CentraleSupélec

Algorithmics and complexity

TD 4/7 – Dynamic Programming

The goal of this TD is to model problems and solve them using dynamic programming.

This TD is 3h and contains practice sessions in Python.

Exercise 1 : Trading policy

We have one Euro that we want to invest in an optimal way for 10 periods. We know exactly the gain factor c_i for an investment at the beginning of period i.

Period i	1	2	3	4	5	6	7	8	9	10
c_i	2	4	1	2	6	2	2	4	1	4

At the beginning of each period, we have two choices: we can either invest the totality of the available money, or keep the money and invest nothing. Let's assume that at the beginning of period i we have an amount of x. If we do not invest any money for period i, we will have x available at the beginning of period $i+1$. However, if we invest x at the beginning of period i, we will get $c_i x$, but only at the end of period $i+1$, so the money will not be available during two periods, and it will be possible to invest it again only at the beginning of period $i+2$.

We are looking for an investment policy which gives us the most money at the beginning of period 12 (so there will not be any investment at the beginning of period 11).

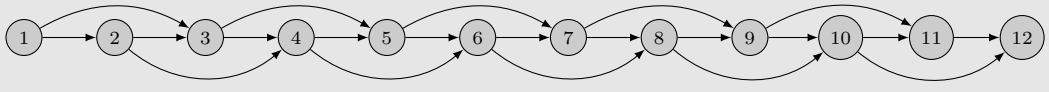
Question 1

By first ignoring the gain coefficients, model the problem by an non-weighted graph (the vertices and edges) where each total investment policy will correspond to a path on this graph. What is the nature of this graph? Draw the graph.

Solution elements :

The graph is **oriented**. There are 12 vertices which represent 12 moments (a beginning of period) when one decides to invest or not the available sum. The terminal vertex of each arc indicates the nearest beginning of period where the money will be available. Each vertex has therefore a outgoing degree of 2 (except for the last two periods) and an incoming degree of 2 (except the first two periods).

We obtain a **DAG**.



Question 2

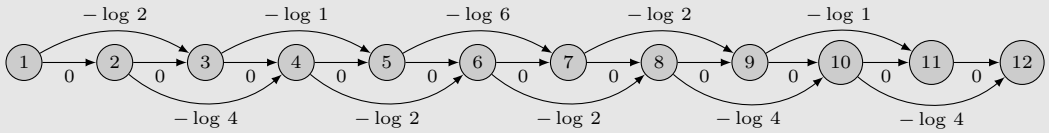
If we want to get back to the the shortest path problem, what weights should be assigned to the arcs? Complete the graph.

Solution elements :

It is a matter of **maximizing** a **multiplicative** function of c_i gain factors!

1. $c_i \mapsto \log(c_i)$ to pass to a function (of distances) that is **additive**
2. $\log(c_i) \mapsto -\log(c_i)$ to switch to a **minimization** problem

The weight of an arc between two non-successive vertices is $-\log(c_i)$ where c_i is the gain factor of the period of the starting vertex i. 0 between two successive vertices.



Question 3

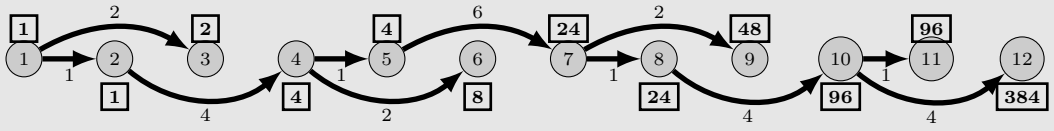
What algorithm seen in class can be used to respond to the exercise? Find the solution of the numerical example.

Solution elements :

In case of negative arcs, the Ford-Bellman algorithm should be used, for $i \in 1 \dots |V|$:

$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{u \in V}(\mathrm{OPT}(i-1, u) + \omega((u, v)))\right)$$

Here is the final result of the shortest path tree after running Ford-Bellman:



An optimal path from period 1 to 12 has a gain factor of 384! and uses the vertices [2, 4, 5, 7, 8, 10].

It should be noted that since this is a DAG, i.e. no cycles and in particular no absorbing cycles, the Ford-Bellman algo leads to the right solution.

To go further: As we naturally have a topological order on the nodes of the graph, we can consider the exercise of the previous TD2, and notice that the adaptation of the shortest paths algorithm studied in TD2 will work for this exercise also with a much better complexity!

Exercise 2 : Ski Rental

Decision Problem

A ski shop has n pairs of skis. There are m customers who wish to rent skis, with $m < n$. The shop owner wants to optimize the comfort of skiers. He wants to assign skis to skiers so that the sum of the absolute differences of the heights of each skier and his skis is minimized.

Question 1

Formalize this optimization problem.

Solution elements :

Inputs:

- a set of n integers $s_1, \ldots, s_n$ representing the lengths of each skis pair
- a set of m integers $c_1, \ldots, c_m$ representing the sizes of each customer

Output: an injective mapping function $f : [1, m] \to [1, n]$ such that $\sum_{i \in [1,m]} |c_i - s_{f(i)}|$ is minimal.

We can see that the complexity of a naive and exhaustive algorithm will be exponential $\mathcal{O}(n^m)$.

Question 2

Consider a greedy algorithm which allocates skis using the following method: we seek for the couple (ski, customer) having the smallest absolute difference, we allocate this pair of skis to that customer and we start again with remaining skis and customers.

Write the Python code of this algorithm. To help you, go to the practice page of the TD:

https://wdi.centralesupelec.fr/1CC2000/TD5ProgEn

Solution elements :

A proposition of code:

```
############### TODO : complete code ######################
import math
while C:
    ecart_min = math.inf
    for c in C:
        for s in S:
            e = abs(S[s]-C[c])
            if e < ecart_min:
                ecart_min = e
                client = c
                ski = s
    S.pop(ski)
    C.pop(client)
    mapping[client] = ski
```

Question 3

What is the complexity of this algorithm?

Solution elements :

This proposition is in $\mathcal{O}(m^2 \times n)$. Indeed, the search for the best match is in $\mathcal{O}(m \times n)$ (two nested 'for' loops) and it is done exactly m times (a client is served at each iteration of the 'while' loop).

Question 4

Does the greedy algorithm give the optimal solution?

Solution elements :

A counter example with 2 skiers and 2 pairs of skis: $C = \{170, 140\}$ et $S = \{160, 200\}$.

Dynamic Programming

We assume without loss of generality that the n pairs of skis and the m customers are sorted in ascending order of size. We note $Sol[i, j]$ the cost of the optimal solution (i.e. the sum of size differences) for the i first pairs of skis and the j first customers. We want to apply the dynamic programming approach to find the cost of the optimal solution $Sol[n, m]$.

We observe that for two pairs of skis and 2 customers, it is better to allocate the smallest pair to the shortest customer and the longest pair to the tallest one. So if we generalize this result to i pairs of skis and j customers, when the first $j - 1$ customers are already served, the customer j can choose his pair of skis and we will not find a better solution by swapping his pair with a smaller customer (served before him).

This observation allows us to define a resolution algorithm based on dynamic programming.

Question 5

Give the recursion formula to define the value of $Sol[i, j]$ using $Sol[i - 1, j]$ and $Sol[i - 1, j - 1]$.

Solution elements :

There are two cases:

- The skier j doesn't take the pair of skis number i. He has already taken a smaller pair and there would be no benefit in giving this pair i to a skier smaller than j (property stated above). In this case, $Sol[i, j] = Sol[i - 1, j]$.
- The skier j takes the pair of skis i and in this case, $Sol[i, j] = Sol[i - 1, j - 1] + |C[j] - S[i]|$.

The recursion formula is thus:

$$Sol[i, j] = min(Sol[i - 1, j], Sol[i - 1, j - 1] + |C[j] - S[i]|)$$

We have to initialize all $Sol[i_{\geq 0}, 0]$ to 0 (no cost when there are no skiers) and all $Sol[0, j_{\geq 1}]$ to ∞ (infinite cost when we don't have enough skis to give to skiers).

Question 6

Go back to the practice page of the TD to write in Python an algorithm that uses the optimal substructure explained in the previous question. Give the complexity of the algorithm.

Solution elements :

```
############### TODO : complete code ####################

# when no customers
for i in range(n+1):
    Sol[i,0] = 0

# when no skis
for j in range(1,m+1):
    Sol[0,j] = np.Infinity

for i in range(1, n + 1):
    for j in range(1, m + 1):
        Sol[i,j] = min(Sol[i - 1,j - 1] + abs(skis[S[i]] - customers[C[j]]), Sol[i - 1,j])

return Sol, S, C
```

The complexity of sorting pairs of skis and skiers is $\mathcal{O}(n \log n) + \mathcal{O}(m \log m)$ and the time complexity of the dynamic programming algoithm is $\mathcal{O}(n \times m)$

Maximum flow – Minimum cost [advanced]

The advanced students can explore a second technique to compute an optimal solution of the ski rental problem. The maximum flow of minimum cost problem is as follows. Given that :

- a graph oriented $G = (V, E)$;
- two source and sink vertices $sinV$, $tinV$;
- a capacity function $cap : E \rightarrow mathbbN$;
- a cost function $cost : E \rightarrow mathbbN$;

$\Rightarrow$ find a maximum flow f: $E \rightarrow \mathbb{N}$, such that $\sum_{e \in E} cost(e) * f(e)$ is minimum.

Question 7

Propose a model to solve the skis rental problem using a maximum flow minimum cost problem.

Indication: We are looking for a graph with unary capacities to force the flow to be 0 or 1.

Solution elements :

Vertices:

- a source s
- a set C of customers
- a set S of skis
- a target t

Arcs:

- For each $v \in C$, an arc (s, v), with capacity 1, cost 0
- For each $v \in S$, an arc (v, t), with capacity 1, cost 0
- For each $u \in C$, $v \in S$, an arc (u, v), with capacity 1 and cost $abs(u - v)$

We look for a max flow of min cost from s to t.

Question 8

To code a Python algorithm to solve such a problem, go back to the practice page of the TD.

Solution elements :

```
# Flow max -- cost min
def flow_map(skis, customers):

    ############### TODO : complete code ####################

    G = nx.DiGraph()

    # adding nodes
    G.add_node('s')

    for c in customers:
        G.add_node(c)

    for s in skis:
        G.add_node(s)

    G.add_node('t')

    # adding edges
    for c in customers:
        G.add_edge('s', c, weight=0, capacity=1)

    for c in customers:
        for s in skis:
            G.add_edge(c, s, weight=abs(skis[s] - customers[c]), capacity=1)

    for s in skis:
        G.add_edge(s, 't', weight=0, capacity=1)

    # compute the flow
    flow = nx.algorithms.max_flow_min_cost(G, 's', 't')

    # retrieve the mapping
    mapping = {}
    for c in customers:
        for s in skis:
            if flow[c][s] == 1:
                mapping[c] = s
                break

    return mapping
```