Branch and Bound

Recherche opérationnelle
Dimitri Watel - ENSIIE

2024

The Branch and Bound method is a method for constructing an exact algorithm to solve a combinatorial optimization problem. The idea is to use a complete enumeration algorithm in a tree-like structure to which additional information is added to avoid enumerating all the solutions.

1 Examples

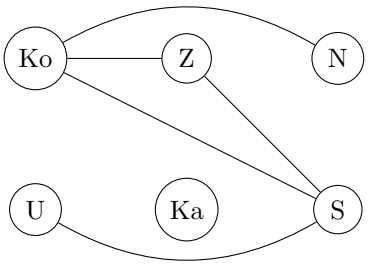
In this section, we will describe the method through two examples. The next section will synthesize these examples and detail the algorithms. The two examples provided here are graph problems, but of course, the technique can be applied to any combinatorial problem (where the decision is discrete).

1.1 The maximum independent set problem

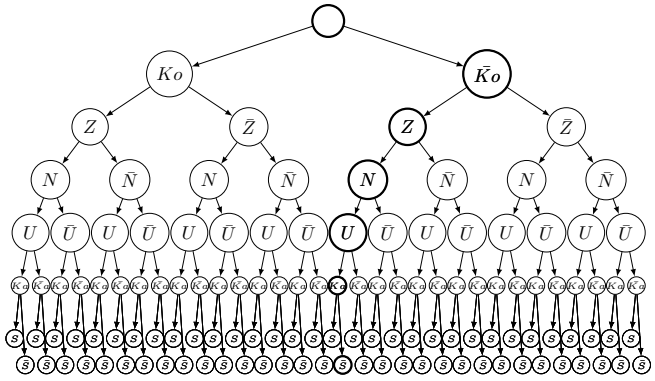
An *independent set* in a graph $G = (V, E)$ is a subset of nodes $V' \subset V$ such that no two nodes in V' are connected by an edge from E.

Problem 1. The maximum stable problem consists, given a graph $G = (V, E)$, in the search for an independent set V' of G such that the size $|V'|$ is maximized.

This problem can be solved fairly simply by testing all possible subsets V'. One way to generate these subsets is to use a tree representation of the solutions, which is the branching step. Consider the following example:



A tree would be the following, where at each internal node of the tree, a decision is made to include or not include one of the vertices of the graph in V'. Each branch of the tree connecting the root to a leaf is a solution. This solution may be infeasible (as is the case with the left branch). An optimal solution can then be extracted from the tree by looking at, among the feasible solutions, those that maximize the number of nodes in V'.



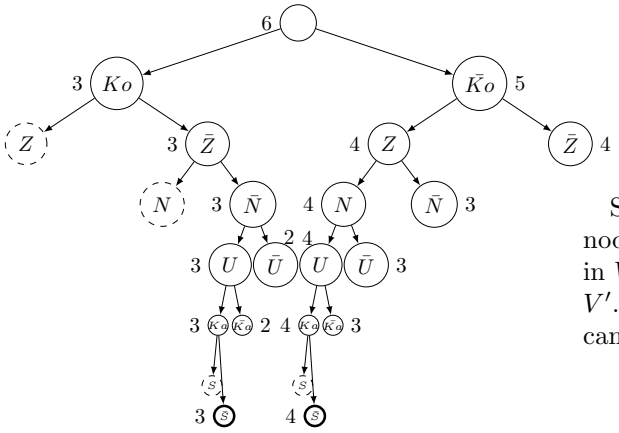
The goal of a branch and bound algorithm is to not explore, construct, or store the entire tree by cutting branches that do not lead to an optimal solution. There are two methods for cutting a branch.

- The first consists of revealing unfeasible decisions. For example, here, any leaf whose branch starts with Ko and Z is unfeasible because no independent set contains these two nodes.
- A second idea is to look at the weight of the best solution and the weight of the current solution. If we explore the tree by going deep to the left, the first solution we will reveal is $Ko, \bar{Z}, \bar{N}, U, Ka, \bar{S}$. We thus have a solution with 3 nodes. Now consider the branch $\bar{K}o, Z, \bar{N}$, which includes all solutions containing Z and excluding both Ko and N. Regardless of the choices made for U, Ka, and S, we can never have more than 3 vertices in the independent set. Indeed, Z prevents the selection of S, so this stable will contain, at best, Z, U, and Ka. Since we have already found a solution with 3 nodes, it is unnecessary to explore the solutions arising from the choices $\bar{K}o, Z, \bar{N}$ (we remind that we seek an optimal solution, not all optimal solutions).

  As an example, consider another branch, the branch $\bar{K}o$ and $\bar{Z}$. At best, any realizable solution arising

from this decision can contain only 4 vertices. Indeed, the vertices Ko and Z are not selected. Thus, there are 4 remaining vertices that we can hope to include in a realizable solution. If we have already found a solution with 4 vertices or more, there is no need to explore this branch further. Note with this example that this value of 4 is an upper bound. There is no solution of size 4 that does not include Ko or Z. Indeed, U and S are incompatible. Thus, we are dealing with an upper bound for the best possible solution and not the value of the best possible solution. The only way to know this latter value is to explore all the nodes descending from the branch.

By combining these two ideas, we obtain the following explored tree where the upper bound is indicated next to the node. The dotted nodes represent non feasible decisions.



We do find the optimal solution, but we explored many fewer nodes. We cut 4 branches because the solution was not feasible. We cut 6 branches because the upper bound indicated that we would not do better than the best solution. And we revealed two feasible solutions, with values of 3 and 4, respectively.

Here we performed a depth-first search to the left. We explored 13 nodes (the nodes that were not pruned). We could have done a depth-first search to the right, or a random depth-first search. The result would have been the same but the number of pruned and explored nodes would have been different. In the worst case, we are forced to reveal the entire tree. In the best case, we directly hit the optimal solution, and if the bounds are low, we prune all other nodes.

We will describe below another example using a different exploration method.

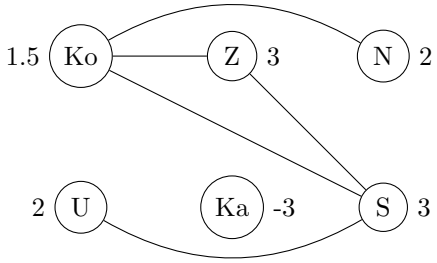
1.2 The maximum weighted selection problem

We consider a relaxed variant of the maximum independent set problem. We can now take two nodes in V' that are neighbors, but this incurs a penalty.

Problem 2. The maximum weight selection problem consists, given a graph $G = (V, E)$ and weights $\omega : V \to \mathbb{R}$, of finding a subset $V' \subset V$ that maximizes $\sum_{v \in V'} \omega(v) - |E[V']|$, where $E[V']$ are the edges of the subgraph induced by V'.

We can still solve this problem fairly simply by testing all possible subsets V' with a tree exploration. We can use the branch and bound technique, but we need a new upper bound.

Consider the following example where $\omega(v)$ is written next to the node v.



Suppose we have already made a decision for some nodes, let W^+ be the nodes that we have decided will be in V' and W^- those that we have decided will not be in V'. We still need to make a decision for the others. We can already count the value that these nodes bring:

$$b \leftarrow \sum_{v \in W^+} \omega(v) - |E[W^+]|$$

For example, consider above that $W^+ = \{K_o\}$ and $W^- = \emptyset$, then b starts with 1.5.

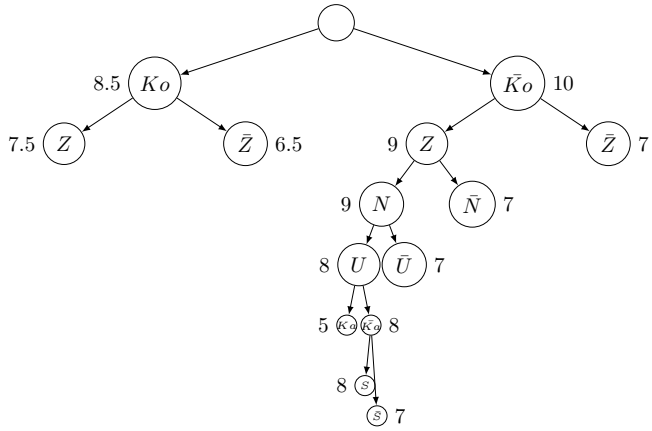
We then calculate, for each node $v \in V \backslash W^+ \cup W^-$, the following value. It corresponds to the gains and penalties that relate v and W^+. If this value is positive, we add it to b. Otherwise, we do nothing; we then consider that it is better not to add v to V'. This is the case, for example, if v introduces a lot of penalties.

$$c_v \leftarrow \omega(v) - |\{(u, v) \in E | u \in W^+\}|$$
$$b \leftarrow b + \max(c_v, 0)$$

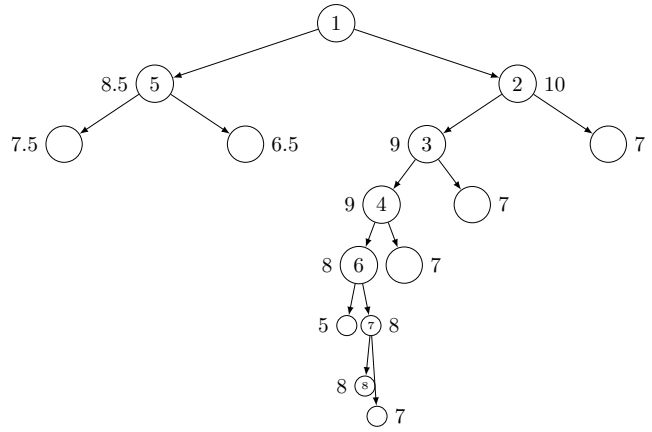
In our example, $c_Z = 2, c_N = 1, c_U = 2, c_{Ka} = -3$ and $c_S = 2$. Therefore, we have $b = 8.5$ at the end of the calculation. Once again, we are indeed talking about the upper bound of the best solution containing W^+ and not the optimal value among these solutions. There is no solution containing K_o with a weight of 8.5, but all these solutions have a weight less than 8.5. If we find

a solution with a value greater than 8.5, we know it is unnecessary to explore the branch starting at K_o.

Unlike the first example where we explored the tree deeply to the left, we will use a best-first search here, always exploring the node with the greatest bound in the tree.



The order in which the nodes were explored is indicated below. We explored 8 nodes and cut 7 nodes.



2 Summary

The branch and bound method is a method for constructing an exact algorithm for a combinatorial optimization problem.

The first step is to **separate the problem into several subproblems**. Generally, this branching is done by making a decision for one of the problem variables (in our examples, selecting or not selecting a node), but more general approaches can be taken depending on the problem we wish to solve.

Let Π be an optimization problem for which the set of feasible solutions is $\xi(\Pi)$. We say the function $S(\Pi)$ divide Π into subproblems if it returns a set of optimization problems $(\Pi_1, \Pi_2, \ldots \Pi_p)$ such that

$$\bigcup_{j=1}^{p} \xi(\Pi_j) = \xi(\Pi)$$

It is necessary, as far as possible, that the solutions of the subproblems intersect as little as possible to reduce the unnecessary search space. In the examples above, we have Π corresponding to the root node of the tree and Π_1, Π_2 corresponding to the nodes K_o and $\bar{K_o}$.

The second step is to **choose a bound** from the set of values of all solutions. In our two examples, we had a maximization problem. In this case, an upper bound is needed. **Note**, in the case of a minimization problem, a lower bound is required.

Let Π be a **minimization** problem. Let $\omega(s)$ be the value of a feasible solution s of $\xi(\Pi)$. Finally, let s^* be an optimal solution of Π (minimizing $\omega(s^*)$).

A bound $B(\Pi)$ is a number that **can be computed fast** satisfying

$$B(\Pi) \leq \omega(s^*)$$

In the case of a maximization problem, we will have $B(\Pi) \geq \omega(s^*)$. We recall that each problem can be divided into subproblems which themselves are optimization problems to which we can apply the bound B.

As mentioned above, the bound must be quick to calculate, as we will need to compute it for every node of the tree that we will reveal. However, the bound must also be effective; the closer it is to $\omega(s^*)$, the more likely we are to prune the current node. For example, in the maximum stable set problem, a very quick bound to calculate is $|V|$. This is indeed an upper bound of a stable set of G, but it is not a very useful bound since no node will be pruned due to this bound. Conversely, the best bound would be to calculate and return $\omega(s^*)$, but that would take a long time (possibly exponential time for each node of the tree).

The last step is to **determine an exploration strategy**. Given a partially revealed tree, which node should be explored? An important detail is to clearly define the notion of exploration.

Definition 1. *Exploring a node* Π of the tree means compute $S(\Pi)$ and $B(\pi)$ for all $\pi \in S(\Pi)$.

It is not because a node is displayed that it is explored. A node can be pruned because its bound is too small/large or because it corresponds to an infeasible solution. In this case, it is considered unexplored: its children $S(\Pi)$ in the tree are not revealed. The calculation of the bound of the node is not part of the exploration; it is part of the exploration of the parent node. The only exception is the root, which has no parent. However, the calculation of the bound of the root is rarely useful.

As we have seen, there are two simple exploration techniques, **leftmost depth-first exploration** which

always explores the leftmost unexplored node, and **best-first exploration** which always explores the unexplored node with the most promising bound. The first is less memory-intensive and quickly finds feasible solutions. The second immediately finds very good solutions but takes more time and memory to achieve them. Of course, one should not limit oneself to these techniques. Depending on the problem and the instance, one must choose the right technique. It is also possible to change tactics during the algorithm; there are no strict rules.

We end by giving pseudocode algorithms for the two standard exploration techniques.

Algorithm 1 Depth first search exploration

Require: a **minimization** problem Π, and a real $uB \geq \omega(s^*)$. The default value of uB is $+\infty$.
Ensure: The weight $\omega(s^*)$ of an optimal solution s^* of Π

```
1: function BB(Π, uB)
2:     if Π cannot be separated then return The weight ω(s*) of an optimal solution s* of Π
3:     Explore Π
4:     for Π_j ∈ S(Π) such that ξ(Π_j) ≠ ∅ do
5:         lb ← B(Π_j)
6:         if lb < ub then
7:             ω ← BB(Π_j, ub)
8:             ub ← min(ub, ω)
9:     return uB
```

Algorithm 2 Best first search exploration

Require: a **minimization** problem Π
Ensure: The weight $\omega(s^*)$ of an optimal solution s^* of Π

```
1: function BB(Π)
2:     L ← [Π] ; uB ← +∞
3:     while L ≠ ∅ do
4:         π ← arg min_{π∈L}(B(π)) ; Remove π from L
5:         if B(π) ≥ ub then
6:             Break
7:         if π cannot be separated then
8:             ub = min(ub, Weight ω(s*) of an opt sol s* of π)
9:         else
10:            Explore Π
11:            Add all elements of S(Π) to L
12:    return uB
```