CAUET Christopher
DEUTSCH Rémi
LOIGNON Lucas
VINCENT Pierre

Rapport de projet de Complexité

Projet 1

TABLE DES MATIERES

I. INTRODUCTION

Ceci est le rapport de projet 1 de Complexité du groupe composé de CAUET Christopher, DEUTSCH Rémi, LOIGNON Lucas et VINCENT Pierre.
Dans ce rapport, nous essayerons d'expliquer au mieux la démarche prise par le groupe pour l'implémentation des différents algorithmes demandés, ainsi que les différents choix faits au cours du projet.
Pour ce faire, pour chaque algorithme, nous détaillerons le but du programme et l'idée directrice de notre implémentation, puis nous présenterons un pseudo-code de l'algorithme, enchainant avec une proposition de validité de l'algorithme, pour finir par un calcul de complexité.

II. STRUCTURE DE DONNEES ET PARSING DES FICHIERS

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, en les modifiant très légèrement au fil du projet. Les structures de bases sont celles présentées ci-dessous :

```
#define n_max 1000

typedef int sommet;

typedef struct {
      int a[n_max][n_max];
      int n;
} Graph;


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

typedef maillon *liste ;
```

On définit une constante n_max, qui constituera la limite en nombre de sommets que nous imposerons à nos graphes.
L'implémentation des graphes se fait ensuite par matrice d'adjacence a. On lui ajoute l'attribut n qui définit le nombre de sommets du graphe.
Enfin, on définit de façon standard une structure pour nos listes chainées, principalement utilisées pour nos ensembles de sommets.

Une fois nos structures mises en place, nous avons implémenté une fonction de parsing afin de pouvoir initialiser nos structures à partir de fichiers passés en paramètre. Pour cela, nous extrayons les données ligne par ligne, en suivant le schéma général présenté :

```
n m
xi1 xj1
xi2 xj2
…
xim xjm
```

où n est le nombre de sommets du graphe, m le nombre d'arcs, et (xi, xj) des arêtes.
Afin de produire une fonction de parsing adapté à notre problème, qui est la recherche de sous-graphe désert dans un graphe **non-orienté**, nous symétrisons les arêtes, ce qui nous permet de traiter des graphes orientés en les transformant en des graphes non-orientés associés, et de considérer les arêtes comme des arcs.

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 s'il existe un arc entre x et y. 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 :

Le pire des cas a lieu lorsque le maximal généré est le plus grand possible et que chaque sommet de sa génération a imposé le marquage du plus grand nombre de sommets.
Cependant, l'augmentation du premier facteur, le nombre de sommets dans le maximal, diminue le deuxième facteur, le nombre d'adjacents par sommet choisi, puisque si l'on choisit beaucoup de sommets, c'est que ceux-ci ont peu d'adjacents. Et inversement, si les sommets choisis ont beaucoup d'adjacents, on en choisira que peu.
On peut donc considérer trois constantes, a, b, c $\in \mathbb{N}$, tel que

$$\begin{cases} \dfrac{n}{a} = |X| \\ b = \forall x \in X, moyenne\big(degre(x)\big) = \dfrac{n}{c} \end{cases}$$

Cette considération est valable puisque le nombre de sommets dans le maximal dépend de n, et que leurs degrés dépendent de m = |A|, qui dépend donc de n.
On sait alors que l'on fera $\frac{n}{a}$ choix de sommets, et que pour chacun d'eux, on marquera $\frac{n}{c}$ adjacents.
Le choix du sommet étant un parcours du tableau d'adjacence, il est donc toujours linéaire en le nombre de sommets du graphe, n.
On sait donc que l'on fera un nombre linéaire en n de choix de sommets et pour chacun d'eux on fera un nombre linéaire en n de marquage.

L'algorithme de recherche de maximal est donc d'une complexité en

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

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 for-

mation 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.
En pratique, le tableau_degré est initialisé au parsing du fichier contenant le graphe G. On notera qu'il est important de ne pas compter en double les degrés (ou de ne pas en compter du tout), lors de la symétrisation.

```
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-marqué 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, que nous appellerons pour chacun des sommets de degrés non-marqué minimum.
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 « degré non-marqué minimum ».

//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.
        }

}
```

B) VALIDITE DE L'ALGORITHME :

L'algorithme inexact n'étant qu'une approximation de l'algorithme exact, nous nous attarderons sur la validité de l'algorithme exact. Afin de prouver cette propriété, nous devons démontrer que :

Théorème 1 :

On définit Γ comme l'ensemble des graphes G = (S, A).

Soit la fonction f : (Γ, S) ⇾ Γ, qui retire un sommet x ∈ S ainsi que ses adjacents à un graphe G ∈ Γ, qui ajoute x à un sous-ensemble de sommets SGM et qui retourne le graphe G' obtenu.

*Si, à partir d'un graphe G ∈ Γ, l'on applique f en utilisant **toujours le** sommet de degré minimum jusqu'à que f retourne le graphe vide, alors SGM est un sous-graphe maximum de G.*

Théorème 2 :

S'il existe n sommets de degré minimum dans un sous-graphe G' de G construit par applications successives de f avec les sommets de degrés minimum, il existe au moins n sous-graphes maximum « potentiels », parmi lesquels au moins un est un sous-graphe maximum du graphe G.

Le théorème 1 prouverait que la méthode utilisée est valide lorsqu'il n'y a jamais de divergence, puisqu'il n'est valide que si au court des applications de f, il n'y a toujours qu'un seul sommet de degré minimum.
Or, en pratique, ce n'est pas toujours le cas. Et c'est pour cela que lors des divergences, nous testons toutes les possibilités. Le théorème 2 complèterait ainsi le théorème 1 et validerait notre algorithme.

<u>Démonstration 1 :</u>
Par définition, on sait que :

$$f(G,x) = G' = G - \{X\}\ tel\ que\ X = \{x;\ \forall\ y \in S\ tel\ que\ (x,y) \in A\}.$$

On enlève donc |X| sommets, qui par définition est égal à degre(x) + 1.
Si x est le seul sommet de degré minimum, alors f enlèvera le moins de sommets (et intrinsèquement d'arcs) possibles.

Ainsi, en appliquant n fois f avec le sommet de degré minimum jusqu'à retourner le graphe vide, on ajoute n sommets à SGM.
*Or, s'il existait une façon d'augmenter la taille de SGM, c'est qu'il existait une façon de faire plus de récurrences, et donc qu'il existait au moins une façon d'enlever moins de sommets lors d'une application : ceci entre en contradiction avec ce que l'on vient d'énoncer ci-dessus, puisque nous utilisons toujours **le** sommet de degré minimum.*
Enfin, on sait que SGM sera un sous-graphe désert puisque sa construction est une spécification de la construction d'un sous-graphe maximal.
SGM étant un sous-graphe désert maximal de G de taille maximum, SGM est donc un sous-graphe maximum de G.

<u>Démonstration 2 :</u>
Soit G' un sous-graphe de G possédant n sommets de degré minimum, construit par applications successives de f en utilisant les sommets de degrés minimum.
Il existe alors n façons d'enlever le moins de sommets possibles en utilisant f à partir de G'.
Or, s'il existait une façon de créer un sous-ensemble de sommets désert plus grand qu'en utilisant un de ces sommets, c'est que cette façon enlèverait moins de sommets à G', ce qui n'est pas possible.
On sait donc qu'au moins une des applications de f aboutit à un sous-graphe maximum de G' et donc de G.
Enfin, puisqu'il est possible que certains sous-graphes de G' possèdent cette propriété au cours des n applications de f, on sait qu'il existera au moins n sous-graphes maximum « potentiels » à cause de G', et autant d'autres qu'il y aura de sous-graphes G'' de G' construit par application de f possédant la même propriété.

Les deux théorèmes étant démontrés, nous pouvons en conclure que notre algorithme est correct.

D) COMPLEXITE :

La complexité de l'algorithme inexact est très proche de celle de l'algorithme de recherche de maximal : en effet, les deux seules différences sont le choix du sommet à marquer et la mise à jour des degrés des adjacents des adjacents du sommet choisi.
Ici, le choix du sommet à marquer est une simple recherche de minimum dans un tableau, ce qui se fait de façon linéaire en la taille du tableau (et donc du nombre de sommets), de la même façon que le parcours. Nous faisons en pratique plus d'opérations, mais la complexité reste la même.
Cependant, nous faisons aussi un certain nombre d'opérations supplémentaires lorsque nous supprimons les arcs des adjacents du sommet choisi : nous parcourons une ligne de la matrice d'adjacence pour chaque adjacent du sommet choisi, ce qui se fait linéairement en le nombre de sommets n du graphe.
Nous faisons donc un nombre linéaire de marquage pour chaque sommet adjacent du sommet choisi, qui sont eux-mêmes rassemblés en un temps linéaire, le tout pour un nombre linéaire de sommets.
La complexité de l'algorithme inexact est donc en

$$\theta(n^3).$$

Cette complexité vaut pour le test d'une seule combinaison de sommets.
Or, pour l'algorithme exact, on doit tester au moins une combinaison par sommet de degré minimum.
Ainsi, pour chacun des $\frac{n}{a}$ choix de sommets, s'il y existe $\frac{n}{k_i} \leq \min\left(\frac{n}{d_i}, n\right) \leq \frac{n}{a}, i \in [0; \frac{n}{a}]$ de sommets de même degré minimum $d_i < a, i \in [0; \frac{n}{a}]$, alors on devra tester $\prod_{i=0}^{\frac{n}{a}} \frac{n}{k_i}$ combinaisons.

Or, $\prod_{i=0}^{\frac{n}{a}} \frac{n}{k_i} \leq n!$ puisque ce nombre de combinaisons ne peut pas dépasser le nombre total de combinaisons avec ordre possibles pour n sommets, qui vaut $n!$.
De plus, pour chaque combinaison, nous allons donc faire un nombre linéaire de marquage d'adjacents, et pour chacun desquels nous ferons en plus un nombre linéaire de marquage de leurs propres adjacents.
Finalement, le nombre d'appels récursifs étant majoré par $n!$ et la complexité des appels récursifs majoré par le carré du nombre de sommets, la complexité de l'algorithme exact est donc en :

$$O(n^2 * n!).$$

E) OPTIMISATION DE L'ALGORITHME : HEURISTIQUES SUPPLEMENTAIRES :

L'algorithme exact est long, ce qui n'est pas étonnant au vu du problème auquel il répond. Cependant, cet algorithme ne traite pas que le cœur du problème : en fait, il teste un bon nombre de cas inutile pour le problème de recherche proposé ici. Nous allons ainsi essayer de réduire le nombre de combinaisons possibles à tester en soulignant deux propriétés :

Propriété 1 :
Soit G' = (S', A') un sous-graphe de G = (S, A) construit à partir d'appels récursifs de l'algorithme exact.
Pour tout sommet x ∈ S' tel que degre(x) = 0 dans G', x appartient à un sous-graphe maximum de G.

Propriété 2 :
Soit G' = (S', A') un sous-graphe de G = (S, A) construit à partir d'appels récursifs de l'algorithme exact.
Si $\forall\, x, x' \in S',\ degre(x) = degre(x')$*, alors le choix du sommet pour le prochain appel récursif n'aura pas d'influence sur la taille du sous-graphe retourné à la fin de l'algorithme.*

Démonstration 1 :
Grâce à la validité de l'algorithme, on sait que tous les sommets de G' sont des sommets susceptibles d'intégrer l'ensemble de sommets courant sans briser le caractère désert.
Cependant, si on choisit un sommet x parmi les n sommets de degré 0 pour un appel récursif, on ne supprime aucun sommet ni arc puisque x n'a pas d'adjacents. Les degrés des autres sommets restent donc inchangés. Le prochain choix se fera donc entre les n-1 sommets de degré 0 restants, et pas plus puisque les autres degrés n'ont pas été modifiés.
On utilisera donc les n sommets jusqu'à qu'il n'y ait plus de sommets de degré 0.
Ainsi, peu importe l'ordre dans lequel on choisit les sommets de degré 0, ils appartiendront tous au sous-graphe maximum de G construit.

Démonstration 2 :
Si tous les sommets ont le même degré d, alors le graphe G' est l'union strict de k sous-graphes connexes de taille n_k *dont tous les sommets sont de degré d. Choisir un sommet dans un sous-graphes connexes ne modifiera pas la structure des autres sous-graphes, nous pouvons donc choisir un sommet indifféremment du sous-graphe dans lequel il se trouve.*

Nous nous placerons donc dans le sous-graphe $x = (Sx, Ax)$ *de taille* n_x*. Comme tous les sommets de* x *sont de même degré, ils ont tous une structure voisine similaire : un voisinage de d de sommets ayant eux-mêmes d sommets voisins, etc …*
On peut ainsi mettre n'importe quelle étiquette sur un certain sommet et reconstruire un isomorphe de x *à partir des adjacences de cette étiquette : par exemple, on place l'étiquette 1 sur n'importe quel sommet, on étiquette les sommets adjacents avec les adjacents de 1 dans* x*, puis*

on étiquette les adjacents de ces nouveaux sommets avec les adjacents des adjacents de 1 dans x, etc …
Puisque l'on peut le faire pour n'importe quel sommet ou n'importe quel étiquette, il existe au moins n_x isomorphismes de x.
On dessine maintenant x dans le plan. Il existe donc un sommet g qui est le plus à gauche dans le plan. On considère que le sommet choisi pour le prochain appel récursif de l'algorithme sera ce g.
Or, on sait que ce sommet peut être étiqueté avec n'importe quel $i \in Sx$ tout en formant un isomorphe de x. On sait aussi que supprimer les adjacents de g, quelque soit son étiquetage, supprimera bien ses adjacents. Peu importe l'étiquetage du sommet, nous modifierons donc la structure de x de la même façon.
On peut en conclure que :

$$\forall g, g' \in Sx, f(x, g) \text{ est un isomorphisme de } f(x, g').$$

Enfin, puisque que peu importe le sommet choisi, l'application de f sur x en choisissant g forme un isomorphisme des autres applications sur x, alors le choix du sommet n'aura aucune influence sur le nombre de récursions nécessaires à la « destruction » de x.

Finalement, sachant que si tous les sommets du graphe G' ont le même degré, ni le sous-graphe de G' dans lequel on choisit le sommet, ni le sommet lui-même dans ce sous-graphe n'a d'importance, alors le choix du sommet pour la prochaine récursion lorsque tous les degrés de G' sont égaux n'a donc aucune importance.

Ces deux propriétés prouvées, nous allons pouvoir les utiliser pour créer des heuristiques sur le calcul de notre sous-graphe maximum, en fonction du degré minimum dans le graphe G' courant :

- si celui-ci est nul, on ajoutera simplement tous les sommets de degré minimum à l'ensemble de sommets courant. On évite ainsi le parcours de la matrice pour trouver les adjacents et surtout de tester les différentes permutations d'un ensemble contenant ces sommets.
- si tous les sommets ont le même degré, on choisit le premier de ceux-ci, on le marque lui et ses adjacents, on l'ajoute à l'ensemble de sommets et on continue. On évite ainsi de tester pour chacun des sommets les différentes combinaisons, en n'en testant qu'une seule.

Ces deux optimisations ne changent pas la complexité de l'algorithme, mais le rendent bien plus performant en pratique, en lui permettant d'éviter de tester un bon nombre de combinaisons.

VII. DONNEES PRATIQUES :

Ci-dessous différentes exécutions de l'algorithme de calcul du maximum exact sur un i7-3632QM @ 2.20GHz avec 8Go de RAM :

```
./calcul_maximum_exact Benchs/tr1
Nombre de sommets : 50
Nombre d'aretes : 319
```

```
Le sous-graphe maximum exact généré est :
[ 14 33 38 18 49 24 46 ], taille 7.
Temps d'exécution en secondes = 0.012505
```

```
./calcul_maximum_exact Benchs/tr2
Nombre de sommets : 50
Nombre d'aretes : 380

Le sous-graphe maximum exact généré est :
[ 18 49 34 26 39 ], taille 5.
Temps d'exécution en secondes = 0.013865
```

```
./calcul_maximum_exact Benchs/myciel4
Nombre de sommets : 23
Nombre d'aretes : 71

Le sous-graphe maximum exact généré est :
[ 11 13 21 12 16 18 17 20 19 15 14 ], taille 11.
Temps d'exécution en secondes = 0.009160
```

```
./calcul_maximum_exact Benchs/1-FullIns_3
Nombre de sommets : 30
Nombre d'aretes : 100

Le sous-graphe maximum exact généré est :
[ 4 7 1 21 23 19 25 22 18 28 26 24 20 5 ], taille 14.
Temps d'exécution en secondes = 0.035159
```

```
/calcul_maximum_exact Benchs/queen8_12
Nombre de sommets : 96
Nombre d'aretes : 2736

Le sous-graphe maximum exact généré est :
[ 27 41 78 95 58 21 68 7 ], taille 8.
Temps d'exécution en secondes = 0.012286
```

```
./calcul_maximum_exact Benchs/CELAR6-SUB4
Nombre de sommets : 44
Nombre d'aretes : 499

Le sous-graphe maximum exact généré est :
[ 23 34 43 19 29 ], taille 5.
Temps d'exécution en secondes = 0.000267
```

On notera que l'ordinateur test plante sur des graphes plus difficiles (plus grand ou moins complets), comme tr3 ou myciel5, où le nombre d'appels récursifs explose.

A titre de comparaison, voici deux exécutions de l'algorithme inexact :

```
./calcul_maximum_inexact Benchs/tr1
```

```
Nombre de sommets : 50
Nombre d'aretes : 319

Le sous-graphe maximum approché généré est :
[ 0 25 39 36 19 18 47 ], taille 7.
Temps d'exécution en secondes = 0.000053
```

```
./calcul_maximum_inexact Benchs/tr5
Nombre de sommets : 1000
Nombre d'aretes : 8474

Le sous-graphe maximum approché généré est :
[ 577 433 652 455 716 340 335 319 698 543 510 138 915 862 417 165 929 818
639 842 527 942 736 686 368 217 499 1 786 955 807 188 891 876 110 761 627
380 669 855 831 595 476 619 359 122 94 50 275 34 901 253 799 779 571 488
775 399 307 208 199 299 157 139 67 909 750 886 680 235 149 963 769 497 616
452 298 178 985 853 980 954 610 606 540 28 412 285 266 23 889 732 185 105
87 62 390 983 911 602 397 969 292 962 995 887 272 85 263 998 908 474 ],
taille 112.
Temps d'exécution en secondes = 0.005021
```

VIII. REPARTITION DU TRAVAIL :

	Algorithme	Rapport	Complexité	Validité
Parsing/Structures	Lucas LOIGNON	Remi DEUTSCH	_	Pierre VINCENT
Vérification désert	Lucas LOIGNON Christopher CAUET	Remi DEUTSCH	Lucas LOIGNON	Pierre VINCENT
Vérification maximal	Lucas LOIGNON Christopher CAUET	Remi DEUTSCH	Lucas LOIGNON	Pierre VINCENT
Calcul maximal	Lucas LOIGNON	Remi DEUTSCH	Lucas LOIGNON	Pierre VINCENT
Calcul maximum inexact	Pierre VINCENT	Remi DEUTSCH	Pierre VINCENT	Remi DEUTSCH
Calcul maximum exact	Remi DEUTSCH	Remi DEUTSCH	Remi DEUTSCH	Pierre VINCENT