Complexité I
Rapport de projet

Alexandre Léonardi Ghislain Dugat Guélaud Lepetit
Abdelkader Benameur Loïc Vierin Amine Zayd

2 novembre 2015

Résumé

Rapport du projet de Complexité de Master 1. Ce rapport comporte une analyse des algorithmes utilisés pour résoudre les problèmes suivants :

- Vérification de la propriété « désert » d'un sous-graphe
- Vérification de la propriété « désert maximal » d'un sous-graphe
- Calcul d'un sous-graphe désert maximal quelconque à partir d'un graphe donné
- Calcul d'un sous-graphe désert maximum quelconque à partir d'un graphe donné
- Calcul d'un sous-graphe désert "'presque maximum"' à l'aide d'une heuristique laissée à notre discrétion

Chacun des algorithmes sera brièvement expliqué et accompagné de sa complexité et de son temps d'exécution.

Par ailleurs le tableau récapitulant la participation des différents membres du groupe, ainsi que la configuration de l'ordinateur utilisé pour mener les tests et enregistrer les temps d'exécution, seront présentés dans une première partie.

Table des matières

1 Participations au projet & configuration de tests **3**

2 Types de donnée utilisés **4**
2.1 Graphes 4
2.2 Sous-graphes 4
2.3 Ensembles de sous-graphes 4

3 Alogirthmes de vérification **5**
3.1 Propriété « désert » 5
3.1.1 Principe 5
3.1.2 Validité 5
3.1.3 Complexité 6
3.2 Propriété « maximal » 6
3.2.1 Principe 6
3.2.2 Validité 7
3.2.3 Complexité 7

4 Génération d'un sous-graphe maximal **8**
4.1 Principe 8
4.2 Validité 8
4.3 Complexité 8

5 Génération d'un sous-graphe maximum **9**
5.1 Méthode exacte 9
5.1.1 Principe 9
5.1.2 Validité 10
5.1.3 Complexité 10
5.2 Méthode incomplète 11
5.2.1 Principe 11
5.2.2 Validité 11
5.2.3 Complexité 11

Algorithme	**Auteur du code**	**Auteur du rapport**	**Complexité**	**Validité**
is_desert	Guélaud	Alexandre	Guélaud	Guélaud
is_maximal	Ghislain	Alexandre	Ghislain	Ghislain
maximal	Loïc	Alexandre	Loïc	Loïc
maximum_exact	Alexandre	Alexandre	Alexandre	Alexandre
maximum_partial	Abdelkader	Alexandre	Abdelkader	Abdelkader

TABLE 1 – Récapitulatif des contributions au projet

1 Participations au projet & configuration de tests

Le tableau TAB 1 a été difficile à établir car la participation de chacun n'est pas toujours précisément mesurable, notamment quand il s'agit de réfléchir à plusieurs sur l'écriture d'un algorithme ; cela a notamment été le cas pour la fonction maximum incomplète qui, bien que probablement triviale, nous a demandé un certain temps de réflexion.

Comme on peut le voir, les personnes qui se sont occupées de la rédaction d'un algorithme se sont aussi chargées de sa vérification et de sa complexité, tandis que le présent rapport est pour sa part l'œuvre d'une seule personne.

Configuration de test :

- *OS* Windows 8.1 Professionnel 64 bits
- *CPU* Intel Core i7-7400MQ 2.40GHz
- *GPU* Nvidia GeForce GTX 760M
- *RAM* 8GB DDR3

2 Types de donnée utilisés

Les types présentés ci-après sont trouvables dans le fichier `types.h`.

2.1 Graphes

Les graphes sont représentés à l'aide de la méthode des matrices d'adjacence. Ce choix a été fait pour des raisons de simplicité de la représentation principalement.

En effet, même si elle s'avère plus consommatrice en mémoire qu'une représentation par liste chaînée, le gain en temps de codage n'est pas négligeable. Dans la mesure où l'ordinateur de test disposait d'une grande quantité de mémoire vive, ce défaut ne nous a pas paru rédhibitoire.

L'autre raison est la vérification en temps constant de l'existence d'un arc qui est utilisée à de nombreuses reprises dans le code.

2.2 Sous-graphes

Les sous-graphes sont représentés comme des tableaux de booléens dont la case d'indice i vaut 1 si le sommet i appartient au graphe.

Un alias nous permet de faciliter la représentation en donnant un nom explicite à ces tableaux et en fixant leur taille maximum comme étant le nombre de sommets que peut comporter un graphe au maximum.

Cette représentation a également l'inconvénient de la perte de mémoire, mais encore une fois la quantité de RAM disponible nous a mené à penser que ce ne serait pas un problème.

2.3 Ensembles de sous-graphes

Cette structure de donnée est utile dans le cadre des sous-graphes maximum, où, comme expliqué plus tard, il faudra trouver un ensemble de sous-graphes maximaux et les comparer entre eux.

Il s'agit en fait d'une liste simplement chaînée de sous-graphes. Chaque maillon, appelé élément, comporte donc un sous-graphe et un pointeur sur son successeur.

Une représentation statique aurait été problématique car il était impossible de savoir à l'avance combien de sous-graphes il nous faudrait stocker. Qui plus est, même avec une importante mémoire vive disponible, une telle structure codée srtatiquement pourrait s'avérer problématique.

3 Alogirthmes de vérification

Cette partie regroupe les deux algorithmes de vérification prenant en paramètre un graphe et un sous-graphe, et vérifiant si le sous-graphe est, respectivement, désert et maximal.

Étant très similaires de fonctionnement, nous avons regroupé ces deux algorithmes dans la même partie. À noter qu'ils ne vérifient pas si le sous-graphe est bien un sous-graphe du graphe passé en paramètre : on suppose que les paramètres passés sont corrects.

3.1 Propriété « désert »

3.1.1 Principe

```
booléen is_desert(graphe G, sous-graphe S){
    booléen desert = vrai
    entier n = taille de G
    matrice arc = ensemble des arcs de G
    pour i de 0 à n
        pour j de i+1 à n
            si (S[i]==1 et S[j]==1 et arc[i][j]==1) alors
                desert = faux
                sortir des boucles
            fsi
        fpour
    fpour
    retourner desert
}
```

Le principe de cet algorithme est de parcourir l'ensemble des arcs possibles à l'aide d'une double boucle et de vérifier, pour chaque arc :

1. si ses deux extrémités sont présentes dans le sous-graphe
2. s'il est présent dans le graphe

Si tel est le cas, cela veut dire que le sous-graphe n'est pas désert (deux de ses sommets sont reliés entre eux), on sort donc des boucles et on retourne faux.

Si rien ne nous arrête, alors on retourne vrai. En pratique pour éviter l'usage d'un `break`, les boucles sont des `while` qui vérifient à chaque itération la valeur du booléen.

3.1.2 Validité

La rigueur d'une preuve formelle n'est pas requise pour ce projet, or si l'on ne va pas jusque là, il n'y a que peu à rajouter : l'algorithme applique exhaustivement la définition d'un sous-graphe désert.

Tous les arcs possibles sont dûment vérifiées compte tenu de cette définition (deux sommets d'un sous-graphe désert ne peuvent pas être reliés par un arc du graphe) ; le seul cas non-traité possible est celui où le sous-graphe comporte des sommets qui ne feraient pas partie du graphe de départ, dans cette éventualité ces somemts sont simplement ignorés.

3.1.3 Complexité

La taille de G (le nombre de sommets qui le compose) notée n s'obtient en temps constant car elle fait partie de la structure représentant un graphe, de même que la matrice arc.

Le pire des cas possibles est celui où le sous-graphe est bel et bien désert : il faudra parcourir l'intégralité des deux boucles pour arriver à cette conclusion, sachant que le traitement à l'intérieur des boucles est lui aussi en temps constant (trois tests). On a donc une complexité $C_1(n)$ de la forme suivante :

$3\sum_{k=0}^{n-1} k = \frac{3n(n+1)}{2}$

Cela étant de l'ordre de n^2, on obtient la complexité suivante pour notre algorithme :

$\theta(n^2)$

Temps d'exéc!!!

3.2 Propriété « maximal »

3.2.1 Principe

```
booléen is_maximal(graphe G, sous-graphe S){
    booléen maximal = vrai
    entier n = taille de G
    matrice arc = ensemble des arcs de G
    si(is_desert(G,S) == 0) maximal = faux
    sinon
        entier i=0
        tant que (i<n et maximal==vrai)
            si(S[i] == 0)
                S[i] = 1
                maximal = is_desert(G,S)
                S[i] = 1
            fsi
        ftq
    fsi
    retourner maximal
}
```

Le principe de cet algorithme est de parcourir l'ensemble des sommets de G non-présents dans S, et pour chacun d'entre eux effectuer la procédure suivante :

1. ajouter ce sommet à S
2. regarder si S est désert
3. retirer ce sommet de S

Si une fois au moins on obtient un S désert dans l'étape 2, cela veut dire que notre sous-graphe n'est pas désert maximal et donc on quitte la boucle et renvoie faux, sinon on consomme les n tours de boucle puis on renvoie vrai.

3.2.2 Validité

On sait qu'un sous-ensemble d'un sous-graphe désert sera lui-même désert. Ainsi pour un sous-graphe désert composé de k sommets il n'y a que 2 possibilités, soit il est maximal, soit il est strictement inclus dans un sous-graphe désert maximal M.

En ajoutant chaque sommet de G un à un à S, on obtient au moins une fois M, soit un sous-graphe S' tel que $S \subset S' \subset M$.

De ce fait tester le non-respect de la propriété « désert » pour chaque sous-graphe de taille k+1 contenant S nous permet de savoir si ce dernier est maximal ou pas.

3.2.3 Complexité

Le pire des cas est celui où S est désert maximal et ne comporte qu'un seul sommet (le cas de S vide impliquerait que G soit vide aussi, donc que n==0, ce qui fait que les boucles de la forme `pour i de 0 à n` seraient somme toute assez rapides à exécuter).

Nous ferons dans ce cas n tours de boucle, dont n-1 mènent à un appel à `is_desert` dont la complexité est en $\theta(n^2)$. Il y a de plus un appel à `is_desert` en-dehors de la boucle ce qui nous donne la complexité suivante pour `is_maximal` :

$\theta(n^3)$

Temps d'exéc ! ! !

4 Génération d'un sous-graphe maximal

Il s'agit de fournir un des sous-graphes maximaux possibles pour un graphe donné, sans aucune préférence dans le cas où ils seraient multiples.

4.1 Principe

```
void maximal(graphe G, sous-graphe S){
    entier n = taille de G
    entier i = 0
    tant que (i<n et non(is_maximal(G,S))
        S[i] = 1
        si(non(is_desert(G,S)) S[i] = 0
        i++
    ftq
}
```

L'idée ici est de traiter tous les sommets en partant de 0 et en allant jusqu'à n. Le sous-graphe est initialement vide et l'on y ajoute tous les sommets de G un à un, sous réserve qu'ils ne fassent pas disparaître la propriété « désert ».

Quand notre sous-graphe S est maximal, la fonction se termine.

4.2 Validité

Les cas où G comporte 0 ou 1 sommet sont triviaux : dans le premier cas la boucle n'est pas exécutée et nous obtenons un S vide, ce qui est bien un sous-graphe maximal d'un graphe vide. Dans le second cas la boucle est exécutée une fois et S sera composé de l'unique sommet de G ce qui encore une fois est un sous-graphe maximal valide.

Considérons donc le cas où n $\geq$ 1 et faisons une preuve par récurrence sur i :

- *i=0* S étant un singleton contenant le sommet d'indice 0 de G, il ne contient *a priori* pas deux sommets reliés par un arc et est donc désert.
- *i=k/k>0 et k<n* On suppose qu'à l'itération k-1 S était bel et bien désert. S'il était en plus maximal, on le découvre à la condition d'entrée de la boucle, et celle-ci ne s'exécute pas (on quitte la fonction comme prévu). Si l'ajout i^{eme} sommet de G à S rend ce dernier non-désert on le retire immédiatement après. Sinon, il est conservé. Dans tous les cas : nous avons vérifié si le sous-graphe de l'itération précédente était maximal, et si non, nous avons avancé dans le parcours des sommets tout en conservant un sous-graphe désert.
- *conclusion* Notre graphe de départ est bien désert, et cette propriété est conservée à chaque itération. G étant fini, on ne peut pas ajouter des sommets à l'infini dans un de ses sous-graphes, et la propriété « désert » étant conservée de proche en proche, un sous-graphe désert maximal finit nécessairement par être atteint.

4.3 Complexité

Il y a au pire n tours de boucle, chaque tour de boucle comportant un appel à une fonction en $\theta(n^2)$ et une fonction en $\theta(n^3)$. On a donc une complexité globale en $\theta(n(n^3))$ soit : $\theta(n^4)$

Temps d'exéc!!!

5 Génération d'un sous-graphe maximum

Regroupés ici sont les deux algorithmes permettant d'obtenir des sous-graphes maximum. Le premier utilise une méthode exacte mais particulièrement lourde tandis que le second se contente d'une heuristique arbitraire offrant un sous-graphe s'approchant d'un maximum mais sans garantie précise sur sa 'qualité'.

5.1 Méthode exacte

5.1.1 Principe

```
void maximum_exact(graphe G, sous-graphe S){
    sous-graphe tmp
    ensemble-de-sous-graphes ens
    maximum_exact_rec(G,tmp,&ens,0)
    extract_maximum(ens,S)
}

void maximum_exact_rec(graphe G, sous-graphe S, ensemble-de-sous-graphes ens, entier pro
    booléen continuer = vrai
    entier n = taille de G
    si(S non-vide)
        si(is_desert(S,G))
            si(is_maximal(S,G))
                memorize(S,ens)
                continuer = faux
            fsi
        sinon
            continuer = faux
        fsi
    fsi
    si(continuer)
        si(prof<n)
            sous-graphe S1,S2
            copy(S,S1)
            copy(S,S2)
            S2[prof] = 1
            maximum_exact_rec(G,S1,ens,prof+1)
            maximum_exact_rec(G,S1,ens,prof+1)
    fsi
    fsi
}
```

Voici une bréve explication des fonctions de plus bas niveau qui sont appelées par celles détaillées ici :

- `extract_maximum` prend en paramètre une liste chainée de sous-graphes et un sous-graphe vide ; elle copie un des sous-graphes de taille maximum de l'ensemble dans le sous-graphe vide
- `memorize` enregistre un sous-graphe donné dans une liste chainée de sous-graphes en créant un nouveau maillon et l'y copiant

- `copy` copie, comme son nom l'indique, un sous-graphe source dans un sous-graphe destination, écrasant tout ce qui se trouvait dans ce dernier auparavant

Le principe de la partie récursive de l'algorithme est de vérifier dans un premier temps si on est dans un cas terminal, c'est-à-dire si le sous-graphe passé en paramètre est désert maximal. Si c'est le cas, on le stocke dans notre liste de sous-graphes et il n'y a pas d'appel récursif.

Si ce n'est pas le cas (mais si le sous-graphe est tout de même désert, sinon il est inutile de continuer à développer cette chaîne d'appels) on crée deux branches, une où l'un des sommets de G est ajouté à S en fonction de la profondeur à laquelle on se trouve dans la suite d'appels récursifs, et une où S n'est pas modifié.

Puis deux appels récursifs sont faits, avec une profondeur incrémentée et chacun des deux sous-graphes mentionnés précédemment. Incidemment, nous allons créer tous les sous-graphes possibles de G et tester chacun d'entre eux pour mettre de côté tous les maximaux.

Une fois ces appels récursifs terminés, c'est la fonction `extract_maximum` qui parcourt tous les résultats obtenus et en extrait un de taille la plus grande.

5.1.2 Validité

Tout d'abord, notre fonction récursive atteindra toujours son cas terminal. La mécanique est la même que pour la fonction de génération d'un sous-graphe maximal : nous rajoutons des sommets au sous-graphe, si bien que l'on arrive nécessairement dans un des deux cas terminaux possibles (soit le sous-graphe cesse d'être désert, soit il devient désert maximal).

La principale différence vient du fait que nous testons exhaustivement les sous-graphes possibles plutôt que de suivre un seul chemin pseudo-aléatoire.

Le bon fonctionnement de la fonction `maximum_exact` découle de la supposée validité des fonctions `is_maximal` et `is_desert` ; en effet, ce sont celles-ci qui permettent de stopper la récursion et de reconnaître les sous-graphes à mémoriser.

5.1.3 Complexité

La complexité de cette fonction est plus longue à calculer que les précédentes. Chaque appel récursif en engendrant 2 nouveaux jusqu'à une profondeur maximum de n, où n est la taille de G, nous avons au final 2^n appels récursifs.

n d'entre eux sont des cas terminaux donc la complexité est de l'ordre de celle de la fonction `memorize` (à savoir $\theta(NMAX)$ où NMAX est le nombre maximum de sommets d'un graphe).

$2^n - n$ sont des cas non terminaux d'une complexité de l'ordre de celle de `copy` (à savoir $\theta(NMAX)$).

Ainsi l'ensemble de l'arbre d'appels récursifs est d'une complexité globale de l'ordre de $\theta(2^n \times NMAX)$.

Outre ces appels récursifs, la fonction `maximum_exact` fait appel à `extract_maximum` qui appelle autant de fois `copy` qu'il y a de sous-graphes maximaux dans G, mais je n'ai pas su calculer la complexité de cette fonction.

Pour conclure, je pense donc que la fonction `maximum_exact` est d'une complexité exponentielle par rapport à la taille du graphe G en entrée, mais sans en être sûr :

$\theta(2^n)$

Temps d'exéc ! ! !

5.2 Méthode incomplète

5.2.1 Principe

```
void maximum_partial(graph G, sous-graphe S){
    entier index
    sous-graphe sommets_intedits
    tant que(!is_maximal(G,S))
        index = vertex_with_less_edges(G,sommets_interdits)
        sommets_interdits[index] = 1
        S[index] = 1
        si(!is_desert(G,S)) S[index] = 0
    ftantque
}
```

`vertex_with_less_edges` est une fonction qui, pour un graphe G et un sous-graphe de sommets à ne pas prendre en compte I, va retourner l'indice du sommet de G privé de I ayant le plus petit nombre d'arcs incidents.

L'idée est que, pour avoir de bonnes chances de générer un sous-graphe maximum, nous allons regarder le sommet avec le moins d'arcs incidents et l'ajouter à notre sous-graphe.

Si le sous-graphe est toujours désert, nous poursuivons les tours de boucle jusqu'à ce qu'il soit maximal, mais il faut non seulement rajouter ce sommet à l'ensemble des sommets à ne plus traiter, mais aussi tous ses sommets voisins.

Si le sous-graphe n'est plus désert, le sommet en est retiré pour revenir à un sous-graphe désert, mais il est quand même rajouté à la liste des sommets interdits pour ne pas être traité une seconde fois par la suite (sans quoi l'algorithme tournerait en boucle à traiter encore et encore le même sommet).

Quand on a atteint un sous-graphe désert maximal comportant les sommets avec le moins d'arcs incidents possibles, on le retourne comme étant (probablement) maximum.

5.2.2 Validité

En supposant la validité de `vertex_with_less_edges`, au pire des cas tous les sommets de G seront traités et ajoutés à S, mais nous finirons nécessairement par avoir un sous-graphe désert maximal.

Or, ne cherchant pas l'exhaustivité ici, nous conservons le premier désert maximal rencontré comme résultat final.

5.2.3 Complexité

Dans `maximum_partial`, au maximum n tours de boucle sont effectués (dans le cas où tous les sommets doivent être parcourus pour obtenir un sous-graphe désert maximal). Chaque tour de boucle comporte 2 appels, `vertex_with_less_edges` et `is_desert`.

On a déjà vu que la seconde fonction a une complexité en $\theta(n^2)$. Le première a également une complexité en $\theta(n^2)$ car elle comporte 2 boucles imbriquées (l'une étant de la forme pour i de 0 à n, l'autre pour j de i+1 à n), et ces boucles effectuent un traitement en temps constant.

Notre fonction `maximum_partial` a donc une complexité globale en : $\theta(n^3)$

Temps d'exéc ! ! !