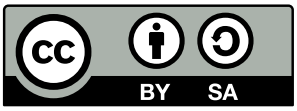
Greedy Algorithms

Master MoSIG — Algorithms and Program Design

Florent Bouchez Tichadou

June 2020



Objectives:

- Learn the limitations of the most basics of algorithms.

1 Greedy Algoritms

Greedy algorithms strategy is a meta-heuristic used to solve optimisation problems. It is applied on combinatory optimization problems, where we try to make a selection in building a solution amongst a large set of possible while optimizing a metric (minimize or maximize). Here are some examples:

- Maximizing the number of tasks we can accomplished in a given time without overlapping;
- Minimizing the length of the path connecting to vertices in a graph.

What makes a greedy algorithm *greedy* is the way decisions of selection are done. Each of the decisions is simply made from a local point of view, favoring what looks the best choice given a very limited knowledge. This makes these algorithms simple to implement and quite fast. However as the decision does not take into account the whole problem, it often fails at finding the optimal solution.

1.1 Structure

Greedy algorithm structure is quite simple to overtake. The greedy algorithms take decision, selecting the current optimal element of the set based only on local information. It does not take into account past nor future selections.

In other words, the same decision will be applied on each subproblem, created from the original problem after having applied local greedy choices. The greedy decision will be evaluated on every subproblem, leading to smaller and smaller problems until the problem is solved.

The greedy decision is definitive. Each time a decision (selection) is taken by the greedy algorithm, it will never come back on this choice. More precisely, it will not try to explore over solutions. In that condition, the local selection will be part of the final solution found by the greedy algorithm.

As described before, greedy algorithm are intrinsically recursive algorithms. Applying a decision and calling itself to apply the same decision one a smaller subproblem. However, they usually are almost always terminal recursive functions, hence it is then often possible to write greedy algorithms as iterative ones.

1.2 Choices in the Greedy Strategy

The most difficult part of greedy algorithms is the definition of the locally optimal choice.

As stated before, this choice is a purely local one. In the selection, for a given step of the algorithm, we can only rely on information available at this step. Past information and more important future step information are not available.

The idea is then trying to reach the optimal solution of the problem based on the sequence of local optimal choices. This, indeed, requires to define correctly the problem model in a way that each of the local decision gives good chances to reach this global optimal solution.

1.3 Optimality of Greedy Algorithms

Greedy algorithms usually do not succeed in giving the global optimal solution of problems but for highly particular cases. So it is quite safe to consider, in a first approach, that it fails at finding it...

The success condition is two folded. First the modelling of the problem should exhibit the correct structure, called optimal structure, to ensure greedy algorithm can find optimal solution. This means that no optimal solution can be found without fullfilling this constraint.

However, the problem itself is also a key for greedy algorithm strategy. Some usual problems can be solved using greedy heuristics, for example:

- Huffman code building,
- Dijkstra's shortest path,
- Prim's Minimal Spanning Tree, and so on.

These cases suits perfectly the greedy algorithm strategy and optimal structures are described in the algorithms. Unfortunately, in many other cases the optimal structure simply does not exists or is difficult to find. In these cases, greedy algorithms give only sub-optimal solutions to the problem.

1.4 Complexity

The complexity of greedy algorithms mainly depends on the complexity of the decision function. It depends on the way the problem has been modelled for the greedy algorithm resolution.

As far as the greedy algorithm strucutre is concerned, the complexity is close to the linear complexity. It is mainly due to the fact that each choice of the algoithm steps are part of the final solution.

However, the choices in the model stated before can dramatically influence the overall complexity. For example, using ordered sets instead of simple sets can on one hand help in resolving the greedy decision and on the other hand reduce the complexity of this greedy decision.

2 Example

2.1 Problem definition

Let us see an example of a greedy algorithm. We consider a tree where each node holds a value. Figure 1 gives two examples of such trees where each node hold i/w_i with i the label of the node and w_i the weight of this node. The objective is to find the path to a leaf of the tree that maximizes the sum of the weights of all node present on the path.

In the first example, the optimal choice in going to leaf 3 using path $0 \rightarrow 1 \rightarrow 3$. The total weight is $152 = 16 + 8 + 128$. In the second, the optimal path is $0 \rightarrow 2 \rightarrow 5$.

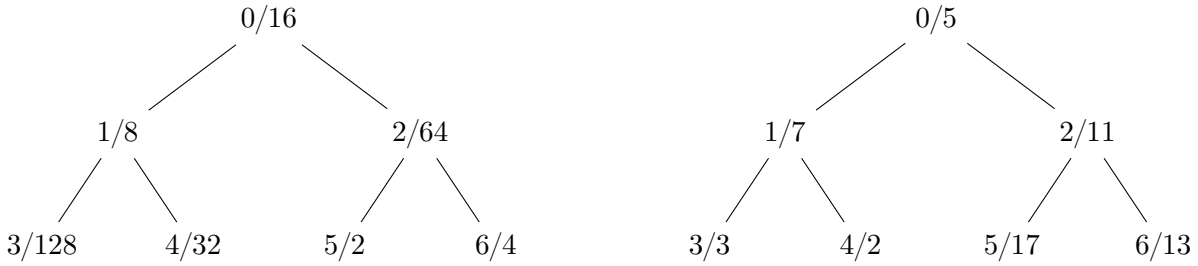


Figure 1: Tree examples

2.2 Greedy algorithm resolution

Looking for the optimal path given before can be easily reached by a quick look at the complete tree and find the heaviest nodes, to force passing through them. Let us imagine that the tree is made of a million nodes now. This strategy cannot apply anymore. We need the help of algorithmic to solve this problem. Let us try to solve it using a greedy algorithm.

First we need to model the problem. This task is easy in that case since the structure is a tree. The overall structure of our algorithm is taking a decision for each node. Once the decision is taken it will repeat recursively on a sub-tree until reaching a leaf. The starting point is the root of the tree.

In that case the local information is the weights of the children of the current node, and the best local choice is to choose the child with the largest weight.

To summarize, our greedy algorithm selects the heaviest child for each node visited, starting at the root and ending on a leaf. The algorithm below represents this behavior, using an iterative version. The recursive version is straightforward and is easy to de-recurse to reach this iterative version.

```
1  queue ← ∅;
2  node ← {root};
3  weight ← 0;
4  while node ≠ ∅ do
5      queue ← node;
6      weight ← weight + node.weight;
7      if node is a leaf then
8          node ← ∅;
9      else
10         if node.left.weight <
             node.right.weight then
11             node ← node.right;
12         else
13             node ← node.left;
```

Complexity: O(height of tree)

```
void greedy_search_path(struct node *n) {
  queue *queue;
  queue_init (queue);
  int weight = 0;

  struct node *node = n;

  while(node != NULL){
    queue_append(queue, node);
    weight += node->weight;

    if(node->left == NULL){
      node = NULL;
    }else{
      if(node->left->weight < node->right
          ->weight){
        node = node->right;
      }else{
        node = node->left;
      }
  }
}
```

2.3 Optimality

Let us apply this greedy algorithm on our small examples. For example 2, the greedy algorithm depicted before will chose path $0 \to 2 \to 5$, which is the optimal path. Indeed, at the first stage,

it will have to decide between nodes 1 and 2, with respective weights 7 and 11. Of course it will choose node 2 and select between nodes 5 and 6 at next step. At this second step, the choice is between weights 17 and 13 for respectively nodes 4 and 5. The algorithm selects node 5 and ends as it is a leaf. The greedy algorithm succeeded at finding the optimal path of the tree example 2.

However, the application on example 1 is not as good. Indeed, the first step is a choice between weights 8 and 64 (for nodes 1 and 2). Based on this purely local choice, our greedy algorithm will select node 2 as it is the local best choice. The second step is then the selection between nodes 5 and 6 whose weights are 2 and 4. It will select node number 6 and get a final weight of 84 for path $0 \rightarrow 2 \rightarrow 6$. This result, as shown before, is not the tree optimal one, since we can reach 152 with path $0 \rightarrow 1 \rightarrow 3$.

This is an example where the greedy algorithm cannot reach the optimal solution, but only suboptimal ones. In some cases this suboptimal solution is the optimal one (as shown on example 2), but it is not at all guarantied by the algorithm. Reaching the optimal solution in these conditions is mainly a question of luck in the repartition of data.