Université Libre de Bruxelles
Faculté des Sciences
Département d'informatique

Projet

Luyckx Marco

Professeur: Jean Cardinal

Cours : Algorithmique 2 INFO-F203

Remis en avril 2021

Année académique 2020-2021

Table des matières

1 Introduction

Dans ce projet d'Algorithmique 2, nous devons décrire des algorithmes qui permettent de répondre à deux questions :
1) Est-ce que le nombre chromatique d'un graphe est inférieur, égale ou supérieur à un nombre entier k positif donné en paramètre ? (Coloration)
2) Est-il possible de répartir une collection de tâches sur un nombre k de machine donné en paramètre tout en sachant que ces tâches doivent être effectuées à un moment précis (temps de début et de fin) ? (Ordonnancement)

Ce sont tous les deux des problèmes intéressants à leur manière. L'un permet d'entrevoir les difficultés et limitations, à l'heure actuelle, de "grands" problèmes en informatique tandis que l'autre est plus "simple" algorithmiquement parlant. Pour les deux problèmes, on commence par expliquer la manière dont fonctionne l'algorithme, les réponses à différentes questions, les explications sur les structures utilisées avec des commentaires et finalement les exemples d'exécutions.

Le langage utilisé est Java 15.0.2.

Le fichier *README.md* contient toutes les informations nécessaires à la compilation et au lancement des programmes.

Pendant la réalisation de ce projet, j'ai discuté avec Attilio Discepoli et Tiago Marques Correia.

2 Définitions

Définition 2.1 *Le **backtracking** peut être défini comme, je cite : "une technique algorithmique générale qui envisage de rechercher toutes les combinaisons possibles afin de résoudre un problème informatique" [1].*

Définition 2.2 *Un **algorithme glouton** construit une solution étape par étape, en choisissant toujours un optimum local qui offre donc le bénéfice le plus évident et immédiat pour ainsi essayer d'obtenir un optimum global [2][3].*

Définition 2.3 *Je cite : "L'acronyme **"NP"** signifie "Nondeterministic Polynomial". La classe NP est une classe de problèmes algorithmiques de décision (dont la réponse est "oui" ou "non") et pour lesquels on peut rapidement vérifier que la réponse est "oui" à l'aide d'un certificat" [4].*

3 Coloration

3.1 Explications de l'algorithme

L'algorithme consiste à parcourir tous les sommets du graphe en vérifiant qu'ils soient bien colorables avec la couleur choisie. Si le sommet est colorable avec cette couleur particulière, on le colore et on passe au prochain sommet. S'il ne l'est pas, on tente avec la couleur

suivante. Si aucune couleur ne peut être utilisée, on revient sur le dernier sommet coloré (backtracking) en essayant de lui assigner la couleur suivante et on répète l'opération de vérification de couleur. Si on arrive à colorer tous les sommets, alors le graphe est bien colorable avec maximum k couleurs.
Ci-dessous, le pseudo-code du backtracking :

Algorithm 1 Coloration

```
Require: integer indiceSommet
Ensure: boolean isColored
 1: procedure BACKTRACKING(indiceSommet)
 2:     isColored ← false
 3:     if (indiceSommet == nombreSommet) then isColored = true
 4:     else
 5:         indiceCouleur ← 1
 6:         while (indiceCouleur <= nombreChromatique and not(isColored)) do
 7:             if test pour colorer le sommet avec cette couleur then
 8:                 colorer le sommet
 9:                 if Backtracking(indiceSommet+1) then isColored = true
10:                 else retirer la couleur du sommet
11:                 end if
12:             end if
13:             indiceCouleur ← indiceCouleur + 1
14:         end while
15:     end if
16:     return isColored
17: end procedure
```

L'algorithme 1 s'inspire du pseudo-code de backtracking de Shivangi Jain [5].

La condition de la ligne 7 est détaillée ci-dessous et permet de s'assurer que le sommet est colorable avec une certaine couleur. Si un sommet adjacent a la même couleur que la couleur à tester, alors on ne peut pas colorer le sommet de cette couleur.
Ci-dessous, le pseudo-code de cette vérification :

Algorithm 2 Coloration

```
Require: integer indiceSommet, integer couleur
Ensure: boolean isColorable
 1: procedure ISCOLORABLE(indiceSommet, couleur)
 2:     isColorable ← true
 3:     i ← 0
 4:     while (i < nbrSommets and isColorable) do
 5:         if (∃ une arête allant du sommet indiceSommet jusqu'au sommet i) and (couleur == couleur du
    sommet i) then
 6:             isColorable ← false
 7:         end if
 8:         i ← i + 1
 9:     end while
10:     return isColorable
11: end procedure
```

3.2 Réponses aux différentes questions

3.2.1 Donner une borne supérieure, sous forme de $O(.)$, de la complexité de votre algorithme.

En posant k comme étant le paramètre à vérifier et n comme le nombre de sommets. Pour vérifier si un sommet est colorable, on doit au maximum parcourir tous les sommets du graphe ce qui correspond à une complexité linéaire par rapport au nombre de sommets. Pour notre parcours du graphe, comme le backtracking est utilisé et que l'on va donc potentiellement tenter toutes les possibilités, le nombre de couleurs à vérifier sera exponentiel par rapport au nombre de sommets, ce qui correspond à une complexité en $O(k^n)$. En cumulant les deux complexités, la complexité maximale obtenue est donc de $O(nk^n)$ [5]. C'est une complexité exponentielle donc inutilisable en pratique. Avec de très grands nombres, on favorisera un algorithme glouton (voir fin de la section 3.3 pour de plus amples explications), à défaut d'avoir un résultat optimal.

3.2.2 Quelle est la complexité du problème dans le cas particulier où $k = 2$?

Dans ce cas précis, on cherche à trouver si le graphe est biparti. Cette fois-ci, le backtracking n'est pas nécessaire. Il suffit de faire un parcours en profondeur (DFS) et pour chaque sommet non-coloré on attribue une couleur. Les sommets adjacents prennent la couleur inverse. Si un sommet déjà coloré doit être coloré par la couleur inverse alors le graphe n'est pas biparti. Si on arrive à colorer tous les sommets, sans avoir le cas cité juste avant, alors le graphe est biparti. La complexité est linéaire par rapport aux sommets et arêtes, $O(n+a)$ où n correspond au nombre de sommets et a au nombre d'arêtes [6][7].

3.2.3 Pensez-vous qu'il existe un algorithme pour résoudre ce problème dans le cas particulier où $k = 3$, et dont la complexité est bornée supérieurement par un polynôme de degré constant en la taille du graphe (par exemple $O(n^2)$, ou $O(n^5)$, où n est le nombre de sommets du graphe) ?

Ce problème est dit **NP**-complet (**N**ondeterministic **P**olynomial-time complete), ce qui signifie, je cite : *"qu'on ne connaît pas, à l'heure actuelle, d'algorithme efficace en complexité polynomiale. Néanmoins, il n'a pas été démontré que ces algorithmes n'existaient pas. Les problèmes NP-complet sont les problèmes "les plus difficiles" dans NP"* [4].

Pour prouver que ce problème est NP-complet, il y a deux méthodes possibles [6], je cite : *"soit par la définition, on donnera alors une réduction en temps polynomial pour chaque problème Q de NP vers P; soit on prend un autre problème NP-complet Q, et on montre que Q peut être réduit vers P en temps polynomial."*. C'est cette deuxième méthode qui a été utilisée. Grâce au problème *3-SAT* qui a déjà été prouvé *NP-complet*, on peut montrer que le problème de Coloration est *NP-complet*. Pour information, voici une preuve complète [8].

Si je devais donner mon avis sur ces problèmes, en toute subjectivité, alors je dirai que, par optimisme compulsif, il existe probablement un algorithme en temps polynomial. Cependant, cet avis ne se base sur aucun fondement mathématique, juste une foi inébranlable en la créativité, l'ingéniosité et l'inventivité humaine.

3.3 Description synthétique du programme et commentaires

J'ai décidé de créer moi-même ma structure de graphe en créant une classe *GrapheC* qui est essentiellement composée d'une matrice d'adjacence, c'est-à-dire une matrice binaire m de taille $n \times n$ où n est le nombre de sommets du graphe. Le sommet m_{ij} est à 1 s'il existe une arête allant du sommet i à j et à 0 si elle n'existe pas. Vu que le graphe n'est pas orienté, la matrice d'adjacence est symétrique par rapport à la diagonale, ce qui signifie que $m_{ij} = m_{ji}$. Dans mon programme, cette matrice est représentée par un tableau de tableau. La complexité en temps pour créer cette matrice est de $O(n^2)$ où n est le nombre de sommets. La complexité en espace est elle aussi $O(n^2)$ [9]. Les seules opérations permises sur cette structure sont l'ajout d'un sommet et un simple accesseur pour chaque sommet qui renvoie si le sommet est lié à un autre sommet donné.
J'ai choisi cette représentation de graphe pour deux raisons [10] :
a) l'ajout d'une arête dans une matrice d'adjacence se fait $O(1)$;
b) la consultation se fait elle aussi en $O(1)$.
Un exemple pour illustrer et pour que ce soit plus clair, le fichier graphe1.txt serait représenté comme ceci : [[0,1,1],[1,0,1],[1,1,0]].

Un tableau d'Integer, qui a la taille du nombre de sommets, est utilisé pour stocker la couleur de chaque sommet.

L'algorithme fonctionne avec des nombres entiers positifs uniquement.

Concernant l'algorithme, j'ai choisi de faire du backtracking pour être sûr et certain d'avoir une solution optimale. J'avais d'abord implémenté un algorithme glouton mais celui-ci dépendait du sommet sur lequel on commençait. J'ai donc préféré avoir une réponse optimale mais plus lente, qu'une réponse rapide mais parfois faussée.

L'algorithme glouton consiste à itérer sur chaque sommet, et à essayer de le colorer avec la première couleur, en vérifiant avant, que les sommets adjacents ne soient pas colorés de cette même couleur. Si aucun sommet adjacent n'est coloré de cette couleur particulière, alors on le colore de celle-ci. Si un sommet adjacent est déjà coloré de cette couleur, on essaye avec la couleur suivante et on répète l'opération jusqu'à soit, arriver à colorer le sommet ou alors arriver à ne plus avoir de couleur disponible, ce qui signifierait que le graphe ne peut pas être coloré avec le nombre de couleurs passé en paramètre. Un exemple typique qui montre la faiblesse de cet algorithme est le *Crown graph* [11].

3.4 Exemples d'exécutions

Exécutions sur différents fichiers donnés en exemple. La première colonne correspond à l'indice du sommet et la deuxième correspond à sa couleur :
- java coloration graphe1.txt 3
1 1
2 2
3 3

- java coloration graphe2.txt 3
1 1
2 2
3 3
4 2
5 2

- java coloration graphe3.txt 3
1 1
2 2
3 1
4 2
5 2
6 3
7 1
8 3
9 3
10 1

J'ai créé d'autres fichiers pour faire des tests supplémentaires sur d'autres cas, vous les trouverez dans le dossier *tests* du fichier zip.

4 Ordonnancement

4.1 Explications de l'algorithme

Au préalable, on trie toutes les durées de tâches par ordre croissant par rapport à leur date de début.

Pour chaque tâche, on va vérifier si sa date de début est bien postérieure ou égale à la date de la machine, qui représente la date de fin de la dernière tâche insérée sur cette machine. Si c'est le cas, on insère la tâche sur cette machine et on met à jour la date de la machine par la date de fin de la tâche insérée. Si ce n'est pas le cas, on passe à la machine suivante et on réitère le procédé. S'il n'y a plus de machines disponibles et que la tâche n'est toujours pas insérée, on peut conclure qu'il n'y a pas de solution avec le nombre de machine donné.

Ci-dessous le pseudo-code de la méthode qui effectue le travail expliqué précédemment :

Algorithm 3 Ordonnancement

```
Require: void
Ensure: void
 1: procedure findOrdo
 2:     for i ← 1 jusqu'à nombreTâches do
 3:         inserted ← false
 4:         j ← 0
 5:         while (j < nombreMachine and not(inserted)) do
 6:             if (listeMachine[j] <= début de la tâche i) then
 7:                 placer le numéro de la machine sur la tâche
 8:                 inserted ← true
 9:                 mise a jour du temps de la dernière tâche effectué sur la machine i
10:             end if
11:             j ← j + 1
12:         end while
13:         if not(inserted) then
14:             exit
15:         end if
16:     end for
17: end procedure
```

4.2 Réponses aux différentes questions

4.2.1 Montrer que ce problème est un cas particulier du problème précédent, mais n'est pas équivalent au problème précédent.

Les deux problèmes présentés sont des problèmes de coloration de graphe car dans les deux cas, chaque sommet a besoin d'une information supplémentaire qui contient sa couleur (dans le cas de l'ordonnancement, la couleur est le numéro de la machine). Cependant, la manière de résoudre ces problèmes, de manière optimale, est *différente* pour chacun. Pour *Coloration*, le backtracking est une solution de sécurité car toutes les possibilités vont, potentiellement, être essayées. Pour *Ordonnancement*, un algorithme glouton est suffisant car les dates de début et de fin de tâches sont fixées.

Une différence non-négligeable est que pour *Ordonnancement*, les sommets sont moins dépendants entre eux. Le mot *dépendance* ici signifie que les décisions futures n'influeront pas sur leur couleur. Pour *Coloration*, si un sommet essaye toutes les couleurs disponibles sans succès, il faut *backtracker* pour essayer de débloquer la situation en changeant la couleur du sommet précédent, tandis que pour *Ordonnancement*, à partir du moment où une tâche est placée sur une machine elle est immuable. Cette différence vient du fait que, pour le problème *Ordonnancement*, les dates des tâches sont fixées (un début et une fin).

Une autre différence est que pour *Ordonnancement*, les sommets sont liés entre eux uniquement quand leurs intervalles de temps s'intersectent. Dans ce cas-là, ils auront donc besoin de machines différentes pour être insérés correctement, comme il aurait fallu des couleurs différentes pour des sommets adjacents pour le problème de *Coloration*.

4.2.2 Donner une borne supérieure, sous forme de $O(.)$, de la complexité de votre algorithme.

Avant toute chose, il faut trier les données reçues par ordre croissant sur les dates de début. Pour faire ceci, Array.sort(arr) a comme complexité maximale $O(nlog(n))$ [12]. En posant n correspondant au nombre de tâches et m le nombre de machines. Comme on va itérer une fois sur toutes les tâches, cela se fait en $O(n)$. Cependant, pendant cette itération, à chaque tâche, on doit tenter au maximum d'insérer la tâche sur m machines différentes. Donc les deux complexités s'imbriquent pour donner une complexité maximum de $O(mn)$. Comme le tri est effectué à part, la complexité finale est : $O(nlog(n)) + O(mn) = O(mn)$.

4.3 Description synthétique du programme et commentaires

Pour ce programme-ci, j'ai créé une classe GrapheO (qui n'est pas la même que GrapheC car il n'y a pas les mêmes besoins). Le graphe est représenté comme un tableau de tableau. Le tableau emboîté est composé de 4 éléments. En posant i comme étant l'indice de la tâche dans le tableau, le premier élément du sous-tableau correspond au temps de début de tâche i, le deuxième au temps de fin de la tâche i, le troisième au numéro de la machine attribué à cette tâche i et le quatrième est le numéro de cette tâche i qui servira pour l'affichage d'un potentiel résultat. La complexité en temps pour créer ce tableau 2D est de $O(4n)$ où n correspond au nombre de tâches [13]. La complexité en espace est elle aussi de $O(4n)$, donc deux complexités linéaires. Il y a deux types d'opérations que l'on peut réaliser sur cette structure :
a) la modification de données déjà existantes qui se fait en temps constant [14] ;
b) l'accès à un élément qui se fait également en temps constant [14].
Un exemple pour illustrer et pour que ce soit plus clair, le fichier ord1.txt serait représenté comme ceci : [[0.0, 1.0, 0.0, 0.0],[0.5, 2.0, 0.0, 1.0],[1.0, 2.0, 0.0, 2.0]].

Un tableau de Float, qui a la taille du nombre de machines, est utilisé pour stocker la date de fin de la dernière tâche insérée sur chaque machine i.

L'algorithme est un algorithme de type glouton et fonctionne avec des nombres uniquement.

Avant d'implémenter l'algorithme, j'ai considéré deux approches pour représenter mon graphe. La première représentation est une matrice d'adjacence avec un lien explicite entre deux tâches. Deux tâches seraient connectées si leurs intervalles s'intersectent tandis que pour la première approche, les arêtes sont implicites. Cependant, créer cette matrice d'adjacence demanderait en espace et en temps une complexité en $O(n^2)$. Comme celle-ci est plus lourde que celle de la première approche expliquée ci-dessus, et que la complexité de l'algorithme n'aurait pas changé, cette deuxième approche n'a pas été choisie.

Remarque 1 *L'algorithme glouton est valable uniquement car des dates de début et de fin nous sont données. Si elles étaient remplacées par des durées, le problème deviendrait plus compliqué.*

4.4 Exemples d'exécutions

Exécutions sur différents fichiers donnés en exemple. La première colonne correspond au numéro de la tâche et la deuxième correspond au numéro de la machine sur laquelle elle est assignée :
- java ordonnancement ord1.txt 2
1 1
2 2
3 1

- java ordonnancement ord2.txt 3
1 1
2 2
3 3
4 1
5 2
6 3
7 1
8 2
9 3

J'ai créé d'autres fichiers pour faire des tests supplémentaires, vous les trouverez dans le dossier *tests* du fichier zip.

5 Conclusion

Pour commencer par le problème *Coloration*, nous avons deux choix possibles pour résoudre le problème. Soit avoir un algorithme très lent avec une complexité exponentielle mais qui garantit que le nombre chromatique trouvé est optimal par rapport au nombre donné en paramètre grâce au backtracking, soit utiliser un algorithme glouton qui ne pourra pas certifier que le nombre chromatique renvoyé est le plus petit possible mais il aura une complexité polynomiale. Si jamais quelqu'un arrivait à trouver un algorithme optimal en temps polynomial pour ce problème, cela bouleverserait le monde de l'informatique !

Cependant pour le problème *Ordonnancement*, un algorithme glouton est tout à fait valide et optimal même s'il s'agit également d'un problème de coloration de graphe.

Les problèmes NP-complets sont, et seront encore probablement pendant longtemps, un sujet de discussion chez les informaticiens et les mathématiciens.

6 Références

[1] Chaturvedi Aayush. geeksforgeeks(2021). Backtracking : Introduction. Reperé le 07/04/2021 à https://www.geeksforgeeks.org/backtracking-introduction/

[2] Algorithme glouton. (1 février 2021.) Dans Wikipédia, l'encyclopédie libre. Repéré le 09/04/2021 à https://fr.wikipedia.org/wiki/Algorithme_glouton

[3] geeksforgeeks. geeksforgeeks(2019). Greedy Algorithms. Reperé le 07/04/2021 à https://www.geeksforgeeks.org/greedy-algorithms/

[4] Cardinal Jean. (2020). Foire aux questions du cours d'Algorithmique 2. Réponse à la question Problèmes NP-complets le 31 mars 2021. Université Libre de Bruxelles.

[5] Shivangi Jain. includehelp(2018). Graph coloring problem's solution using backtracking algorithm. Repéré le 07/04/2021 à https://www.includehelp.com/algorithms/graph-coloring-problem-solution-using-backtracking-algorithm.aspx

[6] Assistants du cours d'Algorithmique 2. (2020). Parcours de graphes 37-38. Correctif du cours. Séance 6. Université Libre de Bruxelles.

[7] Samuel Fiorini. (2020). Graphes et arbres, 7.5 Colorations de graphes 65-66. Syllabus de Mathématiques discrètes. Université Libre de Bruxelles.

[8] /. Enseignement polytechnique(2013). Quelques problèmes NP-complets. 7-9. Repéré le 16/04/2021 à https://www.enseignement.polytechnique.fr/informatique/INF423/uploads/Main/chap12-good.pdf

[9] baeldung. baeldung(2020). Time and Space Complexity of Adjacency Matrix and List. Repéré le 15/04/2021 à https://www.baeldung.com/cs/adjacency-matrix-list-complexity

[10] Alina. stackoverflow(2017). Graph as adjacency matrix time complexity. Repéré le 15/04/2021 à https://stackoverflow.com/questions/46548954/graph-as-adjacency-matrix-time-complexity

[11] Crown graph. (17 avril 2020.) Dans Wikipédia, l'encyclopédie libre. Repéré le 26/04/2021 à https://en.wikipedia.org/wiki/Crown_graph

[12] aminy. stackoverflow(2014). Will Arrays.sort() increase time complexity and space time complexity?. Repéré le 28/04/2021 à https://stackoverflow.com/questions/22571586/will-arrays-sort-increase-time-complexity-and-space-time-complexity

[13] Java Hermit. stackoverflow(2017). What is the time complexity of declaring a 2d array. Repéré le 15/04/2021 à https://stackoverflow.com/questions/45788319/what-is-the-time-complexity-of-declaring-a-2d-array

[14] Nikhil. stackoverflow(2014). Array Access Complexity. Repéré le 15/04/2021 à https://stackoverflow.com/questions/20615908/array-access-complexity