Lecture 14 Branch and Bound

Can Li

ChE 597: Computational Optimization
Purdue University

Intuition: Divide and Conquer

$$Z = \min\{cx : x \in S\}$$

where S is MILP-representable.

- How can we break the problem into a series of smaller problems that are easier, solve the smaller problems and then put the information together again to solve the original problem?
- **Theorem** Let $S = S_1 \cup \ldots \cup S_K$ be a decomposition of S into smaller sets, and let $Z^k = \min\{cx : x \in S_k\}$ for $k = 1, \ldots, K$. Then $Z = \min_k Z^k$.

Motivating example: binary enumeration

- $S \subseteq \{0,1\}^3$. We first divide S into $S_0 = \{x \in S : x_1 = 0\}$ and $S_1 = \{x \in S : x_1 = 1\}$, then $S_{00} = \{x \in S_0 : x_2 = 0\} = \{x \in S : x_1 = x_2 = 0\}$, $S_{01} = \{x \in S_0 : x_2 = 1\}$, and so on.
- A leaf node $S_{i_1 i_2 i_3}$ is nonempty if and only if $x = (i_1, i_2, i_3)$ is in S.
- Explicit enumeration needs 2^n leaf nodes.

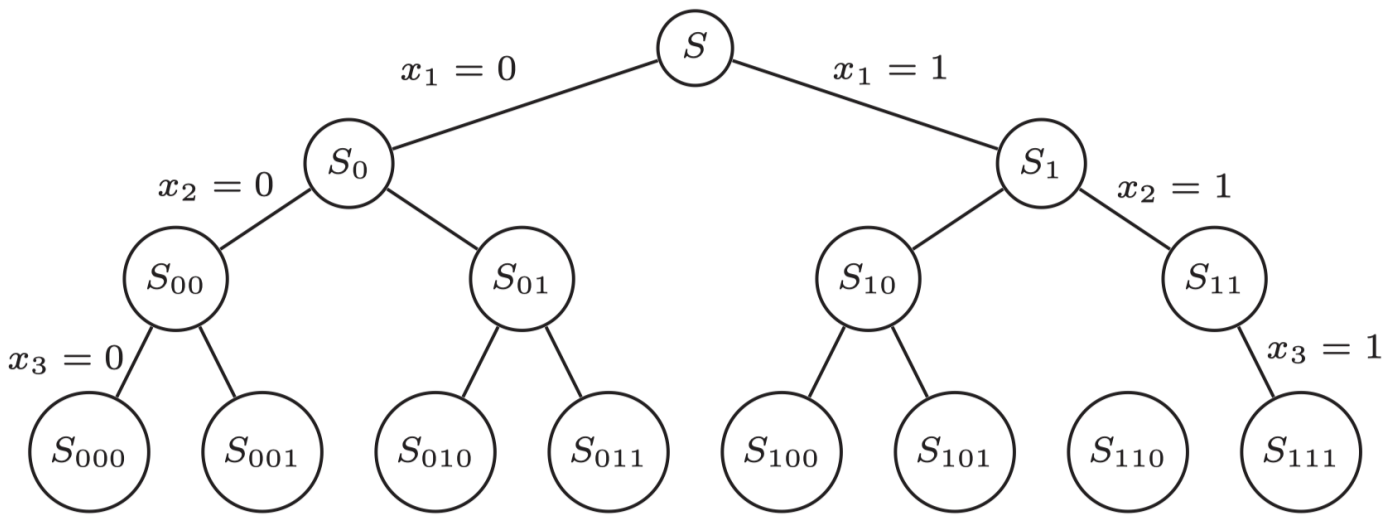
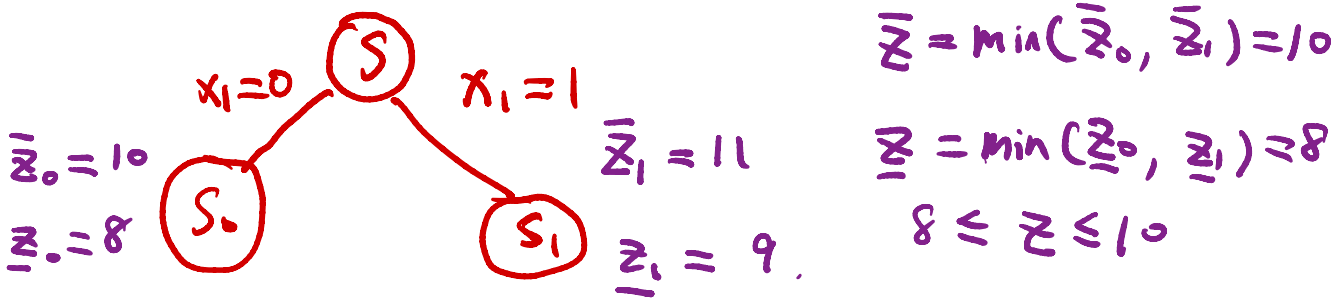


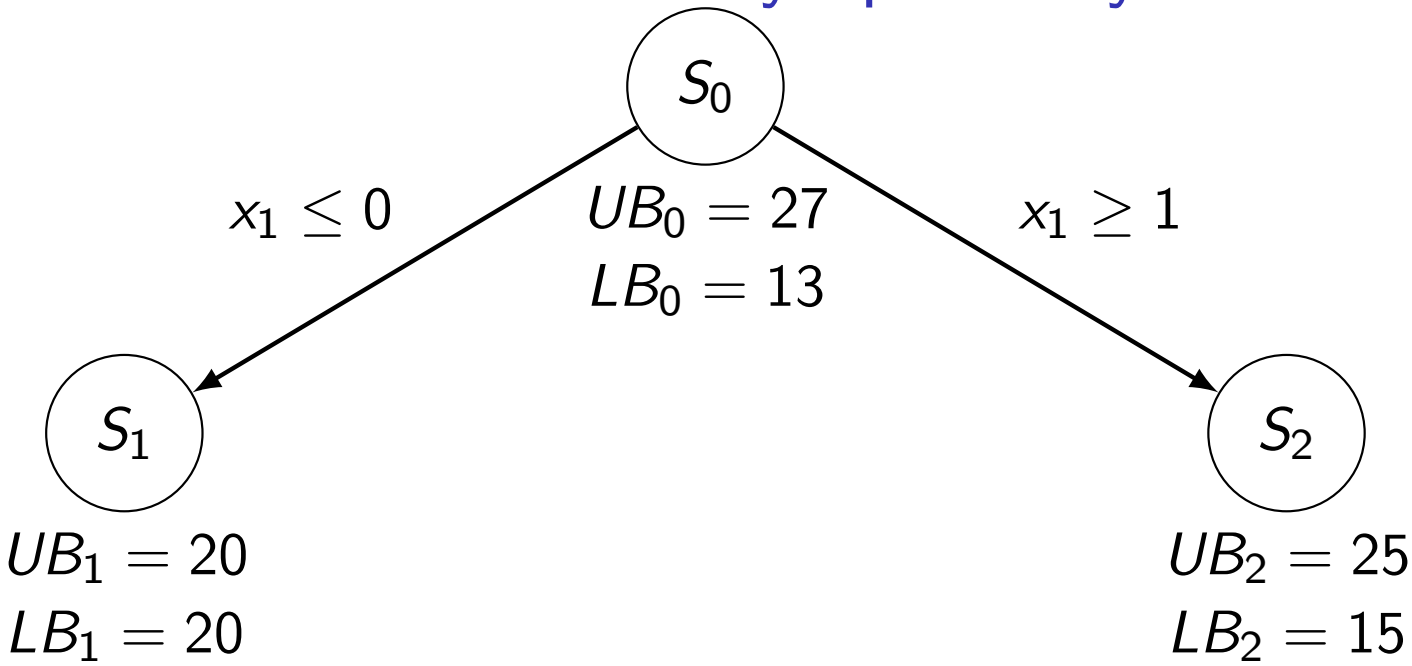
Figure: Binary enumeration tree (Wolsey, 2020)

Implicit enumeration

- Branch and bound is an implicit enumeration scheme that intelligently use bound information.
- For minimization problem, the upper bound is provided by a feasible solution, the lower bound is provided by a relaxation, e.g., the LP relaxation at the rootnode.
- **Theorem** Let $S = S_1 \cup \cdots \cup S_K$ be a decomposition of S into smaller sets, and let $Z^k = \min\{c^T x : x \in S_k\}$ for $k = 1, \ldots, K$, $\bar{Z}^k$ be an upper bound on Z^k and $\underline{Z}^k$ be a lower bound on Z^k. Then $\bar{Z} = \min_k \bar{Z}^k$ is an upper bound on Z and $\underline{Z} = \min_k \underline{Z}^k$ is a lower bound on Z.

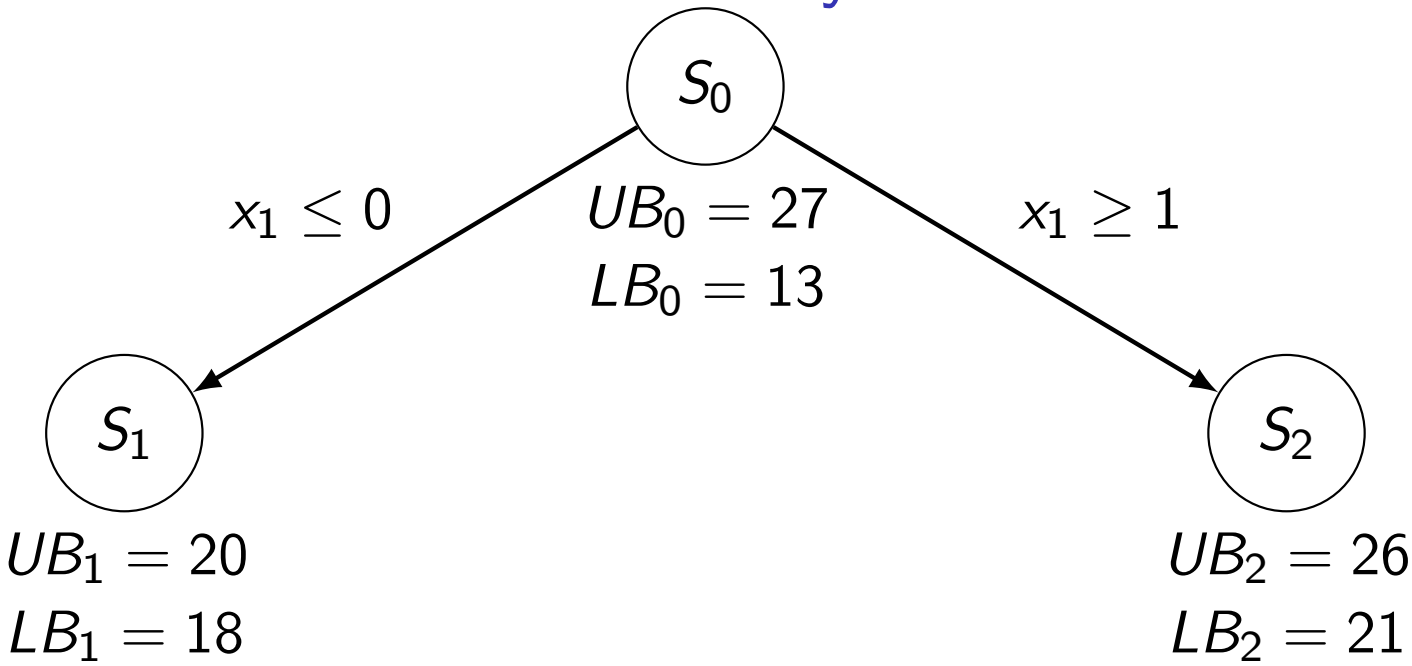


Prune by optimality



- Think about LB_1 being adding $x_1 \leq 0$ to the LP relaxation. The UBs come from some heuristics, e.g., local search, rounding.
- Node S_1 can be pruned by optimality, i.e., $UB_1 = LB_1$.
- Pruning S_1 avoid further branching on S_1, potentially saving the time to solve 2^{n-1} LPs compared with explicit enumeration.
- A special case is when the solution to the relaxation at a given node is integral. "prune by integrality".

Prune by bound



- Node S_2 can be pruned by bound. 于 $LB_2 = 20$
- $LB_2 = 21$, meaning the best solution that can be found on S_2 always be ≥ 21. We already find a feasible solution $UB_1 = 20$ on node S_1. Therefore, it makes no sense to continue exploring node S_2
- A special case of "prune by bound" is when the relaxation at a given node is infeasible. This case is also called "prune by infeasibility".

Branch and Bound Method for MILP

$$(P) \quad Z = \min\{c^T x : Ax \leq b, x \geq 0, x_j \text{ integer}, j \in \mathcal{N}_1 \subseteq \mathcal{N}\}$$

1. **Initialization.** Put S_0 (root node relaxation) on the list $\mathcal{A}$ of active subproblems. Set the upper bound $\overline{Z} := +\infty$ ($\overline{Z}$ is the value of the best integer solution found so far).
2. **Node Selection.** If $\mathcal{A} = \emptyset$, stop: solution associated with $\overline{Z} < +\infty$ is optimal. Or, if $\overline{Z} = +\infty$, P is infeasible.
   Else choose node $S_i \in \mathcal{A}$ according to a *node selection rule*, set $\mathcal{A} := \mathcal{A} \setminus S_i$.

Branch and Bound Method for MILP

3. **Lower Bounding (and Pruning).** Solve the relaxation corresponding to S_i. Let x' be the solution, the lower bound at node S_i is $\underline{Z}_i = c^T x'$.
   If $\underline{Z}_i \geq \overline{Z}$ (including $\underline{Z}_i = +\infty$, i.e. S_i infeasible) discard S_i and go to 2. (Pruning by bound or by infeasibility.)
   If $\underline{Z}_i < \overline{Z}$ and x' is integer,
   - set $\overline{Z} := \underline{Z}_i$, store x'
   - discard S_i and go to 5. (Pruning by optimality.)

   If $\underline{Z}_i < \overline{Z}$ and x' is not integer, continue to step 4.

Branch and Bound Method for MILP

4. **Upper Bounding (optional).** Perform heuristics, e.g., local search, to find integer solution. If a better solution is found, update $\overline{Z}$.
5. **Pruning by bound.** Discard all $S_j \in \mathcal{A}$ such that $\underline{Z}_j \geq \overline{Z}$.
6. **Branching.** If the selected node S_i has not been pruned, apply a *branching rule* to S_i, i.e., generate new subproblems $S_{i1}, \ldots, S_{iq}$, put them into $\mathcal{A}$ and go to 2.

Node selection rules

- **Depth-First Search** moving as far as possible down a branch before backtracking. Rationale: we should descend as quickly as possible in the enumeration tree to find a first feasible solution. This helps pruning by bound/optimality.
- **Best Node First** choose the active node with the best (smallest lower) bound (i.e. choose node s, where $\underline{Z}_s = \min_{i \in \mathcal{A}} \underline{Z}_i$). With this rule, the global lower bound will always be improved after branching. This is the most commonly used branching rule.

Branching rules

After the node is selected, which variable should we branch on?

- **Most fractional variable** For an integer variable x_j taking a fractional value x_j^*, the down-fractionality is $f_j = x_j^* - \lfloor x_j^* \rfloor$ and the up-fractionality is $1 - f_j$. Branch on the variable j whose fractionality is closest to 0.5.
- Rationale: make the variable less fractional.
- The issue with this branching rule is that it does not explicitly account for the improvement in bounds. The bound may be more sensitive to some variables than others.

Strong branching

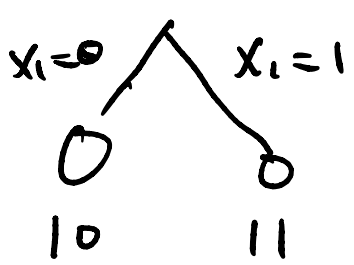
- Idea: try branching on all the fractional variables and pick the one with the best bound improvement.
- Let C be the set of fractional integer variables at a given node. Branches up and down on each of them in turn, and reoptimizes on each branch either to optimality, or for a specified number of dual simplex pivots. Now, for each variable $j \in C$, it has lower bounds $\underline{Z}_j^D$ for the down-branch and $\underline{Z}_j^U$ for the up-branch. The variable having the largest effect (largest dual bound) is selected for branching.

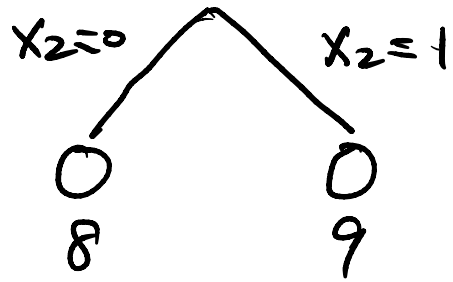
$$j^* = \arg\max_{j \in C} \min \left[\underline{Z}_j^D, \underline{Z}_j^U \right]$$

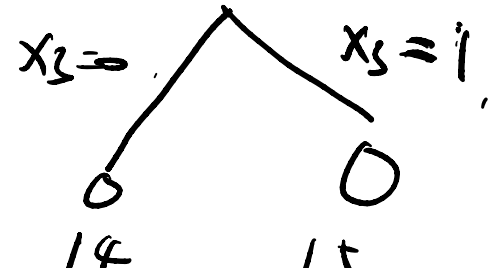
- Pro: Strong branching usually yields small tree size.
- Con: Each step of strong branching is expensive. We need to solve $2|C|$ LPs to decide which variable to branch on.

root: $x_1 = 0.3$, $x_2 = 0.5$, $x_3 = 0.6$.

$LB = 6$

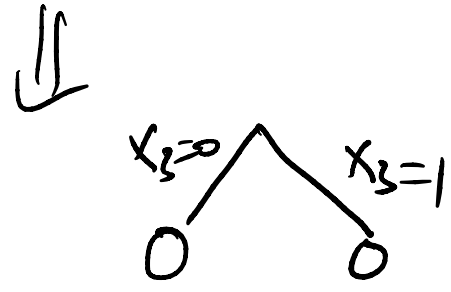






LB: 10 11 8 9 14 15

$\bar{z}$ 10 $P_2^- = \frac{2}{0.5}$ 8 $P_2^+ = \frac{3}{0.5}$ 14



Pseudo-cost branching

- Run strong branching for a small number of iterations to calculate the pseudo-cost for each variable

$$P_j^- = \frac{Z_{LP} - Z_j^-}{f_j} \text{ and } P_j^+ = \frac{Z_{LP} - Z_j^+}{1 - f_j}$$

Let Z_j^- be the cost when resolving the LP with $x_j \leq \lfloor x_j^* \rfloor$, with Z_j^+ being solve LP with $x_j \geq \lceil x_j^* \rceil$. Each strong branching step will obtain pseudo-costs for all the fractional variables. Take the average of these pseudo-costs over the first few strong branching iterations.

- After a few strong branching iterations, we use the pseudo-costs to estimate the bound improvement instead.

Let the estimated bound improvements be

$$D_j^- = f_j P_j^- \text{ and } D_j^+ = (1 - f_j) P_j^+$$

Pick

$$\hat{j} = \text{argmax}_j \left[\min \left(D_j^-, D_j^+ \right) \right]$$

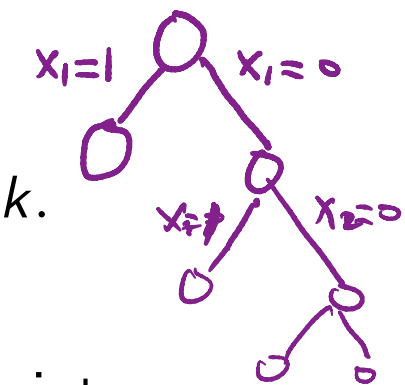
Branching priority

- The user may have better knowledge about their application. The MILP solvers usually allow the user to specify branching priority
- Example: After branching on binary variables y, the values of binary variables z (depend on y through constraints) will be fixed. Then we should branch on y first.
- Unit commitment problem has Start-up and shut-down conditions:
  $y_{gt} - z_{gt} = u_{gt} - u_{g,t-1}$ for all g, t.
  Once u is fixed, y and z are fixed.
- Gurobi branching priority: `https://www.gurobi.com/documentation/current/refman/branchpriority.html`
- This technique can reduce solution time significantly for some problems, e.g., scheduling problems.

Branching Rule on Special Ordered Sets

- A Special Ordered Set of type 1 (SOS1) consists of variables $x_1, \ldots, x_k$, where at most one variable can be positive.
- In a generalized upper bound (GUB) constraint scenario, the set is defined as:

$$\sum_{j=1}^{k} x_j = 1, \quad x_j \in \{0, 1\} \text{ for } j = 1, \ldots, k.$$



- Standard branching rule divides the solution space into:
  - $S_1 = S \cap \{x : x_j = 0\}$ and
  - $S_2 = S \cap \{x : x_j = 1\}$ for some j.
- This typically results in an unbalanced tree due to the SOS1 constraint.

SOS1 Branching Strategy

- A more balanced division of S into S_1 and S_2 is achieved by specifying an ordering $j_1, \ldots, j_k$ of the variables.
- Branching scheme:

$$S_1 = S \cap \{x : x_{j_i} = 0 \text{ for } i = 1, \ldots, r\} \text{ and}$$
$$S_2 = S \cap \{x : x_{j_i} = 0 \text{ for } i = r+1, \ldots, k\}.$$

where $r = \min \left\{ t : \sum_{i=1}^{t} x^*_{j_i} \geq \frac{1}{2} \right\}$

SOS2 and piecewise linear functions

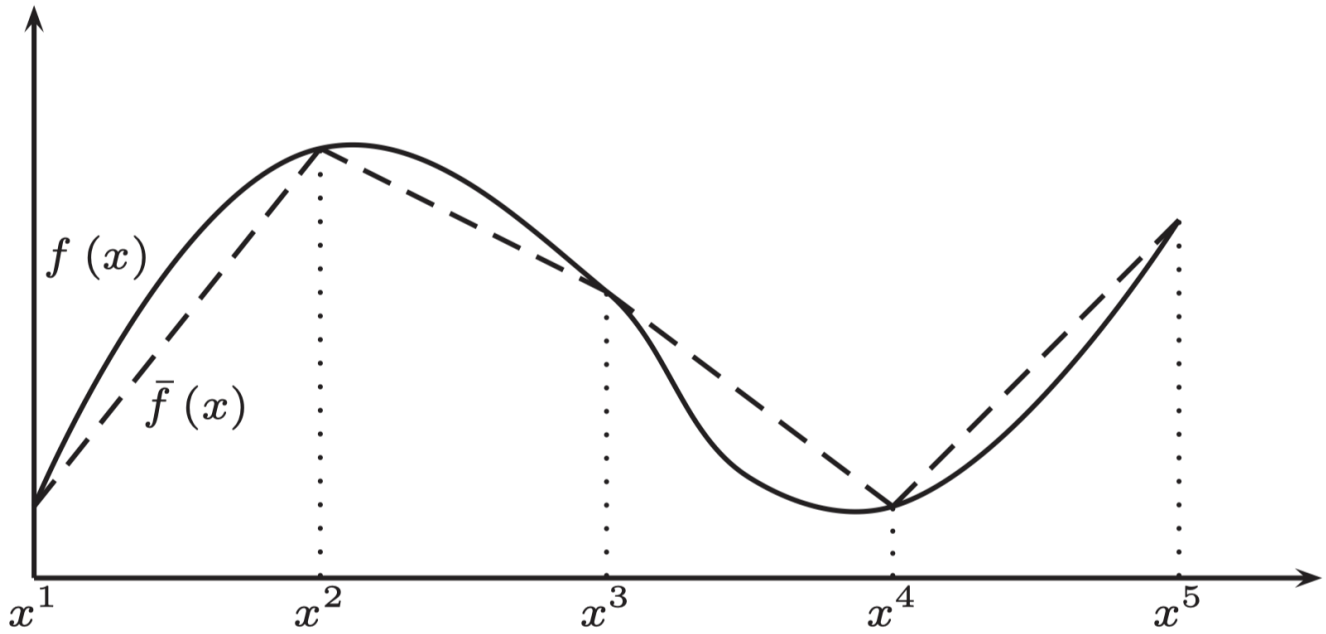


Figure: Piecewise linear approximation of nonlinear function

SOS2

Definition

A set of variables of which at most two can be positive. If two are positive, they must be adjacent in the set.

- Typically modeled using special ordered sets of type 2.
- The adjacency conditions of SOS2 are enforced by the solution algorithm.
- All commercial solvers allow you to specify SOS2 constraints.

$$x = \sum_{i=1}^{k} \lambda_i x^i$$

$$f(x) = \sum_{i=1}^{k} \lambda_i f(x^i)$$

$$\sum_{i=1}^{k} \lambda_i = 1$$

$$\lambda_i \geq 0 \quad \forall i$$

$$\lambda_1, \ldots, \lambda_k \text{ SOS2}$$

SOS in solvers

1. Gurobi: `https://www.gurobi.com/documentation/current/refman/sos_constraints.html`
2. Implementation in pyomo: `https://pyomo.readthedocs.io/en/stable/library_reference/kernel/sos.html`

Symmetry

Many problem formulations contain symmetries, meaning that certain variables can be interchanged so that one feasible solution is converted into another feasible solution of the same value. This is the case in graph coloring in which the colors can be interchanged, in room assignments, when the rooms are identical, or in capacity expansion planning problems where all the units are identical.

Example

$$\max 3\sum_{j=1}^{3} x_j + 2\sum_{j=4}^{6} x_j$$

$$x_1 + x_2 + x_3 \leq 2$$

$$x_4 + x_5 + x_6 \geq 1$$

$$x_4 + x_5 + x_6 \leq 2$$

$$x \in \{0,1\}^6.$$

It is not difficult to see that variables $\{1,2,3\}$ are interchangeable and also variables $\{4,5,6\}$.
More specifically, $(x_1, x_2, x_3) = (0,0,1), (0,1,0), (1,0,0)$ are identical solutions. Similarly, $(x_4, x_5, x_6) = (0,0,1), (0,1,0), (1,0,0)$ are identical. However, the standard branching rules are not aware of that.

Symmetry-breaking constraints

Add symmetry breaking constraints

$$x_1 \geq x_2 \geq x_3$$

$$x_4 \geq x_5 \geq x_6$$

With these constraints, the only feasible solution is $(x_1, x_2, x_3) = (1,0,0)$. $(x_4, x_5, x_6) = (1,0,0)$. For example, in room assignment problem, the second and the third room can only be used after the first room is assigned.

- Practical implication: consider adding these symmetry breaking constraints to your MILP model.

Orbital branching

Besides add symmetry breaking constraints, orbital branching is a branching rule that leverages symmetry.

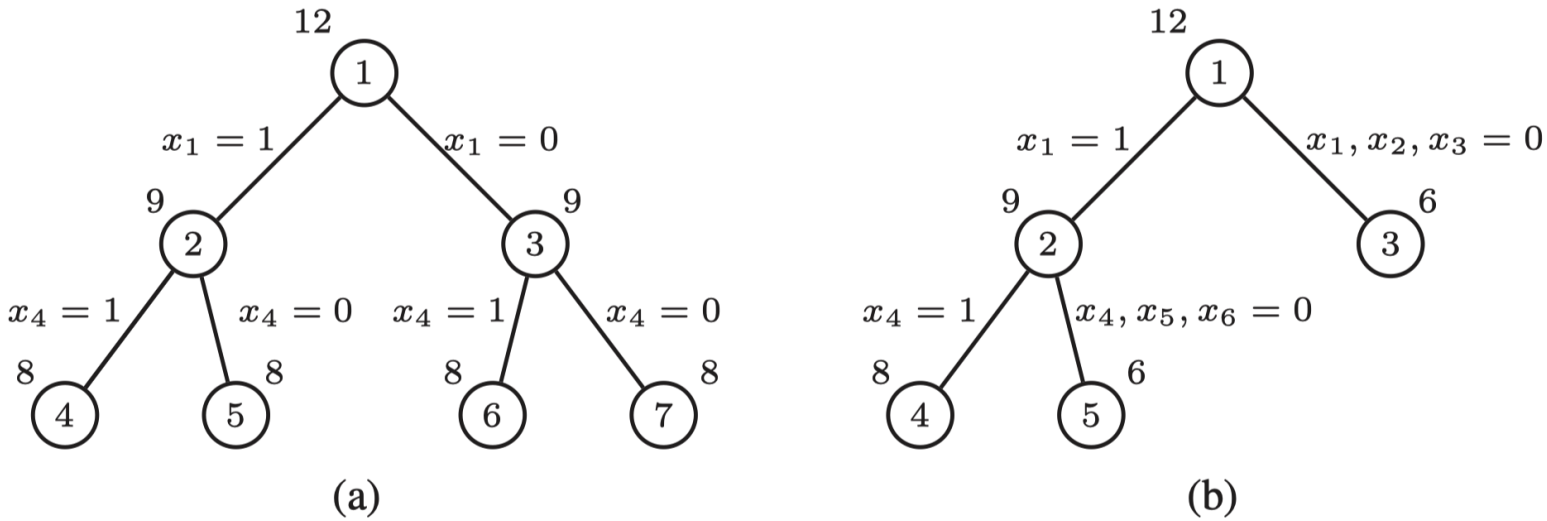


Figure: (a) standard branching. (b) orbital branching

- Practical considerations: implementing orbital branching is not as easy as symmetry breaking constraints. If you have a problem with symmetry, implementing symmetry breaking constraints is recommended.

Preprocessing /presolve

- Idea: Quickly detect and eliminate redundant constraints and variables, and tighten bounds where possible. Then if the resulting linear/integer program is smaller/tighter, it will typically be solved much more quickly.
  - Tighten variable bounds
  - Remove redundant variables and constraints
  - Variable fixing
  - Strengthen inequalities involving binary variables.

References

- Wolsey, L. A. (2020). Integer programming. John Wiley & Sons.
- Achterberg, T. (2007). Constraint integer programming. (PhD thesis)