Ecole Doctorale IAEM Lorraine
DEA Informatique de Lorraine
Filière **A**lgorithmique **N**umérique et **S**ymbolique
Université Paul Verlaine - Metz
Année académique 2004-2005

Algorithmes exponentiels

Rapport de stage

Serge Gaspers
encadré par Dieter Kratsch

Vandoeuvre-les-Nancy, 28 juin 2005

Membres du Jury :
Noëlle Carbonell (membre permanent)
Dominique Méry (membre permanent et responsable de DEA)
Didier Galmiche (membre permanent)
Dieter Kratsch (encadrant et représentant filière ANS)

Remerciements

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

Je remercie aussi les deux thésards de M. Kratsch, Michael Rao et Mathieu Liedloff, pour les discussions inspirantes sur des thèmes proches des algorithmes exponentiels et surtout Mathieu pour avoir relu mon rapport. Un grand merci aussi à tous ceux qui vont participer à ma soutenance blanche pour m'aider à améliorer ma présentation.

Mes remerciements vont aussi à mes collègues de bureau pour la bonne ambiance. Pour la même raison, je dis merci à tout mon entourage proche, mes amis et ma famille.

Je voudrais aussi remercier 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 28 juin 2005.

Table des matières

1 Introduction

La résolution exacte de problèmes algorithmiques est non seulement désirable, mais parfois aussi nécessaire. Pour les problèmes NP-difficiles[1], aucun algorithme polynomial n'est connu pour les résoudre avec exactitude. Comme la plupart des experts pensent que la conjecture $P \neq NP$ est vraie, il semble improbable qu'il existe des algorithmes polynomiaux pour ces problèmes. Le domaine des algorithmes exponentiels (parfois aussi appelés algorithmes exacts) traite des algorithmes pour résoudre exactement des problèmes NP-difficiles avec une complexité en temps qui est certes exponentielle, mais meilleure que la complexité triviale.

Contrairement aux méthodes heuristiques, on essaye ici d'améliorer la borne supérieure du temps d'exécution au pire des cas. Considérons, par exemple, le problème Ensemble Stable (Maximum Independent Set).

Problème : Maximum Independent Set
Entrée : Un graphe non-orienté $G = (V, E)$
Sortie : Un ensemble stable de taille maximale. Un sous-ensemble de sommets $I \subseteq V$ est un ensemble stable s'il n'y a pas d'arête entre deux sommets de I.

Comme pour tous les "problèmes de sous-ensemble" NP-difficiles, il est facile de trouver un algorithme trivial en $O^*(2^n)$ en testant tous les sous-ensembles de sommets et en retenant un sous-ensemble de taille maximale qui satisfait la propriété recherchée (ici : la stabilité)[2]. Dans [TT77], Tarjan et Trojanowski présentent un algorithme en $O^*(2^{n/3}) = O^*(1.3803^n)$ pour résoudre ce problème. Ça veut dire qu'avec cet algorithme, ils arrivent à traiter des graphes trois fois plus grands qu'avec l'algorithme trivial en un *temps raisonnable*. En 2001, Robson [Rob01] présente un algorithme en $O^*(1.1889^n)$, permettant de traiter des instances quatre fois plus grandes que l'algorithme trivial.

1.1 Motivations

Chaque problème NP-difficile est résoluble par une recherche exhaustive qui fournit des temps d'exécutions triviaux, par exemple $O^*(2^n)$ pour des problèmes de sous-ensemble ou $O^*(n!)$ pour des problèmes de permutation. Un problème de sous-ensemble a comme

[1] Un problème d'optimisation est NP-difficile si le problème de décision correspondant est NP-complet.
[2] n est la taille de l'entrée, par exemple le nombre de sommets d'un graphe. La notation O^* sera expliquée dans 1.3.

solution une partie de l'entrée et un problème de permutation une permutation (d'une partie) de l'entrée. Pour quelques problèmes NP-difficiles, il est néanmoins possible de concevoir des algorithmes significativement plus rapides que l'algorithme trivial, même s'ils ne sont pas polynomiaux. L'intérêt croissant au cours de la dernière décennie pour les algorithmes exponentiels a beaucoup de raisons.

De nos jours, les experts de la complexité pensent que $P \neq NP$, c'est-à-dire qu'il n'est pas possible de trouver des algorithmes polynomiaux pour résoudre des problèmes NP-difficiles exactement. Le meilleur qu'on peut donc espérer, c'est des algorithmes super-polynomiaux rapides. Et par l'expérience, on sait que les algorithmes exponentiels rapides peuvent effectivement donner des algorithmes faisables en pratique, au moins pour des entrées pas trop grandes.

Opérations	2^{30}	2^{36}	2^{42}	2^{48}	2^{54}
Temps	1 sec.	1 min.	1 heure	3 jours	$>$ 6 mois
1.05^n	426	511	596	681	767
1.1^n	218	262	305	349	392
1.2^n	114	136	159	182	205
1.3^n	79	95	111	127	142
1.4^n	62	74	86	99	111
1.5^n	51	61	72	82	92
2^n	30	36	42	48	54
3^n	19	23	26	30	34
$n!$	12	14	15	17	18

TAB. 1.1: taille maximale de l'entrée pour un temps d'exécution donné

Concevoir un algorithme en $O^*(c^n)$ avec une petite constante c améliore vraiment le temps d'exécution en pratique. La table 1.1 montre les tailles des entrées qui sont encore faisables par un algorithme exponentiel en fonction de sa complexité. Ici, on suppose qu'un ordinateur moderne arrive à traiter 2^{30} opérations par seconde et on se demande pour quelles tailles de l'entrée le temps d'exécution de l'algorithme reste raisonnable. En se basant sur cette table, on remarque qu'un algorithme en $O^*(2^n)$ utilise plus de 3 jours pour une entrée de taille 50 environ, tandis qu'un algorithme en $O^*(1.5^n)$ nécessite moins d'une seconde. De même, si l'algorithme en $O^*(1.5^n)$ nécessite plus de 6 mois pour une entrée de taille 92, un algorithme en $O^*(1.3^n)$ résout le problème en moins d'une minute. En plus, les techniques d'analyse des temps d'exécutions au pire des cas n'étant pas encore très matures, il se peut que la vraie complexité au pire des cas de l'algorithme analysé soit inférieure à la borne supérieure trouvée par les techniques d'analyse connues pour le moment.

Dans le passé, plusieurs méthodes ont été utilisées pour attaquer les problèmes NP-complets, mais ils ont tous leurs points faibles. Les algorithmes d'approximation, par exemple, utilisent un temps polynomial, mais trouvent seulement une solution appro-

chée de la solution optimale. En plus, certains problèmes sont difficiles à approximer ou trouvent des solutions loin de l'optimum. Si on restreint les entrées (à une classe de graphes, par exemple), la moindre anomalie de l'entrée peut rendre l'algorithme inutile. La méthode dont traite ce stage, a bien sûr aussi un désavantage ; c'est le temps d'exécution important pour les grandes entrées.

1.2 Historique

Les premiers algorithmes exponentiels datent des années 60 et 70. En 1962, Held et Karp [HK62] présentent un algorithme en $O(n^2 \cdot 2^n)$ pour le problème du voyageur de commerce, qui est toujours l'algorithme le plus rapide connu. L'algorithme de Lawler [Law76] pour le problème du coloriage d'un graphe de 1976 a été le plus rapide pendant longtemps avant d'avoir été amélioré par Eppstein [Epp03] en 2003 : avec une meilleure analyse, il améliore le temps de $O^*(2.4422^n)$ à $O^*(2.4150^n)$. Pour Ensemble Stable, Robson [Rob86] publie un algorithme en espace polynomial et en temps $O^*(1.2278^n)$ et une méthode utilisable pour d'autres algorithmes exponentiels qui lui permet de diminuer le temps d'exécution à $O^*(1.2108^n)$, mais en utilisant un espace exponentiel.

Au cours des dernières années, un intérêt croissant des algorithmes exponentiels a pu être constaté. Ainsi, trois groupes d'auteurs ont publié en 2004 indépendamment des algorithmes pour le problème NP-difficile Ensemble Dominant [FKW04, Gra04, RS04]. Les chercheurs de ce domaine s'intéressent aussi beaucoup au problème SAT et ses variantes (k-SAT, XSAT, etc.). Dans sa thèse de 2004 [Bys04], Byskov présente un algorithme pour XSAT en $O^*(1.1749^n)$. Et pour 3-SAT, le meilleur algorithme déterministe connu a une complexité de $O^*(1.4802^n)$ [DGH$^+$02].

1.3 Préliminaires techniques et notations

La notation O^*, qui apparaissait dès le début du rapport, est définie comme ceci.

Définition 1.1. *Soient f et g deux fonctions. On note $f(n) = O^*(g(n))$ si $f(n) = O(g(n) \cdot poly(n))$ où $poly(n)$ est une fonction polynomiale.*

On va utiliser quelques notations standards de graphe. Etant donné un graphe non-orienté $G = (V, E)$ à n sommets et m arêtes, le voisinage ouvert d'un sommet v est défini par $N(v) = \{u \in V : \{u, v\} \in E\}$ et son voisinage fermé par $N[v] = N(v) \cup \{v\}$. Etant donné un sous-ensemble de sommets $V' \subseteq V$, le graphe induit (ou sous-graphe) est défini par $G[V'] = (V', E')$ où $E' = E \cap V' \times V'$. Soit $v \in V$ un sommet, son degré $d(v)$ est le nombre de ses voisins : $d(v) = |N(v)|$. On note $\Delta(G)$ (resp. $\delta(G)$) le degré maximal (resp. minimal) de tous les sommets de G.

Un K_i est un graphe complet à i sommets, c'est-à-dire un graphe avec toutes les arêtes possibles ; un C_i est un cycle à i sommets et P_i un chemin à i sommets. Le complémentaire d'un graphe $G = (V, E)$ est noté $\overline{G}$ et défini par $\overline{G} = (V, V \times V - E)$.

Un ensemble de sommets $S \subseteq V$ est une clique si $G[S]$ est un graphe complet ; S est un ensemble stable s'il n'y a pas d'arêtes dans $G[S]$. Afin de simplifier la notation, on va écrire $G - v$ à la place de $G[V - \{v\}]$ et $G - S$ au lieu de $G[V - S]$ si v est un sommet de G et S un sous-ensemble de sommets de G. Les problèmes CLIQUE et ENSEMBLE STABLE consistent dans la recherche d'une clique, respectivement d'un ensemble stable, de cardinalité maximale.

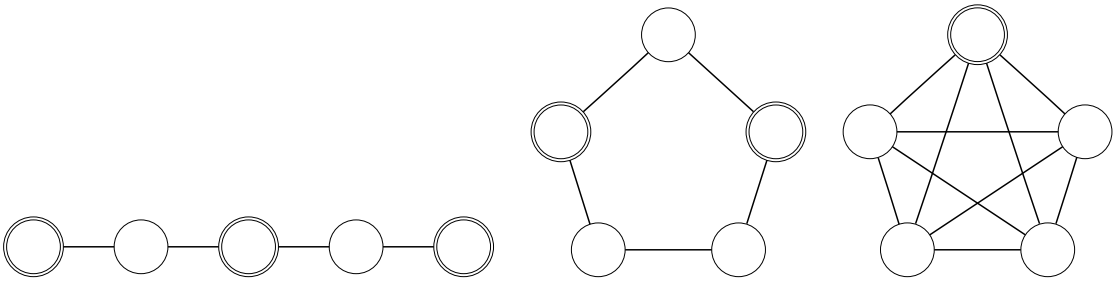


FIG. 1.1: P_5, C_5 et K_5 et un ensemble stable maximum pour ces graphes

1.4 Techniques standards pour la conception d'algorithmes exponentiels

Dans son papier [Woe01], Woeginger résume ce qui s'est passé dans le domaine des algorithmes exponentiels jusqu'en 2003 et présente quatre techniques standards pour concevoir des algorithmes exponentiels.

1.4.1 Prétraitement

L'idée est de faire un prétraitement qui permet de résoudre plus rapidement le problème, par exemple en triant (une partie de) l'entrée. Cette technique a été utilisé pour les problèmes SUBSET SUM et BINARY KNAPSACK, par exemple.

1.4.2 Recherche locale

Cette technique explore l'espace des solutions faisables en se déplaçant à chaque pas vers une solution "proche". Cette technique a été utilisée pour les problèmes SAT et k-SAT.

1.4.3 Programmation dynamique

La programmation dynamique est une méthode de résolution ascendante, qui détermine une solution optimale d'un problème à partir des solutions de ses sous-problèmes. Pour

le problème du voyageur de commerce, le meilleur algorithme [HK62] traite tous les sous-ensembles de villes dans l'ordre croissant de leur cardinalité, ce qui donne une complexité de $O^*(2^n)$ [3]. Pour le problème de la coloration d'un graphe, Lawler [Law76] a développé un algorithme en $O^*(2.4422^n)$ qui traite les sous-ensembles de sommets en ordre croissant de leur cardinalité. Dans [Epp03], Eppstein a amélioré cet algorithme en faisant une analyse plus fine de l'algorithme.

1.4.4 Arbre de recherche

Dans la technique par arbre de recherche (en anglais "search tree"), on se concentre sur une structure locale du problème, identifie les différents sous-cas et fait des branchements (le plus souvent récursivement) selon ces sous-cas. Ceci définit naturellement un arbre de recherche : un noeud de l'arbre correspond à un appel récursif de l'algorithme. Parfois, on peut argumenter que certains sous-cas ne peuvent pas donner des solutions faisables ou optimales ; on ne va donc pas traiter ces sous-cas, ce qui permet de gagner en termes de temps d'exécution.

1.5 Vue d'ensemble

La technique par arbre de recherche est sans doute celle qui est la plus utilisée. Je vais la décrire dans le chapitre suivant et l'appliquer pour concevoir deux algorithmes pour le problème Ensemble Stable sur des graphes de degré maximum 3. On verra tout de suite que le défi des algorithmes exponentiels repose plus dans l'évaluation du temps d'exécution (*analyse*) que dans la construction même de l'algorithme. Le fil conducteur de ce stage était de concevoir des algorithmes pour trouver des sous-graphes induits d'un graphe en entrée qui satisfont certaines conditions, comme l'appartenance à une classe de graphes, par exemple.

Le chapitre 2 a un rôle introductif et vise plus à présenter les techniques utilisées que la conception d'algorithmes compétitifs. Dans le chapitre 3, je vais présenter un algorithme plus compliqué en $O^*(1.6330^n)$ pour le problème Maximum Bipartite Subgraph qui est le meilleur connu pour le moment. Il utilise une façon naturelle de colorer et demi-colorer les sommets en deux couleurs et repose sur un lemme important pour limiter le nombre de sous-cas qu'on analyse. Le chapitre 4 va traiter des généralisations de Maximum Bipartite Subgraph et des problèmes connexes. J'y utiliserai les techniques "programmation dynamique" et "arbres de recherche".

[3] Remarquons qu'un algorithme trivial pour le problème du voyageur de commerce aurait une complexité de $O^*(n!)$ vu que c'est un problème de permutation.

2 Conception d'algorithmes exponentiels par la technique des arbres de recherche

Comme exemple introductif, considérons le problème ENSEMBLE STABLE sur des graphes de degré maximum trois. Les algorithmes de cette section vont êtres simples et non compétitifs pour bien pouvoir présenter les techniques utilisées. La première section va présenter un algorithme avec une analyse classique de son temps d'exécution, et la deuxième va présenter un algorithme légèrement modifié avec une analyse plus fine de la complexité de l'algorithme.

2.1 Algorithme avec une analyse standard

Remarquons d'abord qu'il existe un algorithme polynomial pour ENSEMBLE STABLE sur des graphes de degré maximal 2. Il est donc logique d'essayer de se débarrasser des sommets de degré 3 pour ensuite résoudre polynomialement le graphe restant.

Propriété 2.1. *Considérons un graphe $G = (V, E)$ et un sommet $v \in V$. Alors, chaque ensemble stable I respecte un des deux cas suivants :*

1. $v \notin I$
2. $v \in I$ *et* $I \cap N(v) = \varnothing$

Cette propriété implique directement l'algorithme 2.1.

Algorithme 2.1 Ensemble Stable pour des graphes de degré maximal 3

```
EnsStable3 ( G=(V,E) : Graphe, I : Ensemble ) : Entier
   Si Δ(G) ≤ 2 alors
      return EnsStable2(G, I)
   Sinon
      v ← un sommet de degré 3
      tailleES1 ← EnsStable3( G − v, I )
      tailleES2 ← 1 + EnsStable3( G − N[v], I ∪ {v})
      return max ( tailleES1, tailleES2 )
```

Si tous les sommets ont un degré inférieur ou égal à 2, la procédure EnsStable2 va résoudre le problème en temps polynomial. Si $\Delta(G) \leq 2$, alors le graphe est une union disjointe de cycles et de chemins et on peut choisir chaque deuxième sommet pour construire l'ensemble stable (voir figure 1.1 à la page 4).

Dans le cas contraire, il existe au moins un sommet v de degré 3. Conformément à la Propriété 2.1, deux cas sont possibles : soit on ajoute v à l'ensemble stable et on supprime v et ses voisins, soit on ne l'ajoute pas. A chaque fois que le branchement est appliqué, on diminue strictement le nombre de sommets de degré 3.

L'algorithme 2.1 calcule la taille d'un ensemble stable maximum, mais on peut facilement le modifier pour qu'il retourne un ensemble stable maximum.

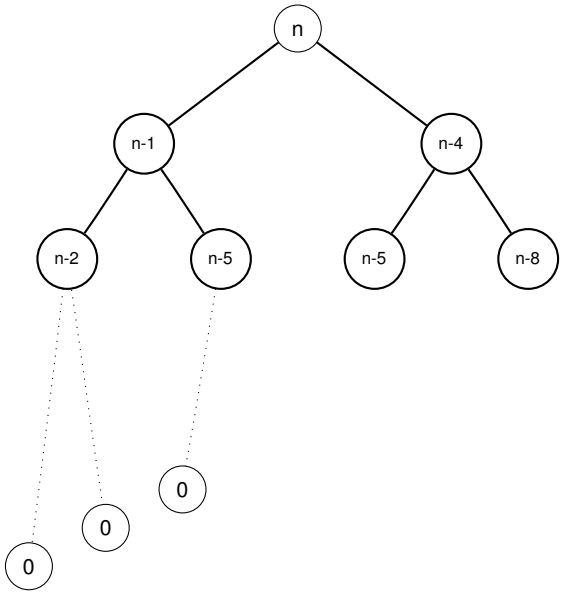


FIG. 2.1: Arbre représentant l'exécution d'un algorithme par arbre de recherche

Dans cet algorithme, on a une règle de réduction et une règle de branchement. On parle d'une *règle de réduction*, si on fait un seul appel récursif dans un sous-cas et d'une *règle de branchement* si on fait plusieurs appels récursifs. Ici, on dit qu'on fait un branchement selon $G - v$ et $G - N[v]$.

L'exécution de l'algorithme peut être représenté graphiquement par un arbre comme celui de la figure 2.1. Chaque noeud de l'arbre correspond à un appel récursif et ils sont étiquetés par la taille du graphe restant. La mesure du temps d'exécution de l'algorithme est fait classiquement en calculant une borne supérieure du nombre de noeuds (ou de feuilles) de cet arbre. On a la relation de récurrence suivante sur le temps d'exécution $T(n)$ de l'algorithme avec une entrée de taille n :

$$T(n) \leq T(n-1) + T(n-4)$$

Elle résulte du fait que les deux appels récursifs dans l'algorithme se font en supprimant 1, respectivement 4 sommets du graphe. Le vecteur de branchement $[1, 4]$ est une manière plus concise de représenter la relation de récurrence. Le polynôme caractéristique de cette relation de récurrence linéaire et homogène est $z^4 - z^3 - 1$; il a 4 racines dont une et une seule est réelle et positive. Cette racine réelle positive est la constante α dans la complexité de l'algorithme $T(n) = O^*(\alpha^n)$ [KL99, pp. 18-22]. Si on est confronté à plusieurs règles de branchement, et donc à plusieurs relations de récurrence, on peut prendre comme valeur de α la plus grande racine réelle positive des polynômes caractéristiques des relations de récurrence. Evidemment, ceci est très pessimiste, car on se place toujours dans le cas où on applique la règle de branchement avec le plus grand temps d'exécution, alors qu'il n'est probablement pas possible qu'on tombe toujours dans le plus mauvais cas.

En cherchant la racine réelle positive de $z^4 - z^3 - 1$ par la méthode de Newton, on trouve que l'algorithme s'exécute en $O^*(1.3803^n)$.

2.2 Algorithme avec une analyse améliorée

Afin d'améliorer l'analyse de l'algorithme, on veut pouvoir profiter des *effets de bord* des branchements. Il est profitable d'avoir beaucoup de sommets de degré inférieur à 3 dans le graphe, car on peut appliquer EnsStable2 plus tôt dans ce cas. Mais dans l'analyse de l'algorithme précédent, on ne tient pas compte du fait qu'on produit éventuellement des sommets de petit degré en supprimant des sommets.

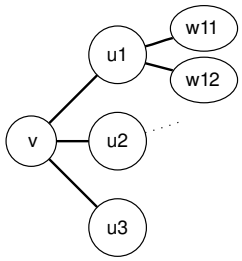


FIG. 2.2: si on supprime v, le degré de ses voisins diminue

Pour en tenir compte, il faut différencier les sommets de degré 3 des autres. Pour cela, on va introduire des poids :

- w_2 est le poids des sommets de degré inférieur ou égal à 2
- w_3 est le poids des sommets de degré 3

Notons $\mu(G)$ la fonction de mesure de la taille du problème. Si V est partitionné en $V_{\leq 2}$ et V_3, l'ensemble des sommets de degré inférieur ou égal à 2 et l'ensemble des sommets de degré 3, on a l'expression suivante pour $\mu(G)$:

$$\mu(G) = w_2 \cdot |V_{\leq 2}| + w_3 \cdot |V_3| \tag{2.1}$$

Il faut garantir que $\mu(G) \leq n$ à tout moment. Pour cela, on impose les conditions suivantes sur w_2 et w_3 :

$$0 < w_2, w_3 \leq 1 \tag{2.2}$$

On va supposer qu'il est toujours préférable de supprimer un sommet de V_3 qu'un sommet de $V_{\leq 2}$ et qu'on préfère supprimer un sommet de $V_{\leq 2}$ à passer un sommet de V_3 dans $V_{\leq 2}$. Ceci donne les conditions suivantes :

$$w_2 \leq w_3 \leq 2 \cdot w_2 \tag{2.3}$$

L'algorithme 2.2 est le même que l'algorithme 2.1, sauf qu'on a rajouté trois règles de réduction qui vont assurer que lors du branchement, les voisins de $N[u]$ seront assez nombreux et qu'il y aura donc assez de sommets qui vont diminuer de degré si $v \in I$. Avec une analyse comme celle de la section précédente, on trouverait le même temps d'exécution pour cet algorithme que pour le précédent.

Algorithme 2.2 Ensemble Stable pour des graphes de degré maximal 3 pour une analyse pondérée

```
EnsStable3p ( G=(V,E) : Graphe, I : Ensemble ) : Entier
   Si Δ(G) ≤ 2 alors
(1)   return EnsStable2(G, I)
   Sinon Si ∃v ∈ V : d(v) ≤ 1 alors
(2)   return EnsStable3p( G − N[v], I ∪ {v}) + l(v)
   Sinon Si ∃u, v ∈ V : N(u) = N(v) alors
(3)   return EnsStable3p( Fusion(G, u, v ), I)
   Sinon Si ∃u, v, w : uvw forme un C3 et ∃x ∈ {u, v, w} : d(x) = 2 alors
(4)   return EnsStable3p( G − N[x], I ∪ {x} ) + l(x)
   Sinon
      Si ∃u ∈ V : d(u) = 3 et ∃w ∈ N(u) : d(w) = 2 alors
         v ← u
      Sinon
         v ← un sommet de degré 3
      tailleES1 ← EnsStable3p( G − v, I )
      tailleES2 ← EnsStable3p( G − N[v], I ∪ {v}) + l(v)
      return min ( tailleES1, tailleES2 )
```

On va dire que l'ajout d'un sommet v à I est sûr s'il existe un ensemble stable de taille maximale contenant v.

- **Réduction 2.** S'il existe un sommet v de degré 0 ou 1, on peut l'ajouter à l'ensemble stable I immédiatement et éventuellement supprimer son voisin. De toute façon, on ne peut pas ajouter v et son voisin à l'ensemble stable. L'ajout de v à I est donc sûr.

- **Réduction 3.** Si deux sommets u, v ont le même voisinage, on peut les traiter de la même manière. Ils ne peuvent pas être adjacents. Si on ajoute v à I, on ajoute aussi u et inversement, et si on doit supprimer u, on doit aussi supprimer v. La fonction FUSION va fusionner les deux sommets en un seul uv et retourner le graphe résultant. Le sommet qui résulte de cette fusion devra compter "double" si on l'ajoute à I. C'est pour ça qu'on va étiqueter chaque sommet s par une fonction $l(s)$ qui vaut initialement 1 pour chaque sommet. Donc, $l(uv) = l(u) + l(v)$.
- **Réduction 4.** S'il y a un triangle dans G avec au moins un sommet x de degré 2, on peut ajouter x à I et supprimer le triangle. On peut seulement ajouter un seul sommet du triangle à I et en en choisissant un de degré 2, il n'y aura pas d'effets sur des sommets extérieurs au triangle. L'ajout de x à I est donc sûr.

On peut supposer que le graphe en entrée est connexe ; s'il ne l'est pas, on exécute l'algorithme sur chacune de ses composantes connexes. Soient $C_1, C_2, ..., C_k$ les composantes connexes de G, un ensemble stable de taille maximale de G est l'union des ensembles stables de taille maximale de chacune des composantes connexes : $I(G) = \bigcup_{i=1}^{k} I(C_i)$.

Analysons la règle de branchement :

- **(Cas 1 : 3 voisins de v ont degré 2)** Dans ce cas, les voisins u_1, u_2 et u_3 de v ont 3 autres voisins x_1, x_2 et x_3 : à cause de la réduction 4, les $u_i, i = 1..3$ ne peuvent pas être adjacents entre eux.
  Si $v \in I$, on va supprimer $N[v] = \{v, u_1, u_2, u_3\}$ $(-3w_2 - w_3)$. En plus, les x_i perdent un degré $(+3w_2 - 3w_3)$.
  Si $v \notin I$, on va supprimer v $(-w_3)$. Ensuite, u_1, u_2et u_3 auront degré 1 et la règle de réduction 2 va être appliquée 3 fois dans les appels récursifs suivants : $u_1, u_2, u_3 \in I$ et $N[u_1], N[u_2]$ et $N[u_3]$ vont être supprimées $(-3w_2 - 3w_3)$. Au pire des cas, tous les x_i ont degré 3, et il y a deux arêtes entre les x_i . On peut supposer que les x_i ne forment pas un triangle ; sinon on serait confronté à une composante connexe de taille constante pour laquelle on trouve un ensemble stable en temps constant. Il y aura donc en plus 2 sommets qui passent de degré 3 à degré 2 $(+2w_2 - 2w_3)$.
  En tout, on a donc $T(n) \leq T(n - 4w_3) + T(n - w_2 - 6w_3)$.

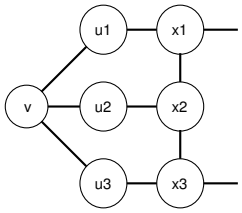


FIG. 2.3: Cas1

– **(Cas 2 : 2 voisins de v ont degré 2 et 1 voisin de v a degré 3)** Dans ce cas, u_1, u_2 et u_3 ont 4 autres voisins (x_1, x_2, x_3, x_4) au pire des cas. Soit u_1 le voisin de degré 3.
Si $v \in I$, on va supprimer $N[v] = \{v, u_1, u_2, u_3\}$ $(-2w_2 - 2w_3)$. En plus, les 4 autres voisins perdent un degré $(+4w_2 - 4w_3)$.
Si $v \notin I$, on va supprimer v $(-w_3)$. Ensuite, $u_2$et u_3 auront degré 1 et comme dans le cas 1, la règle de réduction 2 va être appliquée 2 fois. u_2, u_3 et deux x_i sont supprimés $(-2w_2 - 2w_3)$ et deux sommet passent de degré 2 à degré 3 $(+2w_2 - 2w_3)$ au pire des cas. Le sommet u_1 va juste passer de degré 3 à degré 2 $(+w_2 - w_3)$.
En tout, $T(n) \leq T(n + 2w_2 - 6w_3) + T(n + w_2 - 6w_3)$.

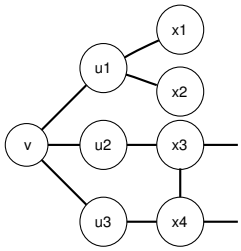


FIG. 2.4: Cas2

– **(Cas 3 : 1 voisin de v a degré 2 et 2 voisins de v ont degré 3)** Soit u_3 le voisin de degré 2. Dans ce cas, u_1, u_2 et u_3 ont 3 autres voisins et u_1 et u_2 sont adjacents au pire des cas.
Si $v \in I$, on va supprimer $N[v] = \{v, u_1, u_2, u_3\}$ $(-w_2 - 3w_3)$. En plus, les 3 autres voisins perdent un degré $(+3w_2 - 3w_3)$.
Si $v \notin I$, on va supprimer v $(-w_3)$. Ensuite, u_3 aura degré 1 et la règle de réduction 2 va être appliquée une fois $(+w_2 - 3w_3)$. Les sommets $u_1$et u_2 vont passer de degré 3 à degré 2 $(+2w_2 - 2w_3)$.
En tout, on a donc $T(n) \leq T(n + 2w_2 - 6w_3) + T(n + 3w_2 - 6w_3)$. Remarquons que cette relation de récurrence donne un temps pire que celle du Cas 2.

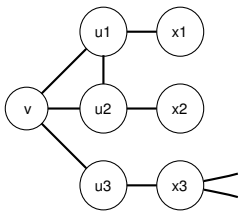


FIG. 2.5: Cas3

– **(Cas 4 : 3 voisins de v ont degré 3)** Ce cas peut seulement arriver une seule fois au cours du déroulement de l'algorithme, à savoir si le graphe d'entrée est cubique,

c'est-à-dire si $\Delta(G) = \delta(G) = 3$, car aucune règle de réduction et de branchement ne peut produire un graphe cubique. Le cas 4 peut donc seulement agrandir l'arbre de recherche par un facteur constant et on pourra négliger ce cas.

Finalement, on a donc à résoudre le système de relations de récurrences

$$T(n) \leq max \begin{cases} T(n-4w_3) + T(n-w_2-6w_3) \\ T(n+2w_2-6w_3) + T(n+w_2-6w_3) \\ T(n+2w_2-6w_3) + T(n+3w_2-6w_3) \end{cases}$$

sous les conditions

$$\begin{cases} 0 < w_2, w_3 \leq 1 \\ w_2 \leq w_3 \leq 2 \cdot w_2 \end{cases}$$

Pour trouver les valeurs optimales pour les poids, il est indispensable de faire recours à des programmes spécialisés pour le faire, même lorsque le nombre de récurrences et de poids est faible. Dans [Epp04], Eppstein présente une méthode par descente de gradients multiples pour trouver les valeurs optimales pour les poids w_2 et w_3. Dans le cadre de ce stage, un programme a été développé qui utilise une recherche locale randomisée pour trouver les valeurs optimales pour les poids.

Pour trouver la constante α dans le temps d'exécution de l'algorithme $T(n) = O^*(\alpha^n)$, on peut choisir des valeurs pour les poids, résoudre chaque relation de récurrence séparément (comme pour l'analyse standard de la section précédente) et prendre le maximum de ces résultats. Pour cet algorithme, les valeurs optimales pour les poids sont $w_2 = 0,5$ et $w_3 = 1$, ce qui donne un temps d'exécution de $O^*(1.1573^n)$. On dit que la relation de récurrence $T(n) \leq T(n+2w_2-6w_3) + T(n+3w_2-6w_3)$ est *juste*, car son résultat est maximal (et vaut 1.1573). L'analyse présentée ici donne une borne supérieur du temps d'exécution au pire des cas et considère qu'on se trouve toujours dans le Cas 3 lorsqu'on exécute l'algorithme. Mais il est possible qu'il n'existe pas d'entrée pour laquelle on se trouve toujours dans ce cas et que la vraie complexité au pire des cas est inférieure à celle déterminée ici.

Les meilleurs algorithmes connus pour ce problème sont un algorithme de Beigel [Bei99] en $O^*(1.1259^n)$ et un algorithme de Chen et al. [CKX03] en $O^*(1.1255^n)$.

3 Maximum Bipartite Subgraph

Dans ce chapitre, je vais présenter un algorithme pour MAXIMUM BIPARTITE SUBGRAPH. Ce problème est une suite logique du problème ENSEMBLE STABLE, car il consiste à trouver deux ensembles stables disjoints, tel que leur union soit de taille maximale. Le meilleur algorithme connu pour ce problème est un algorithme de Byskov [Bys04] qui liste tous les sous-graphes bipartis maximaux [1] en temps $O^*(1.7724^n)$. L'algorithme présenté ici va utiliser des techniques similaires et aura un temps d'exécution de $O^*(1.6330^n)$.

Définition 3.1. *Un graphe non-orienté $G = (V, E)$ est biparti si on peut partitionner V en deux sous-ensembles W et B tel que toutes les arêtes du graphe ont une extrémité dans W et l'autre dans B.*

Etant donné un graphe non-orienté $G = (V, E)$, le problème de *sous-graphe biparti de taille maximale* (MAXIMUM BIPARTITE SUBGRAPH) consiste à trouver le plus grand sous-ensemble V' de sommets tel que $G[V']$ soit biparti. Ce sous-ensemble V' de sommets est partitionné en l'ensemble des sommets noirs et l'ensemble des sommets blancs.

Etant une instance du problème plus général Maximum Induced Subgraph With Property II, ce problème est NP-complet [Yan78].

Définition 3.2. *Un graphe demi-coloré G est un quintuplet (V, F, B, W, E) où (V, E) est un graphe non-orienté et F, B et W forment une partition de V. On appelle les sommets dans F les sommets incolores (full vertices), les sommets dans B sont colorés demi-noirs et les sommets dans W sont colorés demi-blancs.*

La règle pour le coloriage des sommets est que les sommets dans F peuvent être colorés noirs ou blancs, les sommets dans B peuvent seulement être colorés noirs et les sommets dans W peuvent seulement être colorés blancs.

Au début de l'algorithme, tous les sommets sont incolores. En colorant un sommet en noir, on colore ses voisins en demi-blanc : ils ne pourront plus être colorés en noir, car l'ensemble des sommets noirs doit former un ensemble stable.

Définition 3.3. *Soit $S \subseteq V$ un sous-ensemble de sommets et $u \in V$ un sommet d'un graphe (éventuellement demi-coloré) G. On note $d_S(u)$ le nombre de voisins de u parmi S.*

$$d_S(u) = |N(u) \cap S|$$

[1] un sous-graphe biparti maximal (par induction) est un sous-graphe biparti tel qu'on ne peut pas obtenir un sous-graphe biparti en lui ajoutant un sommet

Le lemme suivant va être important pour l'algorithme de ce chapitre, car il va permettre de descendre le temps d'exécution à $O^*(1.6330^n)$.

Lemme 3.1. *Soit $G = (V, F, B, W, E)$ un graphe demi-coloré. Soit $u \in B$ un sommet demi-noir du graphe ayant au moins deux voisins dans $F \cup B$. Alors, il existe un sous-graphe biparti de taille maximale vérifiant une et une seule des deux conditions suivantes :*

1. *u est coloré en noir*
2. *au moins deux voisins de u sont colorés en noir*

Preuve. Il suffit de démontrer que si u n'est pas noir, alors au moins deux de ses voisins peuvent être colorés en noir.

Si, dans la coloration finale du graphe, aucun voisin de u n'est noir, alors u peut être coloré en noir, ce qui correspond à la première condition du lemme. Si un seul des voisins de u est noir, u ne peut pas appartenir à l'ensemble des sommets du sous-graphe biparti. Dans ce cas, on peut en supprimer le voisin noir et colorer u en noir à sa place.

□

Considérons un sommet demi-noir u de degré 3 : $N(u) = \{v_1, v_2, v_3\}$. On fera donc trois branchements : soit u est coloré en noir, soit on supprime u et on colore v_1 en noir. Et à cause du lemme, il restera une seule possibilité : si u et v_1 ne sont tous les deux pas colorés en noir, on peut colorer v_2 et v_3 en noir. Le sommet u est supprimé dans ce cas et v_1 est soit supprimé, soit coloré en demi-blanc, selon qu'il était demi-noir ou incolore.

L'algorithme de cette section va être présenté par une suite de règles de réductions et de branchements. On applique toujours la première règle applicable.

Le graphe d'entrée est supposé connexe. Si ce n'est pas le cas, l'algorithme est exécuté sur chaque composante connexe du graphe G et le sous-graphe biparti de taille maximale est l'union des sous-graphes bipartis de taille maximale des composantes connexes de G. En plus, chaque fois qu'un graphe intermédiaire devient non-connexe, l'algorithme va traiter les parties connexes séparément.

Pour l'analyse de l'algorithme, introduisons des poids pour les sommets :

1. h est le poids pour les sommets demi-colorés
2. f est le poids pour les sommets incolores

Avec ces poids, nous obtenons la mesure de complexité d'un graphe demi-coloré $G = (V, F, B, W, E)$:

$$\mu(G) = f \cdot |F| + h \cdot |B \cup W|$$

et avec les conditions

$$\begin{cases} 0 < h, f \leq 1 \\ h \leq f \leq 2 \cdot h \end{cases} \tag{3.1}$$

on assure que $\mu(G) \leq n$ et on suppose qu'il est plus avantageux de supprimer un sommet incolore qu'un sommet demi-coloré et qu'il vaut mieux supprimer un sommet dans $B \cup W$ que de demi-colorer un sommet de F.

3.1 Règles de réduction

Les règles de réduction suivantes vont assurer que, lors des branchements, tous les sommets ont degré au moins 2, que les sommets demi-colorés de degré 2 ne sont pas contenu dans un triangle, qu'un sommet demi-noir n'est pas adjacent à un sommet demi-blanc, qu'un sommet incolore n'a pas seulement des voisins demi-colorés de la même couleur et qu'il existe au moins un sommet demi-coloré dans le graphe.

1. Sommet isolé
   S'il y a un sommet isolé dans le graphe, on peut le colorer en noir et l'enlever du graphe.

2. Sommet de degré 1
   S'il y a un sommet u de degré 1 dans le graphe, appelons v son voisin unique.
   Si $u \notin F$, alors on colore u en blanc ou en noir selon si u est demi-blanc ou demi-noir. Il faudra ensuite éventuellement supprimer ou demi-colorer son voisin v.
   Si $u \in F$ et $v \in B$, alors on colore u en blanc et on l'enlève du graphe. Vu que le cas $u \in F$ et $v \in W$ est symétrique, il reste juste le cas $u \in F$ et $v \in F$: on enlève u du graphe et à la fin de l'algorithme, on colore u en noir ou en blanc, selon la couleur de v.

3. Sommet demi-coloré de degré 2 contenu dans un triangle
   Si un sommet $u \in B$ (resp. $u \in W$) a deux voisins v_1 et v_2 adjacents entre eux, on peut colorer u en noir (resp. blanc) et l'enlever du graphe. Selon le lemme 3.1, on peut colorer en noir soit u, soit v_1 et v_2. Vu que v_1 et v_2 ne peuvent pas avoir la même couleur parce qu'ils sont adjacents, il existe un sous-graphe biparti de taille maximale contenant u comme sommet noir.

4. Une arête entre un sommet demi-noir et un sommet demi-blanc
   On peut toujours supprimer une telle arête, car elle n'influence pas la coloration du reste du graphe.

5. Un sommet $u \in F$ sans voisins dans $F \cup B$ (resp. $F \cup W$)
   On colore u en noir (resp. blanc) et on l'enlève du graphe.

6. Il n'existe pas de sommet demi-coloré
   Dans ce cas, on colore en demi-noir un sommet du graphe.

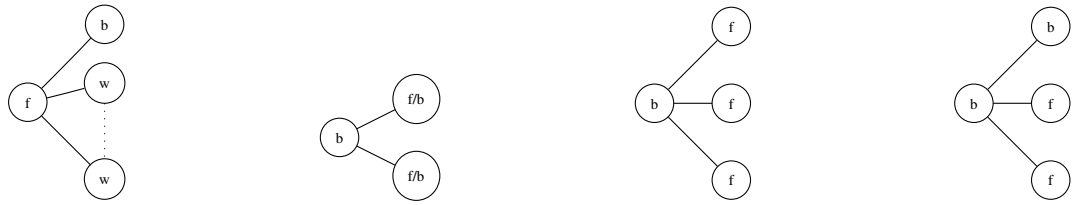
3.2 Règles de branchement

Pour les branchements, on choisit un sommet u demi-coloré de degré minimal (sauf pour le premier branchement). A cause de la réduction 6, un tel sommet existe forcément. Ici, on présente juste les cas où u est demi-noir, mais les cas où u est demi-blanc sont symétriques.

Le sommet demi-noir traité est u et ses voisins sont $v_1, v_2, ..., v_{d(u)}$. u ne peut pas avoir de voisins demi-blancs à cause de la réduction 4 et donc $d_{F\cup B}(u) = d(u)$. On va dire dans la suite qu'on colore un sommet en noir si on l'ajoute aux sommets noirs du sous-graphe biparti et qu'on l'enlève du graphe et on dit qu'on supprime un sommet si on l'enlève du graphe et qu'il ne fera pas parti du sous-graphe biparti.

1. $\exists v \in F$ avec $d_B(v) = 1$ et $d_F(v) = 0$
   v ou son voisin demi-noir est coloré en noir. v a seulement des voisins demi-colorés dont un seul, appelons le x, est demi-noir. A cause de la réduction 2, x a un autre voisin dans $F \cup B$, qui est dans F au pire des cas. Si v est coloré en noir $(-f)$, on supprime x $(-h)$ et si x est coloré en noir $(-h)$, on colore ses deux voisins incolores en demi-blanc au pire des cas $(+2h - 2f)$. On suppose que l'autre voisin de x est incolore, car c'est le pire des cas sous les conditions 3.1.

$$T(n) \leq T(n - h - f) + T(n + h - 2f)$$



(a) Cas 1 (b) Cas 2 (c) Cas 3a (d) Cas 3b

FIG. 3.1: Schémas pour les premiers cas. *Les sommets f sont incolores, les sommets b demi-noirs et les sommets w demi-blancs*

2. $d(u) = 2$
   u est coloré en noir ou supprimé (sans le colorer en noir ou blanc). Si u est coloré en noir $(-h)$, on supprime ses voisins demi-noirs $(-d_B(u)h)$ et on colore en demi-blanc ses voisins incolores $(+d_F(u)h - d_F(u)f)$. Si u est supprimé $(-h)$, ses voisins sont colorés en noir $(-d_B(u)h - d_F(u)f)$. On sait que les voisins de u ont au moins un autre voisin dans $F \cup B$ à cause de la réduction 2 et du branchement 1, qui va

passer de F dans W au pire des cas $(+h-f)$.

$$\begin{aligned} T(n) &\leq T(n+h-2f)+T(n-3f) \\ T(n) &\leq T(n-h-f)+T(n-h-2f) \\ T(n) &\leq T(n-3h)+T(n-2h-f) \end{aligned}$$

3. $d(u)=3$

   a) $d_F(u)=3$
   Colorer u en noir, ou supprimer u et colorer v_1 en noir ou supprimer u et colorer v_1 en demi-blanc. Si u est coloré en noir $(-h)$, ses 3 voisins passent de F dans W $(+3h-3f)$. On va supposer que les voisins de u ont au moins deux autres voisins ; sinon, des sommets de degré 1 ou 0 seraient produits en supprimant u et la règle de réduction 2 peut être appliquée, ce qui améliore considérablement les relations de récurrence. Si u est supprimé $(-h)$ et v_1 est coloré en noir $(-f)$, au moins deux autres voisins de v_1 deviennent demi-blancs au pire des cas $(+2h-2f)$. Si u est supprimé $(-h)$ et v_1 est coloré en demi-blanc $(+h-f)$, v_2 et v_3 deviennent noirs à cause du lemme 3.1 $(-2f)$ et ils ont au moins deux autres voisins qui passent de F dans W $(+2h-2f)$.

$$T(n) \leq T(n+2h-3f)+T(n+h-3f)+T(n+2h-5f)$$

   Remarque : Supposons qu'il existe une arête entre v_1 et v_2. On sait alors que v_3 devient noir si on supprime u et on obtient des récurrences meilleures. Ceci est vrai en général : on obtient de meilleures récurrences s'il y a des arêtes entre les voisins de u et c'est pour ça qu'on va supposer dans la suite qu'il n'y a pas d'arêtes entre les voisins de u pour l'analyse au pire des cas.

   b) $d_F(u)=2$
   Soit v_1 le voisin demi-noir de u. Colorer u en noir, ou supprimer u et colorer v_1 en noir ou supprimer u et v_1. La coloration de u en noir $(-h)$ implique que v_2 et v_3 sont colorés demi-blancs $(+2h-2f)$ et que v_1 est supprimé $(-h)$. Si on colore v_1 en noir $(-h)$, on supprime u $(-h)$ et au moins deux autres voisins de v_1 passent au demi-blanc $(+2h-2f)$. v_1 a au moins 3 voisins, sinon, il aurait été traité dans le branchement 2. Si u et v_1 sont supprimés $(-2h)$, alors v_2 et v_3 sont colorés en noir $(-2f)$ et au moins deux voisins de v_2 et v_3 sont colorés demi-blancs $(+2h-2f)$.

$$T(n) \leq 2T(n-2f)+T(n-4f)$$

   c) $d_F(u)=1$
   On applique les mêmes règles que dans le cas précédent. La coloration de u en noir va cette fois supprimer le sommet demi-noir v_2 au lieu de le demi-colorer $(-2h+f$ par rapport au cas précédent). La suppression de u et de v_1 implique cette fois la coloration d'un sommet demi-coloré v_2 en noir $(-h+f)$.

$$T(n) \leq T(n-2h+f)+T(n-2f)+T(n-h-3f)$$

d) $d_F(u) = 0$
Mêmes règles. Si u devient noir, son troisième voisin est aussi supprimé au lieu d'être demi-coloré $(+2h - f)$. Si u et v_1 sont supprimés, v_3 était dans B (au lieu de F) avant de devenir noir $(-h + f)$.

$$T(n) \leq T(n - 4h) + T(n - 2f) + T(n - 2h - 2f)$$

4. $d(u) = 4$

a) $d_F(u) = 4$ et plus d'un voisin de u a degré au moins 3
Soient v_1 et v_2 des sommets de degré ≥ 3. L'algorithme va analyser 4 sous-cas :

i. Colorer u en noir $(-h)$. Ses voisins sont colorés demi-blancs $(+4h - 4f)$.

ii. Supprimer u $(-h)$ et colorer v_1 en noir $(-f)$. Deux voisins de v_1 sont colorés demi-blancs au pire des cas $(+2h - 2f)$.

iii. Supprimer u $(-h)$, colorer v_1 en demi-blanc $(+h - f)$ et colorer v_2 en noir $(-f)$. Deux voisins de v_2 sont colorés demi-blancs au pire des cas $(+2h - 2f)$.

iv. Supprimer u $(-h)$, colorer v_1 et v_2 en demi-blanc $(+2h - 2f)$ et colorer v_3 et v_4 en noir $(-2f)$. Deux voisins de v_3 et v_4 sont colorés demi-blancs au pire des cas $(+2h - 2f)$.

$$T(n) \leq T(n+3h-4f)+T(n+h-3f)+T(n+2h-4f)+T(n+3h-6f)$$

b) $d_F(u) = 4$ et au moins 3 voisins de u ont degré 2

i. Colorer u en noir $(-h)$. Ses voisins deviennent demi-blancs $(+4h - 4f)$ et au moins 3 d'eux deviendront des sommets demi-blancs de degré 1, qu'on peut colorer définitivement en blanc $(-3h)$.

ii. Supprimer u $(-h)$

$$T(n) \leq T(n - 4f) + T(n - h)$$

c) $d_F(u) = 3$
Soit v_1 le voisin demi-noir de u. On utilise les mêmes règles de branchement que dans le cas 4a.

i. Colorer u en noir $(-h)$. Trois voisins sont colorés demi-blancs $(+3h - 3f)$ et v_1 est supprimé $(-h)$.

ii. Supprimer u $(-h)$ et colorer v_1 en noir $(-h)$. Trois voisins de v_1 sont colorés demi-blancs au pire des cas $(+3h - 3f)$.

iii. Supprimer u et v_1 $(-2h)$ et colorer v_2 en noir $(-f)$. Un voisin de v_2 est coloré demi-blanc au pire des cas $(+h - f)$.

iv. Supprimer u et v_1 $(-2h)$, colorer v_2 en demi-blanc $(+h-f)$ et colorer v_3 et v_4 en noir $(-2f)$. Deux voisins de v_3 et v_4 sont colorés demi-blancs au pire des cas $(+2h-2f)$.

$$T(n) \leq 2T(n+h-3f) + T(n-h-2f) + T(n+h-5f)$$

d) - f) Pour $d_F(u) = 2$, $d_F(u) = 1$ et $d_F(u) = 0$, on utilise les mêmes règles de branchement que dans les cas 4a et 4c et on obtient les relations de récurrences suivantes :

$$\begin{aligned} T(n) &\leq T(n-h-2f) + T(n+h-3f) + T(n-3f) + T(n-h-4f) \\ T(n) &\leq T(n-3h-f) + T(n+h-3f) + T(n-3f) + T(n-h-4f) \\ T(n) &\leq T(n-5h) + T(n+h-3f) + T(n-3f) + T(n-h-4f) \end{aligned}$$

5. $d(u) = 5$

a) $d_F(u) = 5$ et au moins un voisin de u a degré 2.
Colorer u en noir $(-h)$ ou le supprimer $(-h)$. Si on le colore en noir, ses 5 voisins deviennent demi-blancs au pire des cas $(+5h-5f)$. Au moins un sommet demi-blanc de degré 1 sera produit et on pourra le colorer en blanc $(-h)$.

$$T(n) \leq T(n+3h-5f) + T(n-h)$$

b) $d_F(u) = 5$ et tous les voisins de u ont degré au moins 3.
L'algorithme va analyser les sous-cas suivants :

i. Colorer u en noir $(-h)$. Cinq voisins sont colorés demi-blancs $(+5h-5f)$.

ii. Supprimer u $(-h)$ et colorer v_1 en noir $(-f)$. Deux voisins de v_1 sont colorés demi-blancs au pire des cas $(+2h-2f)$.

iii. Supprimer u $(-h)$, colorer v_1 en demi-blanc $(+h-f)$ et colorer v_2 en noir $(-f)$. Deux voisins de v_2 sont colorés demi-blanc au pire des cas $(+2h-2f)$.

iv. Supprimer u $(-h)$, colorer v_1 et v_2 en demi-blanc $(+2h-2f)$ et colorer v_3 en noir $(-f)$. Deux voisins de v_3 sont colorés demi-blanc au pire des cas $(+2h-2f)$.

v. Supprimer u $(-h)$, colorer v_1, v_2 et v_3 en demi-blanc $(+3h-3f)$ et colorer v_4 et v_5 en noir $(-2f)$. Trois voisins de v_4 et v_5 sont colorés demi-blancs au pire des cas $(+3h-3f)$.

$$\begin{aligned} T(n) \leq\ & T(n+4h-5f) + T(n+h-3f) + T(n+2h-f) + \\ & T(n+3h-5f) + T(n+5h-8f) \end{aligned}$$

c) $d_F(u) = 4$ et au moins un voisin de u a degré 2.
Colorer u en noir $(-h)$ ou le supprimer $(-h)$. Si on le colore en noir, un voisin

demi-noir sera supprimé $(-h)$ et ses autres 4 voisins deviennent demi-blancs au pire des cas $(+4h-4f)$. Au moins un sommet demi-blanc de degré 1 sera produit et on pourra le colorer en blanc $(-h)$.

$$T(n) \leq T(n+h-4f) + T(n-h)$$

d) $d_F(u) = 4$ et tous les voisins de u ont degré au moins 3.
Soit v_1 le voisin demi-noir. L'algorithme va analyser les sous-cas suivants :

i. Colorer u en noir $(-h)$. v_1 est supprimé $(-h)$ et les 4 autres voisins sont colorés demi-blancs $(+4h-4f)$.

ii. Supprimer u $(-h)$ et colorer v_1 en noir $(-h)$. Quatre voisins de v_1 sont colorés demi-blancs au pire des cas $(+4h-4f)$.

iii. Supprimer u et v_1 $(-2h)$ et colorer v_2 en noir $(-f)$. Deux voisins de v_2 sont colorés demi-blancs au pire des cas $(+2h-2f)$.

iv. Supprimer u et v_1 $(-2h)$, colorer v_2 en demi-blanc $(+h-f)$ et colorer v_3 en noir $(-f)$. Deux voisins de v_3 sont colorés demi-blanc au pire des cas $(+2h-2f)$.

v. Supprimer u et v_1 $(-2h)$, colorer v_2 et v_3 en demi-blanc $(+2h-2f)$ et colorer v_4 et v_5 en noir $(-2f)$. Trois voisins de v_4 et v_5 sont colorés demi-blancs au pire des cas $(+3h-3f)$.

$$T(n) \leq 2T(n+2h-4f) + T(n-3f) + T(n+h-4f) + T(n+3h-7f)$$

e) $d_F(u) = 3$
Soient v_1 et v_2 les voisins demi-noirs. L'algorithme va analyser les sous-cas suivants :

i. Colorer u en noir $(-h)$. v_1 et v_2 sont supprimés $(-2h)$ et les 3 autres voisins sont colorés demi-blancs $(+3h-3f)$.

ii. Supprimer u $(-h)$ et colorer v_1 en noir $(-h)$. Quatre voisins de v_1 sont colorés demi-blancs au pire des cas $(+4h-4f)$.

iii. Supprimer u et v_1 $(-2h)$ et colorer v_2 en noir $(-h)$. Quatre voisins de v_2 sont colorés demi-blancs au pire des cas $(+4h-4f)$.

iv. Supprimer u, v_1 et v_2 $(-3h)$ et colorer v_3 en noir $(-f)$. Un voisin de v_3 est coloré demi-blanc au pire des cas $(+h-f)$.

v. Supprimer u, v_1 et v_2 $(-3h)$, colorer v_3 en demi-blanc $(+h-f)$ et colorer v_4 et v_5 en noir $(-2f)$. Trois voisins de v_4 et v_5 sont colorés demi-blancs au pire des cas $(+3h-3f)$.

$$\begin{aligned} T(n) \quad \leq \quad & T(n-3f) + T(n+2h-4f) + T(n+h-4f) + \\ & T(n-2h-2f) + T(n+h-5f) \end{aligned}$$

f) - h) Pour $d_F(u) = 2$, $d_F(u) = 1$ et $d_F(u) = 0$, on utilise les mêmes règles de branchement que dans le cas précédent et on obtient les relations de récurrences suivantes :

$$\begin{aligned} T(n) &\leq T(n-2h-2f) + T(n+2h-4f) + T(n+h-4f) + \\ &\quad T(n-4f) + T(n-2h-4f) \\ T(n) &\leq T(n-4h-f) + T(n+2h-4f) + T(n+h-4f) + \\ &\quad T(n-4f) + T(n-h-5f) \\ T(n) &\leq T(n-6h) + T(n+2h-4f) + T(n+h-4f) + \\ &\quad T(n-4f) + T(n-h-5f) \end{aligned}$$

6. $d(u) \geq 6$
   Colorer u en noir $(-h)$ ou le supprimer $(-h)$. Si on le colore en noir, 6 de ses voisins deviennent demi-blancs au pire des cas $(+6h - 6f)$.

$$T(n) \leq T(n+5h-6f) + T(n-h)$$

Le système de relations de récurrences comporte ici 23 relations de récurrence. En le résolvant par la technique décrite dans le chapitre 2, on trouve que les poids optimaux sont $h = 0.5$ et $f = 1$ et que le temps d'exécution est $O^*(1.6330^n)$. Il y a une seule récurrence juste : celle du cas 5b. Il parait difficile de l'améliorer sans modifier complètement l'algorithme ou l'analyse.

4 Généralisations et problèmes connexes

Ce chapitre va présenter d'autres résultats obtenus lors du stage sans rentrer trop dans les détails.

4.1 Maximum k, l-Subgraph

Les problèmes Maximum Bipartite Subgraph et Maximum Split Subgraph consistent à chercher respectivement 2 ensembles stables disjoints et un ensemble stable et une clique disjoints. On peut généraliser ces deux problèmes en cherchant k ensembles stables et l cliques :

Problème : Maximum k, l-Subgraph
Entrée : Un graphe non-orienté $G = (V, E)$
Sortie : k ensembles stables $I_1, I_2, ..., I_k$ et l cliques $C_1, C_2, ..., C_l$, tous disjoints, tel que la cardinalité de leur union $I_1 \cup I_2 \cup ... \cup I_k \cup C_1 \cup C_2 \cup ... \cup C_l$ est la plus grande possible.

Dans le chapitre précédent, on a étudié le cas où $k = 2$ et $l = 0$. De part la nature complémentaire des deux problèmes Clique et Ensemble Stable, on peut exécuter l'algorithme précédent sur le graphe complémentaire pour résoudre le cas $k = 0$ et $l = 2$. En général, si on a un algorithme pour $k = a$ et $l = b$, alors on a aussi un algorithme pour $k = b$ et $l = a$; il suffit juste de passer au graphe complémentaire avant d'exécuter l'algorithme.

Considérons le cas $k = l = 1$, correspondant à Maximum Split Subgraph. On va brièvement décrire un algorithme pour le résoudre.

Les sommets demi-noirs ne peuvent pas appartenir à l'ensemble des sommets blancs C (la clique) et les sommets demi-blancs ne peuvent pas appartenir à l'ensemble des sommets noirs I (l'ensemble stable). On utilise les mêmes poids que dans le chapitre 3. Les réductions sont celles du chapitre 3 avec des légères modifications : on ne traite pas les sommets demi-blancs dans les branchements et les réductions 2, 3 et 5 et on supprime la réduction 6. L'idée est qu'on cherche l'ensemble stable comme dans l'algorithme pour Maximum Bipartite Subgraph et qu'on cherche la clique de la même façon sur le graphe complémentaire. La réduction 6 est remplacée par les branchements du cas 1 dans l'algorithme ci-dessous.

1. $V = F$ (il n'existe pas de sommet demi-coloré)
   Pour ce problème, on ne peut pas simplement colorer un sommet quelconque en demi-noir. Pour MAXIMUM BIPARTITE SUBGRAPH, l'ensemble des sommets blancs et celui des sommets noirs était complètement interchangeable, ce qui n'est pas le cas ici.

   a) $\exists u : d(u) \geq 5$
   Colorer u en noir (l'ajouter à I) $(-f)$ ou en demi-blanc $(+h - f)$. Si u est coloré en noir, au moins 5 voisins deviennent demi-blancs $(+5h - 5f)$.

   $$T(n) \leq T(n + 5h - 6f) + T(n + h - f)$$

   b) $n \leq 11$
   résoudre le problème en temps constant

   c) $\exists u : 3 \leq d(u) \leq 4$
   Colorer u en noir $(-f)$, en blanc $(-f)$ ou supprimer u $(-f)$. Si u est coloré en noir, ses voisins deviennent demi-blancs $(+d(u)h - d(u)f)$. S'il est coloré en blanc, tous ses non-voisins deviennent demi-noirs $(+(11 - d(u))h - (11 - d(u))f)$.

   $$\begin{aligned} T(n) &\leq T(n + 4h - 5f) + T(n + 7h - 8f) + T(n - f) \\ T(n) &\leq T(n + 3h - 4f) + T(n + 8h - 9f) + T(n - f) \end{aligned}$$

   d) $\Delta(G) \leq 2$
   résoudre le problème en temps polynomial

2. $B \neq \varnothing$
   On utilise les branchements de la section 3.2 avec les mêmes règles de récurrence, mais sans les cas symétriques pour les sommets demi-blancs.

3. $B = \varnothing$ et $W \neq \varnothing$
   On utilise les mêmes règles de branchements sur le graphe demi-coloré complémentaire. Le complémentaire d'un graphe demi-coloré $G = (V, F, B, W, E)$ est le graphe demi-coloré $\bar{G} = (V, F, W, B, \bar{E})$ avec $\bar{E} = V \times V - E$.

Les règles de branchement du cas 1 ont été choisies de façon à ne pas augmenter le temps d'exécution obtenu par les récurrences du chapitre 3 et cet algorithme s'exécute donc en temps $O^*(1.6330^n)$ également.

Pour résoudre MAXIMUM k, l-SUBGRAPH dans le cas général, une manière est d'énumérer les k, l-sous-graphes maximaux par induction et de mémoriser un de ces sous-graphes de taille maximale. Un k, l-sous-graphe est maximal par induction si on ne peut pas obtenir un autre k, l-sous-graphe en lui ajoutant un sommet du graphe initial. Dans [Bys04], Byskov présente des algorithmes pour énumérer des sous-graphes k-colorables, ce qui est équivalent à énumérer des $k, 0$-sous-graphes. Il est immédiat qu'on peut utiliser ces algorithmes pour énumérer des $0, l$-sous-graphes en raisonnant sur le graphe complémentaire.

Mais aussi pour les autres cas, les techniques présentées dans [Bys04, chapitre 5] peuvent être utilisées pour construire des algorithmes afin d'énumérer les k, l-sous-graphes maximaux par induction, avec une petite restriction sur l'énumération des $1, 1$-sous-graphes.

Par manque de place, je ne vais pas rentrer dans les détails, mais juste présenter le tableau avec les constantes α dans le temps d'exécution $O^*(\alpha^n)$ pour les différentes valeurs de k et l. Les algorithmes pour $k + l \geq 3$ et pour $k = l = 1$ sont des algorithmes par programmation dynamique et les autres utilisent des arbres de recherche.

k\l	0	1	2	3	≥4
0	-	1.4423*	1.7724*	2.1809	2.4023
1	1.4423*	1.8613*	2.1809	2.4023	2.4023
2	1.7724*	2.1809	2.4023	2.4023	2.4023
3	2.1809	2.4023	2.4023	2.4023	2.4023
≥4	2.4023	2.4023	2.4023	2.4023	2.4023

FIG. 4.1: Lister les k, l-sous-graphes maximaux par induction

* Pour ces cas, il existe des meilleurs algorithmes pour trouver des k, l-sous-graphes de taille maximale. Pour ENSEMBLE STABLE ($k = 1, l = 0$) et CLIQUE ($k = 0, l = 1$), il existe un algorithme en $O^*(1.1889^n)$ [Rob01] ; pour $k + l = 2$, il existe des algorithmes en $O^*(1.6330^n)$ présentés dans ce rapport.

4.2 Maximum Induced Subgraph without H

Une autre manière de caractériser les problèmes précédents est en donnant leurs sous-graphes induits interdits.

- Ensemble Stable ≡ MAXIMUM INDUCED SUBGRAPH WITHOUT K_2
- Clique ≡ MAXIMUM INDUCED SUBGRAPH WITHOUT $\overline{K_2}$
- Maximum Bipartite Subgraph ≡ MAXIMUM IND. SUBGRAPH WITHOUT C_i, i *impair*
- Maximum Split Subgraph ≡ MAXIMUM INDUCED SUBGRAPH WITHOUT $2K_2, C_4, C_5$

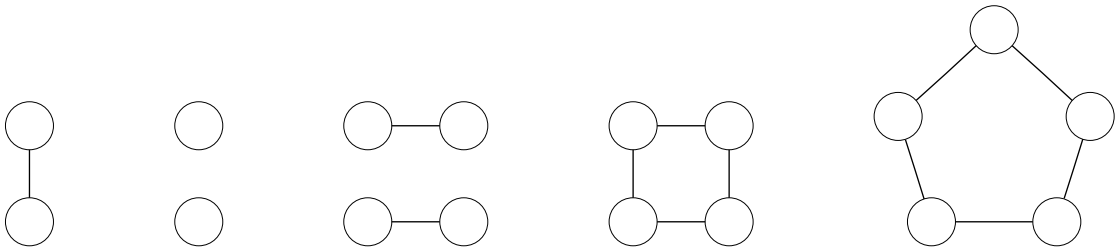


FIG. 4.2: K_2, $\overline{K_2}$, $2K_2$, C_4, C_5

Je vais présenter ici un algorithme qui résout le problème MAXIMUM INDUCED SUBGRAPH WITHOUT H où H est un graphe de taille fixe ayant un temps d'exécution meilleur que la complexité triviale $O^*(2^n)$. Ce problème consiste donc à trouver un sous-graphe induit de G de taille maximale qui n'a pas le graphe H comme sous-graphe induit.

Un algorithme trivial pour ce problème serait d'énumérer tous les 2^n sous-graphes induits de G et vérifier pour chacun s'il a H comme sous-graphe induit. L'algorithme présenté ici va procéder comme suit :

On cherche d'abord en temps polynomial tous les sous-graphes de G isomorphes à H et on construit une collection d'ensembles C dont chaque ensemble correspond aux sommets d'un sous-graphe "interdit" de G. Il ne peut pas exister plus de $n^{|H|}$ sous-graphes interdits et il est facile de trouver un algorithme polynomial pour construire cette collection C, par exemple en analysant tous les sous-ensembles de sommets de taille $|H|$.

Ayant fait cette transformation, on a ramené le problème MAXIMUM INDUCED SUBGRAPH WITHOUT H avec $|H|$ borné au problème MINIMUM k-HITTING SET, où k est le nombre de sommets de H ; il suffit de supprimer les sommets correspondant au Hitting Set du graphe initial. Cette transformation marche aussi si on est confronté à un nombre limité de sous-graphes interdits de taille fixe ; k est alors le nombre de sommets du plus grand sous-graphe interdit.

Problème : MINIMUM K-HITTING SET
Entrée : Une collection C de sous-ensembles d'un ensemble fini S. Les sous-ensembles contiennent au plus k éléments.
Sortie : Un Hitting Set de cardinalité minimale. Un Hitting Set pour C est un sous-ensemble $S' \subseteq S$ tel que S' contient au moins un élément de chaque sous-ensemble dans C.

L'algorithme va procéder comme suit. On choisit un sous-ensemble A de cardinalité minimale et on parcourt ses éléments $a_1, a_2, ..., a_t$: un des a_i doit appartenir au Hitting Set.

La première possibilité est qu'on choisit a_1 et on peut donc supprimer tous les sous-ensembles qui contiennent a_1.

La deuxième possibilité est que a_1 n'appartient pas au Hitting Set, mais a_2 ; on supprime alors a_1 de tous les sous-ensembles et on supprime tous les sous-ensembles qui contiennent a_2.

En général, si on choisit a_i, on supprime tous les $a_{1..i-1}$ des sous-ensembles (ce qui ne peut pas produire des sous-ensembles vides, car on choisit A de cardinalité minimale) et on supprime tous les sous-ensembles qui contiennent a_i.

Dans le pseudo-code suivant, les réductions et les branchements sont cachés dans la boucle "Pour" : si $t = 1$, on effectue une réduction et si $t \geq 2$, on effectue t branchements.

Algorithme 4.3 k-HITTING SET

```
k-HittingSet ( C : Collection d'Ensembles ) : Ensemble
   Si C = ∅ alors
      return ∅
   Sinon
      A = {a_1, a_2, ..., a_t} ∈ C : A est de cardinalité minimale
      Pour i = 1 à t faire
         R_i ←{a_i} ∪ k-HittingSet( ChoixElement(i, A, C) )
      return R_j : j ∈ 1..t et |R_j| = min_{i=1..t} {|R_i|}


ChoixElement (i : Entier, A = {a_1, a_2, ..., a_t} : Ensemble,
              C : Collection d'Ensembles) : Collection d'Ensembles
   Si i ≠ 1 alors
      Pour j = 1..i − 1 faire
         enlever a_j de tous les sous-ensembles de C
   enlever tous les sous-ensembles de C qui contiennent a_i
   return C
```

Soit $e = |S|$ le nombre d'éléments dans S, ce qui correspond au nombre de sommets différents dans les sous-graphes interdits du problème MAXIMUM INDUCED SUBGRAPH WITHOUT H. On a donc que $e \leq n$.

Si $|A| = 1$, on supprime un éléments de S : $T(e) \leq T(e-1)$.

Si $|A| = 2$, on supprime soit un, soit deux éléments de S : $T(e) \leq T(e-1) + T(e-2)$.

Si $|A| = 3$, on supprime soit 1, soit 2, soit 3 éléments de S :
$T(e) \leq T(e-1) + T(e-2) + T(e-3)$.

Si $|A| = i$, on supprime soit 1, soit 2, ..., soit i éléments de S :
$T(e) \leq T(e-1) + T(e-2) + ... + T(e-i)$.

Pour chaque valeur de k, on doit donc résoudre un système de k règles de récurrence pour trouver une borne supérieure du temps d'exécution de l'algorithme, ce qui donne les valeurs de la table 4.1.

Le cas où $k = 2$ est équivalent au problème VERTEX COVER sur le graphe $G = (S, C)$.

Problème : VERTEX COVER
Entrée : Un graphe non-orienté $G = (V, E)$
Sortie : Une couverture de sommets (vertex cover) de cardinalité minimale. Une couverture de sommets est un sous-ensemble de sommets $V' \subseteq V$ tel que pour chaque arête $\{u, v\}$, au moins un sommet parmi u et v appartient à V'.

---

k	Temps
2	$O^*(1.6181^n)$
3	$O^*(1.8393^n)$
4	$O^*(1.9276^n)$
5	$O^*(1.9660^n)$
6	$O^*(1.9836^n)$
10	$O^*(1.99902^n)$
20	$O^*(1.99999905^n)$

TAB. 4.1: Temps d'exécution pour k-Hitting Set

VERTEX COVER peut être résolu en cherchant un ensemble stable I pour le graphe G et en retournant ensuite le sous-ensemble de sommets $V-I$. On peut donc utiliser l'algorithme de Robson [Rob01] en $O^*(1.1889^n)$ pour résoudre VERTEX COVER et 2-HITTING-SET.

De la même façon, on peut facilement ramener le problème k-HITTING SET au problème MINIMUM TRANSVERSALS IN RANK-k HYPERGRAPHS, une généralisation de la couverture de sommets.

Problème : MINIMUM TRANSVERSAL
Entrée : Un hypergraphe $\mathcal{H} = (V, E)$, c'est-à-dire un graphe non-orienté où les arêtes peuvent relier un ou plusieurs sommets. Un hypergraphe de rang k a des arêtes d'au plus k sommets.
Sortie : Un transversal de cardinalité minimale pour $\mathcal{H}$. Un transversal d'un hypergraphe $\mathcal{H} = (V, E)$ est un sous-ensemble de sommets $V' \subseteq V$ qui contient au moins un sommet de chaque arête.

Pour les hypergraphes de rang 3, et donc aussi pour 3-HITTING-SET, on peut utiliser l'algorithme de Wahlström [Wah04] pour améliorer le temps à $O^*(1.6538^n)$ en utilisant un espace polynomial ou à $O^*(1.6316^n)$ en utilisant un espace exponentiel.

Au cours de ce stage, je me suis intéressé aux sous-graphes avec un autre graphe interdit particulier : MAXIMUM P_3-FREE SUBGRAPH. En utilisant la propriété qu'un graphe sans P_3 est une collection disjointe de cliques et en utilisant des sommets demi-colorés, j'ai conçu un algorithme en $O^*(1.5789^n)$ pour ce problème.

5 Conclusion

Ce stage m'a permis d'entrer en contact avec le monde de la recherche en informatique théorique et de voir ce qui se passe surtout en algorithmique de nos jours. Les chercheurs essayent de nouveau à résoudre les problèmes NP-difficiles exactement, même si les algorithmes ne seront pas applicables à des grandes entrées. Les arbres de recherche sont un joli moyen pour améliorer un algorithme *brute-force* et considérer seulement les cas qui contiennent à coup sûr une solution optimale. Malheureusement, on n'arrive pas à déterminer la vraie complexité au pire des cas avec les techniques d'analyse actuelles, mais seulement une borne supérieure. C'est aussi pour ça que les chercheurs s'intéressent parfois à des bornes inférieures de la complexité d'un algorithme, en cherchant des instances pour lesquelles l'algorithme met beaucoup de temps.

Les problèmes traités dans ce stage sont des problèmes fondamentaux qui généralisent deux des problèmes NP-difficiles les plus importants sur les graphes : Ensemble Stable et Clique. Les sous-graphes cherchés ici correspondent souvent à des classes de graphes particulières, et on pourrait utiliser ces algorithmes pour corriger des anomalies dans les entrées, par exemple.

Finalement, ce stage m'a plu énormément et j'espère avoir la possibilité de continuer à faire de la recherche dans cette direction.

Bibliographie

[Bei99] Richard Beigel. Finding maximum independent sets in sparse and general graphs. In *SODA '99 : Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, 1999.

[Bys04] Jesper Makholm Byskov. Exact algorithms for graph colouring and exact satisfiability. Dissertation Series DS-04-4, BRICS, Department of Computer Science, University of Aarhus, November 2004. PhD thesis.

[CKX03] Jianer Chen, Iyad A. Kanj, and Ge Xia. Labeled search trees and amortized analysis : Improved upper bounds for NP-hard problems. In *ISAAC*, pages 148–157, 2003.

[DGH+02] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2\text{-}2/(k+1))^n$ algorithm for k-SAT based on local search. *Theor. Comput. Sci.*, 289(1) :69–83, 2002.

[Epp03] David Eppstein. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms & Applications*, 7(2) :131–140, 2003. Special issue for WADS'01.

[Epp04] David Eppstein. Quasiconvex analysis of backtracking algorithms. In *Proc. 15th Symp. Discrete Algorithms*, pages 781–790. ACM and SIAM, January 2004.

[FKW04] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *LNCS : Proceeding of WG2004*, pages 245–256, 2004.

[Gra04] Fabrizio Grandoni. *Exact Algorithms for Hard Graph Problems*. PhD thesis, Università di Roma "Tor Vergata", Roma, Italy, March 2004.

[HK62] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1) :196–210, mars 1962.

[KL99] O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. preprint, 71 pages, available from `http://cs-svr1.swan.ac.uk/~csoliver/papers.html`, February, 1999.

[Law76] Eugene L. Lawler. A note on the complexity of the chromatic number problem. *Inf. Process. Lett.*, 5(3) :66–67, 1976.

[Rob86] J. M. Robson. Algorithms for maximum independent sets. *J. Algorithms*, 7(3) :425–440, 1986.

[Rob01] Mike Robson. Finding a maximum independent set in time $O(2^{n/4})$? Technical report, Université Bordeaux 1, Département d'Informatique, 2001.

[RS04] Bert Randerath and Ingo Schiermeyer. Exact algorithms for MINIMUM DOMINATING SET. Technical report, Zentrum für Angewandte Informatik Köln, Lehrstuhl Speckenmeyer, April 2004.

[TT77] R.E. Tarjan and A.E. Trojanowski. Finding a maximum independent set. *SIAM J. Comput.*, 6(3) :537–546, 1977.

[Wah04] Magnus Wahlström. Exact algorithms for finding minimum transversals in rank-3 hypergraphs. *J. Algorithms*, 51(2) :107–121, 2004.

[Woe01] Gerhard J. Woeginger. Exact algorithms for NP-hard problems : A survey. In *Combinatorial Optimization*, pages 185–208, 2001.

[Yan78] Mihalis Yannakakis. Node- and edge-deletion NP-complete problems. In *STOC '78 : Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 253–264, New York, NY, USA, 1978. ACM Press.