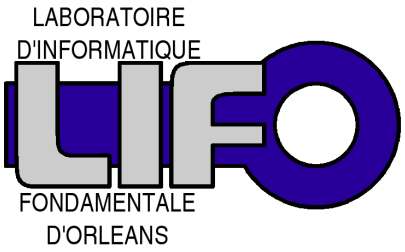

Rapport de stage

Problèmes et variantes de domination romaine dans les graphes d'intervalles

Écrit par Nicolas Schivre

Avril - Juin 2023

Maître de stage :
Mathieu LIEDLOFF

Référent de l'université :
Anaïs HALFTERMEYER

L3 informatique parcours ingénierie

Remerciement

Tout d'abord, je tiens à remercier Mathieu Liedloff, mon maître de stage, sans qui ce stage n'aurait pas eu lieu. L'accompagnement qu'il m'a apporté durant ce stage m'a permis de réussir dans mon travail. Il m'a fait découvrir le monde de la recherche et m'y a fait prendre goût.

Je souhaite également remercier Anaïs Halftermeyer qui m'a également conseillé dans la conception de ce rapport.

Je remercie aussi le personnel de l'équipe GAMoC du LIFO qui m'a acceuilli durant ce stage.

Je tiens aussi à remercier tout mes autres amis en stage au LIFO ou en projet universitaire : Benjamin, Loïc, Finaritra, Maxime, Matteo, Simon, Bastien, Olivia et Marc. Ils ont rendu cette expérience professionnelle unique et inoubliable.

Sommaire

Introduction

Les problèmes algorithmiques appliqués aux graphes sont nombreux et très communs. En effet, ces problèmes peuvent s'appliquer à de nombreux domaines comme les mathématiques, la biologie, l'informatique, etc. Ils peuvent même être utile pour des problèmes du quotidien ! Parmi les problèmes les plus connus on peut par exemple citer le problème du voyageur du commerce qui consiste à chercher un cycle hamiltonien (cycle passant par tous les sommets du graphe) de poids minimum, ou le problème de coloration de graphes (plus de détails dans la section 1.3). Dans ce rapport nous allons nous intéresser au problème de domination ainsi qu'à la domination romaine et ses variantes pour lesquels on se restreint aux graphes d'intervalles.

Dans le **premier chapitre**, nous allons présenter la classe des graphes d'intervalles avec ses diverses propriétés ainsi qu'un état de l'art des différentes sous-classes et super-classes des graphes d'intervalles.
Ensuite, dans un **deuxième chapitre**, on définira le problème de domination et nous proposerons plusieurs algorithmes pour le résoudre.
Enfin, dans un **troisième chapitre**, on définira le problème de domination romaine ainsi que la domination romaine quasi totale et nous proposerons 2 algorithmes pour résoudre cette dernière.

Contexte

Le LIFO est le laboratoire d'informatique fondamentale d'Orléans situé sur le campus de l'université d'Orléans au sein duquel j'ai réalisé mon stage de fin de licence. Le LIFO est composé de différentes équipes :

— l'équipe CA : Contraintes et Apprentissage
— l'équipe LMV : Langages, Modèles et Vérification
— l'équipe Pamda : Parallélisme, calcul distribué, bases de données
— l'équipe SDS : Sécurité des Données et des Systèmes
— l'équipe GAMoC : Graphes, Algorithmes et Modèles de Calcul

C'est au sein de cette dernière équipe que j'ai été accueilli pour effectuer mon stage.

Durant ce stage, mon but était d'étudier différents sous-problèmes de la domination romaine tel que :

— la domination romaine quasi totale (QTRD)
— la domination romaine totale (TRD)
— la domination romaine indépendante (IRD)
— la domination romaine signée (SRD)

L'étude des variantes de la domination romaine a pour but de concevoir des algorithmes polynomiaux (en la taille de l'entrée) résolvant ces problèmes pour la classe des graphes d'intervalles ou alors une preuve de NP-difficulté et de NP-complétude si on ne peut concevoir un tel algorithme. La découverte de tels algorithmes permettrait de faire avancer la recherche sur différentes variantes des problèmes de domination.

Travail réalisé

Afin de pouvoir comprendre efficacement les propriétés des problèmes de domination romaine, je me suis d'abord penché sur la résolution du problème de domination pour les graphes d'intervalles. C'est un problème qui a déjà été résolu par le passé mais ce fut un bon exercice pour l'utilisation de la stratégie gloutonne [20] et de la programmation dynamique [19]. Par la suite, j'ai lu des articles scientifiques pour mieux comprendre les différentes propriétés du problème de domination romaine ainsi que ses variantes. J'ai ensuite développé divers algorithmes ainsi que leurs preuves pour résoudre ces problèmes. Je n'ai pas rencontré, pendant mon stage, la nécessité de faire une preuve de NP-difficulté et NP-complétude. J'ai eu cependant l'occasion de programmer afin d'implémenter mes algorithmes et d'automatiser la recherche de contre-exemples [1].

1. Lien du github ou se trouve le code : https://github.com/Itasuka/roman-domination

Chapitre 1

Graphes et classes de graphes

Dans sa forme la plus commune, un graphe $G = (V, E)$ est un couple composé d'un ensemble de points V appelés sommets et un ensemble d'arêtes E qui définissent des relations entre les sommets de V. Les graphes permettent de modéliser des problèmes qui peuvent s'appliquer à de nombreux domaines tel que l'informatique, les mathématiques ou encore la biologie. Il existe de nombreuses classes de graphes afin de créer des représentations adaptées aux divers problèmes rencontrés. Parmi toutes les classes, on peut par exemple citer les graphes parfaits qui sont liés au problème de coloration (colorer les sommets d'un graphes par une couleur de manière à ce qu'aucun sommets voisin ne possède la même couleur)[2, 18], les graphes cordaux[1], ou encore les graphes d'intervalles.

C'est en considérant des graphes de cette dernière classe que nous allons concevoir nos algorithmes car elle va simplifier la modélisation et la résolution des problèmes de dominations que nous rencontrerons par la suite. Dans un premier temps, nous présentons les propriétés relatives aux graphes d'intervalles ainsi que l'intérêt de ce choix pour la modélisation de nos problèmes à venir. Nous établirons ensuite, afin de mieux appréhender les différentes propriétés des graphes d'intervalles, un état de l'art des différentes sous-classes et super-classes de cette classe de graphes.

Les travaux de cette partie concernant les définissions des classes de graphes sont en grande parti obtenu grâce au site ISCGI [1] répertoriant les différentes classes de graphes ainsi que leurs inclusions.

1.1 Graphes d'intervalles

Considérant un ensemble $\mathcal{I}$ de n intervalles de la droite réelle, un graphe d'intervalles $G_{\mathcal{I}} = (V, E)$ est le graphe d'intersection des intervalles de l'ensemble $\mathcal{I}$ tel que chaque sommets est un intervalle et les sommets sont reliés entre eux si leur intervalles correspondant s'intersectent [2, 18] (Voir Figure 1.1).

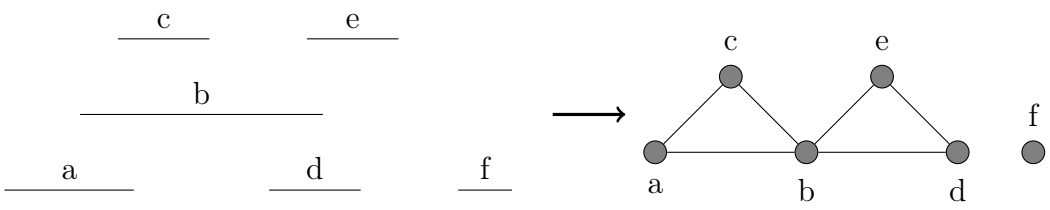

FIGURE 1.1 – Exemple d'un graphe d'intervalles et de son graphe d'intersections

Par la suite nous considérerons un graphe d'intervalles par son ensemble d'intervalles plutôt que par son graphe d'intersection car l'ensemble d'intervalles permet une meilleure visualisation des propriétés des intervalles du graphe. Chaque intervalle est représenté par une date de début et une date de fin ou autrement nommée gauche et droite de l'intervalle. Considérant un intervalle $v \in V$, on note $\mathrm{d}(v)$ et $\mathrm{f}(v)$, ou $\mathrm{l}(v)$ et $\mathrm{r}(v)$, le début et la fin de v ou respectivement la gauche et la droite de v (left and right). La représentation du graphe en un ensemble d'intervalles nous permettra, dans la conception de nos algorithmes, de diversifier nos possibilité d'approches pour un problème. Nous pourrons, par exemple, parcourir les intervalles du graphes par date de fin croissante, par date de début croissante, etc.
Une propriété intéressante des graphes d'intervalles c'est qu'ils ne peuvent contenir de C_4 (cycle de 4 sommets) sans corde [8]. On peut très facilement visualiser cette propriété sur un exemple (Voir Figure 1.2) ci-après.

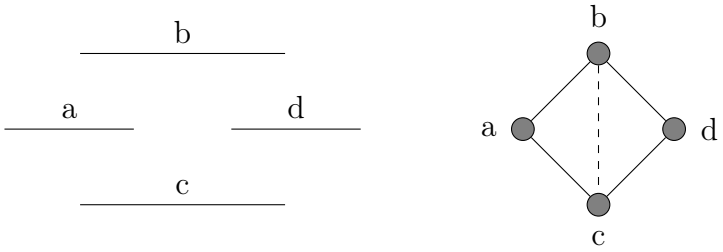

FIGURE 1.2 – Pas de C_4 dans les graphes d'intervalles

1.1.1 Modèle d'intervalles normalisé

L'ensemble de n intervalles d'un graphe d'intervalles peut être représenté sur un axe de 0 à $2n$ par un ensemble de n-uplet de 2 éléments (debut, fin). Chaque nombre de l'axe est assigné à un début ou à une fin d'un intervalle avec comme seule règle que la date de début d'un intervalle se situe avant sa date de fin. Ce modèle d'intervalles nous permet de borner le nombre de possibilités pour créer un graphe d'intervalles. On aura donc, en utilisant ce modèle, plus de facilité à construire des contre-exemples différents pour nos algorithmes.

1.2 Sous-classes des graphes d'intervalles

Une sous-classe des graphes d'intervalles est une classe dont les graphes sont aussi des graphes d'intervalles, elles ont pour super-classe les graphes d'intervalles. Autrement dit, une classe C est une sous-classe des graphes d'intervalles si $C \subseteq$ graphes d'intervalles.

Graphe clique

Une clique est un sous ensemble de sommets d'un graphe tel que son sous-graphe est complet. On note $\omega(G)$ la taille maximale d'une clique d'un graphe $G = (V, E)$. Aussi connu sous le nom de graphe complet, c'est un graphe ou $\omega(G) = |V|$.
Un graphe complet à n sommets est noté K_n.

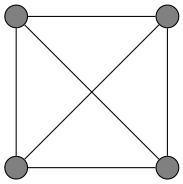

FIGURE 1.3 – Exemple d'un graphe clique

Sur la figure 1.3 on peut remarquer $|V| = 4 = \omega(G)$.

Graphes d'intervalles propre / unitaire

Les graphes d'intervalles propres sont des graphes d'intervalles où aucun intervalle n'est contenu dans un autre.

Les graphes unitaire sont des graphes où tous les intervalles sont de la même taille.

Il a été démontré que ces deux classes de graphes sont équivalentes [4].

Graphes seuil

Ce sont des graphes qui peuvent être construits à partir d'un sommet en répétant 2 opérations :

— ajouter un sommet isolé ;

— ajouter un sommet dominant (relié à tout les autres sommets).

On peut aussi définir ces graphes comme des graphes pour lesquels tous les sommets ont un poids et le graphe possède un seuil tel que il existe une arrête entre deux sommets si le poids de ces deux sommets additionnés dépasse le seuil du graphe.

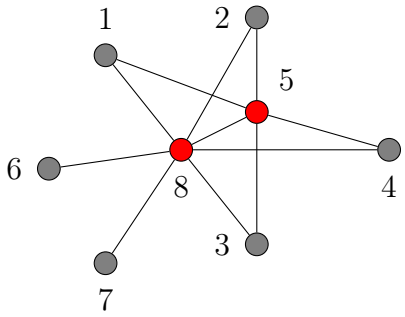

FIGURE 1.4 – Exemple d'un graphe seuil

Les sommets de la figure 1.4 sont numéroté selon l'ordre de création par les opération. Les sommets gris sont les sommets isolés et les sommets rouge les dominant.

Graphes d'intervalles *probe*

Les Graphes d'intervalles probe, dont le nom vient de l'anglais *probe* signifiant *sonder*, sont des graphes dont on peut séparer les sommets en 2 catégories (probe P et non-probe N) de manière à ce que N soit indépendant et que les nouvelles arrêtes peuvent être rajoutées entre les intervalles de N afin de faire un graphe d'intervalles [9].

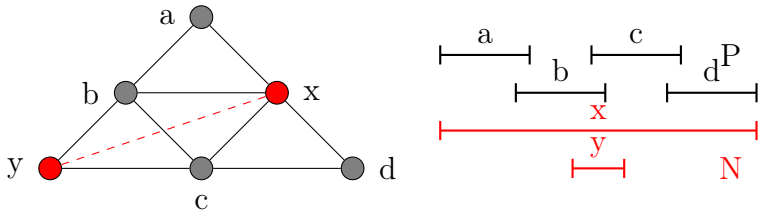

FIGURE 1.5 – Exemple d'un graphe d'intervalles *probe*

Graphes d'intervalles de confinement

C'est un graphes d'intervalles ou deux sommets sont adjacent si l'un des intervalles est contenu dans l'autre.

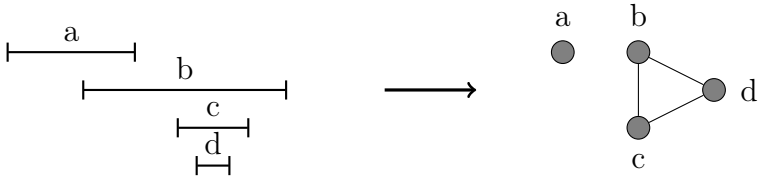

FIGURE 1.6 – Exemple d'un graphe d'intervalles de confinement et sa représentation en graphe simple

Graphes biparti

Graphe ou l'on peut séparer les sommets en deux ensembles E1 et E2 de manière à avoir :

E1 ∪ E2 = E et E1 ∩ E2 = ∅ et que aucun sommets d'un des 2 ensembles ne voisinent un sommet du même ensemble [2, 9].

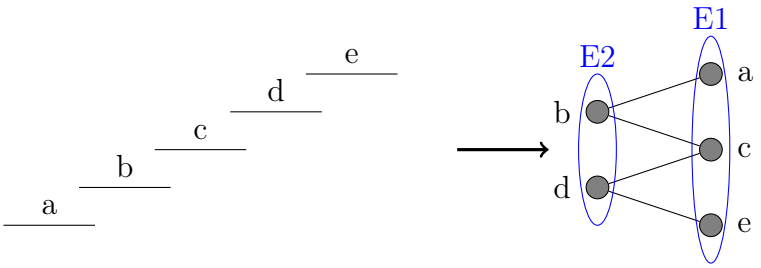

FIGURE 1.7 – Exemple d'un graphe biparti à partir d'une collection d'intervalles

Graphes d'intervalles circulaire

Graphe d'intervalles ou les intervalles sont disposé autour d'un cercle. Aucun intervalle n'en contient un autre. Cette classe de graphe est aussi appelée graphe d'arc circulaire propre.

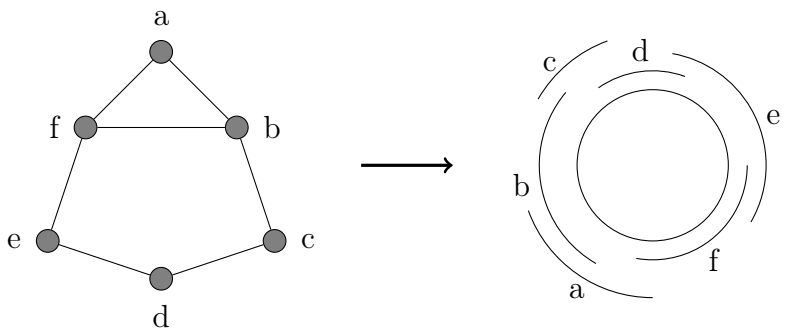

FIGURE 1.8 – Exemple d'un graphe d'intervalles circulaire et de sa représentation en graphe simple

Cette classe de graphe permet la représentation de trou qui sont impossible dans la classe des graphes d'intervalles.

1.3 Super-classes des graphes d'intervalles

Une super-classe des graphes d'intervalles est une classe de graphe dont les graphes d'intervalles font parti. Autrement dit, une classe C est une super-classe des graphes d'intervalles si graphes d'intervalles $\subseteq C$.

Graphes parfaits

Cette classe de graphes est liée fortement au problème de coloration des graphes qui consiste à attribuer une couleur aux sommets d'un graphe de sorte que chaque paires de sommets voisins ne sont pas de la même couleur. C'est un problème d'ordonnancement courant et a souvent pour but de trouver le nombre chromatique du graphe qui est le nombre minimum de couleur pour faire la coloration de ce graphe. On note $\chi(G)$ le nombre chromatique de G. On peut définir un graphe parfait par un graphe $G = (V, E)$ tel que pour tout sous-graphe H de G, on a : $\omega(H) = \chi(G)$ [2, 8].

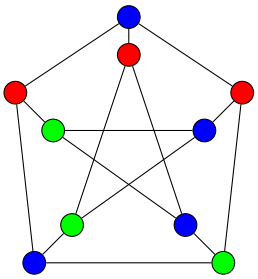

FIGURE 1.9 – Exemple du graphe de Petersen avec sa coloration

Sur la figure 1.9 on peut voir la coloration minimum de ce graphe qui à un nombre chromatique $\chi(G) = 3$.

Graphes cordaux

Les graphes cordaux sont une sous-classe des graphes parfait. Ce sont des graphes $G = (V, E)$ tels que, pour chaque cycle de G composé d'au moins 4 sommets, il existe au moins une corde. Une corde est une arrête reliant deux sommets non adjacent d'un cycle.

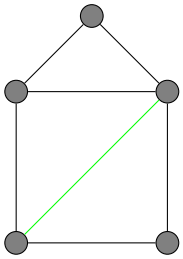

FIGURE 1.10 – Exemple d'un graphe cordal

Graphes fortement cordaux

C'est un graphe cordal tel que chaque cycle de taille paire avec au moins six sommets possède une corde impair. Une corde impaire est une corde reliant deux sommets à distance impaire l'un de l'autre d'un cycle.

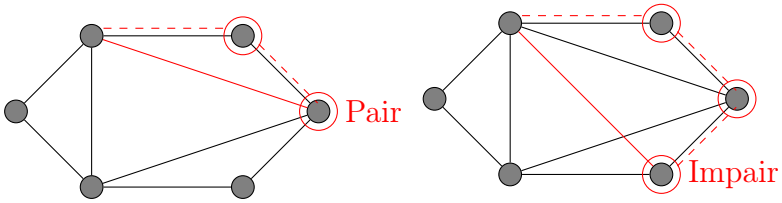

FIGURE 1.11 – Exemple d'un graphe fortement cordal

Graphes Meyniel

De la même manière que les graphes fortement cordaux, les graphes Meyniel sont cordés. Ce sont des graphes tels que, dès qu'il y a un cycle impair de longueur cinq ou plus, il y a au moins deux cordes à l'intérieur.

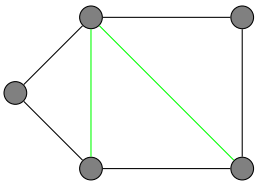

FIGURE 1.12 – Exemple d'un graphe Meyniel

Graphes sans trou pair

Ce sont des graphes qui ne contiennent pas de cycles induit avec un nombre pair de sommets (au minimum quatre ou six). Un cycle induit est un cycle vide (sans cordes) que l'on peut aussi appeler « trou ».

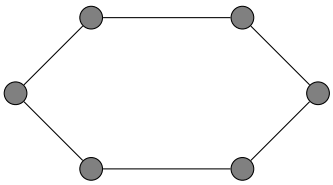

FIGURE 1.13 – Exemple d'un trou pair

Graphe Helly

C'est un graphe dont, pour chaque paire de cliques du graphe, ces paires ont une intersection non vide [18].

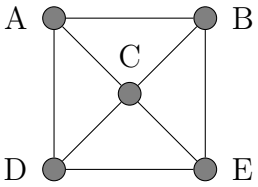

FIGURE 1.14 – Exemple d'un graphe Helly

Sur la figure 1.14 les différentes cliques du graphe sont $\{A, B, C\}$, $\{A, D, C\}$, $\{D, E, C\}$ et $\{B, E, C\}$. On peut remarquer que l'intersection de toutes ces cliques n'est pas vide car ils ont tous en commun le sommet C.

Graphes d'intervalles multiple

Un graphe d'intervalles multiple est un graphe où un sommet est représenté par un ensemble non vide d'intervalles.

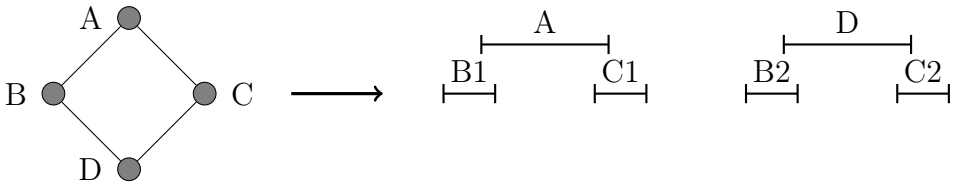

FIGURE 1.15 – Exemple d'un graphe et de sa représentation en graphe d'intervalles multiple

La figure 1.15 nous permet de remarquer que la classe des graphes d'intervalles multiple peut créer certains graphes qui ne peuvent être représentés en graphe d'intervalles [7].

Graphes de comparabilité

Ce sont les graphes dont on peut orienter ses arrêtes de façon transitive de sorte que s'il existe un arc de i vers j et de j vers k alors il existe également un arc de i vers k [2, 8].

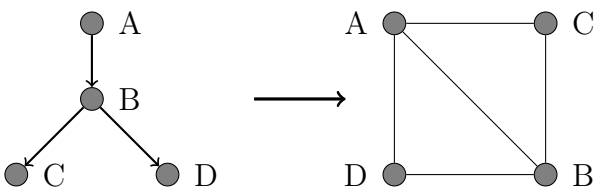

FIGURE 1.16 – Exemple d'un graphe de comparabilité et de sa représentation en graphe simple

Le graphe complémentaire d'un graphe d'intervalles est un graphe de comparabilité. Les graphes de comparabilité sont parfaits. Un graphe complémentaire d'un graphe G est noté $\overline{G}$, et est un graphe ayant les mêmes sommets que G tel que deux sommets sont adjacents si et seulement si ils ne le sont pas dans G.

Chapitre 2

Problème de domination

2.1 Définition du problème

Le problème de l'Ensemble Dominant (Dominating Set en anglais) est un problème très connu en informatique théorique. Il consiste à calculer un ensemble de sommets tel que chaque sommet appartient à cet ensemble ou a au moins un voisin dans l'ensemble. On peut le définir de façon formelle comme un problème de décision [12] :

Domination (Dominating Set)

Entrée : Un graphe quelconque $G = (V, E)$ et un entier $k \leq |V|$
Sortie : Existe-t-il un ensemble dominant D de taille au plus k pour le graphe G, c'est-à-dire un ensemble $D \subseteq V$ avec $|D| \leq k$ tel que tout sommet de $v \in V$ fait parti de D ou est voisin avec un sommet $u \in D$?

On note $\gamma(G)$ le nombre de domination d'un graphe G, soit la taille du plus petit ensemble dominant pour ce graphe. Ce problème est applicable dans de nombreux domaines et sa version d'optimisation qui consiste à calculer l'ensemble dominant de taille minimum peut être appliqué à de nombreux problèmes d'ordonnancement, en voici un exemple :
Vous disposez d'une carte des villes dans un secteur déserté médicalement. Votre but est de placer un minimum de structures médicalisées dans les villes pour subvenir au besoin de toute la population. Cependant, il y a une contrainte : chaque ville doit avoir une structure médicalisée ou être voisine avec une ville en possédant une.
Cet exemple est très concret mais ce problème peut être appliqué à de nombreux domaines différents [17].

Ce problème est très simple à comprendre, mais pour autant, complexe à résoudre avec un algorithme efficace (en temps polynomial) en considérant des graphes quelconques. En effet, Jon KLEINBERG et Eva TARDOS rappelent dans leur livre *Algorithm Design* que ce problème de l'ensemble dominant appliqué sur des graphes

quelconques est NP-complet [10]. En effet, la preuve de NP-complétude du problème de domination appraît déjà dans d'autres livre comme par exemple « Computers and intractability » de GAREY et JOHNSON datant de 1979 [5] ou aussi dans cet article de R. MOTWANI [16].

2.2 Classe P, NP, NP-complétude et NP-difficulté

P et NP sont des classes de complexité. Elles permettent de classifier les problèmes en fonction de la possibilité de les résoudre efficacement, par une machine de Turing déterministe ou non déterministe. On considère, en algorithmique, un algorithme efficace comme un algorithme qui s'exécute en temps polynomial. NP vient de l'anglais *nondeterministic polynomial time* et regroupe tous les problèmes où il est possible de vérifier une solution en temps polynomiale. Il est connu que $P \subseteq NP$. Il existe une question fondamentale concernant les classes de complexité P et NP qui reste encore sans réponse malgré de nombreuses année de recherche, cette question est : **est-ce que P = NP ?** Cette question fait partie des sept problème du millénaire proposés par l'institut de mathématiques Clay pour une récompense d'un million de dollars. Après de nombreuses années de recherches sur le sujet, certains chercheurs pensent même que $P \neq NP$ parce que si il était prouvé que $P = NP$ était vrai, ça signifierait qu'il serait possible de créer des algorithmes polynomiaux pour tout les problèmes NP.

On peut définir les problèmes NP-complets comme :

— un problème appartenant à la classe NP.

— un problème pour lequel tous les autres problèmes de la classe NP peuvent se réduire à ce problème par une réduction polynomiale.

Les problèmes remplissant la seconde condition uniquement sont des problèmes dit NP-difficiles. Il possible de faire des preuve de NP-complétude pour des problèmes grâce une réduction polynomiale qui consiste à réduire un problème Y à un problème X. Pour ce faire on considère une "boite noire" capable de résoudre X en une étape polynomiale. Si le problème Y est résolvable avec un nombre polynomial d'opération basique plus un nombre polynomial d'appel à la boite noire alors cela signifie que le problème Y est polynomialement réductible à X. On note ceci $Y \leq_p X$. Le livre *Algorithm Design* de Jon KLEINBERG et Eva TARDOS [10] possède un chapitre dédié à ces classes de problèmes.

Pour résoudre le problème de domination de manière efficace, nous pouvons nous restreindre à certains type de graphes ou alors à des variantes du problème de DOMINATION.

2.3 Résolution du problème pour les graphes d'intervalles

Le problème de domination étant NP-complet pour les graphes quelconques, nous allons nous restreindre à la classe des graphes d'intervalles pour le résoudre. Nous allons créer des algorithmes à l'aide de stratégies gloutonnes et de la programmation dynamique.
Nous allons considérer tout les graphes que rencontrerons nos algorithme connexes car on peut facilement remarquer que la valeur de la domination du graphe est la somme de la valeur de domination de toute les parties connexe.

Les premières approches que nous allons considérer par la suite ne sont pas toutes valides ou n'ont pas toutes aboutie à une preuve de pars la difficultés de leurs corrections.

2.3.1 Approche Gloutonne

Première approche

Soit G un graphe d'intervalle construit selon un modèle normalisé.
Soit Res une solution construite par l'algorithme pour G.

Algorithm 1: Domination-Intervalle(G=(V,E)

Data: Un graphe d'intervalles G=(V,E) construit selon un modèle normalisé.
Result: Une valeur de Domination minimale dans G $\gamma(G)$.

```
Res ⟵ ∅
foreach v ∈ V par date de début croissante do
    if ∃u ∈ Res : v ∈ N[u] then
        if r(v) > r(u) and N[u] \ N[Res \ {u}] ⊆ N[v] then
            Res ⟵ (Res \ {u}) ∪ {v}
        else if N[v] \ N[u] ≠ ∅ then
            Res ⟵ Res ∪ {v}
    else
        Res ⟵ Res ∪ {v}
return Res
```

Demonstration : Les sommets du graphe sont parcourus par date de début croissante. Pour avoir un ensemble dominant, il faut s'assurer que chaque sommet soit dominé par un autre sommet ou alors dans l'ensemble dominant. Or, on remarque que dans l'algorithme, si v n'est pas dominé (il n'existe aucun sommets dans l'ensemble sélectionné dominant v) alors v devient un sommet dominant ce qui

satisfait la propriété des dominating set décrite ci-dessus.
Inversement, si un sommet v est déjà dominé par un sommet $u \in V$, il faut vérifier si ce sommet peut permettre de couvrir de nouveaux sommets donc s'il possède des voisins que le sommet u ne domine pas. Si tel est le cas, ce sommet doit devenir un nouveau sommet dominant.

Cependant, avant de rajouter ce sommet à l'ensemble, il faut vérifier s'il ne fait pas « mieux » que le sommet u qui le domine. C'est-à-dire si les sommets dominés seulement par u sont dominables aussi par v et que v peut posséder des voisins supplémentaires à u, c'est-à-dire des voisins qui se trouvent à droite de v (voisins avec v mais pas avec u) car de part le choix de notre parcours $l(v) > l(u)$ alors v peut devenir dominant à la place de u.

Dans cet algorithme, on utilise ces propriétés ainsi que le parcours par date de début croissante afin de s'assurer que le résultat soit un ensemble dominant. Cet algorithme renverra toujours un ensemble dominant car dès qu'un sommet n'est pas dominé il devient dominant. La seule opération pouvant briser la propriété est le fait de changer un dominant par un autre. Et la encore, on vérifie que le nouveau dominant domine toujours les sommets dominé uniquement par l'ancien sommet.

On peut donc en conclure que *Res* est toujours une solution de Dominating set. □

Nous démontrons maintenant que la solution *Res* construite pour G par l'algorithme est de taille minimum. Par contradiction, supposons que *Res* ne soit pas optimale. Parmi toutes les solutions optimales, notons *Opt* une solution optimale qui fait les mêmes choix que *Res* le plus longtemps possible. Étant donnée deux solution $S1$ et $S2$ on définit $div(S1, S2) = \min\{k \in \mathbb{N}$ tel que $i_k \in S1$, $i'_k \in S2$ et $i_k \neq i'_k\}$. *Opt* est une solution optimale pour laquelle $div(Res, Opt)$ est le plus grand possible. Donc si on trie les intervalles de *Res* et *Opt* par date de début croissante, on a :
$Res = \{i_1, i_2, ..., i_t\}$ et $Opt = \{i'_1, i'_2, ..., i'_k\}$.

Comme on a supposé que *Res* n'était pas optimale, ce qui veut dire qu'il existe un indice l tel que $i_x = i'_x \ \forall x < l$ et $i_l \neq i'_l$.
On a donc :

$$\begin{array}{lcccccccccc} Res & = & i_1 & i_2 & i_3 & \cdots & i_{l-1} & i_l & i_{l+1} & \cdots & i_t \\ & & \| & \| & \| & & \| & \not\parallel & ? & & ? \\ Opt & = & i'_1 & i'_2 & i'_3 & \cdots & i_{l-1} & i_l & i_{l+1} & \cdots & i_k \end{array}$$

Montrons que si, dans *Opt*, on remplace i'_l par i_l alors *Opt* est encore une solution optimale.
Considérons plusieurs cas entre i_l et i'_l.

— Si $l(i'_l) < l(i_l)$ alors cela signifie que l'algorithme a remplacé i'_l par i_l car, le parcours des intervalles se fait par ordre de début croissant, donc i'_l a déjà été traité avant de traiter i_l. On peut aussi déduire, d'après les conditions de l'algorithme, que $r(i'_l) < r(i_l)$ et que, d'après la 2^{eme} condition, le voisinnage dominé uniquement par i'_l est compris dans le voisinnage fermé de i_l. On peut donc remplacer i'_l par i_l dans Opt pour obtenir une nouvelle solution optimale pour laquelle $div(Res, Opt)$ est plus grand. Contradiction sur le fait que Opt était une solution optimale avec le plus grand $div(Res, Opt)$ possible.

— Si $l(i'_l) > l(i_l)$ alors cela signifie que l'algorithme n'a pas remplacé i_l par i'_l d'après le parcours des intervalles. On peut ainsi en déduire soit que $r(i'_l) \leq r(i_l)$ donc i'_l est contenu dans i_l donc $N[i'_l] \subseteq N[i_l]$ ou alors que les voisins dominables uniquement par i'_l sont aussi dominables par i_l. On peut donc remplacer i'_l par i_l dans Opt pour obtenir une nouvelle solution optimale pour laquelle $div(Res, Opt)$ est plus grand. Contradiction sur le fait que Opt était une solution optimale avec le plus grand $div(Res, Opt)$ possible.

Nous ne considérons pas $l(i'_l) = l(i_l)$ car, selon quel intervalle de i'_l et i_l sera parcouru en premier on sera soit dans le cas $l(i'_l) > l(i_l)$ ou $l(i'_l) < l(i_l)$ dans le modèle d'intervalles normalisé.

Comme on peut remplacer i'_l par i_l dans Opt pour obtenir une nouvelle solution optimale, d'après le choix glouton de l'algorithme, on peut étendre cette propriété à l'ensemble des éléments de Res, donc Res est bien une solution optimale. □

Seconde approche

Soit G un graphe d'intervalle construit selon un modèle d'intervalles normalisé. Soit Res une solution construite par l'algorithme pour G et Dom une solution pas forcément optimale pour dominating set.

Algorithm 2: Domination-Intervalle-2(G=(V,E)

Data: Un Graphe d'intervalles G=(V,E) construit selon un modèle normalisé.

Result: Une valeur de Domination minimale dans G $\gamma(G)$.

$Res \longleftarrow \emptyset$

foreach $v \in V$ *par date de fin croissante* **do**

  if $v \notin N[Res]$ **then**

    $\exists u \in N[v] : \max(r(u))$

    $Res \longleftarrow Res \cup u$

return Res

Demonstration : Lorsque l'algorithme rencontre un intervalle non dominé, il choisit son voisin (dans son voisinnage fermé) qui fini le plus tard. Comme cet algorithme est glouton et que l'on effectue ce choix sur tout les intervalles alors on est

assuré que tout les sommets seront dominé à la fin de l'exécution de l'algorithme. □

Nous démontrons maintenant que la solution *Res* construite pour G par l'algorithme est de taille minimum. Par contradiction, supposons que *Res* ne soit pas optimale. Parmis toutes les solutions optimale, notons *Opt* une solution optimale qui fait les mêmes choix que *Res* le plus longtemps possible. Étant donnée deux solution $S1$ et $S2$ on définit $div(S1, S2) = \min\{k \in \mathbb{N}$ tel que $i_k \in S1$, $i'_k \in S2$ et $i_k \neq i'_k\}$. *Opt* est une solution optimale pour laquelle $div(Res, Opt)$ est le plus grand possible. Donc si on trie les intervalles de *Res* et *Opt* par date de fin croissante on a $Res = \{i_1, i_2, \cdots, i_t\}$ et $Opt = \{i'_1, i'_2, \cdots, i'_k\}$.

Comme on a supposé que *Res* n'était pas optimale, ce qui signifie qu'il existe un indice l tel que $i_x = i'_x \forall x < l$ et $i_l \neq i'_l$.
On a donc :

$$\begin{array}{lcccccccccc} \text{Res} & = & i_1 & i_2 & i_3 & \cdots & i_{l-1} & i_l & i_{l+1} & \cdots & i_t \\ & & \| & \| & \| & & \| & \not\parallel & ? & & ? \\ \text{Opt} & = & i'_1 & i'_2 & i'_3 & \cdots & i_{l-1} & i_l & i_{l+1} & \cdots & i_k \end{array}$$

Montrons que si, dans *Opt*, on remplace i'_l par i_l alors *Opt* est encore une solution optimale.
Considérons plusieurs relations entre i_l et i'_l.
Si $r(i'_l) < r(i_l)$, cela signifie que l'algorithme, pour un sommet x voisin de i'_l et i_l, à choisi de sélectionner i_l car il fini plus tard. On peut en déduire que les voisins que i'_l aurait pu dominer sont aussi dominables par i_l. On peut donc remplacer i'_l par i_l dans *Opt* pour obtenir une nouvelle solution optimale.
Si $r(i'_l) > r(i_l)$, cela signifie que l'algorithme avait sélectionné i_l, pour un sommet x voisin de i_l, car si ce x est était aussi voisin avec i'_l on l'aurait choisit car il fini plus tard. On peut donc en déduire que i_l était un bon choix de part le choix glouton de l'algorithme, on prend toujours le sommet le plus grand à droite. On peut donc déduire que $l(i_l) < l(i'_l)$ et donc $i'_l \notin N[x]$ parce que sinon i'_l aurait été privilégié. On peut donc remplacer i'_l par i_l dans *Opt* pour obtenir une nouvelle solution optimale pour laquelle $div(Res, Opt)$ est plus grand. Contradiction sur le fait que *Opt* était une solution optimale avec le plus grand $div(Res, Opt)$ possible.

Comme on peut remplacer i'_l par i_l dans *Opt* pour obtenir une nouvelle solution optimale, d'après les choix glouton de l'algorithme, on peut étendre cette propriété à l'ensemble des éléments de *Res*, donc *Res* est bien une solution optimale. □

2.3.2 Approche par programmation Dynamique

Première approche

Soit G un graphe d'intervalles créé selon un modèle d'intervalles normalisé.
Notons $opt[i]$ une solution construite par l'algorithme pour G et pour les i premiers

intervalles numérotés dans l'ordre croissant de leur date de fin. Il faut donc trier les intervalles par dates de fin croissante, ce qui peut être réalisé facilement en temps $O(nlogn)$ par tri fusion, ou mieux encore, en temps $O(n)$ par un tri par dénombrement puisque le modèle d'intervalles est normalisé.

Algorithm 3: Domination-Dynamique(G=(V,E))

Data: Un Graphe d'intervalles G=(V,E) construit selon un modèle normalisé.

Result: Une valeur $\gamma(G)$ de Domination minimale pour G.

$opt[0] \longleftarrow \emptyset$

for *i de 1 à* $|V|$ *faire* **do**

  $j \longleftarrow \min(N[i]) - 1$

  if $i \notin \bigcup_{u \in opt[i-1]} N[u]$ **then**

    $opt[i] \longleftarrow opt[j] \cup i$

  else

    if $|opt[j] \cup i| \leq opt[i-1]$ **then**

      $opt[i] \longleftarrow opt[j] \cup i$

    else

      $opt[i] \longleftarrow opt[i-1]$

return $opt[|V|]$

Le j dans cet algorithme correspond au sommet qui fini le plus tard mais qui n'est pas voisin avec le sommet courant car les intervalles sont triés par date de fin croissantes. On regarde donc si le sommet courant est dominé dans la solution précédente. S'il ne l'est pas alors on prend la solution de j à laquelle on ajoute le sommet i.
Si le sommet courant est dominé, alors on regarde qu'elle est la plus petite solution entre la solution de j à laquelle est ajoutée i et la solution précédente.

Cet algorithme, de part le choix de j ne trouve que des ensembles dominant disjoints (ensemble dominant D tel que $\forall x1, x2 \in D$, $x1$ et $x2$ ne se voisinent pas), donc, lorsque la solution de domination ne peut être disjointe, la solution n'est pas minimale. Voici ci-dessous un contre-exemple pour cette algorithme :

FIGURE 2.1 – Solution de l'algorithme à gauche et la solution de taille minimum à droite

Dans la figure 2.1, on peut remarquer que la solution optimale nécessite de

prendre deux sommets voisins. Or, d'après le choix du j et de la solution qui en découle, on ne peut avoir deux sommets dans l'ensemble dominant adjacent. On peut donc en conclure que cet algorithme renvoi toujours une solution dominante mais pas toujours de taille minimale.

Deuxième approche

Soit G un graphe d'intervalles construit selon un modèle d'intervalles normalisé. Soit $opt[i]$ une solution construite à partir de l'algorithme pour G en considérant les i premiers intervalles numérotés dans l'ordre croissant de leur date de fin.

Algorithm 4: Domination-Dynamique-2(G=(V,E))

Data: Un graphe d'intervalles G=(V,E) construit selon un modèle normalisé.
Result: Une valeur $\gamma(G)$ de domination minimale dans G.

```
opt[0] ⟵ ∅
for i de 1 à |V| faire do
    if ¬∃j ∈ opt[i − 1] ∩ N(i) then
        opt[i] ⟵ opt[i − 1] ∪ {i}
    else
        j ⟵ u ∈ opt[i − 1] ∩ N(i) : min(l(u))
        if min(N[j] \ (N[opt[j] \ {j}])) < min(N[i] \ N[opt[j] \ {j}]) then
            if min(N[j] \ {i}) < max(N(i)) then
                opt[i] ⟵ opt[j] ∪ {i}
            else
                opt[i] ⟵ opt[j]
        else
            opt[i] ⟵ opt[min(N[i]) − 1] ∪ {i}
return opt[|V|]
```

Demonstration : Prouvons tout d'abord que l'algorithme renvoie toujours une solution qui est un ensemble dominant. Pour le cas où 0 sommets sont considéré dans le graphe, nous avons une solution vide qui est bien un ensemble dominant pour le graphe vide.
Ensuite nous parcourons tous les sommets par date de fin croissante. Si le sommet i parcouru n'est pas dominé par la dernière solution, donc s'il n'existe pas un $j \in opt[i-1] \cap N(i)$ alors il nous faut rajouter i à la dernière solution $opt[i-1]$.

Si le sommet i est déjà dominé par un ou plusieurs sommets de la dernière solution, définissons un j comme étant le dominant du sommet i qui commence le plus tôt.

On peut assez facilement remarquer que s'il y a plusieurs sommets qui dominent un seul i, comme i a une fin plus grande que ses dominants alors le sommet j n'est pas voisin avec ces derniers et les dominants autre que j sont inclus dans i. Cela peut être appuyé par le fait que si j était voisin avec ces sommets, on les aurait pas rajoutés dans la solution car ils auraient été couvert par j. Le sommet j n'étant pas inclus dans i, il commence donc avant i. De ce fait, $\min(N[j]) \leq \min(N[i])$. De la même manière, i finit plus tard que j donc $\max(N[j] \setminus \{i\}) \leq \max(N(i))$.
Maintenant que j est défini, il faut considérer plusieurs cas pour former la solution :

— j domine des sommets en plus par rapport à i qui ne sont pas dominé par un autre sommet de la solution à l'indice j : Sachant que $\max(N[j] \setminus \{i\}) \leq \max(N(i))$, on peut en déduire que les voisins supplémentaires de j sont situés à gauche de ce dernier et ils sont, de ce fait, numérotés avec un plus petit nombre par rapport aux voisins de i. On peut en déduire que si $\min(N[j] \setminus N[opt[j] \setminus \{j\} < \min(N[i] \setminus N[opt[j] \setminus \{j\}])$ alors cela signifie qu'il existe un sommet dominé exclusivement par j qui n'est pas voisin de i. Il faut donc que $opt[j]$ soit inclus dans $opt[i]$.

— i domine exclusivement des sommets en plus que j : De la même manière que le point précédent et sachant que $\min(N[j]) \leq \min(N[i])$ alors on peut en déduire que les voisins supplémentaire de i sont à droite de j et, par définition du parcours par date de fin croissante, à gauche de la fin de i donc inclus dans i. Ainsi $\max(N[j] \setminus \{i\}) < \max(N(i))$ vérifie l'existence de ces sommets. Si ils existent, alors il faut que le sommet i fasse parti de la solution.

En s'appuyant sur ces deux propriétés autour de j et i, on peut former une solution (construite par l'algorithme).
Si j fait parti de la solution alors on regarde si i doit en faire parti aussi. Ainsi nous avons soit, comme solution pour $opt[i]$, la plus petite solution parmi :

— $opt[j] + \{i\}$
— $opt[j]$
— $opt[\min(N[i]) - 1] \cup \{i\}$

Le dernier cas est celui ou j ne fait pas mieux que i à gauche, dans ce cas prendre i est obligatoire qu'il soit meilleur que j à droite ou équivalent à l'indice i. Il faut donc prendre i avec la solution du plus grand sommet qui fini avant que i commence donc que i ne voisine pas, soit $opt[\min(N[i]) - 1]$ auquel on rajoute i.

Cet algorithme permet donc d'obtenir une solution de dominating set pour n'importe quel graphes d'intervalles. □

Il ne reste qu'à prouver que cet algorithme renvoie toujours une solution minimale de domination. Cependant, l'algorithme dans l'état actuel ne nous permet pas facilement d'établir une preuve de part le manque de spécificité sur la forme de la solution créée.
Nous allons donc modifier cette algorithme dans la troisième approche ci-dessous pour faciliter la preuve.

Troisième approche

Soit G un graphe d'intervalles construit selon un modèle d'intervalles normalisé. Soit $opt[i]$ une solution construite à partir de l'algorithme pour G en considérant les i premiers intervalles numérotés dans l'ordre croissant de leur date de fin.

Algorithm 5: Domination-Dynamique-3(G=(V,E))

Data: Un graphe d'intervalles G=(V,E) construit selon un modèle normalisé.

Result: Une valeur $\gamma(G)$ de Domination minimale dans G.

$opt[0] \longleftarrow \emptyset$

for *i de 1 à* $|V|$ *faire* **do**

$Soitk \in V : r(k) < l(i)$ et $\max(l(k))$

$Soitj \in N[k] : r(j) \leq r(i)$ et $\min(l(j))$

$opt[i] \longleftarrow opt[j] \cup \{i\}$

$\exists z \in V : \max(l(z))$

return $\gamma(G) = \min(opt[u]), u \in N[z]$

Demonstration : La valeur retournée par l'algorithme dépend des différents $opt[i]$ qui eux-même dépendent des valeurs $opt[l]$ pour des $l < i$.

(*) Montrons que $\forall l$, $opt[l]$ contient un ensemble dominant de taille minimum pour le graphe constitué des k premiers intervalles et tel que l fait partie de cet ensemble.

Comme cas de base, nous avons $opt[0] \longleftarrow \emptyset$ ce qui est bien un ensemble de taille minimum pour le graphe des 0 premiers intervalles tel que l'intervalle 0 fasse partie de la solution (il n'existe pas).

Supposons que $\forall l < i$, $opt[l]$ est une solution car la propriété (*) est vérifiée, et montrons que $opt[i]$ est calculé correctement par l'algorithme.
Connaissant i et sachant qu'il doit faire parti de la solution alors il faut l'ajouter à une solution minimale qui permet de dominer tous les intervalles qui terminent avant le début de i. Prenons alors un intervalle k tel que $r(k) < l(i)$ avec $l(k)$ le plus grand possible. De cette manière, on a l'intervalle allant le moins à gauche parmi les intervalles non dominés par i. Un tel choix de k permet de s'assurer de dominer tous les intervalles à gauche de i car ils seront tous avec un début à gauche de k aussi. Comme les intervalles non dominéd par i sont tous avec une fin avant le début de ce dernier, le k choisi au préalable possède un j dans son voisinage tel que j commence le plus tôt possible alors ce j est un choix optimale pour dominer ce k car il rencontre au moins le même nombre d'intervalles non dominé par i que les autres voisins de k.
On peut en déduire que j fait parti de la solution donc $opt[j]$ fait parti de la solution

$opt[i]$.
Comme $opt[j]$ est une solution minimale contenant j pour les j premiers intervalles et que cette solution domine tout les intervalles non dominés par i alors $opt[j] \cup \{i\}$ est une solution minimale pour les i premiers intervalles noté $opt[i]$.
Donc cela veut dire que la propriété (*) est vrai au rang i. Par récurrence elle est vrai pour tout $i \leq n$.

Pour former la solution finale optimale, prenons un z ayant le plus grand $l(z)$. Pour que ce z soit dominé, il faut qu'un de ses voisins ou lui-même fasse parti de la solution optimale.
Comme z est choisi avec le début le plus grand alors tous les autres intervalles du graphe sont à gauche de z. On peut en déduire que les *opt* calculés pour les voisins de z (z inclus) sont des solutions pour tout le graphe.
Donc comme un des voisin de z doit être dominant pour avoir une solution, alors, un *opt* d'un voisin de z doit être choisi obligatoirement. En vue d'avoir une solution minimale, il nous suffit de choisir la solution la plus petite des voisins de z (z inclu). □

Par l'absurde : Soit y tel que $opt[y]$ est une solution optimale pour tout le graphe et $y \notin N[z]$. Pour que la solution soit optimale, z doit être dominé donc il doit y avoir un dominant parmi $N[z]$.
D'après (*), $opt[i]$ est une solution pour les i premiers intervalles tel que i est dans la solution. Vu que $opt[y]$ est optimal pour tout le graphe, alors il y a un intervalle de $N[z]$ dans la solution, donc $r(y) \geq r(z)$ d'après le remplissage de *opt*. Comme $y \notin N[z]$, on a $l(y) > r(z)$ donc $l(y) > l(z)$. On a donc une contradiction sur z car il est choisi comme étant le plus grand possible. □

Analyse du temps d'exécution : Cet algorithme comporte une seule boucle for qui s'exécute n fois et comporte des opération que l'on suppose exécutable en temps constant $O(1)$. La suite de la boucle s'exécute lui aussi en temps constant ainsi on peut donc en déduire que le temps d'exécution de cet algorithme est de l'ordre de $O(n)$.

Chapitre 3

Domination Romaine et Domination Romaine Quasi Totale

3.1 Définition du problème

3.1.1 Domination Romaine

La Domination Romaine (Roman Domination en anglais) est une variante du problème de domination. Il conserve donc le même but qui est celui de créer un ensemble de sommets dominant l'entièreté du graphe. Cependant, il y a des paramètres en plus : Chaque sommet est affecté à un type de sommets, les **V1**, les **V2** et les **V0**. Les sommets de type V1 sont des sommets qui se domine uniquement eux-même, les sommets de type V2 se dominent eux-même plus les sommets qui leurs sont voisins et les sommets de type V0 qui ne domine rien et doivent donc être dominé par un V2. Le choix des V1 et V2 a aussi un impact dans la valeur de la solution qui est définie par $\gamma_R(G) = |V1| + 2 * |V2|$ pour un graphe G [11, 13, 3].
De ce fait, pour trouver une solution minimale de domination romaine il faut donc prendre en compte le poids des différents types de sommets pour minimiser la valeur de la solution (voir Figure 3.1).

Dans l'article [13] de M. LIEDLOFF accompagné par T. KLOKS, J. LIU et S.-L. PENG, ils ont construit un algorithme dynamique avec un temps d'exécution linéaire pour résoudre le problème de domination romaine limitée au graphes d'intervalles. Pour ce faire, ils ont énuméré les différents cas possibles que pouvait prendre la structure d'une extension d'une sous-solution de domination romaine partant d'un sommet de V2 et s'assurant de revenir sur un V2 afin de pouvoir conserver la même structure pour l'extension suivante.
Voici ci-après un exemple d'une solution minimale de domination romaine pour un graphe d'intervalles G :

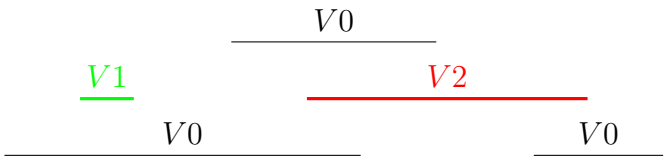

FIGURE 3.1 – Exemple d'une solution minimale de domination romaine pour un graphe d'intervalle d'ordre 5

On peut distinguer sur la Figure 3.1 les sommets de type $V0$ en noir, les sommets de type $V1$ en vert et les sommets de type $V2$ en rouge. On peut donc calculer la valeur de la solution qui est $\gamma_R(G) = 3$.

3.1.2 Domination Romaine Quasi Totale

La DOMINATION ROMAINE QUASI TOTALE (QUASI TOTAL ROMAN DOMINATION EN ANGLAIS (QTRD)) est une variante de la domination romaine. Il possède donc le même objectif de créer un sous-ensemble dominant avec des sommets de type $V0$, $V1$ et $V2$ avec comme valeur $\gamma_{QTRD}(G) = |V1| + 2 * |V2|$ pour un graphe G.
Pour comprendre le problème de domination romaine quasi totale, il faut connaître la notion de sous-graphe induit : G' est un sous-graphe induit d'un graphe $G = (V, E)$ si pour tout couple $(u, v) \in |V|$, u et v sont connectés dans $G' = (V', E')$ si et seulement si ils sont connectés dans G.
Une fonction de QTRD f_{QTRD} d'un graphe G est similaire à une fonction de domination romaine mais se voit rajouter une propriété. En effet, une fonction $f_{QTRD} : V(G) \longrightarrow \{0, 1, 2\}$ pour le graphe G possède une règle en plus : Si x est un sommet isolé du sous-graphe induit par l'ensemble des sommets étiquettés par 1 et 2 alors $f_{QTRD}(x) = 1$. Autrement dit, un $v \in V(G)$ tel que $f_{QTRD}(v) = 2$ doit être voisin avec un $u \in V(G)$ tel que $f_R(u) = 1$ ou 2 [6, 14, 15].
Voici ci-après un exemple d'une solution minimale de domination romaine pour un graphe d'intervalles G :

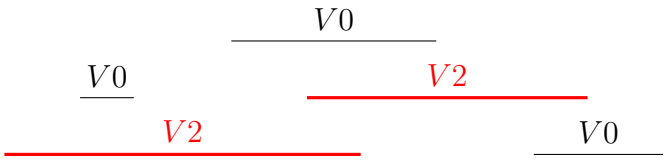

FIGURE 3.2 – Exemple d'une solution minimale de domination romaine quasi totale pour un graphe d'intervalle d'ordre 5

Sur la Figure 3.2, de la même manière que pour la Figure 3.1, on peut distinguer les sommets de type $V0$, $V1$ et $V2$ les uns des autres. On remarque aussi que la solution minimale est différente que celle pour la domination romaine. En effet, cette fois ci la valeur de la solution est $\gamma_{QTRD}(G) = 4$. Ce changement s'explique par le fait

que le sommet de type $V2$ que nous avions choisi pour la solution de la domination romaine était isolé ce qui ne respectait pas les règles d'une solution de QTRD.

3.2 Résolution de QTRD pour les graphes d'intervalles

De la même manière que pour la section 2.3 nous considérons uniquement les graphes connexes car on peut facilement remarquer que la valeur de la domination du graphe est la somme de la valeur de domination de toute les parties connexe.

3.2.1 Approche gloutonne

Première approche

Soit G un graphe d'intervalles quelconque. Soit Res une solution construite par l'algorithme pour G.

Algorithm 6: Domination-Romaine-Quasi-Totale-Intervalles(G=(V,E)

Data: Un graphe d'intervalles G=(V,E) construit selon un modèle normalisé.

Result: Une valeur $\gamma_{QTRD}(G)$ de Domination Romaine Quasi Totale minimale dans G.

$Res[0] \longleftarrow V$
$Res[1] \longleftarrow \emptyset$
$Res[2] \longleftarrow \emptyset$
foreach $v \in V$ *par degré (nombre de voisins) croissant et date de fin croissante si conflit* **do**
  if $v \notin Res[1] \cup Res[2]$ ***and*** $\nexists u \in N(v) \cap Res[2]$ **then**
    Soit $u \in N[v]$ avec le plus grand nombre de voisins non dominé et de plus grand degré en cas de conflit
    if $|N[u]| \geq 3 - |N[u] \cap Res[2]|$ **then**
      $Res[2] \longleftarrow Res[2] \cup \{u\}$
      $Res[0] \longleftarrow Res[0] \setminus \{u\}$
    else
      $Res[1] \longleftarrow Res[1] \cup \{v\}$
      $Res[0] \longleftarrow Res[0] \setminus \{v\}$
foreach $v \in Res[2]$ *par date de fin croissante* **do**
  if *v est isolé dans le sous-graphe induit par* $Res[2] \cup Res[1]$ **then**
    Soit $u \in N(v) \cap Res[0]$ tel que $r(u) est le plus grand possible$
    $Res[1] \longleftarrow Res[1] \cup \{u\}$
    $Res[0] \longleftarrow Res[0] \setminus \{u\}$
return *Res*

La première boucle permet de construire une solution de domination romaine particulière. En effet, on parcours les intervalles par degré croissant cette fois et lorsque l'on rencontre sur un sommet non dominé, on regarde quel est le voisin avec le plus grand voisinage. On regarde ce qui est le plus économe entre le mettre dans V2 ou dans V1 le voisin sélectionné. Pour se faire, on regarde si le voisin a au moins 3 voisins (avec lui même) non dominés. Si c'est le cas, c'est aussi rentable de le prendre en V2 ou les 3 en V1 car on devra peut être rajouter un sommet de ce voisinage dans V1 si le sommet de V2 est isolé du sous-graphe induit par l'ensemble des V1 et V2.
Cependant, il pourra devenir voisin avec un autre V2 par la suite de l'algorithme donc on privilégie le fait de prendre des V2 plutôt que des V1. Comme on parcourt tous les sommets et on fait en sorte que tous les sommets soit dominés alors la première boucle renverra toujours une solution de domination romaine où l'on prend des V2 seulement si c'est mieux que de prendre des V1.

La deuxième boucle rajoute le moins de V1 possible pour que aucun sommet de V2 ne soit isolé. En parcourant les sommets de V2 isolés par date de fin croissante, on peut rajouter son voisin qui va le plus loin à droite dans V1 car c'est le sommet qui a le plus de chance de croiser un autre V2.

Comme la première boucle s'assure que les V2 sont choisis pour être le meilleur choix et que la deuxième boucle s'assurent de respecter les propriétés de QTRD en ajoutant le moins de V1 possible donc on est assuré d'avoir une solution de domination romaine Quasi Totale. □

Cependant, cet algorithme ne renvoit pas toujours une solution minimale. En effet, de part le choix du parcours des sommets par degré croissant, on effectue parfois un choix de V2 optimal pour dominer un sommet à moindre coût mais ce choix de V2 n'est pas optimal en considérant le graphe en entier.
On peut donc créer un contre-exemple où l'algorithme trouvera qu'il sera optimal de dominer un certain sommet par un V2 parce qu'il a un voisin qui possède au moins 3 sommets non dominés alors que ce n'est pas un choix menant à une solution optimale. Les Figures 3.3 et 3.4 qui vont suivre sont des contre-exemples qui ont était générés par un programme python implémentant l'algorithme et en comparant ses résultats à un brute force [1].

1. Le code est hébergé sur github à l'adresse suivante : https://github.com/Itasuka/roman-domination

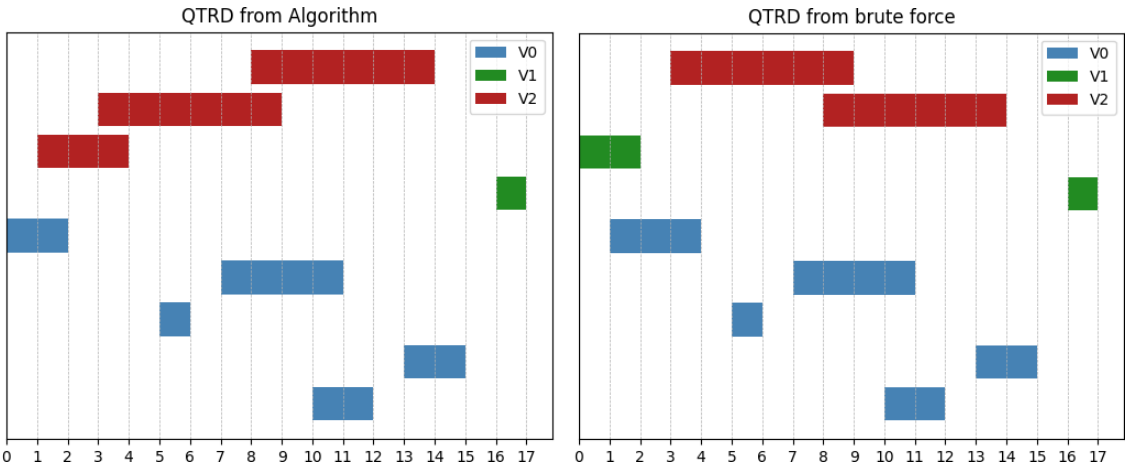

FIGURE 3.3 – Contre exemple pour la première version de l'algorithme de QTRD

Sur la figure 3.3 on peut remarquer la différence des solutions entre l'algorithme et un solveur pour la QTRD utilisant la force brute. Un solveur utilisant la force brute est un programme qui essaie toutes les solution pour trouver une solution d'un problème, dans notre cas, une valeur minimale d'une solution de QTRD. Le solveur utilisé ici a été programmé en python pendant ce stage. Il possède malheureusement un temps d'exécution exponentiel, il ne nous donc d'être utilisé que sur des graphes de taille modeste.

Sur cet exemple, on remarque que la seule différence entre la solution de l'algorithme et la solution du brute force se trouve sur les 2 premiers intervalles. En effet, on remarque que l'intervalle qui débute à 1 et qui fini à 4 noté $(1, 4)$ a était sélectionné comme $V2$ parce que comme l'intervalle $(0, 2)$ est de degré 2 il est parcouru en 2^{eme} après l'intervalle (16, 17) (de pars la définition du parcours par degré croissant), qui lui est affecté à l'ensemble $V1$ car il est isolé. Le choix de prendre l'intervalle (1, 4) en $V2$ est un choix optimal à l'instant où il est choisi car il possédait 3 voisins non dominés (y compris lui même) mais ne l'est pas pour la solution minimale du graphe car l'intervalle (3, 9) couvre 2 des sommets que domine (1, 4) et domine d'autres intervalles supplémentaires. L'erreur de l'algorithme vient donc du choix du parcours des sommets car il fait les bons choix mais pas au bon moment.

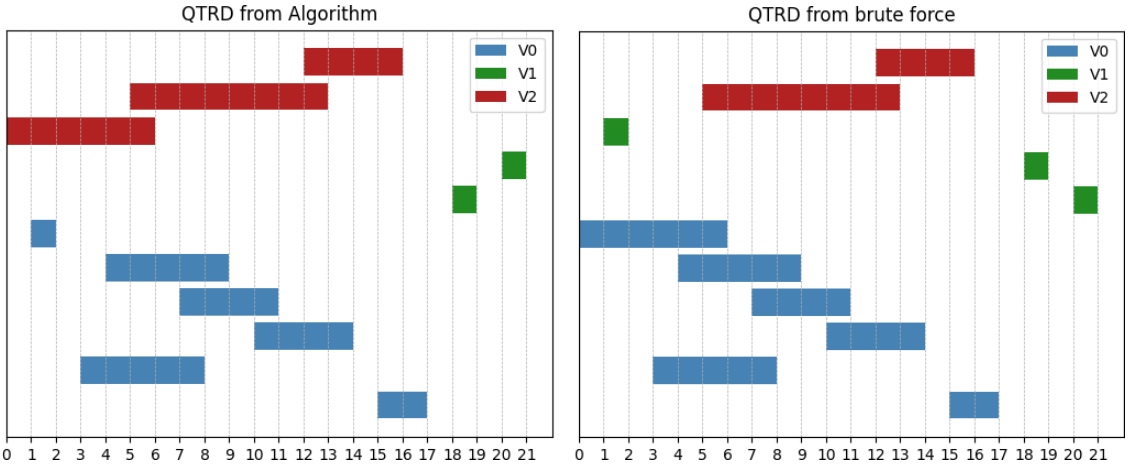

FIGURE 3.4 – Contre exemple pour la première version de l'algorithme de QTRD

De la même manière que sur la figure 3.3, on peut remarquer sur la figure 3.4 le même mauvais choix entre l'intervalle (1, 2) et l'intervalle (0, 6).

3.2.2 Approche par programmation dynamique

Seconde approche

Pour cette seconde approche, nous allons nous appuyer sur un algorithme de programmation dynamique renvoyant la solution minimale de domination romaine établie dans [13] pour les graphes d'intervalles. Cet algorithme dynamique s'appuient sur l'énumération d'un nombre fini cas. En effet, ils établissent dans ce papier, à partir d'une situation de départ qui est un sommet v de type $V2$, l'ensemble des possibilités d'extensions de la sous-solution de v finissant toujours par un sommet de $V2$ sauf si on arrive en fin de parcours du graphe. L'algorithme teste donc toutes les possibilités d'extensions et enregistre les résultats minimaux dans un tableau. Le fait de finir chaque extension sur un sommet de $V2$ permet de boucler sur les mêmes cas et donc de contrôler le déroulement de l'algorithme.

Comme nous concevons un algorithme pour la domination romaine quasi totale, il nous faut donc faire des modifications concernant les possibilités d'extensions d'une sous-solution ainsi que sur la modélisation de celles-ci. En effet, considérant un sommet v et sa sous-solution, il nous faut mémoriser si la propriété de QTRD est respectée pour le sommet v parce que, si elle n'est pas respectée, les extensions sont différentes.

Dans l'article [13], les auteurs énoncent divers lemmes concernant la solution de QTRD. Certains sont encore valides pour notre problème mais il nous faut en construire/adapter quelques uns.
Parmi les lemmes encore valide, on peut citer :

— 2 sommets de type $V2$ ne peuvent être contenus l'un dans l'autre dans une solution optimale car si le sommet inclut devient un $V1$ alors on obtient une solution de moindre coût

— il ne peut exister une clique de V2 de taille 3 ou plus dans la solution optimale et ce pour la même raison que décrite dans le papier [13]. En effet, considérons trois sommets $i1, i2, i3 \in V2$ qui forment une clique. Comme aucun intervalles n'est contenu dans un autre alors $l(i1) < l(i2) < l(i3) < r(i1) < r(i2) < r(i3)$. On remarque donc que le voisinage de $i2$ est contenu dans l'union des voisinages de $i1$ et $i3$, donc $i2$ n'est pas nécessaire à la formation d'une solution optimale.

Cependant, il existe un lemme qui ne correspond plus. Ce lemme est le fait que l'ensemble des sommets de type $V1$ d'une solution optimale forme un 2-packing. Un 2-packing est un ensemble $S \subseteq V$ tel que $\forall x, y \in S$, $N[x] \cap N[y] \neq \emptyset$ [13].
Pour la QTRD, l'ensemble des sommets de type $V1$ n'est pas forcémemnt un 2-packing car il est possible de choisir des sommets de type $V1$ voisins entre eux car un sommet de type $V1$ n'est pas utilisé uniquement pour dominer des sommets isolé comme pour la domination romaine, mais peut aussi servir pour remplir la propriété de QTRD d'un sommet de type $V2$.

On veut donc énumérer toutes les extensions possibles partant d'un sommet de type $V2$ (dont la propriété peut être déjà vérifiée ou dont il nous faut la vérifier) finissant par un autre sommet de type $V2$. Pour pouvoir ce faire, il suffit de connaître le nombre maximum de sommets de type $V1$ qui pourront se trouver entre les 2 sommets de type $V2$.

Propriété 1. La solution minimale de domination romaine quasi totale dans le cas d'un graphe qui forme un chemin élémentaire (dont on ne parcours pas les sommets qu'une fois dans le chemin) de taille au moins 3 est le nombre de sommets du graphe.

Preuve : On peut justifier cela car le nombre de voisins maximum de chaque sommet du graphe est de 2 de par la définition d'un chemin élémentaire. Le choix d'un sommet de type $V2$ est donc équivalant à des sommets de type $V1$ dans ce graphe (sauf pour les extrémités car le nombre de voisin est de 1) car un sommet de type $V2$ dominera 3 sommets (dont lui-même) pour un coût de 2. Cependant, comme il faut valider la propriété de QTRD, il faut absolument prendre un voisin avec une fin plus tard que ce sommet dans l'ensemble $V1$ ou $V2$. Si on prend le sommet dans l'ensemble $V1$ alors on dominera 3 sommets pour un coût de 3 ce qui est équivalent prendre que des $V1$. Si on prend le voisin dans l'ensemble $V2$ alors on dominera 4 sommets pour un coût de 4 et là encore c'est équivalent à prendre que des sommets de $V1$.
La valeur $\gamma_{QTRD}(G)$ pour un graphe $G = (V, E)$ formant un chemin élémentaire est donc $|V|$. □

Théorème 1. Si la propriété de QTRD est vraie pour un certain v, il ne peut avoir que 2 sommets de $V1$ entre v et le prochain sommet de $V2$ v'.

Preuve : Le cas où le maximum de sommet de type $V1$ consécutif est une solution de poids minimum, c'est quand les sommets non dominés ont un maximum de 2 autres sommets voisins non dominés. Comme on considère uniquement les graphes connexes, alors ce cas est un chemin de sommets tel que chaque sommet du chemin en voisine deux autres du chemin. Nous savons, d'après la propriété 1 ci-dessus, que la solution minimale de QTRD pour un chemin élémentaire de taille au moins 3 est le nombre de sommet du chemin. On peut en conclure que l'on peut toujours trouver une solution minimale avec moins de 3 sommets de type $V1$ entre les sommets de type $V2$. Or pour un chemin de taille 2, il est impossible d'avoir une solution dominante de poids minimum utilisant un sommet de type $V2$ car il faudra valider la propriété de QTRD pour ce sommet et donc avoir une solution avec un coût de 3 au lieu de 2 en prenant que des sommets de type $V1$.
On peut donc en conclure qu'il peut y avoir jusqu'à 2 sommets de type $V1$ entre 2 sommets de type $V2$ dans une solution dominante de poids minimum pour QTRD.
□

Il nous faut maintenant trouver le nombre maximum de sommets entre 2 sommets de type $V2$ quand la propriété du premier sommet de type $V2$ v n'est pas validée. Quand la propriété de QTRD n'est pas vérifiée, il nous faut donc prendre un voisin u de v pour valider la propriété, or, si on prend le voisin u qui va le plus loin, comme le graphe est connexe, si u n'est pas le dernier sommet du graphe, alors u voisin forcément au moins un sommet non dominé donc soit on prend u dans $V2$, et u dominera au moins le premier sommet non dominé, soit on le prend dans $V1$ si c'est la fin du graphe ou s'il existe un sommet qui pourrait dominer tous les sommets non dominés qu'u peut dominer.
On peut ainsi conclure qu'il ne peut y avoir qu'un seul sommet de type $V1$ entre deux sommet de type $V2$ quand la propriété de QTRD du premier sommet du type $V2$ n'est pas encore validé.

On représente sur la figure 3.5 les différents cas d'une extension d'une sous-solution de QTRD que nous avons définis auparavant.

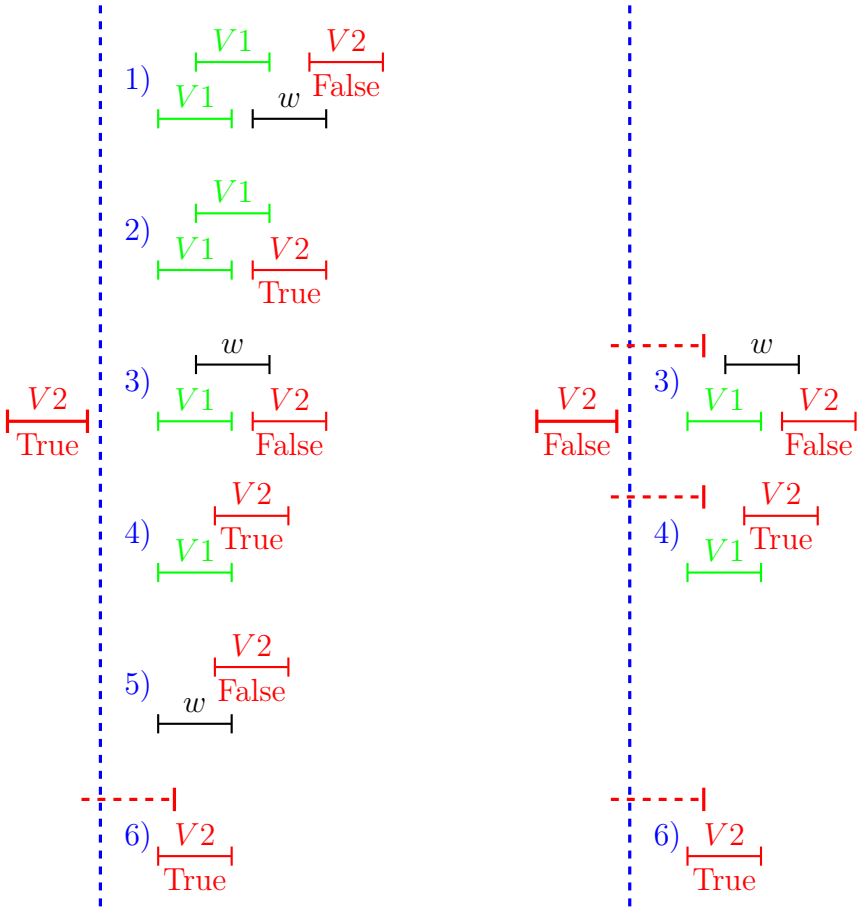

FIGURE 3.5 – Les différents cas d'une extension d'une sous-solution de domination romaine quasi totale pour les graphes d'intervalles

Maintenant que les cas sont définis on écrit des procédures qui créent des extensions pour une sous-solution de QTRD en fonction de ces derniers.

Dans le cadre de la programmation dynamique, on conserve les résultats des sous-solutions dans un tableau *opt* qui est un tableau indexé par le numéro du dernier sommet pris en considération et un booléen. $opt[i, True]$ contient donc la taille d'une solution où les i premiers intervalles sont dominés et le sommet i est un sommet de type $V2$ dont la propriété de QTRD est vérifiée. $opt[i, False]$ quand la propriété n'est pas vérifiée.

Procedure Solution1-2(v)

Data: Un sommet v d'un graphe G=(V,E) ayant déjà une sous-solution (V_1^v, V_2^v).
Result: Une extension de (V_1^v, V_2^v) selon le 1^{er} et 2^{eme} cas à partir d'une sous-solution.

$i_1 \longleftarrow \min(droite(x)), x \in V, droite(v) < gauche(x)$
if $i_1 \neq Nil$ **then**
  $i'_1 \longleftarrow \min(droite(x)), x \in V \setminus \{i_1\}, droite(v) < gauche(x)$
  if $i'_1 \neq Nil$ **then**
    $w \longleftarrow \min(droite(x)), x \in V \setminus \{i_1, i'_1\}, droite(v) < gauche(x)$
    if $w \neq Nil$ **then**
      $i_2 \longleftarrow \max(droite(x)), x \in N[w]$
      if $i_2 \notin N[i_1]$ **then**
        if $i_2 \notin N[i'_1]$ **then**
          $opt[i_2, False] \longleftarrow \min(opt[i_2, False], opt[v, prop] + 4)$
        else
          $opt[i_2, True] \longleftarrow \min(opt[i_2, True], opt[v, prop] + 4)$
      $i_{22} \longleftarrow \max(droite(x)), x \in N[i'_1]$
      if $i_{22} \neq i'_1$ **and** $i_{22} \notin N[i_1]$ **then**
        $opt[i_{22}, True] \longleftarrow \min(opt[i_{22}, True], opt[v, prop] + 4)$
    else
      $n \longleftarrow \max(droite(x)), x \in V$
      $opt[n, True] \longleftarrow \min(opt[n, True], opt[v, prop] + 2)$

On considère v comme le sommet où s'arrête la sous-solution la plus grande.

Sur cette première procédure on teste les cas 1 et 2. Pour ce faire on définit i_1 qui est le premier sommet non dominé que l'on rajoute dans l'ensemble $V1$, i'_1 le deuxième sommet non dominé que l'on rajoute également dans l'ensemble $V1$, w le troisième sommet non dominé dans le cas 1 ou i_{22} le plus grand voisin de i'_1 que l'on rajoute dans l'ensemble $V2$ pour le 2^{eme} cas s'il est voisin au troisième sommet non dominé, et, i_2 le plus grand voisin de w (cas 1 uniquement) que l'on rajoute à l'ensemble $V2$.
On forme donc les solutions 1 et 2 avec ses différents sommets ci-dessus et on garde les plus petites solutions.

Procedure Solution3-4(v,prop)

Data: Un sommet v d'un graphe G=(V,E) ayant déjà une sous-solution (V_1^v,V_2^v), et *prop* un booléen représentant la véracité de la propriété de QTRD dans cette sous-solution.

Result: Une extension de (V_1^v,V_2^v) selon le 3^{eme} et 4^{eme} cas à partir d'une sous-solution.

```
if prop then
    i1 ⟵ min(droite(x)), x ∈ V, droite(v) < gauche(x)
else
    if max(droite(x)), x ∈ N[v] ≠ v then i1 ⟵ max(droite(x)), x ∈ N[v]
    else i1 ⟵ Nil
if i1 ≠ Nil then
    if prop then
        w ⟵ min(droite(x)), x ∈ V \ i1, droite(v) < gauche(x)
    else
        w ⟵ min(droite(x)), x ∈ V, droite(v) < gauche(x)
    if w ≠ Nil then
        i2 ⟵ max(droite(x)), x ∈ N[w]
        if i2 ∉ N[i1] then
            opt[i2, False] ⟵ min(opt[i2, False], opt[v, prop] + 3)
        else
            opt[i2, True] ⟵ min(opt[i2, True], opt[v, prop] + 3)
        i22 ⟵ max(droite(x)), x ∈ N[i1]
        if i22 ≠ i1 and i22 ∈ N[w] then
            opt[i22, True] ⟵ min(opt[i22, True], opt[v, prop] + 3)
    else
        n ⟵ max(droite(x)), x ∈ V
        opt[n, True] ⟵ min(opt[n, True], opt[v, prop] + 1)
```

On considère v comme le sommet où s'arrête la sous-solution la plus grande.

Ici on teste les cas 3 et 4 pour à la fois la propriété validée et non validée. De la même manière que pour la première procédure, i_1 est le premier sommet non dominé que l'on rajoute dans l'ensemble $V1$, w le deuxième sommet non dominé dans le cas 3 ou i_{22} le plus grand voisin de i_1 que l'on rajoute dans l'ensemble $V2$ pour le 4^{eme} cas s'il est voisin au deuxième sommet non dominé, et, i_2 le plus grand voisin de w (cas 3 uniquement) que l'on rajoute à l'ensemble $V2$.

On forme donc les solutions 3 et 4 quand la propriété de QTRD est vérifiée avec ses différents sommets ci-dessus et on garde les plus petites solutions. Quand la propriété de QTRD n'est pas validée, on doit prendre le sommet de type $V1$ i_1 de sorte à ce qu'il soit le plus grand voisin de v et le w et i_{22} deviennent donc le premier sommet non dominé et i_2 reste le plus grand voisin de w.

Procedure Solution5-6(v,prop)

Data: Un sommet v d'un graphe G=(V,E) ayant déjà une sous-solution (V_1^v,V_2^v), et *prop* un booléen représentant la véracité de la propriété de QTRD dans cette sous-solution.

Result: Une extension de (V_1^v,V_2^v) selon le 5^{eme} et 6^{eme} cas à partir d'une sous-solution.

```
if prop then
    w ⟵ min(droite(x)), x ∈ V, droite(v) < gauche(x)
    if w ≠ Nil then
        i2 ⟵ max(droite(x)), x ∈ N[w]
        if i2 ∉ N[v] then
            opt[i2, False] ⟵ min(opt[i2, False], opt[v, prop] + 2)
        else
            opt[i2, True] ⟵ min(opt[i2, True], opt[v, prop] + 2)
        i22 ⟵ max(droite(x)), x ∈ N[v]
        if i22 ≠ v and i22 ∈ N[w] then
            opt[i22, True] ⟵ min(opt[i22, True], opt[v, prop] + 2)
    else
        n ⟵ max(droite(x)), x ∈ V
        opt[n, True] ⟵ min(opt[n, True], opt[v, prop])
else
    i2 ⟵ max(droite(x)), x ∈ N[v]
    opt[i2, True] ⟵ min(opt[i2, True], opt[v, prop] + 2)
```

On considère v comme le sommet où s'arrête la sous-solution la plus grande.

Ici on teste les cas 5 et 6, ce dernier cas à la fois la propriété validée et non validée. On définit donc w premier sommet non dominé dans le cas 5 ou i_{22} le plus grand voisin de v que l'on rajoute dans l'ensemble $V2$ pour le 6^{eme} cas que la propriété de QTRD soit validée ou non si i_{22} est voisin au premier sommet non dominé, et, i_2 le plus grand voisin de w (cas 5 uniquement) que l'on rajoute à l'ensemble $V2$.
On forme donc les solutions 5 et 6 avec ses différents sommets ci-dessus et on garde les plus petites solution. Quand la propriété de QTRD n'est pas validée, on doit prendre le sommet de type $V1$ i_1 de sorte à ce qu'il soit le plus grand voisin de v et le w et i_{22} deviennent donc le premier sommet non dominé et i_2 reste le plus grand voisin de w.

Algorithm 7: QTRD-Interval(G)

Data: Un Graphe d'intervalles G=(V,E) construit selon un modèle normalisé.
Result: Une valeur minimale de QTRD dans G $\gamma_{QTRD}(G)$.

$maxValue \longleftarrow 2 * |V|$
$initialValue \longleftarrow (-1, -1)$
$opt[initialValue, True] \longleftarrow 0$
for $v \in V$ **do**
    $opt[v, True] \longleftarrow maxValue$
    $opt[v, False] \longleftarrow maxValue$
Solution1-2($initialValue$)
Solution3-4($initialValue, True$)
Solution5-6($initialValue, True$)
$Trierlegraphepardatedefincroissante$ **for** $v \in V$ **do**
    if $opt[v, True] \neq maxValue$ **then**
        Solution1-2(v)
        Solution3-4($v, True$)
        Solution5-6($v, True$)
    if $opt[v, False] \neq maxValue$ **then**
        Solution3-4($v, False$)
        Solution5-6($v, False$)
$n \longleftarrow \max(droite(x)), x \in V$
return $\gamma_{QTRD}(G) = opt[n, True]$

L'algorithme ci-dessus utilise les 3 procédures précédemment créées afin de calculer la valeur d'une solution minimale pour un graphe d'intervalles $G = (V, E)$. On initialise la valeur de *opt* au double du nombre de sommets car c'est la pire solution où l'on considère tous les sommets de V dans l'ensemble $V2$. Pour le cas 0 $True$ car si le graphe n'existe pas la taille d'une solution minimale est 0.

Ensuite, on parcourt tous les sommets, et, si leur valeur dans *opt* a été changée pour True ou False alors on applique les procédures concernée en précisant pour certaines le booléen.

Enfin, on renvoie $\gamma_{QTRD}(G)$ qui est $opt[n, True]$ avec n le nombre de sommets du graphe.

Démonstration de l'algorithme : Comme l'algorithme teste tous les cas d'une extension d'une sous-solution de domination romaine quasi totale et que nous avons prouvé plus haut qu'il n'en existait pas d'autre. Par conséquent l'algorithme calcule toujours une solution de domination romaine quasi totale de poids minimum. □

Conclusion

C'est au sein de l'équipe GAMoC du LIFO que j'ai effectué ce stage enrichissant dans la recherche en informatique qui m'a permis de découvrir le monde du travail en informatique ainsi que d'utiliser et développer mes connaissances acquises pendant mes trois années de licence. Lors de ce stage, j'ai été amené à effectuer plusieurs tâches qui m'ont permis d'apprendre de nombreuses choses. En effet, le développement d'un algorithme par programmation dynamique résolvant la domination romaine quasi totale pour les graphes d'intervalles m'a appris à chercher des connaissances dans divers articles afin de les utiliser pour créer mes algorithmes. Cela a au passage enrichi mes connaissances dans le domaine des graphes ainsi que leurs problèmes. Enfin, ma manière d'appréhender des problèmes complexes s'est aussi développée.
Cependant, je n'ai pas toujours réussi ce que j'ai entrepris au cours de ce stage. En effet, avant d'aboutir à un algorithme fonctionnel, beaucoup d'algorithmes ont été abandonné/recommencé. Cela m'a permis de progresser afin de comprendre de mes erreurs.

En tant que stagiaire dans l'équipe GAMoC du LIFO, j'ai assisté aux divers séminaires ayant eu lieu pendant mon stage. Ces séminaires ont favorisé le développement de ma culture scientifique en informatique fondamentale ainsi que de découvrir certains thèmes parmi les nombreux thèmes de recherche possible en informatique.

Pendant ce stage, j'ai pu accomplir de nombreuses choses. Durant ces trois mois, j'ai conçu des algorithmes efficaces pour résoudre le problème de domination ainsi qu'un algorithme utilisant la programmation dynamique pour résoudre la domination romaine quasi totale. Cependant, il reste encore beaucoup de travail. En effet, il y a encore des variantes de la domination romaine à résoudre comme la domination romaine totale, la domination romaine indépendante ainsi que la domination romaine signée. De plus, il est possible de résoudre ces problèmes pour d'autres classes de graphes ce qui modifie la manière dont le problème peut être résolu.

Au final, ce stage m'a beaucoup enrichi personnellement que ce soit humainement ou en rapport avec mes connaissances en informatique. J'ai apprécié ces trois moi passé au sein de l'équipe GAMoC et je serai ravi de continuer à travailler dans cette équipe.

Annexes

Algorithme QTRD-Interval : Analyse du temps d'exécution

Les procédures :

Les 3 procédures utilisées par l'algorithme ne possèdent pas de boucles et sont seulement composées de conditions et d'opérations. Ces procédures servent à construire les différentes extensions d'une sous-solution utilisée par l'algorithme en sélectionnant les différents sommets des ensembles dominants. On peut choisir ces sommets de 2 manières différentes :

- En prenant le plus grand voisin (le voisin avec la fin la plus grande) d'un sommet v ce qui est effectué actuellement en temps $O(n)$ avec un maximum sur l'ensemble des voisins du sommet v. On peut réduire ce temps en créant au préalable un tableau regroupant l'ensemble contenant le plus grand voisin de chaque sommet du graphe, ainsi, on pourrait rechercher le plus grand voisin d'un sommet en temps constant.
- En prenant soit le premier, deuxième où troisième sommet non dominé ce qui est effectué actuellement en temps $O(n)$ en parcourant tous les sommets et avec des opérations de maximum et minimum. On peut réduire ce temps d'exécution en construisant une matrice de taille $3 * 2n$ contenants les 3 premiers intervalles non dominés à partir de l'indice i. Ainsi, nous pourrons récupérer les trois premiers sommets non dominés à une certaine étape en temps constant.

Pour créer ce tableau et cette matrice, on peut se baser sur l'article de Mathieu Liedloff intitulé *Efficient algorithms for Roman domination on some classes of graphs* dans lequel il existe déjà une implémentation de ces tableaux en temps linéaire. Pour pouvoir les utiliser il faudra donc l'ancienne manière de récupérer les sommets sélectionnés par une recherche en temps constant dans le tableau.

Dans l'algorithme principal, il y a 2 boucles for entourées d'opérations simples ($O(1)$).
La première boucle exécute n fois (n correspond au nombre d'intervalles) une affectation de valeur dans une matrice de taille $2n$. Cette boucle est donc exécutable en temps linéaire $O(n)$.

La deuxième boucle s'exécute n fois aussi et effectue au plus 5 appels à une procédure exécutable en temps $O(1)$.
On a donc un temps d'exécution de $O(5n)$ pour cette boucle. Le temps d'exécution de l'algorithme est donc majoré par cette deuxième boucle for.
Comme on fait abstraction de la constante linéaire, on a donc, comme temps d'exécution pour l'algorithme $O(n)$ qui est un temps d'exécution linéaire.

État de l'art de la domination romaine quasi-totale

Il existe plusieurs articles étudiant ce problème sur diverses classes de graphes. Dans 2 articles de Abel Cabrera Martínez et d'autres co-auteurs, ils ont étudié la valeur d'une solution minimale de QTRD mais n'ont produit aucun algorithme ne renvoyant une solution minimale.
Article 1 : *On the quasi-total Roman Domination Number of graphs (mathematics).*
Article 2 : *Quasi-total Roman domination in graphs (mathematics).*

Dans l'article *Algorithmic aspects of quasi-total Roman domination in graphs* par Vikas Mangal, il y a des résultats de NP-complétude et de programmation linéaire pour certaines classes de graphes mais pas pour les graphes d'intervalles.

On peut donc en conclure que, dans la recherche actuelle, il n'existe pas d'article public proposant un algorithme efficace (linéaire) résolvant le problème de la domination romaine quasi totale pour les graphes d'intervalles.

État de l'art de la domination romaine totale

On peut trouver de nombreux articles en lien avec la domination romaine totale (TRD) sur internet. Parmi les différents résultats il n'existe pas d'algorithme résolvant le problème pour les graphes d'intervalles. Voici ci-dessous une liste (non-exhaustive) des différents travaux porté sur la domination romaine totale :

— « Total Roman Domination In Special Type Interval Graph » par M. Reddappa : Etude de TRD sur un certains type de graphes d'intervalles (pas d'algorithme, seulement des propriétés).
— « On the Total Roman domination in Trees » par Jafar Amjadi : Caractérisation de TRD sur les arbres.
— « Total Roman Domination in graphs » par Hossein Abdollahzadeh Ahangar : Relation entre TRD et RD.
— « Total Roman domination edge-critical graphs » par Chloe Lampman : Résultats basique de TRD pour les graphes *edge-critical*.
— « Total Roman Domination in direct product graphs » par Abel Cabrera Martínez : $\gamma_{TR}(GXH)$ étudié (GXH représente le produit du graphe G et du graphe H).
— « Total Roman Domination Number of rooted product graphs » par Abel Cabrera Martínez : Étude de la valeur de TRD pour les graphes de produits enracinés.
— « Total Roman Domination in the lexicographic product of graphs » par Nicolàs Campanelli : Caractérisation de TRD pour les produits de graphes lexicographique.
— « Total Roman Domination in digraphs » par Xinhong Zhang : Caractérisation de TRD sur les digraphs.
— « On Total Roman Domination in graphs » par P. Roushini Leely Pushpam : Étude de la valeur de TRD pour les graphes en général.

Liste (non-exhaustive) des variantes de la domination romaine totale

— Domination romaine totale **globale** : Une solution de domination romaine totale pour un graphe G telle que c'est aussi une solution pour son graphe complémentaire $\bar{G}$.
Voir l'article « Global total Roman domination in graphs » de J. Amjadi.

— **k-**Domination romaine totale **signée** : Un mixte entre la domination romaine totale et signée. Il faut former une solution de domination romaine totale avec quelques contraintes en plus. En effet, chaque sommet non dominant (de type V0) ne valent plus 0 mais -1 et chaque sommet de type V2 doit avoir, en calculant la sommet des valeurs des sommets de son entourage ouvert, une valeur au moins égale à k.
Voir l'article « Complexity of signed total k-Roman domination problem in graphs » de Saeed Kosari.

— Domination romaine totale **non signée négativement** : Pareil que la k-domination romaine totale signée à l'exception que le seuil k est remplacé par 0.
Voir l'article « Nonnegative signed total Roman domination in graphs » de Nasrin Dehgardi.

— **Hop** total roman domination : Une solution de domination totale tel que chaque sommet u dans V0 doit avoir un voisin v dans V2 à une distance de 2 de lui.
Voir l'article « Hop total Roman domination in graphs » de H. Abdollahzadeh Ahangar.

Extensions d'une sous-solution de domination romaine totale

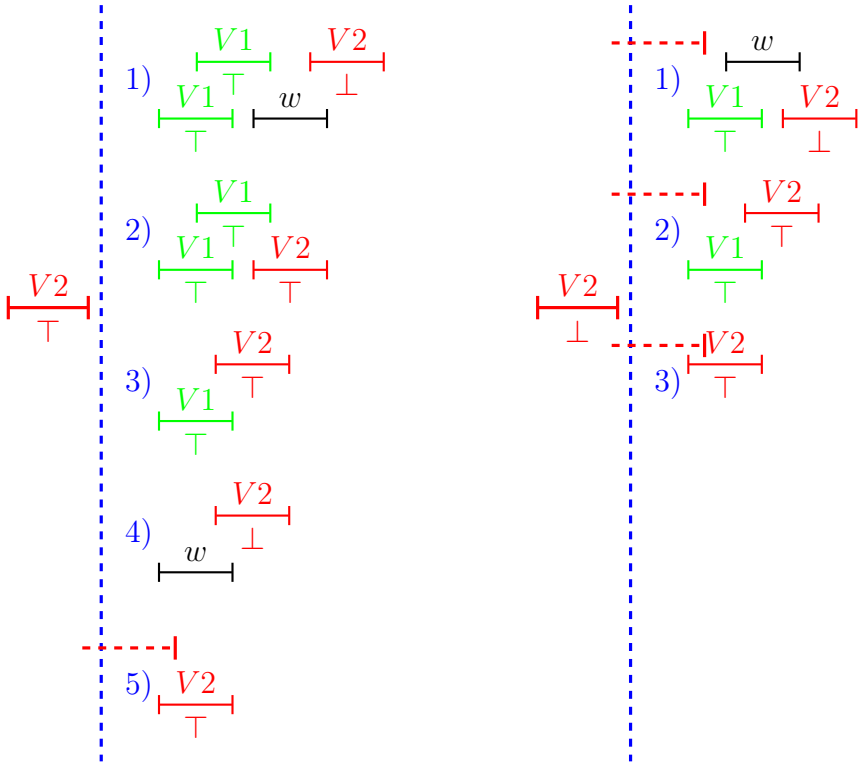

FIGURE 3.6 – Les différents cas d'une extension d'une sous-solution de domination romaine totale pour les graphes d'intervalles

Sur le schéma ci-dessus 3.6, on représente ici une adaptation des cas de l'algorithme dynamique résolvant la domination romaine quasi totale pour le problème de la domination romaine totale.
Le choix des sommets pour les cas varie donc quelques peu. En effet, dans la domination romaine totale est un problème très similaire à la domination romaine quasi totale mais avec une contrainte en plus : Il faut que non seulement les sommets de type V2 ne soit pas isolé dans le sous-graphe induit de la solutions $(V1 \cup V2)$ mais aussi les sommets de type V1. Ceci va donc particulièrement impacter le choix des V1 car ils seront donc forcément voisins avec un autre sommet de l'ensemble dominant.

On peut remarquer sur les différents cas que les sommets de type V1 ont maintenant eux aussi l'état de leur propriété indiquéé en dessous de leur intervalles (indiquée par un $\top$).

On peut constater que la domination romaine totale est plus restrictive que QTRD. En effet, il n'existe pas de nouvelles combinaisons de sommets valide pour TRD qui ne sont pas valide pour QTRD. De ce fait, on peut donc en partant des cas d'une extensions d'une sous-solution de QTRD, enlever les cas qui ne sont plus valide pour TRD et nous obtenons ainsi les différents cas de la figure 3.6.

Bibliographie

[1] H. de Ridder et al. – « Information system on graph classes and their inclusions (isgci) », 2001-2014, https://www.graphclasses.org.

[2] A. Brunel – « Les graphes parfaits », Thèse, École Normale Supérieure de Lyon, 2009.

[3] E. J. Cockayne, P. A. Dreyer, S. M. Hedetniemi & S. T. Hedetniemi – « Roman domination in graphs », *Discrete Mathematics* **278** (2004), no. 1, p. 11–22.

[4] F. Gardi – « The roberts characterization of proper and unit interval graphs », *Discrete Mathematics* **307** (2007), no. 22, p. 2906–2908.

[5] M. R. Garey & D. S. Johnson – *Computers and intractability : A guide to the theory of np-completeness (series of books in the mathematical sciences)*, first edition éd., W. H. Freeman, 1979.

[6] S. C. Gar´ia, A. C. Mart´inez & I. G. Yero – « Quasi-total roman domination in graphs », *Mathematics* (2019), p. 173.

[7] W. T. T. J. . F. Harary – « On double and multiple interval graphs », *Journal of Graph Theory* (1979), p. 205–211.

[8] A. Hertz – « Quelques classes de graphes », https://www.gerad.ca/~alainh/Quelques-Classes.pdf.

[9] D. E. B. . A. H. B. . G. Isaak – « Structure of bipartite probe interval graphs », 2009.

[10] J. Kleinberg & E. Tardos – *Algorithm design*, Addison Wesley, 2006.

[11] R. Letourneur – « Algorithme exacts et exponentiels pour des problèmes de graphes », Thèse, Université d'Orléans, https://theses.univ-orleans.fr/public/2015ORLE2022_va.pdf, 2015.

[12] M. Liedloff – « Algorithme exacts et exponentiels pour les problèmes np-difficiles : domination, variantes et généralisations », Thèse, Université Paul Verlaine - Metz, http://docnum.univ-lorraine.fr/public/UPV-M/Theses/2007/Liedloff.Mathieu.SMZ0727.pdf, 2007.

[13] M. Liedloff, T. Kloks, J. Liu & S.-L. Peng – « Efficient algorithms for roman domination on some classes of graphs », *Discrete Applied Mathematics* **156** (2008), no. 18, p. 3400–3415.

[14] V. Mangal & P. V. S. Reddy – « Algorithmic aspects of quasi-total roman domination in graphs », *Communication in Combinatorics and Optimization (CCO)* (2022), p. 93–104.

[15] A. C. Martínez, J. C. Hernández-Gómez & J. M. Sigarreta – « On the quasi-total roman domination number of graphs », *Mathematics* (2021), p. 2823.

[16] R. Motwani – « Introduction to automata and complexity theory », Tech. report, Stanford University, https://theory.stanford.edu/~rajeev/CS154/solution7.pdf, 2009.

[17] J. Rani & P. Mor – « Domination in graph and some of its applications in various fields », *JETIR* (2019), p. 322–326.

[18] C. Robin – « Classes héréditaires de graphes : De la structure vers la coloration », Thèse, Université Grenoble Alpes, https://theses.hal.science/tel-03587403/document, 2021.

[19] T. Roughgarden, K. Kollias, A. Nguyen, J. TibShirami & S. Choi – « Guide to dynamic programming », 2013.

[20] T. Roughgarden, A. Sharp & T. Wexler – « Guide to greedy algorithms », 2013.