

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

<b>1</b>	<b>Introduction et Problématique</b>	<b>2</b>
<b>2</b>	<b>Notions de base</b>	<b>2</b>
<b>3</b>	<b>Tree Width</b>	<b>2</b>
3.1	Méthode Naïve . . . . .	3
3.2	Notions Complémentaires . . . . .	4
3.3	Meilleur Algorithme . . . . .	4
3.4	Algorithme Récursif . . . . .	4
3.5	Algorithme Récursif Amélioré . . . . .	4
<b>4</b>	<b>Autres Invariants</b>	<b>5</b>
<b>5</b>	<b>Comparaison Python/Rust</b>	<b>5</b>
<b>6</b>	<b>Conclusion</b>	<b>5</b>

# 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 supervision de Sébastien Bonte.

Tout au long de ce stage, j'ai été amené 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 amené 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  $G$ ,  $X_1, \dots, X_t$ , 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 sommet  $v$ , alors tous les noeuds du chemin 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 ait 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'arêtes de  $V$  tel que les sommets du sous-ensemble sont connectés deux à deux. Une clique maximale d'un graphe  $G$  est une clique tel qu'il n'en existe 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 parmi 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 algorithmes.

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 utilise 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 contenant  $G$  et calculer à chaque fois la taille de sa plus grande clique. Ensuite nous prendrions le minimum parmi ces tailles et nous obtiendrions le treewidth en le soustrayant de un.

Ici l'algorithme s'occupera différemment 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 treewidth 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 parmi 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'arêtes entre  $j$  et first, alors nous devons l'ajouter car s'il existe 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 arêtes nous comptons les voisins de  $i$  et lorsque  $i$  sera un sommet auquel on a ajouter une ou plusieurs arêtes celui ci comptera les voisins supplémentaires et donc la tailles des cliques maximales

pouvant être 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 parmi tous les graphes cordaux correspond bien au treewidth.

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

L'algorithme est en temps factoriel parce que on énumère toutes les possibilités 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 clique égale au lower-bound permet de ne pas itérer sur toutes les permutations. Cependant il n'est pas sûr qu'on atteigne ce lowerbound, ce qui se produit fréquemment sur les graphes de grand  $n$  au plus 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'exécution.

Pseudo-code ; #TODO

## 3.2 Notions Complémentaires

Prochains algorithmes viennent d'un papier qui introduit quelques notions supplémentaires. Problème d'ordre linéaire  $TW()$   $Q$ ,  $R$ ,  $P$

## 3.3 Meilleur Algorithme

$$TW(S) = \min \max_{v \in S} TW(S-v, |Q(S-v, v)|) \text{ Basé là dessus}$$

## 3.4 Algorithme Récursif

$$TWR(L, S) = \min \max_{i \in L} Q(L \cup \{i\}, S), \quad TWR([], V) = TW[V] = tw(G)$$

=> Calculer de  $TWR$  sur  $[], V$  pour trouver  $tw(G)$

## 3.5 Algorithme Récursif Amélioré

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

## **4    Autres Invariants**

Variance des degres, Proximity/Remoteness, Girth

## **5    Comparaison Python/Rust**

Rust mieux mais assez bizarrement pas pour prox et remote

## **6    Conclusion**