Complexité et Calculabilité

Devoir maison (projet)

Emmanoe DELAR, Najeebullah Ahmadzai

03 Novembre 2019

université de BORDEAUX

Table des matières

1 Definition du probleme Distance commune

Un graphe pointé est un tuple G = (V, E, s, t) tel que (V, E) est un graphe orienté, $s \in V$ est le sommet source et $t \in V$ est le sommet destination. Un chemin est une sequence (non-vide) de sommets v_1 , . . . ,v_n t.q.$(vi, vi + 1) \in E$ pour tout 0 ≤ i < n (la longueur du chemin est n-1). Il est dit valide si v_1 = s et v_n = t. Il est dit simple si il ne contient qu'au plus une seule fois chaque sommet.
Le probleme Distance commune est :

Entrée : $G_1 = (V_1, E_1, s_1 , t_1), . . . , G_N = (V_N , E_N , s_N , t_N)$ graphes pointés.
Question : Est-ce qu'il existe un entier n tel que chacun des graphes G_1 , . . . , G_N possède un chemin simple et valide de longueur n ?

2 Complexité du problème

2.1 Vérificateur

Le vérificateur pour Distance commune prend en entrée la paire (ensemble de graphes G_N et une liste de sous-ensemble C_N) tq les éléments des C_i sont des sommets rangés dans l'orde de parcour, en partant de s_i, pour atteindre t_i et pour chaque graphe G_i vérifie si le chemin C_i associé est bien un chemin simple et valide de longueur n

Entree : $G_1 = (V_1, E_1, s_1 , t_1), . . . , G_N = (V_N , E_N , s_N , t_N)$ graphes pointés.
Certificat : $C_1 \subseteq V_1$,...,$C_N \subseteq V_N$

Pour chaque C_i vérifier que :
- C_i est non-vide
- Le premier élement de $C_i == s_i$
- Pour chaque élément de la liste (le dernier exclue), (C[i],C[i+1]) $\in E_i$
- Chaque élément ne réapparait pas dans la liste
- La taille de $C_i == n$

2.2 Complexité du vérificateur

Pour chaque C_i vérifier : $O(N)$ tel que N est égale au nombre de graphes

C_i est non-vide : $O(1)$

Le premier élement de $C_i == s_i$: $O(1)$

Pour chaque élément de la liste (le dernier exclue), (C[i],C[i+1]) $\in$E$_i$: $O(n)$ * $O(m)$ tel que m est le nombre d'arêtes dans l'ensemble E_i et n le nombre de sommet de l'ensemble V_i. m est borné par m<=$\frac{(n-1)(n)}{2}$<n^2

Preuve Soit V l'ensemble des sommets d'un graphe G orienté acyclique contenant n sommets. Soit v_1 un sommet de V, alors le nombre de sommets auxquelles il peut être relié (le nombre d'arcs) vaut n-1. Soit v_2 un deuxième sommet de V. v_2 peut se connecter à tous les sommets mis à part lui même et v_1 (car acyclique). Et ainsi de suite par récurrence, le prochain sommet peut se connecter à tous les noeuds sauf les précédents et lui même. Et le dernier sommet ne peut se connecter à aucun autre sommet. La somme de tous la taille de tous les arcs est alors :

$$(n-1) + (n-2) + ... + 1 + 0 = \frac{(n-1)(n)}{2} \tag{1}$$

Chaque élément ne réapparait pas dans la liste $O(n^2)$

La taille de C_i **== n** $O(1)$

La complexité du vérificateur est alors

$$O(N * ((n * m) + (n^2))) \tag{2}$$

Le vérificateur est en temps polynomial.

2.3 Taille du certificat

Le certificat est composé de N listes de taille au plus n_i. La première liste C_1 a pour taille n_1 tel que n_1 est borné par le nombre de sommet inclut dans V_1. C'est aussi vrai pour toutes les liste C_i :

$$\textstyle\sum_{i=1}^{N} \mathbf{n}_i \tag{3}$$

Les témoins des instances positives sont de taille polynomial.

On a donné un algorithme polynomial (vérificateur) qui accepte un certificat de taille polynomial tel que le vérificateur **vérifie en temps polynomial** que ce certificat est une preuve qu'une est instance de Distance commune est positive ou pas. Distance commune appartient donc à **NP**.

3 Cas particulier acyclique

3.1 Algorithme polynomial pour le probleme DC

On utilise l'algrithme de parcour en profondeur modifié. Pour chaque graphe, on crée une matrice array contenant les tailles de tous les chemins que notre algorithme récurssif, parcour_en_largeurr_path a trouvé pour chaque graphe. Cette matrice est convertie en matrice de booléen "size_found" qui pour chaque graphe si un chemin de taille "indice" existe alors cette matrice va valoir 1 à l'indice (sinon 0). Ensuite en sommant pour chaque matrice en fonction de l'indice qu'on cherche, si cette somme vaut N c'est à dire que l'indice apparait dans les N graphes et donc tous les graphes ont un chemin de taille "indice".

```
Data: G_1 = (V_1, E_1, s_1 , t_1 ), . . . , G_N = (V_N , E_N , s_N , t_N )
Result: Vrai ou Faux
global array[N] ;
gloabal res ;
global max_computed_path_size ;
for G ∈Data do
    nb_path = [] ;
    parcour_en_largeur_path(G, s(G), t(G), nb_path) :
    array.append(nb_path)
end
size_found[N][max_computed_path_size]
for i :0 -> len(array) do
    for j :0 -> len(array[i]) do
        size_found[i][array[i][j] ] = 1
    end
end
res = [0] * max_computed_path_size
for i :0 -> N do
    for j :0 -> max_computed_path_size do
        res[j] += k[i][j]
    end
end
for i :0 -> max_computed_path_size do
    if res[j] == N then
        return Vrai;
    end
end
return False;
```

Voici l'implémentation de l'algorithme récurssif parcour_en_largeur_path :

```
Data: Graph graph, Vertex start, Vertex goal, Array nb_path
queue = [(start, [start])]
i = 0
while queue : do
|   (vertex, path) = queue.pop(0) ;
|   for next ∈ V :  do
|   |   if next == goal : then
|   |   |   nb_path.append(len(path)+1)
|   |   end
|   |   else
|   |   |   queue.append((next, path + [next]))
|   |   end
|   end
end
```

3.1.1 Temps d'exéxution

```
while queue : do                                O(|V|
|   (vertex, path) = queue.pop(0) ;             O(1)
|   for next ∈ V :  do                          O(|E|)
|   |   if next == goal : then                  O(1)
|   |   |   nb_path.append(len(path)+1)         O(1)
|   |   end
|   |   else
|   |   |   queue.append((next, path + [next])) O(1)
|   |   end
|   end
end
```

FIGURE 1 – parcour_en_largeur_path

Complexité $= O(|V| + |E|)$

```
global array[N] ;
gloabal res ;
global max_computed_path_size ;
for G ∈Data do
    nb_path = [] ;                                        O(1)
    parcour_en_largeur_path(G, start, goal, nb_path) ;    O(n+m)
    array.append(nb_path)                                 O(1)
end
size_found[N][max_computed_path_size]
for i :0 -> len(array) do                                 O(len(array))
    for j :0 -> len(array[i]) do                          O(len(array(i)))
        size_found[i][array[i][j] ] = 1
    end
end
res = [0] * max_computed_path_size
for i :0 -> N do                                          O(N)
    for j :0 -> max_computed_path_size do                 O(max_computed_size)
        res[j] += k[i][j]                                 O(1)
    end
end
for i :0 -> max_computed_path_size do                     O(max_computed_size)
    if res[j] == N then                                   O(1)
        return Vrai;
    end
end
return False;
```

Figure 2 – Algorithme pour DC

3.2 Cas général

L'algorithme précédent ne fonctionne pas dans le cas général car l'algorithme tournera à l'infini si un sommet a lui même comme voisiin à cause du while du parcour en prorfondeur.

4 Reduction de Distance commune vers SAT

4.1 Une seule formule a partir de l'entrée

4.1.1 Valeur maximale du paramètre k

Pour trouver un chemin simple et valide, on peut calculer par induction sur k ce chemin. La **valeur maximal** du paramêtre k qu'on doit considérer est **n** le nombre total de sommet de G car un sommet n'apparait qu'une fois dans un chemin.

4.1.2 Formule propositionnelle

Pour construire la formule qui est satisfiable si et seulement si chacun des graphes en entrée dispose d'un chemin simple et valide de longueur k,nous avons construit deux sous-formules.

4.1.3 Première sous-formule ϕ_1

La première sous-formule doit être satisfiable si et seulement si chacun des graphes dispose d'un chemin simple $\iff$ quelque soit le sommet q, q ne peut pas être le j-ème et le l-ème sommet du chemin simple de longueur k du graphe.

$$\bigwedge_{i=1}^{N} \bigwedge_{q \in V_i} \bigwedge_{j=0}^{k} \bigwedge_{l=0, l \neq j}^{k} \neg(x_{i,j,k,q} \wedge x_{i,l,k,q}) \tag{4}$$

4.1.4 Deuxième sous-formule ϕ_2

Et la deuxième sous-formule s'évalue à vrai si et seulement si chacun des graphes dispose d'un chemin valide de longueur k $\iff$ le sommet q (respectivement r) est le premier (resp. dernier) sommet du chemin simple de longueur k du graphe pointé G :

$$\bigwedge_{i=1}^{N} x_{i,1,k,q} \wedge x_{i,k,k,r} \tag{5}$$

pour $(q \in V_i | \text{q=s}_i), (\text{r} \in V_i | \text{r=t}_i)$

4.1.5 Première implication

Si il existe un chemin simple et valide de longueur k dans chacun des graphes, ça implique que pour chaque graphe G_i, le premier sommet de son chemin P_i est s_i et le dernier sommet t_i ce que vérifie (5). Ce chemin est de taille `k` car ce chemin ne contient au plus qu'une seul fois chaque sommet, il est donc simple, ce que vérifie (4) et comme t_i existe à la position k (5), ce chemin est donc de taile k.

4.1.6 Deuxième implication

Si il existe une valuation satisfaisant notre formule cela implique que :

(5) pour tout graphe G_i il existe un sommet q (resp. r) tel que q (resp. r) est le 1er (resp. le k-ième ou dernier) sommet du chemin simple de longuer k du graphe pointé G_i. Cela correspond aux sommets s_i et t_i du graphe G_i.

Et le sommet v_j ne peut pas être le j-ème et le l-ème sommet du chemin simple de longuer k du graphe G_i (4). Effectivement, comme c'est un chemin simple, alors c'est une séquence (non vide) de sommets, donc des aretes du graphe G_i, qui s'écrit sous la forme : $(v_1,...,v_k)$ tel que $v_1 = s_i$, $v_k = t_i$ et $(v_j,v_{j+1}) \in \mathrm{E}_i$, $1 \leq j \leq k$.

4.1.7 Réduction

À toute instance X de Distance Commune on associe une instance Ẍ de SAT tel que :

$$X satisfaisable \iff \ddot{X} satisfaisable \tag{6}$$

On en déduit alors la réduction :

$$(4) \wedge (5) =$$

$$\bigwedge_{i=1}^{N} \bigwedge_{q \in V_i} \bigwedge_{j=0}^{k} \bigwedge_{l=0, l \neq j}^{k} \neg(x_{i,j,k,q} \wedge x_{i,l,k,q}) \wedge \bigwedge_{i=1}^{N} x_{i,1,k,q} \wedge x_{i,k,k,r} \tag{7}$$

(4). Pour chacun des graphes G_i, pour chacun des sommets q du graphe, le sommet q ne peut pas apparaitre plusieurs fois dans le chemin simple de longuer k du graphe G_i

(5). Pour chacun des graphes G_i, il existe deux sommets q $= s_i$, r $= t_i$ tel que le sommet q (resp. r) est le premier (resp. le k-ième ou le dernier) sommet du chemin simple de longueur k du graphe pointé G_i.

4.2 Plusieurs formules

4.2.1 Algorithme Distance Commune par induction sur k

```
1 DistanceCommune(G_1,G_2,...,G_N,k)
2     foreach g in Input do
3         if !(BFS_mod(s(g),k))
4             return False
5         fi
6     return True
7 }
```

```
1 BFS_mod(s,k)
2         Set vis[v] <- false for all v in V_i
3         Set all L_i for i=1 to k to 0;
4         L_0 = s;
5         vis[s] <- True;
6         for i:0 to k-1 do
7             if L_i = 0
8                 return False;
9             fi
10            While L_i do
11                u<- L_i.pop()
12                for each x in Voisin(u) do
13                    if vis[x] is false then
14                    vis[x] <- True
15                    L_i+1.insert(x)
16                done
17            done
18        done
19        if t(g) in L_i
20            return True
21        return False
22
23 }
```

À la ligne 7, la sous-formule (5) vérifie que s_i soit le sommet à la première position du chemin. À la ligne 12, c'est la sous-formule (4) qui vérifie que chaque sommet soit unique dans le chemin qu'on parcourt et à la ligne 15, la sous-formule vérifie que t_i soit le dernier sommet du chemin trouvé.

4.2.2 Gain

Avec cette approche algorithmique, on aura moins de variable à générer car contrairement à la formule précédente, si on trouve un cas qui ne satisfait pas une condition alors on arrête l'évaluation immédiatement grâce à la méthode `return`.

Dans le meilleur des cas, cette deuxième approche nous procure un gain de temps qui est le temps d'exécution de notre algorithme sur juste un seul graphe si l'algorithme venait à s'arréter à l'évaluation de la première condition (si le chemin est vide à la ligne 7 de l'algorithme BFS_mod et le temps de l'évaluation de cette même condition par la formule précédente, pour les N graphes. La formule précédente devra évaluer la sous-formule (2) au moins N fois.

4.2.3 Analyse des formules obtenues

À la formule obtenue à la question précédente correspond les bouts de codes de l'algorithme. La formule s'écrit :

$$\bigwedge_{i=1}^{N} \bigwedge_{j=0}^{k} \bigwedge_{l=0, l \neq j}^{k} \neg(x_{i,j,k,q} \wedge x_{i,l,k,q}) \wedge \bigwedge_{i=1}^{N} x_{i,1,k,q} \wedge x_{i,k,k,r} \tag{8}$$

On a :

$\bigwedge_{i=1}^{N}$ qui correspond à la boucle principal `ligne 1 - algorithme DistanceCommune`

$\bigwedge_{j=0}^{k} \bigwedge_{l=0,l\neq j}^{k}$ sont les indices de parcour par induction sur k et représente la boucle for `ligne 6 - Algorithme BFS_mod`.

$\neg(x_{i,j,k,q} \wedge x_{i,l,k,q})$ cette formule est équivalent aux itérations et évaluations de la boucle for `ligne 7 - Algorithme BFS_mod`

$\wedge \bigwedge_{i=1}^{N} x_{i,1,k,q} \wedge x_{i,k,k,r}$ représente la fin de la portée de la boucle for précédente et la vérification que le chemin obtenu est valide à partir du `if ligne 19 - Algorithme BFS_mod`.