complexité et calculabilité

Bah Elhadj amadou et abdelamine MEHDAOUI (GA4-3)

Octobre 2019

Contents

1 Réponses aux questions théoriques **3**
- 1.1 Définition du problème Distance commune . . . 3
  - 1.1.1 Montrons que le problème des distances communes est dans NP . . . 3
- 1.2 Cas particulier acyclique . . . 4
  - 1.2.1 **Algorithme polynomial pour le problème Distance Commune.** . . . 4
  - 1.2.2 **Le temps de calcul de l'algorithme donnée:** . . . 4
  - 1.2.3 **Pourquoi l'algorithme ne fonctionne pas dans le cas général:** . . . 4
- 1.3 Réduction de distance commune vers SAT . . . 5
  - 1.3.1 La valeur maximale du paramètre k qu'on doit considérer . . . 5
  - 1.3.2 La formule propositionnelle qui est satisfiable si et seulement si chacun des graphes en entrée dispose d'un chemin simple et valide de longueur k est composé de plusieurs sous formules qu'on peut rassembler . . . 5
  - 1.3.3 Démontrons que si il existe un chemin simple et valide de longueur k dans chacun des graphes, alors la formule est satisfiable (Distance commune $\implies$ $FormuleSAT$) . . . 5
- 1.4 Génération des formules pour chaque valeur de k = 1, 2, . . , et les tester une par une . . . 6
  - 1.4.1 Description d'un algorithme permettant d'obtenir la solution au problème Distances Commnunes . . . 6
  - 1.4.2 Pouvez-vous avoir moins de variables sur l'ensemble de l'algorithme qu'à la question précédente ? Pourquoi ? . . . 6
  - 1.4.3 Le gain pour cette approche: . . . 6
  - 1.4.4 Justifions que cette approche est équivalente à la précédente . . . 6

2 Rapport **7**
- 2.1 Choix d'implémentation . . . 7
  - 2.1.1 Optimisations réalisées . . . 7
  - 2.1.2 Longueur maximale du chemin à chercher . . . 7
- 2.2 Fonctionnalités implémentées . . . 7
- 2.3 Fonctionnalités non implémentées . . . 8
- 2.4 Bugs ou limites . . . 8
- 2.5 Modifications apportées . . . 8
- 2.6 Difficultés rencontrés . . . 9
- 2.7 Manuel d'utilisation du programme . . . 9

1 Réponses aux questions théoriques

1.1 Définition du problème Distance commune

1.1.1 Montrons que le problème des distances communes est dans NP

- **Certificat:** Le certificat pour le problème des distances communes est un nombre entier n.
  La taille de ce certificat est log(n) (—certificat—=log(n) est donc meilleur qu'une complexité polynomiale)

- **Vérificateur V(G1,...,Gn, certificat):**

  - **Entrée:** Un ensemble de graphes G1,G2,...GN et le certificat (un nombre entier)
  - **Sortie** Renvoie Vrai s'il existe un entier n tel que chacun des graphes G1, . . . , GN possède un chemin simple et valide de longueur n et Faux sinon .
  - **Description:** Le vérificateur effectue un parcours en profondeur (ou un parcours en largeur) de chaque graphe à partir du sommet source jusqu'à une profondeur égale au certificat et vérifie si l'un des chemins obtenus est acceptant (si il existe au moins un chemin simple et valide).
    Pour vérifier qu'un chemin est valide il suffit de vérifier que le premier sommet du chemin est le sommet source du graphe et le dernier sommet du chemin est le final du graphe. Comme le parcours en largeur ou en profondeur ne visite chaque sommet qu'une seule fois, alors le chemin obtenu est simple.
  - **La Complexité:**
    - **Vérificateur:** Parcourir tous les graphes reçus en entrés est d'une complexité O(N) où N est les nombre de graphes. La complexité d'un parcours en largeur ou d'un parcours en profondeur est de O(m+n) (par exemple selon une représentation en matrice liste d'adjacence).
      D'où la complexité du vérificateur est de O(N * (n+m)) et est donc polynomiale.
    - **La taille du certificat:** est en temps logarithmique (log(n)). Cette complexité est donc acceptable vu qu'elle est inférieur à une complexité polynomiale.

1.2 Cas particulier acyclique

1.2.1 Algorithme polynomial pour le problème Distance Commune.

Algorithm 1 Parcours_en_Profondeur(G, nodeSource, nodeFinal, pathLenth)

```
for tout sommet v de G do
|  couleur(v) = Blanc
end
return visiterPP(nodeSource, nodeFinal, 0, pathLength)
```

Algorithm 2 visiterPP(s, Nodefinal, time, pathLength)

```
time = time + 1
couleur(s) = Gris
if time InferieurOuEgal pathlength then
|  for all w in Adj(s) do
|  |  if couleur(w) == Blanc then
|  |  |  if w==NodeFinal && time == pathLength then
|  |  |  |  return true
|  |  |  end
|  |  |  VisiterPP(w, time, pathLength)
|  |  end
|  end
end
```

Algorithm 3 Solution_Distance_commune(G1, ...,GN)

```
min_sommet = nombreSommets(G0) for chaque Graphe G1 GN do
|  if nombreSommets(Gi) InferieurOuEgal min_sommet then
|  |  min_sommet = nombreSommet(Gi)
|  end
end
for k=0; k InferieurOuEgal min_sommet; k++ do
|  for i=0; ijN; i++ do
|  |  result = Parcours_en_profondeur(Gi, NodeSource, k)
|  |  if result == false then
|  |  |  break
|  |  end
|  end
|  if i==N then
|  |  return true;
|  end
end
return false
```

1.2.2 Le temps de calcul de l'algorithme donnée:

Effectuer le parcours en profondeur de chaque graphe est d'une complexité O(N * (n+)). Dans le pire des cas l'algorithme *solution_distance_commune(...)* test pour les k longueurs de chemins possible (chemin de longueur1, ... , chemin de longueurk).
D'où une compléxité de O(K * N * (m+n)).

1.2.3 Pourquoi l'algorithme ne fonctionne pas dans le cas général:

la détection d'un cycle n'est pas prévu dans l'algorithme précédente, si on considère le cas ou les graphes possèdent des cycles l'algorithme peut tourner à l'infini.

1.3 Réduction de distance commune vers SAT

1.3.1 La valeur maximale du paramètre k qu'on doit considérer

Cette valeur est n-1 où n est le nombre de sommets du graphe qui a le plus petit nombre de sommets. On choisit n-1 et non n parce que vu que le chemin cherché ne doit pas contenir de cycles, alors dans le pire des cas il passe par chaque sommet du graphe une et une seule fois.

1.3.2 La formule propositionnelle qui est satisfiable si et seulement si chacun des graphes en entrée dispose d'un chemin simple et valide de longueur k est composé de plusieurs sous formules qu'on peut rassembler

- Formule pour un chemin :

– $\wp 1 = \bigwedge_{\substack{i=0,\\ i<=N}} (\bigwedge_{\substack{j=0,\\ j<=k-1}} \bigwedge_{u \in V(Gi)} \overline{x_{i,j,k,u}} \vee \bigvee_{v \in Voisin(u)} x_{i,j+1,k,v})$

  Cette formule exprime le fait que si un sommet est vraie à une position j, alors il doit exister au moins un de ses voisins qui est vraie à la position j+1 (sauf pour le dernier sommet). Elle exprime donc la notion d'arête entre les sommets du chemin.

- Formule pour un chemin valide:

– $\wp 2 = \bigwedge_{\substack{i=0,\\ i<=N}} (x_{i,0,k,s} \wedge x_{i,k,k,t} \wedge \bigwedge_{u!=t} \overline{x_{i,k,k,u}})$

  Cette formule exprime le fait que le sommet source est vraie à la position 0, le sommet cible est vraie à la position k (dernière position).

- Formule pour chemin simple:

– $\wp 3 = \bigwedge_{\substack{i=0,\\ i<=N}} (\bigwedge_{\substack{j=0,\\ j<=k}} (\bigvee_{u \in V(Gi)} (x_{i,j,k,u} \wedge \bigwedge_{v!=u} \overline{x_{i,j,k,v}} \wedge \bigwedge_{\substack{j'=0,\\ j'!=j, j'<=K}} \overline{x_{i,j',k,u}})))$

  Cette formule exprime le fait que pour toute position dans le chemin:

  – Il existe au moins un sommet qui est vraie à cette position.
  – Et si un sommet est vraie à une position, aucun autre sommet ne peut être vraie à cette position.
  – Et Si un sommet est vraie à une position, il ne peut être vraie sur aucune autre position (d'où le nom de chemin simple)

- Formule pour Distance Commune

– $\wp DistanceCommune = \wp 1 \wedge \wp 2 \wedge \wp 3$

1.3.3 Démontrons que si il existe un chemin simple et valide de longueur k dans chacun des graphes, alors la formule est satisfiable (Distance commune $\implies$ $FormuleSAT$)

- **Démonstration pour la formule pour un chemin:**

- **Démonstration pour la formule d'un chemin valide:**
  Comme le sommet "s" est le sommet source et le sommet "t" le sommet de destination alors la position du "s" et de "t" seront respectivement "0" et "k", donc quelque soit un sommet "u" telque "u" diffèrent du sommet de destination "t" alors variable X_i,k,k,u = vrai, c'est a dire elle se trouve dans la k_ieme position d'un chemin valide de longueur k, on conclure que la formule d'un chemin valide est satisfaisable.

- **Démonstration pour la formule d'un chemin simple:**
  Pour tout sommet "u" et pour chaque position "j", le sommet "u" est à la J_ième position d'un chemin simple et valide de longueur "k", ce qui implique que la variable X_i,j,k,u est vrai, d'autre part pour tout autre sommet "v" différente de "u", ni ce sommet ni "u" peuvent être à la Jième et j' ieme position respectivement du même chemin simple et valide de longueur k, ce qui fait que la variable X_i,j,k,v et X_i,j',k,v sont fausse donc la négation de chacune des variables est vrai, on conclure que la formule de chemin simple est valide.

1.4 Génération des formules pour chaque valeur de k = 1, 2, . . , et les tester une par une

1.4.1 Description d'un algorithme permettant d'obtenir la solution au problème Distances Commnunes

Un algorithme utilisant cette approche consisterait à faire une boulce sur le nombre de longueur de chemins possibles (0 à n-1) et générer une formule SAT lors de chaque tour de boucle. Si cette formule générée est satisfiable alors on peut dire que le graphe admet un chemin de longueur correspondant à l'indice courant dans la boucle. Si elle n'est pas satisfiable, passer au prochain tour de boucle. A la fin de la boucle s'il n'y a pas eu une formule satisfiable, alors on pourra dire que le graphe n'admet aucun chemin simple et valide.

1.4.2 Pouvez-vous avoir moins de variables sur l'ensemble de l'algorithme qu'à la question précédente ? Pourquoi ?

Non il n'est pas possible d'avoir moins de variables sur l'ensemble de l'algorithme. Cela est dû au fait que sur l'ensemble de l'algorithme, on va générer une formule pour chaque k. Donc au final on a pas moins de variables qu'à la formule précédente. Le seule cas ou on aurait moins de variables c'est si on s'arrêtait dès qu'une formule générer est valide et donc de ne pas tester les autres formules restantes.

1.4.3 Le gain pour cette approche:

Le gain pour cette approche est si l'on veut tester les longueurs de chemins possible par profondeur. Il est plus rapide pour le solveur de resoudre une formule pour une longueur de chemin possible que pour toutes les longueurs de chemins possibles. Ainsi on a gain de temps dû à la taille des différentes formules générés pour chaque K.

1.4.4 Justifions que cette approche est équivalente à la précédente

Cette approche est équivalente à la précédente. La différence entre les deux est la forme des formules. Avec cette approche nous avons une formule F pour une longueur de chemin donnée alors que avec l'autre approche, nous avons une union de formules où chaque formule correspond à une longueur de chemin possible.

2 Rapport

Dans ce devoir de complexité et de calculabilité, il nous est demandé de réaliser dans un premier temps une réduction du problème *Distances communes vers SAT*. Ensuite, il nous est demandé d'implémenter cette réduction du problème *Distances communes* vers SAT à fin d'utiliser un SAT solveur pour résoudre ce problème. Pour cela il fallait implémenter un ensemble de fonctions dont une fonction *main* les utilisant pour illustrer le fonctionnement de notre programme avec un système d'options.

2.1 Choix d'implémentation

Pour réaliser cette partie 2 du devoir, nous avons dès le début penser à réaliser une version optimisée dans le but de permettre un traitement rapide non seulement pour la construction des formules mais aussi pour l'ensemble des traitements (parcours des graphes, taille de la formule à construire ...). Cette optimisation est principalement ce qui caractérise notre implémentation.

2.1.1 Optimisations réalisées

Comme mentionné précédemment, nous avons réalisé une optimisation qui nous permet d'avoir une exécution beaucoup plus rapide que l'exécutable qui nous a été fournie (voir ci dessous pour plus d'informations sur cette optimisation).

- optimizeAndMakeFormula
  Dans cette fonction nous avons réaliser l'optimisation de la réduction. En effet, sans optimisation, notre formule tient en compte de tous les sommets du graphe à chaque traitement sur une position quelconque du chemin. Or cela n'est pas nécessaire car pour tout sommet à une distance d, les seules sommets candidats à la distance d+1 sont les successeurs de ces sommets. Par exemple le sommet source étant à la position 0, les sommets qui peuvent être traités à la position 1 sont uniquement les voisins de s. Il ne sert donc à rien de traiter tous les sommets du graphe à chaque position.
  Il est à noter que cette optimisation peut être activer ou désactiver selon le choix de l'utilisateur. Cependant cela ne se fait pas via une option en ligne de commande (mais plutôt une variable globale "OPTIMIZE" dans solving.c) vu que nous ne pouvions pas ajouter de fichier sources ou modifier les headers (fichiers.h).

- Avantages de l'optimisation
  Cette optimisation permet d'avoir des formules plus courtes et limite donc le temps de traitement du solveur sur la formule générée. Cet gain de temps est constaté surtout lors de l'exécution des graphes avec un grand nombre sommets (comme ceux du répertoire "graphs/instances_prime_length").

- Limites de l'optimisation
  La principale limite de cette optimisation est l'utilisation de mémoire. En effet, avant de générer la formule, il est nécessaire d'effectuer un traitement pour limiter les variables qui seront générées pour la formule. Pour cela on utilise une file pour traiter les sommets selon leurs distances, et un tableau à deux dimensions (les lignes correspondes aux positions et les colonnes correspondent aux différents sommets possibles à ces positions). Donc plus il y a d'arêtes et plus la file sera grande.
  Il est possible d'aller encore plus vite en optimisant cette file (ne pas ajouter la même variable deux fois) et par la même occasion donc limiter la consommation en mémoire.

2.1.2 Longueur maximale du chemin à chercher

La longueur maximale du chemin à chercher est n-1 (où n est le nombre de sommets du graphe qui a le plus petit nombre de sommets). Nous avons considéré qu'il n'était pas nécessaire de chercher jusqu'à un sommet de longueur n vu que le sommet qu'on cherche ne contient pas de cycle et donc contient au plus n sommets.

2.2 Fonctionnalités implémentées

Nous avons réalisé toutes les fonctionnalités demandées dans le devoir ainsi que l'ensemble des améliorations (voir la sections ci dessous pour les fonctionnalités non implémentées).

- getNodeVariable:
  Cette fonction consiste à implémenter une formule consistant en une variable caractérisée par: le numéro du graphe, la position du sommet, la longueur du chemin dans lequel appartient le sommet et en fin le

sommet. Elle s'écrit sous la forme Xi,j,k,q. Nous précisons que dans notre implémentation, pour que deux variables soit égales il faut qu'elles portent **exactement le même nom**. Par exemple X1,2,4,b est identique à X1,2,4,b mais pas à X1,3,4,b.

- graphToPathFormula:
  graphToPathFormula construit une formule SAT pour une longueur de chemin donnée en parcourant tous les graphes. Cette formule utilise la fonction d'optimisation (optimizeAndMakeFormula) pour générer cette formule. Il suffit de faire un "ET LOGIQUE" entre la formule générée pour chaque graphe.

- graphToFullFormula:
  Comme décrit dans la documentation du projet, cette fonction permet de construire une formule qui est satisfiable si et seulement si tous les graphes en entrés possèdent un chemin acceptant de même longueur. Pour cella depuis cette fonction on utilise la fonction précédente pour construire une formule qui exprime:
  Tous les graphes possèdent un chemin de longueur 0 ou tous les graphes possèdent un chemin de longueur1 ou ... ou tous les graphes possèdent un chemin de longueur k (où k est la longueur maximale à chercher pour un chemin)

- printPathFromModel:
  Cette fonction permet d'afficher le chemin obtenu dans chaque graphe en se basant sur le modèle obtenu à partir de la formule SAT. Pour un graphe donnée le chemin correspondant est l'ensemble des sommets qui sont valués à vrai. Et pour obtenir la position d'un sommet dans le chemin, on teste pour quelle position le sommet est vrai. Car dans le devoir il est mentionné qu'une variable Xi,j,k,q est vraie si et seulement si 'q' est le j ème sommet d'un chemin de longueur k dans le graphe Gi en partant du sommet Si.

- CreateDotFromModel:
  Dans cette fonction, nous avons implémenté la fonctionnalité permettant de générer les fichier .dot contenant les solutions des chemins trouvés. Il est important de signaler que ces fichiers sont générés dans le répertoire "sol" et que donc il doit exister.

- GetSolutionLengthFromModel:
  Cette fonction permet d'obtenir la longueur du chemin trouvé en commun pour les graphes à partir du modèle issu de la formule SAT correspondante.

2.3 Fonctionnalités non implémentées

Nous avons implémenté toutes les fonctionnalités (l'ensemble des fonctions et des système d'options de l'exécutable). Cependant l'amélioration 2 n'a été fait que partiellement. Ainsi nous avons implémenté la fonction *GetSolutionLengthFromModel* mais nous n'avons pas terminé l'amélioration (faire en sorte que le modèle ne contienne que les chemins témoins de la solution).

2.4 Bugs ou limites

- A l'état actuel de notre programme, nous rencontrons un seul bug qui ne se produit que lorsque le programme est exécuté sur ma machine en locale. L'erreur générée est du "stack smashing" lorsque le graphe est assez grand. Cependant le programme s'exécute normalement sur les machines du crémi. L'erreur a été constaté assez tardivement vu que nous avons implémenté tout le projet sur les machines du crémi. Cependant nous comptons bien corrigé ce problème avant la soutenance du projet.

2.5 Modifications apportées

Après l'extension de la date de rendu, nous avons effectué un ensemble de modifications qui portent principalement sur la partie du code et sur la réduction.

- **Modification de la réduction**
  Nous avons changé la réduction que nous avions mis pour le premier rendu non seulement elle était incomplète mais aussi il fallait faire différents parcours du graphe pour trouver tous les chemins. Cependant avec la nouvelle réduction, on ne parcours même pas le graphe, il suffit juste de générer exactement la réduction mis en place (sauf si on active le mode optimisation qui fait un traitement avant la génération des formules).

- **Modification de l'implémentation** Après avoir changé la réduction, nous avons aussi modifié une partie du code notamment les fonctions de génération des formules. En plus nous avons ajouté une fonction qui permet d'effectuer des optimisations. Avant on utilisait un parcours en largeur mais maintenant nous ne parcourrons plus le graphe pour générer ces formules.

2.6 Difficultés rencontrés

- La principale difficulté rencontrés dans ce devoir est la *totale* dépendance de la deuxième partie à la première. En effet pour pouvoir implémenter la deuxième partie il faut avoir la formule SAT (et qu'elle soit correcte). Pour un début, nous avions mis en place une formule sauf qu'on s'est rendu compte lors de la deuxième partie qu'elle ne permettait pas de répondre aux besoins attendus. Notre réduction a été implémenté en parallèle avec la deuxième partie car dans ce cas on pouvait tester la formule à chaque fois qu'on effectue un changement.

- L'autre difficulté rencontrée est la différence entre le fait de générer une formule SAT globale depuis l'entrée et le fait de générer une formule pour chaque valeur de k (k=1, 2, 3, ...). Nous considérons que la formule globale n'est rien d'autre qu'un "OU LOGIQUE" entre les formules pour k=0, pour k=1, ..., pour K=n.

- Nous avons eu aussi des difficultés dans la rédaction des preuves. Malgré avoir fait une réduction qui fonctionne parfaitement bien, et aussi avoir implémentée cette formule, nous avons eu du mal dans la rédaction de preuves.

2.7 Manuel d'utilisation du programme

Pour obtenir de l'aide sur l'utilisation du programme, il faut utilisé l'option -h qui décrit tous les systèmes d'options que nous avons mis en place (ses systèmes d'options sont identiques à ceux de l'exécutable qui nous a été fourni).