Master Informatique Théorique

Rapport de Stage
Algorithmique pour le consensus d'ordres

Auteur
Picasarri-Arrieta Lucas

Encadrants
Bérard Sèverine
Grenet Bruno

Résumé

Ce rapport présente les travaux que j'ai menés au cours d'un stage à l'Institut des Sciences de l'Évolution de Montpellier, co-encadré par Sèverine Bérard et Bruno Grenet. Ce stage fait suite à un projet que nous avons réalisé avec Areski Himeur dans le cadre de l'unité d'enseignement HMIN201 du Master d'Informatique Théorique de l'Université de Montpellier. Ce projet faisait quant à lui suite aux travaux de Lisa De Mattéo, réalisés lors d'un stage. Ainsi, ce rapport présente les points principaux de ce qui a été réalisé précédemment, et l'ensemble des nouvelles pistes ayant été abordées dans ce stage.

Ce rapport décrit ainsi plusieurs méthodes de résolution d'un problème de la théorie des graphes ayant des applications dans le contexte de la bioinformatique. Ce problème étant $\mathcal{NP}$-Difficile, nous présentons quelques algorithmes qui s'exécutent en temps polynomial, et qui apportent une solution approchée au problème. Nous reprenons également l'algorithme exact issu des travaux précédents, que nous optimiserons afin de le rendre utilisable sur certains cas pratiques. Enfin, nous présentons une modélisation du problème vis-à-vis de la programmation linéaire. Toutes ces méthodes de résolution proposées sont finalement comparées les unes aux autres dans une dernière partie.

Remerciements

Je tiens à remercier Mme Sèverine Bérard et M. Bruno Grenet, sans qui ce stage n'aurait pas eu lieu. Je les remercie également pour leur patience, leurs conseils et le temps qu'ils ont accordé à ce projet. Je tiens également à remercier Mme Lisa De Mattéo qui est à l'origine de ces travaux et M. Areski Himeur pour son aimable et efficace collaboration.

Table des matières

1 Graphes noirs et verts

Cette section présente une définition formelle des graphes noirs et verts et présente quelques premiers résultats.

1.1 Définitions et Notations

Définition 1.1 (chaîne). Dans un graphe non orienté, une chaîne reliant x à y est définie par une suite finie d'arêtes consécutives, reliant x à y.

Définition 1.2 (Graphe noir et vert). Un graphe noir et vert $G = (P, H_v)$ correspond à l'union d'une chaîne P et d'un graphe quelconque H_v. P correspond au graphe noir de G, H_v au graphe vert de G.
Dans ce contexte :

- Une arête est soit noire, soit verte.
- Un sommet est noir (respectivement vert) s'il est l'extrémité d'une arête noire (respectivement verte). Il peut donc être noir et vert.

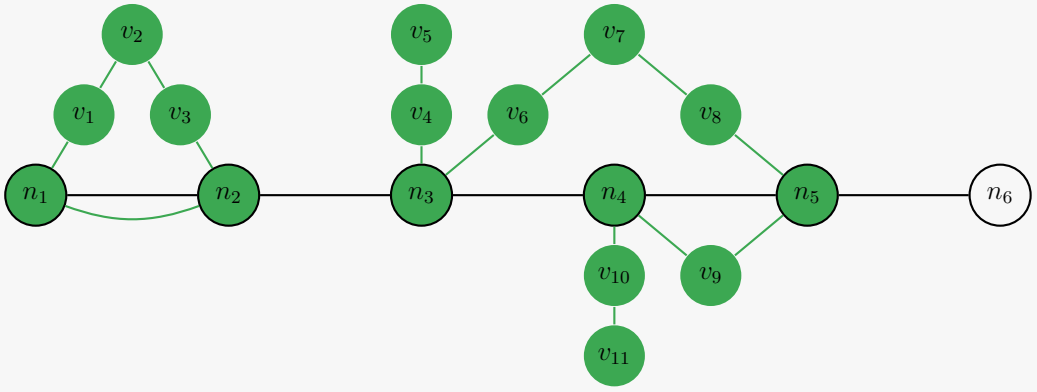

FIGURE 1 – Exemple d'un graphe noir et vert

Exemple. Sur la figure 1, n_6 est un sommet noir, v_1 est un sommet vert et n_1 est un sommet noir et vert.

Notation 1.1. Dans un graphe noir et vert $G = (P, H_v)$ et pour un sommet

$x \in V(G)$, on note :
- n_v (respectivement n_b) le nombre de sommets verts (respectivement noirs).
- m_v (respectivement m_b) le nombre d'arêtes vertes (respectivement noires).
- $N_G^n(x)$ (respectivement $N_G^v(x)$) l'ensemble des sommets $v \in V(G)$ tels que xv soit une arête noire (respectivement verte).
- $\deg_v(x) = |N_G^v(x)|$ et $\deg_n(x) = |N_G^n(x)|$ les degrés vert et noir du sommet x.

Définition 1.3 (Mauvais cycle). Un cycle C d'un graphe noir et vert G 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.

Exemple. Sur la figure 1, le graphe noir et vert admet un seul mauvais cycle : $n_3, v_6, v_7, v_8, n_5, n_4$.

Définition 1.4 (Énumération linéaire). Soit $G = (P, H_v)$ un graphe noir et vert. Une énumération linéaire $(x_1, ..., x_n)$ des sommets de G est une énumération de $V(G)$ telle que :
- pour toute arête verte uv de G, u et v sont adjacents dans l'énumération ;
- pour toute arête noire uv de G, il n'existe pas de sommet noir entre u et v dans l'énumération.

Définition 1.5 (Linéarisable). Soit G un graphe noir et vert. G est dit linéarisable s'il admet une énumération linéaire.

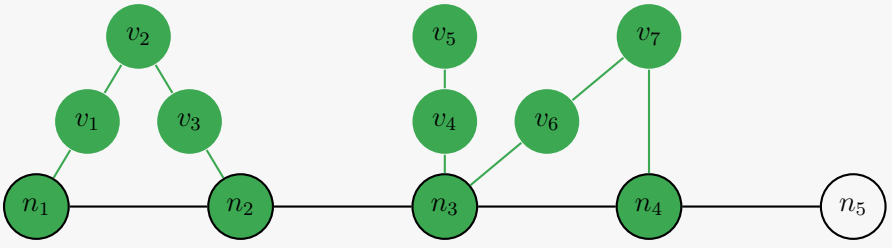

FIGURE 2 – Exemple d'un graphe P-noir G_k-vert linéarisable

Exemple. Sur la figure 1, le graphe noir et vert n'est pas linéarisable. En revanche, le graphe noir et vert de la figure 2 est linéarisable et admet l'énumération linéaire suivante : $n_1, v_1, v_2, v_3, n_2, v_5, v_4, n_3, v_6, v_7, n_4, n_5$.

1.2 Caractérisation des graphes linéarisables

L. De Mattéo [1] a apporté une caractérisation des graphes noirs et verts linéarisables dans ses travaux :

Théorème 1 (De Mattéo [1]). Soit $G = (P, H_v)$ un graphe noir et vert. G est linéarisable si et seulement si H_v est une union de chemins sommet-disjoints, et si G ne contient aucun mauvais cycle.

Corollaire 1 (De Mattéo [1]). Soit $G = (P, H_v)$ un graphe noir et vert. G est linéarisable si et seulement si les trois propriétés suivantes sont vérifiées :
- G ne contient pas de mauvais cycle ;
- tout sommet de G a au plus 2 arêtes vertes incidentes ;
- G ne contient pas de cycle vert.

On peut aisément déduire des résultats précédents que savoir si un graphe noir et vert est linéarisable est un problème de la classe $\mathcal{P}$.

L. De Mattéo a également écrit un algorithme polynomial donnant l'énumération linéaire d'un graphe noir et vert linéarisable, ou qui détecte une erreur si le graphe en question n'est pas linéarisable. Plus précisément, cet algorithme est en complexité $\mathcal{O}(n + m)$.

Ainsi, calculer une énumération linéaire d'un graphe noir et vert linéarisable est également un problème de la classe $\mathcal{P}$. La suite de ce rapport traite de graphes noirs et verts à priori non linéarisables.

2 Le problème Min-Cassure

L'ensemble de ce qui a été réalisé durant ce stage porte sur le problème Min-Cassure. Ce problème est présenté dans cette partie.

2.1 Définition et premiers résultats

Entrée : un graphe noir et vert G

Sortie : un ensemble $Cassure$ d'arêtes vertes minimum tel que $G - Cassure$ soit linéarisable.

Théorème 2 (De Mattéo [1]). Le problème Min-Cassure est $\mathcal{NP}$-difficile

La preuve fournie par L. De Mattéo est une réduction du problème du cycle hamiltonien au problème Min-Cassure.

Théorème 3 (Himeur-Picasarri [2]). Soit $G = (P, H_v)$ un graphe noir et vert connexe, et A l'ensemble des arêtes vertes uv de G telles que :
- $\deg(u) \leq 2$,
- $\deg(v) \leq 2$,
- u et v soient des sommets verts.

Alors il existe une solution C au problème Min-Cassure telle que $A \cap C = \emptyset$.

Théorème 4. Soit $G = (P, H_v)$ un graphe noir et vert, et soit F l'ensemble des arêtes vertes uv telles que u et v soient reliés par une chaîne noire de taille supérieure à 2. Si C est un ensemble d'arêtes vertes tel que $G-C$ soit linéarisable, alors $F \subseteq C$.

Démonstration. C'est immédiat, si $G-C$ est linéarisable, alors $G-C$ ne contient aucun mauvais cycle (par le corollaire 1). Or, si $uv \in F$, par définition de F, uv forme un mauvais cycle dont l'unique arête verte est uv. Alors, l'unique moyen de supprimer ce mauvais cycle est de supprimer uv, donc $uv \in C$. □

Dans la suite, on peut admettre que les graphes noirs et verts considérés ne contiennent pas de telles arêtes, quitte à faire un prétraitement naïf qui supprime ces arêtes incohérentes. C'est-à-dire, on peut admettre que les instances du problème Min-Cassure ne contiennent pas d'arête verte reliant deux sommets noirs

non reliés par une arête noire.

On peut également s'intéresser au problème complémentaire du problème Min-Cassure, que l'on nommera Max-Green-Edge :

> **Entrée :** un graphe noir et vert $G = (P, H_v)$
>
> **Sortie :** un ensemble F d'arêtes vertes tel que $E(H_v) \setminus F$ soit une solution au problème Min-Cassure.

La taille d'un tel ensemble F correspond alors au nombre maximum d'arêtes vertes qu'une énumération linéaire peut vérifier (une énumération linéaire "vérifie" une arête verte uv si u et v sont adjacents dans l'énumération linéaire). Cette définition est notamment utilisée dans la partie 3.

2.2 Algorithmes existants

Il existe actuellement 3 algorithmes pour résoudre le problème Min-Cassure. Chacun de ces algorithmes est disponible en annexe, et admet une implémentation en C++.

2.2.1 Un algorithme glouton [2]

Cet algorithme renvoie une solution minimale pour l'inclusion. Il prend donc en entrée un graphe noir et vert $G = (P, H_v)$, et construit un graphe noir et vert G' restreint aux sommets de G et à ses arêtes noires.

Ensuite, chaque arête verte uv de G est considérée. Si G' reste linéarisable en lui ajoutant uv, alors l'arête uv est ajoutée à G'. Sinon, l'arête est ajoutée à l'ensemble $Cassure$. À la fin, G' correspond à G privé des arêtes de $Cassure$, qui est donc bien un ensemble minimal pour l'inclusion.

2.2.2 Une heuristique [1]

Cette heuristique utilise la même stratégie que l'algorithme glouton, mais en donnant un ordre précis sur les arêtes. Le but est de considérer prioritairement les arêtes qui apparaissent dans le moins de mauvaises structures. Les mauvaises

structures correspondent aux cycles verts, aux mauvais cycles et aux sommets incidents à plus de deux arêtes vertes.

Le dénombrement des cycles simples (c'est-à-dire des cycles passant au plus une fois par chaque sommet) d'un graphe étant un problème connu $\#\mathcal{P}$-complet, il existe actuellement deux versions de cette heuristique :

— une version en complexité polynomiale qui dénombre les marches fermées, c'est-à-dire des cycles pouvant passer plusieurs fois par le même sommet ;
— une version en complexité FPT qui dénombre bien les cycles simples, où le paramètre est la taille du plus grand cycle du graphe.

2.2.3 Un algorithme exact [2]

Cet algorithme exact est en complexité exponentielle, mais a été optimisé par les résultats du théorème 3, ce qui le rend utilisable sur certains exemples en pratique.

3 Approximation du problème Max-Green-Edge

Théorème 5. Le problème Max-Green-Edge appartient à la classe $\mathcal{APX}$ et son ratio d'approximation est au plus égal à 2.

Démonstration. La preuve du théorème précédente consiste à exhiber un algorithme polynomial qui calcule une 2-approximation du problème Max-Green-Edge, et à montrer sa correction. On considère alors le simple algorithme consistant à renvoyer un couplage maximum sur $E(H_v)$.

1. *L'algorithme renvoie une solution correcte*
   Soit M la solution obtenue. Il faut prouver que G réduit aux arêtes noires et aux arêtes vertes de M est linéarisable. Notons ce graphe G'.
   M étant un couplage, c'est à fortiori une union de chemins sommet-disjoints. Par le théorème 1, il reste à montrer que G' ne contient aucun mauvais cycle. Mais M étant un couplage, si le graphe contient un mauvais cycle, ce cycle contient une seule arête verte (sinon deux arêtes de M auraient une extrémité en commun), ce qui contredit les hypothèses précédentes (grâce au prétraitement, cf. 2.1).
2. *La solution est au pire 2 fois moins bonne qu'une solution optimale*
   Soit M la solution obtenue, et M^* une solution optimale. Alors on a $|M| \geq \frac{1}{2}|M^*|$, montrons-le par l'absurde :
   Si $|M| < \frac{1}{2}|M^*|$, on sait que M^* forme une union de chemins sommet-disjoints. En considérant une arête sur deux le long de ces chemins, on obtient un couplage M' de taille $\geq \frac{1}{2}M^*$.
   Finalement on a $|M| < \frac{1}{2}|M^*| \leq |M'|$.
   Ce qui contredit le fait que M soit maximum.
3. *L'algorithme s'exécute en temps polynomial*
   On rappelle en effet que le calcul de M s'effectue en temps polynomial en la taille de H_v, par l'algorithme d'Edmonds par exemple.

Finalement, on a bien exhibé un algorithme qui calcule une 2-approximation en temps polynomial du problème Max-Green-Edge. Ce problème est donc bien dans la classe $\mathcal{APX}$, et son ratio d'approximation est bien au plus égal à 2. □

Remarque. On peut bien sûr améliorer l'algorithme en calculant une solution maximale pour l'inclusion, mais ce n'est pas nécessaire pour la preuve.

4 Un algorithme probabiliste

4.1 Retour sur l'algorithme glouton

Il est important pour la suite de revenir sur un point de l'algorithme glouton.

Théorème 6. Il existe un ordre des arêtes vertes pour lequel l'algorithme glouton donne la solution optimale.

Démonstration. Le résultat est assez immédiat. Soit $G = (P, H_v)$ un graphe noir et vert et C^* une solution optimale pour le problème Min-Cassure sur G. Posons $F = E(H_v) \setminus C^*$ l'ensemble des arêtes vertes qui ne sont pas dans C^*.

Considérons alors un ordre des arêtes où toutes les arêtes de F sont avant celles de C^*. Sur cet ordre, l'algorithme glouton va conserver chaque arête de F, car G restreint à F est linéarisable. L'algorithme va ensuite renvoyer exactement C^*. □

4.2 L'algorithme probabiliste

Le but de l'algorithme probabiliste est alors de trouver différentes solutions minimales pour l'inclusion. Parmi les solutions rencontrées, on en renvoie une de cardinalité minimum.

Pour cela, on maintient tout le long de l'algorithme une énumération des sommets *enum* qui vérifie tout le temps la condition sur les arêtes noires. C'est-à-dire, si uv est une arête noire, il n'y a pas de sommet noir entre u et v dans *enum*.

Puis à chaque étape, on choisit aléatoirement une arête verte qui n'est pas vérifiée par *enum*. C'est-à-dire une arête verte uv telle que u et v ne soient pas successifs dans *enum*. On modifie ensuite *enum* afin que cette arête soit vérifiée. Cette action ne doit pas casser la condition sur les arêtes noires, mais peut casser certaines arêtes vertes.

On obtient ainsi un ensemble $Cassure$, composé de l'ensemble des arêtes vertes uv telles que u et v ne soient pas successifs dans *enum*. On peut rendre minimal cet ensemble, et obtenir une nouvelle valeur pour *enum*, et répéter l'opération. À la fin, il suffit de renvoyer l'ensemble $Cassure$ de cardinalité minimum rencontré.

Voici l'algorithme :

Algorithme 1 Min-Cassure : algorithme probabiliste

Données : $G = (P, H_v)$ un graphe noir et vert prétraité
Résultat : Une solution (à priori non optimale) pour Min-Cassure

```
1: enum ← énumération aléatoire des sommets cohérente vis-à-vis des arêtes noires ;
2: Cassure* ← Ensemble des arêtes non couvertes par enum.
3: pour i allant de 1 à N faire
4:    Choisir aléatoirement une arête verte uv non vérifiée par enum
5:    si une telle arête n'existe pas alors
6:       renvoyer Cassure*
7:    fin si
8:    Déplacer u à côté de v dans enum, en restant cohérent vis-à-vis des arêtes noires.
9:    Cassure ← Ensemble des arêtes non couvertes par enum.
10:   pour toute arête verte uv de Cassure faire
11:      si G \ Cassure ∪ {uv} est linéarisable alors
12:         Cassure ← Cassure \ {uv}
13:      fin si
14:   fin pour
15:   si |Cassure| < |Cassure*| alors
16:      Cassure* ← Cassure
17:   fin si
18: fin pour
19: renvoyer Cassure*
```

Il faut également fixer la valeur de N. En augmentant cette valeur, on augmente la probabilité d'obtenir une solution proche de la solution optimale. En pratique, on fixe N au nombre d'arêtes vertes du graphe.

Puisque l'algorithme glouton s'exécute en temps $\mathcal{O}(nm)$, l'algorithme probabiliste s'exécute en temps $\mathcal{O}(nm^2)$ si on fixe N au nombre d'arêtes vertes du graphe.

5 Une optimisation

On présente ici une propriété sur le problème Min-Cassure qui permet d'améliorer la complexité de certains algorithmes.

5.1 L'optimisation

Théorème 7. Soit $G = (P, H_v)$ un graphe noir et vert connexe, et soit uv une arête de G telle que :
- $\deg_v(u) \leq 2$
- $\deg_v(v) \leq 2$
- uv n'appartienne à aucun cycle de taille ≥ 3.

$G \setminus \{uv\}$ se divise alors en 2 composantes connexes G_1 et G_2. Soient C_1 et C_2 des solutions optimales pour le problème Min-Cassure sur les instances G_1 et G_2. Alors, $C_1 \cup C_2$ forme une solution optimale pour le problème Min-Cassure sur G.

Démonstration. On montre le résultat en montrant que $C_1 \cup C_2$ forme une solution pour Min-Cassure sur G, puis que cette solution est optimale.

1. *$C_1 \cup C_2$ forme une solution au problème Min-Cassure sur G*
   L'arête uv n'appartient à aucune mauvaise structure de G : que ce soit un mauvais cycle, un cycle vert ou un sommet de degré vert ≥ 3. De plus, uv n'appartenant à aucun cycle, chaque mauvaise structure de G appartient soit à G_1, soit à G_2. Elle sera donc éliminée soit par les arêtes de C_1, soit par celles de C_2.
2. *Cette solution est optimale.*
   On montre le résultat par l'absurde : soit C^* une solution optimale pour le problème Min-Cassure sur G telle que $|C^*| < |C_1 \cup C_2|$.
   On sait que l'arête uv (telle que décrite dans l'énoncé) n'appartient pas à C^*, car elle ne permet d'éliminer aucune mauvaise structure. Notons C_1^* (respectivement C_2^*) les arêtes de C^* appartenant à G_1 (respectivement à G_2).
   Puisque C_1 et C_2 n'ont aucune arête en commun, et que $|C^*| < |C_1 \cup C_2|$, alors $|C_1^*| < |C_1|$ ou $|C_2^*| < |C_2|$. Sans perte de généralité, fixons $|C_1^*| < |C_1|$.
   Mais alors C_1^* forme une solution au problème Min-Cassure sur G_1 car aucune mauvaise structure de G_1 ne peut être supprimée par une arête de

C_2^* (sinon, uv appartiendrait à un cycle). Puisque $|C_1^*| < |C_1|$, on contredit le choix de C_1.

□

En utilisant le théorème 7, on peut diviser un graphe noir et vert connexe G en 2 composantes connexes telles que l'union des solutions optimales sur chacune des composantes connexes soit une solution optimale pour G. Mais on peut bien sûr supprimer de G toutes les arêtes ayant les propriétés décrites dans le théorème. On obtient ainsi un plus grand nombre de composantes connexes.

5.2 Application sur l'algorithme exact

L'algorithme exact a une complexité exponentielle en m_v, où m_v est le nombre d'arêtes du graphe noir et vert en entrée : $\mathcal{O}(2^{m_v}(m+n))$

En choisissant d'appliquer l'algorithme exact sur chacune des composantes connexes obtenues grâce aux résultats du théorème 7, on peut borner le nombre d'opérations par $\sum_{i=1}^{k} 2^{m_{v,i}}(n_i + m_i)$ où k,$m_{v,i}$, n_i et m_i représentent respectivement le nombre de composantes connexes obtenues, le nombre d'arêtes vertes, le nombre de sommets et le nombre d'arêtes de la i-ème composante connexe. On obtient ainsi une meilleure complexité en temps dans certains cas.

Illustrons cette optimisation par un exemple :

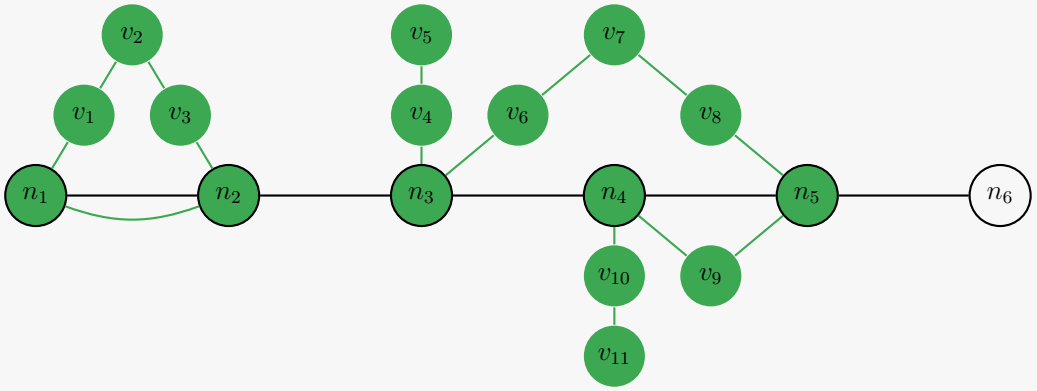

FIGURE 3 – Exemple d'un graphe noir et vert

Exemple. Si on applique l'algorithme exact sans optimisation sur cet exemple, le nombre d'opérations effectuées sera de l'ordre de 2^{15}, car le graphe contient 15 arêtes vertes.

En utilisant l'optimisation, on va traiter séparément 7 composantes connexes (dont 5 sont des sommets isolés) :

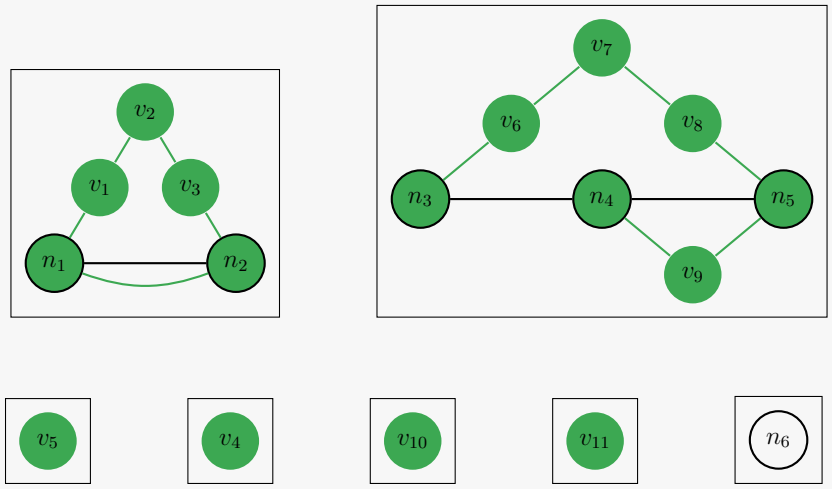

FIGURE 4 – Les sept composantes connexes obtenues en appliquant l'optimisation

Avec l'optimisation, le nombre d'opérations effectuées sur cet exemple sera de l'ordre de $2^5 + 2^6 + 5 \times 2^0$, soit moins de 2^7.

De la même manière, l'algorithme probabiliste a une complexité en $\mathcal{O}(nm_v^2)$ (en fixant N à m_v). On peut donc également améliorer sa complexité en traitant séparément les différentes composantes connexes obtenues. On peut ainsi borner le nombre d'opérations par $\mathcal{O}(\sum_{i=1}^{k} n_i m_i^2)$ où k est le nombre de composantes connexes obtenues, et m_i (respectivement n_i) est le nombre d'arêtes vertes (respectivement de sommets) de la i-ème composante connexe.

Remarque. L'algorithme exact présentait déjà une optimisation[2] issue du théorème 3. Les deux optimisations peuvent bien sûr être combinées pour de meilleurs temps de calcul.

6 Modélisation en programmation linéaire

On présente ici une modélisation du problème Min-Cassure comme un problème d'optimisation linéaire en nombres entiers.

À partir d'un graphe noir et vert $G = (P, H_v)$, on associe les variables suivantes :

- pour tout sommet $v_i \in V(G)$, on associe une variable continue x_i. Cette variable doit induire une relation d'ordre sur les sommets de G qui correspond à l'ordre dans une énumération linéaire. C'est-à-dire, $x_i < x_j$ si et seulement si v_i est avant x_j dans l'énumération linéaire trouvée.
- pour toute paire de sommets $\{v_i, v_j\}$ (avec $i < j$), on associe une variable binaire z_{ij} qui permettra, avec les contraintes, de spécifier que $|x_i - x_j| \geq 1$.
- pour toute arête verte $v_iv_j \in E(H_v)$ (avec $i < j$), une variable binaire y_{ij} qui vaut 1 si l'arête verte v_iv_j est supprimée, 0 sinon.

On associe ensuite les contraintes suivantes :

- pour tout $i \in \{1...n\}$,
$$1 \leq x_i \leq n$$
- pour toute paire $\{i, j\}$ (avec $i < j$), on associe les deux contraintes suivantes :
$$x_i - x_j + nz_{ij} \geq 1$$
$$x_j - x_i - nz_{ij} \geq 1 - n$$

  Ces deux contraintes permettent bien de spécifier $|x_i - x_j| \geq 1$. En effet, si $z_{ij} = 1$, alors la première contrainte est toujours vérifiée et la seconde impose $x_j - x_i \geq 1$. Sinon, $z_{ij} = 0$, et dans ce cas, la seconde contrainte est toujours vérifiée et la première impose $x_i - x_j \geq 1$.
- pour toute arête verte $v_iv_j \in E(H_v)$ (avec $i < j$), on associe les deux contraintes suivantes :
$$x_i - x_j - ny_{ij} \leq 1$$
$$x_j - x_i - ny_{ij} \leq 1$$

  Ces deux contraintes permettent de spécifier que si l'arête verte v_iv_j n'est pas supprimée, alors v_i et v_j sont adjacents dans l'énumération.
  En effet, si v_iv_j n'est pas supprimée, alors $y_{ij} = 0$ et on obtient $|x_i - x_j| \leq 1$ par ces deux contraintes. Avec les contraintes précédentes, on obtient ainsi $|x_i - x_j| = 1$.

— pour toute arête noire $v_i v_j$ le long de P, on associe la contrainte

$$x_i < x_j$$

afin de respecter les contraintes sur le graphe noir.

Enfin, le but est de minimiser

$$\sum y_{ij}$$

afin de minimiser le nombre d'arêtes vertes supprimées. Au final, on peut noter que le nombre de variables et de contraintes créées est en $\mathcal{O}(n^2)$.

Dans la partie 7, les tests liés à la programmation linéaire sont effectués avec le solveur GLPK.

7 Performances

Nous allons dans cette partie présenter les résultats de différents tests de performance réalisés avec les méthodes de résolution présentées précédemment.

7.1 Jeux de données

Les méthodes seront comparées sur plusieurs jeux de données.

Dans un premier temps avec plusieurs graphes de densité assez faible, extraits de données génétiques de plusieurs espèces, et contenant de 17 à 210 sommets :

- G_1 *:* un graphe possédant 17 sommets et 15 arêtes vertes ;
- G_2 *:* un graphe possédant 22 sommets et 26 arêtes vertes ;
- G_3 *:* un graphe possédant 34 sommets et 28 arêtes vertes ;
- G_4 *:* un graphe possédant 28 sommets et 29 arêtes vertes ;
- G_5 *:* un graphe possédant 34 sommets et 32 arêtes vertes ;
- G_6 *:* un graphe possédant 34 sommets et 33 arêtes vertes ;
- G_7 *:* un graphe possédant 210 sommets et 191 arêtes vertes.

Le graphe complet à 8 sommets (K_8). Deux sommets sont reliés par une arête noire, et chaque paire de sommets forme une arête verte. Le but de ce graphe, non issu de données génétiques, et de tester les limites des différentes méthodes de résolution sur des graphes avec une densité plus élevée.

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.

7.2 Résultats des tests

Tous les tests ont été réalisés dans les mêmes conditions. Ils visent à évaluer les différentes méthodes les unes par rapport aux autres. Une absence de données signifie que le programme n'a pas terminé après 2h de calcul.

	G_1		G_2		G_3	
Méthode	Solution	Durée	Solution	Durée	Solution	Durée
Glouton	**2**	1ms	**10**	2ms	**3**	4ms
Heuristique : marches fermées	**2**	30ms	**9**	1s	**3**	60ms
Heuristique : cycles < 10	**2**	200ms	**9**	30s	**3**	1s
Exact sans optimisation	**2**	1ms	**9**	18s	**3**	33ms
Exact avec optimisation	**2**	3ms	**9**	18s	**3**	8ms
Probabiliste	**2**	3ms	**9**	12ms	**3**	9ms
Optimisation Linéaire	**2**	2s	×	×	**3**	127s

Tableau 1 – Résultats des tests sur les instances G_1, G_2 et G_3

	G_4		G_5		G_6	
Méthode	Solution	Durée	Solution	Durée	Solution	Durée
Glouton	**10**	2ms	**8**	2ms	**8**	2ms
Heuristique : marches fermées	**10**	394ms	**8**	185ms	**9**	246ms
Heuristique : cycles < 10	**10**	287s	**8**	73s	**8**	103s
Exact sans optimisation	**10**	5s	**7**	21s	**8**	35s
Exact avec optimisation	**10**	6s	**7**	20s	**8**	33s
Probabiliste	**10**	12ms	**7**	27ms	**8**	16ms
Optimisation Linéaire	×	×	×	×	×	×

Tableau 2 – Résultats des tests sur les instances G_4, G_5 et G_6

	G_7		K_8	
Méthode	Solution	Durée	Solution	Durée
Glouton	**9**	99ms	**21**	1ms
Heuristique : marches fermées	**9**	9s	**21**	156ms
Heuristique : cycles < 10	×	×	**21**	1s
Exact sans optimisation	**9**	**44min**	×	×
Exact avec optimisation	**9**	171ms	×	×
Probabiliste	**9**	346ms	**21**	8ms
Optimisation Linéaire	×	×	**21**	**203s**

Tableau 3 – Résultats des tests sur G_7 et K_8

	Extrait du moustique		Extrait du canard	
Méthode	Solution	Durée	Solution	Durée
Glouton	**6**	202ms	**90**	4s
Heuristique : marches fermées	**6**	44s	**87**	57s
Heuristique : cycles < 10	**6**	**1h30**	×	×
Exact sans optimisation	**6**	**38min**	×	×
Exact avec optimisation	**6**	167ms	**87**	47s
Probabiliste	**6**	643ms	**87**	68s
Optimisation Linéaire	×	×	×	×

Tableau 4 – Résultats des tests sur les instances extraites des données du moustique et du canard

8 Conclusion et interprétation des résultats

Au vu des tests effectués, nous pouvons tirer quelques conclusions. Il conviendrait néanmoins de confirmer ces conclusions sur d'autres jeux de données issus de données génétiques.

8.0.1 L'algorithme glouton et l'heuristique

L'algorithme glouton et l'heuristique basée sur les marches fermées sont assez semblables en pratique : ils apportent une solution rapide au problème, mais présentent parfois de légères erreurs. Ils semblent être de bons algorithmes d'approximation.

L'heuristique basée sur les cycles simples de taille inférieure à 10 présente également assez peu d'erreurs (une seule erreur sur G_5). Néanmoins, elle semble assez peu utilisable sur de grands jeux de données au vu des temps d'exécution sur G_7, et sur les données issues du moustique et du canard.

8.1 L'algorithme exact

L'impact de l'optimisation de l'algorithme exact est assez flagrant sur les grands jeux de données. Il invite à penser que cette optimisation rend l'algorithme exact utilisable sur les cas pratiques, où les graphes sont issus de données génétiques.

Néanmoins, l'exemple de K_8 montre que l'algorithme exact reste en complexité exponentielle, et qu'il ne sera pas, à priori, toujours exploitable.

8.2 L'algorithme probabiliste

L'algorithme probabiliste présente de bons résultats. Les temps d'exécution sont relativement faibles : ils sont semblables à ceux de l'heuristique où l'on considère uniquement les marches fermées. De plus, il ne présente aucune erreur sur l'ensemble des tests. Ces résultats invitent à penser que l'algorithme probabiliste est la meilleure alternative à l'algorithme exact lorsque celui-ci n'est pas exploitable.

8.3 L'optimisation linéaire

L'optimisation linéaire n'offre pas de bons résultats : elle est inutilisable sur la quasi-totalité des exemples. Ces résultats invitent, dans une prochaine étude, à trouver une meilleure modélisation du problème en programmation linéaire, ou à proposer une relaxation des contraintes binaires afin d'obtenir une solution approchée.

8.4 Bilan

Je profite de ce bilan pour inviter quiconque le désire à approfondir les résultats trouvés sur le sujet. Pour ce faire, vous pouvez retrouver l'ensemble des programmes utilisés sur *Github* [3]. 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*.

Les résultats de nos travaux motivent la poursuite de l'étude et la recherche sur ce sujet. Il semble notamment intéressant de pouvoir classifier le problème Min-Cassure dans les classes d'approximation (et pas seulement le problème Max-Green-Edge), et d'affiner la modélisation vis-à-vis de la programmation linéaire.

Finalement, ce stage fut pour moi l'occasion de développer mes capacités de réflexion, mais aussi de synthèse et de présentation de résultats. Il m'a également conforté dans l'idée de m'orienter vers les métiers de la recherche.

9 Annexes

9.1 Algorithmes

Algorithme 2 Min-Cassure : Algorithme glouton

Données : $G = (P, H_v)$ un graphe noir et vert
Résultat : Un ensemble $Cassure$ d'arêtes vertes de G, minimal pour l'inclusion, tel que $G - Cassure$ soit linéarisable.

1: $Cassure \leftarrow \emptyset$
2: $G' \leftarrow P$
3: **pour** toute arête verte uv de G **faire**
4: **si** $G' \cup \{uv\}$ est linéarisable **alors**
5: Ajouter dans G' l'arête uv
6: **sinon**
7: $Cassure \leftarrow Cassure \cup \{uv\}$
8: **fin si**
9: **fin pour**
10: renvoyer $Cassure$;

Algorithme 3 Min-Cassure : Heuristique

Données : $G = (P, H_v)$ un graphe noir et vert

Résultat : Un ensemble $Cassure$ d'arêtes vertes de G, minimal pour l'inclusion, tel que $G - Cassure$ soit linéarisable.

1: Bijection $w : E(G_i) \to \mathbb{N}, \forall i$
2: **pour** $uv \in E(H_v)$ **faire**
3: $w(uv) \leftarrow 0$
4: **si** $\deg_v(u) > 2$ **alors**
5: $w(uv) \leftarrow w(uv) + \deg_v(u) - 2$
6: **fin si**
7: **si** $\deg_v(v) > 2$ **alors**
8: $w(uv) \leftarrow w(uv) + \deg_v(v) - 2$
9: **fin si**
10: /* les stratégies de dénombrement de cycles sont celles décrites dans la partie 2.2.2 */
11: $c_{vert} \leftarrow \#\{$ Cycle vert C, C contenant uv $\}$
12: $w(uv) \leftarrow w(uv) + c_{vert}$
13: $c_{mauvais} \leftarrow \#\{$ Mauvais cycle C, C contenant uv $\}$
14: $w(uv) \leftarrow w(uv) + c_{mauvais}$
15: **fin pour**
16: Appliquer l'algorithme glouton en prenant les arêtes de G par ordre croissant selon w.
17: **renvoyer** $Cassure$

Algorithme 4 Min-Cassure : Algorithme exact

Données : $G = (P, H_v)$ un graphe noir et vert

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

1: $l \leftarrow 0$
2: $Trouve \leftarrow Faux$
3: $A \leftarrow \emptyset$
4: **pour** toute arête verte uv de G **faire**
5: **si** $u \in V(P)$ ou $v \in V(P)$ ou $\deg_v(u) > 2$ ou $\deg_v(v) > 2$ **alors**
6: $A \leftarrow A \cup \{uv\}$
7: **fin si**
8: **fin pour**
9: **tant que** (non $Trouve$) **faire**
10: **pour** tout $F \subseteq A$ de taille l **faire**
11: **si** $G - F$ est linéarisable **alors**
12: $Cassure \leftarrow F$
13: $Trouve \leftarrow Vrai$
14: **fin si**
15: **fin pour**
16: $l \leftarrow l + 1$
17: **fin tant que**
18: renvoyer $Cassure$;

Références

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

[2] A. Himeur et L. Picasarri-Arrieta, "Rapport de projet : Algorithmique pour le consensus d'ordres," 2020, Université de Montpellier.

[3] ——, "Projet algorithmique pour le consensus d'ordres," mai 2020. [En ligne]. Disponible : https://github.com/Hareski/ConsensusOrdres

[4] S. Bessy, "Notes de cours : Théorie et algorithmique de graphes," 2020, Université de Montpellier.

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

[6] 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, nº. 7, p. 2716–2737, Jul 2019. [En ligne]. Disponible : https://doi.org/10.1007/s00453-019-00552-1

[7] 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