Comparaison des algorithmes brutal_force et path_pattern

Comparaison de l'efficacité et des performances des algorithmes (notation Big-O)

Pour chaque étape de l'algorithme, nous allons étudier la complexité du code, afin d'en déduire leur complexité globale.

1. brutal_force

- Récupérer les données issues d'un fichier csv : **O(N)**
- Créer une liste contenant chaque action sous forme d'objet Share. L'objet Share a une complexité O(1). Donc la complexité de la liste est de l'ordre de N*O(1) soit: **O(N)**
- Générer l'ensemble des combinaisons possibles d'actions de chaque taille possible est de : **O(2^N)**
- Depuis la liste des combinaisons, calculer le coût et les bénéfices de chaque portefeuille d'action est donc aussi de : **O(2^N)**
- La complexité du trie de la liste des portefeuille d'action par bénéfice, en utilisant la méthode "sort" de "list", est de : **O(N.logN)**
- Récupérer le premier portefeuille d'action qui répond au critère du coût maximum, correspond à une complexité de : **O(N)**
- Afficher le résultat : **O(1)**

La complexité globale de l'algorithme "brutal_force" est donc:

O(N) + O(N) + O(2^N) + O(2^N) + O(N.logN) + O(N) + O(1)

Sachant que les ordres de complexité sont :

O(logN) < O(N) < O(N.logN) < O(N^2) < O(2^N) < O(N!)

On peut simplifier la complexité globale de l'algorithme "brutal_force", qui est donc de l'ordre de:

O(2^N)

2. path_pattern

- Récupérer les données issues d'un fichier csv : **O(N)**
- Créer une liste contenant chaque action sous forme d'objet Share. L'objet Share a une complexité O(1). Donc la complexité de la liste est de l'ordre de N*O(1) soit: **O(N)**
- Créer deux tableaux : **O(N)**
- Dérouler l'algorithme "pathfinding" correspond à:
  - Les boucles "for" imbriquées ont une complexité de l'ordre de :

    O(K*N)

    (avec K qui correspond à la constante représentant le coût maximum du portefeuille)
  - Que l'on multiplie par la somme des traitements suivant :
    - initialiser des variables : O(1)
    - traitements de condition (on prend le pire cas, il s'agit d'affectation de variable) : O(1)
  - La complexité de l'algorithme "pathfinding" est donc de :

    O(N^2)
- Récupérer la liste des actions :
  - Dans une boucle for, la complexité est de l'ordre de :

    O(N)
  - Que l'on multiplie par la somme des traitements suivant :
    - initialiser une variable : O(1)
    - traitements d'une condition avec affectation de variable :

      O(1)
  - La complexité globale de la récupération de la liste des actions est donc de : **O(N)**
- Afficher le résultat : **O(1)**

La complexité globale de l'algorithme "path_pattern" est donc:

O(N) + O(N) + O(N) + O(K*N) + O(N) + O(1)

Sachant que les ordres de complexité sont :

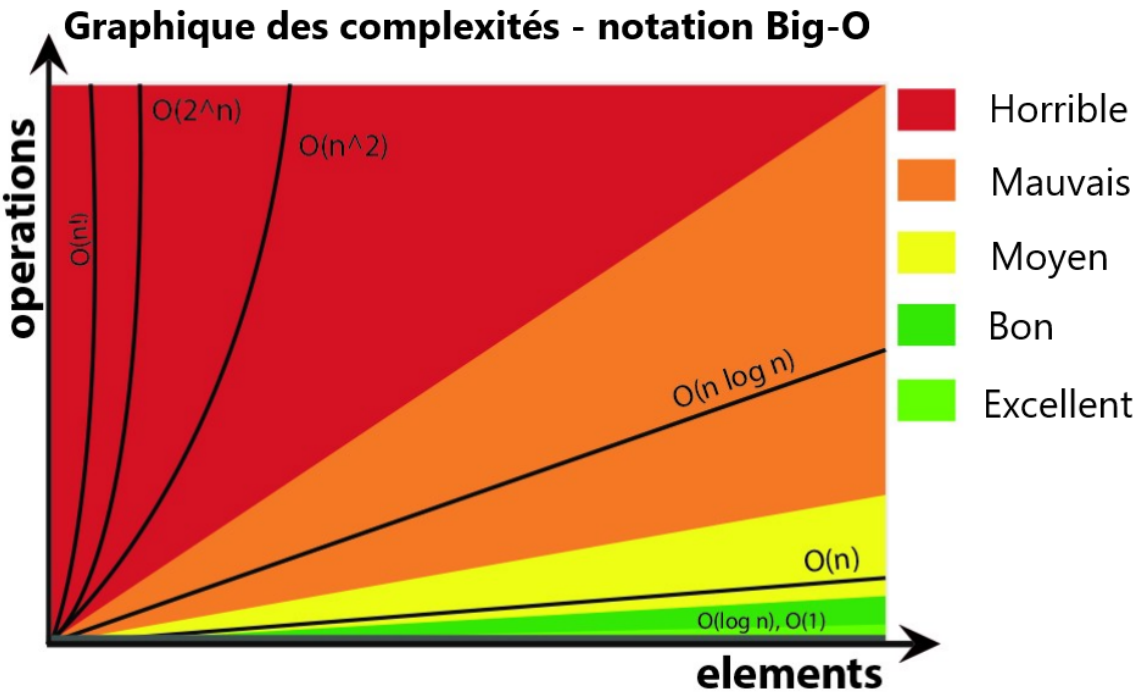
O(logN) < O(N) < O(N.logN) < O(N^2) < O(2^N) < O(N!)

On peut simplifier la complexité globale de l'algorithme "path_pattern", qui est donc de l'ordre de:

O(K.N)

3. Synthèse

En se référant au tableau ci-dessous, on constate que l'algorithme "path_pattern" est bien plus efficient que l'algorithme "force_brute".



Comparaison de la complexité temporelle

1. brutal_force

Pour cette mesure, le programme a été exécuté 10 fois afin de mesurer le temps moyen. Ce temps a ensuite été utilisé pour calculer le temps que prendrait l'algorithme sur un set de 1000 actions.

a. Liste de 20 actions : Le temps de traitement est de 13,26s
b. dataset1_Python+P7 : Le temps de traitement est de 1,35*10^296 an
c. dataset2_Python+P7 : Le temps de traitement est de 1,35*10^296 an

L'estimation a été réalisée en se basant sur l'ordre de complexité de O(2^N) de cet algorithme.
Si on utilise une méthode de calcul avec le nombre de combinaisons pour 20 actions, ainsi que le nombre de combinaisons pour 1000 actions, et qu'on l'on considère le temps de traitement proportionnel, on trouve un temps de traitement de l'ordre de 10^288 an. Ce qui conforte notre première analyse.

2. path_pattern

Pour cette mesure, le programme a été exécuté 100 fois afin de mesurer les temps moyen suivant:

a. Liste de 20 actions : Le temps de traitement est de 0,017s
b. dataset1_Python+P7 : Le temps de traitement est de 0,160s
c. dataset2_Python+P7 : Le temps de traitement est de 0,130s
d. dataset1000 : Le temps de traitement est de 1,631s

Pour comprendre l'écart de temps entre le traitement du dataset1 et celui du dataset2, la librairie pandas a été utilisée pour analyser les données.
En les affichant, on constate que de nombreuses données ont un coût inférieur ou égale à 0, dans le dataset2. On en dénombre 459.
Comme ces données ont été écartées par l'algorithme, qui les considère comme erronées, le nombre de traitement effectué par l'algorithme est inférieur, ce qui explique cet écart de 30 ms.

Afin de s'assurer de la linéarité de l'algorithme, un set de 10000 actions a été créé. Puis le temps moyen de traitement a été mesuré sur 100 itérations.
Et l'on peut constater que pour 10 fois plus d'action, le temps de traitement est 10 fois plus grand.

3. Synthèse

Cette comparaison de la complexité temporelle confirme l'efficacité de l'algorithme "path_pattern". Et elle confirme également l'analyse de la complexité du code.

Analyse de la mémoire

Pour mesurer la complexité spatiale, le module “Memory Profiler” a été utilisé et fourni les résultats suivant :

1. brutal_force

a. Liste de 20 actions : 6 431 Ko
b. dataset1_Python+P7 : -
c. dataset2_Python+P7 : -

Cet algorithme utilise 6 431 Ko.

2. path_pattern

a. Liste de 20 actions : 610 Ko
b. dataset1_Python+P7 : 1 176 Ko
c. dataset2_Python+P7 : 805 Ko
d. dataset10000 : 1 815 Ko

3. Synthèse

Cette comparaison de la complexité spatiale confirme également les ordres de complexité.
De plus, elle nous permet de constater, pour l’algorithme “path_pattern” que la taille mémoire n’est pas proportionnelle à la quantité de données d’entrée mais bien à la taille des données stockées dans la matrice.