Rapport de projet

2019-04-15

Licence informatique

Comparaison de tri en complexité linéaire

Par :
MAJDOUL Kaoutar
CHAMROUK Laila
MESSAOUDI Soukaina
LAAFOU Ayoub

Sous la direction de :
M. GIROUDEAU

Remerciement

Introduction

Dans le cadre du module du projet se déroulant tout le long du second semectre de L2, nous avons été chargé de proposer et d'analyser les performances de plusieurs algorithmes de tri de données dont la complexité en temps est linéaire.

Ce travail a été effectué par notre groupe composé de quatres personnes : MAJDOUL Kaoutar, CHAMROUK Laila, MESSAOUI Soukaîna et LAAFOU Ayoub. Ceci sous la direction du Mr. GIROUDEAU pour une durée de 15 semaine soit de janvier à mi-mai 2019.

Présentation du projet

L'objectif principale d'un algorithme de tri est de résoudre un problème de façon correct en un temps fini. Il décrit un ensemble de règles qui s'applique à des données. Puis lorqu'il est executé sur ces données il termine en un temps fini et produit un résultat représentant une solution du problème donné.

Selon la méthode de tri la complexité en temps et en mémoire peut être exponentielle, il est donc intéressant de visualiser la durée d'exécution de données selon un nombre d'éléments afin de choisir le tri le plus optimal à l'utilisation voulue.

De ce fait, nous nous intéresserons à plusieurs algorithme de tri dont la complexité modulo une utilisation maximale de la mémoire est linéaire que nous programmerons et exécuterons de sorte à visualiser leur performance en temps et en mémoire.

Enfin, nous avons choisi le langage *Python* pour toute la partie programmation de ce projet. En effet, ce langage étant facile à apprendre, interactif et orienté objet il nous a permis de traiter les données de façon simple et lisible afin de nous concentré sur l'algorithmique de chaque fonction plutôt que la syntaxe. De plus, les nombreuses bibliothèques incluses ainsi l'abondance de documentation présente sur internet nous ont conforté dans le choix de ce langage de programmation.

Cahier des charges

[PLAN]

Table des matières

1 Tri par dénombrement

1.1 Description

Afin de décrire l'algorithme de tri par dénombrement de manière formelle nous allons dérouler les différentes étapes avant d'obtenir un tableau de données trié. Tout d'abord, nous créons un tableau de n valeurs aléatoires comprises entre deux valeurs choisies. Suite à cela nous parcourons le tableau et nous comptons le nombre de fois que chaque élément apparaît dans un tableau des effectifs E dont la taille est la valeur maximum du tableau de départ et avec *E[i]* le nombre de fois où le nombre i apparaît dans le tableau. Enfin nous le parcourrons dans le sens croissant et plaçons dans le tableau trié *E[i]* fois l'élément i avec i allant de la plus petite valeur à la plus grande.

D'autre part, de manière informelle, le tri dénombrement est un algorithme de tri reposant sur le principe de construction d'un histogramme des données puis le balayage de celui-ci de façon croissante afin d'obtenir les données triées. De ce fait, ici, plusieurs éléments identiques seront représentés par un unique élément quantifié. Ce tri convient exclusivement aux données constituées de nombres entiers compris entre une valeur minimale et une valeur maximale définies. Dans un souci d'efficacité, celles-ci doivent être assez proches l'une de l'autre et le nombre d'éléments doit être relativement grand afin d'obtenir une utilisation optimale de cette algorithme.

Ainsi, l'algorithme donne le psuedo-code suivant :

```
Créer un tableau T de n valeurs aléatoires bornées
Chercher l'élément maximum du tableau
Créer un tableau E de taille max + 1, initialisé à 0

Pour chaque élément du tableau de départ
    Incrémenter E[T[i]]

Pour chaque élément du tableau E
    Recopier E[i] fois le nombre i dans le tableau trié
Lire le tableau E dans l'ordre croissant
```

1.2 Exemple

Afin d'illustrer cet algorithme nous allons dérouler un exemple de son utilisation. Soit le tableau d'entier : 9, 7, 1, 4, 8, 1, 1 que l'on souhaite trier dans l'ordre croissant en utilisant le tri par dénombrement. Dans un premier temps nous allons créer notre tableau des effectifs H puis nous allons le parcourir et recopier les valeurs dans le tableau trié :

i	H[i]	Action	Tableau trié
0	0	on ne fait rien	
1	3	on ajoute trois fois 1	1 1 1
2	0	on ne fait rien	1 1 1
4	1	on ajoute une fois 4	1 1 1 4
4	0	on ne fait rien	1 1 1 4
5	0	on ne fait rien	1 1 1 4
7	1	on ajoute une fois 7	1 1 1 4 7
7	0	on ne fait rien	1 1 1 4 7
9	2	on ajoute deux fois 9	1 1 1 4 7 9 9

On a atteint le maximum de notre tableau des effectifs H, notre tableau est donc trié : 1, 1, 1, 4, 7, 9, 9.

1.3 Complexité

Notons n la taille de la liste considérée. Le calcul du minimum et du maximum coûte un parcourt de la liste, soit *(n)* et l'allocation de l'histogramme *(m)* où m désigne l'étendue de la liste L. Notons H l'histogramme, la construction de celui-ci demande un nouveau parcourt de la liste et à donc un coût de *(n)*. Pour la reconstruction de la liste dans la boucle on vide chacun des *H[i]* et on sait que :

$$\sum_{i=1}^{maxL-minL+1} H[i] = n$$

donc la reconstruction de la liste coûte elle aussi *(n)*.

Les seules opérations que l'on effectue dans notre fonction se font en temps linéaire. L'initialisation du tableau des effectifs se fait en *O(n)* avec n la taille de la liste en entrée et la copie des éléments dans notre tableau trié en *O(m)* avec m correspondant à max.

La complexité de l'algorithme de tri par dénombrement T pour une liste L de taille n et d'étendue m est :

$$T(n, m) = (n + m)$$

Cet algorithme n'est donc linéaire que si la taille m de l'histogramme reste raisonnable O(n) au regard de la taille de la liste, autrement dit quand la dispersion est faible et dans ce cas *T(n,m)=(n)*. C'est la dispersion qui conditionnera l'utilisation ou non de ce tri.

La complexité finale de notre algorithme est donc *O(n+m)*, soit une complexité en temps linéaire.

1.4 Avantages et inconvénients

Points positifs

- Très efficace si les n nombres à trier son petits (¡= n).

- Ne fait aucune comparaison.

- Peut optimiser d'autre algorithme de tri, tel que le tri par base.

- Stable puisqu'il préserve l'ordre initial dans le tableau height

Points négatifs

- S'exécute uniquement sur des nombres entiers.

- La complexité en mémoire est mauvaise car l'algorithme peut prendre très vite beaucoup de place. le tableau des effectifs E a pour taille
la valeur maximale du tableau, or si cette valeur est très grande,
le tableau H prendra énormément de place en mémoire.

- Si les valeurs du tableau sont éloignées entre elles, alors beaucoup d'espace mémoire restera inutilisé.

Cet algorithme est donc très efficace mais il faut savoir faire un choix entre rapidité et stockage, en plus de ne pouvoir l'utiliser que sur des nombre entiers.

2 Tri lexicographique

2.1 Description

L'ordre lexicographique est la manière d'organiser les mots en se basant sur l'ordre alphabétique. Il rappelle la façon de rechercher un mot dans un dictionnaire : nous commençons par chercher la première lettre du mot puis la seconde, et ainsi de suite. Le tri lexicographique prend donc en paramètre une liste de mots à trier. Chacun des mots sont créés au hasard et la liste est au départ non triée. Le nombre de mots dans la liste est choisie et donnée en paramètre. Plus il y a de mots, plus le tri mettra du temps. L'algorithme parcourt la liste en prenant le premier mot et en le comparant au reste des mots, lettre par lettre. Le mot est ensuite placé dans la liste suivant l'ordre alphabétique.

Nous avons ainsi le pseudo-code suivant :

```
Créer une nouvelle liste de mots, non triée

Pour chaque mot de la liste
        Comparaison du mot avec le reste de la liste et replace le mot dans la liste

Retourne la liste triée
```

2.2 Exemple

En vue de mieux comprendre l'algorithme, nous allons l'utiliser sur une liste de mots L_i : (xyz, abc, Ade). La liste de mots sera modifiée pour donner à la fin la liste triée.

i	Action	Liste
0	on va comparer le premier mot au suivant	(xyz, abc, Ade)
1	on va comparer ce même mot au reste de la liste	(abc, xyz, Ade)
2	on va comparer le nouveau premier élément au reste de la liste	(abc, Ade, xyz)
3	-	(Ade, abc, xyz)

2.3 Complexité

Soit n la taille de la liste à trier. La création de la liste est une opération élémentaire. Pour le triage de la liste, nous utilisons la méthode sort() du langage python. Sa complexité est de l'ordre de $O(nlogn)$. La complexité de l'algorithme est donc de l'ordre de $O(nlogn)$ dans un cas moyen et dans le pire des cas.

2.4 Avantages et inconvénients

En ce qui concerne les point positifs, cet algorithme reste constant dans sa complexité en temps, même avec une liste assez grande. De plus, la liste est triée dans cette même liste, la mémoire requise pour le tri n'excède donc pas la taille de la liste.

Cependant, le tri ne se fait que sur des mots. On peut également rajouter que la complexité en temps n'est plus linéaire mais linéarithmique.

3 Tri par répartition

3.1 Description

Le tri par dénombrement utilise la même procédure de fonctionnement qu'un facteur dans un centre de tri postal. Le facteur dispose d'une armoire de tri constituée d'autant de casiers qu'il y a d'adresses dans sa tournée. Avant de commencer sa tournée, le facteur récupère les enveloppes à distribuer, les éléments d'une liste L par exemple, et range chaque enveloppe L[i] dans le casier correspondant à l'adresse sur l'enveloppe. Une fois toutes les enveloppes réparties dans les casiers, le facteur n'a plus qu'à les rassembler dans l'ordre des casiers (les concaténer) pour commencer sa tournée. Par ailleurs, nous prenons en compte que le facteur aura une tournée optimale pour minimiser la distance à parcourir.

Nous avons ainsi le pseudo-code suivant :

...

3.2 Exemple

...

3.3 Complexité

Les seules opérations qu l'on effectue dans notre fonction se font en temps linéaire. L'initialisation du tableau des effectifs se fait en O(N) (avec N la taille du tableau en entrée), et la copie des éléments dans notre tableau trié en O(M) (avec M correspondant à Max). La complexité de cet algorithme est donc simple à calculer, et elle se fait ainsi en O(N+M), une complexité linéaire.

3.4 Avantages et inconvénients

Le tri par répartition reste un algorithme très efficace. Cependant, le tri ne se fait que sur des nombres entiers, sa complexité en mémoire est mauvaise car l'algorithme peut prendre très vite beaucoup de place.

4 Tri par base

4.1 Description

Pour trier des entiers de c chiffres, l'algorithme utilise un tri stable, ici, le tri par dénombrement pour trier les nombres par rapport à chacun de ces chiffres en commençant par le chiffre des unités. Appelé radix sort en anglais, cette algorithmes de tri utilise la décomposition des objets à trier dans une base donnée. Son fonctionnemenet consiste à trier les éléments en commençant par leur chiffre le moins significatif, autrement dit en les considérant de la droite vers la gauche : les éléments sont d'abord triés suivant leur dernier chiffre, puis suivant l'avant-dernier et ainsi de suite.

Pour trier des entiers contenant c chiffres, l'algorithme utilise un tri par dénombrement pour trier les éléments selon chacun de ces chiffres en commençant par le chiffre des unités tel que le décrit l'algorithme en pseudo-code suivant :

```
//*Tri un tableau A ayant des nombres d'au plus c chiffres*//
pour i←1 à c faire
        utiliser un tri par dénombrement pour trier A selon le chiffre i
fin pour
Retourne la liste triée
```

4.2 Exemple

Nous allons maintenant utiliser cette algorithme à travers un exemple, Soit le tableau d'entier : 19, 5, 66, 123, 245, 802, 221 que l'on va trier dans l'ordre croissant à l'aide du tri par base. Ce tri nécessitera 3 tours, en effet durant le premier tour on triera par rapports aux chiffres des unités, le deuxième tour on triera par rapport aux chiffres des dizaines et le derniers tour on triera par rapport aux chiffres des centaines.

début	premier tour	deuxième tour	troisième tour
19	22$\underline{1}$	5	5
5	80$\underline{2}$	8$\underline{0}$2	19
66	12$\underline{3}$	$\underline{1}$9	66
123	34$\underline{5}$	1$\underline{2}$3	$\underline{1}$23
345	$\underline{5}$	2$\underline{2}$1	$\underline{2}$21
802	6$\underline{6}$	3$\underline{4}$5	$\underline{3}$45
221	1$\underline{9}$	$\underline{6}$6	$\underline{8}$02

Une fois les chiffres triés nous obtenons la liste triée suivante : 5, 19, 66, 123, 221, 345, 802.

4.3 Complexité

Premièrement la complexité de cette algorithme dépendra de l'algorithme utilisé dans la boucle principale. Ici nous utilisons l'algorithme de tri dénombrement.

C'est un tri en temps linéaire $O(N + M)$ avec N la taille du tableau, et M l'élément maximum du tableau. Pour ce tri lorsqu'on a N nombres à C chiffres on obtient une complexité égale à $O(C \times (N + M))$.

Cependant dans le cas de cette algorithme M n'ira jamais au-delà de 9. Donc M devient une constante négligeable, ce qui revient à une complexité $O(C \times N)$,que l'on peut simplifier à une complexité $O(N)$.

4.4 Avantages et inconvénients

Si les nombres utilisés sont de petits entiers, le tri est relativement rapides, les comparaisons sont faites avec quelques opérations qui opèrent en un temps constant. Dans ce cas, le tri par base s'exécutera en O(n) et est alors plus rapide que d'autres algorithmes de tri.

Le désavantage du tri par base est qu'il peut être très lent pour des clefs longues.

5 Conception

6 Résultats

7 Organisation

8 Conclusion

9 Mode d'emploi

10 Annexes