39. Greedy Algorithms for Solving NP-Complete Problems

1 Ali-Baba Problem: an Example of an NP-Complete Problem

Actual Ali-Baba problem. In one of the stories from the Arabian Nights, Ali-Baba, when traveling with his donkey, found a cave where 40 thieves have been hiding their loot. As a law-abiding citizen, of course, he reported it to the police – not really, he decided to steal as much as possible for himself. The problem is that there is too much loot, and the donkey cannot carry it all, there is a limit W on the weight the donkey can carry.

Let us assume that the thieves were well-organized, so each stolen item has a label on which they indicated the price p_i and the weight w_i of each item. The question is: can Ali-Baba carry enough items so that they price is at least equal to some given value P? and if yes, which items should be select?

Towards formulating this problem in precise terms. Similarly to the exact change problem, let us introduce, for each item i, a variable t_i which is equal to 1 if we select the i-th item and to 0 if we do not. Then, the overall weight of all selected items is

$$t_1 \cdot w_1 + \ldots + t_n \cdot w_n$$

and their overall price is

$$t_1 \cdot p_1 + \ldots + t_n \cdot p_n.$$

Formulating the problem in precise terms.

- *given:* an integer n, n positive values $w_1, \ldots, w_n$, n positive values $p_1, \ldots, p_n$, numbers W and P;
- *find:* the values $t_i \in \{0, 1\}$ for which $t_1 \cdot w_1 + \ldots + t_n \cdot w_n \le W$ and $t_1 \cdot p_1 + \ldots + t_n \cdot p_n \ge P$ (or generate a message that such a choice is not possible).

This problem is also known as a *knapsack* problem, since a similar choice happens if, instead of a donkey, we can only carry a knapsack of limited capacity.

This problem is indeed NP-complete. Indeed, we can reduce the exact change (subset sum) problem to this one: for any coins $s_1, \ldots, s_n$ and for each amount S, we can take $w_i = p_i = s_i$ and $W = P = S$. Then, the two desired inequality become $t_1 \cdot s_1 + \ldots + t_n \cdot s_n \le S$ and $t_1 \cdot s_1 + \ldots + t_n \cdot s_n \ge S$.

These two inequalities are equivalent to the equality $t_1 \cdot s_1 + \ldots + t_n \cdot s_n = S$ – which is exactly what we wanted in the exact change problem.

So what do we do? OK, the problem is NP-complete, so, unless P = NP, we cannot hope to have a feasible algorithm for solving all the instances of this problem.

But we still need to solve these problems in practice. So, let us use *heuristic* algorithms, algorithms that do not always succeed.

2 Greedy Algorithms

What does greedy mean. A greedy person is a person who wants to get as much money now, without thinking of long-terms consequence – such as going to jail, losing friends, or ruining the environment.

What does a greedy algorithm mean. For people, greedy is bad. For NP-complete problems, when we cannot feasibly consider all the consequences, some algorithms like that are unavoidable.

First greedy algorithm for the Ali Baba problem. Ali Baba clearly wants to maximize the price of his loot.

- So, on the first step, a natural idea is to carry the most expensive item that the donkey can carry, i.e., the most expensive item i among the items for which $w_i \le W$.
- If for the selected item i, we have $w_i < W$, this means that the donkey can carry an additional weight of $W - w_i$. Thus, we select the most expensive item j among those for which $w_j \le W - w_i$.
- If there is still some weight available, then we pick the third item – etc., until either the donkey is fully loaded, or the remaining weight is so small, that none of the remaining items can fit.

First example. Suppose that we have 7 items:

- a big silver necklace that weighs $w_1 = 9.6$ kg and whose price is $p_1 = 3000$ units;
- five identical golden watches each of which weighs $w_2 = w_3 = w_4 = w_5 = w_6 = 0.5$ kg and whose price is $p_2 = p_3 = p_3 = p_5 = p_6 = 2500$ units; and
- a small diamond that weighs $w_7 = 0.01$ kg and whose price is $p_7 = 2000$ units.

Suppose that the donkey can carry $W = 10$ kg. Then:

- In this example, each of the items can fit on a donkey, i.e., we have $w_i \le W$ for all i. Thus, on the first step, we select the most expensive item – the big silver necklace, i.e., we set $t_1 = 1$.
- After this selection, the remaining weight is $W - w_1 = 10 - 9.6 = 0.4$ kg. The only remaining item that can still fit is a small diamond, so we take it: $t_7 = 1$.
- Now, the remaining weight is $W - w_1 - w_7 = 0.39$ kg, we cannot fit any other item, so we stop.

The resulting arrangement is $t_1 = 1$, $t_2 = t_3 = t_4 = t_5 = t_6 = 0$, and $t_7 = 1$.

The price of all the selected items is $3000 + 2000 = 5000$. So, if $P \le 5000$, we solved the problem. So far, so good.

Will it always work? Of course, since

- the problem is NP-complete – meaning that no feasible algorithm is expected, and
- the greedy algorithm *is* clearly feasible,

we cannot expect that this algorithm will always work.

And indeed, it is easy to find an example when the above greedy algorithm will not be able to find a solution.

Modified example. What if Ali-Baba wanted $P = 10000$? In this case:

- the above greedy algorithm would still lead to the same selection with the overall price 5000, which, in this case, is not a solution,
- but a solution is possible: e.g., we can take all 5 golden watches.

How can we improve the situation? From the viewpoint of common sense, since we are limited by weight, it makes sense to select items for which the price per kilogram is the largest. This leads us to the second greedy algorithm.

Second greedy algorithm for the Ali Baba problem.

- On the first step, among all the items i that fit on a donkey (i.e., for which $w_i \le W$), we select an item with the largest possible price-to-weight ratio p_i/w_i.
- If for the selected item, we have $w_i < W$, this means that the donkey can carry an additional weight of $W - w_i$. Thus, among all the items j that still fit on a donkey (i.e., for which $w_i \le W - w_i$), we select an item with the largest possible price-to-weight ratio p_j/w_j, etc.
- We continue either the donkey is fully loaded, or the remaining weight is so small, that none of the remaining items can fit.

How will this work on the above example? For the above seven items, the price-per-weight ratios are

$$p_1/w_1 \approx 300, \quad p_2/w_2 = \ldots = p_6/w_6 = 5000, \text{ and } p_7/w_7 = 200000.$$

So, clearly:

- first, we take the diamond,
- then we take the 5 watches, and
- after that, we cannot fit anything else on a donkey.

Here, $t_1 = 0$ and $t_2 = \ldots = t_6 = t_7 = 1$, and the overall price is $5 \cdot 2500 + 2000 = 14500 > 10000$, which solves the modified problem.

Is the second algorithm a panacea? The second algorithm cannot be a panacea either, since:

- the problem is still NP-complete and
- the algorithm is still feasible.

It is easy to find an example when neither of the two greedy algorithms can solve the problem. Let us assume that we have three items:

- the first item has $p_1 = 15$ and $w_1 = 14$; and
- the second and the third items are identical, they have $p_2 = p_3 = 10$ and $w_2 = w_3 = 10$.

Here, $W = 20$ and $P = 20$.

- In the first greedy algorithm, we first select the most expensive item, i.e., Item 1. After this, there is no room left for anything else, and the resulting price is $15 < 20$.
- In the second greedy algorithm, we first select the item with the largest price-to-weight ratio, i.e., again Item 1. So again, we get $15 < 20$.
- However, the solution does exist: namely, pick the second and the third items.

Other greedy algorithms. Greedy algorithms can be designed for other NP-complete problems as well.

For example, SAT means finding the values of the Boolean variables that make all the clauses true. So, it makes sense to first select the literals that makes the largest number of clauses true.

Let us illustrate it on the example of the following formula:

$$(x_1 \vee x_2 \vee x_3) \,\&\, (\neg x_1 \vee \neg x_2 \vee \neg x_3) \,\&\, (x_1 \vee \neg x_2 \vee \neg x_3) \,\&\, (x_1 \vee \neg x_2 \vee x_3),$$

with four clauses:

$C_1 = x_1 \vee x_2 \vee x_3,\ C_2 = \neg x_1 \vee \neg x_2 \vee \neg x_3,\ C_3 = x_1 \vee \neg x_2 \vee \neg x_3,\ C_4 = x_1 \vee \neg x_2 \vee x_3.$

Here, we have 6 possible literals: x_1, $\neg x_1$, x_2, $\neg x_2$, x_3, and $\neg x_3$. Let us see how many clauses are satisfied if we assume that each of these literals is true:

- if we take x_1 to be true, then 3 clauses will be satisfied: C_1, C_3, and C_4;
- if we take $\neg x_1$ to be true, then only 1 clause will be satisfied: C_2;
- if we take x_2 to be true, then 1 clause will be satisfied: C_1;
- if we take $\neg x_2$ to be true, then 3 clauses will be satisfied: C_2, C_3, and C_4;
- if we take x_3 to be true, then 2 clauses will be satisfied: C_1 and C_4;
- if we take $\neg x_3$ to be true, then 2 clauses will be satisfied: C_2 and C_3.

So, we select either the literal x_1 or the literal $\neg x_2$ to be true. If we select x_1, the clauses C_1, C_3, and C_4 are already satisfied, and the clause C_2 takes the form $\neg x_2 \vee \neg x_3$.

For this clause, we can select either $\neg x_2$ or $\neg x_3$ to be true. For example, if we select $\neg x_2$ to be true, then we take $x_1 =$"true" and $x_2 =$"false". This is indeed a satisfying vector.