Decide whether to branch

Solving the LP Relaxation

- The LP relaxation is typically solved using a simplex-based algorithm.
  - This yields the advantage of efficient warm-starting of the solution process.
  - Many standard cut generation techniques require a basic solution.
- Interior point methods may be useful in some cases where they are much more effective (set packing/partitioning is one case in which this is typical).
- It may also be fruitful in some cases to explore the use of alternatives, such as the Volume Algorith.

Cut Generation

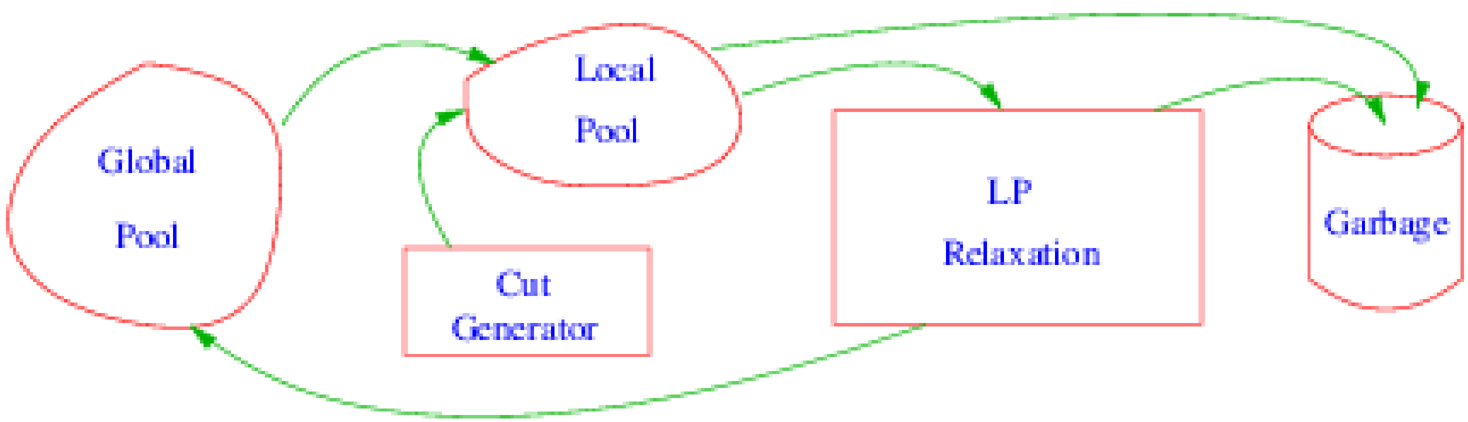
- Standard methods for generating cuts
  - Gomory, GMI, MIR, and other tableau-based disjunctive cuts.
  - Cuts from the node packing relaxation (clique, odd hole)
  - Knapsack cuts (cover cuts).
  - Single node flow cuts (flow cover).
  - Simple cuts from pre-processing (probing, etc).
- We must choose from among these various method which ones to apply in each node.
- We must in general decide on a general level of effort we want to put into cut generation.

Managing the LP Relaxations

- In practice, the number of inequalities generated can be HUGE.
- We must be careful to keep the size of the LP relaxations small or we will sacrifice efficiency.
- This is done in two ways:
  - Limiting the number of cuts that are added each iteration.
  - Systematically deleting cuts that have become *ineffective*.
- How do we decide which cuts to add?
- And what do we do with the rest?
- What is an ineffective cut?
  - One whose dual value is (near) zero.
  - One whose slack variable is basic.
  - One whose slack variable is positive.

Managing the LP Relaxations

- Below is a graphical representation of how the LP relaxation is managed in practice.
- Newly generated cuts enter a buffer (the *local cut pool*).
- Only a limited number of what are predicted to be the most effective cuts from the local are added in each iteration.
- Cuts that prove effective locally may eventually sent to a global pool for future use in processing other subproblems.



Cut Generation and Management

- A significant question in branch and cut is what classes of valid inequalities to generate and when?
- It is generally not a good idea to try all cut generation procedures on every fractional solution arising.
- For generic mixed-integer programs, cut generation is most important in the root node.
- Using cut generation only in the root node yields a procedure called *cut and branch*.
- Depending on the structure of the instance, different classes of valid inequalities may be effective.
- Sometimes, this can be predicted ahead of time (knapsack inequalities).
- In other cases, we have to use past history as a predictor of effectiveness.
- Generally, each procedure is only applied at a dynamically determined frequency.

Deciding Which Cuts to Add

- Predicting what cuts will be effective is difficult in general.
- Degree of violation is an easy-to-apply criteria, but may not be the most natural or intuitive measure.
- Other measures
  - Bound improvement (difficult to predict/calculate)
  - Euclidean distance from point to be cut off.
- It is possible to generate cuts using a different measure than that which is used to add them from the local pool.
- This might be done because generation by a criteria other than degree of violation is difficult.

Deciding When to Branch

- Because the cutting plane algorithm is a finite algorithm in itsef (at least in the pure integer case), there is no strict requirement to branch.
- The decision to branch is thus a practical matter.
- Typically, branching is undertaken when “tailing off” occurs, i.e., bound improvement slows.
- Detecting when this happens is not straightforward and there are many ways of doing it.
- Ultimately, branching and cutting (using the same disjunction) have the same impact on the bound.
- Tailing off may simply be a result of numerical issues
- We will consider the numerics of the solution process in a later lecture.

Balancing the Effort of Branching and Bounding

- To a large extent, the more effort one puts into branching, the smaller the search tree will be.
- Branching effort can be tuned by adjusting
  - How many branching candidates to consider.
  - How much efort should be put into estimating impact (pseduo-cost estimates versus strong branching, etc.).
  - The same can be said about efforts to improve both primal and dual bounds.
    * For dual bounds, we need to determine how much effort to spend generating various classes of inequalities.
    * For primal bounds, we need to determine how much effort to put into primal heuristics.
  - One of the keys to making the overall algorithm work well is to tune the amount of effort allocated to each of these techniques.
  - This is a very difficult thing to do and the proper balance is different for different classes of problems.

Primal Bounding Strategy

- The strategy space for primal heuristics is similar to that for cuts.
- We have a collection of different heuristics that can be applied.
- We need to determine which heuristics to apply and how often.
- Generally speaking, we do this dynamically based on the historical effectiveness of each method.

Computational Aspects of Search Strategy

- The search must find the proper balance between several factors.
  - Primal bound improvement versus dual bound improvement.
  - The savings accrued by diving versus the effectiveness of best first.
- When we are confident that the primal bound is near optimal, such as when the gap is small, a diving strategy is more appropriate.
- We can also adjust our strategy based on what the user's desire is.

Adjusting Strategy Based on User Desire

- In general, there is always a tradeoff between improvement of the dual and the primal bound.
- The user may have partcular desires about which of these is more important.
- Some solvers change their strategy according to the emphasis preferred by the user.
  - Proving optimality
  - Finding good solution quickly