IUT CLERMONT AUVERGNE
Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

DÉPARTEMENT
INFORMATIQUE
Clermont-Ferrand

SAE 2.02

Exploration algorithmique d'un problème

Agostinho Alexandre

24 mars 2023

Table des matières

1 Partie 1 **3**
1.1 Question 1 . . . 3
1.2 Question 2 . . . 3
1.3 Question 3 . . . 3
1.4 Question 4 . . . 3
1.5 Question 5 . . . 5
1.5.1 Analyse de l'algorithme de test de l'accès direct . . . 5
1.5.2 Analyse de l'algorithme de test de l'accès indirect . . . 6

2 Partie 2 **7**
2.1 Question 1 . . . 7
2.2 Question 2 et 4 . . . 7
2.2.1 Analyse de l'algorithme n°1 . . . 7
2.2.2 Analyse de l'algorithme n°2 . . . 7
2.2.3 Analyse de l'algorithme n°3 . . . 8
2.3 Question 3 . . . 9

Introduction

Dans ce rapport nous allons voir differentes études sur l'exploration et la complexité algorithmique d'un problème. Le but étant d'analyser un problème avec méthode et de comparer des algorithmes pour des problèmes classiques.
Nous verrons donc dans un premier temps, l'étude d'un réseau de points caractéristiques et de rues d'une ville, et dans un second temps l'étude de 3 algorithmes de tri avec leur différances, leurs avantages et inconvéniants.

Je tenais aussi à préciser que, par manque de temps, je n'ai pas pu réaliser l'exercice 2.1 et 2.3. Par ailleurs, la question 2.2 et 2.4 ont été répondu ensemble.

1 Partie 1

1.1 Question 1

Pour représenter ce réseaux, je propose la structure de données suivante :

```
#define NAME_MAX_LEN 40
#define NB_PTS_CARACT_MAX 10

typedef struct pointCaracteristique {
    char nom[NAME_MAX_LEN];
    struct pointCaracteristique **tPtsCaract;
    int nbPtsCaract;
} PointCaracteristique, *Rue;

PointCaracteristique *reseau[NB_PTS_CARACT_MAX];
```

La structure du point se compose de son nom et d'un tableau de pointeurs sur d'autre points. Le réseau est ainsi composé d'un tableau de pointeur sur des structures *pointsCaracteristique*, elles-même pointant sur un ou plusieurs point de la liste.

Le réseau est caractérisé par ses points caractéristiques. Chaque point connaît les points accessibles de manière direct. Cela permet de représenter les rues et leur sens. Prenons par exemple, une rue qui part du point A et qui a pour destination le point B. Elle sera caractérisée par la présence du point B dans la liste des points accessibles du point A. De cette manière on peut facilement dessiner le réseau et calculer les trajets à suivre. En somme, les pointeurs servent à caractériser les rues et leurs sens.

1.2 Question 2

Le choix d'une telle structure est fait sur la base de la simplicité d'implémentation. En effet, cette structure est relativement simple. Elle ne travaille quasiment que sur de la manipulation de pointeurs. Le parcours des points caractéristiques est donc facile.

De plus, si on se penche sur la complexité de l'implémentation, le nombre d'opérations requise pour le parcours du réseau est proportionnel au nombre de points que l'on parcourt. Au pire, tout les points seront parcouru : c'est-à-dire que la complexité maximale s'élève au nombre de point qui compose le réseaux.

1.3 Question 3

Pour savoir si A est accessible directement depuis le point B, il suffit juste de regarder si B se trouve dans le tableau des points caractéristiques accessibles directment.

Voici le code correspondant :

```
int isAccessDirect(PointCaracteristique *A, PointCaracteristique *B)
{
    for (int i = 0; i < A->nbPtsCaract; i++)
    {
        if (A->tPtsCaract[i] == B)
        {
            return 1;
        }
    }
    return 0;
}
```

1.4 Question 4

Pour savoir si B est accessible depuis A, directement ou non (par une ou plusieurs rues), il est possible de répéter récursivement la fonction précédante. Ainsi, elle va tester chaque

point independamment et si un chemin direct est trouvé, c'est que il y a forcément un chemain qui rend B accessible.

Voici le code correspondant :

```
// version recusrive
int isAccessREC(PointCaracteristique *A, PointCaracteristique *B)
{
    for (int i = 0; i < A->nbPtsCaract; i++)
    {
        if (isAccessDirect(A, B))
        {
            return 1;
        }
        else if (isAccessREC(A->tPtsCaract[i], B))
        {
            return 1;
        }
    }
    return 0;
}
```

Voici maintenant une ébauche de travail qui visait à créé une version itérative de cette fonction :

```
// version iterative (ebauche de travail)
int isAccessITE(PointCaracteristique *A, PointCaracteristique *B)
{
    int i = 0;

    PointCaracteristique *ptsMarque[NB_PTS_CARACT_MAX];
    int nbPtsMarque = 0;

    PointCaracteristique *ptsTest = A;

    // pour tout les points du point teste
    for (i = 0; i < ptsTest->nbPtsCaract; i++)
    {
        // test si point deja teste
        for (int j = 0; j < nbPtsMarque; j++)
        {
            if (ptsTest == ptsMarque[j])
            {
                break;
            }
        }

        // test si pour ce point le chemin est direct
        if (isAccessDirect(ptsTest, B))
        {
            return 1;
        }

        // le point viens d etre teste, il entre dans la liste
        ptsMarque[nbPtsMarque++] = ptsTest;

        // on teste ensuite le prochain
        ptsTest = ptsTest->tPtsCaract[i];
    }

    return 0;
}
```

Je me suis rendu compte que je n'arrivait pas à penser l'algorithme autrement que récursivement, c'est pourquoi j'ai décidé de mettre cette ébauche de côté.

Néanmoin, il est quand même possible de réutiliser la partie de code qui teste si un point à déjà été testé. Ceci diviserait, par chaque points testé, la complexité de cette algorithme.

Voici ce que donne l'algorithme avec les modifications apportés :

```
int isInTab(PointCaracteristique *P, PointCaracteristique **TabP, int
    lenTabP)
{
    for (int i; i < lenTabP; i++)
    {
        if (TabP[i] == P)
        {
            return 1;
        }
    }
    return 0;
}

// version rapide
int isAccess(PointCaracteristique *A, PointCaracteristique *B,
    PointCaracteristique **ptsMarque, int *nbPtsMarque)
{
    for (int i = 0; i < A->nbPtsCaract; i++)
    {
        if (! isInTab(A, ptsMarque, *nbPtsMarque))
        {
            ptsMarque[*nbPtsMarque++] = A;
            if (isAccessDirect(A, B))
            {
                return 1;
            }
            else if (isAccess(A->tPtsCaract[i], B, ptsMarque, nbPtsMarque))
            {
                return 1;
            }
        }
    }
    return 0;
}
```

1.5 Question 5

1.5.1 Analyse de l'algorithme de test de l'accès direct

Analysons le premier algorithme, c'est-à-dire celui de la question n°3.

Cet algorithme se base sur un parcours de tableau grâce à une boucle for. Dans le principe de ce parcours, on commence du début du tableau, passe un par un les éléments et termine la boucle au moment où l'on trouve l'élèment que l'on cherche.

La boucle for est prévue pour boucler N fois, N étant le nombre d'élèments du tableau dans lequel on effectue la recherche. Cependant, cette boucle est capable de se terminer prématurément si la condition à l'intérieur se valide.

La complexité maximale de ce code est donc linéaire : $o(N)$, où N est le nombre d'élèments maximums à parourir. Le nombre total d'opérations se calcul par la formule :

$$3N + 1$$

avec l'initialisation de la variable i, le test et l'incrémentation de la boucle for, et le test d'égalité de valeur dans la boucle.

1.5.2 Analyse de l'algorithme de test de l'accès indirect

Analysons le second algorithme, c'est-à-dire celui de la question n°4.

Cet algorithme se base sur un parcours récursif sur l'ensemble des points accessibles depuis le point de départ, à la recherche du point d'arrivé. Nous allons étudier la version dite "améliorée" car elle se trouve être plus pertinente. En effet, elle propose de garder en mémoire les points sur lequels l'algorithme est déjà passé et donc déjà testé, de manière à omettre un ènième passage récursif sur ce point, et donc d'éviter une complexité exponentielle et potentiellement une boucle interminable.

Commençons par analyser la fonction $isInTab$, servant de fonction utilitaire afin de rechercher si un $PointCaracteristique$ se trouve dans le tableau des points déjà passé. Cette fonction, est en réalité identique à la fonction de l'exercice n°3, à la différance qu'elle ne recherche pas les valeurs au même endroit. Sa complexité est donc d'ordre linéaire($o(N)$), avec $3N + 1$ instructions effectuées par appel.

Attaquons-nous maintenant au gros de l'algorithme. Comme expliqué plus haut, le principe est d'appliquer cet algorithme sur tout les points en accès direct d'un point. Alors on y retrouve une boucle for qui boucle N fois, N étant ce nombre de points directement accessibles. Ensuite, on utilise la fonction $isInTab$ pour savoir si l'on doit lancer une recherche sur ce point où si cela a déjà été fait. On est donc à une complexité calculée par l'expression :

$$N * 3P + 1$$

où P est le nombre de point déjà testés, et P tendant vers N. Sa complexité maximale est donc donnée par la formule :

$$N\frac{1}{2}N(N+1)$$

Ensuite, après avoir rajouté le point courant à la liste des points déjà testés, on utilise la fonction $isAccessDirect$ afin de savoir si parmi la liste de point en accès direct on retrouve notre destination. On rajoute donc $3N+1$ à la complexité, ce qui nous donne l'expression :

$$N * 2(3P + 1)$$

où N est le nombre de points directement acessible et P même variable que le pour le calcul précédant. Cela nous donne donc l'expression de complexité "au pire" :

$$2N\frac{1}{2}N(N+1)$$

Enfin, et il s'agit ici de la partie la plus délicate de l'analyse, on étudie l'appel récursif de cette fonction. On doit donc imaginer combien de fois cette fonction sera appelée pour multiplier ce nombre par le reste de la complexité de ce code. Mais en réalité, comme on fait attention à ne pas revenir en arrière sur notre chemin, au plus nous parcourerons tout les points du plans. Appelons ce nombre Q. Dans ce cas la complexité maximale, dite "au pire" est donnée par la formule :

$$2Q\frac{1}{2}Q(Q+1)$$

avec Q le nombre total de points du plan de la ville. La compléxité maximal de cet algorithme est donc d'ordre quadratique ($o(N^2)$).

2 Partie 2

2.1 Question 1

...

2.2 Question 2 et 4

Interressons-nous maintenant au calcul de la complexité de chaque algorithmes. Un calcul de omplexité s'effectue en comptabilisant le nombre d'actions que le processeur doit effectuer. Cependant, pour effectuer un tel calcul, il faudrait comptabiliser la totalité des actions effectuées par l'ensemble des bibliothèques utilisées dans les algorithmes. Nous n'avons pas besoin d'aller aussi loin, notamment car dans ce cas, ce calcul changerait en fonction d'un language à un autre, d'un compilateur à un autre, ou encore d'un interpréteur à un autre. Nous allons donc nous intéresser uniquement au nombre de lignes executés par les programmes en fonction de parmamêtres donnés.

2.2.1 Analyse de l'algorithme n°1

Pour ce qui est du premier algorithme proposé, nous pouvons déjà comptabiliser les 4 actions effectuées par la fonction annexe *echanger*. Ajoutons-y les 4 créations de variables en début d'algorithme. On remarque que les instructions d'après sont bouclées en fonction de paramètres variables. On en déduit donc que nous avons déjà 4 instructions fixes et 4 autres que l'on multiplira par le nombre de fois que la fonction *echanger* est appelée.

La première boucle *while*, qui est la boucle principale, tourne tant que la variable *echange* est à 1. Cette variable comptabilise le nombre d'échanges de valeurs qui est effectué pour ordonner le tableau. Notre équation de complexité se construit donc avec la forme :

$$4 + N * (4 + P)$$

où N est le nombre de fois que la boucle tourne et P le nombre d'actions effectuées dans cette boucle.

Dans cette boucle *while*, on trouve deux boucles for pour trier les valeurs du début et de la fin. En analysant l'algorithme caractérisant ces boucles, on en déduit que celui-ci tri le tableau en mettant, pour chaque tours de boucle *while*, l'élement le plus petit en premier, et le plus grand à la fin. P est donc de complexité $N * 2$. On peut alors en déduire que au pire, la complexité de cet algorithme est de :

$$4 + N * (4 + (N * 2))$$

.

En conclusion, on trouve que cette fonction est de complexité quadratique ($o(N^2)$). Cette algorithme ne comporte pas vraiment d'avantages et son gros inconvéniant est que sa complexité est exponentielle : elle augmente de plus en plus vite. Il serait donc très compliqué de trier un très grand nombre de valeurs avec. Cependant, elle peut fonctionner sur de petits nombres de valeurs.

2.2.2 Analyse de l'algorithme n°2

Maintenant, analysons le second algorithme. Pour celui-ci, nous omettrons l'analyse des instructions fixes, car la réelle compléxité des algorithme se base notamment sur les tours de boucles au sein du programme.

La première boucle sert à récupérer les valeurs maximales du tableau. Le nombre de tours que celle-ci effectue est donc égale à la taille du tableau. Appelons cette taille N.

L'instruction *calloc* qui suit cette boucle initialise un tableau de pointeur en initialisant toute les valeurs par defaut à $NULL$. Ce tableau est composé de toutes les valeurs possibles entre la valeur minimale du tableau, et la valeur maximale. C'est-à-dire, si la valeur maximale est 7 et la valeur minimale est 4, ce tableau sera composé de 3 cases. La complexité de cette instruction est donc dépendante de l'écart entre ces deux valeurs. Appelons cette écart P.

La seconde boucle for compte le nombre de valeurs identiques et stocke ce nombre dans le tableau compteur décrit juste avant. Cette boucle tourne autant de fois qu'il y a de valeurs dans le tableau, c'est à dire : N fois.

Viens enfin la dernière boucle for. On retrouve à l'intérieur de celle-ci une boucle $while$, qui tourne tant que le nombre sur lequel pointe le compteur ne vaut pas 0. Le compteur compte le nombre de récurences d'un nombre dans le tableau donc au maximum cette boucle fera $N + P$ tours au total, en comptant les tours de la boucle for dans laquelle elle se trouve.

L'équation final se présente comme :

$$N + P + N + N + P = 3N + 2P$$

La complexité de l'algorithme est donc d'ordre : $o(N)$, c'est à dire une complexité linéaire. Le gros avantage de cet algorithme est que son nombre d'opération augmente proportionnellement en fonction du nombre d'élèments à trier. Cependant, son point faible se situe au niveau de l'écart de valeur qu'il peut y avoir dans la plage de données. Un trop grand écart de valeur resulterait à un très grand nombre P, nombre qui finalement peut avoir plus d'importance dans le calcul de cette complexité.

2.2.3 Analyse de l'algorithme n°3

Interessons nous enfin au troisième algorithme proposé. Tout comme pour la seconde analyse, nous omettrons l'analyse du nombre d'actions fixes.

Regardons tout d'abord la complexité de la fonction de recherche de position : $recherchePos$. Cette fonction contient une boucle for qui, au pire effectue P instructions, P étant passé en second paramètre. En effet, la boucle est prévue pour tourner P fois, mais suivant la réussite de la condition placé à l'intérieur, elle peut se terminer prématurémant.

Maintenant, si l'on regarde la fonction principale de cet algorithme, on remarque une boucle for majeure, paramètrée pour boucler N fois, N étant le nombre d'éléments du tableau à trier. Dans cette boucle, la première chose qui est faite est d'appeler la fonction $recherchePos$, avec en second paramètre une valeur incrémentée de 0 à N à chaque passage de boucle. On déduit donc que le calcul de complexité maximale à ce moment est de :

$$\frac{1}{2}N(N+1)$$

Le principe de l'algorithme à ce moment est de parcourir tout les éléments du tableau et de rechercher leur position idéale.

Ensuite, nous avons la présence d'une condition. Celle-ci teste simplement si l'élèment est oui ou non dans la position idéale considérant que ce tableau est trié. Elle renvoie sur une boucle le cas échéant. Cela veut dire que dans le pire des cas, c'est-à-dire le cas où aucun élément n'est dans sa position idéale, elle renvoie N fois sur cette boucle.

Analysons enfin cette dernière boucle. Celle-ci boucle autant de fois qu'il y a d'écart entre le énième élément du tableau et sa position idéale. Au maximum, cette valeur peut donc s'élever à $N-1$ et décroit au fûr et à mesure que l'on avance dans la boucle principale.

On a donc pour cette boucle, une complexité maximale s'exprimant par :

$$\frac{1}{2}N(N+1)-1$$

Ici, le principe est de décaler toutes les valeurs pour insérer à l'emplacement idéal l'élèment courrant.

Nous pouvons donc maintemant déduire l'équation générale de la compléxité maximale de ce troisième algorithme :

$$(\frac{1}{2}N(N+1))(\frac{1}{2}N(N+1)-1) = \frac{1}{4}N(N+1)(N^2+N-2)$$

On remarque donc la complexité exponantielle de cet algorithme, $o(N^4)$ qui est la plus importante complexité des trois analyses. Cependant, ce n'est pas forcément un algorithme mauvais car il ne s'agit là que de la complexité "au pire". En effet, dans la majeure partie des cas, plusieurs valeurs sont à leur place, et donc cette complexité à tendance à baisser aussi vite qu'à monter. L'avantage est donc que, avec un tableau quasiment trié, cette fonction sera plus rapide que les autre. Dans d'autre cas, elle sera largement plus lente. Donc si l'on ne connaît pas les arrangements de valeurs au préalable du tableau qu'on lui passe, c'est un peu comme si on pariait sur la rapidité de cet algorithme.

2.3 Question 3

...

Conclusion

Nous avons donc pu analyser et comprendre par ces differantes études algorithmiques de problèmes l'importance et l'impact que cela pouvait avoir dans un programme, notamment lorsque il est nécessaire d'effectuer de gros calculs.

Liens annexes

Lien vers le dépot git du projet : https://codefirst.iut.uca.fr/git/alexandre.agostinho/SAE-2.02