Master Informatique Théorique

Travaux d'Étude et de Recherche

Rapport de Projet

Algorithmique pour le consensus d'ordres

Auteurs

Himeur Areski

Picasarri-Arrieta Lucas

Encadrante

Bérard Sèverine

Résumé

En utilisant la théorie des graphes, ce rapport décrit plusieurs méthodes pour obtenir un consensus d'ordre. C'est-à-dire transformer une relation binaire quelconque en une relation d'ordre partielle avec une perte d'information minimisée.

Pour cela, le rapport présente le contexte bio-informatique de la cartographie génétique ainsi que la problématique de consensus d'ordre ou de linéarisation. Nous étudions une heuristique déjà existante et nous proposons une nouvelle implémentation de cette heuristique, ainsi que deux nouveaux algorithmes pour la résolution de ce problème. Finalement, nous comparons la performance et la qualité des résultats des algorithmes sur des exemples.

Remerciements

Nous tenons à remercier tout particulièrement Mme Sèverine Bérard de nous avoir proposé ce sujet. Nous la remercions aussi pour ses explications et son aide. Nous tenons aussi à remercier Mme Lisa De Mattéo pour ses travaux qui nous ont permis de bien appréhender ce sujet.

Table des matières

1 Introduction

Dans cette première partie, nous présentons le problème biologique et sa représentation sous forme de graphe. La seconde partie présente des définitions et notations. La troisième partie traite du rapport réalisé par L. De Matteo sur des algorithmes pour le consensus d'ordres. La quatrième partie présente un article de P. Giscard, N. Kriege et R. C. Wilson sur un algorithme de dénombrement des cycles.

Nous mettons en relation les deux documents dans une cinquième partie où nous montrons une nouvelle implémentation de l'heuristique. Nous présentons dans une sixième partie un algorithme exact. Dans une dernière partie, nous testons et comparons les performances de l'heuristique et de l'algorithme exact.

1.1 Cartographie génétique

La cartographie génétique est la construction d'une carte donnant les locus c'est-à-dire les positions fixes sur un chromosome de gènes d'une espèce. Une telle reconstruction de l'agencement des gènes est un problème de bio-informatique classique, pour lequel diverses méthodes ont été développées.

La méthode *ARt-DeCo* présentée dans l'article *Ancestral gene synteny reconstruction improves extant species scaffolding* [1] et mise au point par des chercheurs de l'équipe Phylogénie et Évolution Moléculaires de l'ISEM donne une nouvelle approche pour la réalisation de cette cartographie. Cette méthode utilise les adjacences de gènes présentes dans les banques de données déjà constituées pour des espèces proches pour inférer de nouvelles adjacences entre les gènes des espèces étudiées.

Cependant, cette méthode ne permet pas directement d'obtenir la carte génétique de l'espèce. En effet, différentes contradictions apparaissent dans les adjacences inférées. Par exemple, il est possible qu'un gène soit décrit comme adjacent à plus de deux autres ce qui ne peut pas convenir à l'agencement linéaire des gènes. Il faudra donc produire une carte génétique cohérente selon ces informations contradictoires. Pour cela, on peut utiliser des cartes considérées fiables et les compléter à l'aide de prédictions moins fiables.

1.2 Représentation sous forme de graphe

Pour représenter les données des cartes produites par la méthode *ARt-DeCo*, nous utilisons des multigraphes noirs et verts. Dans ces graphes, un sommet représente un gène, et une arête une adjacence entre deux gènes.

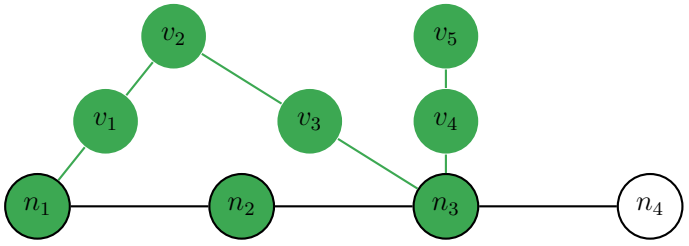

FIGURE 1 – Multigraphe noir et vert

Les sommets noirs n_1, n_2, n_3 et n_4 ainsi que les liens noirs représentent les localisations connues d'après une cartographie génétique existante. Un gène n_1 est relié par une arête noire à un gène n_2 s'il n'existe pas de gène noir tel que sa localisation soit entre n_1 et n_2. Ces informations sont considérées comme les plus fiables.

Les adjacences inférées par la méthode *ARt-DeCo* sont représentées par les arêtes vertes. Un gène v_1 est relié par une arête verte à un gène v_2 si v_1 et v_2 ont été inférés comme consécutifs.

1.3 Linéarisation et consensus d'ordre

À partir de la représentation décrite précédemment, le problème de linéarisation ou de consensus d'ordre a pour objectif de proposer une énumération de tous les gènes, c'est-à-dire de tous les sommets du graphe noir et vert, telle que l'agencement représenté par le graphe soit au mieux respecté.

Une solution optimale est un agencement des gènes convenant aux lois biologiques n'autorisant un gène à n'être voisin que d'un ou deux gènes. Cependant, cette solution doit conserver toutes les paires ordonnées noires et un maximum d'adjacences inférées par la méthode *ARt-DeCo*.

Exemple. La Figure 2 est la représentation d'une solution optimale pour la représentation de la Figure 1. Seule l'adjacence représentée par l'arête verte (v_3, n_3) a été supprimée dans la solution.

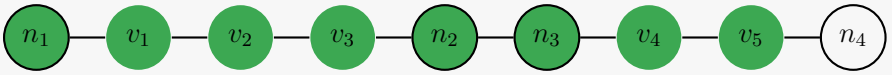

FIGURE 2 – Multigraphe noir et vert linéaire [1]

L'arête (n_1, n_2) est retirée, cependant, cette arête noire entre les gènes n_1 et n_2 de la Figure 1 indique qu'il n'existe pas de gène noir tel que sa localisation soit entre n_1 et n_2. Cette information reste représentée dans la solution proposée dans la Figure 2 par la chaîne $(n_1, v_1, v_2, v_3, n_2)$. Il en est de même pour (n_3, n_4) avec la chaîne (n_3, v_4, v_5, n_4).

Il est important de noter que la Figure 2 est la représentation d'une des solutions optimales. Il en existe d'autres comme le montre la Figure 3 qui est la représentation d'une autre solution optimale pour la représentation de la Figure 1. Dans cette solution il n'apparaît pas de chaîne (n_3, v_4, v_5, n_4) mais une chaine (n_2, v_5, v_4, n_3).

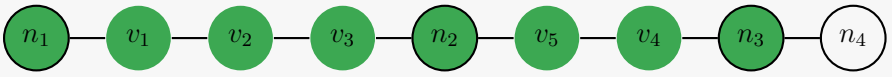

FIGURE 3 – Multigraphe noir et vert linéaire [2]

2 Notations et définitions

2.1 Théorie des graphes

Nous effectuons ici un rappel sur certains points de la théorie des graphes, que nous utiliserons tout au long de ce rapport.

Définition 2.1 (Graphe). Un graphe fini $G = (V, E)$ est constitué d'un ensemble fini de sommets V et d'un ensemble d'arêtes E contenant des paires d'éléments de V.

Notation 2.1. Pour un graphe G, on note $V(G)$ les sommets de G et $E(G)$ les arêtes de G. En l'absence de précisions, n désigne $|V(G)|$ et m désigne $|E(G)|$.

Définition 2.2 (Adjacence). Deux sommets x et y sont dits adjacents ou voisins s'il existe une arête $e = xy \in E$. Les sommets x et y sont appelés les extrémités de e. Deux arêtes sont adjacentes si elles partagent une de leurs extrémités. Aussi, les sommets x et y sont dits *incidents* à l'arête xy.

Définition 2.3 (Matrice d'adjacence). Une matrice d'adjacence pour un graphe à n sommets est une matrice de dimension $n \times n$ dont l'élément a_{ij} est le nombre d'arêtes liant le sommet i au sommet j.

Notation 2.2. Pour un graphe G et un sommet x, l'ensemble des voisins de x se nomme *voisinage* de x et se note $N_G(x)$. Le nombre de voisins de x est appelé *degré* de x et se note $deg_G(x)$ ou $|N_G(x)|$.

Définition 2.4 (Graphes simples). Un graphe est simple s'il ne contient pas d'arêtes multiples, c'est-à-dire plusieurs arêtes dans E pour une même paire de sommets. Un graphe contenant de telles arêtes est appelé *multigraphe*.

Remarque. La matrice d'adjacence d'un graphe simple ne contient donc que des 0 et des 1.

Définition 2.5 (Marche). Dans un graphe, une marche est une suite de sommets $x_0, x_1, ..., x_k$ telle que $\forall i \in [0, k-1]$, $x_i x_{i+1} \in E(G)$. Sa longueur est alors k.

Définition 2.6 (Chaine). Dans un graphe, une chaine est une marche qui ne passe pas deux fois par le même sommet.

Définition 2.7 (Cycle). Dans un graphe G, un cycle est une chaine $x_0, x_1, ..., x_k$ telle que $(x_k, x_0) \in E(G)$. Sa longueur est alors $k + 1$.

Définition 2.8 (Graphe connexe). Un graphe G est dit connexe si quels que soient les sommets x et y de $V(G)$, il existe une chaîne reliant x à y.

Définition 2.9 (Sous-graphe induit). Un sous-graphe induit est une restriction d'un graphe à un sous ensemble de sommets.
Formellement, H est un sous-graphe induit de G si :
— $V(H) \subseteq V(G)$
— $\forall (xy) \in E(G)$, $(xy) \in E(H)$ si et seulement si $x \in V(H)$ et $y \in V(H)$.
Pour $X \subseteq V(G)$, on note $G \setminus X$ le sous-graphe de G induit par $V(G) \setminus X$.
Pour $F \subseteq E(G)$, on note $G - F$ le graphe G privé des arêtes de F.

2.2 Complexité paramétrée

La complexité paramétrée classifie la complexité des problèmes selon des paramètres sur les données ou les sorties du problème. L'idée est de choisir des paramètres qui classifient des algorithmes efficaces en pratique.

Formellement, un problème paramétré L est défini comme un couple (Q, k) avec Q définissant un problème et k un paramètre sur l'entrée ou la sortie du problème. Il est alors possible d'associer une classe de complexité pour ces problèmes paramétrés.

Définition 2.10 (Classe FPT). La classe FPT pour *Fixed-Parameter Tractable* contient l'ensemble des problèmes (Q, k) qui peuvent être résolus par un algorithme de complexité $\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)})$, avec n la taille de la donnée et $f(.)$ une fonction calculable dépendant uniquement du paramètre k. [2]

Exemple. Un exemple d'un problème paramétré dans FPT est le problème SAT paramétré par le nombre de variables. Le problème SAT est un problème $\mathcal{NP}$-complet d'après le théorème de Cook. Cependant pour une formule de taille n avec k variables, une solution peut être trouvée en $\mathcal{O}(2^k \cdot n)$. Ici, Q est le problème SAT, le paramètre k est le nombre de variables en entrée et $f(k) = 2^k$.

3 Formalisation du problème

Nous donnons ici une définition formelle de graphes noirs et verts et présentons les différents résultats obtenus par L. De Mattéo dans ses travaux [3].

3.1 Linéarisation de multigraphes noirs et verts

Définition 3.1 (Multigraphe P-noir G_k-vert). Un multigraphe $G = P \cup G_1 \cup ... \cup G_k$ est un multigraphe P-noir G_k-vert si et seulement si :
- P est une chaîne non vide.
- $G_1, ..., G_k$ sont des graphes disjoints deux à deux.

Dans ce contexte :
- P est appelé graphe noir de G et $G_1, ..., G_k$ sont les graphes verts de G.
- Une arête est soit noire, soit verte.
- Un sommet est noir (respectivement vert) si il est l'extrémité d'une arête noire (respectivement verte). Il peut donc être noir et vert.

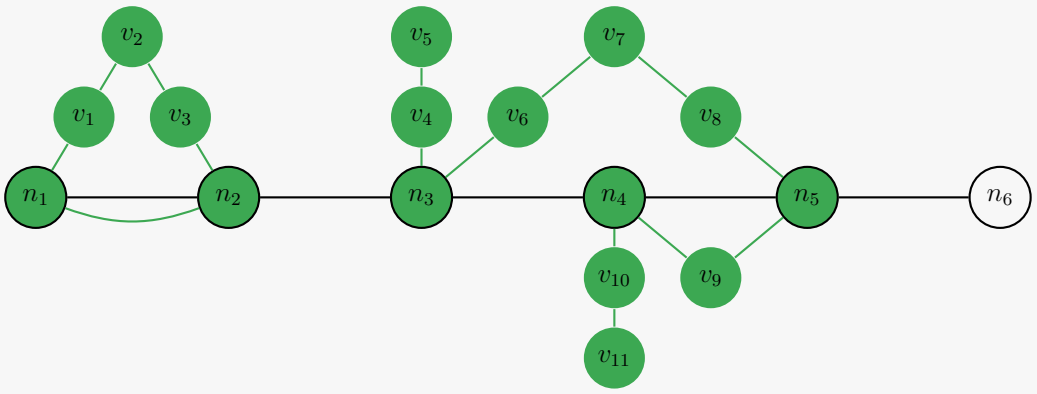

FIGURE 4 – Exemple d'un graphe P-noir G_k-vert

Exemple. Sur la figure 4, n_6 est un sommet noir, v_1 est un sommet vert et n_1 est un sommet noir et vert.

Définition 3.2 (Multigraphe P-noir P_k-vert). Un multigraphe P-noir G_k-

vert G est un multigraphe P-noir P_k-vert si et seulement si $\forall i \in \{1, ..., k\}, G_i$ est une chaine. Autrement dit, tous les graphes verts sont des chaines.

Notation 3.1. Dans un graphe P-noir G_k-vert G, pour un sommet x, $N_G^n(x)$ (respectivement $N_G^v(x)$) désigne l'ensemble des sommets noirs (respectivement verts) qui sont voisins de x. De plus, on note $deg_v(x) = |N_G^v(x)|$ et $deg_n(x) = |N_G^n(x)|$, les degrés vert et noir du sommet x.

Définition 3.3 (Chaine verte (resp. noire)). Une chaîne verte (resp. noire) d'un multigraphe P-noir G_k-vert G est une chaîne ne passant que par des arêtes vertes (resp. noires) de G.

Définition 3.4 (Mauvais cycle). Un cycle C d'un multigraphe P-noir G_k-vert est un mauvais cycle si et seulement si C contient au moins 2 arêtes noires, et que toutes ces arêtes noires sont successives le long de C.
Ainsi, C peut s'écrire $C = v_1...v_i n_1...n_j$, où $n_1, ..., n_j$ sont des sommets noirs, v_1 et v_i sont des sommets noirs et verts, et $v_2, ..., v_{i-1}$ sont des sommets verts.

Exemple. Sur la figure 4, le multigraphe noir et vert admet un seul mauvais cycle : $n_3, v_6, v_7, v_8, n_5, n_4$.

Définition 3.5 (Énumération linéaire). Soit G un multigraphe P-noir G_k-vert. Une énumération linéaire $(x_1, ..., x_{|V(G)|})$ des sommets de G est une énumération de $V(G)$ telle que :

- $\forall uv \in E(G_i), i \in \{1...k\}, \exists j$ tel que $(u = x_j$ et $v = x_{j+1})$ ou $(u = x_{j+1}$ et $v = x_j)$.
- $\forall uv \in E(P)$, posons $u = x_i$ et $v = x_j$ avec $i < j$. Alors $\nexists k \in \{i+1, ..., j-1\}$ tel que $x_k \in V(P)$

Définition 3.6 (Linéarisable). Soit G un multigraphe P-noir G_k-vert. G est dit linéarisable s'il admet une énumération linéaire.

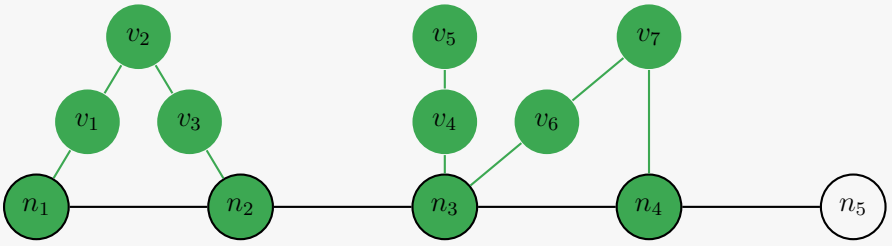

FIGURE 5 – Exemple d'un graphe P-noir G_k-vert linéarisable

Exemple. Sur la figure 4, le multigraphe noir et vert n'est pas linéarisable. En revanche, le multigraphe noir et vert de la figure 5 est linéarisable et admet l'énumération linéaire suivante : $n_1, v_1, v_2, v_3, n_2, v_5, v_4, n_3, v_6, v_7, n_4, n_5$.

3.2 Énumération linéaire d'un multigraphe P-noir G_k-vert

Dans cette partie, on s'intéresse au calcul, à partir d'un graphe P-noir G_k-vert linéarisable, d'une énumération linéaire de ses sommets.

Théorème 1. Soit G un multigraphe P-noir G_k-vert. G est linéarisable si et seulement si c'est un multigraphe P-noir P_k-vert sans mauvais cycle(s).

Ce théorème a été prouvé par L. De Mattéo dans ses travaux. L. De Mattéo a également écrit un algorithme polynomial donnant l'énumération linéaire d'un graphe P-noir P_k-vert sans mauvais cycle(s). Plus précisément, cet algorithme est en complexité $\mathcal{O}(n+m)$.

Corollaire 1. Soit G un multigraphe P-noir G_k-vert. G est linéarisable si et seulement si les trois propriétés suivantes sont vérifiées :
- G ne contient pas de mauvais cycle.
- $\forall x \in V(G), |N_G^v(x)| \leq 2$
- G ne contient pas de cycle vert.

Puisque tous les graphes ne sont pas linéarisables, on va s'intéresser à rendre ces graphes linéarisable, en leur supprimant un minimum d'arêtes vertes.

3.3 Linéarisation d'un multigraphe P-noir G_k-vert

On s'intéresse à présent au calcul, à partir d'un multigraphe P-noir G_k-vert G, de l'ensemble de cardinalité minimum $Cassure$ d'arêtes vertes, tel que $G - Cassure$ soit un multigraphe linéarisable. L. De Mattéo a démontré que ce problème est $\mathcal{NP}$-difficile.

Elle présente, dans ses travaux, une heuristique pour ce problème. L'algorithme 1 présente l'heuristique qu'elle a écrit. Cet algorithme repose sur le fait qu'un graphe P-noir G_k-vert linéarisable ne contient ni mauvais cycles, ni cycles verts, et que dans un tel graphe, chaque sommet a au plus 2 voisins verts.

Avec ces informations, on peut calculer un certain poids pour chaque arête verte. Ce poids augmente si l'arête est incluse dans de mauvaises structures : cycle vert, mauvais cycle ou degré vert trop élevé. Ces poids sont stockés dans la bijection w, déclarée à la ligne 1 de l'algorithme. La première boucle de l'algorithme calcule le poids de chaque arête verte du graphe.

L'algorithme construit ensuite un graphe G' initialisé au graphe noir de G, en lui ajoutant successivement chaque arête verte de G, à la condition que le graphe G' reste linéarisable.

Les arêtes étant prises par ordre croissant de leur poids, l'algorithme privilégiera les arêtes de poids faible lors de la reconstruction. En mémorisant les arêtes qui n'ont pas été ajoutées à G' dans l'ensemble $Cassure$, on obtient bien une solution au problème posé.

Algorithme 1 Heuristique de linéarisation d'un multigraphe P-noir G_k-vert

Données : G un multigraphe P-noir G_k-vert

Résultat : Un ensemble $Cassure$ d'arêtes vertes de G tel que $G - Cassure$ soit linéarisable.

Bijection $w : E(G_i) \to \mathbb{N}, \forall i$
pour $uv \in E(G_i), \forall i$ **faire**
  $w(uv) \leftarrow 0$
  si $|N^v(u)| > 2$ **alors**
    $w(uv) \leftarrow w(uv) + |N^v(u)| - 2$
  fin si
  si $|N^v(v)| > 2$ **alors**
    $w(uv) \leftarrow w(uv) + |N^v(v)| - 2$
  fin si
  $c_{vert} \leftarrow \#\{$ Cycle vert C, C contenant uv $\}$
  $w(uv) \leftarrow w(uv) + c_{vert}$
  $c_{mauvais} \leftarrow \#\{$ Mauvais cycle C, C contenant uv $\}$
  $w(uv) \leftarrow w(uv) + c_{mauvais}$
fin pour
Trier dans le tableau T les arêtes de w par ordre croissant.
Ensemble $Cassure \leftarrow \emptyset$
Multigraphe $G' \leftarrow P$
pour $uv \in T$ **faire**
  si $G' \cup \{uv\}$ est linéarisable **alors**
    $G' \leftarrow G' \cup \{uv\}$
  sinon
    $Cassure \leftarrow Cassure \cup \{uv\}$
  fin si
fin pour
renvoyer $Cassure$;

4 Dénombrement des cycles d'un graphe

Cette partie s'intéresse à un point précis de l'heuristique que nous venons de présenter, le dénombrement des cycles. Le dénombrement des cycles simples d'un graphe est un problème connu $\#\mathcal{P}$-complet. Il est cependant possible d'affiner l'analyse par une classe de complexité paramétrée selon des paramètres du graphe en entrée. Nous verrons comment intégrer les travaux de cette partie pour l'heuristique dans la Partie 5.

L'article *A General Purpose Algorithm for Counting Simple Cycles and Simple Paths of Any Length* [4] décrit un algorithme pour le dénombrement de cycles de longueur fixée puis fait l'analyse de sa complexité. Il ne prend en compte que les cycles simples dans un graphe c'est-à-dire les cycles qui n'utilisent pas deux fois le même sommet.

4.1 Dénombrement des cycles de longueur ℓ

L'algorithme est basé sur la relation entre le nombre de marches et de cycles simples sur un graphe présenté dans un autre article de P. Giscard et P. Rochet [5]. Cette relation permet d'obtenir une formule nous indiquant le nombre de cycles de taille ℓ dans un graphe. Nous présentons une formule dérivée de celle de l'article pour le cas des graphes non orientés :

$$\gamma(\ell) = \frac{(-1)^\ell}{2\ell} \sum_{H \prec_{conn} G} \binom{|N(H)|}{\ell - |V(H)|} (-1)^{|V(H)|} Tr(A_H^\ell) \tag{1}$$

La formule se décompose avec :
- $\gamma(\ell)$ le nombre de cycles de taille ℓ.
- $\sum_{H \prec_{conn} G}$ la somme parcourant l'ensemble des sous-graphes connexes notés H induits de G.
- $|V(H)|$ désigne le nombre de sommets du graphe H.
- $|N(H)|$ est le nombre de voisins de H dans G. Un sommet s est voisin de H dans G si et seulement si s appartient à $V(G)$ et n'appartient pas à $V(H)$ et s'il existe au moins une arête de s à un sommet de H.
- $Tr(A_H^\ell)$ est la trace de la matrice d'adjacence de H. C'est-à-dire la somme des éléments de sa diagonale principale.

Exemple. Pour mieux appréhender l'équation (1), nous présentons un exemple de son application avec le graphe G de la Figure 6 pour $\ell = 3$.

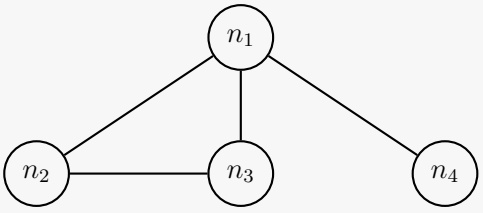

FIGURE 6 – Exemple d'un graphe G avec un cycle de taille 3

FIGURE 7 – Les quatre sous-graphes induits de G de taille 1

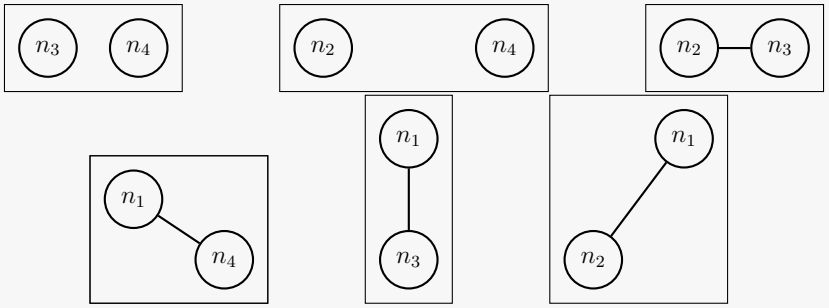

FIGURE 8 – Les six sous-graphes induits de G de taille 2

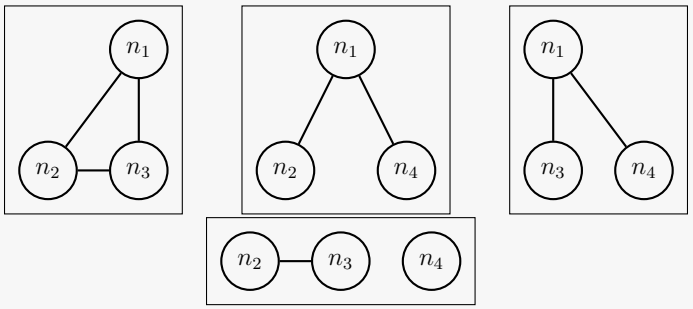

FIGURE 9 – Les quatre sous-graphes induits de G de taille 3

Pour la somme $\sum_{H \prec_{conn} G}$, nous allons utiliser respectivement les graphes des figures 7, 8 et 9. Nous remarquons que les sous-graphes induits de taille 1 auront tous une matrice d'adjacence nulle (il n'y a pas de boucles), nous pouvons donc les exclure de la somme.

$$\gamma_G(3) = \frac{(-1)^3}{2 \times 3} \sum_{H \prec_{conn} G} \binom{|N(H)|}{3 - |V(H)|} (-1)^{|H|} Tr(A_H^3)$$

$$= \frac{-1}{6}(\binom{2}{1} Tr(\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}^3) + \binom{2}{1} Tr(\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}^3) + \binom{1}{1} Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^3) + \binom{2}{1} Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^3)$$

$$+ \binom{2}{1} Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^3) + \binom{2}{1} Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^3) + \binom{1}{0} Tr(\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}^3)$$

$$+ \binom{1}{0} Tr(\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}^3) \binom{1}{0} Tr(\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}^3) + \binom{1}{0} Tr(\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}^3))$$

$$= \frac{-1}{6}(Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}) + 2Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}) + 2Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}) + 2Tr(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix})$$

$$+ Tr(\begin{bmatrix} 2 & 3 & 3 \\ 3 & 2 & 3 \\ 3 & 3 & 2 \end{bmatrix}) + Tr(\begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}) + Tr(\begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}) + Tr(\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}))$$

$$= \frac{-1}{6} \cdot (0 + 0 + 0 + 0 - 6 - 0 - 0 - 0) = 1$$

Nous obtenons ainsi un seul cycle ce qui correspond au nombre de cycles de taille 3 dans G.

4.2 Complexité

4.2.1 Complexité simple

Théorème 2. Soit $t(|S_\ell|)$ la complexité de trouver l'ensemble des sous-graphes connexes induits de G sur au plus ℓ sommets, Δ le degré maximal de G. La complexité du dénombrement $\gamma(\ell)$ des cycles de taille ℓ de G est :

$$\mathcal{O}(t(|S_\ell|) + (\ell^\omega + \ell\Delta)|S^\ell|)$$

Remarque. ω désigne l'exposant de la multiplication des matrices, c'est-à-dire le plus faible exposant ω tel que la complexité de la multiplication de deux matrices de taille $n \times n$ soit $\mathcal{O}(n^\omega)$. En 2014, F. Le Gall a proposé un nouvel algorithme démontrant que $\omega < 2.3728639$ [6].

Démonstration. Premièrement, les coefficients binomiaux dans l'équation (1) sont non nuls si et seulement si $|V(H)| \leq \ell \leq |N(H)| + |V(H)|$. Cela signifie que seuls les sous-graphes connexes notés H induits de G pour lesquels $|V(H)| \leq \ell$ sont nécessaires pour calculer la somme. La complexité de la recherche de tous les H vérifiant cette condition est notée $\mathcal{O}(t(|S_\ell|)$.

Dans un second temps, il faut évaluer pour chaque graphe noté H parmi les $|S_\ell|$ obtenus le nombre $|N(H)|$ de voisins de H dans G et les chemins de taille ℓ de H. Dénombrer les voisins est de complexité $\mathcal{O}(\ell\Delta)$ et trouver les chemins de taille ℓ prend $\mathcal{O}(\ell^\omega)$. D'où le résultat final de complexité $\mathcal{O}(t(|S_\ell|) + (\ell^\omega + \ell\Delta)|S^\ell|)$. □

La recherche de l'ensemble des sous-graphes connexes induits de G sur au plus ℓ sommets peut s'effectuer en utilisant la recherche inversée [7] en temps $\mathcal{O}(|V(G)| + |E(G)| + \ell^2|S_\ell|)$. Un algorithme présenté par K. M. Elbassioni [8] permet d'obtenir une meilleure complexité. Cet algorithme énumère l'ensemble des graphes en $\mathcal{O}((\ell \min(n-\ell, \ell\Delta))^2(\Delta + log(\ell)))$. Cependant, aucune implémentation vérifiée de cet algorithme n'existe à ce jour.

4.2.2 Complexité paramétrée

En utilisant la recherche inversée, nous avons donc un algorithme pour le dénombrement des cycles d'un graphe de complexité $\mathcal{O}(|V(G)| + |E(G)| + \ell^2|S_\ell| + (\ell^\omega + \ell\Delta)|S^\ell|)$.

En bornant $|S^\ell| \leq 2^{|V(G)|}$, la complexité obtenue n'est pas encore intéressante. Les auteurs de l'article bornent donc $|S^\ell|$ d'après les résultats de l'article de R. Uehara [9]. Soit Δ le degré maximum de G alors :

$$|S_\ell| \leq \mathcal{O}(|V(G)|\frac{\Delta^\ell}{(\Delta-1)\ell^2}))$$

Soit le problème du dénombrement des cycles de longueur d'un graphe G paramétré par $k = \Delta(G)$ le degré maximum de G. Alors le nombre d'arêtes $|E(G)|$ dans G est inférieur à $|V(G)|\Delta$. On obtient ainsi la complexité :

$$\begin{aligned}
&\mathcal{O}(|V(G)|(\Delta+1) + (\ell^\omega + \ell\Delta)|V(G)|\frac{\Delta^\ell}{(\Delta-1)\ell^2}) \\
&= \mathcal{O}(|V(G)|\Delta + |V(G)|(\ell^{-1}\Delta + \ell^{\omega-2})\Delta^{\ell-1}) \\
&= \mathcal{O}((\Delta + (\ell^{-1}\Delta + \ell^{\omega-2})\Delta^{\ell-1})|V(G)|)
\end{aligned}$$

Cet algorithme montre ainsi que le problème du dénombrement des cycles de taille maximum ℓ dans un graphe G paramétré par son degré maximum Δ est FPT. En suivant la formule de la classe FPT, nous avons une complexité $\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)})$ avec $n^{\mathcal{O}(1)}$ qui est ici $|V(G)|$ et une fonction calculable $f(\Delta) = \Delta + (\ell^{-1}\Delta + \ell^{\omega-2})\Delta^{\ell-1}$.

4.3 Cycles de taille ℓ passant par un sommet

L'article *A General Purpose Algorithm for Counting Simple Cycles and Simple Paths of Any Length* [4] propose également un algorithme pour le dénombrement de cycles de longueur ℓ passant par un sommet déterminé. Il se base également sur une formule de l'article de Pierre-Louis Giscard et Paul Rochet [5] donnant le nombre $\gamma_i(\ell)$ de cycles de taille ℓ passant par i :

$$\gamma_i(\ell) = (-1)^\ell \sum_{\substack{H \prec_{conn} G \\ i \in H}} \binom{|N(H)|}{\ell - |V(H)|} (-1)^{|V(H)|} (A|_H^\ell)_{ii} \tag{2}$$

La formule reprend la notation de l'équation (1) en y ajoutant $A|_H$ représentant la matrice d'adjacence G restreinte au graphe induit H :

$$(\mathrm{A}|_H)_{ij} = \begin{cases} \mathrm{A}_{ij} & \text{si } i,j \in H \\ 0 & \text{sinon} \end{cases} \qquad i,j \in G$$

4.4 Implémentation de l'algorithme

Il existe deux implémentations de l'algorithme. La première réalisée par les auteurs de l'article est en Matlab[10], un langage propriétaire payant. La deuxième est en Python[11] et ne permet pas à l'heure actuelle de dénombrer les cycles passant par un sommet fixé.

Nous proposons avec ce rapport une version C++ qui sera utilisée dans la suite de nos travaux, eux aussi dans ce langage. Celle-ci permet de dénombrer les cycles simples du graphe, ou plus spécifiquement ceux passant par un sommet déterminé. Le code source est disponible sur *un projet Github* [12] que nous mettons à disposition indépendamment pour sa réutilisation dans d'autres projets. Nous intégrons aussi directement ce code à notre projet [13].

Notre implémentation prend en paramètre la matrice d'adjacence du graphe noté G, une longueur l déterminant la longueur maximum des cycles à dénombrer (qui peut être $|V(G)|$ pour dénombrer tous les cycles), et facultativement un sommet par lequel les cycles dénombrés doivent passer. Notre implémentation permet aussi d'obtenir un tableau nous donnant pour toute paire de sommets le nombre de chemins allant de l'un à l'autre.

Pour le calcul de la trace de l'équation (1), l'implémentation utilise une propriété remarquable des matrices. La trace d'une matrice à la puissance k est égale à la somme de ses valeurs propres à la puissance k. Ainsi, en calculant les valeurs propres, nous évitons de calculer les matrices d'adjacences induites H à la puissance l.

5 Dénombrement des cycles pour la linéarisation

Dans cette partie, nous faisons le lien entre le rapport de L. De Mattéo [3] et l'article *A General Purpose Algorithm for Counting Simple Cycles and Simple Paths of Any Length* [4].

5.1 Complexité de l'heuristique

Il est d'abord important d'analyser la complexité de l'heuristique de linéarisation d'un multigraphe P-noir G_k-vert décrite dans l'algorithme 1 de la page 13.

De la ligne 1 à 14, l'heuristique fait la somme pour chaque arrête uv de $|N^v(u)| - 2$, de $|N^v(v)| - 2$, du nombre de cycles verts et de mauvais cycles contenant uv. Il est possible de déterminer $|N^v(u)|$, de $|N^v(v)|$ en temps $\mathcal{O}(m)$. Nous reprendrons la complexité du dénombrement des cycles verts et des mauvais cycles plus loin.

De la ligne 15 à 25, il n'est question que du tri de l'ensemble des arêtes selon le poids attribué et de générer un ensemble $Cassure$ tel que le multigraphe G privé des éléments de $Cassure$ soit linéarisable. Il est possible de vérifier si un graphe est linéarisable en temps $\mathcal{O}(n+m)$ [partie 3.2]. Cette opération s'effectue pour chaque arête verte, la complexité est donc en $\mathcal{O}(m(n+m))$.

La complexité de l'algorithme dépend donc de la complexité du dénombrement de cycles verts et de mauvais cycles (lignes 10 et 12 de l'algorithme). L'implémentation proposée par L. De Mattéo effectue ce dénombrement en considérant les cycles utilisant deux fois le même sommet.

Nous présentons dans la suite une méthode de dénombrement des cycles verts et mauvais cycles en ne considérant que les cycles simples en complexité FPT, où le paramètre est la taille du plus grand cycle à considérer. Les autres opérations s'effectuant en temps polynomial, cela conduit à une implémentation en complexité FPT de l'heuristique.

5.2 Dénombrement des cycles

Nous allons utiliser les résultats présentés dans la Partie 4, afin de dénombrer pour chaque arête verte, les cycles verts et les mauvais cycles passant par cette arête. Dans la suite, on fixe G un graphe P-noir G_k-vert et uv une arête verte de G. Les méthodes proposées dans la Partie 4 étant de complexité FPT, nous aboutirons au résultat voulu.
Dans ce qui suit, nous illustrerons les méthodes utilisées sur le graphe suivant :

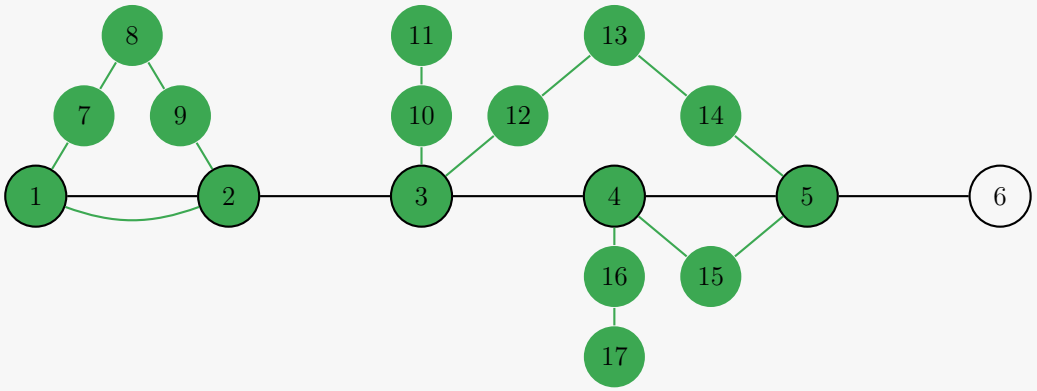

FIGURE 10 – Exemple d'un graphe P-noir G_k-vert

5.2.1 Dénombrer les cycles verts

C'est le cas le plus simple, on considère le graphe $G' = G - E(P)$ correspondant au graphe initial G privé des arêtes noires. Les cycles de ce graphe correspondent aux cycles verts de G. Pour calculer le nombre de cycles verts passant par uv, il suffit de faire la différence entre le nombre de cycles dans G' et le nombre de cycles dans $G' - \{uv\}$.

Exemple. Sur la figure 10, G' correspond au graphe suivant :

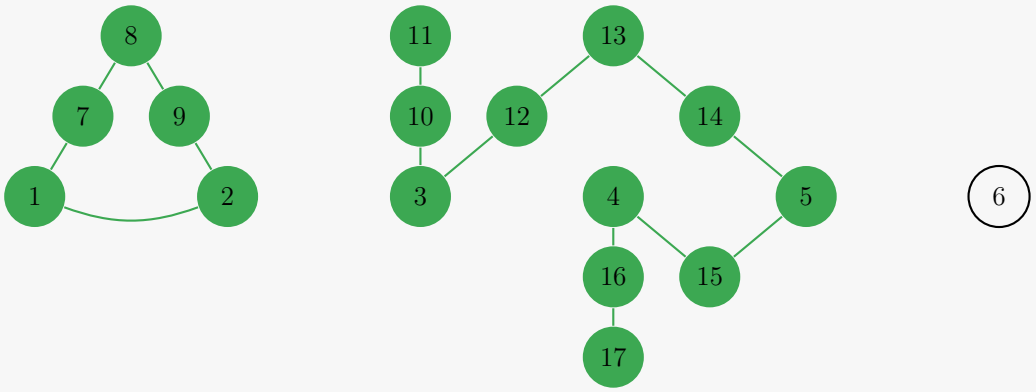

FIGURE 11 – Exemple de dénombrement des cycles verts

On remarque que l'unique cycle restant est l'unique cycle vert du graphe de départ.

5.2.2 Dénombrer les mauvais cycles

De nouveau, il suffit de mettre en place une méthode de dénombrement des mauvais cycles dans l'ensemble du graphe. Ensuite, on peut calculer la différence entre le nombre de mauvais cycles dans G, puis le nombre de mauvais cycles dans $G - \{uv\}$.

Soient alors n_1, n_2 deux sommets noirs distincts de G.
On définit E_{n_1,n_2} comme l'ensemble des arêtes vertes de G dont les extrémités sont soit des sommets verts, soit n_1 ou n_2.
On considère ensuite le graphe $H_{n_1,n_2} = (V(G), E(P) \cup E_{n_1,n_2})$, c'est-à-dire G réduit aux arêtes noires et aux arêtes de E_{n_1,n_2}.

On note $c(n_1, n_2)$ le nombre de cycles passant par n_3 dans H_{n_1,n_2}, où n_3 est un sommet quelconque entre n_1 et n_2 le long de P.
Si n_1 et n_2 sont voisins, on fixe $c(n_1, n_2) = 0$ (car un tel n_3 n'existe pas).

Exemple. Sur la figure 10, $H_{3,5}$ correspond au graphe de la figure 12, et n_3 correspond au sommet 4 :

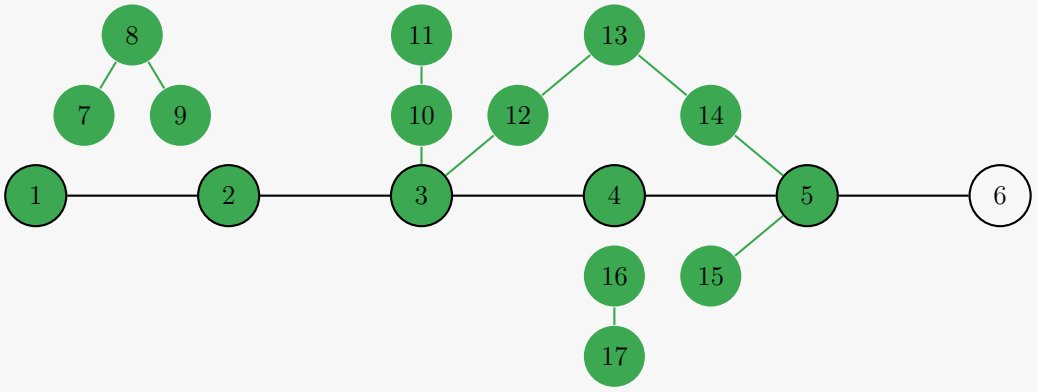

FIGURE 12 – Exemple de dénombrement des mauvais cycles

On remarque que l'unique cycle restant est le mauvais cycle $3, 4, 5, 14, 13, 12$ et qu'il passe par le sommet 4.

Théorème 3. Le nombre de mauvais cycles dans G est égal à

$$\sum_{\{n_1,n_2\}\subseteq V(P)} c(n_1, n_2)$$

.

Preuve. 1. *Tous les cycles dénombrés sont des mauvais cycles*

En effet, soient n_1, n_2 deux sommets noirs non voisins, et n_3 un sommet entre n_1 et n_2 le long de P (si n_1 et n_2 sont voisins, $c(n_1, n_2) = 0$ et aucun cycle n'est dénombré).

Notons $P = x_1, x_2, ..., x_i, n_1, ..., n_3, ..., n_2, x_k..., x_{|V(P)|}$ les sommets de la chaine noire.

Alors un cycle passant par n_3 dans H_{n_1,n_2} passe également par n_1 et n_2, et ne passe pas par x_i ni par x_k.

En effet, tout cycle emprunte au moins une arête verte (car les arêtes noires forment une chaine) et les seuls sommets noirs adjacents à une arête verte sont n_1 et n_2.

Finalement, un cycle passant par n_3 est de la forme $n_1, ..., n_3, ..., n_2, v_1, ...v_l$ où $v_1, ..., v_l$ sont des sommets verts. $n_1, ..., n_3, ..., n_2$ contenant au moins deux arêtes, c'est un mauvais cycle.

2. *Un mauvais cycle est dénombré exactement une fois.* Un mauvais cycle est un cycle $C = v_i...v_j n_k...n_l$ où $P_1 = v_i...v_j$ est une chaine verte à au moins un élément et $n_k...n_l$ une chaine noire à au moins un élément.
   Alors C sera compté une fois dans $c(n_k, n_l)$.
   Puisque n_k et n_l sont les seuls sommets noirs de C voisins de sommets verts, aucune autre paire de sommets $n_{k'}, n_{l'}$ ne dénombrera C.

□

6 Un algorithme exact

6.1 Motivations

L'algorithme proposé par L. De Mattéo est une heuristique qui ne donne donc pas nécessairement une solution optimale. Bien que la linéarisation d'un multigraphe P-noir G_k-vert soit un problème $\mathcal{NP}$-difficile, il est utile de réaliser un algorithme exact pour ce problème. Une implémentation de cet algorithme exact est intéressante pour comparer les solutions fournies par l'heuristique à des solutions exactes. Il est possible pour cela d'écrire un algorithme naïf, testant l'ensemble des solutions possibles.

6.2 L'algorithme

Algorithme 2 Linéarisation d'un multigraphe P-noir G_k-vert

Données : G un multigraphe P-noir G_k-vert

Résultat : Un ensemble $Cassure$ minimum d'arêtes vertes de G tel que $G - Cassure$ soit linéarisable.

1: $l \leftarrow 0$
2: $Trouve \leftarrow Faux$
3: **tant que** (non $Trouve$) **faire**
4: **pour** tout F sous-ensemble des arêtes vertes de taille l **faire**
5: **si** $G - F$ est linéarisable **alors**
6: $Cassure \leftarrow F$
7: $Trouve \leftarrow Vrai$
8: **fin si**
9: **fin pour**
10: $l \leftarrow l + 1$
11: **fin tant que**
12: renvoyer $Cassure$;

6.3 Correction et Complexité

6.3.1 Terminaison

L'algorithme termine. De manière triviale, on peut remarquer que chaque boucle *Pour* termine, car le nombre de sous-ensembles est fini, et la boucle *Tant que* terminera aussi car, au pire pour k égal au nombre d'arêtes vertes, F désigne l'ensemble des arêtes vertes. Ainsi, $G - F$ est réduit à P et à des sommets isolés. Ce graphe est évidemment linéarisable.

6.3.2 Correction

Les sous-ensembles F sont pris par ordre croissant de leur taille. Ainsi, lorsqu'un ensemble $Cassure$ est renvoyé, il n'existe pas d'ensemble F de taille strictement inférieure à $Cassure$ répondant au problème.

6.3.3 Complexité

On a vu précédemment que dans le pire des cas, l'algorithme s'arrêtait après $m_v + 1$ itérations au plus de la boucle $Tant\ que$, où m_v désigne le nombre d'arêtes vertes.
Ensuite, à l'itération k de la boucle $Tant\ que$, la boucle $Pour$ va considérer chaque sous-ensemble de taille k, ce qui donne $\binom{m_v}{k}$ itérations.
Enfin, vérifier si $G - F$ est linéarisable se fait en $\mathcal{O}(n + m)$ opérations.
Le nombre total d'opérations est donc borné par :

$$\mathcal{O}(n+m).\sum_{k=0}^{m_v}\binom{m_v}{k}$$

$$= \mathcal{O}\Big((n+m).2^{m_v}\Big)$$

6.4 Optimisation

Dans l'état actuel, l'algorithme exact proposé n'est pas envisageable en pratique. En effet, avec une complexité exponentielle, il ne terminera pas en temps

raisonnable sur de grands exemples. Nous proposons alors une optimisation de cet algorithme.

Théorème 4. Soit G un multigraphe P-noir G_k-vert connexe, et H l'ensemble des arêtes vertes uv de G telles que :
- $deg(u) \leq 2$
- $deg(v) \leq 2$
- u et v sont des sommets verts

Alors il existe une solution F au problème de la linéarisation tel que $F \cap H = \emptyset$

Preuve. On rappelle [Corrolaire 1] qu'un graphe est linéarisable si et seulement si :

1. $\forall x \in V(G), |N_v(x)| \leq 2$
2. G ne contient pas de cycle vert
3. G ne contient pas de mauvais cycle

Soit alors F une solution au problème de la linéarisation telle que $|F \cap H|$ soit minimum.
Supposons que $|F \cap H| > 0$, et montrons le résultat par l'absurde :
Soit alors $uv \in F \cap H$, x un sommet noir de G et $\{u = x_1, x_2, ..., x_l = x\}$ un chemin de u à x (ce chemin existe car G est connexe).
Soit $x_{i-1}x_i$ la première arête qui n'est pas dans H. Cette arête existe bien car $x_{l-1}x \notin H$ (car x est un sommet noir).
Alors tout cycle passant par uv passe également par $x_{i-1}x_i$, en particulier les cycles verts et les mauvais cycles.
Ainsi, au vu de la propriété rappelée en début de preuve, on remarque que

$$F' = F \backslash \{uv\} \cup \{x_{i-1}x_i\}$$

est aussi une solution au problème de la linéarisation.
Or $|F' \cap H| = |F \cap H| - 1$, ce qui contredit la définition de F.
Finalement, $F \cap H = \emptyset$.

□

On peut alors reprendre l'algorithme précédent, mais en considérant $E_v \backslash H$ plutôt que E_v (où E_v est l'ensemble des arêtes vertes du graphe) puisque par le théorème précédent, une solution est incluse dans $E_v \backslash H$. La complexité devient alors :

$$\mathcal{O}(n+m).2^{m_v - |H|}$$

.

7 Performances des algorithmes

Nous allons dans cette partie présenter les résultats de différents tests de performance réalisés avec les algorithmes présentés précédemment.

7.1 Algorithme glouton

Afin de comparer les résultats, nous avons ajouté un algorithme glouton qui servira de référence pour les autres. Son fonctionnement est simple, il ajoute les arêtes vertes une par une, à condition que le graphe reste linéarisable. Il correspond en fait à l'heuristique présentée à la fin de la section 3, mais sans calculer un ordre particulier sur les arêtes.

Algorithme 3 Algorithme glouton de linéarisation

Données : G un multigraphe P-noir G_k-vert
Résultat : Un ensemble $Cassure$ d'arêtes vertes de G tel que $G - Cassure$ soit linéarisable.

```
1: Ensemble Cassure ← ∅
2: Multigraphe G' ← P
3: pour toute arête verte uv de G faire
4:     Ajouter dans G' l'arête uv
5:     si G' n'est plus linéarisable alors
6:         Cassure ← Cassure ∪ {uv}
7:         Retirer de G' l'arête uv
8:     fin si
9: fin pour
10: renvoyer Cassure ;
```

Il est important de remarquer que cet algorithme n'utilise aucune propriété sur le graphe pour déterminer les arêtes à retirer. Il permettra de vérifier si l'heuristique propose des résultats sensiblement meilleurs. On a montré qu'il est possible de vérifier si un graphe est linéarisable en temps $\mathcal{O}(n+m)$. Cet algorithme est donc en complexité $\mathcal{O}(m(n+m))$: il est polynomial.

7.2 Algorithmes pour le test

Pour réaliser nos tests, nous avons sélectionné plusieurs algorithmes et différents paramétrages de ces algorithmes.

— **Heuristique basée sur le dénombrement de cycles**

Premièrement nous testons l'implémentation proposée dans les travaux de L. De Mattéo de l'heuristique basée sur le dénombrement de cycles verts et de mauvais cycles [Algorithme 1].

Deuxièmement, nous testons l'implémentation proposée durant nos travaux sans limite de taille des cycles. Finalement, nous ajoutons un test avec l'implémentation proposée durant nos travaux avec une limite de la taille des cycles à considérer de 10. Cette limite permet de limiter le temps de calcul au prix d'un dénombrement des cycles moins précis.

— **Algorithme exact**

Le second algorithme considéré est l'algorithme exact [Algorithme 2] tel qu'il est défini dans ce rapport. Nous l'appliquons d'abord sur toutes les arêtes vertes. Puis sur l'ensemble d'arêtes ayant les propriétés du Théorème 4. Rappelons que dans le second cas, le résultat reste exact.

— **Algorithme glouton**

Le dernier algorithme testé est l'algorithme glouton [Algorithme 3]. Cet algorithme polynomial sert de référence pour les autres tests.

7.3 Jeux de données

Les algorithmes seront comparés sur plusieurs données en entrées. Dans un premier temps avec plusieurs graphes de densité assez faible, extraits de données génétiques de plusieurs espèces, et contenant de 17 à 1873 sommets :

— *G1 :* un graphe possédant 17 sommets et 15 arêtes vertes.
— *G2 :* un graphe possédant 22 sommets et 26 arêtes vertes.
— *G3 :* un graphe possédant 34 sommets et 28 arêtes vertes.
— *G4 :* un graphe possédant 28 sommets et 29 arêtes vertes.
— *G5 :* un graphe possédant 34 sommets et 32 arêtes vertes.
— *G6 :* un graphe possédant 34 sommets et 33 arêtes vertes.

Graphe extrait du moustique : un graphe possédant 462 sommets et 347 arêtes vertes. Cet exemple est extrait des données génétiques du moustique.

Graphe extrait du canard : un graphe possédant 1873 sommets et 988 arêtes vertes. Cet exemple est extrait des données génétiques du canard.

7.4 Résultats des tests

Tous les tests ont été réalisés dans les mêmes conditions, avec le processeur *Dual-Core Intel ®Core^{TM}i5-6200U* de fréquence *2.30GHz* et *8GB* de mémoire vive. L'objectif n'est pas de comparer nos résultats avec ceux d'autres tests, mais de comparer les tests entre eux. Nous invitons les personnes qui le souhaitent à reproduire les tests en donnant accès au code source, au programme de test et à l'ensemble des jeux de données sur *notre projet Github* [13].

Cette partie présente les résultats bruts sous forme de tableau. La partie suivante tire les conclusions de ces résultats. Dans les tableaux, un tiret représente l'absence de résultat pour dépassement de la durée limite de 10h.

	Instance :	G1		G2		G3	
Algorithme	Implémentation	Solution	Durée	Solution	Durée	Solution	Durée
Heuristique	Tous les cycles	**2**	30ms	**9**	1s	**3**	60ms
	Cycles simples <10	**2**	200ms	**9**	30s	**3**	1s
	Tous les cycles simples	**2**	350ms	**9**	267s	**3**	4s
Exact	Toutes les arêtes vertes	**2**	1ms	**9**	30s	**3**	92ms
	Sur l'ensemble d'arêtes	**2**	1ms	**9**	14s	**3**	30ms
Glouton		**2**	1ms	**10**	2 ms	**3**	4ms

Tableau 1 – Résultats des tests sur les instances G1, G2 et G3

	Instance :	G4		G5		G6	
Algorithme	Implémentation	Solution	Durée	Solution	Durée	Solution	Durée
Heuristique	Tous les cycles	**10**	394ms	**8**	185ms	**9**	246ms
	Cycles simples <10	**10**	287s	**8**	73s	**8**	103s
	Tous les cycles simples	**10**	8h	**8**	5h	**-**	-
Exact	Toutes les arêtes vertes	**10**	152s	**7**	41s	**8**	104s
	Sur l'ensemble d'arêtes	**10**	4s	**7**	17s	**8**	35s
Glouton		**10**	2ms	**8**	2ms	**8**	2ms

Tableau 2 – Résultats des tests sur les instances G4, G5 et G6

		Extrait du moustique		Extrait du canard	
Algorithme	Implémentation	Solution	Durée	Solution	Durée
Heuristique	Tous les cycles	**6**	44s	**87**	57s
	Cycles simples <10	**6**	1h30mn	-	-
	Tous les cycles simples	**-**	-	-	-
Exact	Toutes les arêtes vertes	**-**	-	-	-
	Sur l'ensemble d'arêtes	**6**	38mn	-	-
Glouton		**6**	180ms	**90**	5s

Tableau 3 – Résultats des tests sur l'instance extrait des données génétiques du moustique et du canard

7.5 Observations sur les performances

7.5.1 Heuristique avec et sans cycles simples

Les solutions de l'heuristique en dénombrant tous les cycles ou seulement les cycles simples sont sensiblement les mêmes dans le Tableau 1 et le Tableau 2.

Le temps de calcul est cependant plus faible pour les tests en dénombrant tous les cycles. La nouvelle implémentation dénombre les cycles simples qui est un problème de plus grande complexité. Cette différence de complexité entre l'implémentation de L. De Mattéo et la nouvelle implémentation s'illustre par les différences de durée qui deviennent significatives sur de plus grands graphes. Dans le Tableau 3, la taille des données extraites de la génétique du canard ne permet pas à la nouvelle implémentation d'obtenir un résultat dans le temps

imparti.

7.5.2 Performance de l'heuristique

Il est intéressant de remarquer que les résultats de l'heuristique sont sensiblement les mêmes que ceux de l'algorithme glouton. Ils sont de meilleure qualité dans le graphe $G2$ et pour le graphe extrait de la génétique du canard du Tableau 3. En revanche, les résultats sont de moins bonne qualité pour l'heuristique que l'algorithme glouton pour le graphe $G6$.

Ces observations posent des questions sur la performance de l'heuristique pour la résolution du problème. Il semble que l'algorithme glouton obtienne des résultats de qualité comparable à celle de l'heuristique, avec une durée plus faible.

7.5.3 Durée de l'algorithme exact

Les tests montrent que l'algorithme exact est bien plus rapide en pratique que ce qui était calculé durant l'étude de la complexité [6.3.3]. Pour le graphe 3 dans le Tableau 1, une solution optimale est obtenu en seulement 152 ms. Pourtant, le graphe possède 34 sommets et 43 arêtes dont 28 arêtes vertes. Nous avons dans le pire des cas une complexité de :

$$\mathcal{O}(n+m) \cdot 2^{m_v} = \mathcal{O}(34+43) \cdot 2^{28} \approx 2 \cdot 10^{10}$$

Alors, si l'on considère une borne inférieure en estimant qu'une étape de calcul prend une nanoseconde, nous devrions être à plus de 20 000 ms de calcul.

Pour expliquer ce contraste, intéressons-nous à la complexité paramétrée par la valeur de la solution optimale. Puisque l'on étudie les solutions par taille croissante et que la taille de la solution trouvée est 3 :

$$\mathcal{O}(n+m)\cdot\mathcal{O}(\text{Nombres de solutions explorées}) \approx 77\cdot\left(\binom{28}{3}+\binom{28}{2}+\binom{28}{1}\right) \approx 3\cdot 10^4$$

Si l'on considère une borne inférieure en estimant qu'une étape de calcul prend une nanoseconde, nous obtenons $\approx 30\mu s$, loin des 20 000 ms estimées sans ce paramètre.

7.5.4 Optimisation de l'algorithme exact

On observe dans le Tableau 2 que l'optimisation présentée dans la section 6.4 permet de réduire le temps de calcul de façon significative. Le ratio entre la durée de calcul avec et sans l'optimisation est d'environ 3 pour les graphes $G5$ et $G6$ et jusqu'à 38 pour le graphe $G4$. Cette différence s'explique par la topologie du graphe.

Cette optimisation permet à l'algorithme exact de terminer dans le temps imparti sur le graphe extrait des données génétiques du moustique dans le Tableau 3. L'algorithme exact avec optimisation termine en 38 minutes alors que sans optimisation, il dure plus de 10 heures.

8 Conclusion

8.1 Bilan de nos travaux

8.1.1 Partie théorique

Nos travaux possèdent un grand aspect théorique avec des apports dans le domaine de la théorie des graphes pour le consensus d'ordre. Ce rapport illustre avec des figures et des exemples les points clés de la modélisation du problème et des outils que nous utilisons pour sa résolution.

Nous avons produit plusieurs théorèmes (théorèmes 3 et 4) utiles pour la linéarisation et le consensus d'ordre. Nous avons prouvé chacun des théorèmes que nous avons produits, et avons fourni une référence vers ceux issus d'autres travaux. Les algorithmes majeurs utiles à la résolution du problème sont présentés durant le rapport, accompagné du calcul de leur complexité. Les algorithmes intermédiaires sont disponibles en Annexes [9.1]. À l'exception de l'algorithme 1, nous avons écrit l'ensemble des algorithmes disponibles dans ce rapport.

8.1.2 Développement

Notre projet a aussi une partie de développement. L'ensemble de notre programme est mis à disposition sur *notre projet Github* [13]. Il est accompagné d'exemples et d'explications pour son utilisation et sa modification. Afin de vérifier la qualité des modifications, chaque mise à jour du code source sur le serveur entraîne automatiquement une vérification de la compilation sur une machine virtuelle en ligne.

De plus, une documentation de tout le projet est automatiquement générée. Pour cela, la totalité de notre code est couverte par une description précise des fonctions, méthodes et paramètres. Le serveur déclenche automatiquement la génération d'une documentation en utilisant *Doxygen* puis publie automatiquement la nouvelle documentation. Vous pouvez retrouver cette documentation complète et mise à jour sur *hareski.github.io/ConsensusOrdres*.

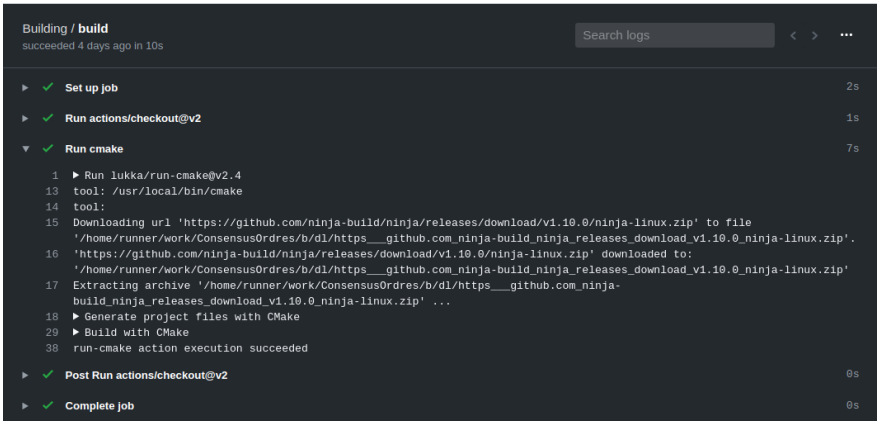

FIGURE 13 – Compilation automatique du projet sur le serveur

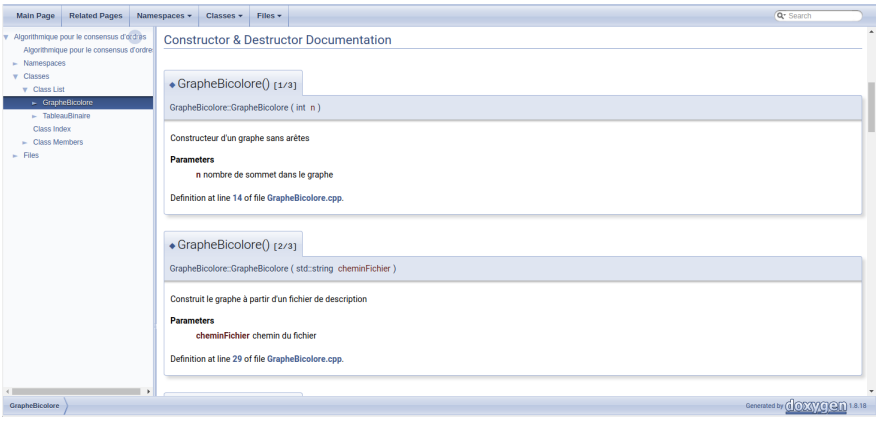

FIGURE 14 – Documentation *Doxygen mise en ligne automatiquement*

8.1.3 Expérimentations

Finalement, nos travaux comportent aussi une partie d'expérimentation. Les tests des algorithmes dans la Partie 7 permettent de tirer de nombreuses conclusions. Ils remettent en question l'heuristique en comparant la qualité des résultats avec l'algorithme glouton. De plus, on observe que l'algorithme exact proposé dans ce rapport fournit en pratique une solution optimale en temps raisonnable.

Le programme ayant permis d'effectuer ces tests et l'ensemble des jeux de données de la Partie 7 sont mis à disposition librement avec *notre projet* [13]. Cette démarche est primordiale pour garantir la reproductibilité et l'utilisation de nos résultats pour d'autres expériences.

8.2 Perspectives

Les résultats de nos travaux motivent la poursuite de l'étude et la recherche sur ce sujet. Il serait notamment utile d'avoir un retour de chercheurs en biologie sur l'utilisation du projet et sur la façon d'intégrer d'autres données biologiques pour améliorer les résultats des algorithmes dans leurs utilisations dans le domaine de la cartographie génétique.

Nos tests montrent que l'optimisation proposée pour l'algorithme exact permet de réduire le temps de calcul de façon significative. Cette observation invite à poursuivre la recherche de propriétés permettant de réduire ou de diviser l'ensemble des solutions à explorer.

9 Annexes

9.1 Algorithmes

9.1.1 Calculer si un graphe est linéarisable

Algorithme 4 Calcule si un graphe noir et vert est linéarisable

Données : G un multigraphe P-noir G_k-vert
Résultat : $Vrai$ si G est linéarisable, $Faux$ sinon.

1: **si** (G contient un sommet x tel que $deg_v(x) > 2$
   ou G contient un cycle vert
   ou G contient un mauvais cycle) **alors**
2: renvoyer $Faux$;
3: **sinon**
4: renvoyer $Vrai$;
5: **fin si**

9.1.2 Existence d'un cycle vert

On utilise l'algorithme classique de détection d'un cycle, en n'effectuant le parcours que sur les arêtes vertes.

Algorithme 5 Existence d'un cycle vert

Données : G un multigraphe P-noir G_k-vert,
Fonction $niveau : V(G) \to \mathbb{N}$
f : File vide d'éléments de $V(G)$
Résultat : $Vrai$ si G contient un cycle vert, $Faux$ sinon.

pour $v \in V(G)$ **faire**
  $niveau(v) \leftarrow NULL$
fin pour
tant que $\exists v$ tel que $niveau(v) = NULL$ **faire**
  Choisir un tel v
  $niveau(v) \leftarrow 0$
  f.Enfiler(v)
  tant que f n'est pas vide **faire**
    $u \leftarrow f$.Defiler()
    pour $x \in N_G^v(u)$ **faire**
      si $niveau(x) = NULL$ **alors**
        $niveau(x) \leftarrow niveau(u) + 1$
        f.Enfiler(x)
      sinon
        si $niveau(x) \geq niveau(u)$ **alors**
          renvoyer $Vrai$
        fin si
      fin si
    fin pour
  fin tant que
fin tant que
renvoyer $Faux$

9.1.3 Existence d'un mauvais cycle

Algorithme 6 Existence d'un mauvais cycle

Données : G un multigraphe P-noir G_k-vert
Résultat : $Vrai$ si G contient un mauvais cycle, $Faux$ sinon.

```
1: pour x ∈ V(P) faire
2:    A ← Ensemble des sommets accessibles depuis x, le long des arêtes vertes.
3:    pour u ∈ V(P) ∩ A faire
4:       si ux ∉ E(P) alors
5:          renvoyer Vrai
6:       fin si
7:    fin pour
8: fin pour
9: renvoyer Faux
```

9.1.4 Énumération linéaire

Nous considérons maintenant en donnée un graphe G linéarisable. Il ne possède donc ni cycle vert, ni mauvais cycle et n'a que des sommets de degré vert au plus 2.

Algorithme 7 Calcule une enumération linéaire d'un graphe linéarisable

Données : G un multigraphe P-noir G_k-vert linéarisable
Résultat : une énumération linéaire du graphe

1: $enumeration \leftarrow [\,]$
2: $dejaVu : V(G) \rightarrow Bool$
3: **pour** $x \in V(G)$ **faire**
4: $\quad dejaVu(x) \leftarrow (x \in V(P))$
5: **fin pour**
6: Notons $P = x_1...x_l$
7: **pour** i de 1 à l **faire**
8: $\quad gauche \leftarrow (\exists v$ voisin de x_i non déjà vu).
9: $\quad$ **pour** tout voisin de x_i non déjà vu **faire**
10: $\quad\quad inter \leftarrow [\,]$
11: $\quad\quad dejaVu(u) \leftarrow Vrai$
12: $\quad\quad$ **tant que** u a un voisin v non deja vu **faire**
13: $\quad\quad\quad inter$.ajout(v)
14: $\quad\quad\quad u \leftarrow v$
15: $\quad\quad\quad dejaVu(u) \leftarrow Vrai$
16: $\quad\quad$ **fin tant que**
17: $\quad\quad$ **si** $deg(u) = 1$ et non $gauche$ **alors**
18: $\quad\quad\quad inter$.inverser() ;
19: $\quad\quad\quad enumeration \leftarrow$ Concatenation($enumeration$,$inter$,$[x_i]$)
20: $\quad\quad\quad gauche \leftarrow Vrai$
21: $\quad\quad$ **sinon**
22: $\quad\quad\quad enumeration \leftarrow$ Concatenation($enumeration$,$inter$)
23: $\quad\quad$ **fin si**
24: $\quad$ **fin pour**
25: **fin pour**
26: **tant que** $\exists u \in V(G)$ non déjà vu **faire**
27: $\quad$ Choisir $u \in V(G)$ non déjà vu de degré 1. //Existe car G est linéarisable
28: $\quad enumeration$.ajout(u)
29: $\quad$ **tant que** u a un voisin v non deja vu **faire**
30: $\quad\quad u \leftarrow v$
31: $\quad\quad enumeration$.ajout(u)
32: $\quad$ **fin tant que**
33: **fin tant que**
34: retourner $enumeration$

Références

[1] Y. Anselmetti, V. Berry, C. Chauve, A. Chateau, E. Tannier et S. Bérard, "Ancestral gene synteny reconstruction improves extant species scaffolding," oct. 2015. [En ligne]. Disponible : https://bmcgenomics.biomedcentral.com/articles/10.1186/1471-2164-16-S10-S11

[2] C. Paul, "Complexité et algorithmes paramétrés," dans *Ecole Jeunes Chercheurs en Informatique Mathématique*, Perpignan, France, 2013. [En ligne]. Disponible : https://hal.archives-ouvertes.fr/hal-01178227

[3] L. De Mattéo, "Étude d'un problème de graphe concernant la compatibilité de données génomiques," Université de Montpellier, France, 2016.

[4] P.-L. Giscard, N. Kriege et R. C. Wilson, "A general purpose algorithm for counting simple cycles and simple paths of any length," *Algorithmica*, vol. 81, n°. 7, p. 2716–2737, Jul 2019. [En ligne]. Disponible : https://doi.org/10.1007/s00453-019-00552-1

[5] P.-L. Giscard et P. Rochet, "Enumerating simple paths from connected induced subgraphs," *Graphs and Combinatorics*, vol. 34, n°. 6, p. 1197–1202, Nov 2018. [En ligne]. Disponible : https://doi.org/10.1007/s00373-018-1966-9

[6] F. Le Gall, "Powers of Tensors and Fast Matrix Multiplication," *arXiv e-prints*, p. arXiv :1401.7714, Jan 2014.

[7] D. Avis et K. Fukuda, "Reverse search for enumeration," *Discrete Applied Mathematics*, vol. 65, n°. 1, p. 21 – 46, 1996, first International Colloquium on Graphs and Optimization.

[8] K. M. Elbassioni, "A polynomial delay algorithm for generating connected induced subgraphs of a given cardinality," *CoRR*, vol. abs/1411.2262, 2014. [En ligne]. Disponible : http://arxiv.org/abs/1411.2262

[9] R. Uehara, "The number of connected components in graphs and its applications," 06 1999.

[10] Cyclecount sur matlab. [En ligne]. Disponible : https://www.mathworks.com/matlabcentral/fileexchange/60814

[11] M. Milani, "A python package to measure balance based on count of simple cycles in a network." [En ligne]. Disponible : https://github.com/milani/cycleindex

[12] A. Himeur et L. Picasarri-Arrieta, "Cyclecount." [En ligne]. Disponible : https://github.com/Hareski/CycleCount

[13] ——, “Projet algorithmique pour le consensus d’ordres,” mai 2020. [En ligne]. Disponible : https://github.com/Hareski/ConsensusOrdres

[14] S. Bessy, “Notes de cours : Théorie et algorithmique de graphes,” 2020, Université de Montpellier.

[15] EuYu, “Eigenvalues and power of a matrix,” Mathematics Stack Exchange. [En ligne]. Disponible : https://math.stackexchange.com/q/241793