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

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 une 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 parcours. 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 chemain direct est trouvé, c'est que il y à 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 :

```
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 pensé l'algorithme autrement que récursivement, c'est pourquoi j'ai décidé de mettre cette ébauche de côté.
Neanmoin, 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 maintenant une ébauche de travail qui visait à créé une version itérative de cette fonction :

```
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

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'action 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 création 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 échange est à 1. cette variable comptabilise le nombre d'échanges de valeurs qui est effectué pour ordonner le tableau. Notre equation 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 tour 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 tres grand nombre de valeurs avec. Cependant, elle peut fonctionner sur des petit nombres de valeurs.

2.2.2 Analyse de l'algorithme n°2

Maintenant, analysons le second algorithme. Pour celui-ci, partons nous ommettrons 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 toute les valeurs

possible entre la valeur minimale du tableau, et la valeur maximale. C'est-à-dire, si la valeur maximal 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 de compteur ne vaut pas 0. Le compteur compte le nombre de récurence 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 ddu 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 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 fixe.

Regardons tout d'abord la complexité de la fonction de recherche de position : "rechercheP os". 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 tournée P fois, mais suivant la réussite de la condition placé à l'intérieur, elle peut se terminée prématurémant.

Maintenant, si 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 "rechercheP os", avec en second paramètre une valeur incrémentée de 0 à N à chaque passage de boucle. On déduit donc que le calcul de complexité maximal 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 test simplement si l'élèment est 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 tabeau 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 toute 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 cette 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'a 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 on ne connaît pas les arrangement de valeur 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