UNIVERSITÉ
DE MONTPELLIER

TER - Algorithmes de coloration de graphes, conjecture de Reed

réalisé par :Aicha Boukhari, Morgan Aizon, Arthur Mittelstaedt
Professeur encadrant: Bessy Stéphane
Année universitaire 2021/2022

Table des matières

1 Introduction

La théorie des graphes est une discipline mathématique et informatique qui étudie les graphes, dont l'usage est d'une grande utilité pour de nombreux domaines, beaucoup de problèmes peuvent être résolus et mieux compris lorsqu'on les modélise sous forme de graphes (des arbres, flots ..).

De nombreux résultats ont été prouvés sur la théorie des graphes et de nombreuses conjectures restant à démontrer ont été proposées. Dans ce projet, on s'intéresse à l'une d'entre elles, la conjecture dite de Reed qui traite de la coloration des graphes en général mais nous nous intéresserons en particulier à ses implications pour les graphes sans triangles.

Les objectifs du projet :

- Étudier différents algorithmes exacts pour le calcul du nombre chromatique d'un graphe (Lawler, Fisrt-Fit, Bjorklund et Husfeld) et d'implémenter au moins deux (First-Fit, Lawler).

- Générer tous les graphes d'un certain type, en particulier nous nous intéresserons aux graphes sans triangles et de degré maximal 6. Un autre objectif est de réduire les temps de calculs pour pouvoir vérifier la conjecture pour des graphes plus grands.

- Tester de manière exhaustive la conjecture de B.Reed, portant sur le lien entre nombre chromatique, clique number et degré maximal. Ce, pour tous les graphes avec un nombre de sommets fixé. Cette conjecture sera présentée sous plusieurs formes, on s'intéressera en particulier aux premiers cas non triviaux.

2 Gestion du projet

2.1 Choix du sujet

Étant étudiants du parcours algorithmique, nous avons suivi au premier semestre le module de "Graphe et structure" d'où des connaissances basiques sur la théorie des graphes et notre intérêt pour ce sujet.
Il nous faut préciser que nous avons commencé le TER avec un autre sujet. Celui-ci était similaire au sujet finalement traité en cela qu'il consistait aussi à tester de manière exhaustive une conjoncture portant sur le nombre chromatique des graphes sans triangles.
Ainsi, lorsque nous avons étudié les algorithmes de coloration décrits ici, nous avions pour but de les utiliser dans un autre contexte.
La raison de ce changement de sujet a été la découverte, au cours de nos recherches bibliographiques d'une publication toute récente annonçant la résolution de la conjecture de notre premier sujet, diminuant l'intérêt du problème.

2.2 Organisation du travail bibliographique

On peut résumer en 3 thèmes les recherches bibliographiques que nous avons mené pour ce TER.

- Recherches autour de la conjecture de Reed et de la génération de graphes : afin de comprendre le contexte de notre problématique, ce travail a été réalisé collectivement avant le début de la phase de programmation.
- Recherches autour des algorithmes de coloration. Le sujet fixait clairement les algorithmes sur lesquels nous devions nous concentrer, nous nous sommes partagé le travail, chacun était chargé d'étudier un algorithme puis d'en présenter le principe aux autres. Nous n'avons pas eu le temps d'étudier en détail le dernier algorithme évoqué dans le sujet, celui de Beigel et Eppstein.
- Recherches autour de la résolution de problèmes annexes. Au cours de la phase de programmation, nous avons rencontré plusieurs problèmes pré-requis pour l'implémention de certains algorithmes : permutations, génération de stables, etc. Ce qui nous a conduit à nous renseigner sur les solutions algorithmiques connues, au fur et à mesure que ces problèmes se présentaient à nous.

2.3 Organisation du travail pratique

Sachant que la durée des calculs sera un enjeu important, nous avons pris la décision d'utiliser le langage C, dans la perspective de pouvoir optimiser notre implémentation.
Nous avons chercher à utiliser les outils modernes disponibles pour la gestion d'un projet de programmation. Ainsi nous avons choisi l'IDE visual studio code et notre projet était hébergé sur un dépôt git hub pour faciliter le travail de groupe sur un même code source. Un outil qui s'est révélé particulièrement utile est l'extension "Live Share" de visual studio code, qui permet à plusieurs utilisateurs de travailler en simultané sur un même fichier depuis leurs machines respectives. Enfin l'écriture de ce rapport en latex, s'est fait tout au long du semestre grâce à un projet "OverLeaf". La partie "algorithmes de coloration" a été reprise de notre sujet précédant.

3 Préliminaires

Pour faciliter la compréhension du sujet, on va présenter certaines notions importantes sur la théorie des graphes qui seront fréquemment utilisées dans la suite du rapport.[1]

Graphe : Un graphe (fini) $G = (V, E)$ est constitué :
- d'un ensemble (fini) de sommets V (ou $V(G)$) de taille n.
- d'un ensemble d'arêtes E (ou $E(G)$), paires d'éléments de V, de taille m.

Deux sommets $x, y \in V$ tels que $\{x, y\} \in E$ sont dits voisins, reliés ou adjacents. On note $\{x, y\} \in E$ ou $xy \in E$ (ou $yx \in E$). L'arête xy est incidente aux sommets x et y qui sont ses extrémités.
Si deux arêtes ont une extrémité en commun, elles sont adjacentes, sinon elles sont disjointes.

Chemin : Un chemin est une suite d'arêtes distinctes $(x_1x_2, x_2x_3, ..., x_{k-1}x_k)$, toutes inclues dans $E(G)$. Si $x_1 = x$ et $x_k = y$, on parle de $xy - chemin$

Graphe connexe : Un graphe G est connexe si pour tous sommets x et y de G, le graphe G contient un $xy - chemin$.

Graphe sans triangle : En théorie des graphes, un graphe sans triangle est un graphe tel que pour tout $x, y, z \in V(G)$, on a $xy \notin E$ ou $yz \notin E$ ou $zx \notin E$

Clique number (noté $\omega(G)$) : La taille de la plus grande clique de G. C'est à dire le plus grand nombre de sommets de G tous voisins deux à deux.

Degré : Pour $v \in V$, on note $d_G(v) = |\{u \in V | uv \in E\}|$

Les degrés de G :
Le degré min noté $\delta(G)$: est $min\{d_G(x) : x \in V(G)\}$
Le degré max noté $\Delta(G)$: est $max\{d_G(x) : x \in V(G)\}$

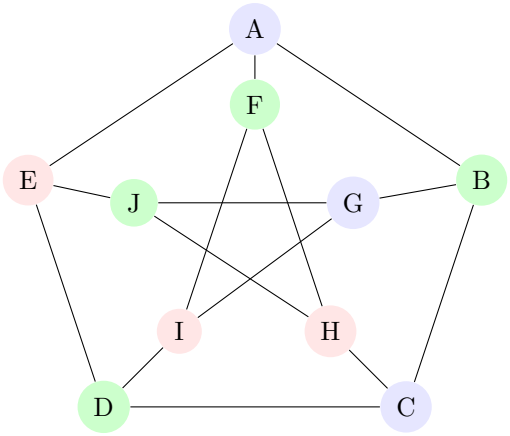
k-coloration d'un graphe G : Une $k - coloration$ de G est une fonction $c : V(G) \to \{1, ..., k\}$ telle que $xy \in E(G) \implies c(x) \neq c(y)$. Si G possède une $k - coloration$, alors G est dit $k - colorable$.

Le nombre chromatique de G noté $\chi(G)$, est $min\{k : G$ admet une $k - coloration\}$.
En pratique, on utilise par exemple la notion de nombre chromatique pour résoudre des problèmes de planification avec des contraintes d'incompatibilités (problème d'emplois du temps).

Stable : Un stable est un ensemble de sommets $I \subseteq V$ tel qu'aucune arête ne relie deux sommets de I.

Stable maximal : I est un stable maximal (par inclusion) si I est un stable et il n'existe aucune stable I' tel que $I \subseteq I'$

Illustrations :



Graphe de Peterson

Ce graphe est connexe sans triangle tels que : $\Delta(P) = 3$, $\chi(G) = 3$ et $\omega(G) = 3$. L'ensemble des sommets d'une même couleur forment des stables. L'ensemble des sommets vert est de plus, un stable maximal.

4 Conjecture de Reed

Déterminer le nombre minimal de couleurs nécessaires pour colorer les sommets d'un graphe G de tel sorte que des sommets adjacents reçoivent des couleurs différentes est un problème NP-complet. Ce nombre est noté $\chi(G)$.

C'est pourquoi, il est utile de chercher des bornes plus faciles à calculer.

Une première borne, inférieure, évidente, est la taille de la plus grande clique contenue dans G, noté $\omega(G)$. Malheureusement déterminer $\omega(G)$ est également un problème NP-complet.

Une borne supérieure assez simple à obtenir est $\Delta(G) + 1$, où $\Delta(G)$ est le dégrée maximum d'un sommet de G. En effet par une approche gloutonne (algorithme first-fit), on obtient une coloration en au plus $\Delta(G) + 1$ couleurs.

Un résultat important sur la coloration, permet d'améliorer cette borne à $\Delta(G)$, c'est le Théorème de Brooks.

La combinaison de ces deux résultats qui a poussé B. Reed à formuler la conjecture suivante :

Conjecture 1. *Tout graphe G vérifie $\chi(G) \leq \lceil \frac{\Delta(G)+\omega(G)+1}{2} \rceil$*

Si on s'attaque à la conjecture en fixant une limite sur $\omega(G)$, le premier cas non trivial est celui des graphes vérifiant $\omega(G) \leq 2$. En effet $\omega(G) \leq 1$ correspond aux indépendants nécessitant au plus une couleur.

L'ensemble des graphes vérifiant $\omega(G) \leq 2$. correspond aux graphes sans triangles (graphes ne contenant pas K_3).

On peut réécrire la conjecture :

Conjecture 2. *Tout graphe sans triangles G vérifie $\chi(G) \leq \lceil \frac{\Delta(G)}{2} \rceil + 2$*

Enfin, on remarque que pour $\Delta(G) \leq 5$, on a $\Delta(G) \leq \lceil \frac{\Delta(G)}{2} \rceil + 2$ puis par le théorème de Brooks, $\chi(G) \leq \lceil \frac{\Delta(G)}{2} \rceil + 2$.
Ainsi, le premier cas non trivial est $\Delta(G) = 6$:

Conjecture 3. *Tout graphe sans triangle G avec $\Delta(G) \leq 6$ vérifie $\chi(G) < 6$*

5 Résumé des articles :

Soit le graphe G de taille de la plus grande clique $\omega(G)$, de nombre chromatique $\chi(G)$ et de plus grand degré $\Delta(G)$.

Acquis : $\omega(G) \leq \Delta(G) + 1$

5.1 Théorème de Erdős :

Pour tout entiers positifs $k, l \geq 3$, il existe un graphe G avec c la longueur du plus court cycle tels que c et $\chi(G) \geq k$

La borne de Vizing : Tout graphe G vérifie $\chi'(G) \leq \Delta(G) + 1$

* Le problème de coloration des arêtes d'un graphe G est équivalent au problème de coloration des sommets de son Line graphe $L(G)$.

* Le degré maximum $\Delta(G)$ d'un graphe, non isomorphe à un triangle, est égal à $\omega(L(G))$. Donc la borne de Vizing de coloration de sommets est : $\chi(L(G)) \leq \omega(L(G)) + 1$.

5.2 Théorème de Brooks :

Si G est un graphe connexe de degré maximum Δ qui n'est pas une clique ou un cycle impair, alors G a une coloration de sommet Δ.
Si on ignore le cas des cycles impairs on obtient le corollaire suivant :
Corollaire : Si un graphe de degré maximum $\Delta \geq 3$ n'a pas de clique contenant plus supérieur à Δ sommets alors il a un nombre chromatique au plus Δ.

D'où la conjecture n°1 de Reed.
En conséquence on obtient les conjectures suivantes :
Conjecture 2 : Pour tout graphe G de degré maximum $\Delta \geq 3$, on a $\chi(G) \leq \frac{2(\Delta+1)}{3} + \frac{\omega}{3}$
Cette conjecture a été prouvée par Brooks pour $\omega = \Delta - 1$.
Conjecture 3 : Il existe une constante a telle que pour tout graphe G, $\chi(G) \leq a\omega(G) + \frac{1}{2}(\Delta(G) + 1)$
Conjecture 4 : Pour tout graphe G, $\chi(G) \leq \frac{\omega(G)+\Delta(G)}{2} + o(\Delta(G))$
Conjecture 5 : Pour tout graphe G, $\chi(G) \leq \frac{\omega(G)+\Delta(G)}{2} + o(\omega(G))$[2]

6 Les algorithmes de coloration

Avant de commencer toute implémentation, il convient d'étudier en détail les algorithmes de coloration que nous aurons à implémenter, ne serait-ce que pour connaître quel genre d'opération nous seront amenés à faire sur les graphes.
Il nous est demandé d'étudier des algorithmes exacts. Nous rappelons que déterminer $\chi(G)$ est NP-complet, et que donc tous les algorithmes connus ont une complexité exponentielle. Néanmoins ce problème a beaucoup intéressé les chercheurs, qui n'ont cessé d'améliorer la complexité asymptotique. On peut d'ailleurs déjà observer un réel écart de temps entre les algorithmes en pratique, même sur des petits graphes. Nous présenterons les algorithmes auxquels nous nous sommes intéressés par ordre décroissant de complexité.

6.1 First-Fit :

L'algorithme First-Fit est la manière gloutonne d'obtenir une coloration. On choisit un sommet sans couleur et on lui assigne la plus petite couleur non encore assignée à l'un de ses voisins, etc.

Algorithm 1 First_Fit(G)

for $v \in V(G)$ **do**
  choisir pour v la plus petite couleur n'apparaissant pas dans $N_G(v)$
end for

Tel quel, l'algorithme ne garantit aucunement que la coloration calculée sera optimale, néanmoins le nombre de couleurs utilisées dépend de l'ordre dans lequel les sommets ont été parcourus.

En fait, il existe toujours une permutation qui donnera le nombre minimal de couleurs. En effet, considérons c^* une coloration qui utilise $\chi(G)$ couleurs, disons que les couleurs sont des entiers de 1 à $\chi(G)$. On considère une permutation des sommets où ceux-ci sont rangés dans l'ordre croissant des couleurs. Si First-fit considère les sommets dans cet ordre, il trouvera une coloration c optimale.

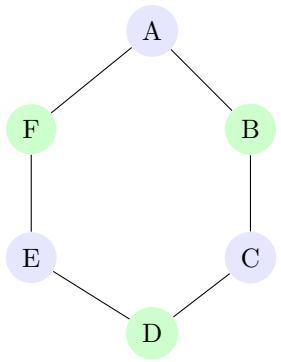
On peut le montrer simplement par récurrence sur les sommets du graphe.
Soit v le sommet choisi par l'algorithme à une certaine étape, tout sommet u tel que $c^*(u) < c^*(v)$ a alors déjà été coloré, supposons que pour ces sommets, on ait $c(u) \leq c^*(u)$. Puisque, $N_G(v)$ ne contient aucun sommet w tel que $c^*(w) = c^*(v)$. $N_G(v)$ peut contenir des sommets tel que $c^*(w) > c^*(v)$ mais ceux ci n'ont pas encore reçu une couleur. Enfin $N_G(v)$ peut contenir des sommets tels que $c^*(w) < c^*(v)$, mais par hypothèse $c(u) \leq c^*(u) < c^*(v)$. Ainsi la couleur $c^*(v)$ est disponible d'où $c(v) \leq c^*(v)$.

Une stratégie pour déterminer $\chi(G)$ est donc d'essayer toutes les permutations et de retenir le nombre de couleurs minimal utilisé.
Complexité :
First-fit a une complexité en $O(m+n)$, et le nombre de permutations des sommets de G est de $n!$. Cette stratégie est donc en $O(n!m)$.
Note :
L'algorithme choisi pour générer les permutations est celui de Steinhaus–Johnson–Trotter. Il est facile d'en faire une implémentation qui évite de devoir garder en mémoire les $n!$ permutations

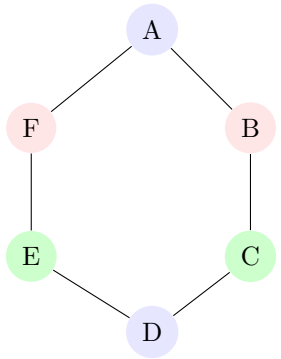
puisque le principe est de passer d'une permutation à l'autre en échangeant la position de deux éléments.[3].

6.1.1 Exemple pour mieux comprendre l'algorithme :

Soit G un graphe (cicle) de taille 6.
Soit l'ordre croissant des couleurs : bleu, vert, rouge.



$\chi(G) = 2$
avec ordre de sommets : $\{A, B, C, D, E, F\}$



$\chi(G) = 3$
avec ordre de sommets : $\{A, D, C, B, E, F\}$

6.2 Algorithme de Lawler :

L'un des premiers à dépasser une approche naïve pour le problème de la coloration est E. Lawler en 1976 [4]. Il obtient ainsi un algorithme en $O(2.44^n)$.

Dans cet algorithme on va indexer toutes les possibilités de sous ensembles de sommets de V, on a donc pour un graphe à n sommets 2^n éléments, qu'on va indexer de sorte que la représentation binaire de chaque index est un vecteur binaire indiquant pour chaque sommet s'il est dans l'ensemble ou pas.
On pose :

$$\chi(G[S]) = \begin{cases} |V| & \text{si } |S|! = 1 \text{ et non vide} \\ 1 & \text{si } |S| = 1 \\ 0 & \text{si } |S| = \emptyset \end{cases}$$

L'algorithme va donc parcourir les sous-ensembles $S \subseteq V$ du graphe, chercher leurs stables et calculer donc leurs nombres chromatiques.

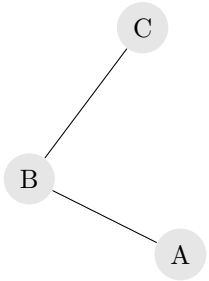
Algorithm 2 Calculer $\chi(G)$

Soit X une liste indexée de 0 à $2^n - 1$
for $S = 0$ à $2^n - 1$ **do**
  for all MISs I dans G[S] **do**
    $X[S] = min(X[S], X[S/I] + 1)$
  end if
end if
renvoyer $X[V]$

Avec $G[S]$ le sous-graphe induits par les sommets de $S \subseteq V$ et I(G) l'ensemble des stables maximaux (par inclusion) de G.

6.2.1 Exemple pour mieux comprendre l'algorithme :

Soit le graphe G suivant :



chercher les sous-graphes S de G :

S en binaire (CBA)	V(S)	initialser X[S]
000		0
001	A	1
010	B	1
011	A,B	3
100	C	1
101	A,C	3
110	B,C	3
111	A,B,C	3

S	I ={ MIS de S }	min(X(S),X(S/I))
A	{A}	1
B	{B}	1
A,B	{{A},{B}}	2
C	{C}	1
A,C	{{A,C},{A},{C}}	1
B,C	{{B},{C}}	2
A,B,C	{{A,C},{B}}	2

6.2.2 La complexité de l'algorithme :

En utilisant le nombre maximal de stables dans un graphe G [5] qui est de : $3^{n/3}$ et le faite qu'ils peuvent être trouvés en un facteur polynomial de cette borne[6], on peut déduire que la complexité de l'algorithme se pose sur la somme des tailles des stables de chaque sous-ensemble S de G :

$$O(\sum_{S\subseteq V}(|I(G[S])|) = O(\sum_{i=0}^{n}\binom{n}{i}3^{i/3}) = O((1+3^{1/3})^n) = O(2.4423^n)$$

6.3 Algorithme de Björklund et Husfeldt

En 2006, Andreas Bjorklund et Thore Husfeld donnent un algorithme en $O(2^n)$ pour le problème du nombre de k-couvertures par ensembles, basé sur le principe d'inclusion-exclusion [7]. Dans cette partie nous présenterons ce problème et sa résolution puis nous verrons comment en déduire un algorithme pour le nombre chromatique qui conserve cette complexité.

6.3.1 Principe d'inclusion-exclusion

Le principe d'inclusion-exclusion permet de calculer le cardinal d'une réunion d'ensemble fini à partir des cardinaux de ces ensembles et de leurs intersections. Il s'agit de généraliser la formule $|A \cup B| = |A| + |B| - |A \cap B|$.

Soit A un ensemble fini et $A_1, A_2, ..., A_n$ des sous parties quelconques de A. On note également $A_0 = A \setminus \bigcup_{i=1}^{n} A_i$. On cherche à exprimer $|A_0|$.
On montre que :

$$|A_0| = |A| + \sum_{k=1}^{n} (-1)^k \sum_{1 \leq i_1 < i_2 < ... < i_k \leq n} |A_{i_1} \cap A_{i_2} \cap ... \cap A_{i_k}|$$

Soit $a \in A$:
— Si $a \in A_0$: a est compté une fois à gauche de l'égalité. De plus $\forall i_1, i_2, ...i_k$ $a \notin A_{i_2} \cap ... \cap A_{i_k}$. Ainsi a est compté une fois également à droite (dans le $|A|$)
— Si $a \in \bigcup_{i=1}^{n} A_i$: soit $I_a = \{i_1, i_2, ..., i_s\}$ tel que $a \in \bigcap_{i \in I_a} A_i$ et $a \notin \bigcup_{i \notin I_a} A_i$
$a \notin A_0$ donc a est compté 0 fois à gauche.
À droite, a est d'abord compté une fois dans $|A|$
Voyons maintenant combien de fois il est compté dans la dernière somme
Soit $\{j_1, j_2, ..., j_r\} \subseteq \{1, 2, ..., n\}$
— si $r > s$
Il existe un j_i tel que $j_i \notin I_a$ donc $a \notin A_{j_i}$ puis $a \notin A_{j_2} \cap ... \cap A_{j_r}$
— si $r \leq s$
$a \in A_{j_2} \cap ... \cap A_{j_r}$ si et seulement si $\{j_1, j_2, ..., j_r\} \subseteq I_a$ a est alors compté une fois avec un facteur $(-1)^r$
Ainsi au k-ème terme de la grande somme, a est compté autant de fois qu'il y a de sous ensemble de I_a de taille k et ce avec un facteur $(-1)^r$, c'est à dire $(-1)^k \binom{r}{k}$

$$\sum_{k=1}^{k} (-1)^k \binom{r}{k} = \sum_{k=0}^{k} (-1)^k \binom{r}{k} - 1$$

$$= \sum_{k=0}^{k} (1)^{r-k} (-1)^k \binom{r}{k} - 1 = (1 + (-1))^r - 1 = -1$$

Au total, a est compté 0 fois à droite.

Si on s'intéresse maintenant au cardinal de l'intersection de $A_1, A_2, ..., A_n$,
pour $W \subseteq \{1, 2, ..., n\}$, on note $N(W) = |A \setminus \bigcup_{i=1}^{n} E_i|$. Posons $E_i = A \setminus A_i, \forall i \in \{0, 1, ..., n\}$ et $X = |A_1 \cap A_2 \cap ... \cap A_n|$ On remarque que $X = A \setminus \bigcup_{i=1}^{n} E_i$, on peut donc appliquer la formule précédente :

$$X = |E_0| = |A| + \sum_{k=1}^{n} (-1)^k \sum_{1 \leq i_1 < i_2 < ... < i_k \leq n} |E_{i_1} \cap E_{i_2} \cap ... \cap E_{i_k}|$$

$$= N(\emptyset) + \sum_{k=1}^{n} (-1)^k \sum_{\substack{W \subseteq \{1,2,...,n\} \\ |W|=k}} N(W)$$

$$= \sum_{W \subseteq \{1,2,...,n\}} (-1)^{|W|} N(W)$$

6.3.2 La k-couverture par ensembles

Étant donné un couple (U, S) ou U est un ensemble et $S \subseteq \mathcal{P}(U)$, une k-couverture pour ce couple est une famille $S_1, S_2, ..., S_k$ d'éléments de S, non nécessairement différents, tel que $S_1 \cup S_2 \cup ... \cup S_k = U$. Notons qu'on peut voir (U, S) comme un hypergraphe.

Soit c_k le nombre de k-couvertures ordonnés que l'on peut construire. "Ordonné" signifie ici que toutes les permutations d'une k-couverture données sont prises en compte. L'algorithme que nous étudierons ici, calcule c_k.
On défini également pour $W \subseteq U$, $s[W] = |\{P \in S | S \cap W = \emptyset\}|$

Pour cela, on va replacer le problème dans le principe d'inclusion-exclusion. L'ensemble que nous étudierons, est celui des k-uplets $O = (S_1, S_2, ..., S_k)$ avec $S_i \in S$. Il faut maintenant définir les propriétés qui nous intéressent : $Q_u(O) \iff u \in \bigcup_{i=1}^{n} S_i$ pour tout $u \in U$. Les O vérifiant $\forall u \in U, Q_u(O)$ correspondent à des k-couvertures. Ainsi d'après le théorème X qui le nombre d'objet vérifiant toutes les propriétés correspond à c_k.

$$c_k = X = \sum_{W \subseteq \{1,2,...,n\}} (-1)^{|W|} N(W) = \sum_{W \subseteq U} (-1)^{|W|} s[W]^k$$

L'algorithme qui en découle consiste à : calculer les 2^n sous ensembles $W \subseteq U$ la valeur de $s[W]$ puis calculer c_k à l'aide de la formule précédente.
Une approche par programmation dynamique permet de calculer toute la valeur de $s[W]$ en temps $O(2^n)$.

Soit χ_S la fonction caractéristique de S :

$$\chi_S : \mathcal{P}(U) \to \{0, 1\}$$

$$s \mapsto \begin{cases} 1 & \text{si } s \in S \\ 0 & \text{sinon.} \end{cases}$$

On peut alors exprimer $s[W]$ ainsi :

$$s[W] = \sum_{s \subseteq U \setminus W} \chi_S(s)$$

Considérons un ordre quelconque des éléments de U : $u_1, u_2, u_3, ..., u_n$ On défini $g_i(W)$ comme étant le nombre de sous-ensembles de S disjoints de W et contenant tous les éléments $u_{i+1}, ..., u_n$
Autrement dit :

$$g_i(W) = \sum_{\substack{s \subseteq U \setminus W \\ s \supseteq \{u_i+1,...,u_n\} \setminus W}} \chi_S(s)$$

Ainsi :

$$g_0(W) = \chi_S(U \setminus W)$$
$$g_n(W) = \sum_{s \subseteq U \setminus W} \chi_S(s) = s[W]$$

Pour mettre en place l'algorithme de programmation dynamique, on observe que :

$$g_i(W) = \begin{cases} g_{i-1}(W) & \text{si } u_i \in W \\ g_{i-1}(W \cup \{u_i\}) + g_{i-1}(W) & \text{sinon.} \end{cases}$$

L'algorithme que l'on en déduit, consiste à : choisir un ordre quelconque des éléments de U puis créer un tableau en deux dimensions donc les colonnes correspondent aux sous ensembles W de U et les lignes correspondent aux i. Ce tableau sera donc de taille $2^n \times n$. Il faudra alors remplir les cases avec les valeurs de $g_i(W)$. On procède ligne par ligne de sorte à pouvoir utiliser la formule récursive donnée plus haut, la première ligne où $i = 0$ sera remplie des $\chi_S(U \setminus W)$

6.3.3 Stratégie pour le nombre chromatique

On peut réduire le problème de la coloration à celui de la k-couverture, étant donné un graphe $G = (V, E)$, on va choisir l'hypergraphe $H = (V, I)$, où I est l'ensemble des indépendants de V, ainsi une k-couverture pour H induit une coloration de G utilisant au plus k couleurs
De plus, le plus petit k tel qu'il existe une k-couverture correspond au plus petit nombre de couleur pour colorer G.
Autrement dit, il nous faut déterminer le plus petit k tel que $c_k > 0$, et nous disposons d'un algorithme en $O(2^n)$ pour calculer c_k.
Les problèmes qu'il reste à résoudre sont la construction de I, l'ensemble des indépendants de G et la stratégie de recherche, on procédera par dichotomie sur les valeurs de k comprises entre 0 et n. Une fois I construit, on peut exprimer la complexité de la recherche par dichotomie avec calcul de c_k à chaque étape comme suit :

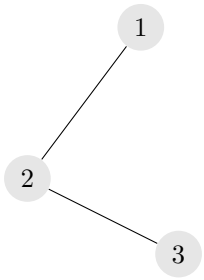
$$\begin{cases} T(m, n) = T(\frac{m}{2}) + O(2^n) \\ T(1, n) = O(2^n) \end{cases}$$

Où m est la taille de l'espace de recherche et $n = |V|$, en fait comme on considère que le nombre chromatique est compris entre 0 et n, au départ $m = n + 1$.

$$T(m, n) = T(\frac{m}{2}) + O(2^n)$$
$$= T(\frac{m}{2^2}) + 2 \times O(2^n)$$
$$...$$
$$= \log_2(m) \times O(2^n)$$

6.3.4 Exemple pour mieux comprendre l'algorithme :

Soit le graphe G suivant :



On cherche I l'ensemble des stables de G : $I = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,3\}\}$

W	$\emptyset$	$\{1\}$	$\{2\}$	$\{3\}$	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$
$g_0(W)$	0	0	1	0	1	1	1	1
$g_1(W)$	0	0	2	1	1	1	2	1
$g_2(W)$	2	1	2	3	1	2	2	1
$s[W] = g_3(W)$	5	3	4	3	2	2	2	1

Il ne reste plus qu'a évaluer la formule d'exclusion exclusion. On procède par dichotomie sur la valeur de k qui est comprise entre 0 et 3.

— $k = 2$, $c_2 = \sum_{W \subseteq U} (-1)^{|W|} s[W]^2 = 25 - 9 - 16 - 9 + 4 + 4 + 4 - 1 = 2$
$c_2 > 0$, on en conclu $\chi(G) \leq 2$.

— $k = 1$, $c_1 = \sum_{W \subseteq U} (-1)^{|W|} s[W] = 5 - 3 - 4 - 3 + 2 + 2 + 2 - 1 = 0$
$c_1 \leq 0$, on en conclu $\chi(G) > 1$.

Ainsi $\chi(G) = 2$

7 Algorithmes de génération des stables d'un graphe

Étant donné un graphe G, $S \subseteq V(G)$ est un stable si $\forall u, v \in S$, $uv \notin E(G)$.
On dit d'un stable qu'il est maximal (au sens de l'inclusion) lorsqu'il n'existe aucun stable $S' \neq S$ de G tel que $S \subseteq S'$.

À plusieurs occasions, dans les algorithmes de coloration et de génération de graphe que nous utilisons, il est nécessaire de pouvoir déterminer tous les stables ou tous les stables maximaux d'un graphe. Cette partie explique les algorithmes qui nous ont permis de faire cela.

7.1 Génération exhaustive des stables d'un graphe

Algorithm 3 Gen_is(G) : Génération exhaustive des stables d'un graphe

```
if G = ({ }, { }) then
  return {{ }}
else
  choisir v ∈ V(G)
  return Gen_is(G \ v) ∪ {v ∪ I | I ∈ Gen_is(G \ (v ∪ N_G(v))}
end if
```

Correction :
On procède par récurrence sur n, le nombre de sommets de G :
On note $IS(G)$, l'ensemble des stables de G Supposons que $IS(G) = Gen\_is(G)$ pour tout graphe à moins de $n-1$ sommets. Il suffit alors de montrer que pour tout $v \in V(G), IS(G) = IS(G \setminus v) \cup \{v \cup I | I \in IS(G) \setminus (v \cup N_G(v)\}$.
Prenons I un stable de $IS(G)$:

- Si $v \notin I$, alors $I \in IS(G \setminus v)$
- Si $v \in I$, alors $I \cap N_G(v) = \emptyset$ puisque I est un stable.
  De plus $I' = I \setminus v \subseteq V(G \setminus (v \cup N_G(v))$. Ainsi $I = v \cup I'$ avec $I' \in IS(G \setminus (v \cup N_G(v)))$

Note :
Même si on l'implémente avec une structure de données qui ne vérifie pas la présence de doublons au moment de l'opération "$\cup$", l'algorithme ne produirait qu'une seule fois chaque stable.

En effet si on suppose qu'un stable I est renvoyé deux fois par l'algorithme, c'est qu'à une étape de la récurrence, une sous partie de I est renvoyé à la fois par l'appel "Gen_is($G \setminus v$)" et par l'appel "$\{v \cup I | I \in$ Gen_is$(G \setminus (v \cup N_G(v))\})$", ce qui est exclu car ces ensembles sont disjoints à cause de v.

Complexité :
Grossièrement, la complexité dans le pire des cas dépend du nombre maximum de stables dans un graphe, ce qui correspond aux indépendants I_n qui ont autant de stables que de sous parties, c'est à dire 2^n. Il faut aussi prendre en compte le coût des opérations ensemblistes qui dépend de l'implémentation.

7.2 Génération exhaustive des stables maximaux d'un graphe

L'algorithme que nous utilisons a été tiré d'une publication de E.Tomita, A.Tanaka et Takahashi.T [8]. Dans cet article, le but était de générer les cliques maximales et non pas les stables. Néanmoins on peut facilement adapter l'algorithme pour chercher des stables.
En effet, les deux problèmes sont équivalents : chercher un stable dans un graphe revient à chercher une clique dans son complémentaire.

Une approche classique pour ce problème est de procéder de manière récursive : étant donné Q le stable qu'on est en train de construire, on choisit les sommets suivants parmi $SUBG = V(G) \setminus (Q \cup N_G(Q)$, jusqu'à ce que cet ensemble de candidats soit vide.

Mais si on applique cette méthode tel quel, on obtient des doublons. C'est pour cela que les auteurs introduisent deux principes qui vont améliorer cette procédure :

— Supposons que u, un sommet de $SUBG$ ai déjà été choisi, tous les stable contenant $Q \cup u$ ont alors déjà été générés. Ainsi en continuant à explorer les stables contenant Q, il n'y aura plus besoin de choisir u.

— Supposons que tous les stables contenant $Q \cup u$ aient déjà été générés. Alors les stable maximaux non encore explorés contenant Q, contient au moins un voisin de u. En effet supposons qu'après plusieurs étapes on arrive à un stable $Q \cup R$ avec $R \cap N_g(u) = \emptyset$ alors $Q \cup R \cup u$ est un stable plus grand, $Q \cup R$ n'est donc pas maximal.
De plus, on a alors intérêt, lorsqu'on choisit le prochain sommet u à explorer, à prendre celui qui maximise $|SUBG \setminus \{u\}|$ pour réduire par la suite l'ensemble des candidats.

Algorithm 4 Gen_mis(G) : Génération exhaustive des stables maximaux d'un graphe

$Q \leftarrow \emptyset$
EXPAND($V(G)$, $V(G)$)

Algorithm 5 EXPAND(SUBG, CAND)

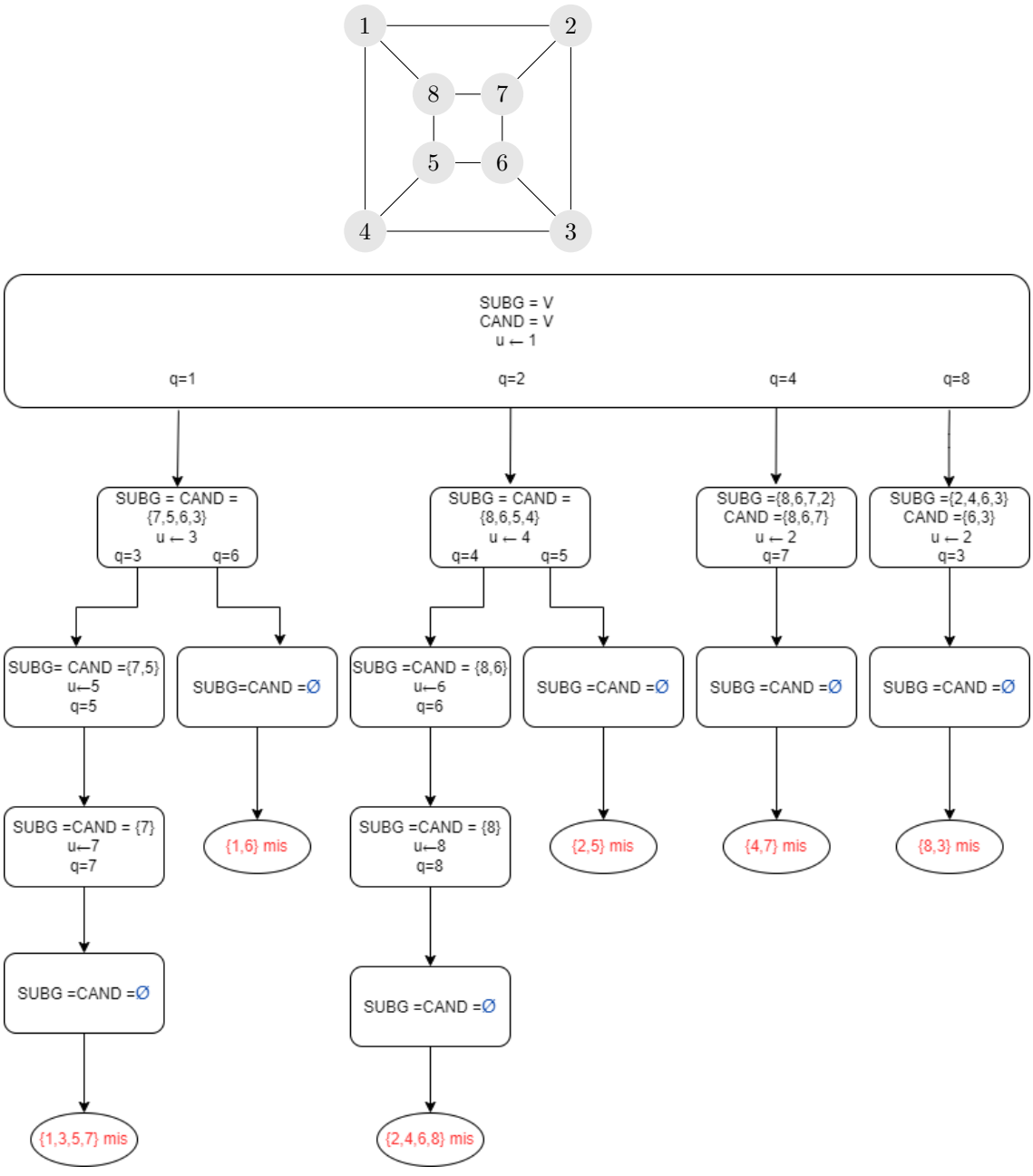
if $SUBG = \emptyset$ **then**
  Q est un stable maximal
else
  choisir u un sommet de $SUBG$ qui maximise $|CAND \setminus (N_G(u) \cup \{u\})|$
  while $CAND \cap (N_G(u) \cup \{u\}) \neq \emptyset$ **do**
    choisir q parmi $CAND \cap (N_G(u) \cup \{u\})$
    $Q \leftarrow Q \cup \{q\}$
    $SUBG_q \leftarrow SUBG \setminus (N_G(q) \cup \{q\})$
    $CAND_q \leftarrow CAND \setminus (N_G(q) \cup \{q\})$
    $EXPAND(SUBG_q, CAND_q)$
    $CAND \leftarrow CAND \setminus \{q\}$
    $Q \leftarrow Q \setminus \{q\}$
  end while
end if

Complexité :
L'auteur original montre que l'algorithme a une complexité en $O(3^{n/3})$, ce qui est optimal puisque un graphe peut contenir jusqu'à $3^{n/3}$ stables maximaux.

Note :
Là aussi, l'algorithme ne génère qu'une seule fois chaque stable.

7.2.1 Exemple pour mieux comprendre l'algorithme :

Soit le graphe $G = (V, E)$ suivant :



8 Algorithmes de génération exhaustive de graphes

Une fois les algorithmes de coloration implémentés et testés, la deuxième partie du travail consiste à générer les graphes que l'on veut tester. L'enjeu est réduire le nombre d'utilisation des algorithmes de coloration, pour cela, on peut chercher à éliminer rapidement des graphes pour lesquels il n'est pas nécessaire de faire de tests. La difficulté principale est la génération de graphes isomorphes, nous détaillerons ce point dans la suite.

8.1 Génération des graphes pour la première conjecture

8.1.1 1 ère Approche : Odomètre

l'approche directe consiste à partir de K_n, le graphe complet à n sommet, tout graphe à n sommet peut être obtenu en retirant des sommets à K_n. Ainsi il suffit de considère l'ensemble des sous parties des arêtes de K_n. Comme il a $\frac{n(n-1)}{2}$ arêtes dans le graphe complet, Cette opération produit donc $2^{\frac{n(n-1)}{2}}$ graphes.

C'est une approche très coûteuse puisque on obtient beaucoup de graphes isomorphes. C'est à dire de graphes identique à renommage des sommets près. Plus rigoureusement deux graphes G et H sont isomorphes s'il existe une bijection $f : V(G) \leftarrow V(H)$, telle que $uv \in E(G) \iff f(u)f(v) \in E(H)$.
Si G et H sont isomorphes, il est inutile de tester la conjecture sur les deux graphes puisqu'ils ont le même nombre chromatique, même clique number et même degré maximum. Malheureusement la détection d'isomorphismes est aussi coûteuse, il s'agit d'un problème NP-Complet. Ainsi il serait vain de chercher à éliminer les graphes générés isomorphes.

L'implémention utilise le principe de l'odomètre. C'est à dire que pour générer tous les ensembles d'arêtes possible, on utilise un tableau de bits de taille $\frac{(n-1)}{2}$ ou variable entière dont les bits vont indiquer quelle arrête sera conservée. Ainsi il suffit de parcourir toute la valeur entre 0 (correspondant à I_n) et $2^n - 1$ (correspondant à K_n) pour énumérer une seule fois chaque ensemble d'arêtes.

8.1.2 2 ème Approche : Graphes connexes

Une première approche pour gagner du temps consiste à remarquer qu'il n'est pas nécessaire de tester la conjecture sur les graphes non connexes.

En effet, soit G un graphe, à n sommets, non connexe avec $C_1, C_2, ..., C_k$ ses composantes connexes.
De manière évidente on a :

$$\chi(G) = max(\chi(C_1), \chi(C_2), ..., \chi(C_k))$$

$$\omega(G) = max(\omega(C_1), \omega(C_2), ..., \omega(C_k))$$

$$\Delta(G) = max(\Delta(C_1), \Delta(C_2), ..., \Delta(C_k))$$

Des plus les tailles des C_i sont toutes inférieures strictement à n.
Supposons maintenant que la conjecture ait été prouvée pour tous les graphes avec un nombre de sommets inférieur strictement à n.
Soit $C_{i_{MAX}}$ tel que $\chi(G) = \chi(C_{i_{MAX}})$. On a alors :

$$\chi(G) = \chi(C_{i_{MAX}}) \leq \lceil \frac{\Delta(C_{i_{MAX}}) + \omega(C_{i_{MAX}}) + 1}{2} \rceil \leq \lceil \frac{\Delta(G) + \omega(G) + 1}{2} \rceil$$

Ainsi si la conjecture est vérifiée pour tous les graphes connexes, aucun contre exemple ne pourra être trouvé parmi les graphes non connexes.

Algorithm 6 Gen_connected(n) : Génération exhaustive des graphes connexes de taille n

```
if n = 1 then
  {le seul graphe à renvoyer est le graphe à un seul sommet}
  return {I_1}
else
  graphs ← ∅
  Smaller = Gen_connected(n − 1)
  for all graph G in Smaller do
    for all S in 𝒫(V(G)) do
      graphs = graphs ∪ { (V(G) ∪ {v}, E(G) ∪ {uv | u ∈ S}) }
      {v est un nouveau sommet}
    end for
  end for
  return graphs
end if
```

Correction :

Lemme 1. *Tout graphe connexe G contient un sommet u tel que $G \setminus u$ est un graphe connexe.*

Démonstration. Supposons par l'absurde que G est un graphe connexe, avec pour tout $u \in G$, $G \setminus u$ n'est pas connexe.
Soit $x \in G$, notons C une composante connexe engendrée par la suppression de x. Comme G est connexe, x a au moins un voisin dans C, notons le c_1, celui-ci doit avoir un autre voisin, sinon $G \setminus c_1$ serait connexe. On en déduit un nouveau sommet c_2 voisin de c_1, aussi $c_2 \in C$ puisque

C est une composante connexe. On peut ainsi répéter ce raisonnement une infinité de fois : $(x, c_1, c_2, ..., c_i)$ forme une chemin dans C, si c_i n'a des voisins que dans $\{x, c_1, ..., c_{i-1}\}$, alors $G \setminus c_i$ reste connexe. On en déduit un sommet c_{i+1}. Ainsi C contient une infinité de sommets, ce qui est absurde. □

On note $Connected(n)$ l'ensemble des graphes connexes à n sommets.
On montre facilement par récurrence que $Gen\_connected(n) = Connected(n)$:

— $Gen\_connected(n) \subseteq Connected(n)$, par construction.

— $Connected(n) \subseteq Gen\_connected(n)$, on suppose : $Connected(n-1) \subseteq Gen\_connected(n-1)$
Soit $G \in Connected(n)$ et $v \in V(G)$ tel que $G \setminus v$ est connexe (le lemme précédant garanti l'existance d'un tel sommet).
$G \setminus v \in Gen\_connected(n-1)$ donc $G \in Gen\_connected(n)$

Note : L'algorithme produit encore beaucoup de graphes isomorphes.

8.2 Génération des graphes sans triangle pour la seconde conjecture

Pour vérifier la deuxième conjecture de Reed, on a besoin de générer les graphes sans triangles. Pour cela on a exploré plusieurs pistes pour avoir de meilleurs résultats.

8.2.1 1 ère Approche : tous les graphes sans triangle (tf)

La première approche consistera à générer tous les graphes sans triangle, sans essayer d'eviter les isomorphismes.

Construction :
On part d'un graphe G de taille $n-1$ et on construit plusieurs graphes tf de taille $n+1$. Soit $I(G)$ l'ensemble des stables de G, et v un nouveau sommet qu'on ajoute à $V(G)$, soit s un stable de $I(G)$, en relie v au sommets de s de toutes les manières possibles.

Correction :
Soit G' un graphe de taille n obtenue par cette construction, vérifions que G' est un graphe sans triangle.
Soit v le nouveau sommet ajouté par l'algorithme au graphe G tf de taille $n-1$ pour obtenir G' et soit u_1 et u_2 deux sommets dans $V(G') \setminus \{v\} = V(G)$ tel que u_1 et u_2 sont voisins de v par construction u_1 et u_2 appartiennent a un stable de G donc $\{v, u_1, u_2\}$ n'est pas un triangle. Donc tout graphe G de taille n obtenue par cette construction, est bien un graphe sans triangle. Maintenant dans l'autre sens. Soit G un graphe de taille n sans triangle, soit $u \in V(G)$, c'est évident de voir que le graphe $G' = (V(G) \setminus \{u\}, E(G) \setminus \{vu | v \in V(G)\})$ est un graphe de taille $n-1$ sans triangle.

La conjecture est vérifiée pour les graphes de taille au plus 6, malheureusement avec notre implémentation, on ne peut tester la conjecture sur les graphes de taille strictement supérieur à 7 car il génère beaucoup de graphes isomorphes, ce qui fait donc augmenter la complexité spatiale et temporelle et en conséquent fait planter le programme.

8.2.2 2 ème Approche : tf amélioré

Pour améliorer la performance de notre algorithme on va restreindre nos générations de graphes en deux étapes :

Etape 1 : On se limite aux graphes de degré maximum au plus 6, et chercher s'il existe un tel graphe 6-chromatique.

Construction similaire à celle de la génération des graphes sans triangle en rajoutant une condition sur les stables parcourus comme expliqué dans l'algorithme suivant :

Algorithm 7 $Gen\_tf(n)$: Génération exhaustive de graphes tf de degré max au plus 6

```
graphs ← ∅
if n = 0 then
  return [empty graph]
else
  smaller = Gen_tf(n − 1)
  for all graph G in smaller do
    for all is in I(G) do
      if |is| ≤ 6 & Δ_G(is) ≤ 5 then
        let be G' = (∅, ∅)
        V(G') = V(G) ∪ {v}
        E(G') = E(G) ∪ {e_i v} with e_i ∈ V(is) , i ∈ {1, .., |is|}
        graphs = graphs ∪ G'
      end if
    end for
  end for
  return graphs
end if
```

correction :

Soit le graphe G obtenue par l'algorithme 7, montrons que G est bien un graphe tf de degré max au plus 6. Soit G' le graphe de taile $n-1$ et de degré max au plus 6 auquel on a ajouté un sommet notons le v, pour obtenir G de taille n. Le sommet v est relié aux stables de taille au plus 6, donc v aura au plus 6 voisins, ces stables sont par construction de degré max 5 donc en leur rajoutant le sommet v, on dépasse pas le degré 6. Donc le graphe G obtenu est bien un graphe tf de taille n et de degré max 6.

Maintenant dans l'autre sens, montrons que tout graphe tf de degré max 6 est généré par l'algorithme 7. Soit $G = (V, E)$ un graphe tf de taille n de degré max au plus 6, soit v un sommet de degré max de G (s'il y a plusieurs sommet qui ont le même degré max on choisit au hasard un d'eux). Soit G' un graphe tel que $G' = (V', E')$ avec $V' = V \setminus \{v\}$ et $E' = E \setminus \{uv | u \in E\}$ c'est facile de voir que G' est un graphe tf de degré max au plus 6, et est de taille $n-1$.

Etape 2 : Générer que les graphes tf connexe de degré maximum 6.

Dans le but de plus affiner les générations de graphes, et comme déjà prouvé dans la partie 8.1.2 qu'il n'est pas nécessaire de tester la conjecture sur les graphes non connexes. Dans cette étape on va garder la même construction que celle de l'algorithme 7 et on rajouter quelque

conditions pour ne générer que les graphes connexes tf de taille n et de degré max au plus 6.

Construction : Dans cette étape on rajoute à l'algorithme 7 une condition sur la taille des ensembles stables parcourus de tel sorte qu'on évite de parcourir les stables vides, et d'obtenir par conséquent des graphes non connexes.

Algorithm 8 $Gen\_tf\_connected(n)$: Génération exhaustive de graphes connexes tf avec $\Delta \leq 6$

```
graphs ← ∅
if n = 0 then
  return [empty graph]
else if n=1 then
  return [({A}, ∅)]
else
  smaller = Gen_tf_conncted(n − 1)
  for all graph G in smaller do
    for all is in I(G) do
      if |is| ≤ 6 & Δ_G(is) ≤ 5 & |is| > 0 then
        let be G' = (∅, ∅)
        V(G') = V(G) ∪ {v}
        E(G') = E(G) ∪ {e_i v} with e_i ∈ V(is) , i ∈ {1, .., |is|}
        graphs = graphs ∪ G'
      end if
    end for
  end for
  return graphs
end if
```

Correction :

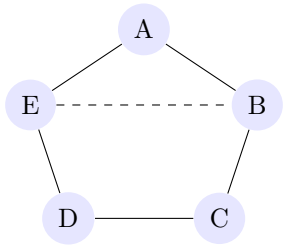
Soit $G = (V, E)$ un graphe généré par l'algorithme 8, montrons que G est un graphe de taille n tf connexe de degré max au plus 6. Soit G' un graphe connexe tf de taille $n-1$, soit v un sommet qu'on ajoute à $V(G')$ pour obtenir G et qu'on relie au stables non vide, donc dans tout les cas v sera relié à au moins un sommet de G' donc G est aussi connexe tf de taille n et de degré max au plus 6.

Soit G un graphe connexe tf de taille n et de degré max au plus 6, montrons qu'on peut obtenir G par l'algorithme 8, ce qui est évident en utilisant le $lemme1$ de la partie 8.1.2 : il existe un sommet v tel que $G \setminus v$ est connexe.

Résultat : Maintenant on arrive a générer les graphes connexes tf de degré max 6 de taille au plus 8 en temps 400 secondes avec 293988 graphes non mtf évités de base 352630 graphes générés par l'algorithme grâce à une heuristique expliquée dans la partie optimisation qui suit.

8.2.3 3 ème Approche : Graphes maximaux sans triangles (mtf)

Tout d'abord un graphe mtf est un graphe tf, tel que l'ajoute d'une arête engendre forcement un triangle.



Le cycle de longueur 5 est un graphe maximal sans triangle, l'ajout d'un arête fait systématiquement apparaître un triangle.

Remarque 1. *Tout graphe G tf, peut être transformé en un graphe G' mtf par ajout d'arêtes. Aussi $\chi(G') \geq \chi(G)$ et $\Delta(G') \geq \Delta(G)$.*

Ainsi, si tous les graphes mtf, vérifient la conjecture, cela implique que tous les graphes tf la vérifient également. On peut donc se restreindre à tester uniquement les mtf.
Cette approche pourrais nous permettre de réduire considérablement le nombre de graphes. Par exemple, pour les graphes à 13 sommets, il existe plus de 20 millions de tf non-isomorphes, et seulement 392 mtf non-isomorphes. Néamoins, le nombre de mtf, reste exponentiel en fonction du nombre de sommets [9].

1ère idée de construction :
Soit G un graphe maximal sans triangle de taille $n-1$, soit $IM(G)$ l'ensemble de tous les stables maximaux de G, soit $I \in IM(G)$, on ajoute un nouveau sommet v à $V(G)$ et on le relie à tout sommets de I.

Tout graphe G' obtenu par cette construction vérifie les propriétés suivantes :
sans triangle : car G est sans triangle et on rajoutant des arêtes pour obtenir G' on risque pas d'avoir des triangles car les voisins de v forment un stable et donc pas reliés.
maximal : car si on rajoute une arête à G' on a deux possibilités :
- l'arête est entre v et un sommet u dans $V(G') \setminus I$, on obtient alors un triangle car u est voisin d'un sommet de I, sinon I ne serait pas maximal.
- l'arête est entre deux sommets x et y dans $V(G') \setminus \{v\} = V(G)$ donc on obtient forcement un triangle car G est maximal sans triangle.

Maintenant l'autre sens, supposons que tout graphe mtf est généré par cette construction, contre exemple : Soit le graphe G (figure 1) maximal sans triangle de taille 4.

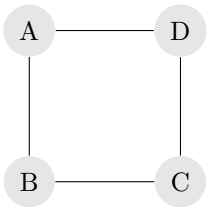


figure 1

En suivant notre construction on obtient les deux graphes (figure 2) maximaux sans triangle de taille 5.

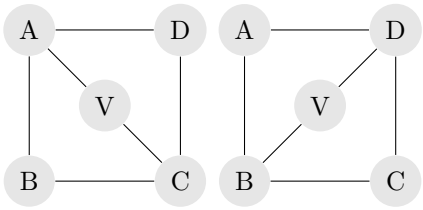


figure 2

Or il existe un 3ème graphe (figure 3) maximal sans triangle de taille 5.

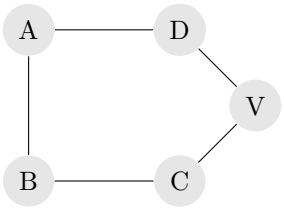


figure 3

Cette construction reposant sur un principe analogue à celles utilisé jusqu'à présent ne suffit pas, puisqu'elle ne permet pas de générer les mtf de manière exhaustive.

8.2.4 Des pistes pour l'approche par mtf

Des publications ont déjà vu le jour concernant la génération de mtf, notamment l'article de Jan Goedgebeur [10] et de Brandt, Brinkmann et Harmuth [9] qui proposent un algorithme prometteur mais que nous n'avons pas pu étudier en détail, car cela aurait demandé beaucoup de travail supplémentaire de comprendre les nombreuses heuristiques utilisés.

9 Optimisations

9.1 Une heuristique pour éviter le calcul du nombre chromatique

Lorsqu'on vérifie les conjecture 2 et 3, nous générons tous les graphes sans triangles. Néanmoins il serait suffisant de vérifier la conjecture uniquement pour les mtf. À défaut d'avoir un algorithme efficace pour générer uniquement les mtf, on peut chercher un moyen rapide de détecter qu'un graphe n'est pas mtf.

Pour un triangle à plus de 3 sommets, être mtf est équivalent à être sans triangle et avoir un diamètre inférieur ou égal à 2 [10]. Le diamètre d'un graphe est la longueur de la plus longue distance (en nombre d'arêtes) entre deux sommets du graphe. La distance entre deux sommets est la longueur du plus court chemin entre ces sommets. Ainsi, pour tout sommet d'un graphe sans triangle, si on explore $N_G(u)$ et $N_G(N_G(u))$, on aura parcouru tous les sommets.

$$\text{G est un mtf} \implies \forall u \in V(G), N_G(N_G(u)) \cup N_G(u) = V(G)$$

L'optimisation que nous en avons tiré consiste à vérifier cette propriété sur un seul sommet de chaque graphe généré, si ce test échoue, le graphe n'est pas mtf, il n'y a pas besoin de calculer son nombre chromatique. Cette vérification étant beaucoup moins coûteuse que la calcul du nombre chromatique le temps gagné est assez important. De plus, en pratique le test échoue très souvent, le calcul de $\chi(G)$ est ainsi évité dans environ 70% des cas.

10 Implémentation

Nous avons choisi d'écrire notre implémentation en C. Un choix majeur auquel nous avons été confrontés a été celui des structures de données utilisé. Il s'agit d'un décision importante car elle détermine la vitesse à laquelle les opérations pourront être effectuées.
Ainsi pour pouvoir accéder rapidement au voisinage d'un sommet, nous avons opté pour un représentation des graphes en listes d'adjacence.

Ainsi un graphe est une liste de sommets, représentés par un identifiant et une liste d'adjacence (liste de sommets voisins). À fin d'accélérer les opérations de suppression d'un sommet et d'exploration d'un voisinage, la liste de voisinage d'un sommet v contient des références sur : l'entrée de chaque voisin dans la liste de sommets, ainsi qu'une référence sur l'entrée de v dans la liste d'adjacence du voisin (voir en annexe un schéma de notre structure de données).

Nous savions que notre implémentation serait destinée uniquement à des petits graphes (pas plus d'une trentaine de sommets) et que nous pourrions être amenés à devoir garder en mémoire un grand nombre de graphes (pour la génération récursive de graphes). Nous en avons tenu compte au moment de choisir les types utilisés pour réduire l'espace mémoire nécessaire. Ainsi nous avons choisi d'utiliser le plus petit type disponible en c *unsigned char* pour les identifiants de sommets. Ce qui fixe une première limite de sommets par graphe à 256. C'est encore bien plus que nécessaire.

Nous voulons également pouvoir représenter un sous graphe à l'aide d'un tableau de bit (chaque bit indiquant si la sommet correspondant à sa position est retenu), le plus commode et rapide et d'utiliser pour cela une variable d'un type entier, on retiendra pour ça le type *unsigned int*, type entier sur 32 bits, ce qui fixe à 32 le nombre maximum de sommets.

11 Résultats

Dans cette partie on va mettre en pratique ce dont on a traité dans le rapport. On va donc exécuter notre code. Pour faire le test on renvoie "Verified" si tous les graphes de degré au plus n (en affichant le n) satisfont la conjecture, "Counter-example" sinon.

11.1 Conjecture de B.Reed N°1 :

La conjecture de B.Reed est vérifiée pour tous les graphes de taille au plus 6 comme on peut le voir dans les captures qui suivent :

Odomètre :

```
Verified for size = 0 (skiped clique for 1/1)
Verified for size = 1 (skiped clique for 1/1)
Verified for size = 2 (skiped clique for 1/2)
Verified for size = 3 (skiped clique for 4/8)
Verified for size = 4 (skiped clique for 32/64)
Verified for size = 5 (skiped clique for 616/1024)
Verified for size = 6 (skiped clique for 24844/32768)

[Done] exited with code=0 in 22.461 seconds
```

exécution du fichier *Conjecture*1.*c*

Graphe connexe :

```
Verified for size = 0 (skiped clique for 1/1)
Verified for size = 1 (skiped clique for 1/1)
Verified for size = 2 (skiped clique for 0/1)
Verified for size = 3 (skiped clique for 2/3)
Verified for size = 4 (skiped clique for 8/21)
Verified for size = 5 (skiped clique for 159/315)
Verified for size = 6 (skiped clique for 6018/9765)

[Done] exited with code=0 in 4.879 seconds
```

exécution du fichier *Conjecture*1*Connected.c*

11.2 Conjecture de B.Reed N°2 :

La conjecture de B.Reed est vérifiée pour les graphes de taille au plus 8 comme on peut le voir dans les captures qui suivent :

Graphe sans triangle :

```
skipped 110818/133501
[Done] exited with code=0 in 23.075 seconds
```

exécution du fichier *test_cnj*2.*c*

Graphe connexe sans triangle de degré max 6 avec heuristique d'optimisation pour $n = 8$:

```
skiped 293988/352630
[Done] exited with code=0 in 395.485 seconds
```

exécution du fichier *test_cnj2_connected.c*

La conjecture est bien vérifiée jusqu'à $n = 8$, reste à améliorer l'algorithme.

12 Difficultés rencontrées

La première difficulté a été l'implémentation, nous avons choisi le langage *C* dans l'espoir d'obtenir un programme plus rapide. Mais étant plus habitués à la programmation orientée objet, nous avons souvent hésité au moment de prendre des décisions concernant l'organisation du code.

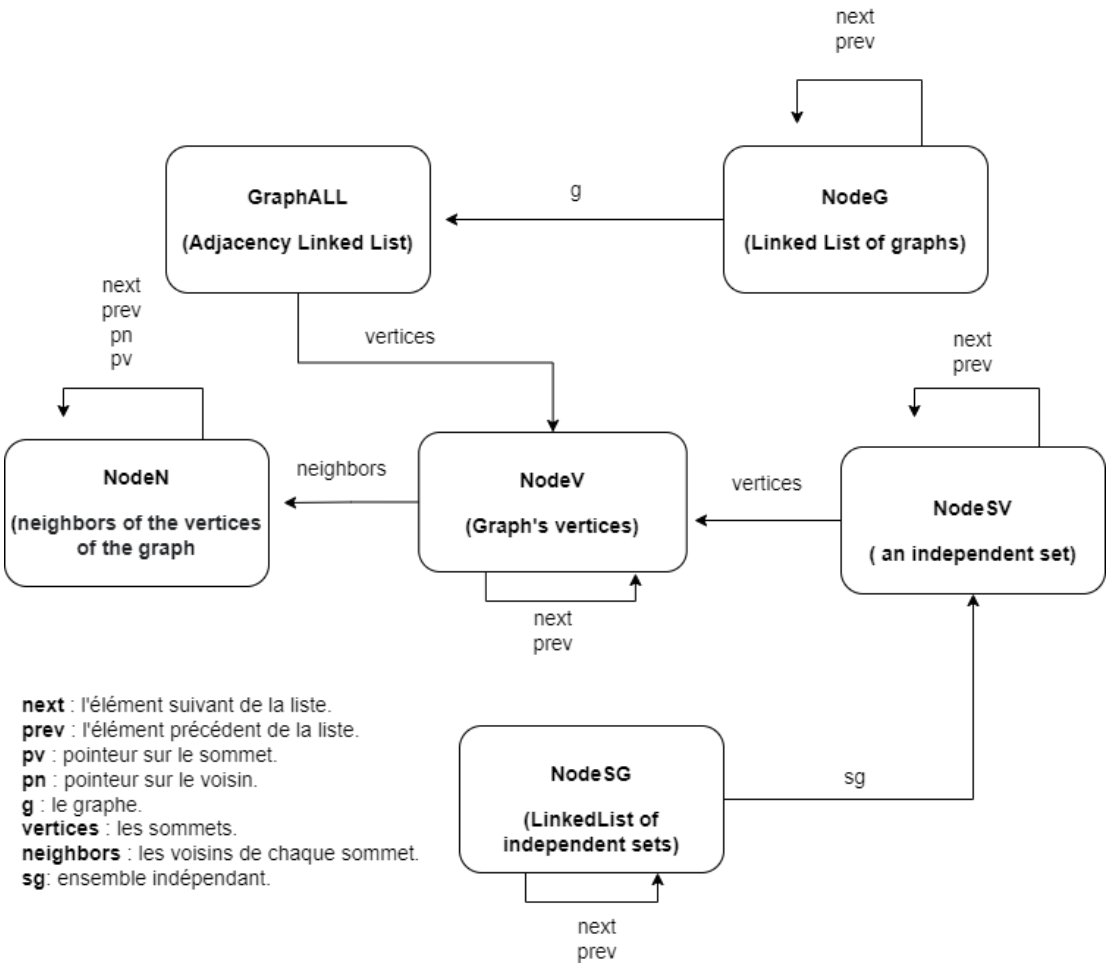
Nous n'avons pas réussi à vérifier la conjecture pour des nombre de sommets aussi élevés que nous l'aurions espéré. Le raison en est le nombre de graphes générés. Les algorithmes de génération que nous avons implémenté sont certes exhaustifs mais ils génèrent un grande quantité de doublons (graphes isomorphes) que nous ne pouvons pas détecter rapidement. Pour espérer améliorer notre programme, il faudrait réduire le nombre de cas à traiter, en fixant certaines arêtes ou en cherchant du côté des mtf.

Nous nous sommes plus concentrés sur la deuxième conjecture puisque la génération de graphes sans triangles est un problème plus spécifique et auquel on avait déjà réfléchi avant notre changement de sujet.

Nous n'avons pas trouvé le temps d'implémenter l'algorithme de Bjorklund et Husfeld, mais ce n'était pas une priorité puisque nous étions satisfaits des temps de calcul obtenus avec l'algorithme de Lawler.

13 Annexe

13.1 Structures des données



Références

[1] Stéphane Bessy. Cours de graphes et structures master 1.

[2] B. Reed. ω, δ, and χ. *Journal of Graph Theory*, 27(4) :177–212, 1998.

[3] Wikipedia contributors. Steinhaus–johnson–trotter algorithm — Wikipedia, the free encyclopedia, 2022. [Online ; accessed 21-May-2022].

[4] Eugene L. Lawler. A note on the complexity of the chromatic number problem. *Inf. Process. Lett.*, 5 :66–67, 1976.

[5] Jesper Byskov. Chromatic number in time $o(2.4023^n)$ using maximal independent sets. *BRICS Report Series*, 9, 12 2002.

[6] John W. Moon and Leo Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3 :23–28, 1965.

[7] Dieter Kratsch Fedor V. Fomin. *Exact Exponential Algorithms*. Springer Berlin, Heidelberg.

[8] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1) :28–42, 2006. Computing and Combinatorics.

[9] Stephan Brandt, Gunnar Brinkmann, and Thomas Harmuth. The generation of maximal triangle-free graphs. *Graphs and Combinatorics*, 16 :149–157, 01 2000.

[10] Jan Goedgebeur. On minimal triangle-free 6-chromatic graphs. *Journal of Graph Theory*, 93, 07 2017.