Université de Mons
Faculté des sciences
Département d'Informatique

Title

Rapport de stage d'initiation à la recherche

Professeur :
Hadrien Mélot
Superviseur :
Sébastien Bonte

Auteur :
William Karpinski

Année académique 2024-2025

Table des matières

1 Introduction et Problématique

Ce rapport s'inscrit dans le cadre d'un stage d'initiation à la recherche dans le service d'algorithmie dirigé par M.Hadrien Mélot sous la supervisation de Sébastien Bonte.

Tout au long ce stage, j'ai été ammené à travailler sur plusieurs invariants de graphes, et plus spécifiquement sur le treewidth de graphes simple non-orienté. Tous les invariants ont été calculés sur des graphes d'ordre 2 à 10. J'ai également été ammené à comparer l'efficacité des algorithmes dans deux langages de programmations, le Rust et le Python.

2 Notions de base

Dans ce rapport, nous allons définir G, un graphe simple non-orienté comme étant composé de V, un ensemble de sommets, et E, un ensemble d'arêtes. On définit également l'ordre de G, comme étant le nombre de sommets dans V et sa taille, comme le nombre d'arêtes dans E.

En théorie des graphes, une décomposition en arbre d'un graphe G est un arbre, tel que chaque noeud de celui-ci contient un sous-ensemble de V et possède les propriétés suivantes. Soient T, une décomposition en arbre de G, $X_1, ..., X_t$, les sous-ensembles de V dans les noeuds de T, nous avons que :

- Tous les sommets de G doivent être dans au moins un noeud de T.
- Si X_i et X_j contiennent un somment v, alors tous les noeuds du chemein entre X_i et X_j dans T contiennent v.
- Pour toutes les arêtes (v,w) dans E, il existe au moins un noeud de T contenant v et w. À noter qu'un noeud de T peut contenir deux sommets de V sans qu'il y aie d'arête entre eux.

Un graphe est dit cordal si pour chaque cycle ayant 4 sommets ou plus dans celui-ci, il existe une arête liant deux sommets non adjacents du cycle.

Une clique est un sous ensemble d'aretes de V tel que les sommets du sous-ensemble sont connectés deux à deux. Une clique maximal d'un graphe G est une clique tel qu'il n'en nexiste pas d'autres dans G avec plus de sommets. Le nombre de sommets dans une clique maximale est notée $\omega(G)$

3 Tree Width

On appelle la largeur, ou width, d'une décomposition en arbre d'un graphe comme étant le nombre de sommets contenu dans le noeud de T en contenant le plus, moins un. La largeur d'arborescence ou treewidth sera la largeur

minimum parmis toutes les décompositions en arbre possibles de G. Intuitivement, on peut dire que le treewidth d'un graphe définit à quel point celui-ci est proche d'un arbre, le treewidth d'un arbre étant 1 et celui d'un graphe complètement connecté est n-1, n l'ordre de G.

Le treewidth peut également être caractérisé d'autres manières qui seront expliquées ultérieurement si utilisés dans les algoritmes.

Le problème de calcul du treewidth est un problème NP-Difficile, la problématique ici va donc se concentrer sur comment calculer efficacement celui-ci.

3.1 Méthode Naïve

L'algorithme utilisé dans cette section utillise une autre définition de treewidth, celle-ci étant que le treewidth de G est la taille de la plus grande clique du graph cordal contenant G ayant le plus petit $\omega(G)$, moins un.

La marche à suivre ici serait de vouloir lister tous les graphes cordaux contennant G et calculer à chaque fois la taille de sa plus grande clique. Ensuite nous prendrions le minimum parmis ces tailles et nous obtiendrons le treewidth en le soustrayant de un.

Ici l'algorithme s'occupera différement la première étape, en effet nous allons calculer la taille de la clique maximale en même temps que de créer le graphe cordal. De plus nous allons calculer les tailles de cliques moins un à partir des voisins d'un noeud, le moins un dans le calcul du treewdith est implicite tout au long des calculs.

L'algorithme fonctionne comme suit, considérons que les sommets de G soient numérotés de 0 à n-1, n étant l'ordre de G et nous voulons calculer les graphes cordaux pouvant être créés à partir de G.

Nous allons donc itérer sur les sommets de G, notons i le sommet courant, et allons explorer tout les voisins de i ayant un numéro strictement supérieur à ce dernier, notons j le voisin courant.

Si le voisin j est le premier voisin exploré parmis tous ceux de i,notons le first, nous allons ajouter une arêtes entre lui et tous les voisins de j.

Cela permettra 2 choses, tout d'abord cela la création du graphe cordal car tous les cycles partant de i auront une corde entre le voisin j de i et first. S'il n'existait pas d'aretes entre j et first, alors nous devons l'ajouter car s'il exister un cycle contenant i,j et first de taille supérieure à 4 il faudra que first et j possède une corde pour que le graphe devienne cordal.

La seconde chose que cela permet de calculer la taille de la clique maximal du graphe cordal. En ajoutant les aretes nous comptons les voisins de i et lorsque i sera un sommet auquel on a ajouter une ou plusieurs aretes celui ci comptera les voisins supplémentaires et donc la tailles des cliques maximales

pouvant etre générés avec les cordes à disposition. En retirant le noeud i du compte on obtient bien la taille de la clique maximale moins 1 ce qui si est minimal parmis tous les graphes cordaux correspond bien au treewidth.

Pour obtenir le treewidth nous devons exploré tous les graphes cordaux possibles, nous allons réattribuer les nombres associés à chaque sommet et aisni change l'ordre dexploration. De ce fait, les aretes ajoutées entre les graphes seront différentes et ainsi nous pourront explorer toutes les possibilités en prenant toutes les permutations possibles des noeuds. Il nous suffit de prendre le minimum des tailles maxiamles de cliques pour tous les graphes cordaux trouver et nous obtiendrons le treewisdth.

l'algo est en temps factoriel parce que on enumere toutes les possibilité sur tous les sommets

L'ajout d'un lower bound peut aider l'algorithme dans pas mal de cas, stopper l'algorithme quand on a trouvé une taille de clqiue egale au lowerbound permet de ne pas itérer sur toutes les permutations Cependant il n'est pas sur qu'on atteigne ce lowerbound, ce qui se produit fréquemment sur les graphes de grand o au plusrdre donc l'algorithme reste très peu efficace, bien que sur les graphes d'ordres inférieurs à 7 le lower bound permet de diviser par 4 le temps d'execution.

Pseudo-code ;

The `algorithm` environment is a *float*, like `table` and `figure`, so you can add float placement modifiers `[hbt!]` after `\begin{algorithm}` if necessary.

3.2 Notions Complémentaires

Les prochains algorithmes sont bien plus efficaces que l'algorithme naïf mais utilisent des propriétés plus complexes qui vont être expliquées dans cette section. Toutes les propriétés énoncés proviennent de la référence suivante, et sont prouvé dans celle-ci. Ici nous ne feront que les énoncés et donner une intuition pour pouvoir comprendre les algorithmes.

Le problème de calcul du treeWidth peut etre formulé comme un probleme d'ordonnancement de ses sommets. Sachant qu'un ordonnancement π de G(V,E) est une bijection π :V ->1,2..|V| notons $\pi_{(<,v)}$ lensemble des sommets qui apparaissent avant le sommet v dans l'ordonnancement π.

Pour un graphe G et un ordonnancement π, si π à la propriété qie lensemble pour chaque i de π tel que

$\{\pi^{-1}(j)|\{\pi^{-1}(i), \pi^{-1}(j)\} \in E \wedge j > i\}$ forme une clique,

alors π défini une triangulation H de G et donc utiliser la definition du treewidth avec les graphes cordaux

Cependant pour eviter d'utiliser explicitement les triangulations dans nos algorithmes, sont definis dans le papier : Pour un certain ordonnancement

Algorithm 1 Naive Algorithm Tree Width

Require: G(V,E) : Graphe avec V comme ensemble de sommets et E comme ensemble d'arêtes
Ensure: Tree Width du graphe

```
n ← V.size
tw ← n ;
lw ← lowerbound
Pour permutations dans V faire
    C(W, F) ← copiedeGavecsommetsrenomms
    nMaxDeg ← 0
    Pour i allant de 0 à n faire nDeg ← 0
        Pour j allant de i+1 à n faire
            Si (i,j) dans F alors
                nDeg ← nDeg + 1
                Si j > i+1 alors
                    Ajouter arêtes (i+1,j) dans F
                Fin Si
            Fin Si
        Fin Pour
        Si nDeg > nMaxDeg alors
            nMaxDeg ← nDeg
        Fin Si
    Fin Pour
    Si nMaxDeg < tw alors
        tw ← nMaxDeg
    Fin Si
Fin Pour
    return tw
```

π :

Soit P(v,w) qui est vrai ssi il y a un chemin de v à w qui n'utilise que des sommets de $\pi_{(<,w)} \cap \pi_{(<,v)}$, donc tjrs vrai si v = w ou (v,w) $\subseteq$ E.

Soit R(v) qui est defini comme le nombre de sommets w tq (π(w) > π(v)) and P(v,w)

Il est prouvé que tw(G) est <= k ssi il existe un ordonnancement π tq R(v) <= k pour tout v

En complément nous avons que pour π, R(v) = |Q(i(<,v),v)| avec Q(S,v) le nombre de sommets w appartenant à V-S-v tel qu'il existe un chemin de v à w utilisant les uniquement les sommets de S,v et w

Aussi noous avons TW(S) = min (parmis tous les i ordonnancement de

V) max (parmis tous les v dans S) |Q(i(<,v),v)| S étant un subset de V Tel que prouvé dans le papier, on a la propriété que tw(G) = TW(V)

3.3 Meilleur Algorithme

L'algorithme ici présenté se base sur la propriété suivante :

$$TW(S) = \min v \in Smax\{TW(S - \{v\}), |Q(S - \{v\}, v)|\} \quad (1)$$

Sachant que TW(V) = tw(G) nous allons vouloir explorer les subsets S $\subseteq$ V de taille 1 et calculer le TW(S). Puis nous ajouterons des sommets v de V aux subsets, de sorte à obtenir tous les subsets de taille 2 et nous utiliserons la propriété pour trouver TW(S + {v}), pour tous les subsets S. Nous réitérons cela jusqu'a la taille n ce qui nous donnera TW(V) et donc tw(G).

L'algorithme présenté fonctionne comme énoncé ci dessus, soit i la taille courante des subsets. Les différents subsets S des rangs i-1 sont stockés dans une liste, tw_prev,avec TW(S) et qui à la fin de l'algorithme stockera, si elle existe, la paire (V,TW(V)). Cette liste est par defaut initialisée avec la paire $(\emptyset, -\infty)$, donc le subset de taille 0 et une valeur suffisament pour que les TW(S) du rang 1 ne privilégie jamais le TW($\emptyset$). Lorsque nous allons ajouter les différents subsets de taille i, il se peut qu'on obtiennet plusieurs fois le meme, il faut donc vérifier avant d'insérer TW(S) dans la liste s'il existe déja un élémént contenant le subset S et si cest le cas prendre comme TW(S) le maximum entre la valeur présente et celle qu'on veut ajouter

Pour améliorer le temps de calcul si q > a un upperbound fixer au préalable nous nallons pas calculer la valeur de TW(S) pour le subset courant cette valeur ne pouvant qu'augmenter

Lorsque nous avons fini d'itérer si nous avons dans la liste TW(V) on retourne sa valeur sinon on retourne l'upperbound fixé au préalable

L'algorithme est de complexité $O(2^n)$ en temps et en espace

Pseudo-code :

3.4 Algorithme Récursif

TWR(L,S) = min max Q(L U i(<,v),v), TWR([],V) = TW[V] = tw(G) => Calcule de TWR sur [],V pour trouver tw(G)

Soient L et S des subsets de V disjoints

Definissons TWR(L,S) = min (parmis tous les i ordonnancement de V) max (parmis tous les v dans S) |Q((L||i(<,v)),v)|

Algorithm 2 Improved Algorithm

Require: G(V,E) : Graphe avec V comme ensemble de sommets et E comme ensemble d'arêtes
Ensure: Tree Width du graphe

```
n ← V.size
upperbound ← n − 1 ;
tw_pred ← [(∅, −∞)]
Pour _ allant de 0 à n faire
    tw_current ← []
    Pour (S, tw) ⊆ tw_pred faire
        Pour v ∈ V − S faire
            r ← max{tw, |Q(S, v)|}
            Si r < up alors
                Si (§ ∪ {v}, t) ⊆ tw_current alors
                    Set t ← min{t, r}
                Sinon
                    Insérer (S ∪ {v}, r) dans tw_current
                Fin Si
            Fin Si
        Fin Pour
    Fin Pour
    Set tw_pred = tw_current
Fin Pour
Si (V, x) ∈ tw_pred alors
    return x
Sinon
    return up
Fin Si
    return tw
```

Ici nous avons que l'ensemble des sommets dans L est avant S dans l'ordonnancement et les sommets qui ne sont ni dans L ni dans S sont apres S.

Il s'agit d'un algorithme recursif avec :

Cas de base S.len() == 1 : Si nous avons ce cas, nous a que S = v et nous savons que TWR(L,v) = Q(L,v) (L contient toutes les valeurs a gauche de v car tous les sommets ni dans L ni dans S sont apres S comme énoncé précédemment) nous pouvons donc dans ce casa retourner la valeur de Q(L,v) Cas général ; Il est montré dans le papier que TWR(L,S)= min (parmi tout les subsets S' de taille k de S) maxTWR(L,S'),TWR(L || S', S-S')

Nous allons donc utiliser cette propriété pour recursivement diminuer la taille de S pour arriver à une taille de 1 et obtenir notre cas de base et ainsi retourner TWR.

Pour obtenir tw(G) via cet algorithme il faut l'appeler sur TW([],V) Car ici nous avons TW([],V) = TW(V) = tw(G)

Lalgorithme est en $O(4^n)$ en temps et polynomial en espace

Pseudo-code :

Algorithm 3 Recursive

Require: G(V,E) : Graphe avec V comme ensemble de sommets et E comme ensemble d'arêtes
L : subsets de V
S : subsets de V
lowerbound
Ensure: Tree Width du graphe
$n \leftarrow V.size$
Si S.size == 1 **alors** :
return |Q(L,S)|
Fin Si
$tw \leftarrow \infty$
Pour $Sub \subseteq S$ avec S.size == n/2 **faire**
$v1 \leftarrow Recursive(G, L, Sub)$
$v2 \leftarrow Recursive(G, L \cup Sub, S - Sub)$
Set $tw = min\{tw, max\{v1, v2\}\}$
Fin Pour
return tw

3.5 Algorithme Récursif Amélioré

dit si tw(G) au plus = k se base sur les composantes connexes du graphes et si leur tw <= k

4 Autres Invariants

Variance des degres,Proximity/Remoteness,Girth

5 Comparaison Python/Rust

Rust mieux mais assez bizarrement pas pour proxi et remote

6 Conclusion