Cours / TD : Les algorithmes gloutons

Spé NSI - Lycée du parc

Année 2020-2021

I Le problème du voyageur de commerce

On suppose que l'on dispose d'un tableau `distances` qui donne les distances entre les n villes que doit parcourir un voyageur de commerce lors de sa tournée : `distances[i][j]` donne la distance entre les villes numéro i et j.

Exercice 1

Le module `itertools` dispose d'une fonction permettant de générer toutes les permutations d'une liste sous la forme d'un itérateur :

```
>>> from itertools import permutations
>>> A = [1, 2, 3]
>>> permutations(A)
<itertools.permutations object at 0x1073d7e50>
>>> list(permutations(A))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
>>> for x in permutations(A):
        print(x)


(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

1. Proposer un algorithme simple pour déterminer un itinéraire (commençant et terminant par la ville numérotée 0) qui minimise la distance parcourue.
2. Écrire une fonction `longueur_iti(distances, itineraire)` qui prend en entrée un tableau `distances` et un tuple `itineraire` de longueur $n-1$ décrivant l'itinéraire (exemple : le tuple (2, 1, 4, 3) décrit l'itinéraire $0 \to 2 \to 1 \to 4 \to 3 \to 0$) et renvoie la distance totale parcourue entre les villes en suivant cet itinéraire.
3. En déduire une fonction `dist_opti(distances)` qui prend en entrée une liste `distances` qui donne les distances entre les villes et qui renvoie la distance minimale et un tuple décrivant l'itinéraire qui réalise ce minimum.
4. Déterminer la complexité de `dist_opti` en fonction de n.

5. Cet algorithme vous semble-t-il praticable ?

Pour tester votre code vous pourrez utiliser le tableau de distances suivant :

```
D = [[0, 55, 303, 188, 183], [55, 0, 306, 176, 203],
     [303, 306, 0, 142, 153], [188, 176, 142, 0, 123],
     [183, 203, 153, 123, 0]]
```

L'approche gloutonne

Dans les faits, on ne connait pas d'algorithme qui permette de donner un itinéraire optimal en un temps raisonnable. Dans cette situation, on change de stratégie et, à défaut d'obtenir le meilleur itinéraire, on essaie d'en déterminer un qui soit aussi proche que possible de l'optimal. Les algorithmes gloutons sont alors généralement de bonnes alternatives.

Définition (Algorithme glouton)

> Un algorithme glouton (greedy algorithm en anglais) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global.

Remarque : Dans certaines situations favorables, un algorithme glouton donnera une solution optimale mais **ce n'est pas le cas en général.**

Exercice 2

1. Proposer un algorithme glouton pour le problème du voyageur de commerce.
2. Écrire une fonction `plus_proche(ville, distances, visitees)` où les paramètres sont respectivement : un entier qui donne le numéro d'une ville, le tableau des distances entre les villes et un tableau de booléens qui indique pour chaque ville si elle a été visitée et renvoie le numéro de la ville la plus proche de `ville` parmi celle qui n'ont pas encore été visitées.
3. Écrire une fonction `pvc_glouton(distances)` qui prend en entrée une liste `distances` donnant les distances entre les villes et qui renvoie les mêmes éléments que pour l'exercice 1 en appliquant la démarche gloutonne. Il est conseillé d'écrire deux fonctions.
4. Déterminer la complexité de la fonction `pvc_glouton`.

II Le problème du rendu de monnaie

Dans le système monétaire européen, on dispose de pièces et de billets dont les valeurs (pour celles qui sont entières) sont celles de la liste :

```
euros = [1, 2, 5, 10, 20, 50, 100, 200]
```

Ce système est dit canonique[1] car il permet à une approche gloutonne de donner un algorithme optimal pour le rendu de monnaie.

Exercice 3

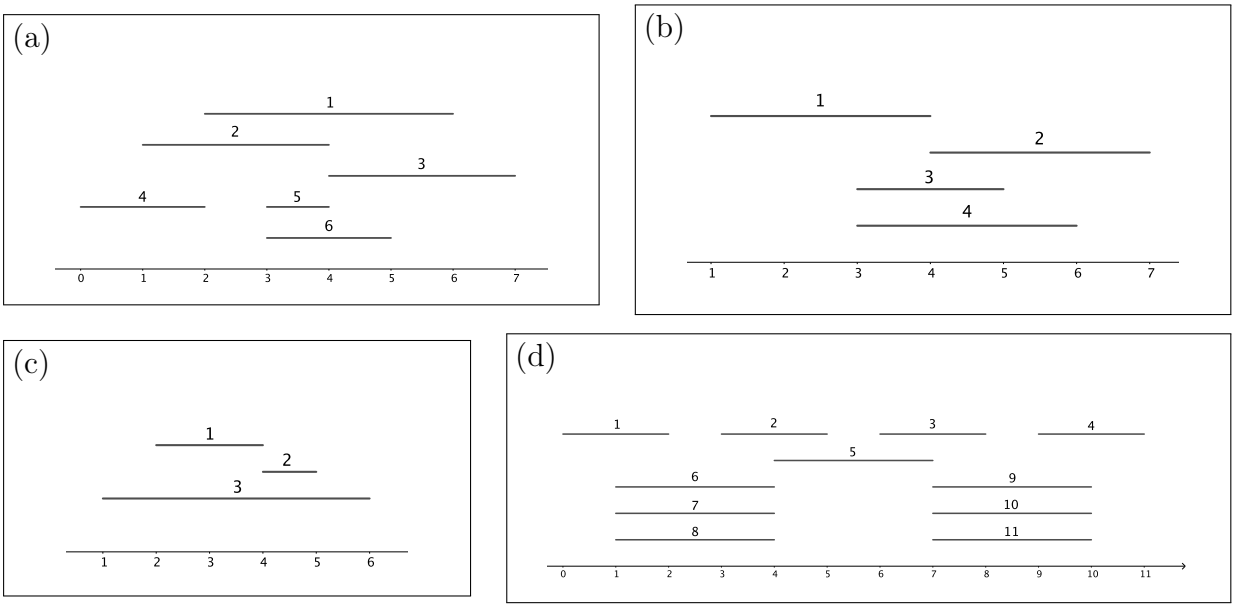
1. Expliciter l'algorithme glouton pour le rendu de monnaie.
2. Écrire une fonction `nb_rendu(sys, n)` où les paramètres sont respectivement : une liste `sys` représentant un système monétaire et un entier `n`. Cette fonction devra renvoyer le nombre minimal de pièces et billets nécessaires pour rendre n euros.
3. Modifier la fonction précédente pour qu'elle fournisse la combinaison de pièces et billets correspondante sous la forme d'une liste de couples (valeur, effectif).
4. Montrer que pour le système monétaire défini par la liste `[1, 4, 6]`, la stratégie gloutonne ne donne pas toujours la réponse optimale.

1. Déterminer si un système est canonique ou non n'est pas chose facile. Les curieux pourront consulter Wikipédia.

III Plusieurs stratégies gloutonnes pour un même problème

On considère un ensemble de n activités, chacune possédant une date de début et une date de fin : $[d_i, f_i[$ où d_i et f_i désignent respectivement l'heure de début et l'heure de fin. Pour des raisons logistiques (par exemple accès à une ressource « rare »), on ne peut pas programmer des activités se déroulant en même temps. La question est la suivante : quel est le nombre maximal d'activités que l'on va pouvoir planifier ?

On représente ci-dessous quelques exemples d'ensembles d'intervalles constituant différentes entrées possibles pour le problème.



On envisage plusieurs stratégies gloutonnes pour construire le planning.

A. **Tri par durée**

Une première méthode consiste à prendre les activités par ordre croissant de leur durée. On peut en effet raisonnablement penser que planifier d'abord les activités les plus courtes laissera plus de temps pour les autres.

1. Déterminer quel planning produit cette stratégie gloutonne sur le problème (a).
2. Cette stratégie donne-t-elle toujours un résultat optimal ?

B. **Tri par date de début**

Une autre approche est de considérer les activités par ordre croissant de leur date de début. L'idée étant de commencer au plus vite une nouvelle activité. Ici, le choix glouton consiste à prendre à chaque étape l'activité commençant le plus tôt, à condition qu'elle soit compatible avec celles déjà planifiées.

1. Déterminer quel planning produit cette stratégie gloutonne sur le problème (a).
2. Cette deuxième stratégie donne-t-elle toujours un résultat optimal ?

C. **Tri par nombre d'incompatibilités**

Une troisième possibilité revient à prendre les activités par ordre croissant de leurs incompatibilités. Procéder ainsi permet de privilégier les activités qui « gènent » le moins, et donc avoir moins de contraintes pour planifier les autres.

1. Déterminer quel planning produit cette stratégie gloutonne sur le problème (a).

2. Cette nouvelle stratégie donne-t-elle toujours un résultat optimal ?

D. **Tri par date de fin**

La dernière stratégie que nous envisageront consiste à laisser le plus de temps disponible après la planification d'une activité.

1. Déterminer quel planning produit cette stratégie gloutonne sur les problème (a), (b), (c) et (d).
2. Cette stratégie donne-t-elle toujours un résultat optimal sur les quatre exemples ?
3. On admet que cette dernière stratégie est optimale. Écrire une fonction `planing(plages)` qui prend en entrée une liste de couples `(d, f)` représentant les plages et renvoie sur le même format un planning optimal. On pourra décomposer en plusieurs fonctions.