Chapitre 9 :
Algorithmes gloutons

I Généralités

1.1 Optimisation

On considère un problème à résoudre. Ce problème peut avoir plusieurs solutions. Le principe de l'optimisation est de chercher la meilleure solution, selon un critère donné.

Pour étudier un problème d'optimisation, il faut :

- déterminer une fonction qui permettra d'évaluer la qualité de chaque solution afin de déterminer la meilleure,
- avoir un grand nombre de solutions, sans cela, il serait simple de trouver la meilleure,
- être assuré de l'existence d'une solution optimale.

➪ **Exemple 1 : : Rendu de monnaie**
Le problème du rendu de monnaie consiste à rendre la monnaie en utilisant le moins possible de pièces (ou billets).

1.2 Algorithme glouton

Le principe de l'algorithme glouton est de faire un choix localement optimal dans le but d'obtenir une solution globalement optimale.

➪ **Exemple 2 : : Rendu de monnaie**
On considère un système monétaire composé des pièces de valeurs Val= $[v_0, v_1, \dots, v_n]$ avec : $v_0 > v_1 > \cdots > v_n$ et on souhaite rendre la somme x en utilisant un nombre minimal de pièces.
Le principe d'un algorithme glouton consiste à donner d'abord la pièce ayant la plus forte valeur, puis, à nouveau la pièce ayant la plus forte valeur, etc.
Ainsi, à chaque étape, on donne le moins de pièces possibles.
A chaque étape, on "avale" tout ce qu'on peut d'où le nom d'algorithme glouton.
Par exemple, si $Val = [500, 200, 100, 50, 20, 10, 5, 2, 1]$ et $x = 144$:

- la plus grande valeur possible est 100, il reste 44 à rendre,
- la plus grande valeur possible est 20, il reste 24 à rendre,
- la plus grande valeur possible est 20, il reste 4 à rendre,
- la plus grande valeur possible est 2, il reste 2 à rendre,
- la plus grande valeur possible est 2, il reste 0 à rendre.

Ainsi, on doit rendre 1 de valeur 100, 2 de valeur 20 et 2 de valeur 2, soit 5 pièces au total.

```
def rendu(x,Val) :
    Sol=[0]*len(Val)
    i=0
    while x>0 :
        if x-Val[i]>=0 :
            x-=Val[i]
            Sol[i]+=1
        else :
            i+=1
    print('Il faut rendre :')
    for i in range(len(Val)) :
        print(Sol[i],'de valeur',Val[i])
```

1.3 Résultat obtenu

Le principe d'un algorithme glouton est d'espérer que le choix optimisé localement donne une solution optimisée globalement mais ce n'est pas toujours le cas.
Si la solution obtenue est bien la meilleure solution, on parle d'**algorithme glouton exact**. Sinon, on parle d'**heuristiques gloutonnes**.

➪ **Exemple 3 : : Rendu de monnaie**
L'algorithme vu précédemment n'est pas toujours exact. Pour le système monétaire européen ($Val = [500, 200, 100, 50, 20, 10, 5, 2, 1]$), on peut montrer qu'il est exact. Mais ce n'est pas toujours le cas :

```
>>> Val2=[30,24,12,6,3,1]
>>> rendu(49,Val2)
Il faut rendre :
1 de valeur 30
0 de valeur 24
1 de valeur 12
1 de valeur 6
0 de valeur 3
1 de valeur 1
```

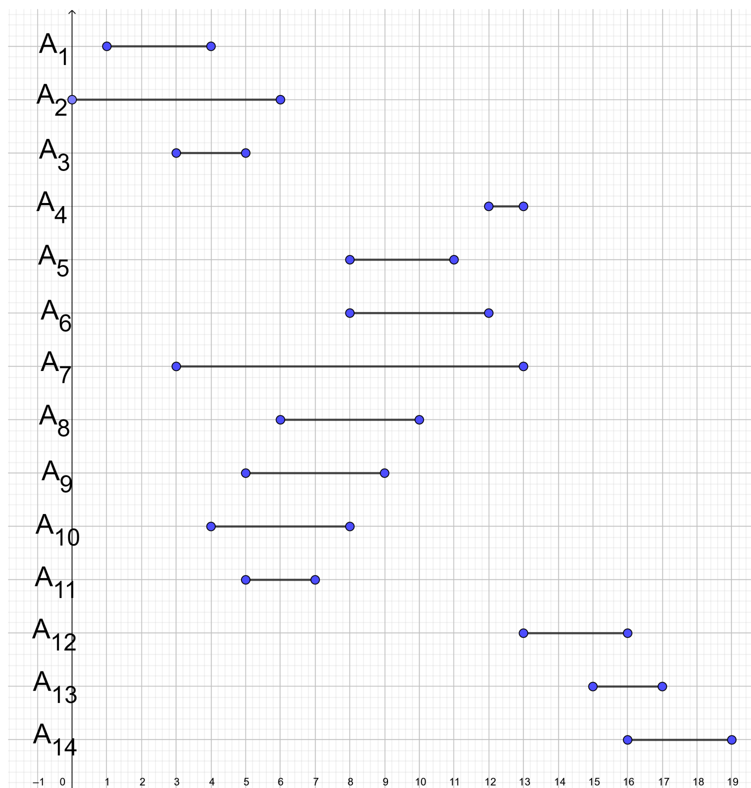
La solution obtenue consiste à rendre 4 pièces alors que 3 suffisent : deux de 24 et une de 1. On dit que le système monétaire considéré n'est pas canonique.

II Un exemple : le problème du choix d'activités

2.1 Problématique

On considère des activités, chacune possédant une heure de début et une heure de fin. On veut choisir le nombre maximal d'activités, sans chevauchement.

➪ **Exemple 4 :** On considère le planning suivant :



On peut ainsi choisir les activités A_2, puis A_8 et enfin A_{12}. C'est un choix possible mais non optimal, car le choix $A_3 - A_9 - A_4 - A_{12}$ permet de faire une activité supplémentaire.

2.2 Modélisation

On considère des activités $A_1, \ldots, A_n$. Pour $i \in [\![1, n]\!]$, l'activité A_i est représentée par le couple (d_i, f_i) où d_i est l'heure de début de A_i et f_i est son heure de fin, avec $d_i < f_i$. L'ensemble des activités est représenté par la liste :

$$S = [(d_1, f_1), \ldots, (d_n, f_n)].$$

➪ **Exemple 5 :**

$$S = [(1,4),(0,6),(3,5),(12,13),(8,11),(8,12),(3,13),(6,10),(5,9),(4,8),(5,7),(13,16),(15,17),(16,19)].$$

On dit que deux activités A_i et A_j sont compatibles si elles ne se chevauchent pas, c'est-à-dire si $d_i \geq f_j$ ou $d_j \geq f_i$.

➪ **Exemple 6 :** Les activités $A_3 - A_9 - A_4 - A_{12}$ sont compatibles et correspondent à la liste :

$$[(3,5),(5,9),(12,13),(13,16)].$$

On va proposer plusieurs solutions pour résoudre le problème.

2.3 Tri par durée

En choisissant d'abord les activités ayant une durée plus courte, on pense pouvoir placer le maximum d'activités.

➪ **Exemple 7 :**

activité	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
début	1	0	3	12	8	8	3	6	5	4	5	13	15	16
fin	4	6	5	13	11	12	13	10	9	8	7	16	17	19
durée	3	6	2	1	3	4	10	4	4	4	2	3	2	3

- L'activité la plus courte est A_4, on la choisit donc : $[(12,13)]$,
- l'activité la plus courte compatible est A_3, on la choisit donc : $[(12,13),(3,5)]$,
- l'activité la plus courte compatible est A_{11}, on la choisit donc : $[(12,13),(3,5),(5,7)]$,
- l'activité la plus courte compatible est A_{13}, on la choisit donc : $[(12,13),(3,5),(5,7),(15,17)]$,
- l'activité la plus courte compatible est A_5, on la choisit donc : $[(12,13),(3,5),(5,7),(15,17),(8,11)]$,
- il n'y a plus d'activité compatible.

Cette méthode nous a permis d'obtenir 5 activités.

2.4 Tri par nombre d'incompatibilités

En choisissant d'abord les activités ayant le moins d'incompatibilités, on aura davantage de possibilités pour placer les activités restantes, on pense pouvoir placer le maximum d'activités.

➪ **Exemple 8 :**

activité	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
début	1	0	3	12	8	8	3	6	5	4	5	13	15	16
fin	4	6	5	13	11	12	13	10	9	8	7	16	17	19
nombre d'incompatibilités	3	6	4	1	4	4	10	6	7	6	5	1	2	1

- L'activité ayant le moins d'incompatibilités est A_4, on la choisit donc : $[(12,13)]$,
- l'activité ayant le moins d'incompatibilités et compatible est A_{12}, on la choisit donc : $[(12,13),(13,16)]$,
- l'activité ayant le moins d'incompatibilités et compatible est A_{14}, on la choisit donc : $[(12,13),(13,16),(16,19)]$,
- l'activité ayant le moins d'incompatibilités et compatible est A_1, on la choisit donc : $[(12,13),(13,16),(16,19),(1,4)]$,
- l'activité ayant le moins d'incompatibilités et compatible est A_5, on la choisit donc : $[(12,13),(13,16),(16,19),(1,4),(8,11)]$,
- l'activité ayant le moins d'incompatibilités et compatible est A_{11}, on la choisit donc : $[(12,13),(13,16),(16,19),(1,4),(8,11),(5,7)]$,
- il n'y a plus d'activité compatible.

Cette méthode nous a permis d'obtenir 6 activités.

2.5 Tri par date de début

En choisissant d'abord la première activité disponible, on perdra moins de temps, on pense pouvoir placer le maximum d'activités.

➪ **Exemple 9 :**

activité	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
début	1	0	3	12	8	8	3	6	5	4	5	13	15	16
fin	4	6	5	13	11	12	13	10	9	8	7	16	17	19

- L'activité commençant en premier est A_2, on la choisit donc : $[(0,6)]$,
- l'activité compatible commençant en premier est A_8, on la choisit donc : $[(0,6),(6,10)]$,
- l'activité compatible commençant en premier est A_4, on la choisit donc : $[(0,6),(6,10),(12,13)]$,
- l'activité compatible commençant en premier est A_{12}, on la choisit donc : $[(0,6),(6,10),(12,13),(13,16)]$,
- l'activité compatible commençant en premier est A_{14}, on la choisit donc : $[(0,6),(6,10),(12,13),(13,16),(16,19)]$,
- il n'y a plus d'activité compatible.

Cette méthode nous a permis d'obtenir 5 activités.

2.6 Tri par date de fin

En choisissant d'abord l'activité qui se termine le plus tôt, on aura davantage de temps pour placer d'autres activités, on pense pouvoir placer le maximum d'activités.

➪ **Exemple 10 :**

activité	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
début	1	0	3	12	8	8	3	6	5	4	5	13	15	16
fin	4	6	5	13	11	12	13	10	9	8	7	16	17	19

- L'activité se terminant en premier est A_1, on la choisit donc : $[(1,4)]$,
- l'activité compatible se terminant en premier est A_{11}, on la choisit donc : $[(1,4),(5,7)]$,
- l'activité compatible se terminant en premier est A_5, on la choisit donc : $[(1,4),(5,7),(8,11)]$,
- l'activité compatible se terminant en premier est A_4, on la choisit donc : $[(1,4),(5,7),(8,11),(12,13)]$,
- l'activité compatible se terminant en premier est A_{12}, on la choisit donc : $[(1,4),(5,7),(8,11),(12,13),(13,16)]$,
- l'activité compatible se terminant en premier est A_{14}, on la choisit donc : $[(1,4),(5,7),(8,11),(12,13),(13,16),(16,19)]$,
- il n'y a plus d'activité compatible.

Cette méthode nous a permis d'obtenir 6 activités.

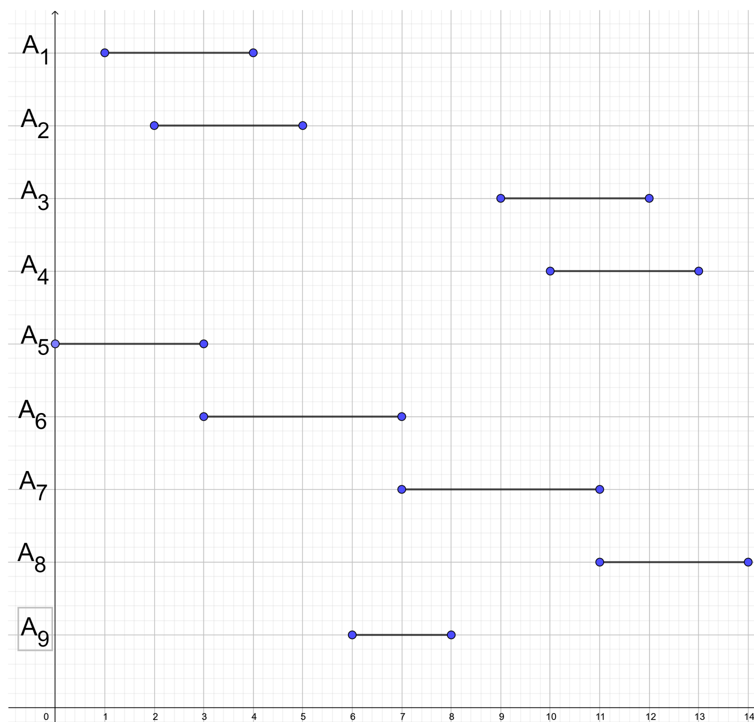
2.7 Conclusion

Le tri par durée et le tri par date de début permettent de placer 5 activités alors que le tri par nombre d'incompatibilités et le tri par date de fin permettent de placer 6 activités.

Notre exemple montre donc que le tri par durée et le tri par date de début ne donnent pas la meilleure solution, il s'agit d'heuristiques gloutonnes.

Le tri par nombre d'incompatibilités est également une heuristique gloutonne, pour le voir, on doit considérer un autre exemple.

➪ **Exemple 11 :** On considère le planning suivant :



Le tri par incompatibilité donne $[(0,3),(11,14),(6,8)]$ soit 3 activités alors que le tri par date de fin donne $[(0,3),(3,7),(7,11),(11,14)]$ soit 4 activités.

Il est possible de prouver que le tri par date de fin est un algorithme glouton exact.