Branch-and-bound

Giovanni Righini

Università degli Studi di Milano

Operations Research Complements

Università degli Studi
di Milano

Branch-and-bound

A branch-and-bound algorithm works as follows.

- A difficult optimization problem $\mathcal{P}$ is recursively decomposed into several easier sub-problems $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$.
  The decomposition (called *branching*) must obey the following condition to ensure the correctness of the algorithm:

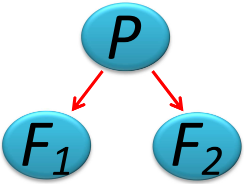
$$\mathcal{X}(\mathcal{P}) = \bigcup_{i=1}^{n} \mathcal{X}(\mathcal{F}_i).$$

- The optimal solution of $\mathcal{P}$ is determined by comparing the optimal solutions of the corresponding sub-problems.
  Assuming minimization:

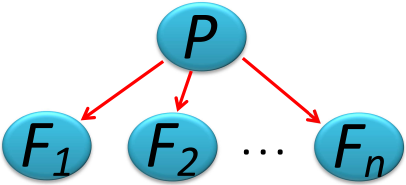
$$z^*(\mathcal{P}) = \min_{i=1,\ldots,n} \{z^*(\mathcal{F}_i)\}.$$

The branch-and-bound tree

The recursive decomposition of problems into sub-problems generates an arborescence (also called *decision tree* or *search tree*), where the root corresponds to the original problem $\mathcal{P}$ and each node corresponds to a sub-problem.



Binary branching



n-ary branching

Branching

For the sake of efficiency, branching usually implies a partition of $\mathcal{X}(\mathcal{P})$ into disjoint sub-sets (so that no solution be considered twice or more):

$$\mathcal{X}(\mathcal{F}_i) \cap \mathcal{X}(\mathcal{F}_j) = \emptyset \ \ \forall i \neq j = 1, \dots, n.$$

There are two main ways for branching:

- variable fixing;
- constraint insertion.

Every sub-problem is a restriction of its predecessor and a relaxation of its successors.

Binary branching

Common binary branching rules are as follows.

- **Branching on a binary variable.**
  A binary variable x is selected (*branching variable*).
  Then two sub-problems are generated by fixing $x = 0$ in a sub-problem and $x = 1$ in the other.

- **Branching on an integer constraint.**
  A vector of integer variables $(x_1, x_2, \ldots, x_n)$, a vector of suitable integer coefficients $(a_1, a_2, \ldots, a_n)$ and a suitable integer k are selected.
  Then two sub-problems are generated by inserting the constraints $ax \leq k$ in a sub-problem and $ax \geq k + 1$ in the other.

n-ary branching

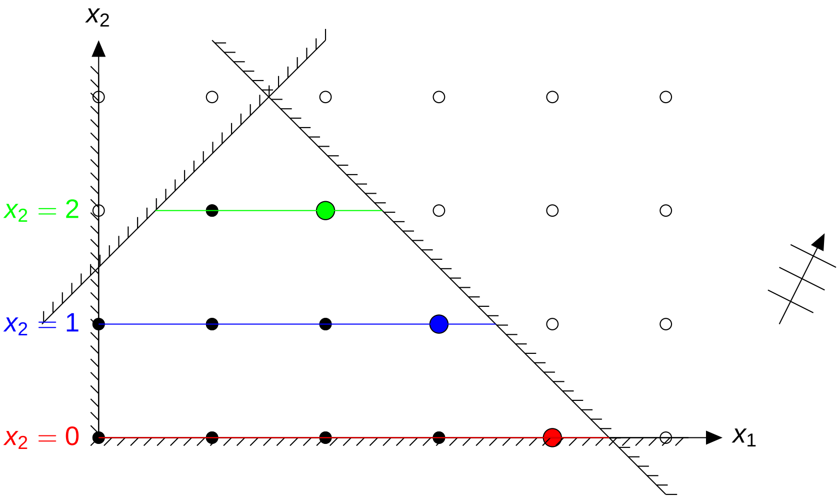
Common n-ary branching rules are as follows.

- **Branching on an integer variable.**
  An integer variable $x \in [1, \ldots, n]$ is selected (*branching variable*).
  Then n sub-problems are generated by fixing $x = 1$, $x = 2$,$\ldots$,
  $x = n$.

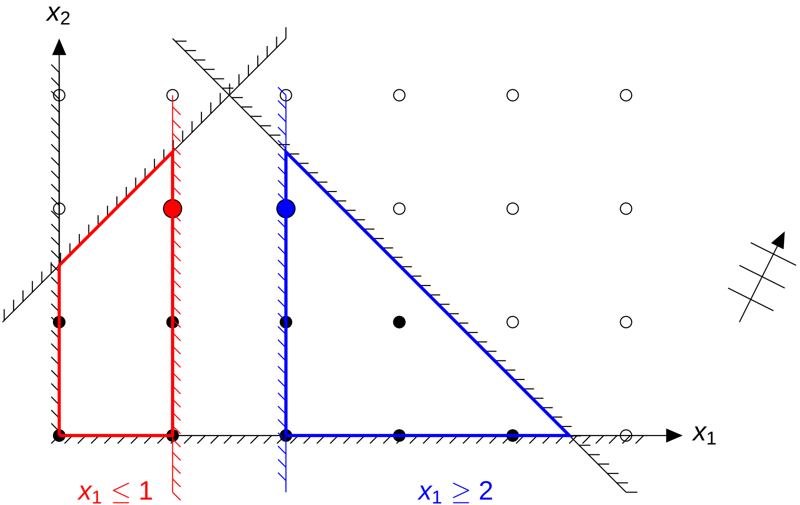
- **Branching on n binary variables.**
  A vector of n binary variables $(x_1, x_2, \ldots, x_n)$ is selected.
  Then $n + 1$ sub-problems are generated by fixing them as follows
  (one row for each sub-problem):

$$
\begin{aligned}
& x_1 = 1 \\
& x_1 = 0, x_2 = 1 \\
& x_1 = x_2 = 0, x_3 = 1 \\
& \ldots \\
& x_1 = x_2 = \ldots = x_{n-1} = 0, x_n = 1 \\
& x_1 = x_2 = \ldots = x_n = 0
\end{aligned}
$$

Branching by variable fixing



Branching by constraint insertion



Search

Every time two or more sub-problems are generated by a branching operation, they are appended to a list of open nodes, i.e. of sub-problems that still need to be solved.

This is necessary because a serial computer cannot examine and solve all sub-problems in parallel.

The policy followed to decide which nodes must explored first is also called search strategy.

We call *current sub-problem* the sub-problem that must be solved at any generic point in time during the search.

Leaves of the tree

Usually a sub-problem is “solved” by branching, i.e. by replacing it with other sub-problems. However, this recursive branching stops when:

- the current sub-problem is detected to be infeasible;
- the current sub-problem is solved to optimality;
- the current sub-problem can be fathomed.

All these three cases (the leaves of the branch-and-bound tree) can be detected by solving a relaxation of the current sub-problem.

Relaxations

As a consequence of the definition of relaxation, these corollaries hold.

Corollary 1. If $\mathcal{R}$ is infeasible, then $\mathcal{P}$ is also infeasible.

Corollary 2. If x^* is optimal for $\mathcal{R}$ and it is feasible for $\mathcal{P}$ and $z_{\mathcal{R}}(x) = z_{\mathcal{P}}(x)$, then x^* is also optimal for $\mathcal{P}$.

Corollary 3. If $z^*_{\mathcal{R}} \geq \bar{z}$, then $z^*_{\mathcal{P}} \geq \bar{z}$.

In branch-and-bound algorithms Corollary 3 is exploited by bounding.

Bounding

Bounding is the operation of associating a dual bound with each *sub-problem* $\mathcal{F}$.

Since

$$z^*_{\mathcal{R}} \leq z^*_{\mathcal{P}}$$

the optimal value of $\mathcal{R}(\mathcal{F})$ (a relaxation of $\mathcal{F}$) provides a dual bound to any sub-problem $\mathcal{F}$:

$$z^*_{\mathcal{R}(\mathcal{F})} \leq z^*_{\mathcal{F}}$$

The dual bound is compared against a primal bound that

corresponds to the value $z_{\mathcal{P}}(\bar{x})$ of a feasible solution $\bar{x} \in \mathcal{X}(\mathcal{P})$.

If the dual bound of $\mathcal{F}$ turns out to be no better than the primal bound, then $\mathcal{F}$ can be fathomed.

$$\text{If } z^*_{\mathcal{R}(F)} \geq z_{\mathcal{P}}(\bar{x}) \text{ then Fathom } \mathcal{F}.$$

Bounding

The correctness of the bounding operation relies upon the concatenation of two inequalities.

- The first inequality guarantees that no solution can exist in $\mathcal{X}(\mathcal{F})$ with a value better than $z^*_{\mathcal{R}(\mathcal{F})}$, since

$$z^*_{\mathcal{F}} \geq z^*_{\mathcal{R}(\mathcal{F})}.$$

- The second inequality is $z^*_{\mathcal{R}(\mathcal{F})} \geq z_{\mathcal{P}}(\bar{x})$.

By concatenating them, we can conclude that

$$z^*_{\mathcal{F}} \geq z^*_{\mathcal{R}(\mathcal{F})} \geq z_{\mathcal{P}}(\bar{x})$$

which means that solving sub-problem $\mathcal{F}$ to optimality is useless, because it cannot provide any feasible solution better than the one we already know, i.e. $\bar{x}$.

Fathoming sub-problems in a branch-and-bound algorithm is crucial to save computing time and memory space.

Bounding

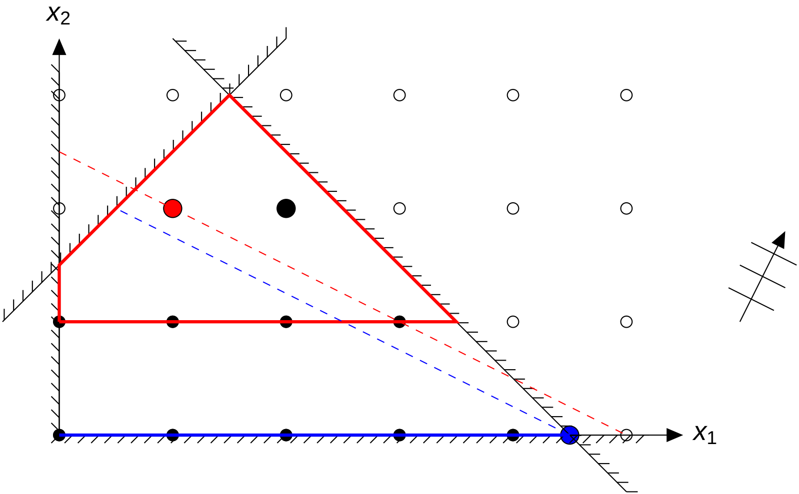


Figure: The blue sub-problem can be fathomed.

Search strategies

Different criteria to manage the list of open nodes correspond to different search strategies:

- FIFO: breadth-first search
- LIFO: depth-first search
- Sorted list: best-first search

Best-first search is usually based on the best dual bound criterion: the most promising sub-problems are explored first.

To keep a sorted list, it is useful to employ a *heap*.