Master Informatique

Parcours Informatique & Interactions

Étude d'algorithmes de décomposition de graphes

Alternant
Theo Qui

Maître d'apprentissage
Nicolas Nisse

Tuteur académique
Arnaud Malapert


COATI


1er septembre 2020

1 Remerciements

Je tiens à offrir mes remerciements à toutes les personnes ayant rendu possible et qui ont contribué au succès de cette année d'alternance.

Tout d'abord, j'adresse mes remerciements à mon maître d'apprentissage, M. Nicolas Nisse, ainsi qu'à M. David Coudert, directeur de l'équipe COATI, pour leurs efforts ayant permis cette alternance, et pour avoir su me conseiller et me guider dans mes travaux.

J'offre aussi mes remerciements à M. Philippe Renevier, responsable de l'alternance en Master 2 Informatique & Interactions, pour son implication et sa réactivité lors du processus de recrutement.

D'autre part, je remercie mon tuteur académique M. Arnaud Malapert pour sa présence, ses conseils et avis sur l'avancée de mon travail m'ayant permis de mieux définir le contenu du présent rapport.

Je tiens à remercier toute l'équipe COATI pour leur accueil chaleureux et l'agréable ambiance de travail ressentie tout au long de cette année.

Enfin, je remercie M. Nisse de m'avoir assisté et conseillé durant la rédaction de ce rapport.

Table des matières

Résumé Nous nous intéressons ici aux décompositions arborescentes de graphes. Savoir si un graphe a une treelength d'au plus 2 est un problème NP-Complet, c'est pourquoi nous cherchons à approximer ce paramètre.

Pour cela, nous présentons 4 algorithmes de la littérature permettant d'obtenir des décompositions de graphes et d'en approximer la treelength : lex-M et Bfs-layering qui sont des 3-approximation de la treelength, uncle-trees pour qui nous ne connaissons pas de rapport d'approximation, et disk-tree qui est une 6-approximation mais que l'on suppose être en réalité une 2-approximation.

Pour certains algorithmes, nous proposons une légère amélioration ne changeant pas le rapport d'approximation, mais permettant tout de même d'obtenir des décomposition de length inférieure dans certains cas.

Après les avoir implémentés, nous lançons des tests sur plusieurs types de graphes afin de comparer leurs temps d'exécution, et la length des résultats.

Enfin, nous présentons quelques résultats de nos travaux en cours qui, nous espérons, nous permettront de prouver que disk-tree est bien une 2-approximation de la treelength.

2 Introduction

Décomposer des graphes est une méthode de type "diviser pour régner" permettant d'accélérer la résolution de certains problèmes dans des graphes. Il peut s'agir de problèmes "simples" pour lesquels nous connaissons des algorithmes efficaces (en temps polynomial), ou de problèmes pour lesquels nous ne connaissons aucun algorithme efficace (e.g. NP-Complets). Ici, nous nous concentrons sur les décompositions dites arborescentes ; informellement, il s'agit d'un *mapping* d'un graphe sur un arbre dont les sommets, appelés *sacs*, sont des sous-ensembles des sommets du graphe. Cependant, pour utiliser une décomposition arborescente de graphe, nous avons besoin de méthodes pour l'obtenir, et il est nécessaire que le graphe en question admette une "bonne" décomposition arborescente. C'est donc l'objet de cette étude : étudier, concevoir, implémenter des algorithmes permettant de décomposer (lorsque c'est possible) efficacement des graphes, c'est-à-dire de minimiser le nombre de sommets dans le plus grand sac (*width* d'une décomposition), ou la plus grande distance entre deux sommets d'un même sac (*length* d'une décomposition), ces termes seront définis formellement par la suite (section 3).

Intuitivement, la treewidth d'un graphe est la mesure d'à quel point un graphe est proche d'être un arbre. Par exemple, on peut obtenir facilement une tree-decomposition de petite width (et de petite length) dans le cas d'un arbre :

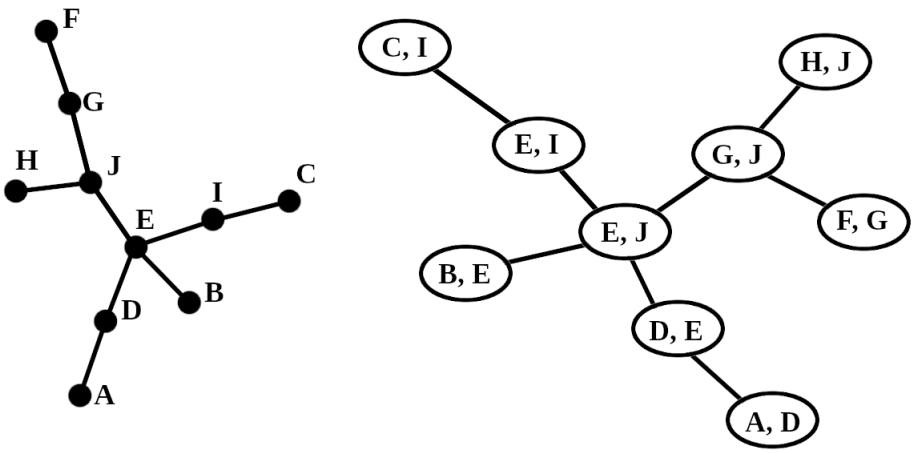

FIGURE 1 – Un arbre et une tree-decomposition de cet arbre

Spécifiquement dans le cas des arbres (fig. 1), la width et la length de la décomposition sont toutes les deux égales à 1. Dans beaucoup d'autres cas obtenir une décomposition de petite width ou length est plus compliqué.

Par exemple dans le cas des cycles, il est établi que la treewidth est égale à 2, et Dourisboure et Gavoille montrent que l'on peut obtenir une tree-decomposition de length égale à $\lceil \frac{|V(G)|}{3} \rceil$ et qu'il s'agit de la length minimum d'un cycle [8] :

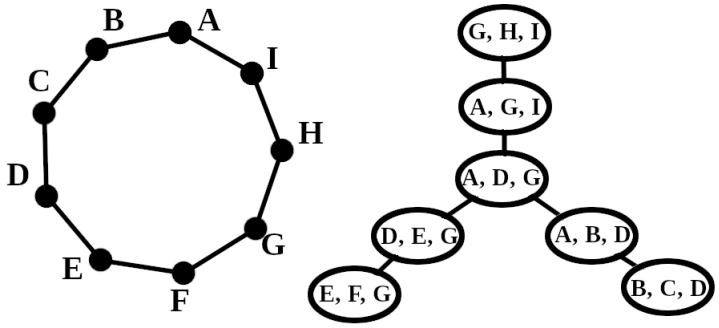

FIGURE 2 – Un cycle et une de ses tree-decompositions possibles

Autrement dit, plus le cycle est grand, plus sa treelength est grande alors que sa treewidth est toujours égale à 2.

Inversement, dans le cas des cliques, il est très facile d'obtenir une décomposition de length 1, mais la treewidth reste toujours égale au nombre de sommets moins 1 :

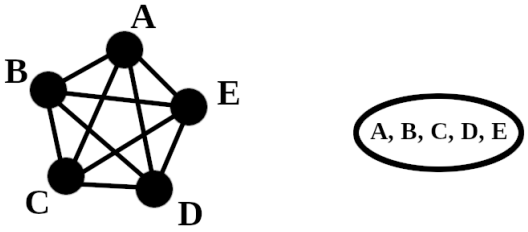

FIGURE 3 – Une clique et une de ses tree-decompositions

D'autre part, dans le cas des grilles (fig. 4), la width et la length d'une décomposition sont toujours égales à la taille du plus petit côté et augmentent donc avec le nombre de sommets.

Dans un graphe quelconque, de nombreux problèmes combinatoires difficiles ne possèdent pas d'algorithme de résolution rapide. On peut prendre comme exemple la coloration : trouver le plus petit nombre de couleurs qu'il faut utiliser pour colorer chaque sommet d'un graphe de manière à ce que deux sommets adjacents ne soient jamais de même couleur. Cependant, ces

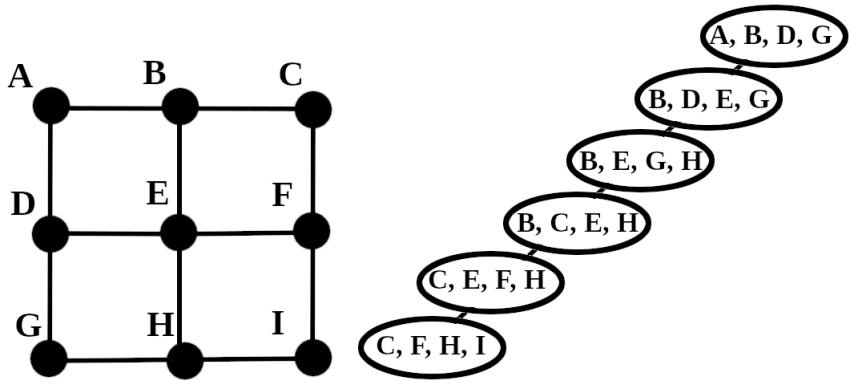

FIGURE 4 – Une grille et une de ses tree-decompositions

mêmes problèmes deviennent souvent plus simples (voire même triviaux) lorsqu'ils sont appliqués à des arbres. Bodlaender montre [3] qu'il est possible de résoudre ces problèmes en utilisant des techniques de programmation dynamique sur des graphes dont la treewidth est bornée.

Plus précisément, le théorème de Courcelle [6] nous dit que "toute propriété de graphe définissable en logique monadique du second ordre peut être décidée en temps linéaire sur les graphes dont la treewidth est bornée". En d'autres termes, il existe un grand nombre de problèmes difficiles décidables en temps linéaire si la treewidth du graphe est bornée.

Pour qu'une décomposition nous aide à trouver une solution à ces problèmes difficiles par programmation dynamique, la minimisation de paramètres tels que la width est très intéressante. Malheureusement, il est prouvé (Arnborg et al. [2]) que décider si la treewidth d'un graphe est au plus k est NP-complet. Il existe cependant un algorithme [4] permettant d'obtenir en temps $O(k^{k^3} \cdot n)$ une tree-decomposition de width k si elle existe. La différence majeure entre ces deux résultats étant que l'algorithme dépend de la valeur de k donnée en entrée.

De la même manière, la length peut aussi être intéressante pour concevoir des algorithmes efficaces pour résoudre des problèmes difficiles. Par exemple, les graphes dont la treelength est bornée trouvent des applications dans des algorithmes de routage compact [7], ou dans le problème du voyageur de commerce. Cependant, il est prouvé que déterminer si la treelength d'un graphe est au plus 2 est NP-complet (Lokshtanov, 2007 [11]). Des algorithmes d'approximation très simples existent, avec en particulier une 3-approximation, et une possible 2-approximation - non prouvée - de la treelength (Dourisboure et al. [8]).

Nous allons ici nous intéresser à différents algorithmes d'approximation de la treelength. Nous commencerons par définir plus formellement le sujet (section 3), puis nous présenterons les différents algorithmes étudiés (section

4), et pour certains les améliorations que nous avons pu y apporter, avant de comparer leurs performances (section 5) en terme de temps et de length de la décomposition obtenue. Enfin, nous présenterons en section 6 quelques pistes pour prouver qu'un des algorithmes étudiés est une 2-approximation de la treelength.

3 Définitions générales

3.1 Tree-decomposition

Tout d'abord, il est important de définir formellement ce qu'est une tree-decomposition (ou décomposition arborescente) telle que définie par Robertson et Seymour en 1986 [12]. Pour un graphe G, on appelle $V(G)$ son ensemble de sommets et $E(G)$ son ensemble d'arêtes :

Définition 1 (Tree-decomposition)**.** *Une tree-decomposition d'un graphe G est un couple $(T, \{X_t : t \in V(T)\})$ tel que T est un arbre, $\{X_t : t \in V(T)\}$ une famille de sous-ensembles de $V(G)$, et :*

- *Pour tout $v \in V(G)$ il existe $t \in V(T)$ tel que $v \in X_t$*
- *Pour tout $uv \in E(G)$ il existe $t \in V(T)$ tel que $u, v \in X_t$*
- *Pour tout $v \in V(G)$, le sous-graphe de T induit par les sommets $i \in V(T)$ tels que $v \in X_i$ est un sous-arbre de T.*

Les tree-decompositions permettent de définir deux paramètres de graphe. La treewidth est définie par Robertson et Seymour en 1986 [12] :

Définition 2 (Width)**.** *Soit T une tree-decomposition. On appelle width(T) le nombre $width(T) = max_{X \in V(T)}|X| - 1$*

Figure 4 nous pouvons voir que le plus grand que la décomposition a une width de 3 car le plus grand sac contient 4 sommets.

Définition 3 (treewidth)**.** *La treewidth d'un graphe G est la valeur minimum de la width parmi toutes les tree-decompositions de G*

Nous pouvons affirmer que la treewidth d'un graphe est toujours définie : en mettant tous les sommets du graphe dans un seul sac, on obtient une décomposition dont la width est égale à $|V(G)| - 1$, ce qui nous donne une borne supérieure sur la treewidth du graphe.

En 2004, Dourisboure et Gavoille définissent la treelength [8] d'un graphe. C'est à ce paramètre que nous nous intéressons principalement par la suite.

On note, pour un sac X d'une décomposition de G, $diam(X)$ la distance dans G maximale dans entre deux sommets du sac X.

Définition 4 (length)**.** *Soit T une tree-decomposition. On appelle length(T) le nombre $length(T) = max_{X \in V(T)} diam(X)$.*

Définition 5 (treelength)**.** *La treelength d'un graph G est la valeur minimum de la length parmi toutes les tree-decompositions de G.*

Dans la figure 4 nous pouvons voir que la length de la décomposition est 3 car la distance maximale entre deux sommets d'un même sac est égale à 3.

La treelength d'un graphe est elle aussi toujours définie : la length de la décomposition constituée d'un seul sac contenant tous les sommets du graphe est égale au diamètre du graphe (la distance entre les deux sommets les plus éloignés), une borne supérieure de la treelength.

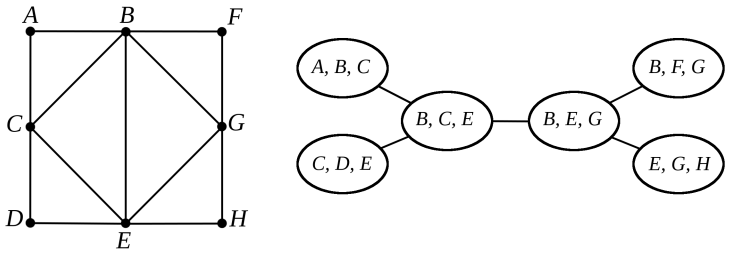

FIGURE 5 – Un graphe et une tree-decomposition de ce graphe

L'exemple de la figure 5 nous permet d'illustrer ces définitions : chaque sommet et chaque arête du graphe se trouve dans au moins un sac de la décomposition et pour chaque sommet du graphe les sacs contenant ce sommet forment un sous-arbre de la décomposition.

D'autre part, tous les sacs contiennent 3 sommets, la width est donc égale à 2. La longueur maximal d'un chemin entre deux sommets d'un même sac est égale à 1, qui est donc la length. Nous pouvons de ce fait en déduire que la treelength du graphe est 1 car il n'est pas possible d'avoir une décomposition de length 0 pour un graphe avec au moins une arête. En ce qui concerne la treewidth, nous savons qu'elle est égale à 2 car nous avons une décomposition de width 2 et qu'on ne peut pas trouver de décomposition de width 1 car le graphe n'est pas un arbre.

Soit G un graphe et $S \subseteq V(G)$ un sous-ensemble de sommets. Le sous-graphe induit $G[S]$ est le graphe dont l'ensemble des sommets est S, et dont les arêtes sont l'ensemble des $uv \in E(G)$ tels que $u \in S$ et $v \in S$.

Définition 6 (Séparateur)**.** *Soient G un graphe et $A \subseteq V(G)$. A est un séparateur de G si $G[V(G) \setminus A]$ n'est pas connexe.*

Une conséquence importante de la troisième propriété des tree-decompositions est que pour deux sacs adjacents, si aucun n'est inclu dans l'autre alors leur intersection est un séparateur du graphe. Dans la figure 5 l'intersection des sacs $\{A, B, C\}$ et $\{B, C, E\}$ sépare A du reste du graphe.

3.2 Graphes chordaux

Il nous faut aussi définir la famille des graphes chordaux car ils sont étroitement liés aux tree-decompositions.

Étant donné un graphe G, un cycle induit est un sous-graphe induit par un ensemble de sommets $v_1...v_n$ tels que $v_i v_{i+1 mod\ n} \in E(G)$ et pour tout u, v non consécutif $uv \notin E(G)$.

Définition 7 (Graphe chordal et triangulation)**.** *La famille des graphes chordaux correspond aux graphes qui ne contiennent pas de cycle induit de longueur supérieure à 3. En particulier, dans un graphe chordal, tout cycle de longueur au moins 4 contient une corde. On appelle triangulation l'ajout d'arêtes à un graphe pour le rendre chordal.*

Soit G un graphe et A un séparateur de G. A est un séparateur minimal de G s'il n'existe pas de sous ensemble $B \subset A$ tel que $G[V(G) \setminus B]$ n'est pas connexe. Tous séparateur minimal d'un graphe chordal induit une clique.

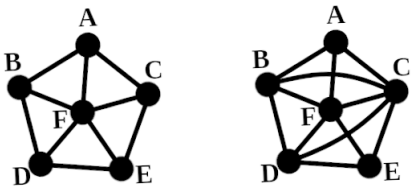

FIGURE 6 – Un graphe non chordal (gauche) et une triangulation chordale de ce graphe (droite)

Par exemple, dans la figure 6, le graphe de gauche n'est pas chordal car il existe un cycle induit (sans corde) de longueur 5 passant par les sommets A, C, E, D, B. En revanche, il n'y aucun cycle induit de longueur supérieure à 3 dans celui de droite, il est donc chordal.

Soit G un graphe, une *triangulation de* G est un graph chordal H tel que $V(G) = V(H)$ et $E(G) \subseteq E(H)$. En d'autres termes, on ajoute des arêtes à un graphe G pour le rendre chordal.

On appelle *graphe des cliques de* G le graphe K_G tel que :

— Il existe une bijection entre les cliques maximales (par inclusion) de G et les sommets de K_G

— Il y a une arête entre $u, v \in V(K_G)$ si les cliques correspondantes à u et v dans G ont au moins un sommet en commun. Le poids d'une telle arête est la taille de l'intersection entre les deux cliques.

Étant donné un graphe, on peut définir une fonction $w(u, v)$ associant à chaque arête de G un entier. On appelle $w(u, v)$ le poids de l'arête $(u, v) \in E(G)$.

Pour un graphe G, un arbre couvrant de G est un arbre T tel que $V(G) = V(T)$, $E(T) \subseteq E(G)$. Cet arbre couvrant est dit maximum s'il n'existe pas

d'arbre couvrant T' tel que la somme du poids des arêtes de T' est strictement supérieure à celle de T.

Nous connaissons un algorithme [9] pour obtenir une décomposition optimale (en terme de length) d'un graphe chordal : il suffit de prendre un arbre couvrant maximum du graphe des cliques maximales d'un graphe chordal. Rendre un graphe chordal (par ajout d'arêtes) est donc une technique permettant de le décomposer. Le problème principal est de trouver une "bonne" triangulation. En effet, transformer le graphe en une clique contenant tous les sommets permet d'obtenir un graphe chordal, mais la décomposition n'aura pas une width intéressante. La treewidth d'un graphe chordal étant égale à la taille de la plus grande clique moins 1, une "bonne" triangulation serait donc une triangulation qui augmente le moins possible la taille de la plus grande clique du graphe.

Inversement, étant donné G un graphe et T une tree-decomposition de G de width k, le graphe obtenu à partir de G en formant des cliques à partir des sommets de chaque sac de T est une triangulation de G dont la clique maximale est de taille $k+1$.

4 Algorithmes étudiés

Les algorithmes présentés ici sont implémentés en Python 3 en utilisant la bibliothèque graphes de SageMath[15]. Nous commencerons par présenter lex-M [13] (section 4.1), puis bfs-layering [8] (section 4.2) avant de nous intéresser à uncle-trees [14] (section 4.3) et disk-tree [8] (section 4.4). Il s'agit d'algorithmes d'approximation et d'heuristiques pour approximer la tree-length du graphe en entrée.

4.1 Lex-M

L'algorithme Lex-M proposé par Rose et al. [13] en 1976 permet de calculer en temps $O(n \cdot m)$ une triangulation d'un graphe. Comme vu précédemment (section 3.2), cette triangulation est équivalente à une tree-decomposition du graphe en entrée. De plus, Dourisboure et Gavoille prouvent [8] le théorème suivant :

Theorem 1. *L'algorithme lex-M retourne une tree-decomposition de G de length au plus $3 \cdot treelength(G) + 1$.*

Intuitivement, on part d'un sommet arbitraire qu'on numérote, puis on passe à un de ses voisins (ce qui s'apparente à un parcours en largeur). On le numérote et on ajoute une arête entre ce sommet et tous les sommets accessibles par un chemin dont les sommets ne sont pas encore numérotés. Puis on répète en choisissant toujours un sommet dont le label est maximal jusqu'à ce que tous les sommets soient numérotés.

L'idée est de numéroter chaque sommet du graphe tel que pour chaque sommet $v \in V(G)$, l'ensemble de ses voisins dont le numéro est supérieur à celui de v forme une clique. Cette numérotation est un ordre sur les sommets qu'on appelle un *perfect elimination ordering (peo)*, elle n'existe qu'à condition que le graphe soit chordal.

Pour cela, on associe à chaque sommet $u \in V(G)$ un label $label(u)$ qui est un ensemble de nombres compris entre 1 et n dans l'ordre décroissant et on définit un ordre sur ces labels.

Algorithme 1 : Lex-M

input : G un graphe
output : Une triangulation de G (i.e. un supergraphe chordal de G)

Assigner un label vide à chaque sommet de G;
$edges \leftarrow \emptyset$ // *les arêtes à ajouter pour rendre G chordal*;
pour *i de 1 à n* **faire**
  Sélectionner un sommet u non-numéroté avec un label maximal;
  Assigner à u le numéro i;
  pour *chaque sommet v non-numéroté tel qu'il y a un chemin $u, w_1, ..., w_p, v$ avec w_j non numéroté et $label(w_j) < label(u)$ pour $j \in \{1, ..., p\}$* **faire**
    $label(v) \leftarrow label(v) \cup i$;
    $edges \leftarrow edges \cup (u, v)$
  fin
fin
retourner *edges*

4.1.1 Implémentation

L'algorithme étant déjà implémenté dans SageMath [15] par David Coudert, nous n'avons besoin que d'un algorithme permettant de passer d'un graphe chordal à la tree-decomposition. En plus de renvoyer H une triangulation de G, l'implémentation existante de Lex-M donne aussi l'ordre d'élimination des sommets (peo) de H.

Soit $PEO = \{v_1 ... v_n\}$ un ordre parfait d'élimination des sommets de H, on note $rank(v_i) = i$ la place d'un sommet v_i dans PEO.

Pour chaque $v_i \in PEO$, on associe un couple $(S(v_i), m(v_i))$. $S(v_i)$ désigne les sommets w du voisinage fermé de v_i (v_i et ses voisins) tels que $rank(w) \geq rank(v_i)$, et $m(v_i)$ désigne le sommet du sac $S(v_i)$ ayant le plus petit rank.

Nous prenons chaque sommet de H dans l'ordre donné par PEO, et nous ajoutons une arête entre $S(v_i)$ et $S(m(v_i))$ pour former une tree-decomposition de H.

Lemma 2. *Le résultat de cet algorithme est une tree-decomposition de G.*

Démonstration. Soit T le résultat de l'algorithme.

Pour tout $v_i \in V(H), v \in S(v) \in V(T)$, donc tous les sommets sont couverts par T.

Pour tout $u, v \in E(H)$, soit $(u, v) \in S(u)$, soit $(u, v) \in S(v)$, donc toutes les arêtes sont couvertes par T.

Nous allons maintenant montrer que le sous-arbre de T induit par les sacs contenant un sommet v est connexe.

Tout d'abord, il est évident que $S(v_1)$ est une tree-decomposition de $G[S(v_1)]$. Admettons par récurrence avoir T_k une tree-decomposition de $G[V(T_k)]$ avec $V(T_k) = \{S(v_1), ..., S(v_k)\}$.

Nous voulons montrer que

$T_{k+1} = (V(T_k) \cup S(v_{k+1}), E(T_k) \cup (S(v_{k+1}), S(m(v_{k+1}))))$ est une tree-decomposition de $G[\bigcup_{X \in V(T_{k+1})} X]$.

Supposons l'inverse : alors il existe un sac X descendant de $S(m(v_{k+1}))$ contenant un sommet $u \neq v_{k+1}$ présent dans $S(v_{k+1})$ mais pas dans $S(m(v_{k+1}))$.

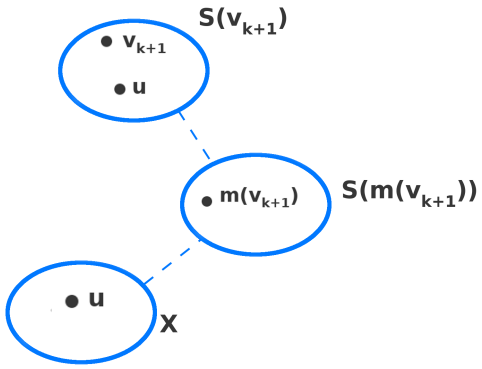

FIGURE 7 – Illustration de la situation décrite pour T_{k+1}

Comme $u \in S(v_{k+1}), rank(u) > rank(v_{k+1})$.

Comme $u \notin S(m(v_{k+1})), rank(u) < rank(m(v_{k+1})$. Or, par définition de PEO, $rank(m(v_{k+1})) < rank(v_{k+1})$.

On a donc $rank(u) < rank(m(v_{k+1})) < rank(v_{k+1}) < rank(u)$, une contradiction.

Donc, T est bien une tree-decomposition de H. □

4.2 BFS-Layering

Nous détaillons ici l'algorithme BFS-layering. Nous commençons par définir les structures intermédiaires nécessaires et détailler leur construction avant de donner l'algorithme lui même. Nous proposons ensuite une petite amélioration permettant d'obtenir une meilleure approximation de la tree-length dans certains cas.

4.2.1 Définitions

Définition 8 (Layering partition [8]). *Soit G un graphe et $s \in V(G)$. Pour tout $i \geq 0$ on définit $L^i = \{u \in V(G) | dist_G(s,u) = i\}$*

Une layering partition de G est une partition de chaque ensemble L^i en $L^i_1 \dots L^i_p$ tels que $u, v \in L^i_j$ si et seulement si il existe un chemin entre u et v ne passant que par des sommets w tels que $dist_G(s,w) \geq i$.

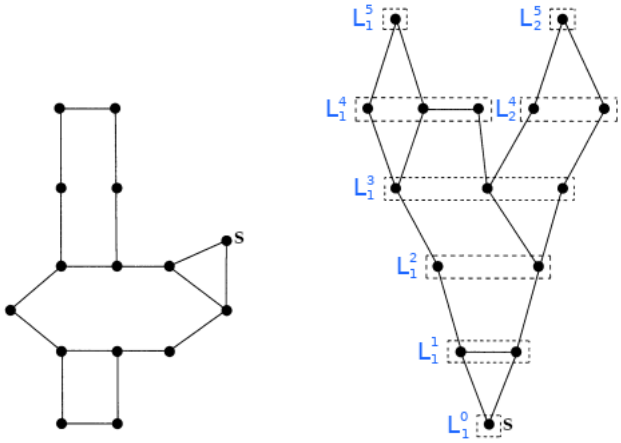

FIGURE 8 – Un graphe (gauche) et une layering-partition de ce graphe (droite) [5]

Définition 9 (Layering-tree [8]). *Soit un graphe G et une layering-partition de ce graphe en des ensembles L^i_j.*

On appelle layering-tree l'arbre H dont l'ensemble des sommets est l'ensemble des L^i_j, et une arête existe entre L^i_j et $L^{i'}_{j'}$ dans H s'il existe $u \in L^i_j$ et $v \in L^{i'}_{j'}$ tels que u et v sont adjacent dans G et $|i - i'| = 1$.

4.2.2 Description de l'algorithme

L'algorithme est basé sur un parcours en largeur du graphe à partir d'un sommet s (BFS). On appelle $S_r(s)$ la sphère de rayon r centrée en s, c'est à dire tous les sommets v_i tels que $dist_G(u, v_i) = r$.

Chepoi et Dragan [5] proposent un algorithme simple pour construire en temps $O(|V(G)| + |E(G)|)$ un layering-tree de G :

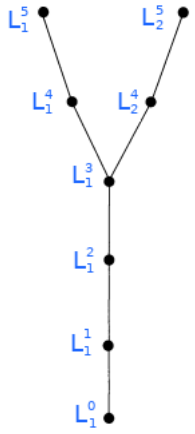

FIGURE 9 – Le layering-tree du graphe précédent [5]

Algorithme 2 : Layering-tree

input : G un graphe
output : Un layering-tree de G

// *On peut modifier G lui même ou travailler sur une copie du graphe*;
$contracted \leftarrow \emptyset$;
Choisir $u \in V(G)$;
Calculer $S_p(u)$ la sphère de rayon maximal centrée en u;
Calculer $CC(n) = \{CC_1, ..., CC_k\}$ les composantes connexes de $G[S_n(u)]$;
Contracter les sommets de $CC(n)$ en k sommets $w(n) = w_1, ..., w_k$;
$contracted \leftarrow contracted \cup w(n)$;
pour i *de* $p - 10$ **faire**
    Calculer $CC(i)$ les composantes connexes de $G[S_i(u) \cup contracted]$;
    Contracter $CC(i)$ en $w(i)$;
fin
retourner G

Nous pouvons maintenant donner l'algorithme BFS-Layering permettant de passer d'un layering-tree à une tree-decomposition en temps $O(|V(G)| + |E(G)|)$:

Algorithme 3 : BFS-Layering

input : G un graphe
output : Une tree-decomposition de G

```
1 H ← Layering − tree(G);
2 T ← copie(H);
3 pour U ∈ V(T) faire
4 |  U ← U ∪ V, V parent de U dans H;
5 fin
6 retourner T
```

Intuitivement, on associe à chaque L_j^i du layering tree un sac contenant l'union de L_j^i et de son parent.

Dourisboure et Gavoille prouvent [8] ainsi le résultat suivant :

Theorem 3. *Le résultat de BFS-Layering est une tree-decomposition de length au plus* $3 \cdot treelength(G) + 1$.

C'est donc une approximation légèrement moins bonne qu'une vraie 3-approximation, mais nous pouvons changer légèrement l'algorithme pour obtenir une meilleure approximation dans certains cas, même si le ratio d'approximation reste le même.

4.2.3 Amélioration proposée

Nous proposons une légère amélioration de BFS-Layering. La ligne 4 de l'algorithme 3 prend une union simple entre une couche et la couche parente dans le layering-tree. S'il n'y a pas d'arête entre un sommet du parent et un sommet de l'enfant, on prend un sommet inutile qui augmente donc le diamètre du sac. Par exemple, dans la figure 11, dans le sac $\{B, C, D, E, F\}$ de la décomposition (2), le sommet D augmente le diamètre du sac alors qu'il est possible de le retirer.

Pour deux sommets du layering-tree A et B, A parent de B, au lieu de définir le sac correspondant à B dans la tree-decomposition comme $B \cup A$, on prend plutôt $B \cup (A \cap N_G(B))$, c'est-à-dire qu'on ajoute à B les sommets de A adjacents à un sommet dans B.

Lemma 4. *Étant donné un graphe G, l'algorithme modifié donne une tree-decomposition de G.*

Démonstration. On considère T_s la décomposition arborescente de G par l'algorithme de BFS-layering modifié en partant de $s \in V(G)$, L_s le layering-tree de G partant de s. Nous voulons prouver que T_s est bien une décomposition arborescente.

— Chaque sommet de G est dans au moins un sac de T_s

- Pour tout $A \in V(L_s)$, il existe un sommet $X = A \cup (parent(A) \cap N_G(A))$ dans $V(T_s)$. Par définition d'un layering-tree, chaque sommet $x \in V(G)$ est contenu dans un sac de L_s. Il est donc obligatoirement dans un sac de T_s.

— Chaque arête de G est dans au moins un sac de T_s
  - Soient $H, B \in V(T_s)$, H parent de B dans T_s, $x \in H$. Pour les sommets $y \in N_G(x)$ on distingue 3 cas :
    - $dist_G(x,s) = dist_G(y,s)$ alors $y \in H$ par définition du layering-tree. Donc l'arête (x,y) est bien dans un sac de T_s
    - $dist_G(x,s) = dist_G(y,s) - 1$ alors $y \in B$. L'arête (x,y) est donc dans le sac $X = B \cup (H \cap N_G(B))$
    - $dist_G(x,s) = dist_G(y,s) + 1$: on applique le même raisonnement que le cas précédent sur H et son parent, (x,y) est bien dans un sac

— Pour tout $x \in V(T_s)$, le sous-arbre de T_s induit par les sacs contenant x est connexe
  - Comme un sommet $x \in V(G)$ ne peut apparaître que dans un seul sac de L_s, il apparaît au plus dans un sac $X \in V(T_s)$ et les successeurs de ce sac. La seule possibilité pour que le sous-graphe induit par les sommets ne soit pas connexe est que X ne contienne pas x et qu'au moins deux de ses successeurs, Y et Z, contiennent x.

    Il existe $A, B, C \in V(L_s)$ tels que $A \subseteq X$, $Y = B \cup (A \cap N_G(B))$ et $Z = C \cup (A \cap N_G(C))$. Supposons que $x \notin X$. Alors $x \notin A$, et donc on peut en déduire que $x \in B, C$ puisque $x \in Y, Z$. On trouve que x est inclu dans deux sacs de L_s, par définition du layering-tree c'est une contradiction (c.f. figure 10).

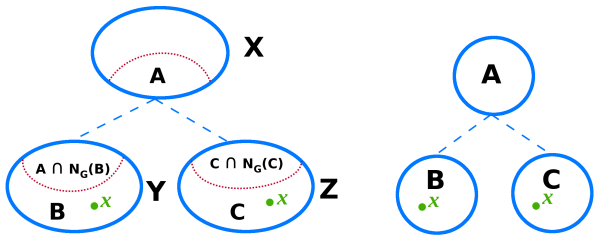

FIGURE 10 – Contradiction : la décomposition (gauche) et le layering-tree correspondant (droite)

  - Donc le sous-graphe de T_s induit par les sacs contenant x est bien connexe.

T_s est donc bien une tree-decomposition de G. □

Dans la figure 11 ci-dessus, nous avons un graphe en (1) et la décomposition par la première version de BFS-Layering en (2) en partant du sommet

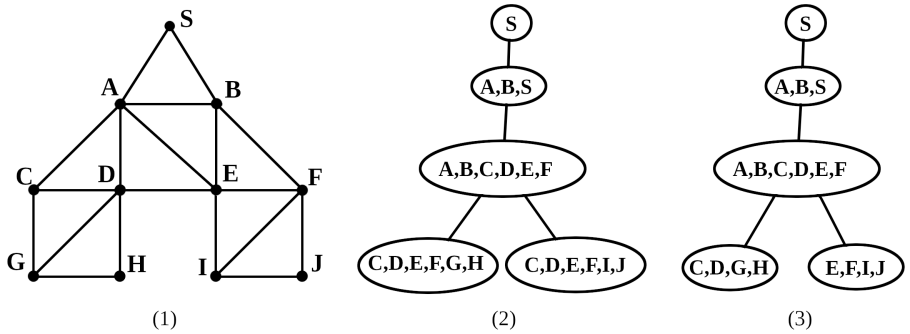

FIGURE 11 – Un graphe décomposé avec les deux versions de BFS-Layering

S. Ce graphe est chordal, il a donc une treelength de 1. Cette décomposition a une length de 4 : J et C sont dans un même sac et leur distance est égale à 4, mais les sommets C et D sont superflus dans le sac $\{C, D, E, F, I, J\}$ car si on les retire du sac la décomposition est toujours valide, mais la distance entre les deux sommets le plus éloignés de ce sac est égale à 2.

En (3) nous avons la décomposition avec notre amélioration en partant du même sommet, et cette fois-ci la length est égale à 3 (distance entre C et F dans le sac $\{A, B, C, D, E, F\}$). Les sommets superflus ne sont pas pris dans les sacs, et la length est effectivement inférieure.

Nous pouvons par ailleurs prouver que la length de la décomposition obtenue grâce à cette modification est au plus 3 dans les graphes chordaux, donc une vraie 3-approximation dans cette classe de graphe.

Lemma 5. *Soient H la tree-decomposition obtenue par bfs-layering d'un graphe chordal G et $X, Y \in E(H)$. Alors $X \cap Y$ forme une clique qui sépare G.*

Démonstration. Soit $X, Y \in E(H)$, X est un enfant de Y dans H, comme H est une tree-decomposition par bfs-layering de G, $S = X \cap Y$ est un séparateur de G. En effet, il est trivial d'observer qu'une décomposition par bfs-layering ne contient pas de feuille inclue dans son parent. Il reste à montrer que S est une clique :

Supposons que S n'est pas une clique. Alors, il existe u et v dans S tels que u et v ne sont pas adjacents. Soit A, B les noeuds du layering-tree ayant servi à construire le sac X. En supposant que A est parent de B, $X = B \cup (A \cap N_G(B)$, on a donc $S = A \cap N_G(B) \subseteq A$. u et v on respectivement un voisin z et w dans B, possiblement le même. Comme u et v appartiennent au même noeud du layering tree et qu'un noeud induit toujours un sous-graphe connexe de G, il existe un plus court chemin passant par un sommet s dans une couche de profondeur inférieure ou égale à u et v. Il existe donc un cycle sans corde passant par u, z, w, v, s de longueur strictement supérieure à 3 (c.f. figure 11), ce qui est une contradiction puisque G est chordal. □

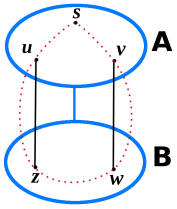

FIGURE 12 – Contradiction : on trouve un cycle induit de longueur supérieure à 3 dans un graphe chordal

Theorem 6. *La length de la décomposition obtenue par bfs-layering d'un graphe chordal est au plus 3.*

Démonstration. Soient un graphe chordal G, L un layering-tree de G et H la décomposition de G par bfs-layering obtenue à partir de L.

Pour tout $X \in V(H)$, il existe une arête $A, B \in E(L)$, avec A enfant de B, telle que $X = A \cup (B \cap N_G(A))$. Nous devons montrer que pour tout $u, v \in X$, $dist_G(u, v) \leq 3$. Nous pouvons distinguer 3 cas :

- $u, v \in B$: Dans ce cas, nous savons que u et v sont dans $B \cap N_G(A)$ qui, d'après le lemme précédent, est une clique. Donc $dist_G(u, v) = 1$
- $u, v \in A$: Ici, soit u et v sont voisins dans A, auquel cas leur distance est 1, sinon il existe deux sommets $x_1, x_2 \in B \cap N_G(A)$ respectivements adjacents à u et v. Ces deux sommets peuvent être égaux, donc d'après le lemme précédent $dist_G(x_1, x_2) \leq 1$. On obtient donc que $dist_G(u, v) = dist_G(u, x_1) + dist_G(x_1, x_2) + dist_G(x_2, v) \leq 3$
- $u \in A$, $v \in B$: Comme $v \in X$ et $v \in B$, alors $v \in B \cap N_G(A)$. Comme vu précédemment, il existe $w \in B \cap N_G(A)$ tel que $w \in N_G(u)$. v et w pouvant être égaux, leur distance est au plus 1 car $B \cap N_G(A)$ est une clique. Donc $dist_G(u, v) = dist_G(u, w) + dist_G(w, v) \leq 2$

On obtient bien dans chaque cas que $dist_G(u, v) \leq 3$, la length de la décomposition est donc au plus 3. □

4.3 Uncle-trees

Introduit par Seymour [14] en 2016, uncle-trees est une heuristique permettant d'obtenir une tree-décomposition pour un paramètre k donné, à condition que le graphe ne contienne pas de cycle induit de longueur supérieure à k.

Nous commencerons par définir ce qu'est un uncle-tree, puis comment obtenir une tree-decomposition à partir de cette structure. Enfin, nous présenterons l'algorithme nous permettant de calculer k.

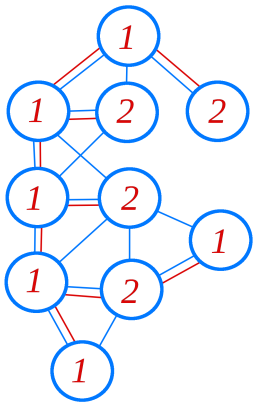

FIGURE 13 – Un graphe (bleu) et un uncle-tree de ce graphe (rouge) avec l'ordre sur les enfants des sommets

4.3.1 Définitions

Tout d'abord, dans un arbre T enraciné en s, pour deux sommets quelconques u et $v \in V(T)$ on définit les termes suivants :

— u et un enfant de v si $(u, v) \in E(T)$ et $dist_T(s, u) = dist_T(s, v) + 1$. Dans ce cas là, v est le parent de u

— u est un ancêtre de v si u est sur le chemin de s à v et $dist_T(s, u) < dist_T(s, v)$

Étant donné un arbre couvrant enraciné et un ordre sur les enfants de chacun de ses sommets (c'est-à-dire qu'on "range" les enfants dans un certain ordre), nous définissons deux termes supplémentaires.

Étant donnés deux sommets u et v d'un arbre T avec un ordre sur les enfants de ses sommets, on dit que u est un oncle de v si le parent de u a un fils w qui est ancêtre de v dans T tel que $w < u$.

Un arbre T avec un ordre sur les enfants est un uncle-tree d'un graphe G si T est un arbre couvrant de G et pour tout $uv \in E(G)$ tels que $uv \notin E(T)$, u est oncle de v ou v est oncle de u

4.3.2 Description de l'algorithme

L'article de Seymour [14] décrit un algorithme permettant de construire un uncle-tree. L'idée principale est de construire l'arbre en effectuant un parcours en profondeur du graphe à partir d'une racine choisie arbitrairement et en numérotant les sommets selon ce parcours (pour obtenir un ordre sur les sommets), soumise à la contrainte que chaque chemin entre la racine et les feuilles de l'arbre doit être un chemin induit dans le graphe.

Algorithme 4 : Uncle-tree

Input : G un graphe
Output : Un uncle-tree de G

Soient D, B, P trois ensembles
/* D les sommets supprimés, P ceux du chemin et B les voisins du chemin */
$Pred$ un dictionaire associant à chaque sommet son parent
$p_k \leftarrow s$ // un sommet initial arbitraire

tant que $P \neq \emptyset$ **faire**
  si $Y \leftarrow N(p_k) \setminus (D \cup B \cup P) \neq \emptyset$ **alors**
    $x \leftarrow Y[0]$
    $T[x] \leftarrow p_k$
    Ajouter p_k à P
    $B \leftarrow B \cup (Y \setminus x)$
  sinon
    /* p_k est "supprimé" du graphe */
    $D \leftarrow D \cup p_k$
    $P \leftarrow P \setminus p_k$
    $old\_p_k \leftarrow p_k$
    $p_k \leftarrow T[p_k]$
    /* certains sommets ne sont plus voisins du chemin */
    pour $v \in (N(p_k) \cup N(old\_p_k)) \cap B$ **faire**
      si $N(v) \setminus D) \cap (P \setminus p_k) = \emptyset$ **alors**
        remove v from B

retourner T

Dans le cadre de l'implémentation, nous avions besoin d'une manière de reconnaître certaines relations (oncle, ancêtre par exemple) entre les sommets plus rapidement qu'en effectuant une recherche à chaque fois. Lors de la construction de l'uncle-tree, les sommets sont étiquetés par des intervalles dont la borne inférieure est le numéro du sommet, et la borne supérieure est le numéro de son dernier descendant numéroté. C'est donc l'ordre donné par une recherche en profondeur (ordre DFS), que nous utilisons pour l'implémentation de la décomposition. La définition ci-après est une simplification de celle donnée par Seymour, elle n'est valide qu'à condition d'utiliser un ordre DFS.

Définition 10 (Junior)**.** *Soit un uncle-tree T et un ordre DFS sur ses sommets. Soient $s, t \in V(T)$, s est junior de t si aucun n'est ancêtre de l'autre et s a été numéroté après t.*

On appelle Q_t le chemin entre le sommet t et la racine s de l'arbre de longueur maximale k.

Étant donné l'uncle-tree T d'un graphe G, Seymour [14] montre que l'arbre $(T, X_t : t \in V(T))$, calculable en temps $O(n^2)$, définit comme suit est une tree-decomposition de G :

Pour tout $t \in V(T)$ on crée un ensemble X_t contenant les sommets v de G tels que :

— $v \in V(Q_t)$ (chemin de longueur au plus k) ou
— v est un enfant de t dans T, ou
— v est junior de t et adjacent dans G à un sommet de Q_t

4.3.3 Calcul de k

L'algorithme décrit précédemment dépend d'un paramètre k, qui est la longueur du plus grand cycle induit du graphe à décomposer. Cependant, nous voulons pouvoir utiliser cet algorithme sans avoir besoin de connaître au préalable ce paramètre. L'idée est donc de trouver un algorithme pour le calculer quand il n'est pas donné en entrée.

L'entier k n'intervient qu'au moment de la construction des sacs de la décomposition. Plus précisément, k est la longueur du chemin Q_t, le chemin qui remonte du sommet t en direction de la racine de l'uncle-treee.

Au cours de la preuve que la construction de ces sacs donne bien une tree-decomposition, Seymour montre que si le graphe ne contient pas de cycle induit de taille supérieure à k, si deux sacs X_w et X_t contiennent un sommet v qui n'est pas sur le chemin entre les deux, alors tous les sacs sur le chemin de X_w à X_t contiennent aussi v.

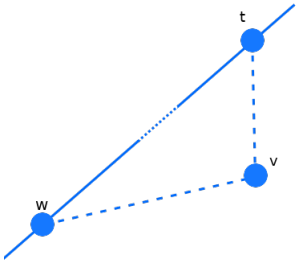

FIGURE 14 – v est voisin de w et t dans le graphe mais pas dans son uncle-tree

C'est donc dans ce cas qu'une valeur de k trop petite donne lieu à une décomposition invalide. Spécifiquement, v est contenu dans le sac de w et de s si v est junior ou enfant des deux à la fois dans l'uncle-tree, mais si k est trop petit, il existera un sommet entre w et s ne contenant pas v. L'ensemble des sacs de la décomposition contenant v n'induira pas un sous-arbre.

L'idée de l'algorithme est donc de retrouver cette configuration dans l'uncle-tree, afin d'obtenir le plus grand $k = dist_T(s,t) + 2$. En d'autres termes, chercher un ensembles de trois sommets w, t, v tels que v est enfant ou junior de s et t, et $dist(w,t)$ est maximum.

Algorithme 5 : Calcul de k

Input : G un graphe, T un uncle-tree de G
Output : $k \geq 3$ tel que la décomposition par uncle-trees est valide

```
1 k ⟵ 3
2 pour v ∈ V(G) faire
3     pour u ∈ N_G(v) faire
4         si u est junior de v ou v est parent de u dans T alors
5             pour w ∈ N_G(u) tel que w est en dessous de v dans T
                faire
6                 si u est junior de w ou w est parent de u dans T alors
7                     k ⟵ max(k, dist_T(v, w) + 2))
8 retourner k
```

Il est toutefois important de noter que la valeur de k retournée n'est pas la longueur du plus grand cycle induit, trouver le plus grand cycle induit dans un graphe étant un problème NP-Complet [10]. Seymour explique que cet algorithme de décomposition rend une décomposition valide si $k \geq l$ (où l est la longueur du plus grand cycle induit dans G). Cependant, cela n'exclut pas la possibilité de trouver une décomposition valide avec une valeur de k plus petite.

La valeur retournée est donc la plus petite valeur de k permettant d'obtenir une décomposition valide avec l'uncle-tree donné.

4.4 Disk-Tree

Disk-tree est aussi une heuristique. Étant donné un graphe et un entier k, il peut renvoyer une décomposition de length au plus $2 \cdot k$ ou échouer. Il est prouvé [8] que si $k \geq 3 \cdot treelength(G) - 2$ l'algorithme retourne toujours une décomposition de length au plus $2 \cdot k$. Cependant, il est possible (mais pas prouvé) que l'algorithme retourne une décomposition pour $k \leq treelength(G)$, auquel cas cet algorithme serait une 2-approximation de la treelength.

4.4.1 Définitions

Nous avons besoin de définir plusieurs termes avant de pouvoir présenter l'algorithme.

Étant donnés un graphe G et un sous-arbre T d'une tree-decomposition de G, on définit $covered(T) = \bigcup_{X \in V(T)} X$ l'ensemble des sommets de G contenus dans au moins un sac de T.

L'ensemble des sommets de $covered(T)$ ayant au moins un voisin hors de $covered(T)$ est appelé $border(T)$.

Pour tout $S \subseteq V(G)$ et $x \in V(G)$, on appelle $out(x, S)$ l'ensemble des sommets de S reliés à x par un chemin dont tous les sommets intermédiaires sont hors de S.

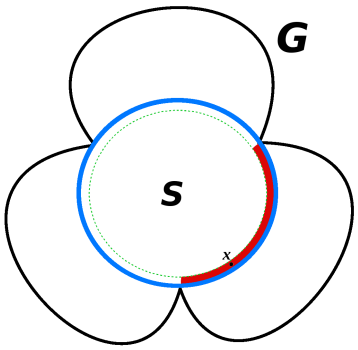

FIGURE 15 – $out(x, S)$ (sommets en rouge)

Soit $S \subseteq V(G)$, et u, v deux sommets de S. u et v sont dits *en conflit* si $v \in out(u, covered(T) \cup S)$ et $dist_G(u, v) > k$.

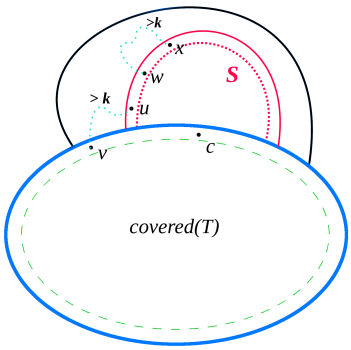

FIGURE 16 – u et v, w et x sont en conflit dans S

Pour un arbre A enraciné en s et $x \in V(A)$, on appelle *profondeur de* x la distance de s à x dans A.

Pour chaque $v \in covered(T)$ on appelle $B(u)$ le sac de T de profondeur minimum contenant v.

4.4.2 Description de l'algorithme

L'algorithme disk-tree est basé sur l'idée que si un graphe admet une tree-decomposition T de length k, alors pour tout $X \in V(T)$, X est inclu dans le disque de rayon k centré en v pour tout $v \in X$. On essaye de construire une décomposition de length $2k$ en construisant des disques de rayon k desquels on peut supprimer quelques sommets car il n'est pas toujours possible d'organiser ces disques sous la forme d'un arbre.

On note $D^k(c)$ le disque de rayon k centré en $c \in V(G)$.

Intuitivement, à chaque itération, étant donné T_i l'arbre de décomposition construit jusque là l'algorithme va sélectionner un sommet $c_{i+1} \in border(T_i)$ tel que $B(c_{i+1})$ est maximum. On construit alors S_{i+1} qui contient les sommets de $D^k(c_{i+1})$ non-couverts par T_i, et les sommets de $out(c_{i+1}, covered(T))$.

On élimine ensuite les conflits : tant que deux sommets u et v dans S_{i+1} tels que $dist_G(u, v) > k$ et qu'il y a une chemin de u à v passant par des sommets hors de $covered(T_i) \cup S_{i+1}$ existent, on retire de S_{i+1} un des deux sommets. Si u et v sont dans $out(c_{i+1}, covered(T_i)$, ou que $S_{i+1} \subseteq covered(T_i)$ on ne peut pas continuer donc on sélectionne un autre sommet de $border(T)$ que c_{i+1} comme nouveau centre. S'il n'est plus possible de trouver un autre centre, l'algorithme échoue.

Une fois que les conflits sont éliminés, on ajoute une arête dans T_i entre S_{i+1} et $B(c_{i+1})$ pour construire T_{i+1}. Quand tous les sommets sont couverts le résultat retourné est une tree-decomposition de G.

Algorithme 6 : Disk-tree

Input : G un graphe, k un entier
Output : Une tree-decomposition de G ou ECHEC

```
1  u ← un sommet arbitraire de G
2  T ← ({u}, ∅)
3  C ← {u} // l'ensemble des centres possibles
4  tant que covered(T) ≠ V(G) faire
5      init :
6      si C = ∅ alors
7          retourner ECHEC
8      c ← un sommet de C
9      S ← composante connexe de c dans G[V(G) \ (covered(T) \ {c})]
10     S ← S ∩ D^k(c)
11     S ← S ∪ out(c, covered(T))
12     Reduction :
13     tant que il existe u, v en conflit dans S faire
14         si u ∉ border(T) ou v ∉ border(T) alors
15             si v ∈ border(T) ou (u ∉ border(T) et
                dist_G(c, u) ≥ dist_G(c, v)) alors
16                 S ← S \ {u}
17             sinon
18                 S ← S \ {v}
19
20     Mise à jour :
21     si S ⊊ covered(T) alors
22         T ← (V(T) ∪ {S}, E(T) ∪ {{B(c), S}})
23         C ← {u ∈ border(T) | ∀v ∈ border(T), profondeur(B(u)) ≥
              profondeur(B(v))}
24     sinon
25         C ← C \ {c}
26 retourner T
```

Cette décomposition est calculée en temps $O(n^4)$.

4.4.3 Amélioration proposée

Nous proposons une amélioration similaire à celle proposée pour bfs-layering (section 4.2.3). Nous pouvons voir qu'à la ligne 9, on choisit la composante connexe de c dans le sous graphe induit par $V(G)$ privé de tous les sommets déjà couverts sauf c. En terme de temps d'exécution et surtout de length du résultat, il est plus intéressant de construire un sac à partir de chaque composante connexe de $G[V(G) \setminus covered(T)]$ ayant un sommet

voisin de c.

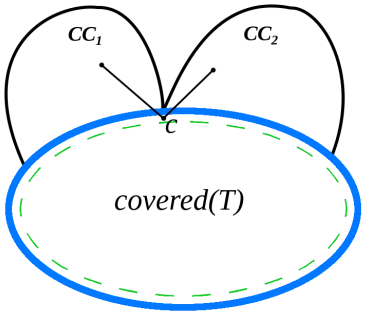

FIGURE 17 – c "voit" plusieurs composantes connexes de $G[V(G) \setminus covered(T)]$

Dans la figure 17, avec l'algorithme tel qu'il est présenté par Dourisboure et al., on essaye de construire un seul sac à partir de CC_1 et CC_2. Avec l'amélioration proposée, on construirait un sac pour CC_1 et CC_2 indépendamment, évitant ainsi de prendre des sommets inutiles (de la même manière que pour bfs-layering) dans le nouveau sac, c'est-à-dire des sommets d'une autre composante connexe, nous permettant ainsi de pouvoir obtenir une décomposition de meilleure length quand le cas se présente.

4.4.4 Implémentations

Dourisboure et Gavoille montrent que si disktree n'échoue pas, le résultat renvoyé est une tree-decomposition de length $2k$. Idéalement, nous serions capable de connaître rapidement la valeur minimum de k pour que l'algorithme termine. Notre idée est d'effectuer une recherche du plus petit k en effectuant plusieurs décompositions successives avec des k différents.

Nous avons donc deux implémentations ; la première est une recherche linéaire : à partir de $k = 1$ on essaye toutes les valeurs de k une par une jusqu'à trouver une décomposition ; la deuxième est une recherche par dichotomie : étant donné une borne inférieure A et une borne supérieure B sur k, on essaye de décomposer avec $k = \frac{A+B}{2}$, si la décomposition échoue on cherche entre A et $\frac{A+B}{2}$, sinon on cherche entre $\frac{A+B}{2}$ et B. Il est possible de préciser des bornes supérieures et inférieures pour accélérer la recherche, les valeurs par défaut étant $k = 1$ (inférieure) et $diam(G)$ (supérieure).

4.4.5 Rapport d'approximation dans les chordaux

Comme nous savons que si pour un k donné l'algorithme retourne une tree-decomposition, la décomposition obtenue a une length d'au plus $2k$, nous pouvons en déduire que s'il s'arrête pour $k \leq treelength(G)$ alors disk-tree est une 2-approximation. Nous n'avons pas de preuve que disk-tree s'arrête

quand $k = treelength(G)$, mais nous pouvons dans un premier temps nous limiter aux chordaux.

Lemma 7. *Soit c_i le centre du sac en construction pendant une itération de l'algorithme et S_i l'ensemble obtenu à la fin de la première étape de l'itération (avant d'éliminer les conflits). Si le graphe G en entrée est chordal, alors pour $k = 1$ il n'y a aucun conflit dans S.*

Démonstration. Soit T_i l'arbre de décomposition à l'étape i, $D^1(c)$ le disque de rayon 1 centré en c, et CC la composante connexe de c dans $G[V(G) \setminus (covered(T_i) \setminus \{c\}]$, on a donc $S = (CC \cap D^1(c)) \cup out(c, covered(T_i))$.

L'ensemble $out(c, covered(T_i))$ est un séparateur minimal d'un graphe chordal, il s'agit donc d'une clique. Supposons avoir $u \in S$ et $v \in out(u, covered(T) \cup S)$ tels que $d_G(u, v) > 1$. u et v sont donc en conflit dans S, alors il existe un chemin de u à v dont les sommets intermédiaires sont tous en dehors de $covered(T) \cup S$. On appelle z un de ces sommets intermédiaires. Comme u et v sont tous deux dans S, ils sont à distance 1 de c car ils sont soit dans $D^1(c) \cap CC$, soit dans $out(c, covered(T))$ qui est une clique contenant c. On peut alors trouver un cycle sans corde passant par z, u, c, v de longueur $dist_G(z, u) + dist_G(u, c) + dist_G(c, v) + dist_G(v, z) \geq 4$, une contradiction puisque le graphe est chordal. Il n'est donc pas possible de trouver deux sommets en conflit dans S.

Le sac ainsi formé sera donc ajouté à la décomposition et ne sera pas ignoré. □

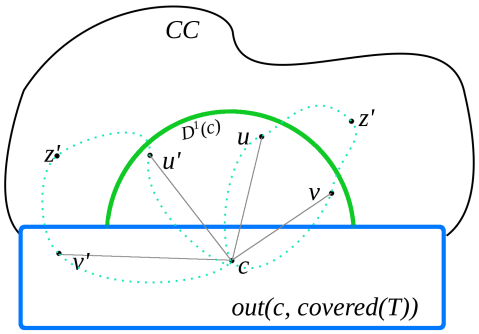

FIGURE 18 – Illustration de la preuve du lemme 5

Theorem 8. *Si G est un graphe chordal, la décomposition par disk-tree de G termine avec succès si $k = 1$ et disk-tree est donc une 2-approximation de la tree-length dans cette famille de graphe.*

Démonstration. Comme nous venons de le voir, quand $k = 1$, aucun centre n'est ignoré, donc il n'est pas possible d'épuiser la liste de centre potentiels tant qu'il existe un sommet non-couvert. Or, l'algorithme n'échoue qu'à

condition qu'à une étape donnée, tous les centres potentiels pour le nouveau sac aient échoué. L'algorithme peut donc progresser à chaque étape, jusqu'à avoir couvert tous les sommets. On obtient alors une tree-decomposition de length au plus 2. □

5 Benchmarks

Les tests suivants portent sur les 5 algorithmes présentés :
— Lex M
— Bfs layering
— Disk-tree (recherche de k par dichotomie)
— Disk-tree (recherche linéaire de k)
— Uncle-trees

Dans le cas de disk-tree, pour réduire l'espace de recherche (et donc diminuer le nombre de tentative de décomposition), nous avons besoin de bornes sur la valeur de k, on utilise la length de la décomposition obtenue par bfs-layering ; puisque c'est une 3-approximation de la treelength, soit x la length obtenue avec bfs-layering, on a $\frac{1}{3} \cdot x \leq tl(G) \leq x$. On utilise donc ces bornes pour calculer la tree-decomposition par disk-tree.

Nous avons généré 11 ensembles de graphes aléatoires de test différents, chacun contenant des graphes de taille (nombre de sommets) comprise entre 10 et 1000 par pas de 10, et 10 de chaque taille. Chaque set correspond à un ensemble généré par une méthode parmi les suivantes (qui seront explicitées en section 5.2) :
— Barabási-Albert, $k' = 2$
— Barabási-Albert, $k' = \log n$
— Barabási-Albert, $k' = \sqrt{n}$
— Erdős–Rényi, $p = \frac{\log n}{n}$
— Erdős–Rényi, $p = 1/n$
— Graphes chordaux
— Cycles
— Grilles
— Graphes de grande treelength
— Graphes série-parallèle
— Triangulations planaires

5.1 Protocole expérimental

Si le graphe généré n'est pas connexe, on ajoute une arête entre un sommet de la première composante connexe et un sommet de chaque autre composante connexe. Nous avons choisi cette méthode pour sa simplicité, et elle nous permet de rendre un graphe connexe sans augmenter le nombre de sommets ou briser les propriétés d'une famille de graphes (par exemple les

chordaux, si la méthode utilisée formait un cycle induit de taille supérieure à 3).

Le principe des tests effectués est de charger tous les graphes d'un set, puis d'exécuter chaque algorithme pour obtenir une décomposition de ces graphes. Pour chaque graphe et chaque algorithme, on mesure le temps d'exécution, la width, et la length du résultat. Pour le temps d'exécution, on impose un timeout de 30 secondes pour tous les algorithmes sauf pour les implémentations de disk-tree, qui ont besoin de plusieurs itérations pour trouver un résultat, qui disposent de 90 secondes. On obtient un fichier par algorithme contenant les résultats de toutes les décompositions.

Nous traitons alors les résultats obtenus pour les représenter de manière claire et lisible. Nous calculons d'abord la moyenne du temps d'exécution pour chaque taille de graphe (en ignorant les timeouts), puis la moyenne de la length obtenue pour chaque taille. Nous listons ensuite les meilleures valeurs de length obtenues pour chaque graphe parmi nos algorithmes, et comptons pour chaque meilleure length combien de fois les algorithmes ont obtenu cette length, et combien de fois ils ont été seuls à la trouver.

5.2 Génération de graphes

Nous allons ici présenter les différentes méthodes de génération adoptées pour construire nos sets de test.

5.2.1 Graphes aléatoires

Barabási-Albert Le principe de l'algorithme de Barabási-Albert pour générer des graphes aléatoires est le suivant[1] :

- Soit deux entiers k' et n
- Le graphe commence par un graphe G connexe aléatoire sur k' sommets
- Tant que le nombre de sommet n'est pas n :
  - On ajoute un sommet u à G
  - Pour chaque $v \in V(G)$ on ajoute une arête u, v avec une probabilité proportionnelle au degré de v (attachement préférentiel)

Dans notre cas, nous avons trois sets construits par cet algorithme en changeant la valeur de k' utilisée pour avoir différentes densités pour ces trois sets :

1. $k' = 2$
2. $k' = log(n)$
3. $k' = \sqrt{n}$

Le modèle de Barabási-Albert pour la génération de graphes aléatoires vise à simuler des réseaux réels tels que le web ou encore les réseaux sociaux dans lesquels nous pouvons retrouver ce mécanisme d'attachement préférentiel.

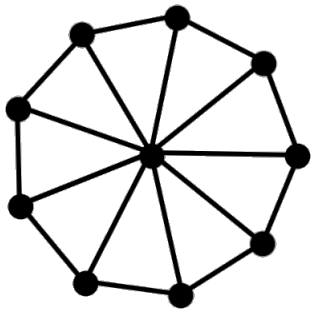

FIGURE 19 – Une triangulation planaire non-chordale

Erdős–Rényi Le principe du modèle Erdős–Rényi est de commencer avec n sommets, puis pour chaque paire de sommets du graphe une arête est ajoutée avec une probabilité p donnée en paramètre.

Nous construisons 2 sets suivant cette méthode, un avec $p = \frac{log(n)}{n}$ qui est le seuil au dessus duquel nous sommes presque sûrs que le graphe obtenu sera connexe, et un avec $p = \frac{1}{n}$ qui donne des graphes très peu connexes.

Triangulations planaires Intuitivement, un graphe planaire est un graphe que l'on peut représenter sur le plan sans que les arêtes ne se croisent. On appelle triangulation planaire un graphe planaire dont toutes les faces (sauf la face extérieure) sont des triangles.

Il s'agit à nouveau d'une fonction de génération implémentée dans SageMath [15]. L'idée est de partir d'un arbre aléatoire qui est progressivement *fermé*, c'est à dire que des arêtes sont ajoutées entre des sommets jusqu'à ce que le graphe soit triangulé.

Il est important de rappeler qu'il ne s'agit pas d'une triangulation au sens des graphes chordaux ; une triangulation planaire n'est pas nécessairement un graphe chordal.

5.2.2 Graphes de treelength connue

Nous connaissons une valeur exacte pour la treelength des graphes suivants. Cela nous permet de comparer la length obtenue par nos algorithmes à la vraie treelength du graphe, et donc de savoir si l'approximation est bonne.

Graphes chordaux La fonction de génération de graphes chordaux aléatoires est implémentée dans SageMath [15]. Elle utilise l'algorithme de Oylum Şeker [16] qui exploite le fait qu'un graphe chordal peut être vu comme le graphe d'intersection de sous-arbres d'un arbre :

— Générer un arbre aléatoire T sur n sommets

— Générer $\{T_1, ..., T_n\}$ n sous-arbres de T non vides
— Retourner le graphe d'intersection de $\{V(T_1), ..., V(T_n)\}$

Nous savons que les graphes chordaux sont les graphes de treelength 1, nous avons donc une valeur exacte pour la treelength.

Cycles Une méthode dans SageMath [15] permet de générer des cycles. L'algorithme est trivial et n'a donc pas besoin d'être détaillé ici.

Comme tous les cycles de taille n sont les mêmes, les tests sur ce set ne sont lancés que sur un graphe par taille. On a donc des cycles de taille comprise entre 10 et 1000 par pas de 10.

Comme vu précédemment la treelength d'un cycle sur n sommets est égale à $\lceil \frac{n}{3} \rceil$[8].

Grilles L'idée est d'utiliser la méthode implémentée dans SageMath[15] pour générer une grille de taille $i \times j$. On choisit donc aléatoirement i et j tels que $i \cdot j = n$, puis on construit une grille de dimensions $i \times j$.

Comme la treelength d'une grille est égale à la longueur de son plus petit côté, nous avons là aussi une valeur exacte sur la length.

5.2.3 Graphes avec borne inférieure connue

Graphes de grande treelength Nous avons une méthode simple permettant d'obtenir des graphes dont la treelength est élevée basé sur le fait que la treelength d'un graphe est au moins le tiers de la taille de son plus grand cycle induit. Pour obtenir un graphe de length au moins k, on part d'un cycle de taille $3k$, puis on sélectionne deux sommets u et v aléatoirement dans le graphe que l'on relie par un chemin de longueur au moins $dist_G(u, v)$ pour ne pas créer de corde dans le cycle original.

Algorithme 7 : Grande treelength

Input : n et k deux entiers
Output : Un graphe G tel que $treelength(G) \geq k$

```
G ← cycle de taille 3k
tant que |G| < n faire
    Choisir u, v ∈ V(G)
    Ajouter un chemin P_{u,v} de longueur au moins dist_G(u, v)
retourner G
```

On choisit un entier $k \in [\frac{n}{4}, \frac{n}{3}]$ aléatoirement, on aura donc $treelength(G) \geq k$.

Graphes Série-parallèle Un graphe série-parallèle est un graphe planaire définit comme suit :

— Une arête (s, t) est un graphe série parallèle

— Soient deux graphes série-parallèle G et H ayant chacun deux sommets distingués, respectivement s_G, t_G et s_H, t_H :
  — Le graphe obtenu en fusionnant t_G et s_H est un graphe série-parallèle (composition en série)
  — Le graphe obtenu en fusionnant t_G avec t_H et s_G avec s_H est un graphe série-parallèle (composition parallèle)

Notre algorithme de génération suit cette définition : on part d'une arête qu'on ajoute à une collection de graphes série-parallèle générés, puis tant que la taille du graphe construit est inférieure à la taille désirée, on choisit dans notre collection un graphe que l'on compose en série ou en parallèle (aléatoirement) avec le graphe construit jusque là. À chaque étape, le nouveau graphe obtenu est ajouté à la collection.

5.3 Résultats

Nous allons maintenant présenter les résultats obtenus dans le but de comparer les performances des algorithmes en termes de temps d'exécution et de length.

Avant de présenter les résultats de chaque famille, nous donnons la distribution du degré et du diamètre des graphes de la famille, c'est-à-dire pour chaque valeur de degré (ou diamètre) combien de graphes ont ce degré (ou diamètre). Sur les figures, le degré (diamètre) est en abscisse rangé par ordre croissant.

Nous commençons par regarder le temps moyen d'exécution et la length moyenne pour chaque taille de graphe, puis nous comparons nos algorithmes à un meilleur algorithme virtuel, qui correspond à un hypothétique algorithme qui aurait, pour chaque graphe, toujours trouvé la meilleure length (parmi les résultats des 4 algorithmes) et obtenu le meilleur temps. Enfin, nous observons pour chaque length obtenue, combien de fois chaque algorithme a obtenu cette length quand elle était la meilleure, et combien de fois ils ont été seuls à trouver cette length (figures en annexe, section 8.1).

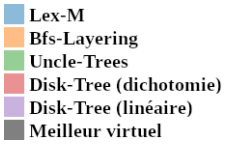

FIGURE 20 – Légende des figures ci-après

5.3.1 Graphes aléatoires

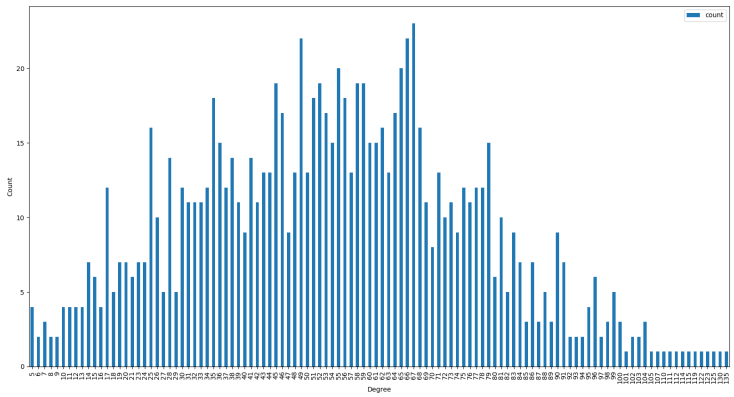

FIGURE 21 – Distribution du degré

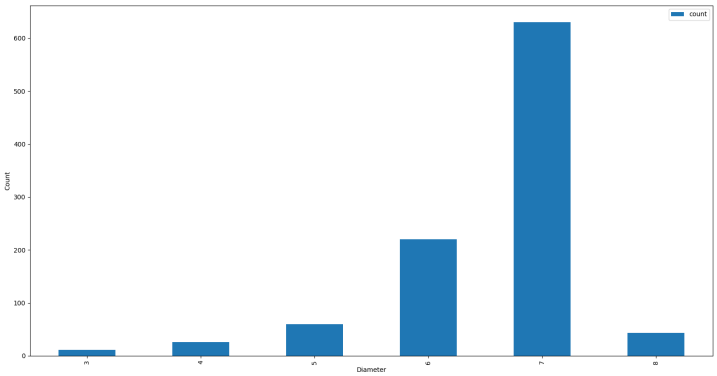

FIGURE 22 – Distribution du diamètre

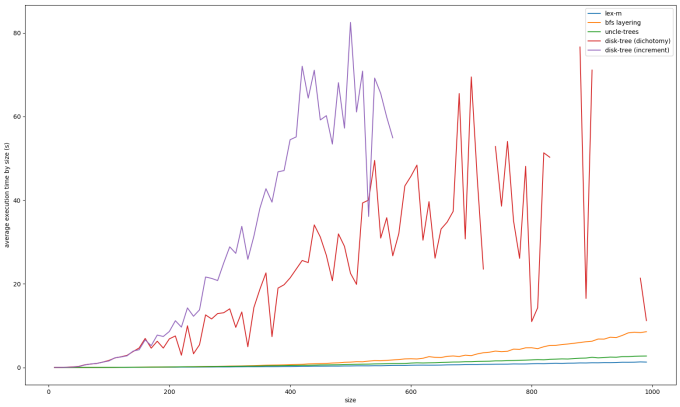

FIGURE 23 – Temps d'exécution moyen en fonction de la taille

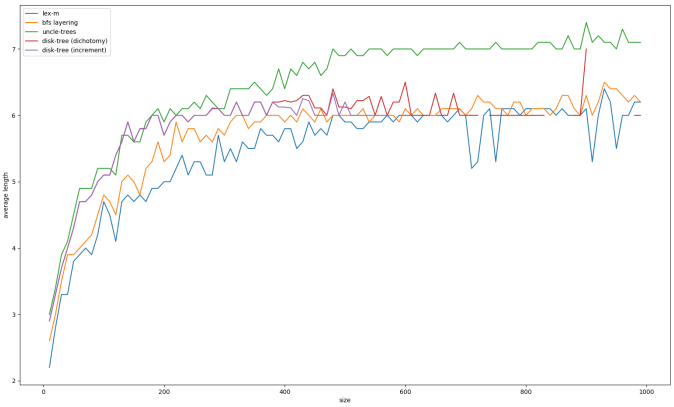

FIGURE 24 – Length moyenne en fonction de la taille

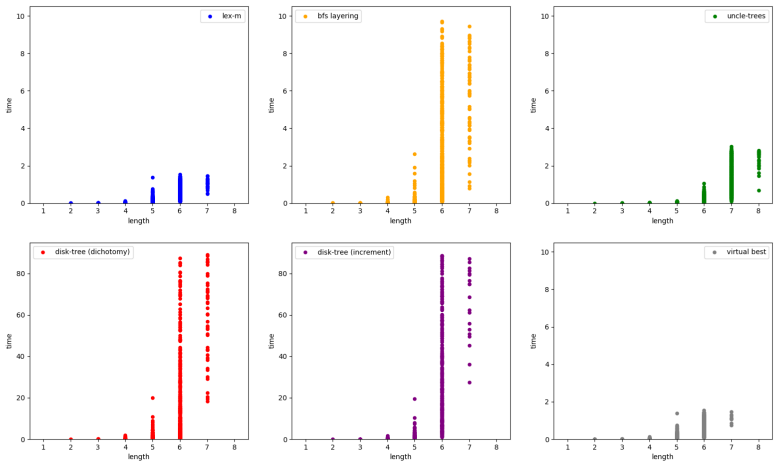

FIGURE 25 – Temps d'exécution en fonction de la length calculée

Barabási-Albert, $k = 2$ Tout d'abord, il n'est pas surprenant de voir (fig. 23) que les deux implémentations de disk-tree sont les plus lentes ; en effet, pour chaque décomposition, il est lancé plusieurs fois afin de trouver la valeur optimale de k.

La recherche de k par dichotomie est ici plus rapide que la recherche linéaire, qui atteint par ailleurs le timeout de 90s sur tous les graphes de taille supérieure à environ 600 sommets. Cet avantage de la recherche par dichotomie correspond bien à ce que nous attendions.

Le temps d'exécution des trois autres algorithmes est très proche, bfs-layering étant légèrement plus lent. Une explication possible est le diamètre des graphes qui reste bas (fig. 22) malgré l'existence de graphes avec un grand nombre de sommets, à chaque étape l'algorithme contracte des sommets et cherche des composantes connexes qui sont donc très grandes (comme le nombre maximum de couches est égale au diamètre du graphe), d'où la lenteur comparée aux autres algorithmes ; pour uncle-trees la figure 21 nous montre que la majorité des graphes a un degré compris entre 30 et 80. Or, nous pouvons voir que lorsque le degré est petit comparé au nombre de sommets du graphe, lors de la construction des sacs il y a relativement peu de sommets voisins à parcourir pour trouver les oncles et juniors. Pour lex-M, il y a peu d'opérations "longues" lors de la triangulation et du calcul de la décomposition.

En ce qui concerne la length calculée (fig. 24) uncle-trees obtient toujours la plus élevée. D'autre part, jusqu'à environ 400 sommets, la décomposition donnée par bfs-layering est de length plus petite que celle donnée par disk-tree. Malgré le fait qu'il s'agisse du même algorithme, les deux implémentations de disk-tree ne sont pas toujours confondues, mais cette différence

est explicable : comme il s'agit de valeurs moyennes, si les deux implémentations n'atteignent pas le timeout sur les mêmes graphes, les moyennes ne seront pas les mêmes. Par ailleurs, comme nous l'avons vu dans la figure 23, l'implémentation avec recherche linéaire ne donne plus de résultat après 400 sommets.

Enfin, nous pouvons voir que lex-M donne en moyenne une meilleure length que les autres algorithmes, ce qui est confirmé par le tableau 1 : c'est bien lui qui trouve la meilleure length le plus souvent, et il est plus souvent seul à trouver cette length que les autres algorithmes. En figure 25 nous observons aussi que Lex-M est presque identique au virtuel meilleur algorithme, ce qui est logique puisqu'il est le plus rapide et donne la meilleure length le plus souvent.

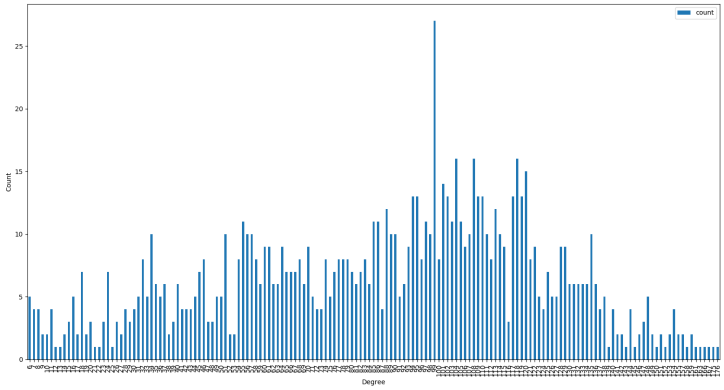

FIGURE 26 – Distribution du degré

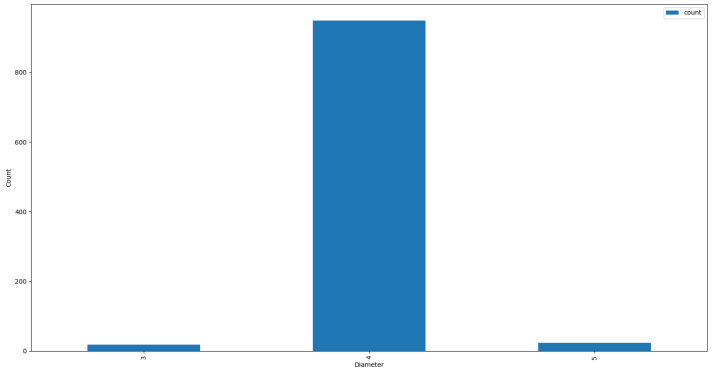

FIGURE 27 – Distribution du diamètre

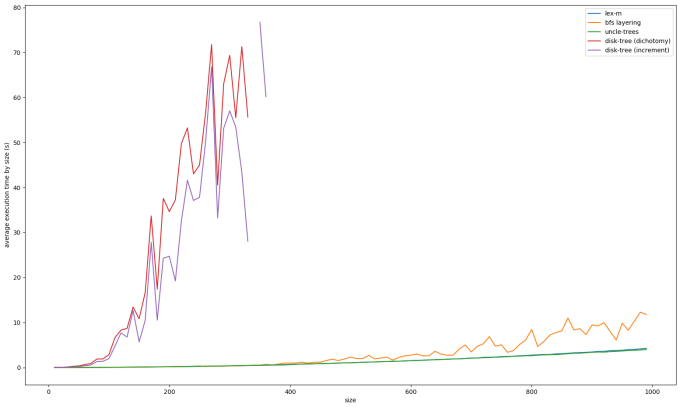

FIGURE 28 – Temps d'exécution moyen en fonction de la taille

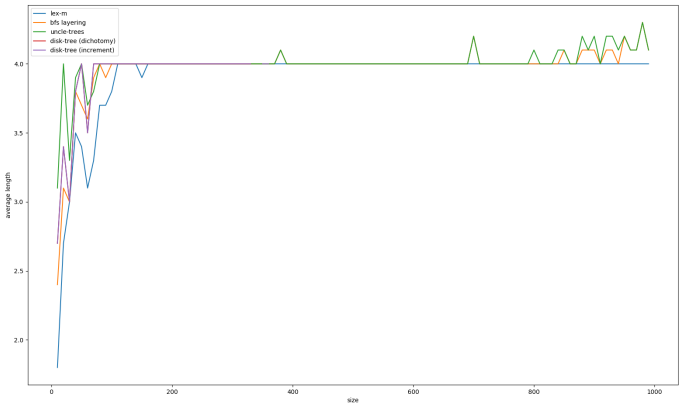

FIGURE 29 – Length moyenne en fonction de la taille

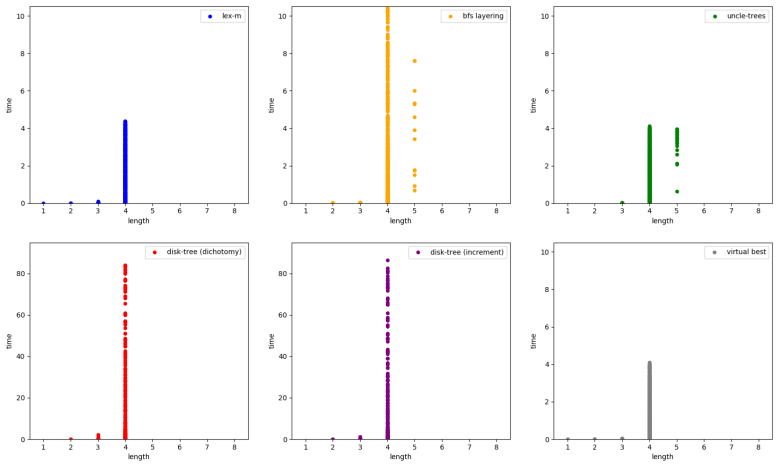

FIGURE 30 – Temps d'exécution en fonction de la length calculée

Barabási-Albert, $k = log(n)$ Cette fois-ci, l'implémentation par dichotomie de disk-tree est plus lente que la version linéaire. La figure 27 nous montre que la majorité des graphes ont un diamètre de 4, dans ce cas là il est possible que la recherche linéaire trouve la bonne valeur de k en moins d'itérations qu'en cherchant par dichotomie, ce qui explique la différence entre les deux implémentations.

Par ailleurs, le timeout est systématiquement atteint à partir de 350 sommets environ pour les deux implémentations sur ce set. Comme les graphes ont un presque tous un diamètre de 4 (fig. 27) peu importe le nombre de sommets, les graphes sont très connexes (les sommets ont un grand voisinage), ce qui peut expliquer pourquoi disk-tree est si lent : en effet, la présence de nombreux voisins implique que le $border(T)$ est grand, donc qu'il y a beaucoup de choix pour le prochain centre. Il peut donc y avoir beaucoup d'itération de la boucle principale pour un k donné, avec beaucoup de conflits à supprimer, ce qui allonge le temps d'exécution (même si on ne teste que 2 valeurs de k).

Pour les autres algorithmes, on observe le même comportement en terme de temps d'exécution que sur le set précédent.

Concernant la length, nous observons (fig. 29) que sur les graphes sur peu de sommets (jusqu'à environ 80 sommets), uncle-trees obtient en général une plus grande length que les autres, et lex-M trouve la meilleure length. Entre 80 et 800 sommets, tous les algorithmes (lorsqu'ils n'atteignent par le timeout) trouvent en moyenne la même length. Au delà, la moyenne de la length trouvée par uncle-trees augmente légèrement. Au vu de la figure 27 nous pouvons supposer que le début des courbes (jusqu'à 80 sommets) correspond aux graphes de diamètre 3, puis ceux de diamètre 4 jusqu'à 800

sommets et enfin ceux de diamètre 5 pour la fin.

La figure 30 nous que lex-M est à nouveau le plus proche du meilleur algorithme. Nous pouvons par ailleurs voir sur le tableau 2 que lex-M trouve dans presque tous les cas la meilleure length, mais qu'il est moins souvent le seul à trouver cette meilleure length. Nous pouvons expliquer la rapidité de lex-M par le fait que comme les graphes sont très connexes, il n'a pas beaucoup d'arêtes à ajouter pour les trianguler.

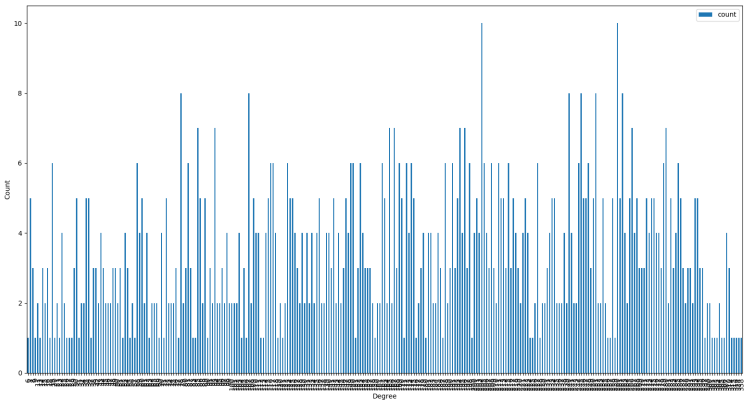

FIGURE 31 – Distribution du degré

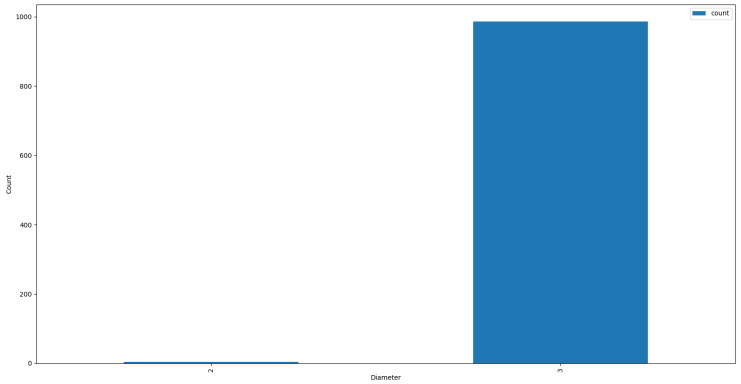

FIGURE 32 – Distribution du diamètre

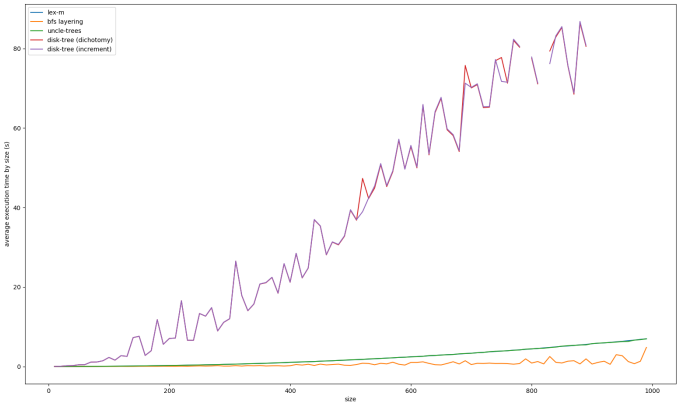

FIGURE 33 – Temps d'exécution moyen en fonction de la taille

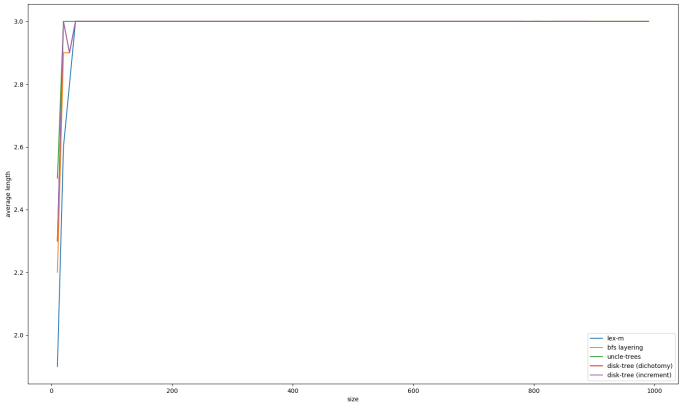

FIGURE 34 – Length moyenne en fonction de la taille

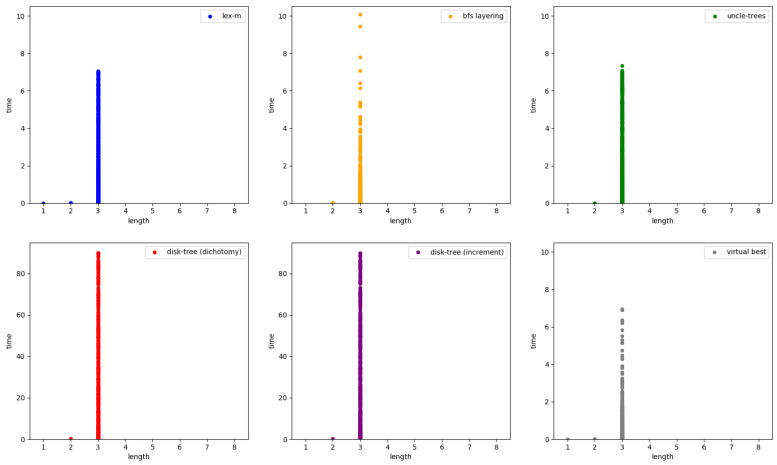

FIGURE 35 – Temps d'exécution en fonction de la length calculée

Barabási-Albert, $k = \sqrt{n}$ Ici, les deux implémentations de disk-tree prennent en moyenne le même temps d'exécution (fig. 33). Contrairement aux observations précédentes, cette fois c'est bfs-layering qui est le plus rapide parmi les 3 autres. Cette rapidité est probablement dûe au faible diamètre des graphes : en effet, bfs-layering calcule un layering-tree sur au plus 3 couches, les opérations longues (recherche de composantes connexes) sont donc peu répétées. En terme de length (fig. 34), il y a de légères différences sur les petits graphes (jusqu'à 30 sommets environ), lex-M semblant être un petit peu meilleur, mais en général tous les algorithmes trouvent une length de 3 sur ces exemples.

Contrairement au set précédent, malgré le fait que les graphes aient majoritairement un diamètre de 3 (fig. 32), le degré des graphes varie fortement et ne semble pas dépendre de la taille (fig. 31). Le fait qu'il n'y a pas une majorité de graphes de grand degré explique pourquoi disk-tree parvient à décomposer des "grands" graphes en moins de 90s (fig. 33) ; il y a moins de centre potentiels à tester et donc moins de conflits.

La figure 35 nous confirme que les trois algorithmes "rapides" (bfs-layering, lex-M et uncle-trees) sont proches du meilleur, tous ayant quelques graphes pour lesquels le temps d'exécution est plus lent que le meilleur.

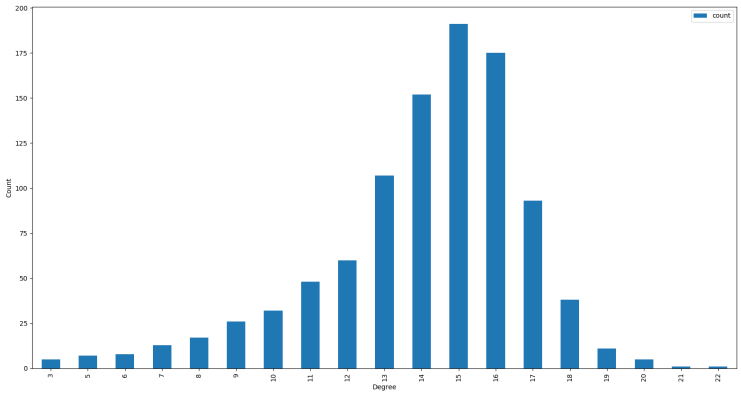

FIGURE 36 – Distribution du degré

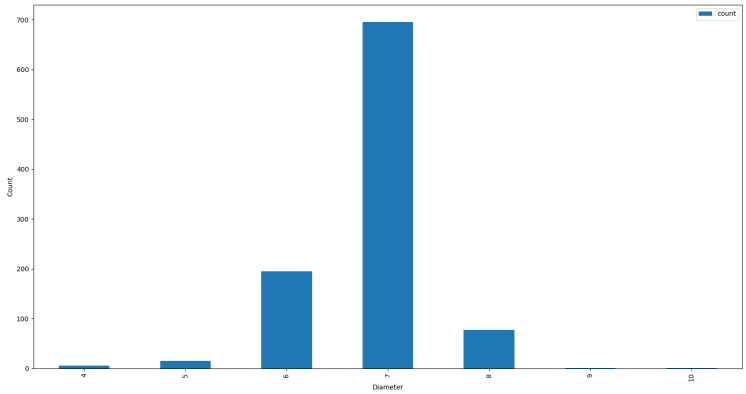

FIGURE 37 – Distribution du diamètre

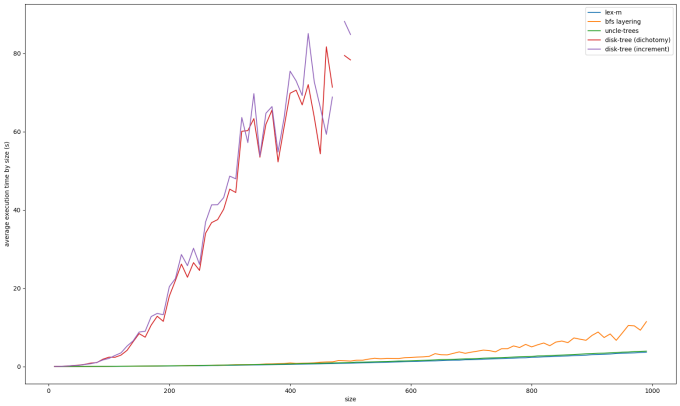

FIGURE 38 – Temps d'exécution moyen en fonction de la taille

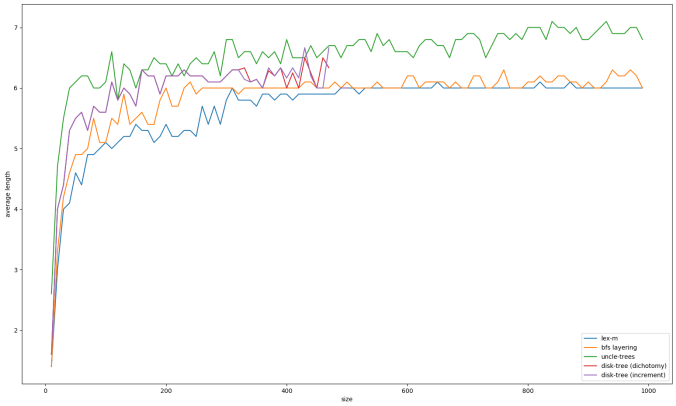

FIGURE 39 – Length moyenne en fonction de la taille

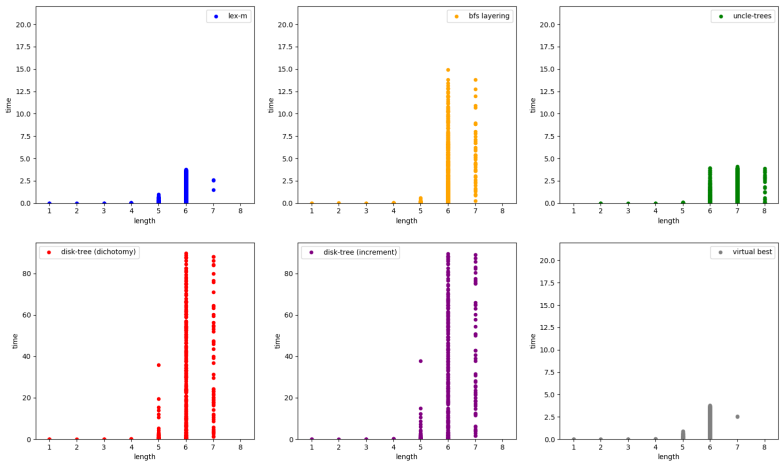

FIGURE 40 – Temps d'exécution en fonction de la length calculée

Erdős–Rényi, $p = \frac{log(n)}{n}$ Sur cet ensemble de graphes, les deux implémentations de disk-tree sont à nouveau très proches en temps d'exécution (fig. 38) et atteignent toutes les deux le timeout à partir d'environ 500 sommets. C'est à nouveau lex-M et uncle-trees qui sont les deux algorithmes les plus rapides. Bfs-layering est un peu plus lent, ce qui peut être expliqué par les diamètres un peu plus élevés (fig. 37) que précédemment, donnant des layering-trees avec plus de couches et donc plus de composantes connexes à calculer.

Nous observons sur le graphique 39 un "classement" très net des lengths calculées : lex-M obtient la meilleure length, bfs-layering trouve une length un peu plus grande, puis disk-tree en trouve une encore légèrement plus grande, et enfin uncle-trees trouve la plus grande length. Ce classement correspond à nos attentes au vu des rapports d'approximation de chaque algorithme, cependant il est étonnant de voir que disk-tree obtient en moyenne une plus grand length que lex-M alors qu'il est supposé être une 2-approximation, le dernier étant une 3-approximation.

Nous en avons la confirmation sur la figure 40 : encore une fois lex-M et le meilleur virtuel sont presque identiques. Uncle-trees est le seul algorithme à trouver des décompositions de length 8. Bfs-layering est plus lent, et trouve beaucoup plus souvent une length de 7 que lex-M alors que le tableau 4 montre qu'il n'y a que deux graphes pour lesquels la meilleure length calculée est 7.

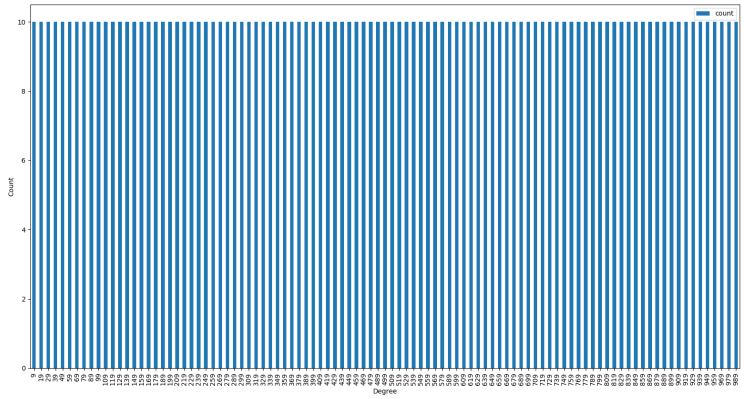

FIGURE 41 – Distribution du degré

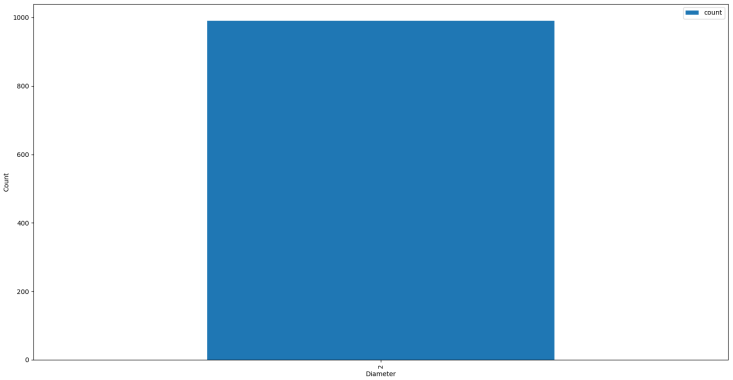

FIGURE 42 – Distribution du diamètre

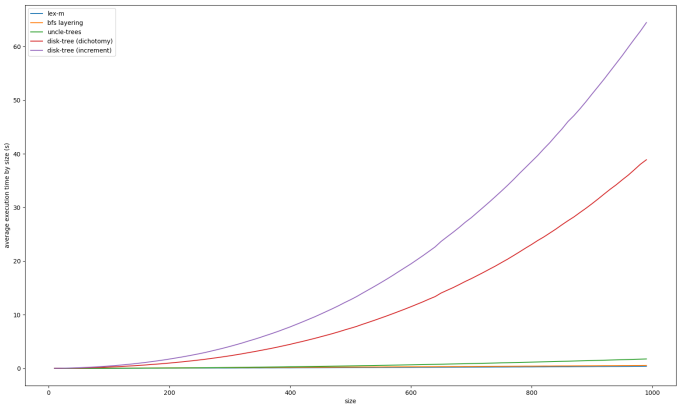

FIGURE 43 – Temps d'exécution moyen en fonction de la taille

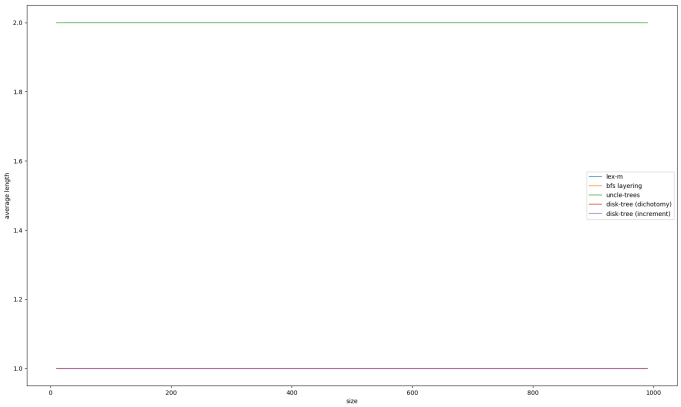

FIGURE 44 – Length moyenne en fonction de la taille

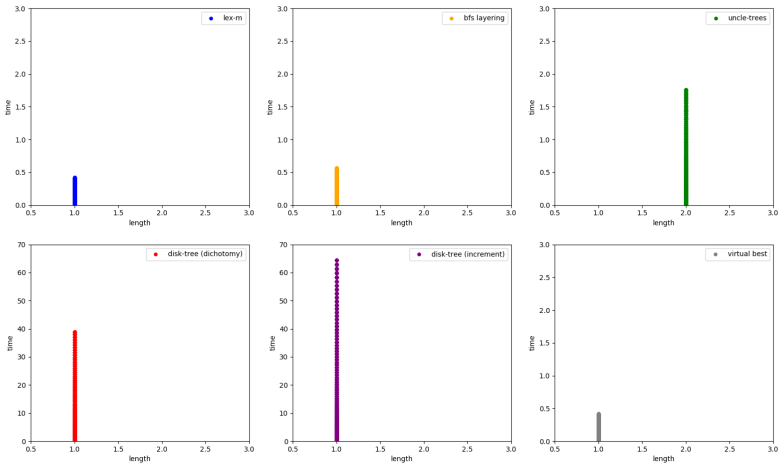

FIGURE 45 – Temps d'exécution en fonction de la length calculée

Erdős–Rényi, $p = \frac{1}{n}$ Tout d'abord, nous remarquons ici qu'il y a 10 graphes par degré (fig. 41) et que le diamètre de tous les graphes du set est égal à 2 (fig. 42). La génération de graphes Erdős–Rényi avec $p = \frac{1}{n}$ donne en fait des graphes avec très peu d'arêtes. Comme nous rendons connexes manuellement les graphes en ajoutant une arête entre un sommet et toutes les autres composantes connexes, nous avons accidentellement généré des arbres (nous avons alors vérifié que tous les graphes sont des arbres). De plus, comme leur diamètre est 2, il s'agit d'étoiles.

Nous pouvons donc voir (fig. 43) que disk-tree avec recherche linéaire est bien plus rapide que la recherche par dichotomie, ce qui n'est pas surprenant étant donné que la treelength des graphes est égale à 1 : il n'y a qu'une itération de faite. Les trois autres algorithmes prennent à peu près le même temps, uncle-trees étant légèrement plus lent car il y a un sommet de très haut degré.

Sur la figure 44, tous les algorithmes sont confondus à part uncle-trees, ce dernier trouvant systématiquement une length de 2 alors que les autres trouvent 1. L'explication est que la valeur de k choisie pour uncle-trees est au moins 3 car il s'agit de la taille minimum d'un cycle. Peu importe l'uncle-tree généré à partir de l'étoile, la décomposition contiendra toujours un sac contenant tous les sommets du graphe. En effet, pour chaque sommet de l'uncle-tree on prend les enfants de ce sommet dans le sac correspondant et (dans ce cas) le chemin entier jusqu'à la racine (puisque $k = 3$), d'où la length de 2.

Nous pouvons voir figure 45 que bfs-layering et lex-M sont presque identiques au meilleur virtuel tandis que uncle-trees est à la fois plus lent et donne une moins bonne length. Les implémentations de disk-tree, elles, donnent

toujours une bonne approximation mais ont un temps d'exécution beaucoup plus long.

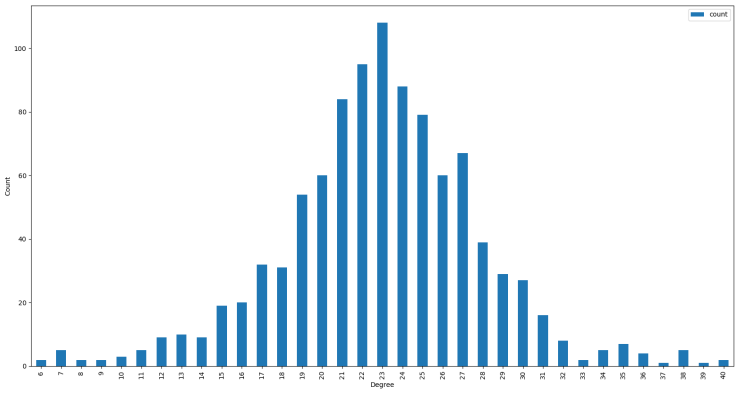

FIGURE 46 – Distribution du degré

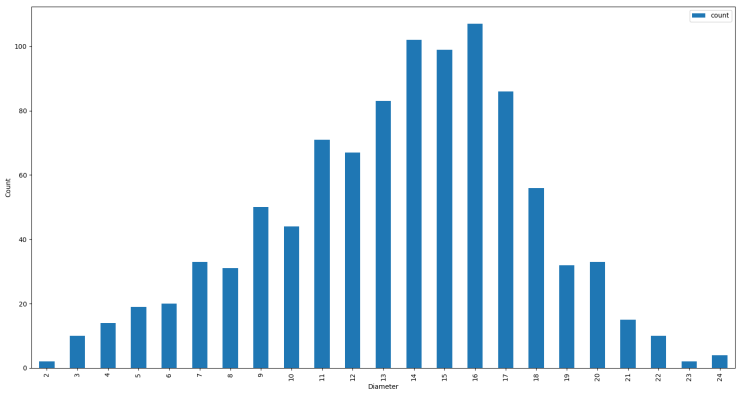

FIGURE 47 – Distribution du diamètre

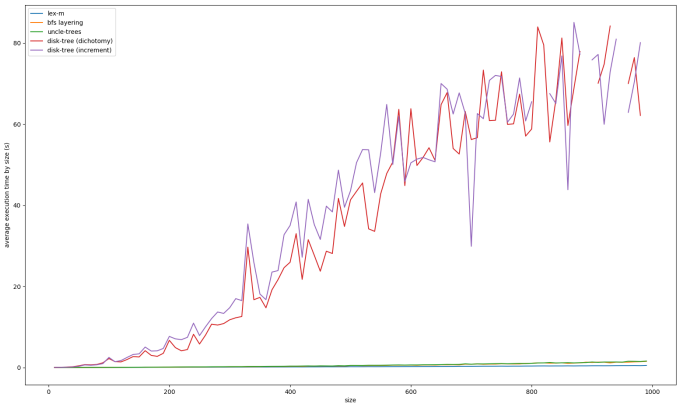

FIGURE 48 – Temps d'exécution moyen en fonction de la taille

FIGURE 49 – Length moyenne en fonction de la taille

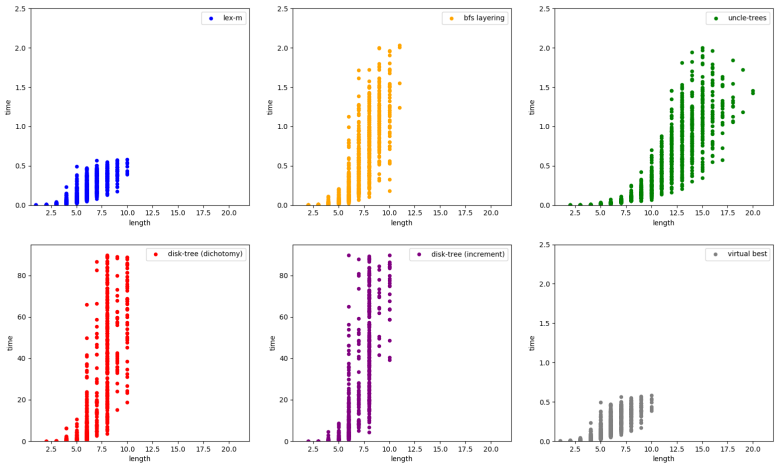

FIGURE 50 – Temps d'exécution en fonction de la length calculée

Triangulations planaires Ici, les deux implémentations de disk-tree ont toutes les deux un temps d'exécution moyen très proche (fig. 48), il en est de même pour les trois autres algorithmes. Pour les trois algorithmes "rapides" cette rapidité de calcul peut être expliquée par le fait que les graphes sont des graphes planaires et ont, par conséquent, "peu" d'arêtes par rapport au nombre de sommets (la formule d'Euler nous dit que $|E(G)| \leq 3|V(G)| - 6$). Les parcours en largeur (ou en profondeur pour uncle-trees) prennent donc peu de temps.

Nous pouvons observer le même "classement" que précédemment pour la length obtenue (fig. 49) : uncle-trees obtient les plus grandes lengths, puis disk-tree, bfs-layering et enfin lex-M obtient les meilleures. Comme précédemment, la seule différence par rapport aux résultats théoriques est le fait que disk-tree ne trouve pas la meilleure length.

Cette observation est renforcée par le tableau 11 : lex-M trouve la meilleure length dans tous les graphes du set (il y en a 990 au total), ce qui implique qu'aucun autre algorithme ne trouve de meilleure length seul. De plus, il est souvent seul à trouver cette meilleure length. Bfs-layering trouve assez souvent la meilleure length, les autres ne la trouvent qu'occasionnellement.

C'est donc lex-M qui est à nouveau le plus proche du meilleur algorithme (fig. 50), on notera toutefois que bfs-layering et disk-tree obtiennent des résultats dans la même plage de lengths, contrairement à uncle-trees qui trouve des lengths jusqu'à 20 alors que le maximum chez les autres algorithmes est 10.

5.3.2 Graphes de treelength connue

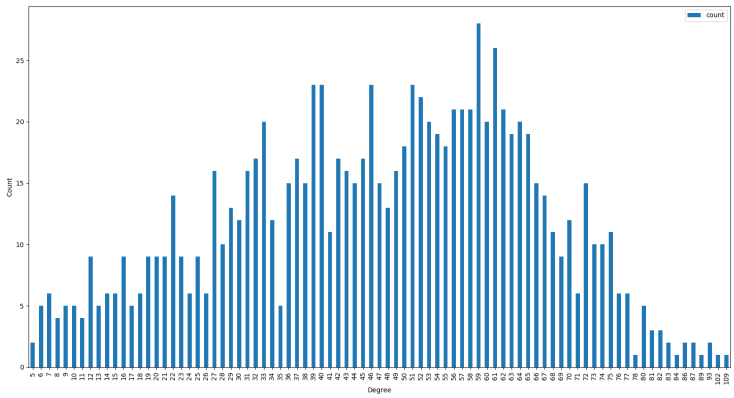

FIGURE 51 – Distribution du degré

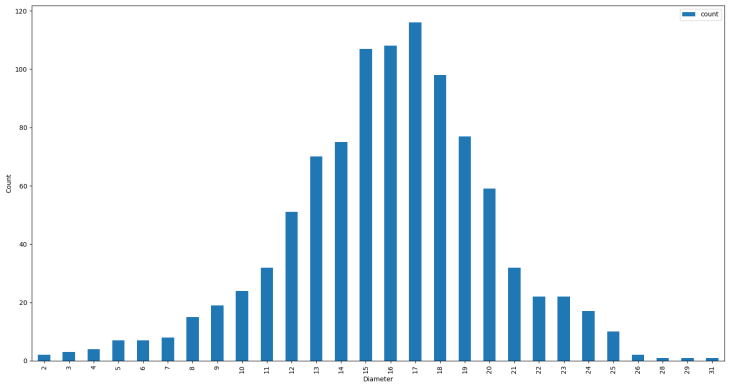

FIGURE 52 – Distribution du diamètre

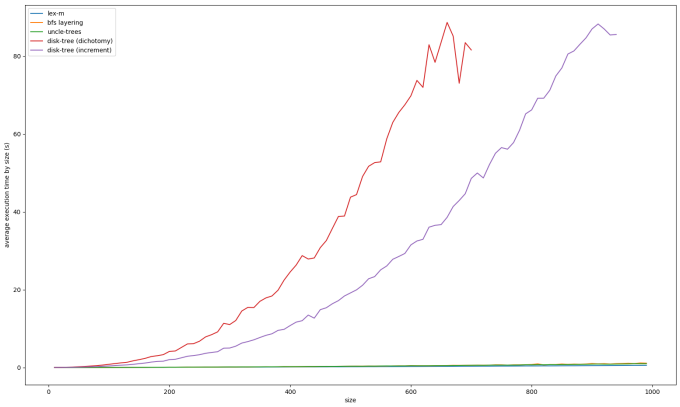

FIGURE 53 – Temps d'exécution moyen en fonction de la taille

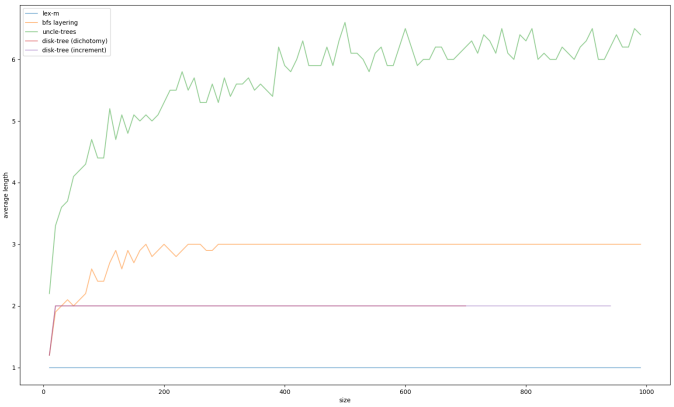

FIGURE 54 – Length moyenne en fonction de la taille

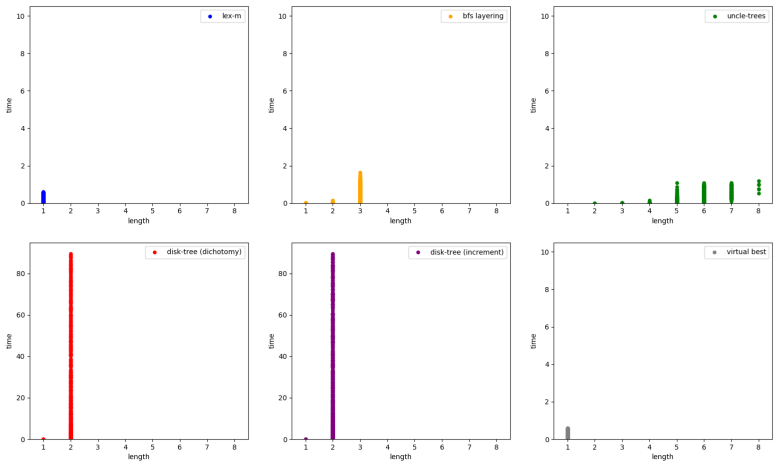

FIGURE 55 – Temps d'exécution en fonction de la length calculée

Graphes chordaux Comme pour les arbres, l'implémentation de disk-tree par dichotomie est plus lente que celle par recherche linéaire (fig. 53) car la treelength des graphes est égale à 1, donc la recherche linéaire trouve k en une seule itération. Les autres algorithmes prennent approximativement le même temps.

Pour la length, on a à nouveau le même type de "classement" que précédemment (fig. 54) : lex-M donne une décomposition optimale (de length 1), disk-tree obtient en général une décomposition de length 2, bfs-layering obtient des décompositions de length 3 (quelques unes de length 1 et 2), et uncle-trees obtient toujours les plus grandes lengths. Ce qui correspond aux résultats attendus dans cette classe de graphes : nous avons vu que lex-M renvoie une décomposition optimale du graphe triangulé, ce qui en fait un algorithme exact dans les chordaux. Nous avons aussi prouvé que disk-tree et bfs-layering (avec l'amélioration proposée) sont respectivement une 2-approximation et une 3-approximation de la treelength.

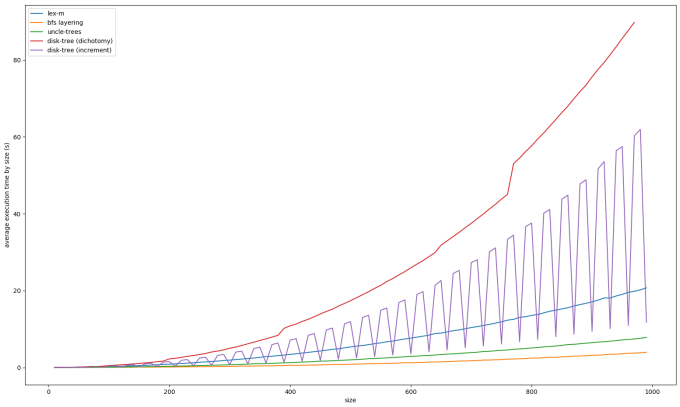

FIGURE 56 – Temps d'exécution moyen en fonction de la taille

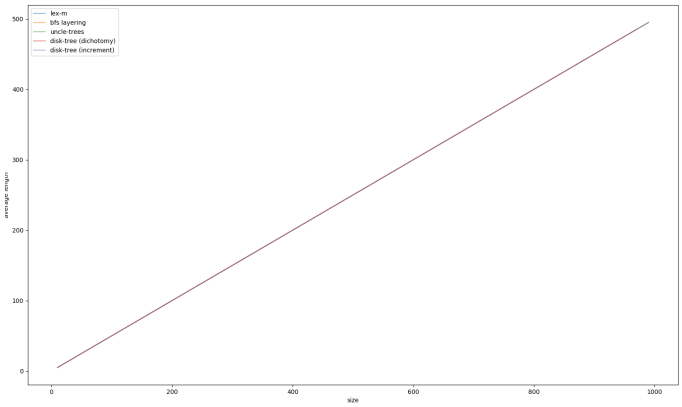

FIGURE 57 – Length moyenne en fonction de la taille

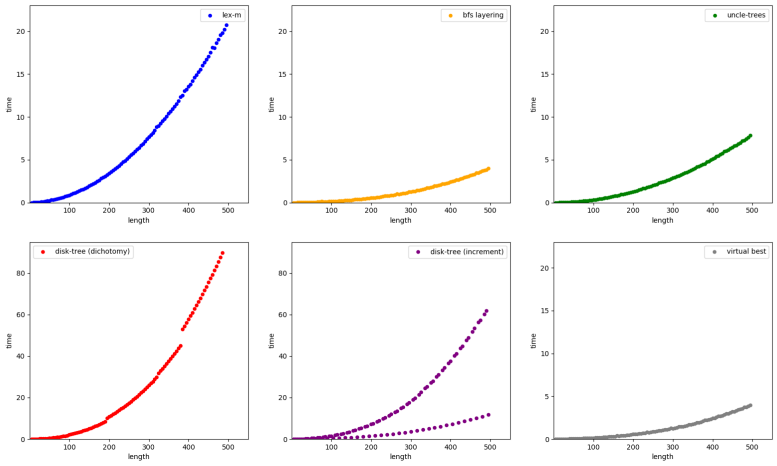

FIGURE 58 – Temps d'exécution en fonction de la length calculée

Cycles L'allure de la courbe du temps de calcul moyen de disk-tree par recherche linéaire en figure 56 saute immédiatement aux yeux. Cela vient du fait que nous utilisons une borne inférieure égale à $\lfloor \frac{n}{3} \rfloor$ pour la décomposition. Comme il s'agit de cycles, la treelength est égale à $\lceil \frac{n}{3} \rceil$, donc lorsque la taille est multiple de 3, la borne inférieure est égale à la treelength, la décomposition n'a besoin de tester qu'une seule valeur de k. Si le nombre de sommet n'est pas multiple de 3, on testera soit 2 valeurs de k, soit 3 (en fonction de la valeur de $n \ mod \ k$).

Nous pouvons d'ailleurs voir (fig. 57) que la length obtenue est égale à $\frac{n}{2}$ pour chaque algorithme, alors que la treelength est égale à $\lceil \frac{n}{3} \rceil$. Cette différence provient du fait qu'aucun des algorithme n'implémente la méthode pour obtenir une décomposition de length minimum d'un cycle, on a donc une length légèrement moins bonne que l'optimale.

Nous pouvons voir sur la figure 57 que tous les algorithmes trouvent la même length pour chaque graphe, la seule différence est leur temps d'exécution. Dans ces graphes, bfs-layering est l'algorithme le plus performant (fig. 58) puisqu'il trouve la meilleure décomposition en moins de temps que les autres.

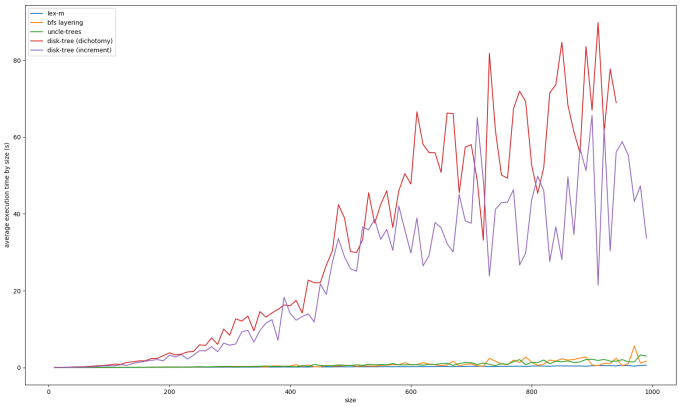

FIGURE 59 – Temps d'exécution moyen en fonction de la taille

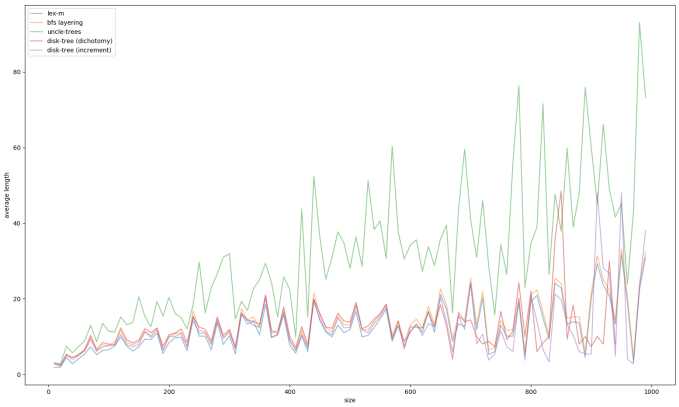

FIGURE 60 – Length moyenne en fonction de la taille

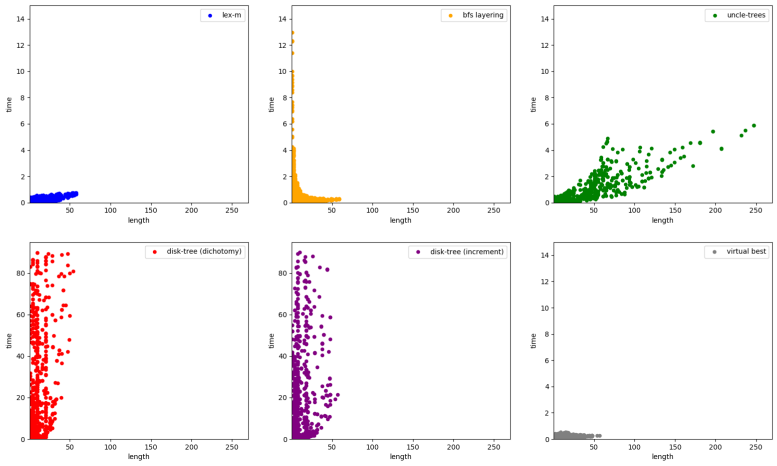

FIGURE 61 – Temps d'exécution en fonction de la length calculée

Grilles Pour les grilles, nous observons figure 59 que l'implémentation de disk-tree avec recherche linéaire est plus rapide que l'implémentation par dichotomie. Cela vient vu fait que, comme nous savons que la treelength d'une grille est la longueur de son plus petit côté, nous avons une borne inférieure égale à la treelength, donc si disk-tree termine pour $k = treelength(G)$ la version par recherche linéaire trouve k en une seule itération.

En ce qui concerne la length obtenue, tous les algorithmes obtiennent en moyenne des lengths similaires (fig. 60), à l'exception de uncle-trees qui

trouve toujours des décompositions de length plus élevée que les autres, ce qui n'est pas surprenant car les grilles contiennent de grands cycles induits, donnant lieu à une valeur de k élevée.

À nouveau, c'est lex-M qui est proche du meilleur algorithme (fig. 61), bfs-layering n'étant pas très loin mais plus lent sur certains graphes. Le tableau 8 nous montre que quand la treelength est 1 (c'est-à-dire quand les graphes sont des chemins), tous les algorithmes trouvent une décomposition optimale. Dans le cas général, Lex-M trouve la meilleure length souvent, et bien souvent il est seul à trouver cette length. Dans quelques cas, disk-tree trouve la meilleure length seul. Nous pouvons aussi noter que dans ce cas il arrive que uncle-trees soit seul à trouver la meilleure length, ce qui n'arrive que très rarement (voire jamais) dans les autres sets.

5.3.3 Graphes avec borne inférieure connue

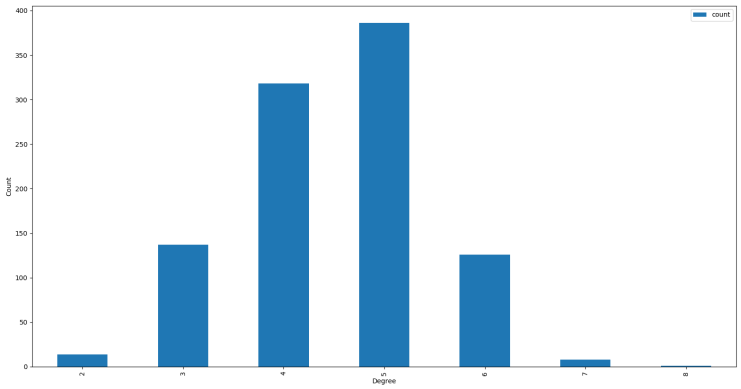

FIGURE 62 – Distribution du degré

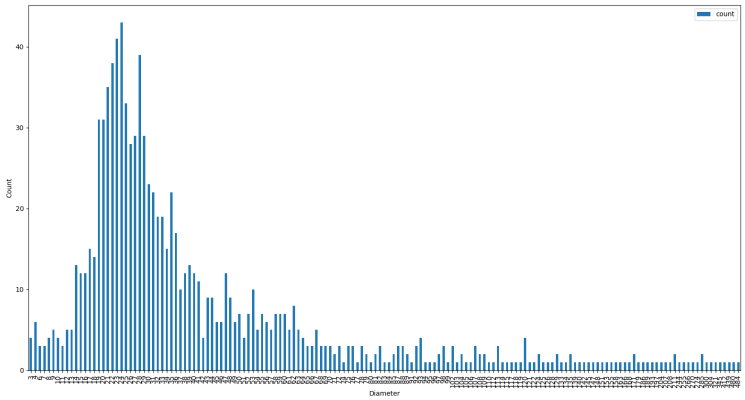

FIGURE 63 – Distribution du diamètre

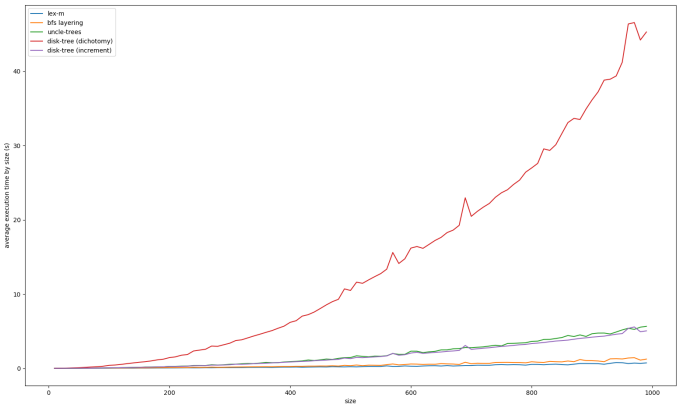

FIGURE 64 – Temps d'exécution moyen en fonction de la taille

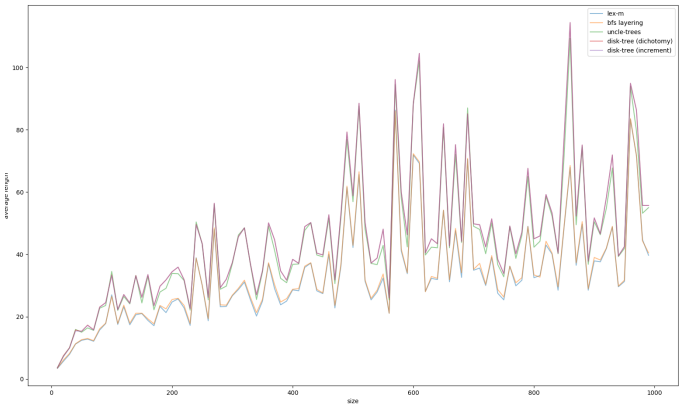

FIGURE 65 – Length moyenne en fonction de la taille

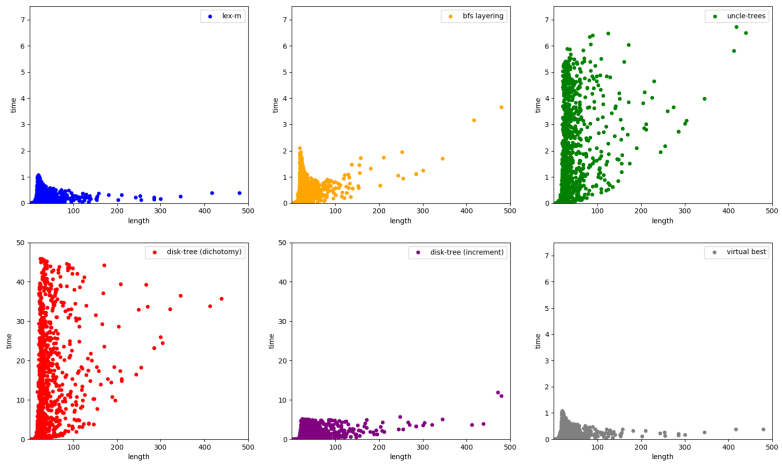

FIGURE 66 – Temps d'exécution en fonction de la length calculée

Grande treelength Nous remarquons tout d'abord (fig. 64) que l'implémentation de disk-tree par dichotomie est très lente, alors que l'implémentation avec recherche linéaire est aussi rapide qu'uncle-trees. C'est un exemples extrême de ce que nous avons vu précédemment : nous avons une bonne borne inférieure sur la treelength du graphe, mais pas de meilleure borne supérieure que le diamètre du graphe ; donc disk-tree avec recherche linéaire trouvera en peu d'itérations, alors que la version par dichotomie devra chercher dans un espace bien plus grand. Lex-M et BFS-Layering sont les plus rapides et leurs temps d'exécution sont très proches.

Nous savons (c.f. section 5.2.3) que ces graphes sont en fait des assemblages de cycles partageant des sommets entre eux, ce qui explique pourquoi les graphes n'ont pas un degré élevé (fig. 62).

La length (fig. 65) obtenue est très similaire pour tous les algorithmes. Bfs-layering et lex-M, par exemple, sont presque confondus sur cette courbe. En effet, comme les deux sont basés sur un parcours en profondeur, dans le cas de "cycles collés" comme dans cette famille de graphe leur comportement sera très similaire : lex-M ajoutera une arête entre deux sommets à distance i du point de départ de la décomposition, et bfs-layering les placera dans une même couche L^i_j, ce qui est équivalent.

Le tableau 9 nous montre que sur les graphes pour lesquels la meilleure length trouvée est inférieure à 70, lex-M et bfs-layering trouvent assez souvent la meilleure length seul, mais au delà il y a très peu de graphes sur lesquels un algorithme est le seul à obtenir la meilleure length car bien souvent, lex-M et bfs-layering trouvent tous les deux la meilleure length.

Enfin, nous observons en figure 66 que c'est à nouveau lex-M qui est le plus proche du meilleur algorithme.

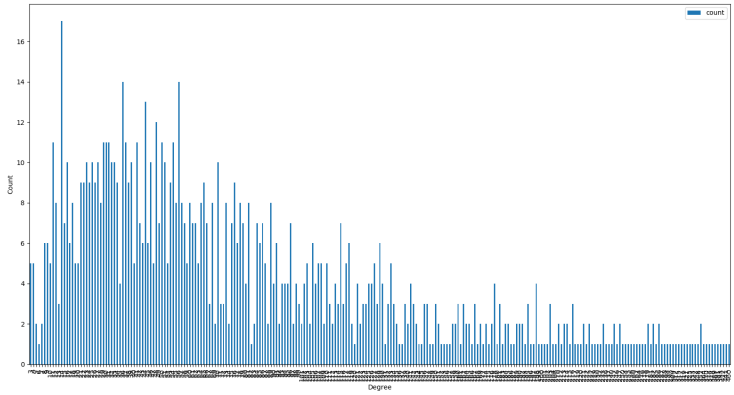

FIGURE 67 – Distribution du degré

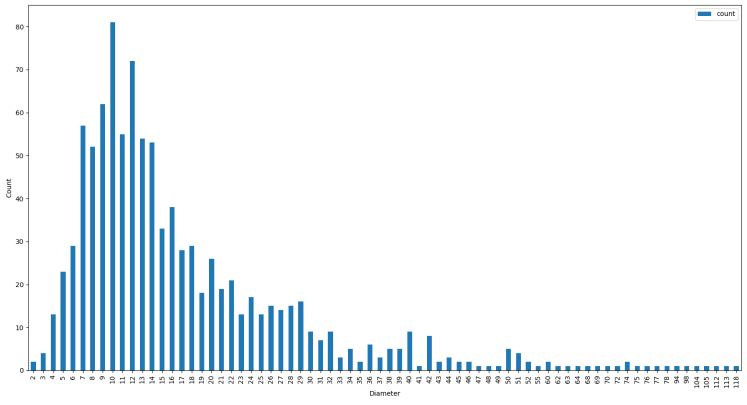

FIGURE 68 – Distribution du diamètre

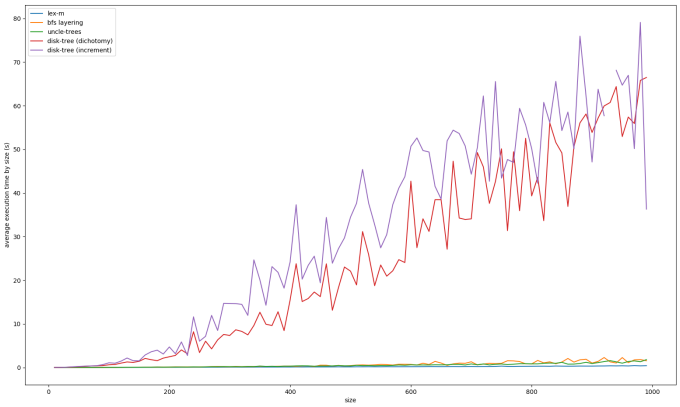

FIGURE 69 – Temps d'exécution moyen en fonction de la taille

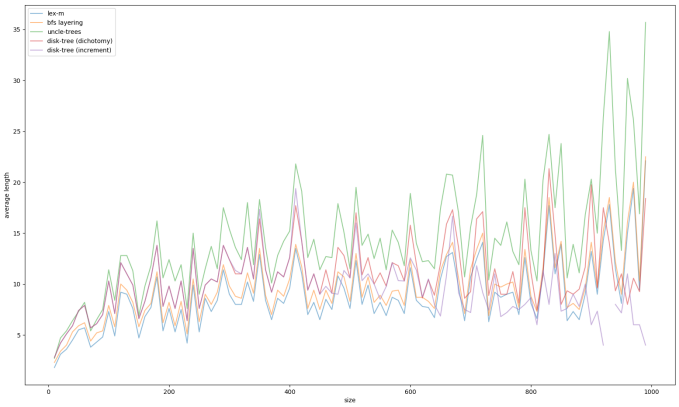

FIGURE 70 – Length moyenne en fonction de la taille

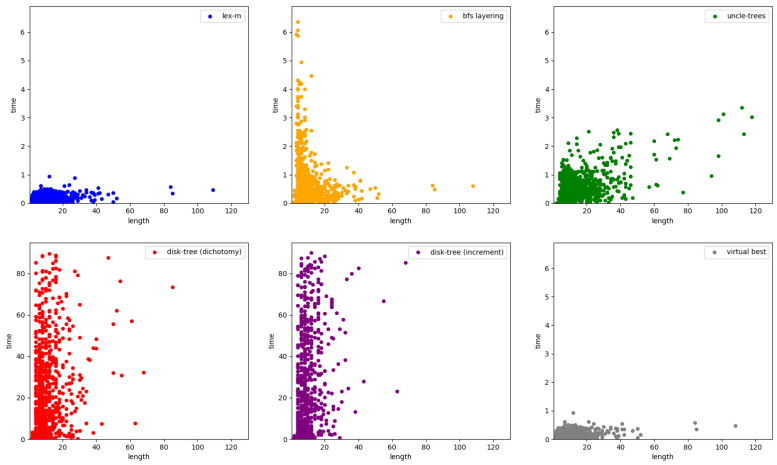

FIGURE 71 – Temps d'exécution en fonction de la length calculée

Graphes série-parallèle Cette fois disk-tree avec recherche dichotomique est plus rapide que la version par recherche linéaire (fig. 69) : nous n'avons pas de borne inférieure sur la treelength, donc la recherche par dichotomie trouve la bonne valeur de k en moins d'itérations que la recherche linéaire. Lex-M est le plus rapide, avec bfs-layering et uncle-trees légèrement plus lents mais très proches.

Concernant la length, nous observons à nouveau le même type de "classement" que précédemment : uncle-trees donne les plus grandes lengths, suivi par disk-tree, bfs-layering puis lex-M (fig. 70), ce qui correspond aux résultats théoriques en ce qui concerne le rapport d'approximation de chaque algorithme pour tous à part disk-tree qui obtient de moins bons résultats que lex-M, alors qu'il s'agit d'une 2-approximation supposée.

C'est encore une fois lex-M qui est le plus proche du meilleur algorithme, comme le montre la figure 71.

5.3.4 Conclusions

Nous présentons maintenant des conclusions plus générales tirées de ces tests.

Les performances relatives des deux implémentations de disk-tree dépendent principalement des bornes données sur la valeur de k. Si nous avons une bonne borne inférieure, il est plus intéressant d'utiliser la version avec recherche linéaire car il y a plus de chances de trouver rapidement la valeur minimum de k. En revanche, si nous n'avons pas de bonnes bornes sur la valeur de k, comme l'espace de recherche de k peut être très grand (entre 1 et le diamètre du graphe) la recherche par dichotomie peut se révéler plus

rapide.

D'autre part, quand le degré est petit par rapport au nombre de sommets, disk-tree et uncle-trees trouvent des décompositions plus rapidement. En effet, pour disk-tree, un petit degré implique qu'il y a moins de choix possibles pour le nouveau centre à chaque étape de la construction de la décomposition, cette dernière est donc calculée en moins d'étapes. Pour uncle-trees, lors de la construction des sacs, le nombre de voisins à parcourir pour chaque sommets pour trouver les juniors est moindre, ce qui a pour conséquence un temps d'exécution plus bas.

Pour bfs-layering, les graphes dont le diamètre est petit par rapport au nombre de sommets sont décomposés plus lentement car le layering tree-contient peu de couches, mais les partitions sont plus longue à calculer car elles contiennent un grand nombre de sommets.

Quand les graphes sont très denses (beaucoup d'arêtes par rapport au nombre de sommets), nous avons pu observer que lex-M obtient une décomposition plus rapidement que les autres algorithmes car les graphes sont proches d'être chordaux ("presque" triangulés), ce qui fait qu'il y a moins d'arêtes à ajouter pour trianguler.

Concernant la length, nous avons pu constater qu'à part uncle-trees (dont on ne connaît pas le rapport d'approximation) tous les algorithmes obtiennent des lengths proches. Dans les graphes très connexes, lex-M obtient des décompositions dont la length est souvent meilleure que celle obtenue par les autres algorithmes car ces graphes sont proches de graphes chordaux, donc la triangulation change peu les distances. Dans ces graphes, nous avons aussi vu que lex-M calculait la tree-decomposition rapidement, il a donc un fort avantage sur les autres algorithmes en temps et en length calculée.

Dans les graphes comportant de grands cycles, uncle-trees obtient toujours une moins bonne length que les autres. En effet, le paramètre k est fortement lié à la taille du plus grand cycle induit, donc lors de la construction des sacs on aura un plus long chemin entre le sommet à l'origine du sac et le point de départ de la décomposition.

Il n'est pas surprenant de voir que bfs-layering et lex-M obtiennent des résultats de length similaires car ils sont tous les deux basés sur un parcours en largeur, la différence étant que bfs-layering prend quelques sommets plus éloignés en construisant le layering tree.

Enfin, un constat important est celui que disk-tree, que l'ont suppose être une 2-approximation de la treelength, calcule peu souvent de meilleures décompositions que bfs-layering et lex-M qui sont des 3-approximations. Plutôt que de supposer que disk-tree n'est pas une 2-approximation, notre hypothèse est que lex-M et bfs-layering sont en pratique souvent meilleurs qu'une 3-approximation.

Il semble donc que parmi ces algorithmes, lex-M soit le meilleur autant en terme de temps d'exécution que de length obtenue. C'est une 3-approximation, mais en pratique la length des résultats obtenus est meilleure

que ceux obtenus avec disk-tree qui est suspecté être une 2-approximation. Il serait tout de même intéressant de pouvoir faire des tests plus longs, afin de comparer les lengths obtenues sans avoir d'information manquante à cause du temps d'exécution, et de pouvoir avoir plus de graphes dont la treelength est connue pour comparer les résultats de disk-tree et lex-M (ou bfs-layering) afin de comprendre pourquoi nos 3-approximations trouvent aussi souvent de meilleurs résultats que disk-tree.

6 Travail en cours

6.1 Analyse de l'approximation de Disk-Tree

Nous présentons ici les pistes explorées pour tenter de prouver que disk-tree est une 2-approximation de la treelength, c'est-à-dire que l'algorithme n'échoue pas pour $k = treelength(G)$. Pour cela, nous devons montrer que, étant donné un ensemble C de centres potentiels pour construire un nouveau sac de la décomposition, il existe toujours $c \in C$ tel qu'il est possible de construire un sac contenant au moins un sommet non couvert jusque là.

Soit CC une composante connexe voisine de c dans $G[V(G) \setminus covered(T_i)$. Si nous pouvons montrer qu'il existe toujours $c \in C$ tel qu'il y a un sommet $v \in N_G(c) \cap CC$ en conflit avec aucun autre sommet de $out(c, covered(T))$, alors nous aurons montré que disk-tree finit toujours pour $k \geq tl(G)$, donc qu'il s'agit bien d'une 2-approximation.

Nous allons approcher le problème par contradiction : nous allons à partir de maintenant supposer que pour chaque sommet dans $c \in out(c, covered(T))$ et pour tout $b_c \in N_G(C) \cap CC$ il existe un sommet $c'(b_c) \in out(c, covered(T))$ tel que $c'(b_c)$ est en conflit avec b_c.

Soit $c \in out(c, covered(T))$ et un sommet $c' \in out(c, covered(T))$. c' est impliqué dans un conflit maximal avec $b_c \in N_G(C) \cap CC$ s'il n'y a aucune paire a, b de sommets tels que a et b sont en conflit et $dist_G(a, b) > dist_G(b_c, c')$.

Lemma 9. *Soit $c \in out(c, covered(T))$ et un sommet $c' \in out(c, covered(T))$ impliqué dans un conflit maximal avec $b_c \in N_G(C) \cap CC$.*

Si $dist_G(b_c, c') > k + 2$, alors le sommet c' est en conflit avec tous les sommets de $N_G(c) \cap CC$.

Démonstration. Soit $b_c \in N_G(c) \cap CC$ un sommet impliqué dans un conflit de longueur maximale avec $c' \in out(c, covered(T))$ tel que $dist_G(b_c, c') > k + 2$.

Soit $b_d \in N_G(c) \cap CC$ en conflit avec un sommet $d \in out(c, covered(T))$. b_d est bien impliqué dans un conflit car sinon nous pouvons construire un sac de centre c et contenant au moins b_d. Comme le conflit entre b_c et c' est de longueur maximale et supérieure à $k + 2$, alors on a $k < dist_G(d, b_d) \leq k + 2$.

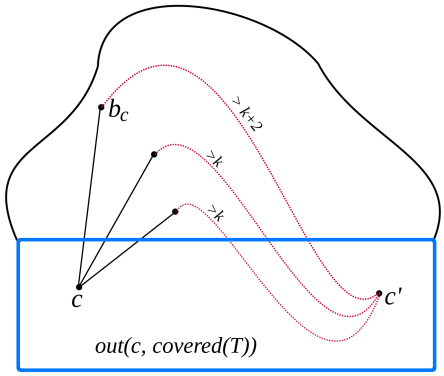

FIGURE 72 – Illustration du lemme 7

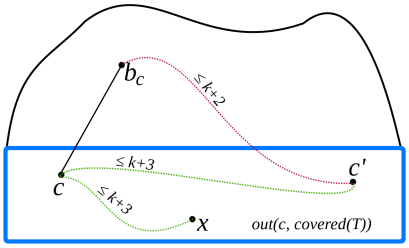

FIGURE 73 – Illustration du lemme 8

Il existe un plus court chemin entre b_d et c' de longueur au moins $k+1$ car sinon $dist_G(b_c, c') \leq dist_G(b_c, c) + dist_G(c, b_d) + dist_G(b_d, c') \leq k+2$.

Donc, $dist_G(b_d, c') > k$ alors b_d est aussi en conflit avec c'. □

Lemma 10. *Soit $c \in out(c, covered(T))$ et un sommet $c' \in out(c, covered(T))$ impliqué dans un conflit maximal avec $b_c \in N_G(C)$.*

Si $dist_G(b_c, c') \leq k+2$, alors pour tout $c'_i \in out(c, covered(T))$, $dist_G(c, c'_i) \leq k+3$.

Démonstration. Soit $b_c \in N_G(c) \cap CC$ un sommet impliqué dans un conflit maximal avec $c' \in out(c, covered(T))$ tel que $dist_G(b_c, c') \leq k+2$.

Supposons qu'il existe un $x \in out(c, covered(T))$ tel que $dist_G(c, x) > k+3$. Comme le conflit entre b_c et c' est maximal, nous savons que $dist_G(b_c, x) \leq k+2$ Donc, $dist_G(c, c_i) \leq dist_G(c, b_i) + dist_G(b_i, c_i) \leq k+3$.

On a alors $dist_G(c, x) \leq dist_G(c, b_c) + dist_G(b_c, x) \leq k+3$, une contradiction puisque nous avons supposé que $dist_G(c, x) > k+3$. □

Lemma 11. *Pour chaque $c_i \in out(c, covered(T))$ en conflit avec un sommet $b_i \in N_G(c) \cap CC$ $dist_G(c, c_i) \geq k$.*

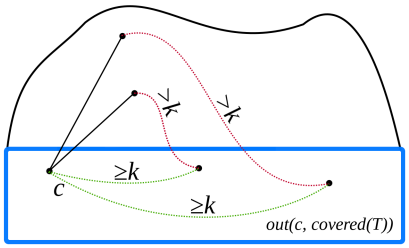

FIGURE 74 – Illustration du lemme 9

Démonstration. Nous savons que pour tout $b_i \in N_G(c) \cap CC$ il existe un $c_i \in out(c, covered(T))$ tel que b_i est en conflit avec c_i. Donc $dist_G(b_i, c_i) > k$. Supposons avoir, pour b_i et c_i en conflit donnés, $dist_G(c, c_i) < k$. Alors $dist_G(b_i, c) + dist_G(c, c_i) < k + 1k$. Donc $dist_G(b_i, c_i) \leq k$ ce qui est une contradiction puisque b_i et c_i sont en conflit. □

6.2 Modification

Nous proposons ici une modification à l'algorithme pouvant potentiellement nous permettre de prouver qu'il s'agit d'une 2-approximation. L'idée est de faire en sorte que la décomposition n'échoue qu'à condition que tous les conflits soient dûs à deux sommets à distance au moins $k + 3$.

Lors de l'élimination des conflits, on ajoute une condition sur la distance entre les sommets : si le sommet u est le dernier sommet restant dans S et qu'il est en conflit avec un sommet $c_i \in out(c, covered(T))$ tel que $dist_G(c_i, u) \leq k+2$, alors ce conflit est ignoré et le sommet conservé dans S.

Nous pouvons prouver qu'avec cette modification, le résultat est toujours une tree-decomposition de G.

Pour tout $i \geq 1$, on appelle T_i l'arbre construit par Disk-Tree avec les i sacs $\{S_1, S_2, ..., S_i\}$, et c_j le sommet choisit pour construire S_j.

Soit pour tout $i \geq 1$ la proposition suivante :

P_i ="T_i *est un sous-arbre d'une tree-decomposition de* G

Lemma 12. *Si* P_i *est vraie, alors* $out(c_{i+1}, covered(T_i)) \subseteq B(c_{i+1})$ *et* T_{i+1} *est une tree-decomposition de* $G[covered(T_{i+1})]$.

Démonstration. Supposons que P_i est vraie. On considère CC une composante connexe de $G[V(G) \setminus covered(T_i)]$ voisine de c_{i+1}.

Soit $u \in CC$, comme $c_{i+1} \in border(T_i)$ est contenu dans un sac de profondeur maximale dans T_i, nous savons que $B(u)$ n'est pas un descendant de $B(c_{i+1})$.

Comme $out(c_{i+1}, covered(T_i))$ est un séparateur du graphe, il existe un sac $X \in V(T_i)$ tel que $out(c_{i+1}), covered(T_i) \subseteq X$. T_i est une tree-decomposition de $G[covered(T_i)]$ donc pour tout $v \in out(c_{i+1}, covered(T_i))$, X est un descendant de $B(v)$, en particulier X est un descendant de $B(u)$ et $B(c_{i+1})$. $B(u)$ est donc soit un ancêtre de $B(c_{i+1})$, soit $B(c_{i+1})$ lui même : $u \in B(c_{i+1})$.

D'autre part, les sommets de S_{i+1} déjà couverts par T_i sont les sommets de $out(c_{i+1}, covered(T_i))$. On obtient T_{i+1} en ajoutant une arête $(S_{i+1}, B(c_{i+1}))$ à T_i. Or, nous venons de montrer que $out(c_{i+1}, covered(T_i)) \subseteq B(c_{i+1})$. Donc, comme T_i est une tree-decomposition de $G[covered(T_i))]$, T_{i+1} est une tree-decomposition de $G[covered(T_{i+1})]]$ □

Cette variation est donc bien une tree-decomposition de G. Il nous faut maintenant montrer que le fait d'ignorer ces conflits ne change pas la length de la décomposition.

Lemma 13. *Si cette variante de disk-tree termine, la décomposition obtenue a une length d'au plus* $2 \cdot k$.

Démonstration. Si les conditions pour accepter un conflit entre deux sommets à distance au plus $k+2$ l'un de l'autre ne sont jamais remplies, alors soit la décomposition échoue, soit la length de la décomposition est au plus $2 \cdot k$ car l'exécution est équivalente à la version de disk-tree sans modification.

Soit $s \in V(G)$ le point de départ de la décomposition. Lorsque le premier sac est construit, on a $c = s$ et $out(c, covered(T)) = \emptyset$. Il est évident qu'il ne peut pas y avoir de voisin de c en conflit avec un autre sommet de $out(c, covered(T))$ à distance au plus $k+2$, le diamètre du sac construit est donc au plus $2 \cdot k$.

Supposons maintenant que tous les sacs de T_i ont un diamètre d'au plus $2 \cdot k$, et que lors de la construction de S_{i+1} on soit contraint d'accepter un conflit entre $c' \in out(c_{i+1}, covered(T_i))$ et $b \in N_G(c_{i+1})$. On a donc $k < dist_G(b, c') \leq k+2$. Nous pouvons montrer que le sac obtenu a bien un diamètre d'au plus $2 \cdot k$: pour tout $c_j \in out(c_{i+1}, covered(T_i))$, $dist_G(b, c_j) \leq k+2$ car sinon il y aurait un conflit entre c_j et b, et b serait retiré du sac. Le diamètre de S_{i+1} est donc au plus $2 \cdot k$ lorsque $k+2 \leq 2 \cdot k$, c'est-à-dire quand $k \geq 2$.

Nous pouvons donc en déduire, par récurrence, que la length de la décomposition obtenue est au plus $2 \cdot k$ si l'algorithme termine. Par ailleurs, pour $k = 1$, l'algorithme termine sans conflit si le graphe est chordal, et la décomposition est de length au plus $2 \cdot k$ aussi.

Donc, pour tout $k \geq 1$, cette version de disk-tree termine et donne une décomposition de length au plus $2 \cdot k$. □

7 Conclusion

Nous avons rappelé la notion de tree-decomposition d'un graphe et des paramètres qui y sont liés, à savoir la treewidth et la treelength. Après avoir expliqué que savoir si un graphe a une treewidth d'au plus k ou une treelength d'au plus 2 est NP-Complet, nous nous sommes concentrés sur l'approximation de la treelength.

Pour cela, nous avons présenté 4 algorithmes issus de la littérature :

— Lex-M (Rose et al., 1976 [13])
— Uncle-trees (Seymour, 2016 [14])
— Bfs-Layering (Dourisboure et al., 2001 [8])
— Disk-tree (Dourisboure et al., 2001 [8])

À part uncle-trees dont nous ne connaissons pas de rapport d'approximation pour la treelength, tous visent à minimiser la length de la tree-decomposition obtenue. Pour lex-M et Bfs-layering, la length du résultat obtenue est au plus $3 \cdot treelength(G) + 1$. Nous avons proposé une légère amélioration de Bfs-layering et prouvé qu'elle nous permet de toujours obtenir une décomposition de length au plus $3 \cdot treelength(G)$ dans les graphes chordaux.

L'algorithme disk-tree est une heuristique qui dépend d'un paramètre k. Si pour un k donné l'algorithme n'échoue pas, la décomposition retournée est de length au plus $2 \cdot k$. Il est prouvé que pour $k \geq 3 \cdot treelength(G) - 2$ l'algorithme n'échoue pas, et par conséquent renvoie une décomposition de length au plus $6 \cdot treelength(G) - 4$. Cependant, il est supposé que cet algorithme n'échoue pas pour tout $k \geq treelength(G)$, auquel cas il s'agirait d'une 2-approximation de la treelength si $k \leq treelength(G)$. Nous avons là aussi proposé une légère amélioration de l'algorithme qui permettrait dans certains cas de diminuer la length du résultat.

Ces algorithmes ont ensuite été implémentés en Python3 sous forme d'un package pour la bibliothèque graphes de SageMath, et nous avons effectué des tests pour comparer leurs performances en terme de temps d'exécution et de length obtenue. Cela nous a permis de mettre en évidence des liens entre la structure des graphes (diamètre, densité, degré, nombre d'arêtes...), le temps d'exécution et la length du résultat.

Nous avons aussi pu constater que lex-M est de loin l'algorithme le plus rapide parmi ceux que nous avons étudiés, et qu'il obtenait en moyenne des décompositions de meilleure length.

Enfin, nous avons présenté quelques résultats portant sur notre travail actuel, celui de montrer que disk-tree renvoie toujours une tree-decomposition pour $k = treelength(G)$, et donc qu'il s'agit d'une 2-approximation de la treelength. Pour cela, nous avons d'abord identifié dans quelle situation l'algorithme peut échouer si $k = treelength(G)$ dans le but de montrer, par contradiction, que cette situation ne peut pas se produire.

8 Bibliographie

Références

[1] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74, 06 2001.

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM JOURNAL OF DISCRETE MATHEMATICS*, 8(2) :277–284, 1987.

[3] Hans Bodlaender. Dynamic programming on graphs with bounded treewidth. *Lecture Notes in Computer Science*, 317, 03 2001.

[4] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 226–234, New York, NY, USA, 1993. Association for Computing Machinery.

[5] Victor Chepoi and Feodor Dragan. A note on distance approximating trees in graphs. *European Journal of Combinatorics*, 21(6) :761–766, 2000.

[6] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1) :12 – 75, 1990.

[7] Yon Dourisboure. Compact routing schemes for bounded tree-length graphs and for k-chordal graphs. In Rachid Guerraoui, editor, *Distributed Computing*, pages 365–378, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] Yon Dourisboure and Cyril Gavoille. Small Diameter Bag Tree-Decompositions, May 2004. Rapport de recherche.

[9] Philippe Galinier, Michel Habib, and Christophe Paul. Chordal graphs and their clique graphs. In *Graph-Theoretic Concepts in Computer Science, 21st International Workshop, WG '95, Aachen, Germany, June 20-22, 1995, Proceedings*, pages 358–371, 1995.

[10] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[11] Daniel Lokshtanov. On the complexity of computing treelength. *Discrete Applied Mathematics*, 158(7) :820 – 827, 2010. Third Workshop on Graph Classes, Optimization, and Width Parameters Eugene, Oregon, USA, October 2007.

[12] Neil Robertson and P.D Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3) :309 – 322, 1986.

[13] Donald J Rose, R Endre Tarjan, and George S Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2) :266–283, 1976.

[14] Paul Seymour. Tree-chromatic number. *Journal of Combinatorial Theory, Series B*, 116 :229 – 237, 2016.

[15] William Stein. Sagemath. https ://www.sagemath.org/index.html.

[16] Oylum Şeker, Pinar Heggernes, Tınaz Ekim, and Z. Caner Taşkın. Generation of random chordal graphs using subtrees of a tree, 2018.

9 Annexe

9.1 Comparaison de la meilleure length obtenue

9.1.1 Barabási-Albert, $k = 2$

		meilleures lengths					
		2	3	4	5	6	7
lex-m	# trouvée	12	24	81	225	599	9
	# trouvée seul	7	12	28	141	47	0
bfs layering	# trouvée	6	11	60	94	564	9
	# trouvée seul	0	0	5	2	2	0
uncle-trees	# trouvée	2	6	13	25	80	8
	# trouvée seul	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	2	9	25	32	226	0
	# trouvée seul	0	0	0	0	0	0
disk-tree (increment)	# trouvée	2	11	25	31	124	0
	# trouvée seul	0	2	0	0	0	0

TABLE 1 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.2 Barabási-Albert, $k = log(n)$

		meilleures lengths			
		1	2	3	4
lex-m	# trouvée	2	12	51	919
	# trouvée seul	2	8	26	17
bfs layering	# trouvée	0	4	25	902
	# trouvée seul	0	0	0	0
uncle-trees	# trouvée	0	0	16	895
	# trouvée seul	0	0	0	0
disk-tree (dichotomy)	# trouvée	0	1	20	185
	# trouvée seul	0	0	0	0
disk-tree (increment)	# trouvée	0	1	23	204
	# trouvée seul	0	0	3	0

TABLE 2 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.3 Barabási-Albert, $k = \sqrt{n}$

		meilleures lengths		
		1	2	3
lex-m	# trouvée	1	15	972
	# trouvée seul	1	8	0
bfs layering	# trouvée	0	8	972
	# trouvée seul	0	0	0
uncle-trees	# trouvée	0	5	972
	# trouvée seul	0	0	0
disk-tree (dichotomy)	# trouvée	0	7	664
	# trouvée seul	0	0	0
disk-tree (increment)	# trouvée	0	8	660
	# trouvée seul	0	1	0

TABLE 3 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.4 Erdős–Rényi, $p = \frac{log(n)}{n}$

		meilleures lengths						
		1	2	3	4	5	6	7
lex-m	# trouvée	7	4	8	32	208	721	2
	# trouvée seul	0	1	3	16	118	46	0
bfs layering	# trouvée	7	3	5	16	91	674	2
	# trouvée seul	0	0	0	0	0	0	0
uncle-trees	# trouvée	0	0	0	1	16	199	2
	# trouvée seul	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	6	1	3	9	36	122	0
	# trouvée seul	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	6	1	3	9	39	121	0
	# trouvée seul	0	0	0	0	3	0	0

TABLE 4 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.5 Erdős–Rényi, $p = \frac{1}{n}$

		meilleures lengths
		1
lex-m	# trouvée	990
	# trouvée seul	0
bfs layering	# trouvée	990
	# trouvée seul	0
uncle-trees	# trouvée	0
	# trouvée seul	0
disk-tree (dichotomy)	# trouvée	990
	# trouvée seul	0
disk-tree (increment)	# trouvée	990
	# trouvée seul	0

TABLE 5 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.6 Graphes chordaux

		meilleures lengths
		1
lex-m	# trouvée	990
	# trouvée seul	981
bfs layering	# trouvée	9
	# trouvée seul	0
uncle-trees	# trouvée	0
	# trouvée seul	0
disk-tree (dichotomy)	# trouvée	8
	# trouvée seul	0
disk-tree (increment)	# trouvée	8
	# trouvée seul	0

TABLE 6 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.7 Cycles

		meilleures lengths				
		[0,100[	[100,200[	[200,300[	[300,400[	[400,500[
lex-m	# trouvée	19	20	20	20	20
	# trouvée seul	0	0	0	0	0
bfs layering	# trouvée	19	20	20	20	20
	# trouvée seul	0	0	0	0	0
uncle-trees	# trouvée	19	20	20	20	20
	# trouvée seul	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	19	20	20	20	18
	# trouvée seul	0	0	0	0	0
disk-tree (increment)	# trouvée	19	20	20	20	20
	# trouvée seul	0	0	0	0	0

TABLE 7 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.8 Grilles

		meilleures lengths					
		[0,10[	[10,20[	[20,30[	[30,40[	[40,50[	[50,60[
lex-m	# trouvée	577	190	60	41	8	1
	# trouvée seul	356	144	40	33	1	0
bfs layering	# trouvée	97	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0
uncle-trees	# trouvée	97	14	17	4	1	0
	# trouvée seul	0	9	7	4	0	0
disk-tree (dichotomy)	# trouvée	74	5	8	4	0	0
	# trouvée seul	0	0	3	2	0	0
disk-tree (increment)	# trouvée	232	74	31	18	26	2
	# trouvée seul	11	28	17	10	18	1

		meilleures lengths																	
		1	2	3	4	5	6	7	8	9	10	11	13	14	15	16	17	18	19
lex-m	# trouvée	97	189	20	16	28	32	84	89	22	28	6	12	2	3	0	73	63	3
	# trouvée seul	0	189	20	0	28	1	84	13	21	21	6	9	2	3	0	68	32	3
bfs layering	# trouvée	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uncle-trees	# trouvée	97	0	0	0	0	0	0	0	0	0	0	2	2	7	1	2	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	2	5	1	1	0	0
disk-tree (dichotomy)	# trouvée	74	0	0	0	0	0	0	0	0	2	0	0	0	1	0	2	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	97	0	0	16	0	32	3	76	8	7	0	1	0	3	4	28	31	0
	# trouvée seul	0	0	0	0	0	1	3	0	7	0	0	0	0	2	4	22	0	0

TABLE 8 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.9 Grande treelength

		meilleures lengths				
		[0,100[	[100,200[	[200,300[	[300,400[	[400,500[
lex-m	# trouvée	841	24	7	2	2
	# trouvée seul	403	4	0	0	0
bfs layering	# trouvée	548	22	7	2	2
	# trouvée seul	104	2	0	0	0
uncle-trees	# trouvée	26	4	3	2	1
	# trouvée seul	1	0	0	0	0
disk-tree (dichotomy)	# trouvée	24	4	3	2	1
	# trouvée seul	0	0	0	0	0
disk-tree (increment)	# trouvée	25	4	3	2	1
	# trouvée seul	0	0	0	0	0

		meilleures lengths									
		[0,10[	[10,20[	[20,30[	[30,40[	[40,50[	[50,60[	[60,70[	[70,80[	[80,90[	[90,100[
lex-m	# trouvée	35	280	296	84	61	30	21	16	11	7
	# trouvée seul	13	145	150	39	26	13	6	4	3	4
bfs layering	# trouvée	26	165	185	61	48	20	19	13	8	3
	# trouvée seul	2	28	37	16	13	3	4	1	0	0
uncle-trees	# trouvée	11	5	3	1	2	0	2	1	1	0
	# trouvée seul	0	0	1	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	11	5	1	1	3	0	2	1	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	12	5	1	1	3	0	2	1	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0

		meilleures lengths						
		[100,110[	[110,120[	[120,130[	[130,140[	[140,150[	[150,160[	[180,190[
lex-m	# trouvée	3	4	7	3	1	5	1
	# trouvée seul	0	1	3	0	0	0	0
bfs layering	# trouvée	4	3	4	4	1	5	1
	# trouvée seul	1	0	0	1	0	0	0
uncle-trees	# trouvée	0	1	1	2	0	0	0
	# trouvée seul	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	0	1	1	2	0	0	0
	# trouvée seul	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	0	1	1	2	0	0	0
	# trouvée seul	0	0	0	0	0	0	0

TABLE 9 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.10 Graphes série-parallèle

		meilleures lengths				
		[0,20[	[20,40[	[40,60[	[80,100[	[100,120[
lex-m	# trouvée	898	66	7	2	0
	# trouvée seul	592	27	1	0	0
bfs layering	# trouvée	308	40	7	2	1
	# trouvée seul	5	1	1	0	1
uncle-trees	# trouvée	8	0	0	0	0
	# trouvée seul	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	31	1	0	1	0
	# trouvée seul	1	0	0	0	0
disk-tree (increment)	# trouvée	33	2	0	0	0
	# trouvée seul	6	0	0	0	0

		meilleures lengths														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
lex-m	# trouvée	13	59	117	107	103	107	82	59	41	30	40	30	22	24	20
	# trouvée seul	10	52	107	73	84	52	59	25	30	13	25	9	13	11	10
bfs layering	# trouvée	3	7	10	34	21	55	23	35	12	17	15	23	8	13	10
	# trouvée seul	0	0	0	0	1	0	0	1	1	0	0	2	0	0	0
uncle-trees	# trouvée	0	0	2	0	1	1	1	2	0	0	0	0	1	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	3	0	0	4	1	4	1	6	2	1	0	1	2	2	0
	# trouvée seul	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	3	0	0	4	1	2	2	7	2	1	0	1	2	1	0
	# trouvée seul	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0

		meilleures lengths														
		16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
lex-m	# trouvée	13	7	15	9	6	12	7	5	2	6	5	5	2	3	1
	# trouvée seul	5	5	5	4	3	7	1	4	1	3	0	2	0	2	1
bfs layering	# trouvée	7	1	9	5	3	5	6	1	1	3	5	3	2	1	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uncle-trees	# trouvée	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	2	2	2	1	0	1	0	0	0	0	0	0	0	0	0
	# trouvée seul	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

		meilleures lengths												
		31	32	33	34	36	37	38	39	41	42	47	50	52
lex-m	# trouvée	1	1	1	2	1	2	3	1	2	1	1	2	1
	# trouvée seul	0	1	0	0	1	0	0	1	0	0	0	1	0
bfs layering	# trouvée	1	0	1	2	0	2	4	0	2	2	1	1	1
	# trouvée seul	0	0	0	0	0	0	1	0	0	1	0	0	0
uncle-trees	# trouvée	0	0	0	0	0	0	0	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	0	0	0	0	0	0	1	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	0	0	0	0	0	0	1	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0	0	0	0

TABLE 10 – Nombre de fois que chaque algorithme a trouvé la meilleure length

9.1.11 Triangulations planaires

		meilleures lengths									
		1	2	3	4	5	6	7	8	9	10
lex-m	# trouvée	3	29	42	107	182	253	214	116	37	7
	# trouvée seul	3	23	39	79	160	201	173	72	23	2
bfs layering	# trouvée	0	6	3	26	21	47	41	38	14	5
	# trouvée seul	0	0	0	0	0	0	0	0	0	0
uncle-trees	# trouvée	0	3	0	1	0	0	0	0	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0
disk-tree (dichotomy)	# trouvée	0	1	0	15	6	12	2	13	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0
disk-tree (increment)	# trouvée	0	1	0	15	6	13	2	15	0	0
	# trouvée seul	0	0	0	0	0	0	0	0	0	0

TABLE 11 – Nombre de fois que chaque algorithme a trouvé la meilleure length