The Branch-and-Cut Algorithm for Solving Mixed-Integer Optimization Problems

IMA New Directions Short Course on Mathematical Optimization

Jim Luedtke

Department of Industrial and Systems Engineering
University of Wisconsin-Madison

August 10, 2016

Branch-and-Bound

Basic idea behind most algorithms for solving integer programming problems

- Solve a *relaxation* of the problem
  - Some constraints are ignored or replaced with less stringent constraints
- Gives an upper bound on the true optimal value
- If the relaxation solution is feasible, it is optimal
- Otherwise, we divide the feasible region (branch) and repeat

Linear Programming Relaxation

Consider the integer program:

$$\text{MILP:} \quad z^* = \max\{c^\top x : x \in S\}$$

where

$$S := \{x \in \mathbb{R}^n_+ : Ax \leq b, x_j \in \mathbb{Z}, j \in J\}$$

Linear programming relaxation of MILP

$$\max\{c^\top x : x \in P'\}$$

where

$$P' := \{x \in \mathbb{R}^n_+ : A'x \leq b'\}$$

and $S \subseteq P'$

Linear Programming Relaxation

$$\text{MILP:} \quad z^* = \max\{c^\top x : x \in S\}$$

where

$$S := \{x \in \mathbb{R}^n_+ : Ax \leq b, x_j \in \mathbb{Z}, j \in J\}$$

Natural linear programming relaxation

$$z_0 = \max\{c^\top x : x \in P_0\}$$

where the P_0 is the natural linear relaxation

$$P_0 = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$$

How does z_0 compare to z^*?

What if the solution x^0 of the LP relaxation has $x^0_j \in \mathbb{Z}$ for all $j \in J$?

Branching: The "divide" in "Divide-and-conquer"

Generic optimization problem:

$$z^* = \max\{c^\top x : x \in S\}$$

Consider subsets $S_1, \ldots S_k$ of S which cover S: $S = \bigcup_i S_i$. Then

$$\max\{c^\top x : x \in S\} = \max_{1 \le i \le k} \Big\{ \max\{cx : x \in S_i\} \Big\}$$

In other words, we can optimize over each subset separately.

- Usually want S_i sets to be disjoint ($S_i \cap S_j = \emptyset$ for all $i \neq j$)

Dividing the original problem into subproblems is called branching

Bounding: The "Conquer" in "Divide-and-conquer"

Any feasible solution to the problem provides a lower bound L on the optimal solution value. $(\hat{x} \in S \Rightarrow z^* \geq c\hat{x})$.

- We can use heuristics to find a feasible solution $\hat{x}$

After branching, for each subproblem i we solve a *relaxation* yielding an upper bound $u(S_i)$ on the optimal solution value for the subproblem.

- Overall Bound: $U = \max_i u(S_i)$

Key: If $u(S_i) \leq L$, then we don't need to consider subproblem i.

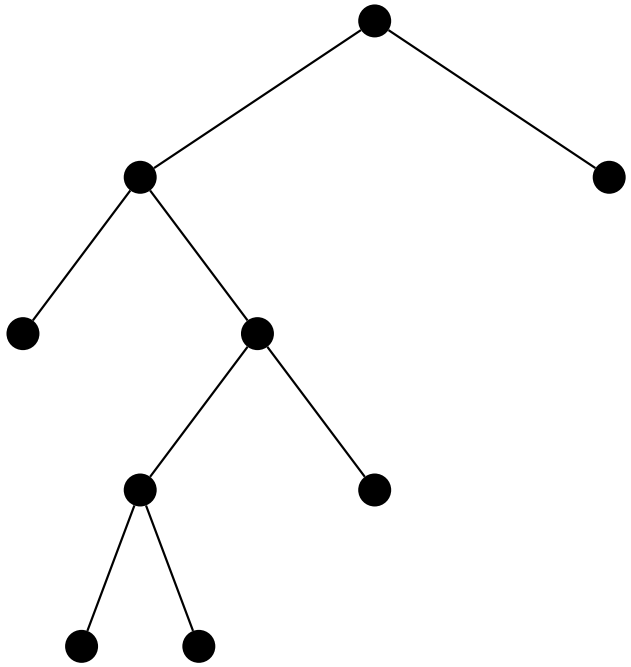
In MIP, we usually get the upper bound by solving the LP relaxation, but there are other ways too.

LP-based Branch-and-Bound for MIP

- Let z^* be the optimal value of the MIP
- In LP-based branch-and-bound, we first solve the LP relaxation of the original problem. The result is one of the following:
  1. The LP in unbounded $\Rightarrow$ the MIP is unbounded or infeasible.
  2. The LP is infeasible $\Rightarrow$ MIP is infeasible.
  3. We obtain a feasible solution for the MIP $\Rightarrow$ it is an optimal solution to MIP. $(L = z^* = U)$
  4. We obtain an optimal solution to the LP that is not feasible for the MIP $\Rightarrow$ Upper Bound. $(U = z_{LP})$.
- In the first three cases, we are finished.
- In the final case, we must **branch** and recursively solve the resulting subproblems.

Terminology

- If we picture the subproblems graphically, they form a **search tree**.
- Eliminating a problem from further consideration is called **pruning**.
- The act of bounding and then branching is called **processing**.
- A subproblem that has not yet been processed is called a **candidate**.
- The set of candidates is the **candidate list**.



LP-based Branch-and-Bound Algorithm

1. Derive a lower bound L using a heuristic method (if possible).
2. Put the original problem on the candidate list.
3. Select a problem S from the candidate list and solve the LP relaxation to obtain the bound $u(S)$
   - If the LP is infeasible $\Rightarrow$ node can be pruned.
   - If $u(S) > L$ and the solution is feasible for the MIP $\Rightarrow$ set $L \leftarrow u(S)$.
   - If $u(S) \leq L \Rightarrow$ node can be pruned.
   - Otherwise, branch. Add the new subproblems to the list.
4. If the candidate list in nonempty, go to Step 3. Otherwise, the algorithm is completed.

The "Global" upper bound

$U^t = \max\{u(\text{parent}(S)) : S \text{ in candidate list at step } t\}$

Let's Do An Example

maximize

$$5.5x_1 + 2.1x_2$$

subject to

$$\begin{aligned} -x_1 + x_2 &\leq 2 \\ 8x_1 + 2x_3 &\leq 17 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\text{ integer} \end{aligned}$$

Choices in Branch-and-Bound

Each of the steps in a branch-and-bound algorithm can be done in many different ways

- Heuristics to find feasible solutions – yields lower bounds
- Solving a relaxation – yields upper bounds
- Node selection – which subproblem to look at next
- Branching – dividing the feasible region

You can "help" an integer programming solver by telling it how it should do these steps

- You can even implement your own better way to do one or more of these steps
- You can do better because you know more about your problem

How Long Does Branch-and-Bound Take?

Simplistic approximation:

$$\text{Total time} = (\text{Time to process a node}) \times (\text{Number of nodes})$$

When making choices in branch-and-bound, think about effect on these separately

Question

Which of these is likely to be *most important* for hard instances?

Choices in Branch-and-Bound: **Heuristics**

Practical perspective: finding good feasible solutions is most important

- Manager won't be happy if you tell her you have no solution, but you know the optimal solution is at most U

A heuristic is an algorithm that *tries* to find a good fesible solution

- No guarantees – maybe fails to find a solution, maybe finds a poor one
- But, typically runs fast
- Sometimes called "primal heuristics"

Good heuristics help find an *optimal solution* in branch-and-bound

- Key to success: Prune early and often
- We prune when $u(S_i) \leq L$, where L is the best lower bound
- Good heuristics $\Rightarrow$ larger $L \Rightarrow$ prune more

Heuristics – Examples

- Solving the LP relaxation can be interpreted as a heuristic
  - Often (usually) fails to give a feasible solution
- Rounding/Diving
  - Round the fractional integer variables
  - With those fixed, solve LP to find continuous variable values
  - Diving: fix *one* fractional integer variable, solve LP, continue
  - Many more clever possibilities
- *Metaheuristics*—Simulated Annealing, Tabu Search, Genetic Algorithms, etc...
- Optimization-based heuristics
  - Solve a heavily restricted version of the problem *optimally*
  - Relaxation-induced neighborhood search (RINS), local branching
- Problem specific heuristics
  - This is often a *very good* way to help an IP solver

Choices in Branch-and-Bound: **Choosing/solving the Relaxation**

- The relaxation is the most important factor for proving a solution is optimal
- Optimal value of the relaxation yields the upper bound
  - Recall: we prune when $u(S_i) \leq L$
  - Smaller (tighter) upper bounds $\Rightarrow$ prune more
  - So the formulation is very important
- Time spent solving the relaxation at each node usually dominates the total solution time
  - Want to solve it fast, but also want to solve fewer
  - Potential trade-off: a formulation that yields a better upper bound may be larger (more time to solve relaxation)
    - Usually, the formulation with a better upper bound wins

Solving the LP relaxation Efficiently

- Branching is usually done by changing bounds on a variable which is fractional in the current solution ($x_j \leq 0$ or $x_j \geq 1$)
- Only difference in the LP relaxation in the new subproblem is this bound change
  - LP dual solution remains feasible
  - Reoptimize with dual simplex
  - If choose to process the new subproblem next, can even avoid refactoring the basis

Choices in Branch-and-Bound: **Node selection**

Node selection: Strategy for selecting the next subproblem (node) to be processed.

- Important, but not as important as heuristics, relaxations, or branching (to be discussed next)
- Often called search strategy

Two different goals:

- Minimize overall solution time.
- Find a good feasible solution quickly.

The Best First Approach

- One way to minimize overall solution time is to try to minimize the size of the search tree.
- We can achieve this if we choose the subproblem with the best bound (highest upper bound if we are maximizing).
- Let's prove this
  - A candidate node is said to be *critical* if its bound exceeds the value of an optimal solution solution to the IP.
  - Every critical node will be processed no matter what the search order
  - Best first is guaranteed to examine only critical nodes, thereby minimizing the size of the search tree (for a given fixed choice of branching decisions).

Drawbacks of Best First

1. Doesn't necessarily find feasible solutions quickly
   - Feasible solutions are "more likely" to be found deep in the tree
2. Node setup costs are high
   - The linear program being solved may change quite a bit from one node LP solve to the next
3. Memory usage is high
   - It can require a lot of memory to store the candidate list, since the tree can grow "broad"

The Depth First Approach

Depth first: always choose the deepest node to process next

- Dive until you prune, then back up and go the other way

Avoids most of the problems with best first:

- Number of candidate nodes is minimized (saving memory)
- Node set-up costs are minimized since LPs change very little from one iteration to the next
- Feasible solutions are usually found quickly

Unfortunately, if the initial lower bound is not very good, then we may end up processing many non-critical nodes.

- We want to avoid this extra expense if possible.

Hybrid Strategies

A Key Insight

If you *knew* the optimal solution value, the best thing to do would be to go depth first

- Idea: Go depth-first until z_{LP} goes below optimal value z^*, then make a best-first move.
- But we don't know the optimal value!
  - Make an estimate z_E of the optimal solution value
  - Go depth-first until $z_{LP} \leq z_E$
  - Then jump to a better node

Choices in branch-and-bound: **Branching**

- If our "relaxed" solution $\hat{x}$ is not integer feasible, we must decide how to partition the search space into smaller subproblems
- The strategy for doing this is called a **Branching Rule**
- Branching wisely is *very* important
  - Significantly impacts bounds in subproblems
  - It is most important at the top of the branch-and-bound tree

Branching in Integer Programming

Most common approach: Changing variable bounds

- If $\hat{x}$ is not integer feasible, choose $j \in N$ such that $f_j := \hat{x}_j - \lfloor \hat{x}_j \rfloor > 0$
- Create two problems with additional constraints
  1. $x_j \leq \lfloor \hat{x}_j \rfloor$ on one branch
  2. $x_j \geq \lceil \hat{x}_j \rceil$ on other branch
- In the case of 0-1 IP, this dichotomy reduces to
  1. $x_j = 0$ on one branch
  2. $x_j = 1$ on other branch

Key question

Which variable to branch on?

The Goal of Branching

Branching divides one problem into two or more subproblems

- We would like to choose the branching that minimizes the sum of the solution times of all the created subproblems.
- This is the solution of the *entire subtree* rooted at the node.

How do we know how long it will take to solve each subproblem?

- **Answer**: We don't.
- **Idea**: Try to branch on variables that will cause the upper bounds to decrease the most
- This will lead to more pruning, and smaller subtrees

Predicting the Bound Change in a Subproblem

How can I (quickly?) estimate the upper bounds that would result from branching on a variable?

- **Strong branching**
  - Actually solve the LP relaxation of each subproblem for each potential branching variable
- **Pseudo-costs**
  - Approximate the bound change based on previous information collected in the branch-and-bound tree
- Hybrid: "Reliability branching"
- **Tentative branching**
  - Like strong branching, but also add valid inequalities to the subproblems, and possibly branch a few times

Strong Branching: Practicalities

Don't fully solve the subproblem LPs – just do a few dual simplex pivots

- This gives an upper bound on the subproblem bound
- How many is "a few"? — empirical study suggests 25 or so

Don't check subproblem for *every* candidate branching variable

- Which to evaluate?
- Look at an estimate of their effectiveness that is very cheap to evaluate
  - E.g., "most fractional" variables, or pseudocost (next slide)
- Perhaps evaluate more candidates near the top of the tree

Fully solving the LPs or evaluating more candidates will probably reduce search tree size, but likely increases total time

Using Pseudo-costs

- The **pseudo-cost** of a variable is an estimate of the per-unit change in the objective function from forcing the value of the variable to be rounded up or down. Like a gradient!
- For each variable x_j, we maintain an up and a down pseudo-cost, denoted P_j^+ and P_j^-.
- Let f_j be the current (fractional) value of variable x_j.
- An estimate of the change in objective function in each of the subproblems resulting from branching on x_j is given by

$$D_j^+ = P_j^+(1 - f_j),$$
$$D_j^- = P_j^- f_j.$$

- How to get the pseudo-costs?

Obtaining and Updating Pseudo-costs

- Empirical data
  - Observe the actual change that occurs after branching on each one of the variables and use that as the pseudo-cost
- We can either choose to update the pseudo-cost as the calculation progresses or just use the first pseudo-cost found
  - Pseudo-costs tend to remain fairly constant
- How to initialize? Possibilities:
  - Use the objective function coefficient
  - Use the average of all known pseudo-costs
  - Explicity initialize the pseudocosts using strong branching – this is the hybrid "reliability branching" approach

Combining Multiple Subproblem Bounds

- For each candidate branching variable, we calculate an estimate of the upper bound change for each subproblem
  - Either via strong branching or pseudo-costs
- How do we combine the two numbers together to form one measure of goodness for choosing a branching variable?
- Idea: branch on variable x_{j^*} with:

$$j^* = \arg\max\{D_j^+ D_j^-\}.$$

- Other alternative: A weighted sum of (min/max)...

Formalizing MIP formulations

Definition: Formulation

Let $X \subseteq \{x \in \mathbb{R}^n : x_j \in \mathbb{Z}, j \in J\}$. A polyhedron $P \subseteq \mathbb{R}^n$ is a formulation for X if

$$X = \{x \in P : x_j \in \mathbb{Z}, j \in J\}$$

There may be multiple valid formulation for a given set X

- E.g., two polyhedron P_1 and P_2, $P_1 \neq P_2$, with

$$X = \{x \in P_1 : x_j \in \mathbb{Z}, j \in J\} = \{x \in P_2 : x_j \in \mathbb{Z}, j \in J\}$$

- When using branch-and-bound, what makes a formulation "good"?

Extended Formulations

The polyhedron P (used to define a MIP formulation) might be represented using extra variables:

$$P = \{x \in \mathbb{R}^n : \exists y \in \mathbb{R}^p \text{ s.t. } Ax + By \leq b\}$$

- If we let
$$Q = \{(x, y) \in \mathbb{R}^{n+p} : Ax + By \leq b\}$$
then we say P is a projection of Q, and write $P = \text{proj}_x(Q)$
- We say Q is an extension of P, and the representation $(Ax + By \leq b)$ is an extended formulation of P

A useful result about projections:

Theorem

A projection of a polyhedron is a polyhedron.

All Formulations are NOT Created Equal

Definition: Better formulation

Given two formulations for X, formulation P_1 is better than P_2 if $P_1 \subsetneq P_2$.

Why this definition?

Best Possible Formulation

Definition: Convex hull

Let $X \subseteq \mathbb{R}^n$. The convex hull of X, denoted $\text{conv}(X)$, is the set of all points that can be written as a convex combination of points in X:

$$\text{conv}(X) = \left\{ \begin{array}{ll} x \in \mathbb{R}^n : & \exists\, x^1, \ldots, x^t \in X \text{ and } \lambda \in \mathbb{R}^t_+ \text{ with} \\ & x = \sum_{i=1}^t \lambda_i x^i, \quad \sum_{i=1}^t \lambda_i = 1 \end{array} \right\}.$$

Theorem

If P is *any* formulation of X, then $\text{conv}(X) \subseteq P$.

Is $\text{conv}(X)$ a Formulation?

To be a formulation, $\text{conv}(X)$ must be a polyhedron.

Theorem

If X is the feasible region of a mixed-integer program defined by rational data, then $\text{conv}(X)$ is a polyhedron.

- Easy to prove for bounded pure integer programs.

IP $\equiv$ LP!

Theorem

If x is an extreme point of $\text{conv}(X)$, then $x \in X$.

- We'll prove it...

Conclusion

$$\max\{c^\top x : x \in X\} = \max\{c^\top x : x \in \text{conv}(X)\}$$

So, if X is the feasible region of our MIP, and we know the convex hull of X, we could solve the MIP as a linear program!

$\Rightarrow$ We'd *really really really* like to know the convex hull of X.

Which Formulation is Better?

To prove P_1 is a better formulation than P_2

1. Prove $P_1 \subseteq P_2$.
   - Formally: let $x \in P_1$, then show $x \in P_2$.
2. Give an example that shows $P_1 \neq P_2$.
   - I.e., for a specific instance of your problem, find a point $\hat{x} \in P_2$ with $\hat{x} \notin P_1$.
   - Or, simply solve LP relaxations: $z_1 = \max_{x \in P_1} c^\top x$ and $z_2 = \max_{x \in P_2} c^\top x$: If $z_1 < z_2$, this shows $P_1 \neq P_2$

Given formulations P_1 and P_2, it is possible that neither is better

- But, then there's an obvious formulation better than them both!

Example: "big-M" constraints

Suppose for a binary variable y, you want to model the condition:

$$y = 1 \quad \Rightarrow \quad a^\top x \leq b$$

where $a \in \mathbb{R}^n$, and $x \in Q$, for some bounded set Q

- Model with:

$$a^\top x - b \leq M(1 - y)$$

  where M is a "big enough" number
- How big is "big enough"? $M \geq M^*$, where

$$M^* = \max\{a^\top x - b : x \in Q\}$$

- Claim: Using M^* instead of any $M > M^*$ is better!

Example: Lot Sizing Problem

Recall the single-item uncapacitated lot sizing problem

$$\begin{aligned}
\min \quad & \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t \\
\text{s.t.} \quad & s_{t-1} + x_t - s_t = d_t, \quad t = 1, \ldots, n \\
& x_t \leq D_{tn} y_t, \qquad t = 1, \ldots, n \\
& s_0 = s_n = 0, s_t \geq 0, x_t \geq 0, y_t \in \{0, 1\}, \ t = 1, \ldots, n
\end{aligned}$$

where $D_{j\ell} = \sum_{t=j}^{\ell} d_t$ for $j \leq \ell$.

- This problem can be solved in polynomial time by dynamic programming
- Question: Why bother trying to find a good MIP formulation for this problem?
- Answer: The same structure may appear in more general production planning problems, e.g., having multiple products

Lot Sizing: Extended Formulation

Let w_{kt} = amount of product made in period k and used to meet demand in period t $(k \leq t)$

- We relate these variables to the x_t variables:

$$x_t = \sum_{l=t}^{n} w_{tl}, \quad t = 1, \ldots, n$$

- Also the inventory variables can be expressed with these new variables:

$$s_t = \sum_{k=1}^{t} \sum_{l=t+1}^{n} w_{kl}, \quad t = 1, \ldots, n$$

Lot Sizing: Extended Formulation

Let w_{kt} = amount of product made in period k and used to meet demand in period t $(k \leq t)$

- So far we've done nothing! What more?

New constraints on new variables:

$$w_{tl} \leq d_l y_t, \quad 1 \leq t \leq l \leq n$$

Compare these to bounds on x (recall $x_t = \sum_{l \geq t} w_{tl}$):

$$x_t \leq \left(\sum_{l=t}^{n} d_l \right) y_t, \quad 1 \leq t \leq n$$

- We keep all other constraints $\Rightarrow$ This formulation can only be better

Lot Sizing: Extended formulation

We can also eliminate the x and s variables

- Constraint that must meet demands in each time period:

$$\sum_{k=1}^{t} w_{kt} = d_t, \quad t = 1, \ldots, n$$

- Expression for holding cost in objective:

$$\sum_{l=1}^{n} \sum_{k=1}^{l} \left(\sum_{t=k}^{l} h_t \right) w_{kl}$$

- Expression for production cost in objective:

$$\sum_{t=1}^{n} \sum_{k=1}^{t} p_k w_{kt}$$

Review: Branch-and-Bound

Branch-and-bound works (well) only if obtain good upper bounds from relaxation

- This motivated search for better formulations

Valid inequalities/cuts:

- Adaptively find a better formulation

Valid Inequalities

Let $X = \{x \in \mathbb{R}^n_+ : Ax \leq b, x_j \in \mathbb{Z}, j \in J\}$

Definition

An inequality $\pi x \leq \pi_0$ is a valid inequality for X if $\pi x \leq \pi_0$ for all $x \in X$. $(\pi \in \mathbb{R}^n, \pi_0 \in \mathbb{R})$

- Valid inequalities are also called "cutting planes" or "cuts"
- Goal of adding valid inequalities to a formulation: improve relaxation bound $\Rightarrow$ explore fewer branch-and-bound nodes

Key questions

- How to find valid inequalities?
- How to use them in a branch-and-bound algorithm?

Using Valid Inequalities

1. Add them to the initial formulation
   - Creates a formulation with better LP relaxation
   - Feasible only when you have a "small" set of valid inequalities
   - Easy to implement

---

2. Add them only as needed to cut off fractional solutions
   - Solve LP relaxation, cut off solution with valid inequalities, repeat
   - Cut-and-branch: Do this *only* with the initial LP relaxation (root node)
   - Branch-and-cut: Do this at all nodes in the branch-and-bound tree

Trade-off with increasing effort generating cuts

\+ Fewer nodes from better bounds

\– More time finding cuts and solving LP

Branch-and-Cut

At each node in branch-and-bound tree

1. Solve current LP relaxation $\Rightarrow \hat{x}$
2. Attempt to generate valid inequalities that cut off $\hat{x}$
3. If cuts found, add to LP relaxation and go to step 1

Why branch-and-cut?

- Reduce number of nodes to explore with improved relaxation bounds
- Add inequalities required to define feasible region

This approach is the heart of all modern MIP solvers

Deriving Valid Inequalities

General purpose valid inequalities

- Assume only that you have a (mixed or pure) integer set described with inequalities

$$X = \{x \in \mathbb{R}^n_+ : Ax \leq b, x_j \in \mathbb{Z}, j \in J\}$$

- Critical to success of deterministic MIP solvers
- Focus of two lectures tomorrow

Structure-specific valid inequalities

- Rely on particular structure that appears in a problem
- Combinatorial optimization problems: matching, traveling salesman problem, set packing, ...
- Knapsack constraints: $\{x \in \mathbb{Z}^n_+ : ax \leq b\}$
- Flow balance with variable upper bounds, etc.

Valid Inequalities: Lot Sizing Problem

Recall again the single-item uncapacitated lot sizing problem

$$\begin{aligned}
\min \ & \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t \\
\text{s.t.} \ & s_{t-1} + x_t - s_t = d_t, \quad t = 1, \ldots, n \\
& x_t \le D_{tn} y_t, \qquad t = 1, \ldots, n \\
& s_0 = s_n = 0, s_t \ge 0, x_t \ge 0, y_t \in \{0, 1\}, \ t = 1, \ldots, n
\end{aligned}$$

where $D_{j\ell} = \sum_{t=j}^{\ell} d_t$ for $j \le \ell$.

- Valid inequalities for this problem useful for any problem containing this structure in it, e.g., production planning problems with multiple products

Lot Sizing Valid Inequalities

Let X be the feasible region of the lot sizing problem.

Theorem

Let $\ell \in \{1, \ldots, n\}$ and $S \subseteq \{1, \ldots, \ell\}$. The inequality

$$\sum_{j \in S} x_j \leq \sum_{j \in S} D_{j\ell} y_j + s_\ell$$

is valid for X.

Strength of the Lot Sizing Valid Inequalities

Let X be the feasible region of the lot sizing problem.

Theorem

$\text{conv}(X)$ is described by the inequalities defining X and the set of all ℓ, S inequalities.

Separation of ℓ, S Inequalities

- For any $\ell \in \{1, \ldots, n\}$ and $S \subseteq \{1, \ldots, \ell\}$: the ℓ, S inequality:

$$\sum_{j \in S} x_j \leq \sum_{j \in S} D_{j\ell} y_j + s_\ell$$

Given an LP relaxation solution $(\hat{x}, \hat{s}, \hat{y})$ how to efficiently find a violated one if one exists?

- Try all $\ell = 1, \ldots, n$. (Not too many of those!)
- For fixed ℓ, what set S defines most violated inequality?

$$\max_{S \subseteq \{1,\ldots,\ell\}} \sum_{j \in S} (\hat{x}_j - D_{j\ell} \hat{y}_j)$$

- $S^*_\ell = \{j \in \{1, \ldots, \ell\} : \hat{x}_j - D_{j\ell} \hat{y}_j > 0\}$
- A violated cut exists (defined by ℓ, S^*_ℓ) if and only if

$$\sum_{j \in S^*_\ell} (\hat{x}_j - D_{j\ell} \hat{y}_j) > \hat{s}_\ell$$