Rapport de TER

Algorithmes pour le problème de domination d'un graphe

Gaspers Serge
(serggasp@yahoo.fr)

Année académique 2003-2004

Tuteur : D. Kratsch
Maîtrise Informatique, Université de Metz

Remerciements

Je tiens à remercier avant tout mon tuteur M. Kratsch, qui m'a guidé et inspiré tout au long de ce TER, pour sa disponibilité et son temps précieux qu'il m'a consacré.

Mes remerciements vont aussi à mes camarades de classe pour la bonne ambiance tout au long de cette année. Pour la même raison, je dis merci à tout mon entourage proche, mes amis, ma copine et ma famille.

Je voudrais aussi remercier M. Brendan McKay pour son programme `nauty` *que j'ai utilisé dans ce TER, ainsi que l'équipe autour de Stephen North qui a développé le paquetage* `Graphviz`*, un ensemble de programmes pour la visualisation de graphes, qui a été utilisé pour tracer les graphes de ce rapport.*

Finalement, j'aimerais remercier le jury pour la lecture et l'évaluation de ce rapport et pour suivre ma présentation qui aura lieu le 27 mai 2004.

Table des matières

1 Introduction

Beaucoup de problèmes NP-complets et NP-difficiles sont des problèmes de graphes. Un des problèmes NP-difficiles les mieux étudiés dans la littérature est DOMINATING SET qui est aussi le sujet de ce TER. Cette partie introductive va d'abord donner les définitions et notions de base qui seront utilisées dans ce rapport et ensuite donner quelques motivations théoriques et pratiques qui justifient le traitement d'un tel problème. La deuxième partie va présenter deux algorithmes exacts (exponentiels) qui sont capables de résoudre ce problème exactement sur des petites entrées. La troisième partie va proposer des méthodes pour attaquer des grands graphes si la solution ne doit pas forcément être optimale, mais proche d'un optimum. Finalement, nous allons voir comment combiner des algorithmes exacts avec des heuristiques et des méthodes d'approximation dans la dernière partie.

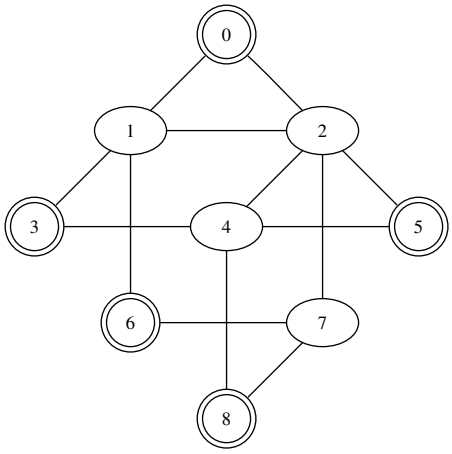
1.1 Définition du problème

Soit $G = (V, E)$ un graphe non-orienté où V représente l'ensemble des n sommets de G et E l'ensemble de ses m arêtes. On dit qu'un sommet $u \in V$ *domine* un sommet $v \in V$ si et seulement si $u = v$ ou v est un voisin de u. Le problème de domination d'un graphe consiste à dominer tous les sommets du graphe avec un nombre minimal de sommets. Ainsi, un sous-ensemble $D \subseteq V$ est un *ensemble dominant* du graphe G si chaque sommet de G est soit dans D, soit adjacent à un sommet de D. En d'autres termes, tous les sommets $v \in V - D$ doivent avoir un voisin $u \in D$.

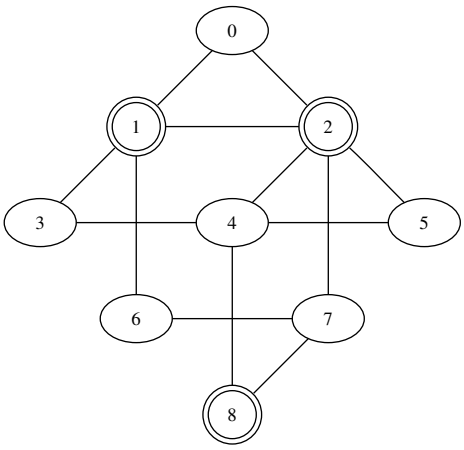
$$\forall v \in V - D, \exists u \in D \mid \{u, v\} \in E$$

Le problème MINIMUM DOMINATING SET (MDS) consiste à trouver un ensemble dominant *de taille minimale*. Par convention [1, 2], notons $\gamma(G)$ la taille minimale d'un ensemble dominant (*domination number*) pour le graphe G.

Deux exemples d'ensembles dominants pour un même graphe sont donnés à la figure 1.1. En effet, (b) montre un ensemble dominant minimum (MDS), car il n'existe aucun sous-ensemble de sommets de taille 2 qui domine tous les sommets du graphe. MDS est à ne pas confondre avec la recherche d'un ensemble dominant minimal, dont un exemple est donné en (a). On dit qu'un sous-ensemble D de sommets d'un graphe G est un *ensemble dominant minimal* si aucun sommet ne peut lui être enlevé sans rompre la dominance. La taille maximale d'un ensemble dominant minimal pour le graphe G est notée $\Gamma(G)$.



(a) Ensemble dominant minimal



(b) Ensemble dominant minimum

FIG. 1.1: Exemples d'ensembles dominants

1.2 Complexité du problème

On peut facilement associer un problème de décision au problème de trouver un ensemble dominant de taille minimale :

Pour un entier naturel $k \leq n$ et un graphe $G = (V, E)$ donnés, est-ce qu'il existe un ensemble dominant de taille k ?

Ce problème est *NP-complet* et donc, le *problème d'optimisation*[1] de trouver un ensemble dominant minimum est *NP-difficile*. Intuitivement, le problème d'optimisation associé est au moins aussi difficile que le problème de décision.

Etant confronté à un problème NP-complet ou NP-difficile, il est parfois indispensable de le "résoudre" quand même. Evidemment, on ne peut pas espérer trouver une solution exacte pour des entrées "trop grandes" en "temps raisonnable". Si on cherchait un algorithme polynomial pour MDS, cela reviendrait à démontrer que $P = NP$, ce qui est une des questions les plus importantes de l'informatique théorique de nos jours. Vu que la plupart des chercheurs du domaine croient qu'il n'existe pas d'algorithme polynomial pour les problèmes NP-complets (donc que $P \neq NP$), il est improbable de trouver un algorithme polynomial capable de résoudre MDS exactement pour toutes ses entrées.

Pour "résoudre" MDS, on doit alors relaxer une ou plusieurs conditions :

– *taille de l'entrée* : on peut implémenter des algorithmes exponentiels exacts pour des entrées relativement petites. Pour des entrées trop grandes, ils mettront un temps beaucoup trop important pour un usage pratique. La partie 2 de ce rapport va présenter deux algorithmes exacts.

[1] Plus précisément, ce problème est un problème de minimisation.

- *qualité de la solution* : au lieu de chercher une solution exacte, il est parfois suffisant de donner une solution approchée ou pas trop mauvaise. La partie 3 de ce rapport va présenter divers algorithmes polynomiaux basés sur des heuristiques pour trouver un ensemble dominant "pas trop grand", comparé au plus petit ensemble dominant.
- *entrées particulières* : MDS peut être résolu en temps polynomial pour certaines classes de graphes particulières, par exemple pour les graphes de permutation et les graphes d'intervalles [1]. Cependant, le problème reste NP-difficile pour d'autres classes de graphes, comme les graphes planaires, par exemple. Dans ce TER, les entrées particulières ne sont pas prises en compte ; on étudie seulement des algorithmes qui marchent pour n'importe quel graphe.

D'autres méthodes souvent utilisées pour attaquer un problème NP-difficile sont les algorithmes randomisés et les algorithmes paramétrisés qui n'ont pas été traités dans ce TER, car ils ne sont pas applicables à ce problème.

1.3 Applications

1.3.1 Accessibilité de ressources

Imaginons que dans un réseau d'ordinateurs, on veuille installer un logiciel sur un sous-ensemble d'ordinateurs tel que chaque poste du réseau a accès à ce logiciel, soit parce qu'il en détient une copie, soit parce qu'il a accès à une copie sur un poste voisin à travers une connexion directe dans le réseau (pour ne pas trop encombrer le réseau). Pour ne pas perdre trop d'espace disque et pour faciliter la maintenance, on souhaite que le nombre de copies soit minimisé. Ce problème peut être réduit au problème de trouver un ensemble dominant minimum en modélisant les postes par les sommets du graphe et les connexions directes dans le réseau par les arêtes du graphe. Un ensemble dominant minimum du graphe sera donc l'ensemble des postes où il faut installer le logiciel.

En général, beaucoup de problèmes de placement de ressources sont des applications de MDS.

1.3.2 Autres

Minimum Dominating Set connaît d'autres applications, surtout dans le domaine des réseaux d'ordinateurs et des sciences sociales.

Pour le routage dans les réseaux mobiles ad hoc, par exemple, un ensemble dominant *connexe* de taille minimale doit pouvoir être calculé rapidement. Cet ensemble dominant constitue "l'épine dorsale" (backbone) virtuelle de ce réseau de communication sans infrastructure fixe. Ce backbone est important pour les protocoles de routage, mais la nature dynamique (les membres de ce réseau peuvent se déplacer aléatoirement) du réseau fait de ce problème un vrai défi.

Etant un problème fondamental, la recherche d'un ensemble dominant minimum connaît beaucoup de variantes [2], comme par exemple la recherche d'un ensemble dominant stable (le graphe induit par l'ensemble dominant contient que des sommets isolés), connexe (le graphe induit par l'ensemble dominant est connexe) ou total (le graphe induit par l'ensemble dominant ne contient pas de sommets isolés).

1.4 Machine de test

Vu que différents algorithmes sont comparés ici entre eux, entre autre en fonction de leur temps d'exécution, il est indispensable de donner la configuration de la machine de test, ainsi que le système d'exploitation et le langage de programmation utilisé :

Processeur :	Intel Pentium IV 2,66 GHz
Mémoire :	1 Go DDR 333 MHz
Système d'exploitation :	Linux (Redhat 9)
Langage de programmation :	C
Compilateur :	gcc 3.2.2 (utilisé avec l'option d'optimisation -O3)

TAB. 1.1: Configuration Machine de test

Chaque fois que des temps d'exécution sont présentés, il s'agit du temps d'exécution réel et non pas du temps CPU.

Le langage de programmation C a été choisi parce qu'il est très puissant (car très proche de la machine) et il ne fait pas beaucoup de contrôles implicites et cachées qui augmenteraient le temps d'exécution.

Remarque : Dans tous les algorithmes, les graphes sont représentés par un tableau de listes d'adjacence.

2 Algorithmes exacts

Cette section traite deux algorithmes exacts (exponentiels) pour le problème de trouver un ensemble dominant de taille minimale. C'est surtout une étude du temps d'exécution en pratique qui permettra de déterminer pour quelles tailles du graphe d'entrée, il est raisonnable d'implémenter un algorithme exact pour ce problème.

Exemple : Si on est sûr que le graphe d'entrée n'aura jamais plus de 30, 100 ou 1000 sommets, est-ce qu'on peut encore résoudre MDS en temps raisonnable ?

2.1 Algorithme 2^n

Une solution immédiate pour MDS est de générer tous les 2^n sous-ensembles de sommets du graphe et de vérifier s'ils constituent un ensemble dominant (DS). A la fin, on choisit un des ensembles qui constituent un DS de cardinalité minimale.

Remarquez qu'on peut décider si un sous-ensemble de sommets D est un ensemble dominant pour un graphe $G = (V, E)$ en temps linéaire, c'est-à-dire en $O(n + m)$ où $n = |V|$ et $m = |E|$:

Algorithm 1 Vérification si un sous-ensemble de sommets est un ensemble dominant

```
EST-ENS-DOM ( D : ENSEMBLE, G = (V, E) : GRAPHE ) : BOOLÉEN

   Pour chaque sommet v ∈ V
      faire couleur[v] ← BLANC
   Pour chaque sommet v ∈ D
      faire couleur[v] ← NOIR
            Pour chaque sommet u ∈ Adj[v]
               faire couleur[u] ← NOIR
   Si ∃v ∈ V tel que couleur[v] = BLANC
      alors retourner FAUX
      sinon retourner VRAI
```

L'algorithme 1 associe une couleur à chaque sommet et l'initialise à BLANC. La boucle principale colorie en NOIR tous les sommets dominés par D. Finalement, l'algorithme retourne VRAI si tous les sommets sont NOIRs, sinon il retourne FAUX.

Remarques :

1. Le test pour déterminer s'il existe un sommet BLANC dans le graphe peut être implémenté de deux façons :
   a) parcourir tous les sommets de G et retourner FAUX si on rencontre un sommet BLANC ; on retourne VRAI si à la fin du parcours de tous les sommets, on n'a pas rencontré de sommet BLANC.
   b) initialiser au début de l'algorithme un compteur (qui représente le nombre de sommets non dominés) à n et le décrémenter chaque fois qu'on colorie un sommet en NOIR qui était BLANC auparavant. Le test se résume donc à une comparaison de ce compteur à 0.
2. Pour l'algorithme de recherche d'un sous-ensemble dominant minimum, une extension du présent algorithme est utilisé :
   a) on ajoute la couleur GRIS : les sommets de l'ensemble dominant seront dorénavant NOIRs, les sommets dominés n'appartenant pas à l'ensemble dominant seront GRIS et les sommets non-dominés seront BLANCS.
   b) Cette modification permettra de plus rapidement colorier le graphe en se basant sur le coloriage précédent. Il ne sera donc plus nécessaire de parcourir tout le graphe, mais il suffit de considérer l'entourage du sommet qu'on ajoute à D ou qu'on supprime de D :
      i. A chaque fois qu'on ajoute un sommet v à l'ensemble dominant D, on colorie v en NOIR et tous ses voisins BLANCs en GRIS.
      ii. A chaque fois qu'on enlève un sommet v de l'ensemble dominant D, on colorie v en BLANC ou en GRIS, selon qu'il est adjacent à un sommet NOIR ou pas. Puis, on fait la même chose pour ses voisins.

Attaquons maintenant le problème de générer les sous-ensembles de sommets du graphe pour appliquer l'algorithme 1 à chaque sous-ensemble. Evidemment, on ne va pas générer *tous* les sous-ensembles et vérifier pour chacun si c'est un ensemble dominant pour finalement choisir le plus petit. Il est, par exemple, plus économique en termes de temps d'exécution, de commencer par analyser tous les sous-ensembles de sommets de taille k avant d'analyser les sous-ensembles de taille $k+1$. Ceci permettra à l'algorithme de s'arrêter dès qu'il a trouvé un ensemble dominant, car on est sûr qu'il est minimal (on a déjà analysé tous les ensembles de taille inférieure)[1].

Considérons l'ensemble des parties de V : $\mathcal{P}(V)$. Si on le munit d'une relation d'ordre (l'inclusion $\subseteq$), on dit que c'est un *treillis booléen* (en anglais, "boolean lattice"). Ce treillis peut facilement être partitionné en ses niveaux A_k, où A_k est l'ensemble de tous les sous-ensembles de sommets de taille k. Pour trouver un sous-ensemble dominant de taille minimale, on a deux possibilités :

[1] Si on veut que l'algorithme sorte tous les ensembles dominants de cardinalité minimale, il ne va pas s'arrêter tout de suite, mais continuer à analyser tous les sous-ensembles de la taille en cours

1. "du bas vers le haut" : commencer par analyser les sous-ensembles de taille 1, puis 2, puis 3, etc. jusqu'à ce qu'on trouve un sous-ensemble qui domine tous les sommets du graphe. On est donc sûr que le premier sous-ensemble dominant qu'on trouve est de cardinalité minimale, car on a déjà analysé tous les sous-ensembles qui sont plus petits.
2. "du haut vers le bas" : parcourir les niveaux du treillis dans l'autre sens. On analyse donc les grands sous-ensembles avant les petits. Dans ce cas, on peut s'arrêter dès qu'on trouve un niveau où aucun sous-ensemble ne domine tout le graphe. Il est clair que s'il existe un sous-ensemble dominant de taille $k < n$, alors il doit exister au moins un sous-ensemble dominant de taille $k + 1$ (il suffit d'ajouter n'importe quel sommet au sous-ensemble dominant de taille k pour avoir un sous-ensemble dominant de taille $k + 1$).

Voici une ébauche de l'algorithme implémentant la technique "du bas vers le haut" :

Algorithm 2 Algorithme exact 2^n commençant par les petits sous-ensembles

```
ALGO-EXACT-MDS ( G = (V, E) : GRAPHE ) : MDS

   Pour k = BORNEINF à BORNESUP
      faire Pour chaque sous-ensemble D ⊆ V de taille k
               faire Si Est-Ens-Dom(D, G)
                        alors retourner D
```

Notez que l'algorithme 2 termine dès qu'il a trouvé le premier ensemble dominant. Il reste à déterminer BORNEINF et BORNESUP.

Comme BORNEINF, on peut prendre 1, car on ne traite pas le cas du graphe vide. Concernant BORNESUP, on va démontrer maintenant qu'on peut prendre $n/2$. Si on note γ la cardinalité minimale d'un ensemble dominant, le théorème d'Ore [3] dit que pour tout graphe de degré minimal $\delta \geq 1$, on a $\gamma \leq n/2$. Ceci est vrai en particulier pour les graphes connexes. Dans le cas où on n'est pas en présence d'un graphe connexe, on peut facilement décomposer ce graphe G en ses composantes connexes $\mathcal{C}_1$,$\mathcal{C}_2$, ... , $\mathcal{C}_k$ et appliquer **ALGO-EXACT-MDS** sur chaque composante connexe. Un MDS pour G sera donc l'union des MDS pour les $\mathcal{C}_i$, $i = 1..k$ et la cardinalité minimale γ d'un ensemble dominant sera la somme des $\gamma(\mathcal{C}_i)$.

Dans la section 4.1 concernant la combinaison d'heuristiques avec un algorithme exact, on essaiera de raffiner d'avantage BORNEINF et BORNESUP.

En constatant qu'il existe 2^n sous-ensembles de sommets dans un graphe et que la borne supérieure pour la taille d'un MDS de $n/2$ pour les graphes connexes n'améliore pas ce nombre en notation O (il existe $O(2^n)$ sous-ensembles de taille $\leq n/2$), on voit immédiatement que le temps d'exécution pour cet algorithme est $O((n+m) \cdot 2^n)$. Généralement, on utilise une notation spéciale pour le temps d'exécution des algorithmes exponentiels

qui ne tient pas compte des facteurs polynomiaux : c'est la notation O^*. Cet algorithme s'exécute donc en temps $O^*(2^n)$.

Pour expliquer les résultats des tests de la figure 2.1, il est d'abord nécessaire de définir ce que c'est la densité d'un graphe : la *densité* d'un graphe est le nombre de ses arêtes divisé par le nombre maximal d'arêtes qu'un graphe de même taille pourrait avoir. Pour les graphes non-orientés de n sommets et de densité d ($0 \leq d \leq 1$), on a donc :

$$m = d \cdot \frac{n \cdot (n-1)}{2}$$

où m est le nombre d'arêtes du graphe.

Pour obtenir des graphes d'une densité d donnée, le générateur de graphes aléatoires crée d'abord un graphe sans arêtes et pour chaque sous-ensemble de deux sommets, il ajoute une arête entre ces deux sommets avec une probabilité d.

Les tests ont été réalisés en utilisant ce générateur de graphes pour des densités variant de 0,05 à 1 par pas de 0,05. Pour chaque densité, le temps d'exécution moyen a été déterminé sur 20 instances de graphes connexes (si le temps d'exécution dépassait une demie heure, le nombre d'instances a été limité à 5).

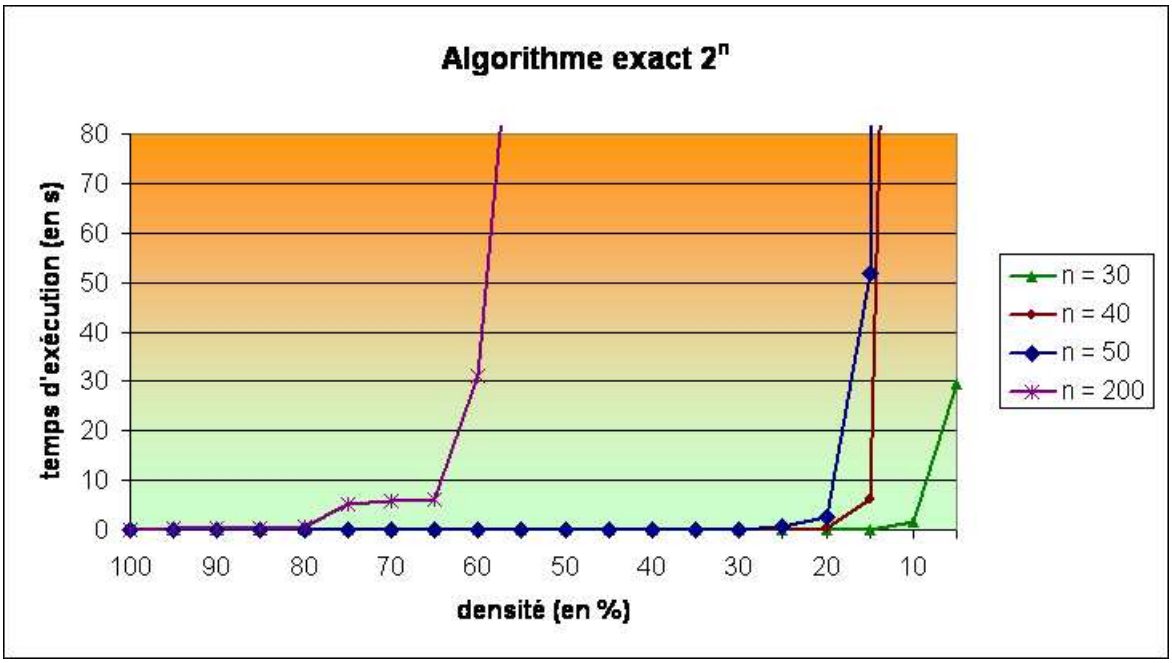


FIG. 2.1: Temps d'exécution de l'algorithme exact 2^n

Sur le graphique de la figure 2.1 ci-dessus, on voit que le temps d'exécution (exprimé en secondes) monte non seulement avec le nombre de sommets du graphe (n), mais il dépend aussi de la densité du graphe. Intuitivement, la taille d'un ensemble dominant

minimum γ est petite si le graphe est dense, et γ est grand si le graphe est creux. Vu que l'algorithme analyse d'abord les petits sous-ensembles du graphe, il termine plus tôt si γ est petit.

Un des objectifs était de déterminer le plus grand n tel que tous les graphes avec n sommets peuvent être traités par cet algorithme en temps raisonnable. On voit que pour $n = 30$, le temps d'exécution peut monter jusqu'à une demie minute au pire des cas. En considérant le tableau complémentaire 2.1 des temps d'exécutions supérieurs à 80 secondes ci-dessous, on voit qu'il y a des graphes de 40 sommets qui nécessitent un traitement de 5 heures pour trouver un MDS.

	n = 40	n = 50	n = 200
d = 55%			126,74 s
d = 50%			148,25 s
d = 10%	342,03 s	2 334,08 s	???
d = 5%	18 009,58 s	???	???

TAB. 2.1: Compléments pour le temps d'exécution de l'algorithme exact 2^n

On voit également que les graphes ayant 200 sommets et de densité au moins 0,5 peuvent être traités en temps raisonnable. Essayons maintenant de déterminer le temps d'exécution pour des graphes creux avec 100 sommets. Un bon exemple pour des graphes creux sont les cycles C_n. En démarrant à n'importe quel sommet et en ajoutant chaque troisième sommet à l'ensemble dominant, on obtient $\gamma(C_n) = \left\lceil \frac{n}{3} \right\rceil$. Pour un C_{100}, on aura donc : $\gamma(C_{100}) = \left\lceil \frac{100}{3} \right\rceil = 34$.

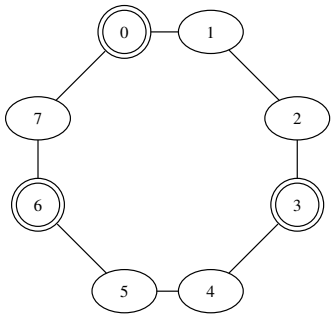


FIG. 2.2: Ensemble dominant minimum d'un C_8

L'algorithme va donc analyser au moins tous les sous-ensembles de sommets de taille 33 avant de trouver un ensemble dominant pour le C_{100}. Pour des graphes de taille 50, les tests ont donné qu'on peut en moyenne analyser $5 * 10^6$ sous-ensembles par seconde. Si on admet que ce temps est supérieur pour les graphes de taille 100, on a la relation suivante :

$TE_{ALG}(C_{100}) \geq \begin{pmatrix} 100 \\ 33 \end{pmatrix} / (5 * 10^6)$ secondes où $TE_{ALG}(C_{100})$ est le temps d'exécution de l'algorithme sur le graphe C_{100} et $\begin{pmatrix} 100 \\ 33 \end{pmatrix}$ est le nombre de sous-ensembles de taille 33 d'un ensemble de taille 100.

En évaluant cette expression, notre algorithme prendra au moins $1,87 * 10^{12}$ années pour trouver un MDS pour un C_{100}, donc plus que 140 fois l'âge de l'univers.

2.2 Algorithme $1,93^n$

La recherche d'algorithmes exponentiels exacts pour problèmes NP-complets était en vogue fin des années '70 jusque début des années '80. Cet intérêt s'est relancé seulement depuis quelques années, sans doute à cause de l'apparition d'ordinateurs de plus en plus rapides. Puisqu'on pense qu'il n'existe pas d'algorithme polynomial pour ces problèmes, on cherche des algorithmes ayant un temps d'exécution en $O^*(c^n)$ avec une constante c aussi petite que possible. Pour l'algorithme précédent, cette constante valait 2, ce qui est une barrière naturelle pour de nombreux problèmes. Dans [4], cette barrière est rompue pour la première fois pour le problème MDS.

Le temps d'exécution de l'algorithme de Fomin, Kratsch et Woeginger dépend largement du théorème de Reed [5] suivant :

Théorème : *Chaque graphe de degré minimal δ au moins 3 contient un MDS de taille au plus $3n/8$.*

En utilisant la technique "pruning the search tree" pour éliminer les sommets de degré inférieur à 3, ils arrivent à un temps d'exécution en

$$O^*(\begin{pmatrix} n \\ 3n/8 \end{pmatrix}) \approx O^*(1,93783^n).$$

Il existe un résultat similaire de Shepherd[6] qui est le suivant :

Théorème : *Chaque graphe de degré minimal δ au moins 2 contient un MDS de taille au plus $2n/5$.*

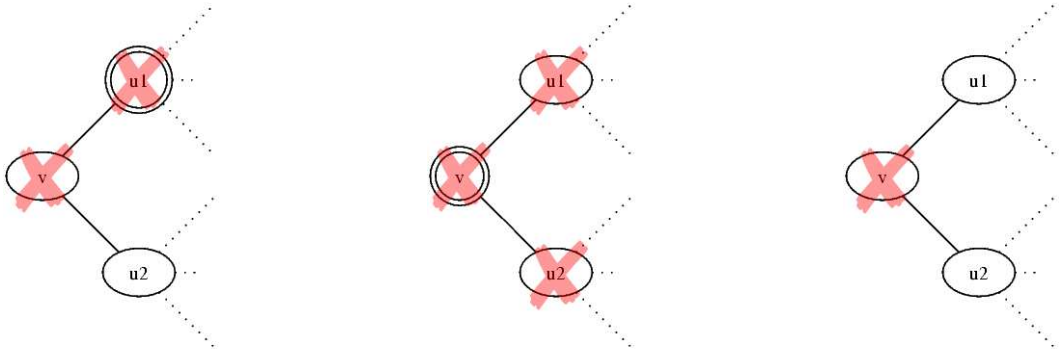
En utilisant seulement ce théorème et en utilisant la même technique, on arriverait à un temps d'exécution de $O^*(1,9601^n)$. Actuellement, il n'existe pas encore de théorème similaire pour un degré minimal supérieur à 3. Cependant, dans [7], Clark a conjecturé que si $\delta(G) \geq 4$, alors $\gamma(G) \leq n/3$.

Dans [4], l'algorithme proposé résout un problème plus général : Soit X un sous-ensemble de sommets du graphe $G = (V, E)$, on considère le problème de trouver un sous-ensemble $D \subseteq V$ qui domine tous les sommets de X. En prenant $X = V$, on a évidemment un

algorithme pour MDS. La technique "pruning the search tree" est utilisée pour se ramener à des cas où $\delta(G) \geq 3$. On peut alors dérouler l'algorithme exact 2, présenté à la page 7 qui analyse au pire des cas tous les sous-ensembles de taille inférieure ou égale à $3n/8$.

L'algorithme choisit un sommet v de degré minimal et fait un branchement (récursif) dans des sous-cas où il enlève un ou plusieurs sommets du graphe, enlève des sommets de l'ensemble X (les sommets qui doivent encore être dominés) et ajoute des sommets dans l'ensemble dominant D. A titre d'exemple, montrons ce qui se passe dans cet algorithme s'il existe un sommet $v \notin X$ de degré 2 dans le graphe.

Cas C : v est de degré 2 et $v \notin X$: Soient u_1 et u_2 les voisins de v. On doit donc résoudre récursivement chacun des 3 cas suivants. Lorsque ces trois cas seront traités, nous choisirons un ensemble dominant retourné qui a une cardinalité minimale. Il est important de noter que v n'a pas besoin d'être dominé dans ce cas. Il peut cependant faire partie de l'ensemble dominant afin de dominer ses voisins.
(C.1) : $u_1 \in D$ et $v \notin D$: si D' est un ensemble dominant de taille minimale pour $X - N[u_1]$ dans $G - \{u_1, v\}$, alors $D' \cup \{u_1\}$ est un ensemble dominant de taille minimale pour X dans G, où $N[u_1]$ est le voisinage fermé de u_1, c'est-à-dire l'ensemble des sommets constitué de u_1 et de ses voisins. Dans ce cas, il y a donc un appel récursif sur $G - \{u_1, v\}$ et $X - N[u_1]$.
(C.2) : $u_1, u_2 \notin D$ et $v \in D$: si D' est un ensemble dominant de taille minimale pour $X - \{u_1, u_2\}$ dans $G - N[v]$, alors $D' \cup \{v\}$ est un ensemble dominant de taille minimale pour X dans G. Dans ce cas, il y a donc un appel récursif sur $G - N[v]$ et $X - \{u_1, u_2\}$.
(C.3) : $u_1 \notin D$ et $v \notin D$: si D' est un ensemble dominant de taille minimale pour X dans $G - \{v\}$, alors D' est aussi un ensemble dominant de taille minimale pour X dans G. Dans ce cas, il y a donc un appel récursif sur $G - \{v\}$ et X.



(a) C.1 (b) C.2 (c) C.3

FIG. 2.3: Cas C : $v \notin X$ de degré 2

Les différents sous-cas sont justifiés par le lemme suivant dont une démonstration est donnée dans [4] :

Lemme : *Soit v un sommet de degré 2 dans G, et soient u_1 et u_2 ses voisins. Alors, pour chaque sous-ensemble $X \subseteq V$, il existe un ensemble dominant de taille minimale D pour X tel qu'une des conditions suivantes est vérifiée :*
(i) $u_1 \in D$ et $v \notin D$
(ii) $v \in D$ et $u_1, u_2 \notin D$
(iii) $u_1 \notin D$ et $v \notin D$

Le cas D où v est de degré 2 et $v \in X$ doit aussi traiter trois sous-cas et les cas où il existe un sommet de degré 1 ou un sommet isolé dans G sont évidemment plus faciles et nécessitent un seul appel récursif (au lieu de 3 si v est de degré 2).

Cet algorithme a été implémenté de deux façons différentes. La différence entre ces deux implémentations se situe surtout au niveau des cas où le plus petit degré du graphe est 2.

1. Pour les appels récursifs, si on doit traiter 3 sous-cas (cas C et D), le graphe initial est dupliqué deux fois. Ensuite, des sommets sont enlevés des graphes conformément au cas C ou D. Le désavantage de cette méthode est la duplication coûteuse des graphes.
2. Dans la deuxième implémentation, le graphe initial n'est jamais copié. On dispose désormais d'un tableau contenant le degré des sommets dans le sous-graphe du graphe initial. Si ce degré atteint -1, alors ce sommet ne fait plus partie du sous-graphe. Pour les appels récursifs, on a donc seulement besoin de copier ce tableau. Cette méthode présente désormais aussi un désavantage : si on parcourt la liste d'adjacence d'un sommet, on doit contrôler pour chaque voisin s'il appartient encore au sous-graphe. On parcourt donc plus de sommets que si l'on avait supprimé directement les sommets du graphe.

Une comparaison du temps d'exécution de ces deux méthodes n'a donné que des petites différences, parfois en faveur de l'une, parfois en faveur de l'autre implémentation. Finalement, la deuxième implémentation a été retenue et un graphique de son temps d'exécution est donné à la figure 2.4.

Comme pour l'algorithme précédent, le temps d'exécution a été déterminé en prenant la moyenne sur 20 instances de graphes, pour un nombre de sommets et une densité donnés. On voit ici que pour les graphes creux, le temps d'exécution est significativement plus petit que pour l'algorithme précédent. Pour les graphes denses, le temps d'exécution est environ le même que pour l'algorithme précédent. Les instances les plus difficiles sont ceux où peu de sommets peuvent être enlevés du graphe et qui ont quand même un ensemble dominant minimum de grande taille. Comme, en moyenne, la taille de l'ensemble dominant croît quand la densité décroît, ce sont les graphes 3-réguliers, c'est-à-dire les graphes où chaque sommet a degré 3.

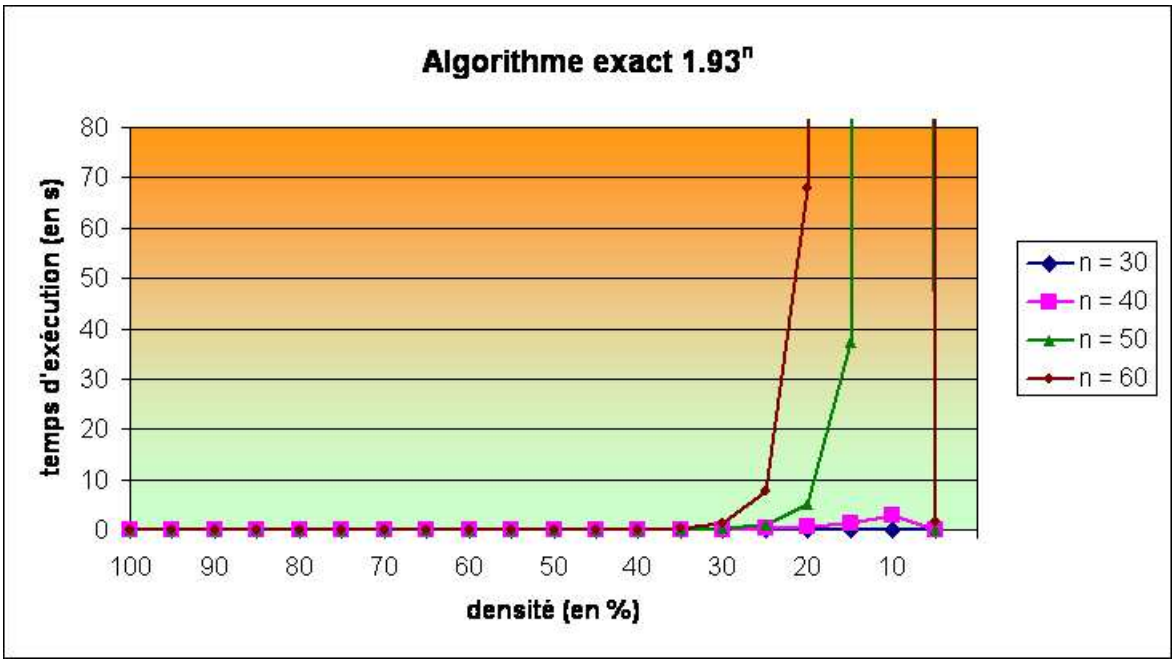


FIG. 2.4: Temps d'exécution de l'algorithme exact $1,93^n$

Lors de ce TER, un algorithme exact [4] en temps $O^*(1.64515^n)$ et un algorithme d'approximation [8, 9] en temps linéaire pour les graphes de degré maximal 3 (qui contiennent comme cas spéciaux les graphes 3-réguliers) ont été étudiés, mais ne seront pas traités dans ce rapport. En particulier, il a été démontré que le facteur d'approximation de l'algorithme d'approximation est constant.

Afin de déterminer le plus grand n tel que tous les graphes avec n sommets peuvent être traités par cet algorithme en temps raisonnable, considérons le tableau complémentaire 2.2. On doit donc s'attendre à des temps d'exécution d'environ 20 minutes pour les graphes d'ordre 50 et de 3 heures pour les graphes d'ordre 60 si la densité du graphe vaut 10. La valeur pour $n = 60$ est toujours meilleure que le temps d'exécution au pire des cas de l'algorithme précédent sur les graphes de taille 40. On peut donc dire que les instances aux bornes de la faisabilité contiennent 50% plus de sommets que pour l'algorithme de la section 2.1.

	n = 50	n = 60
d = 15%	37,23 s	126,74 s
d = 10%	1 279,66 s	10 078,89 s
d = 5%	0,00004 s	1,67 s

TAB. 2.2: Compléments pour le temps d'exécution de l'algorithme exact $1,93^n$

3 Heuristiques et Algorithmes d'approximation

Cette partie va présenter des techniques souvent utilisées pour donner une solution "pas trop mauvaise" en temps polynomial si on est confronté à un problème NP-difficile. Comme on l'a vu dans la partie précédente, les algorithmes exponentiels ne sont pas assez efficaces pour résoudre le problème de domination sur toutes les instances de graphes dépassant 60 sommets. Mais souvent, une application peut se contenter d'une solution quasi-optimale étant confronté à un tel problème. Deux algorithmes approchant une solution optimale sont présentés ici. Le premier est un algorithme d'approximation et le deuxième est basé sur une heuristique. La différence entre un algorithme d'approximation et un algorithme basé sur une heuristique est assez petite.

Remarque : On s'intéresse ici seulement à des méthodes de résolution étant reliées directement au problème qui fournissent en général de meilleures solutions que les méthodes générales comme la recherche tabou, le recuit simulé ou encore les algorithmes génétiques.

Le nom *heuristique* provient du mot grec "Eureka" et désigne en informatique une méthode de résolution de problèmes, non fondée sur un modèle formel et qui n'aboutit pas nécessairement à une solution exacte.

Un *algorithme d'approximation* est un algorithme polynomial qui fonctionne pour toutes les entrées du problème. Cependant, on ne demande pas que l'algorithme trouve toujours une solution exacte, mais on exige quand même une garantie de l'optimalité de la solution trouvée.

Toutes les deux sont donc des algorithmes fournissant une solution assez bonne en temps polynomial. On dira qu'un algorithme A est un algorithme d'approximation si on connaît une certaine garantie de sa solution. Cette garantie est exprimée par son rapport d'approximation R_A qui est défini comme ceci pour les problèmes de minimisation :

Définition : Le rapport d'approximation d'un algorithme A pour une entrée I vaut $R_A(I) = \frac{A(I)}{OPT(I)}$, où $A(I)$ est la taille de la solution de l'algorithme et $OPT(I)$ est la taille d'une solution optimale.

Définition : Le rapport d'approximation R_A d'un algorithme A est pris au pire des cas et vaut $R_A = max\{R_A(I) \mid I \; est \; une \; instance \; du \; problme\}$.

Pour ce type d'algorithmes, la qualité de la solution est souvent plus importante que le temps d'exécution. Afin de juger la qualité des solutions d'un algorithme, on pourrait se limiter à des graphes qui sont traitables par les algorithmes exacts. Mais, si une application utilise un algorithme d'approximation, elle est souvent confrontée à des instances plus grandes du problème. Afin de pouvoir déterminer le degré d'optimalité de ces algorithmes pour des instances plus grandes, un générateur de graphes aléatoires où la taille k d'un ensemble dominant minimum est connue à l'avance est utilisé [10].

La construction d'un graphe G de n sommets, de densité d et avec un ensemble dominant minimum de taille k procède comme suit :

1. Partitionner les sommets de G en k groupes disjointes $V_1, V_2, ..., V_k$.
2. Choisir deux sommets x_i et y_i dans chaque groupe V_i, $1 \leq i \leq k$.
3. Ajouter des arêtes entre les sommets pour satisfaire une densité d, tel que les deux conditions suivantes sont respectées :

   a) Chaque sommet différent d'un x_i est adjacent à un x_i.

   b) Pour chaque i, y_i n'est pas adjacent à un sommet n'appartenant pas à V_i : $\forall i, \{y_i, v\} \in E \implies v \in V_i$

Le graphe résultant aura donc un ensemble dominant minimum de taille k, car par (a), $\{x_1, x_2, ..., x_k\}$ est un ensemble dominant et par (b), chaque ensemble dominant doit posséder au moins un sommet de chaque V_i. Dans [10], il a été montré que les graphes résultants ont une grande diversité et qu'ils sont difficiles de traiter.

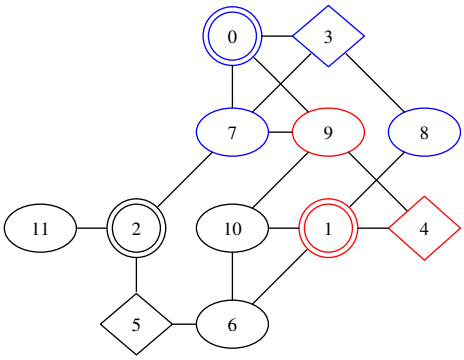


FIG. 3.1: Graphe construit avec le générateur de graphes avec γ connu

Un exemple d'un graphe produit avec ce générateur est donné à la figure 3.1. Sur ce graphe, les sommets sont divisés en trois groupes "bleu", "rouge" et "noir". Les sommets x_i sont entourés d'un double cercle et les y_i sont représentés par des losanges.

3.1 Algorithme d'approximation glouton

Les algorithmes d'approximation sont particulièrement intéressants parce qu'ils nous donnent une garantie sur la qualité de l'approximation. Une bonne introduction à ce type d'algorithmes est donnée en [11].

L'algorithme d'approximation de cette section va construire l'ensemble dominant D en choisissant à chaque fois un sommet qui domine le plus grand nombre de sommets.

Algorithm 3 Greedy-Approx

```
GREEDY-APPROX ( G = (V, E) : GRAPHE ) : DS

   D ← ∅
   Pour chaque sommet v ∈ V
      faire poids[v] ← 1 + degre[v]
            couleur[v] ← BLANC
   Tant que poids[v] ≠ 0
      faire v ← random{u ∈ V | poids[u] = max_{w∈V}{poids[w]}}
            Si poids[v] ≠ 0
               alors D ← D ∪ {v}
                     poids[v] ← 0
                     AJUSTER-POIDS(v)
                     couleur[v] ← NOIR
   retourner D


AJUSTER-POIDS ( v : SOMMET )

   Pour chaque voisin u de v tel que poids[u] > 0
      faire Si couleur[v] = BLANC
               alors poids[u] ← poids[u] - 1
            Si couleur[u] = BLANC
               alors couleur[u] ← NOIR
                     poids[u] ← poids[u] - 1
                     Pour chaque voisin w de u tel que poids[w] > 0
                          faire poids[w] ← poids[w] - 1
```

Dans le pseudo-code 3 que voici, $poids[v]$ désigne le nombre de sommets que le sommet v dominerait si on l'ajoutait à l'ensemble dominant D. Comme dans l'algorithme 1, la *couleur* d'un sommet nous indique si ce sommet est déjà dominé (NOIR) ou pas (BLANC). A chaque pas d'itération de l'algorithme, un des sommets dont le *poids* est maximal est choisi de manière aléatoire et on l'ajoute à D. Ensuite, il faut évidemment mettre à jour le *poids* et la *couleur* de ce sommet et de ses voisins.

Johnson a établi en 1974 que cet algorithme a un facteur d'approximation de $1 + log_2 n$ par une réduction du problème SET COVER. En plus, il est également connu qu'il n'existe pas d'algorithme avec un facteur d'approximation de $(1 - \varepsilon) log_2 n$ pour un $\varepsilon > 0$, sauf si $P \neq NP$ (cf. [12]). On ne peut donc pas espérer trouver d'algorithme avec un meilleur facteur d'approximation (constant, par exemple).

Concernant le temps d'exécution, chaque choix d'un sommet de poids maximal peut être effectué en $O(n)$ et le temps passé dans AJUSTER-POIDS à travers tout l'algorithme est $O(m)$. Si d est la taille de l'ensemble dominant déterminé par cet algorithme, il s'exécute en temps $O(nd + m) \leq O(n^2)$.

Les tableaux de tests sont obtenus en exécutant cet algorithme sur 50 instances de graphes construits avec le générateur de graphes aléatoires avec γ connu, pour une densité et une taille d'un MDS donnés. On donne à chaque fois la taille moyenne, minimale et maximale des solutions obtenues avec l'algorithme GREEDY-APPROX, ainsi que le temps d'exécution moyen.

n	densité	$\gamma(G)$	trouvé-moy	trouvé-min	trouvé-max	temps d'exécution
2000	5%	90	113,10	108	120	0,027 s
2000	5%	80	99,94	96	104	0,027 s
2000	5%	70	88,88	83	93	0,028 s
2000	20%	25	30,24	28	33	0,088 s
2000	20%	20	24,36	22	28	0,090 s
2000	20%	15	17,78	16	21	0,087 s
2000	50%	8	9,08	8	11	0,174 s
2000	50%	7	7,91	7	10	0,173 s
2000	50%	6	6,54	6	8	0,173 s
10000	5%	120	151,96	147	159	1,052 s
10000	5%	110	141,10	134	146	1,054 s
10000	5%	100	129,40	119	137	1,051 s
10000	20%	26	32,18	28	35	4,600 s
10000	20%	24	29,48	26	33	4,480 s
10000	20%	22	26,74	23	30	4,482 s

TAB. 3.1: Tests de l'algorithme d'approximation glouton sur des graphes d'ordre 2000 et 10 000

Pour les graphes de taille 2000, le facteur d'approximation de l'algorithme nous assure que l'ensemble dominant trouvé est au plus $1 + log_2 2000 \approx 11,96$ fois la taille d'un ensemble dominant minimum. En pratique, comme nous le voyons sur la figure 3.1, la différence n'est pas si grande que ça. La différence n'est en effet guère plus grande que 25%.

Cet algorithme nous donne des bons résultats en général, et ce très rapidement. Mais, même s'il réalise un choix aléatoire sur les sommets ayant le même poids, plusieurs exécutions de cet algorithme sur le même graphe donnent des résultats assez proches.

En effet, la plupart des heuristiques pour MDS sont basées sur le degré des sommets (cf. [13]), mais les solutions pour une instance donnée ne diffèrent que peu et peuvent atteindre des limites d'optimalité.

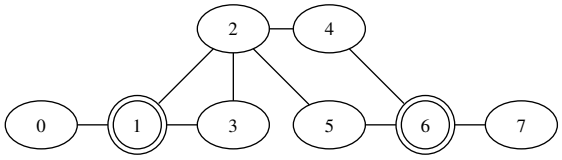


FIG. 3.2: Graphe où Greedy-Approx ne trouve pas de solution optimale

Sur la figure 3.2, un exemple de graphe est donné où l'algorithme GREEDY-APPROX ne peut en aucun cas trouver une solution optimale. En effet, $\{1, 6\}$ constitue un ensemble dominant minimum de taille 2, mais l'algorithme choisirait d'abord le sommet 2 qui a le plus grand degré, ensuite le sommet 6 ou 7 qui dominent encore deux sommets et finalement le sommet 0 ou 1 pour dominer le sommet 0. Pour ce graphe, GREEDY-APPROX ne trouve donc jamais un ensemble dominant minimum.

3.2 Heuristique basée sur un algorithme de Bottleneck Dominating Set

Cette section va présenter un algorithme qui peut être exécuté aussi longtemps qu'on veut et qui essayera de trouver un ensemble dominant de plus en plus petit. Il ne peut désormais pas détecter s'il a déjà trouvé une solution optimale, car comme nous le savons, cette tâche est NP-complète.

Considérons le problème BOTTLENECK DOMINATING SET : Soit $G = (V, E)$ un graphe non-orienté et une fonction $w : V \to \mathbb{R}$ qui assigne à chaque sommet v un poids $w(v)$. Il s'agit de trouver un ensemble dominant D pour G tel que le poids maximal des sommets de D soit aussi petit que possible. On définit le bottleneck d'un ensemble dominant comme ceci :

$$D \subseteq V : \; bottleneck(D) = max\{w(u) \mid u \in D\}$$

Le but est donc de trouver un ensemble dominant D, tel que $bottleneck(D)$ soit minimal.

Bien que ce problème semble très proche de MIMIMUM DOMINATING SET, il n'est pas NP-difficile et dans [14], un algorithme linéaire ($O(n + m)$) est donné pour le résoudre exactement.

L'algorithme 4 est basé sur une fonction $m(v)$ qui assigne à chaque sommet v le plus petit poids de son voisinage fermé. Clairement, si on considère un sommet v, au moins v ou un de ses voisins est contenu dans un ensemble dominant. Le bottleneck du graphe

Algorithm 4 Algorithme linéaire pour le problème Bottleneck Dominating Set

ALGO-BDS (G = (V, E) : GRAPHE, w : $V \to \mathbb{R}$)

1. $\forall v \in V$, calculer $m(v) = min\{w(u) \mid u \in N[v]\}$, où $N[v]$ désigne le voisinage fermé de v, c'est-à-dire l'ensemble constitué de v et de ses voisins
2. $\beta = max_{v \in V}\{m(v)\}$ est le bottleneck minimum du graphe pondéré (G, w)
3. $BDS = \{v \in V \mid w(v) \leq \beta\}$ est un Bottleneck Dominating Set pour (G, w)

est donc au moins $m(v)$. En considérant tout le graphe, chaque sommet doit être dominé et le bottleneck minimum du graphe sera donc la valeur maximale de tous les $m(v)$. A la fin, tous les sommets ayant un poids inférieur au bottleneck minimum sont choisis pour les ajouter dans l'ensemble dominant.

Sur l'exemple de la figure 3.3, les noms des sommets correspondent à leur poids pour ne pas surcharger la figure. Pour le sommet 7, on a $m(7) = min\{4, 3, 7, 2\} = 2$. Après avoir évalué $m(v)$ pour chaque sommet, on obtiendra que $\beta = 5$ et le Bottleneck Dominating Set sera constitué des sommets dont le poids est inférieur ou égal à 5 (entourés d'un double cercle). Remarquons que sur cet exemple, l'algorithme pour BDS a trouvé un ensemble dominant minimal, ce qui est loin d'être toujours le cas. L'ensemble trouvé n'est cependant pas un ensemble dominant minimum, car $\{2, 6, 8\}$ est un ensemble dominant de taille 3.

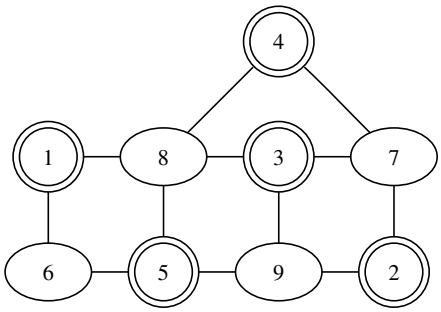


FIG. 3.3: Illustration Algo-BDS

Vu que cet algorithme est très rapide, il est un bon candidat pour élaborer une heuristique qui l'utilise. L'idée est de choisir comme fonction w une permutation aléatoire des nombres entiers de 1 à n qui affectera à chaque sommet un poids. Ensuite, l'algorithme ALGO-BDS est exécuté pour le graphe d'entrée muni de cette fonction w. Finalement, on procède à une optimisation de l'ensemble dominant trouvé par ALGO-BDS. On peut exécuter cet algorithme plusieurs fois et espérer qu'il trouve de meilleurs ensembles dominants pour le graphe d'entrée, mais on ne peut pas être sûr qu'il converge vers une solution optimale.

Algorithm 5 Algorithme heuristique basé sur BDS

```
HEUR-BDS ( G = (V, E) : GRAPHE, ITERATIONS : ENTIER, NB-OPTIMISER : ENTIER )

    D_opt ← V
    Pour i = 1 à Iterations
       faire Créer une permutation aléatoire w de la suite 1..n
             BDS ← ALGO-BDS(G, w)
             Pour j = 1 à Nb-Optimiser
                faire D ← OPTIMISER(BDS)
                      Si |D| < |D_opt|
                         alors D_opt ← D
```

La fonction OPTIMISER va assurer que D soit au moins un ensemble dominant minimal. Pour cela, elle considère chaque sommet $v \in BDS$ et détermine si elle peut l'enlever de l'ensemble dominant sans qu'il y ait des sommets non dominés dans le graphe. Si la suppression de ce sommet ne rompt pas la dominance, alors elle l'enlève, sinon, elle le laisse dans l'ensemble dominant. Les tests ont montré que si elle considère les sommets dans un ordre aléatoire, les résultats sont meilleurs que lorsqu'elle les considère toujours dans le même ordre. C'est aussi pour cela, qu'on peut choisir d'exécuter la fonction OPTIMISER plusieurs fois pour un BDS trouvé par ALGO-BDS.

Sans la fonction OPTIMISER, les résultats de l'algorithme sont assez médiocres, si on l'exécute une fois, alors les résultats sont déjà beaucoup meilleures et si on l'exécute plusieurs fois, l'algorithme trouve des solutions comparables après moins d'itérations et en un temps un peu plus petit.

Le tableau de test de la figure 3.2 a été obtenu dans les mêmes conditions que celui pour l'algorithme d'approximation glouton. Pour ITERATIONS et NB-OPTIMISER, on a choisi les valeurs 10 et 2, respectivement. On voit tout de suite que les temps d'exécution sont plus grands que ceux de l'algorithme précédent. Avec une seule itération et une seule phase d'optimisation, on aurait cependant obtenu des temps d'exécution comparables à l'algorithme d'approximation. En ce qui concerne l'optimalité des solutions, cet algorithme nous fournit aussi des bons résultats. On peut remarquer que pour les graphes creux, il trouve de meilleures solutions et pour les graphes denses, c'était l'algorithme 3 qui trouvait de meilleurs résultats, surtout pour les graphes où γ est petit à la base. Mais la qualité des solutions diffère seulement peu et la meilleure solution pour déterminer un ensemble dominant de taille petite serait d'exécuter l'algorithme d'approximation et ensuite celui-ci avec autant d'itérations jusqu'à ce qu'on est satisfait du résultat.

n	densité	$\gamma(G)$	trouvé-moy	trouvé-min	trouvé-max	temps d'exécution
2000	5%	90	98,56	94	102	1,108 s
2000	5%	80	91,70	88	95	1,114 s
2000	5%	70	85,22	80	89	1,115 s
2000	20%	25	26,80	26	28	3,054 s
2000	20%	20	24,50	22	26	2,862 s
2000	20%	15	22,92	21	24	2,605 s
2000	50%	8	9,03	8	10	4,021 s
2000	50%	7	8,63	7	10	3,748 s
2000	50%	6	8,29	6	9	3,513 s
10000	5%	120	125,90	122	128	51,947 s
10000	5%	110	119,02	116	122	50,844 s
10000	5%	100	113,04	111	117	50,207 s
10000	20%	26	31,54	30	33	89,089 s
10000	20%	24	30,54	27	32	86,652 s
10000	20%	22	29,94	28	31	85,528 s

TAB. 3.2: Tests de l'algorithme HEUR-BDS sur des graphes d'ordre 2000 et 10 000

4 Pour aller plus loin ...

4.1 Combiner heuristiques et algorithmes exacts

Dans l'algorithme 2 à la page 7, les deux bornes BorneInf et BorneSup n'étaient volontairement pas codés en dur, parce qu'on essaye dans cette section de donner des heuristiques pour les déterminer de manière plus optimale. Pour BorneSup, on peut prendre le résultat de l'exécution d'un algorithme de la partie 3. Ceci serait particulièrement intéressant pour l'approche "du haut vers le bas", mais on va essayer ici plutôt de déterminer une meilleure valeur pour BorneInf. Trois méthodes sont utilisées ici pour trouver une borne inférieure pour la taille d'un ensemble dominant minimum.

Diamètre

Le diamètre d'un graphe $G = (V, E)$ est la longueur du plus long chemin parmi tous les plus courts chemins dans le graphe. Pour le déterminer, on effectue pour chaque sommet un parcours en largeur (BFS). Soit $dist(u, v)$ la longueur d'un plus court chemin du sommet u au sommet v. La hauteur de l'arbre résultant du parcours en largeur pour le sommet v donne la longueur maximale $max\{dist(v, u) : u \in V\}$ de tous les plus courts chemins issus de v. Si on effectue un parcours en largeur pour tous les sommets, on obtiendra le diamètre $diam(G) = max\{dist(u, v) : u, v \in V\}$ en temps $O(n^2 + nm)$.

Si on connaît le diamètre d'un graphe, on sait qu'un sous-ensemble de sommets du graphe induit un chemin contenant $diam(G) + 1$ sommets. La domination d'un chemin est facile : il suffit de commencer par le deuxième sommet du chemin et de prendre ensuite chaque troisième pour l'ajouter à l'ensemble dominant. Un chemin P_n de longueur $n - 1$ aura donc comme nombre de domination $\gamma(P_n) = \lceil n/3 \rceil$. Ceci nous donne une première borne inférieure pour la taille d'un MDS du graphe :

$$\gamma(G) \geq \left\lceil \frac{diam(G)+1}{3} \right\rceil$$

Les résultats de cette détermination d'une borne inférieure ne sont en général pas exceptionnellement bons, car seulement une petite partie du graphe (le chemin donné par le diamètre) est considérée.

Algorithme d'approximation glouton

Vu qu'on connaît un facteur d'approximation pour l'algorithme de la section 3.1, on peut l'exploiter pour obtenir une borne inférieure pour $\gamma(G)$. Par une transformation de l'équation du facteur d'approximation de cet algorithme, on obtient que

$$\gamma(G) \geq \frac{A(G)}{1+log_2 n}$$

où $A(G)$ est la taille d'un ensemble dominant retourné par l'algorithme d'approximation glouton.

Vu que le facteur d'approximation est relativement mauvais pour cet algorithme et parce qu'en pratique, il trouve souvent des solutions beaucoup plus proches de l'ensemble dominant minimum, cette approche trouve aussi des bornes inférieures assez petites. Dans les tests, il a été constaté que pour des grands graphes, cette heuristique trouve de meilleures résultats que celle basée sur le diamètre, alors que c'est l'inverse pour les petits graphes.

2-Packing

Si on considère un sommet $v \in V$, alors au moins v ou un de ses voisins appartiendra à l'ensemble dominant. Cette approche va déterminer une borne inférieure pour la taille d'un MDS en considérant toutes les possibilités de dominer v. Pour chaque sommet u qui domine v, c'est-à-dire pour chaque sommet $u \in N[v]$, on va colorier u et ses voisins en NOIR et on incrémente la borne inférieure. Ensuite, on fait la même chose pour les sommets BLANCs restants.

Algorithm 6 2-Packing pour déterminer BORNEINF

```
2-PACKING ( G = (V, E) : GRAPHE ) : ENTIER

   ∀v ∈ V, couleur[v] ← BLANC
   BORNEINF ← 0
   perm[] ← permutation aléatoire des sommets de V
   Pour i = 1 à n
      faire Si couleur[perm[i]] = BLANC
               alors ∀v ∈ N[N[perm[i]]], couleur[v] ← NOIR
                     BORNEINF ← BORNEINF + 1
   retourner BORNEINF
```

Dans l'algorithme 6 que voici, on considère les sommets dans un ordre aléatoire. A chaque fois qu'on choisit un sommet et qu'il est blanc, on dira que ce sommet appartiendra au packing S. On colorie alors en NOIR ce sommet, ses voisins et les sommets à distance 2 et on incrémente la borne inférieure.

Les sommets qui appartiennent au packing S satisfont la condition suivante :

$$\forall s, s' \in S : dist(s, s') > 2$$

A cause de la permutation aléatoire des sommets, cet algorithme peut être exécuté plusieurs fois pour obtenir des meilleurs résultats. Si on le compare avec les autres heuristiques sur des instances de test, on remarque que 2-PACKING obtient pratiquement toujours la plus grande borne inférieure pour $\gamma(G)$.

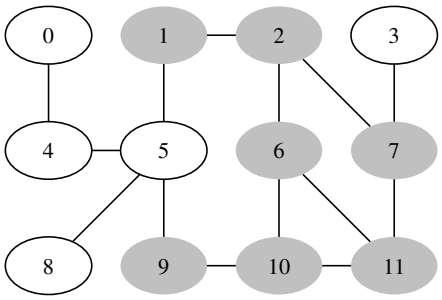


FIG. 4.1: Exemple 2-Packing

Considérons l'exemple de la figure 4.1. Admettons que l'algorithme choisit comme d'abord le sommet 6. Il colorie alors tous les sommets à une distance au plus 2 en NOIR. Ensuite, il choisira, par exemple le sommet 3 et puis le sommet 5, ce qui colorie tout le graphe en NOIR. Ceci nous donne une borne inférieure de 3, ce qui n'est pas mal pour un graphe ayant un ensemble dominant de taille minimale 4 (par exemple {4,5,7,10}). En plus, si on choisit les sommets dans un ordre différent, par exemple {0,3,6,8}, on peut obtenir une borne inférieure de 4.

Voici deux exemples d'exécution de l'algorithme exact en temps $O^*(2^n)$, combiné avec ces trois heuristiques exécutés sur le même graphe. Dans la première exécution, 2-PACKING est exécuté une seule fois :

```
diametre    =   10   --->  |D|>=4
|D_approx|  =   14   --->  |D|>=2
|2-Packing| =   10   --->  |D|>=10
                          ---------
                           |D|>=10
Ensemble dominant trouve de taille 11 :
0 2 8 10 16 22 24 25 31 32 38
```

Le temps d'exécution était de 114 secondes, contre 150 si aucune heuristique n'est exécutée pour trouver la borne inférieure.

```
diametre    =   10   --->  |D|>=4
|D_approx|  =   14   --->  |D|>=2
|2-Packing| =   11   --->  |D|>=11
```

```
                          ---------
                           |D|>=11
Ensemble dominant trouve de taille 11 :
0 2 8 10 16 22 24 25 31 32 38
```

Dans la deuxième exécution, on a exécuté 2-PACKING dix fois et le temps d'exécution était de l'ordre de 28 secondes. On remarque que pour diminuer considérablement le temps d'exécution, une très bonne borne inférieure est nécessaire, car le temps d'exécution dépend largement de l'analyse des grands sous-ensembles de sommets. Malheureusement, il est rare que les heuristiques donnent une si bonne borne inférieure que pour l'exemple ci-dessus.

4.2 Nombre de MDS

Afin d'espérer trouver un meilleur algorithme exact, avec une constante c plus petite dans son temps d'exécution $O^*(c^n)$, on s'intéresse souvent au nombre de solutions qu'il existe au plus, ainsi qu'aux propriétés des instances pour lesquelles il existe un grand nombre de solutions. Pour le problème de domination, on peut ainsi s'intéresser aux graphes pour lesquelles il existe un grand nombre d'ensembles dominants minimums. Appelons $\#mds(G)$ le nombre de MDS différents pour le graphe $G = (V, E)$ et $\#mds(n)$ la valeur maximale des $\#mds(G)$ pour tous les graphes G de taille n.

Afin d'avoir une première idée, un programme a été développé pour déterminer $\#mds(n)$ qui utilise une version légèrement modifiée de l'algorithme exact de la section 2.1 qui s'exécute en $O^*(2^n)$. On n'aurait pas pu utiliser l'algorithme de la section suivante (en $O^*(1,93^n)$), car il n'est pas capable de fournir tous les ensembles dominants minimums. L'algorithme exact est donc exécuté sur tous les graphes non-isomorphes de taille n, en les décomposant d'abord en leurs composantes connexes pour accélérer le traitement.

Définition : On dit que deux graphes $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$ sont *isomorphes* s'il existe une bijection f de V_1 dans V_2 telle que pour tous les sommets u et v de V_1, on a $\{u, v\} \in E_1 \iff \{f(u), f(v)\} \in E_2$.

Deux graphes sont donc isomorphes si ce sont les mêmes graphes, à un renommage de leurs sommets près. Un programme très rapide pour créer des graphes non-isomorphes d'une taille donnée est `nauty`, développé par Brendan McKay à une université en Autriche. On l'a interfacé ici avec une version de l'algorithme exact qui calcule tous les ensembles dominants de taille minimale. Nous obtenons alors le tableau 4.1.

n	1	2	3	4	5	6	7	8	9	10	11
#mds(n) pour graphes généraux	1	2	3	6	9	15	22	36	54	90	135
#mds(n) pour graphes connexes	1	2	3	6	9	15	22	36	51	87	127

TAB. 4.1: $\#mds(n)$ pour graphes généraux et graphes connexes

Un graphe particulièrement intéressant est celui avec 6 sommets et qui admet 15 ensembles dominants différents de taille minimale 2. C'est l'octaèdre de la figure 4.2. En construisant des graphes où toutes les composantes connexes sont des octaèdres, on obtient des graphes où

$$\#mds(G) = 15^{\frac{n}{6}} = \left(\sqrt[6]{15}\right)^n \approx 1,5704^n,$$

car pour obtenir $\#mds(G)$ d'un graphe non-connexe G, on peut calculer le nombre d'ensembles dominants minimums pour toutes les composantes connexes de G et leur produit nous donne $\#mds(G)$:

$\#mds(G) = \prod_{i=1..k} \#mds(C_i)$ où $C_1, ..., C_k$ sont les composantes connexes de G

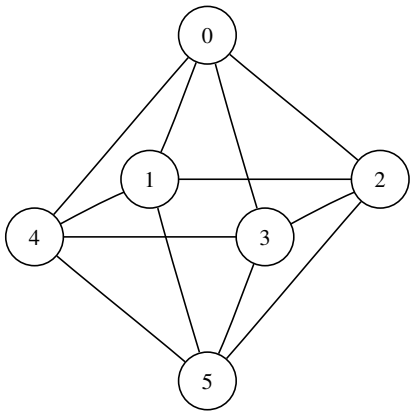


FIG. 4.2: Octaèdre

Une borne inférieure pour la valeur maximale des $\#mds(G)$ est donc $1,5704^n$ et la seule borne supérieure connue aujourd'hui est 2^n. Cela implique qu'il est impossible de construire un algorithme qui sort tous les MDS d'un graphe en temps inférieur à $O^*(1,5704^n)$. Une chose intéressante à démontrer serait que tous les graphes qui ont $\#mds(n)$ ensembles dominants minimaux sont non-connexes si $n \geq 9$. Cela impliquerait que $1,5704^n$ serait aussi une borne supérieure pour $\#mds(n)$.

5 Conclusion

Ce TER m'a permis d'avoir une première impression de ce qui se passe dans la recherche en algorithmique de nos jours. En effet, les chercheurs se sont intéressés de nouveau à des algorithmes exacts pour problèmes NP-difficiles avec un temps d'exécution exponentiel dans les dernières années. Ceci m'a aussi permis de voir sur un exemple concret ce que NP-difficile signifie vraiment en pratique : le problème devient infaisable pour des grandes entrées, malgré la puissance des ordinateurs actuels.

Dans beaucoup d'applications, il n'est souvent pas nécessaire de trouver des résultats optimaux, mais seulement d'approcher une solution optimale. Si les solutions ne doivent pas forcément être optimales, le problème de domination devient traitable pour des grandes instances malgré la complexité du problème.

Finalement, les deux approches - résolution exacte et approchée - d'un problème NP-difficile m'ont fortement intéressées et j'espère pouvoir continuer dans ces domaines.

Bibliographie

[1] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, *Domination in Graphs : Advanced Topics*. Marcel Dekker Inc., New York, January 1998.

[2] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, *Fundamentals of domination in graphs*. Marcel Dekker Inc., New York, 1998.

[3] O. Ore, *Theory of Graphs*. AMS Colloquium Publications 38, 1962.

[4] F. V. Fomin, D. Kratsch, and G. J. Woeginger, "Exact (exponential) algorithms for the dominating set problem." preprint.

[5] B. Reed, "Paths, stars and the number three," *Combinatorics, Probability and Computing 5*, pp. 277–295, 1996.

[6] W. McCuaig and B. Shepherd, "Domination in graphs with minimum degree two," *Journal of Graph Theory*, vol. 13, no. 6, pp. 749–762, 1989.

[7] W. E. Clark and L. A. Dunning, "Tight upper bounds for the domination numbers of graphs with given order and minimum degree," *The Electronic Journal of Combinatorics*, vol. 4, no. 1, pp. 1–25, 1997.

[8] D. C. Fisher, K. Fraughnaugh, R. M. McConnell, and S. M. Seager, "A linear-time algorithm for small dominating sets in graphs of maximum degree 3." preprint.

[9] D. C. Fisher, K. Fraughnaugh, and S. M. Seager, "Domination of graphs with maximum degree three," Tech. Rep. UCD-CCM-090, Center for Computational Mathematics, January 1996.

[10] L. A. Sanchis, "Generating hard and diverse test sets for np-hard graph problems," *Discrete Applied Mathematics*, vol. 58, pp. 35–66, 1995.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction à l'Algorithmique*. Dunod, 2e ed., 2001.

[12] P. Crescenzi and V. Kann, "A compendium of np optimization problems." http ://www.nada.kth.se/~viggo/wwwcompendium/.

[13] L. A. Sanchis, "Experimental analysis of heuristic algorithms for the dominating set problem," *Algorithmica*, vol. 33, pp. 3–18, January 2002.

[14] T. Kloks, D. Kratsch, C. M. Lee, and J. Liu, "Improved bottleneck domination algorithms," technical report, Department of Mathematics and Computer Science, The University of Lethbridge, Canada, May 2003.