CentraleSupélec

Algorithmique et complexité

TD 4/7 – Programmation dynamique

L'objectif de ce TD est de s'exercer à la modélisation de problèmes et à leur résolution par programmation dynamique.

Ce TD est en 3h et comprend des partie de pratique en Python.

Exercice 1 : Politique de placement

Nous disposons d'un euro que nous voulons placer d'une manière optimale pendant 10 périodes. Nous connaissons avec certitude le coefficient de gain c_i pour un placement en début de période i.

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

A chaque période, on a deux options, soit on place la somme totale disponible, soit on garde toute la somme et on n'en place rien. Supposons qu'au début de la période i on dispose d'une somme x. S'il n'y a pas de placement en période i, la somme x sera disponible en début de période $i+1$. Si on place une somme x en début de période i on reçoit la valeur $c_i x$, mais l'argent sera immobilisé pendant cette période et pendant la période $i+1$ et la nouvelle valeur sera disponible seulement en début de période $i+2$ pour être réinvestie ou simplement récupérée.

Il faut trouver la politique de placement total qui rapportera le plus d'argent en début de la période 12 (ce qui signifie qu'il n'y aura plus de placement en début de la période 11).

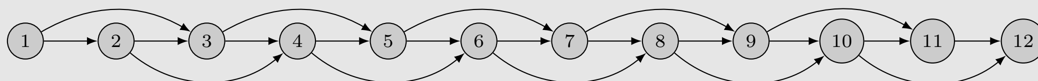
Question 1

En ignorant dans un premier temps les coefficients de gains, modéliser le problème par un graphe non-pondéré (les sommets et les arcs) où chaque politique de placement totale correspondra à un chemin de ce graphe. De quelle nature est-il ? Dessiner le graphe.

Élements de correction :

Le graphe est **orienté**. Il y a 12 sommets qui représentent 12 moments (début de période) où on décide de placer ou non la somme disponible. Le sommet terminal de chaque arc indique le plus proche début de période où l'argent sera disponible. Chaque sommet a donc le degré sortant 2 (sauf pour les deux dernières périodes) et le degré entrant 2 (sauf les deux premières périodes).

On obtient ainsi un **DAG**.



Question 2

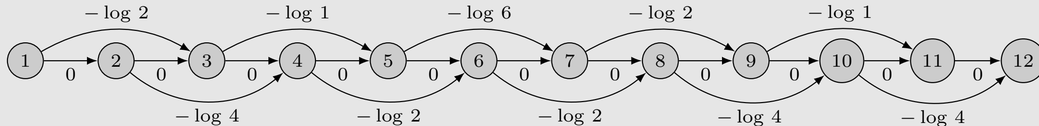
Si on veut se ramener à un problème du plus court chemin, quels poids faut-il affecter aux arcs ? Complétez le graphe.

Élements de correction :

Il s'agit de **maximiser** une fonction de taux c_i **multiplicative** !

1. $c_i \mapsto \log(c_i)$ pour passer à une fonction (de distances) **additive**
2. $\log(c_i) \mapsto -\log(c_i)$ pour passer à une **minimisation**

La valuation d'un arc entre deux sommets non-successifs est $-\log(c_i)$ où c_i est le coefficient de gain prévu par la période du sommet i de départ. 0 entre deux sommets successifs.



Question 3

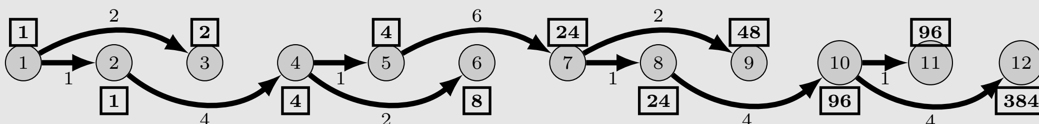
Quel algorithme vu en cours peut-on utiliser pour répondre à l'exercice ? Trouver la solution de l'exemple numérique.

Élements de correction :

En vu des arcs négatifs, on doit opter pour l'algorithme de Ford-Bellman, pour $i \in 1 \ldots |V|$:

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

Voici le résultat final de l'arbre des plus court chemins après exécution de Ford-Bellman :



Un chemin optimal de la période 1 à 12 a comme coeff de gain 384 ! et emprunte les sommets [2, 4, 5, 7, 8, 10]

Il faut noter que s'agissant d'un DAG c-à-d pas de circuits et en particulier pas de circuits absorbants, l'algo de Ford-Bellman aboutit donc à la bonne solution.

Pour aller plus loin : Comme nous disposons naturellement d'un ordre topologique sur les noeuds du graphe, on pourrait faire le rapprochement avec l'exercice du TD2 précédent, et constater que l'adaptation de l'algorithme des plus courts chemins étudié en TD2 fonctionnera pour cet exercice aussi avec une bien meilleure complexité !

Exercice 2 : Location de skis

Problème d'optimisation

Un magasin de ski possède en stock un ensemble de n paires de skis de différentes longueurs. Il reçoit une demande d'un club de vacances pour une location de m paires pour des clients dont la taille est connue avec $m < n$. Le propriétaire du magasin souhaite optimiser le confort des skieurs. Pour cela, il propose de minimiser la somme des écarts (en valeur absolue) entre la taille du client et la taille de ses skis.

Question 1

Formalisez ce problème d'optimisation.

Élements de correction :

Entrées :
- un ensemble de n entiers $s_1, \ldots, s_n$ représentant les longueurs de chaque paire de ski
- un ensemble de m entiers $c_1, \ldots, c_m$ représentant les tailles de chaque client

Sortie : une fonction d'affectation <u>injective</u> $f : [1, m] \to [1, n]$ telle que $\sum_{i \in [1,m]} |c_i - s_{f(i)}|$ est minimale.

On peut constater que la complexité d'un algo naïf et exhaustif sera exponentielle $\mathcal{O}(n^m)$.

Question 2

Considérons un algorithme glouton qui alloue les skis à l'aide de la méthode suivante : on cherche d'abord le couple (ski,client) qui a le plus petit écart *en valeur absolue*, on alloue la paire de ski au client et on recommence avec les skis et les clients restants.

Écrivez en Python le code de cet algorithme. Pour vous aider, rendez-vous sur la page de pratique du TD :

https://wdi.centralesupelec.fr/1CC2000/TD4ProgEn

Élements de correction :

Une proposition de 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

Quelle est la complexité de cet algorithme ?

Élements de correction :

Cette proposition est en $\mathcal{O}(m^2 \times n)$. En effet, la recherche du meilleur appariement est en $\mathcal{O}(m \times n)$ (deux boucles for imbriquées) et le tout est fait exactement m fois (un client est servi à chaque passage dans la boucle while).

Question 4

L'algorithme glouton donne-t-il une solution optimale ?

Élements de correction :

Un contre exemple avec 2 clients et 2 paires de ski : $C = \{170, 140\}$ et $S = \{160, 200\}$.

Programmation dynamique

On suppose sans perte de généralité que les n paires de skis et les m skieurs sont classés par ordre croissant de taille. On note $Sol[i,j]$ le coût de la solution optimale (c'est-à-dire la somme des écarts de taille) pour les i premiers skis et les j premiers clients. On souhaite utiliser une méthode de type ≪ programmation dynamique ≫ pour trouver le coût de la solution optimale $Sol[n,m]$.

On peut constater que, pour 2 paires de skis et 2 skieurs, mieux vaut attribuer la plus petite paire au plus petit skieur et la plus grande paire au plus grand skieur. Ainsi, si on généralise ce résultat à i paires de skis et j skieurs, lorsque les $j-1$ premiers skieurs sont servis, le skieur j peut choisir sa paire de ski et on ne trouvera pas de meilleure solution en permutant sa paire avec un skieur plus petit (servi avant lui).

C'est cette observation qui va nous permettre de définir notre algorithme de résolution par programmation dynamique.

Question 5

Donnez la formule de récurrence qui définit la valeur de $Sol[i,j]$ en fonction des valeurs de $Sol[i-1,j]$ et $Sol[i-1,j-1]$.

Élements de correction :

On a 2 cas :
- Soit le skieur j ne prend pas la paire de ski numéro i. Il a donc déjà pris une paire plus petite et on ne gagnerait pas à donner cette paire i à un skieur plus petit que j (propriété énoncée plus haut). Dans ce cas, $Sol[i,j] = Sol[i-1,j]$.
- Soit le skieur j prend la paire de ski i et dans ce cas, $Sol[i,j] = Sol[i-1,j-1] + |C[j] - S[i]|$.

La formule de récurrence est donc :

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

Il faut initialiser tous les $Sol[i_{\geq 0}, 0]$ à 0 (ça ne coûte rien si on n'a pas de skieurs) et tous les $Sol[0, j_{\geq 1}]$ à ∞ (coût infini quand on n'a pas assez de skis à allouer aux skieurs).

Question 6

Retournez sur la page de pratique du TD pour écrire en Python un algorithme qui utilise la sous-structure optimale explicitée par la question précédente. Donnez la complexité de l'algorithme.

Élements de correction :

```
############### 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
```

La complexité de tri des paires de skis et des skieurs est $\mathcal{O}(n \log n) + \mathcal{O}(m \log m)$ et la complexité en temps de l'algo de programmation dynamique est $\mathcal{O}(n \times m)$

Flot Maximum – Coût minimum [Pour aller plus loin]

Les élèves les plus avancés, pourront explorer une seconde technique pour résoudre le problème de location de skis de manière exacte.

Le problème du flot maximum de coût minimum est le suivant. Étant données :

- un graphe orienté $G = (V, E)$;
- deux sommets source et puits $s \in V$, $t \in V$;
- une fonction de capacité $cap : E \to \mathbb{N}$;
- une fonction de coût $cost : E \to \mathbb{N}$;

$\Rightarrow$ trouver un flot maximum f : $E \to \mathbb{N}$, tel que $\sum_{e \in E} cost(e) * f(e)$ est minimum.

Question 7

Proposer une modélisation permettant de résoudre le problème d'affectation de skis en utilisant un problème de flot maximum de coût minimum.

Indice : On cherche un graphe avec des capacités unaires pour forcer le flot à être 0 ou 1.

Élements de correction :

Sommets :

- une source s
- un ensemble C de clients
- un ensemble S de skis
- un puit t

Arcs :

- Pour chaque $v \in C$, un arc (s, v), avec une capacité 1, un cout 0
- Pour chaque $v \in S$, un arc (v, t), avec une capacité 1, un cout 0
- Pour chaque $u \in C$, $v \in S$, un arc (u, v), avec un capacité 1 et un cout $abs(u - v)$

On cherche un flot max de coût min de s à t.

Question 8

Pour coder en Python un algorithme de résolution d'un tel problème, rendez-vous de nouveau sur la page de pratique du TD.

Élements de correction :

```
# 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
```