Rapport de projet de Complexité

Carac spéciaux :

∀ , ∃,

TABLE DES MATIERES

I. STRUCTURE DE DONNEES

Afin de réaliser ce projet, nous avons décidé de commencer en utilisant une implémentation matricielle pour les graphes, et une implémentation en liste chainée pour les ensembles de sommets.

Obtenant des résultats satisfaisants, nous avons décidé de conserver nos structures pour l'ensemble du projet.

```
#define n_max 1000

typedef int sommet;

typedef struct {
	int a[n_max][n_max];
	int n;
```

```
        int degre[n_max];
} Graph_m;


typedef struct maillon{
        sommet st;
        struct maillon *suiv;
} maillon;


typedef maillon *liste;
```

II. PARSING DES FICHIERS ET INITIALISATION DES STRUCTURES

➔ Symétrisation des graphes afin de les rendre non-orientés quelques soit la déclaration.

III. VERIFICATION SOUS-GRAPHE DESERT

A) PRINCIPE DE L'ALGORITHME :

L'algorithme a pour but de vérifier qu'un ensemble de sommets passé en paramètre est un sous-graphe désert.

Afin de vérifier le caractère « désert » de l'ensemble de sommets, on vérifie que chaque sommet n'est pas voisin d'un des autres sommets de l'ensemble. On évitera de faire des comparaisons inutiles en ne prenant que les sommets suivants dans l'ensemble.

B) PSEUDO-CODE :

```
Verification_graphe_desert (Graph g, liste X) {
        Pour tout sommet x de X {
                Pour tout sommet x' de X-{x0,….,x,x'} {
                        Si x et x' sont adjacents, retourner faux.
                }
        }
        Retourner vrai.
}
```

C) VALIDITE DE L'ALGORITHME :

L'algorithme est valide puisque l'on crée tous les couples (x , y) possibles en assemblant deux sommets de X, et que pour chacun de ces couples, on vérifie si x possède un arc vers y (et inversement). On vérifie ainsi qu'aucun des sommets de l'ensemble n'est voisin avec un quelconque autre sommet de l'ensemble.

D) COMPLEXITE :

Soient n le nombre de sommet du graphe G = (S, A), et n' le nombre de sommets de l'ensemble X donnée en entrée.

Dans le pire des cas, n' = n et X est désert : c'est le cas lorsque G est un nuage de points et que l'on choisit X = S.
On parcourt alors l'ensemble X, et pour chaque sommet, on parcourt le reste de l'ensemble, donc en prenant 1 sommet de moins à chaque itération, soit :

$$\sum_{i=0}^{n} i = \frac{n(n-1)}{2}$$

Pour chaque itération, on effectue un nombre linéaire d'opérations fondamentales (ici deux comparaisons, puis si le test est positif, une affectation).
D'où une complexité en

$$\theta(n^2).$$

IV. VERIFICATION DE LA MAXIMALITE

A) PRINCIPE DE L'ALGORITHME :

L'algorithme a pour but de vérifier si l'ensemble passé en paramètre est un sous-graphe désert maximal, c'est-à-dire qu'il n'existe pas d'autre sous-graphe désert dans lequel il serait strictement inclus, pour le graphe passé en paramètre.

Afin de traiter la question, nous reformulons la propriété de maximalité :

> *(1) Un sous-graphe désert G' = (S', A') de G=(S, A) est maximal s'il n'existe pas de sommet x $\in$ S, x $\notin$ S', tel que x n'est adjacent à aucun sommet y $\in$ S'.*

Ainsi, nous pouvons répondre en problème en parcourant l'ensemble de sommets X passé en paramètre, et en marquant tous les sommets appartenant à X ou étant adjacent à l'un d'eux.
A la fin du parcours, s'il reste au moins un sommet non-marqué, alors on pourrait ajouter ce sommet à X et créer un sous-graphe désert X' dans lequel serait inclus X. X ne serait alors pas maximal. Dans le cas contraire, il ne reste aucun sommet à ajouter à X, du moins sans casser le caractère « désert » du sous-graphe G'= (X, A').

B) PSEUDO-CODE :

Le marquage des sommets se fera par le biais d'un tableau de booléens de taille n_max :

```
Verification_maximalité (Graph g, liste X) {

    Si X n'est pas un sous-graphe désert, retourner faux.

    Créer tableau_adjancence[n_max].
    Initialiser toutes les cases de tableau_adjacence à 1.

    Pour tout sommet x de X {
        tableau_adjacence[x] = 0.
        Pour tout sommet y de g {
            Si x et y sont adjacents, tableau_adjacence[y] = 0.
        }
    }
    Pour tout entier i de 0 à la taille du graphe g {
        Si tableau_adjacence[i] == 1, retourner faux.
    }
    Retourner vrai.

}
```

C) VALIDITE DE L'ALGORITHME :

Pour le problème donné, il n'existe que trois cas possibles. Si l'algorithme répond correctement à chaque cas, il sera valide :
- X n'est pas un sous-graphe désert : l'algorithme répond que X n'est pas désert maximal, puisque non-désert.
- X est désert mais pas maximal : l'algorithme parcourt le graphe G et marque les différents sommets associés directement ou indirectement à X. A la fin du parcours, il détecte les sommets appartenant à l'ensemble X' maximal incluant X, et répond que X n'est pas maximal.
- X est maximal : l'algorithme parcourt le graphe G, marque les sommets. A la fin du parcours, tous les sommets sont marqués, l'algorithme répond que X est donc maximal.

Finalement, l'algorithme répondant correctement aux différents cas auxquels il peut être confronté, il est valide.

D) COMPLEXITE :

Soient n le nombre de sommet du graphe G = (S, A), et n' le nombre de sommets de l'ensemble X donnée en entrée.

Dans le pire des cas, n' = n et X est maximal. : c'est le cas lorsque G est un nuage de points et que l'on choisit X = S.
On parcourt alors l'ensemble X, et pour chaque sommet, on parcourt le graphe à la recherche de ses adjacents. L'implémentation de nos graphes étant matricielle, la recherche des adjacents est une fonction linéaire en le nombre de sommets du graphe G (parcours d'une colonne/ligne de la matrice d'adjacence). On fait ainsi n^2 itérations de boucles, dans lesquelles on fait un nombre constant d'opérations fondamentales.
On parcourt enfin le tableau d'adjacence, de taille n, ce qui se fait en temps linéaire en fonction de n.

La complexité de cet algorithme de vérification de la maximalité d'un ensemble X dans un graphe G est donc en

$$\theta(n^2).$$

V. RECHERCHE D'UN SOUS-GRAPHE DESERT MAXIMAL

A) PRINCIPE DE L'ALGORITHME :

D'après la propriété (1), on sait qu'un ensemble de sommets X est un sous-graphe désert maximal si on ne peut pas lui ajouter de sommets sans casser le caractère désert.
On peut donc, en partant du graphe G, construire un ensemble de sommets X en choisissant un sommet, en le marquant lui et ses adjacents, puis en choisissant un nouveau sommet parmi les sommets

de G non-marqués, et ce jusqu'à qu'il n'y a plus de sommets à prendre. X sera donc désert et maximal.

B) PSEUDO-CODE :

Le marquage se fera de la même façon que dans la vérification : avec un tableau de booléen.

```
Recherche_maximal (Graph g) {

    Créer une liste X.
    Créer tableau_adjancence[n_max].
    Initialiser toutes les cases de tableau_adjacence à 1.

    Tant qu'il reste des sommets non-marqués {
        Prendre un sommet non-marqué x et l'ajouter à la liste X.
        tableau_adjacence[x] = 0.
        Pour tout sommet y de g {
            Si x et y sont adjacents, tableau_adjacence[y] = 0.
        }
    }
    Retourner X.

}
```

C) VALIDITE DE L'ALGORITHME :

L'algorithme construit un ensemble de sommets X tel que :

1. $\forall (x, y) \in X^2$, x et y ne sont pas adjacents ;
2. $\nexists a \notin X ; \forall x \in X$, a et x ne sont pas adjacents.

1. est vrai puisque lorsque l'on prend un sommet non-marqué, on marque tous ses adjacents. Ainsi, parmi les sommets que l'on choisit, on ne prendra jamais de sommets adjacents à un sommet déjà choisi, puisqu'ils seront déjà marqués et inchoisissables.

2. est vrai car si un tel a existait, alors l'algorithme ne se serait pas arrêté, et l'aurait choisi pour le rajouter à l'ensemble X.

D) COMPLEXITE :

VI. RECHERCHE D'UN SOUS-GRAPHE DESERT MAXIMUM

A) PRINCIPE DE L'ALGORITHME :

L'algorithme a pour but de trouver et de retourner un des sous-graphes déserts maximums d'un graphe G passé en paramètre.
Par définition, on sait qu'un sous-graphe désert maximum est maximal. Or, on sait déjà construire un sous-graphe désert maximal. On peut ainsi trouver une nouvelle formulation du problème :

Retourner un sous-graphe désert maximum ⇔ Retourner un sous-graphe désert maximal de taille maximum.

Notre but devient alors de transformer l'algorithme *Recherche_maximal* afin qu'il ne retourne pas n'importe quel maximal, mais un maximal de taille maximum.

Pour cela, il n'est pas possible de sacrifier le marquage de certains adjacents à chaque passage, ce qui reviendrait à abandonner le caractère désert de l'ensemble.
Il ne reste alors que d'imposer une heuristique sur le choix du sommet à ajouter à la liste à chaque itération. Comme l'objectif de l'algorithme est de retourner un ensemble le plus grand possible, il faut faire le plus d'itération possible, et donc choisir à chaque fois les sommets qui marqueront le moins de sommets.
Ces sommets sont, par définition, les sommets non-marqués de degré minimum dans les graphes Gnm, sous-graphes de G, dans lequel il ne reste que les sommets non-marqués. Ce sont les sommets qui ont le moins d'adjacents non-marqués.
Ainsi, le sommet choisi à chaque itération sera le sommet ayant le degré « non-marqué » minimum.
Afin de déterminer quels seront ces sommets au cours du calcul, nous transformons le tableau d'adjacence en un tableau de degré : ce tableau sera initialisé à la création du graphe, le remplissant des degrés de chaque sommet dans le graphe G initial, et sera mis-à-jour à chaque choix, en utilisant les degrés des sommets dans le graphe Gnm courant.

Cependant, il est possible qu'il existe plusieurs sommets ayant le degré « non-marqué » minimum. C'est ce qu'on pourrait appeler une « divergence » : il existe alors plusieurs chemins à prendre pour la construction du maximum, parmi lesquelles des bons, amenant à un maximum, et des mauvais. Dans ce cas, il n'existe aucune heuristique sûre afin de déterminer quels sommets permettront la formation d'un sous-graphe désert maximum. Il faut donc construire les potentiels maximums avec chacun des sommets, et ce autant de fois qu'il y aura de « divergence » au cours de l'exécution.
Cela complexifie énormément le problème, et sur du grand graphe, le rend insolvable en pratique. Il devient alors intéressant d'utiliser un algorithme qui n'est pas fiable à 100%, mais qui permet de traiter les divergences sans faire exploser la complexité.

Nous implémenterons donc deux algorithmes : *Recherche_maximal_inexacte* et *Recherche_maximal_exacte*.

B) PSEUDO-CODE :

Le but ici d'enlever, à chaque choix, le sommet en question, ses adjacents, et tous les arcs de ces adjacents au graphe Gnm courant. Pour cela, on modifie le tableau de degré en « retirant » les degrés du sommet choisi et de ses adjacents, et en décrémentant les degrés des adjacents des adjacents du sommet choisi.
On sait qu'il n'existe plus de sommets non-marqués lorsque Gnm est vide, c'est-à-dire que tous les degrés du tableau_degré sont négatifs.

Recherche_maximum_inexacte (Graph g) {

Créer une liste X.
Créer tableau_degré[n_max].
Initialiser toutes les cases de tableau_degré avec les degrés des sommets du graphe g.

Tant qu'il reste des sommets non-marqués {
Prendre un sommet de degré non-marqué minimum x et l'ajouter à la liste X.
tableau_degré[x] = -1.
Pour tout sommet y de g {
Si x et y sont adjacents, tableau_degré[y] = -1.
Pour tout sommet z de g {
Si y et z sont adjacents, tableau_degré[z] = tableau_degré[z] – 1.
}
}
}
Retourner X.

}

La principale différence entre l'algorithme exacte et inexacte est l'instruction soulignée : ici, on ne prend qu'un sommet de degré non-minimum, et on continue le calcul du maximum. Peu importe la façon de choisir ce sommet (descente partielle au travers des différentes possibilités, choix aléatoire …), c'est le fait de choisir un et un seul sommet qui permet d'éviter l'exponentiation du calcul.

Le but pour l'algorithme exacte est de construire tous les sous-graphes potentiels. Pour cela, nous utiliserons un algorithme récursif.
Il nous faudra toutefois comparer les différents maximums potentiels construits pour ne garder que le plus grand, ou au moins un d'eux. Pour cela, nous utiliserons une nouvelle structure de données facilitant les comparaisons :

```
typedef struct {
      int taille ;
      liste lx ;
} sous_graphe_max ;
```

```
Recherche_maximal_exacte (Graph g) {

    Créer le sous_graphe_max X. L'initialiser avec une taille nulle et une liste vide.
    Créer le tableau_degré et l'initialiser avec les degrés des sommets du graphe g.
    Retourner Recherche_maximal_exacte_rec (g, tableau_degré, X).

}


Recherche_maximal_exacte_rec (Graph g, int[] tableau_degré, sous_graphe_max X) {

    Créer un sous_graphe_max LX.
    Y insérer les sommets de « degrés non-marqués ».

    //Il ne reste plus de sommets ⇔ X est maximal. Fin du calcul.
    Si LX ne contient aucun élément, retourner X.

    //S'il n'y a qu'un seul sommet, on évite de faire les copies des structures.
    Si LX ne contient qu'un seul sommet x {
        Ajouter x à X.
        Incrémenter la taille de X.
        tableau_degré[x] = -1.
        Pour tout sommet y de g {
            Si x et y sont adjacents, tableau_degré[y] = -1.
            Pour tout sommet z de g {
                Si y et z sont adjacents, tableau_degré[z] = tableau_degré[z] – 1.
            }
        }
        Retourner Recherche_maximal_exacte_rec (g, tableau_degré, X).
    }

    //Traitement des divergences.
    Sinon {
        Créer un sous_graphe_max MAX de taille -1.
        Pour tout x ∈ LX {
            Faire une copie Copie_degré du tableau_degré.
            Faire une copie Copie_sgm de X.
            Ajouter x à Copie_sgm.
            Incrémenter la taille de Copie_sgm.
            Copie_degré[x] = -1.
            Pour tout sommet y de g {
                Si x et y sont adjacents, Copie_degré[y] = -1.
                Pour tout sommet z de g {
                    Si y et z sont adjacents, Copie_degré[z]--.
                }
            }
            Copie_sgm = Recherche_maximal_exacte_rec(g, Copie_degré, Copie_sgm).
```

Si la taille de Copie_sgm > la taille de MAX, MAX = Copie_sgm.

}

Retourner MAX.

}

}

C) VALIDITE DE L'ALGORITHME :

D) COMPLEXITE :

E) OPTIMISATION DE L'ALGORITHME : HEURISTIQUES SUPPLEMENTAIRES :