IFT800 - Algorithmique
Notes de cours

Manuel Lafond
Université de Sherbrooke

Table des matières

Chapitre 1

Introduction

Dans ce cours, nous discuterons de techniques pour résoudre des problèmes algorithmiques difficiles de façon efficace. De notre point de vue, un problème est *difficile* s'il n'y a aucun algorithme en temps polynomial connu permettant de le résoudre. Notre but est de développer des algorithmes pour ces problèmes qui offrent *des garanties théoriques* sur la qualité de la solution obtenue ou sur leur rapidité. Nous étudierons deux approches :

1. les algorithmes d'approximation, qui garantissent de toujours retourner une solution à un facteur près de l'optimal ;
2. les algorithmes à complexité paramétrée, qui garantissent un temps exponentiel, mais seulement par rapport à un paramètre k qui est petit en pratique.

Par exemple, le problème du commis voyageur est NP-complet, mais il existe un algorithme en temps polynomial qui retourne un itinéraire qui est, au pire, deux fois l'optimal (si les distances satisfont l'inégalité triangulaire). Ou encore, le problème VERTEX-COVER est NP-complet, mais si k est le nombre de sommets dans une couverture, il existe un algorithme en temps $O(2^k \cdot n)$ (VERTEX-COVER est défini dans le chapitre 2).

En d'autres termes, persone ne connaît de solution efficace pour résoudre les problèmes NP-complet. Ceci ne veut pas dire qu'on peut implémenter n'importe quelle heuristique, car celles-ci peuvent retourner des solutions complètement erronées sur certaines instances. Nous voulons implémenter des algorithmes dont la performance est *démontrable*, que ce soit au niveau de l'approximation ou de la complexité.

1.1 Qu'est-ce qu'un problème NP-complet ?

Si un problème est NP-complet, il n'existe à ce jour aucun algorithme en temps polynomial connu qui permet de le résoudre. La définition formelle d'un problème NP-complet sort du cadre de ce cours. L'intuition qui nous suffira pour ce cours est qu'un premier problème NP-complet a été découvert au début des années 1970. Ce problème s'appelle SAT et consiste à déterminer si une expression booléenne peut être satisfaite. Par exemple, soit l'expression booléenne

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3})$$

Dans le problème SAT, on veut savoir si on peut affecter la valeur True ou False à chacun des x_i pour que ϕ évalue à True. Une solution ici serait $x_1 = True, x_2 = True, x_3 = False$. Cet exemple est facile et a plusieurs solutions, mais sur de plus grandes instances, personne ne connaît d'algorithme en temps polynomial pour décider s'il est possible que $\phi = True$. En fait, le meilleur algorithme connu consiste à tester chque combinaison des valeurs des x_i, ce qui prend un temps $\Omega(2^n)$.

Bref, personne ne connaît d'algorithme efficace pour SAT, malgré des années de recherche. Il s'avère que SAT peut se *réduire* à d'autres problèmes. par exemple, on a montré que trouver une clique de taille maximum dans un graphe est équivalent au problème de SAT. C'est-à-dire, on peut transformer une instance ϕ de SAT en un graphe G tel que si on avait un algorithme $O(n^c)$ qui trouve la plus grosse clique de G, cet algorithme pourrait déterminer si SAT est satisfaisable ou non. Tout se passe dans la tranformation de ϕ en G afin de préserver l'équivalence.

Nous vous invitons à suivre le cours IFT503/IFT711 pour plus de détails à ce sujet. Pour nos fins, il suffit de savoir que si un problème est NP-complet, il n'admet probablement pas d'algorithme en temps polynomial car un tel algorithme permettrait de résoudre SAT, chose que les plus grands génies de la dernière décennie n'ont pas pu faire.

À moins d'indication contraire, tous les problèmes étudiés dans ce cours sont NP-complet.

1.2 Quels sont les préalables pour ce cours

Nous supposons que les techniques commune en algorithmique sont connues. Ceci inclut

— une connaissance des concepts principaux des mathématiques discrètes,

incluant les ensembles, séquences, les permutations, ainsi que les notations et opérateurs associés ($\cap, \cup, |X|$, etc.).
- une connaissance des techniques de preuve communes : preuve directe, preuve par contradiction, preuve par induction, preuve par contre-exemple.
- une familiarité avec la notation O. En particulier, il sera pratique de pouvoir évaluer rapidement la complexité d'un algorithme en parcourant seulement la structure du code (boucles, appels récursifs, ...).
- une familiarité avec les graphes. Un graphe $G = (V, E)$ est une structure où V est l'ensemble des sommets et E les arêtes, où E contient des paires de sommets.
- une connaissance des algorithmes diviser-pour-régner et du théorème maître pour l'analyse de récurrences. Une connaissance sur les récurrences homogènes linéaire pourra aussi aider (mais nous en rediscuterons).
- une connaissance de la programmation dynamique. En particulier, comment établir une récurrence qui exprime l'optimalité, et comment transposer cette récurrence en code.

Il est important de noter que ce cours est de nature *théorique*. Les algorithmes d'approximation et à complexité paramétrée sont maintenant très utiles en pratique, et beaucoup des approches présentées sont implémentées dans des librairies. Dans le cadre de ce cours, nous n'allons pas nous attarder sur les considérations pratiques, ni même tenter de nous convaincre des applications des notions présentées. Les méthodes présentées sont choisies pour illustrer les techniques les plus commnues, et non pour illustrer les algorithmes les plus applicables.

L'objectif de ce cours est de vous former à appliquer ces techniques à de *nouveaux* problèmes — l'intérêt pratique de ce cours ne réside donc pas dans les algortihmes spécifiques choisis, mais plutôt dans les *idées* derrière ces algorithmes que vous pourrez appliquer sur les futurs problèmes que vous rencontrerez.

Première partie

Algorithmes d'approximation

Chapitre 2

Algorithmes d'approximation

En approximation, on cherche à créer un algorithme en temps polynomial qui retourne une solution possiblement sous-optimale, mais aussi près de l'optimal que possible, et ce de façon **démontrable**. La plupart du temps, nous n'allons pas nous soucier de la complexité exacte de nos algorithmes — notre seul intérêt est le temps polynomial, même s'il est $O(n^{100})$.

2.1 Approximation d'un problème de minimisation

Soit P un problème de minimisation, c'est-à-dire un problème dans lequel on veut minimiser une certaine fonction objective parmi un ensemble de solutions faisables. Pour une instance X de P, soit $OPT(X)$ la valeur d'une solution optimale pour X.

Soit A un algorithme qui produit une solution faisable pour toute instance de P, et soit $APP(X)$ la valeur de la solution émise par A lorsqu'on lui donne l'entrée X.

On dit que A est une c-approximation si, pour toute instance X de P,

$$APP(X) \leq c \cdot OPT(X)$$

Donc quand $c > 1$, l'algorithme peut donner une solution sous-optimale, mais pas plus que c fois trop grande.

2.2 Approximation d'un problème de maximisation

Si P est un problème de maximisation, on dit que A est une c-approximation si, pour toute instance X de P,

$$APP(X) \geq c \cdot OPT(X)$$

Donc quand $c < 1$, l'algorithme peut donner une solution sous-optimale, mais pas plus que c fois trop petite. Notez que dans certains textes, on requiert que $c \geq 1$ et $APP(X) \geq \frac{1}{c} \cdot OPT(X)$. Ceci est pour que $c > 1$ dans les deux types de problèmes. Dans ce cours, nous préférons utiliser c directement et non son inverse.

Dans les deux types de problèmes, il se peut que c soit une fonction de n. Par exemple, il existe une $(\log n)$-approximation au problème SET-COVER, que nous verrons plus loin. Mettons aussi l'emphase sur le fait que la solution retournée par APP doit toujours être faisable, c'est-à-dire être une solution valide pour le problème P.

Dans le suite de ce document, nous allons souvent écrire OPT au lieu de $OPT(X)$ et APP au lieu de $APP(X)$.

2.3 Un premier exemple avec 3-SET-COVER

Notre premier exemple d'approximation sera trivial, mais instructif, car il nous permettra d'introduire une technique fondamentale en approximation. Nous étudions le problème 3-SET-COVER. On nous donne des ensembles $S_1, \ldots, S_m$ de taille 3 et on nous demande de couvrir un univers U de taille n avec un nombre minimum d'ensembles.

3-SET-COVER
Entrée : un univers $U = \{u_1, \ldots, u_n\}$ et une collection d'ensembles $S = \{S_1, \ldots, S_m\}$ tel que $|S_i| = 3$ pour tout $i \in [m]$
Sortie : un sous-ensemble $S^* \subseteq S$ de taille minimum tel que $\bigcup_{S_i \in S^*} S_i = U$.

On suppose que chaque $u_i \in U$ a un ensemble de S qui le contient. Voici

un exemple d'instance :

$U = \{1, 2, 3, 4, 5\}$ $\quad S_1 = \{1, 3, 4\}, S_2 = \{1, 4, 5\}, S_3 = \{2, 3, 5\}$
Une solution optimale : choisir $S^* = \{S_1, S_3\}$

Ici, une solution faisable est tout $S^* \subseteq S$ tel que l'union des éléments de S^* donne U. La valeur qu'on veut minimiser est $|S^*|$. Voici une 3-approximation.

fonction *setcover(U, S)*
  $S^* = \{\}$
  pour $i = 1..n$ **faire**
    Soit u_i le i-ème élément de U
    si u_i *n'est pas couvert par* S^* **alors**
      Ajouter à S^* n'importe quel ensemble qui contient u_i
  fin
  return S^*

Il est clair que cet algorithme s'exécute en temps polynomial et retourne toujours une solution valide, donc un S^* qui couvre tous les u_i. Mais comment sait-on que c'est une 3-approximation ? Commençons par nous demander ce que vaut OPT. Puisqu'il y a n éléments à couvrir et que chaque S_i peut en couvrir 3 au maximum, on sait que

$$OPT \geq \left\lceil \frac{n}{3} \right\rceil \geq \frac{n}{3}$$

En contre-partie, que vaut APP ? Dans le pire cas, on ajoute un ensemble à S^* pour chaque élément de U. Donc

$$APP = |S^*| \leq n$$

On obtient

$$APP \leq n = 3 \cdot \frac{n}{3} \leq 3 \cdot OPT$$

Donc, $APP \leq 3 \cdot OPT$ et c'est une 3-approximation.

2.4 Technique fondamentale : trouver une borne sur OPT

L'exemple ci-haut illustre une idée très importante en approximation. Pour montrer que nous ne sommes pas trop loin de OPT, on trouve une borne sur OPT, une borne sur APP, et on les compare. C'est donc très simple : on n'a qu'à réaliser les deux étapes suivantes

1. établir une borne sur OPT ;
2. montrer que APP n'est pas trop loin de cette borne.

Si on a un problème de minimisation, ceci prend la forme suivante : on montre que $OPT \geq k$ pour un certain k. Ensuite, on montre que $APP \leq c \cdot k$. Ces deux faits donnent la chaîne d'inégalités

$$APP \leq c \cdot k \leq c \cdot OPT$$

et on obtient une c-appoximation.

Si on a un problème de maximisation, on montre que $OPT \leq k$ pour un certain k. Ensuite, on montre que $APP \geq c \cdot k$, ce qui donne

$$APP \geq c \cdot k \geq c \cdot OPT$$

La difficulté de cette technique consiste à trouver ce "point milieu" où OPT et APP se rejoignent. Nous verrons plusieurs exemples pendant ce cours.

2.5 Une 2-approximation à la couverture par sommet

Soit $G = (V, E)$ un graphe. Un ensemble $X \subseteq V$ est *couvrant* si, $\forall uv \in E$, on a $u \in X$ ou $v \in X$ (ou les deux). Dit autrement, X "touche" à toutes les arêtes.

VERTEX-COVER
Entrée : un graphe $G = (V, E)$
Sortie : un ensemble couvrant $X \subseteq V$ de taille minimum

Pour approximer ce problème, on observe que pour chaque arête $uv \in E$, il faut inclure soit u, soit v dans la solution X. Ne sachant pas lequel inclure, notre algorithme d'approximation inclut les deux. Ceci aura pour effet de couvrir d'autres arêtes, toutes celles qui sont incidentes à u et v. Notre algorithme trouve, de façon itérative, des arêtes non-couvertes et ajoute ses deux bouts jusqu'à couverture de E en entier.

fonction *vc-matching*$(G = (V, E))$
  $X = \{\}$
  tant que X *ne couvre pas toutes les arêtes* **faire**
    Soit $uv \in E$ une arête non-couverte par X
    $X \leftarrow X \cup \{u, v\}$
  fin
  return X

Cet algorithme porte le nom *vc-matching* car l'ensemble des arêtes pour lesquelles on a ajouté les deux bouts forment un *matching*, c'est-à-dire un ensemble d'arêtes qui ne partagent aucun sommet. Il est possible de montrer que cet algorithme est une 2-approximation. En fait, tout algorithme qui retourne un matching est une 2-approximation.

Théorème 1. *L'algorithme* vc-matching *est une 2-approximation au problème VERTEX-COVER.*

Démonstration. Soit X^* une couverture minimum de G et soit $OPT = |X^*|$. Soient $u_1v_1, u_2v_2, \ldots, u_kv_k$ les arêtes de G pour lesquelles l'algorithme a ajouté les deux sommets, et soit X la couverture retournée par l'algorithme. On observe que pour tout $i \neq j$, $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$ (ceci est parce qu'une fois qu'on a ajouté u_i et v_i, toutes les arêtes touchant u_i et v_i sont couvertes — si vous ne le voyez pas, prenez le temps de vous en convaincre).

De plus, pour tout $i \in [k]$, X^* doit contenir soit u_i, soit v_i (ou les deux). Puisque tous les u_i et v_i sont distincts, ceci implique que $|X^*| \geq k$. La couverture retournée par *vc-matching* est $X = \{u_1, v_1, u_2, v_2, \ldots, u_k, v_k\}$, et donc $|X| = 2k$. On obtient

$$APP = |X| = 2k \leq 2|X^*| \leq 2OPT$$

et donc on a une 2-approximation. □

Notez encore l'utilisation de la technique. On a d'abord trouvé une borne sur OPT, cette fois-ci en lien direct avec la solution retournée par l'algorithme. On a comparé OPT sur la taille de notre solution et avons obtenu une 2-approximtion. Malgré sa simplicité, on ne connaît pas d'algorithme offrant un meilleur taux d'approximation. Une conjecture bien connue stipule que si les problèmes NP-complets n'ont pas d'algorithme en temps polynomial, alors il est impossible de faire mieux que cet algorithme en termes d'approximation.

Notons que nous sommes passés un peu vite sur l'affirmation $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$. Ceci n'aurait pas été toléré dans un cours de 1re ou 2ème année, mais nous nous permettons quelques sauts de la sorte, étant donné la nature avancée du cours. Lorsque vous rédigerez vos propres preuves, il est important de vous demander si de tels sauts sont appropriés — ils entraînent souvent des preuves erronées (même des chercheurs reconnus tombent dans ces pièges). Dans le doute, détaillez !

2.6 Un dernier exemple simple avec MAX-SAT

Dans le monde des expressions booléennes, une *clause* est un ensemble de variables booléennes liées par des "ou" logiques. Une variable booléenne x_i peut être *positive* (x_i) ou *négative* ($\overline{x}_i$).

Par exemple, voici une clause C impliquant trois variables :

$$C = x_1 \vee \overline{x}_2 \vee x_3$$

Une *assignation* attribue la valeur $True$ ou $False$ à chaque variable. Si $x_i = True$, toutes les occurrences positives de x_i évaluent à $True$ et les occurrences négatives à $False$. Si $x_i = false$, toutes les occurrences négatives de x_i évaluent à $True$ et les occurrences positives à $False$. Par exemple, si on prend l'assignation $x_1 = True, x_2 = False, x_3 = False$, la clause ci-haut se traduit par

$$C = True \vee True \vee False$$

ce qui évalue à $True$.

Une clause est *satisfaite* par une assignation A si la clause évalue à $True$ avec cette assignation. Pour ce faire, il faut qu'au moins une variable de la clause évalue à $True$. La seule façon de ne *pas* satisfaire la clause ci-haut est $x_1 = False, x_2 = True, x_3 = False$.

Dans le problème MAX-SAT, on reçoit un ensemble de clauses (avec nombre de variables arbitraire), et on veut trouver une assignation qui maximise le nombre de clauses satisfaites.

MAX-SAT
Entrée : un ensemble de clauses $C_1, \ldots, C_m$ sur variables booléennes $x_1, \ldots, x_n$
Sortie : une assignation A qui satisfait le maximum de clauses parmi $C_1, \ldots, C_m$.

Voici une $\frac{1}{2}$-approximation.

fonction *msat($C_1, \ldots, C_m$ sur variables $x_1, \ldots, x_n$)*
  Soit A l'assignation qui met
   $x_1 = True, x_2 = True, \ldots, x_n = True$
  si *A satisfait au moins $m/2$ clauses* **alors**
    return A
  Soit $\overline{A}$ l'assignation qui met
   $x_1 = False, x_2 = False, \ldots, x_n = False$
  return $\overline{A}$

Théorème 2. *L'algorithme msat est une $\frac{1}{2}$-approximation.*

Démonstration. Il y a m clauses, donc trivialement, $OPT \leq m$ (notre borne sur OPT). Si l'algorithme retourne A, alors $APP \geq m/2$ et on a

$$APP \geq m/2 \geq OPT/2$$

Sinon, supposons que A satisfait $k < m/2$ clauses. On remarque que toute clause non-satisfaite par A a la forme $C_i = (\overline{x}_a \vee \overline{x}_b \vee \overline{x}_c)$. Une telle clauses sera satisfaite par $\overline{A}$. Donc, $\overline{A}$ satisfait au moins $m - k > m - m/2 = m/2$ clauses. On déduit que si l'algorithme retourne $\overline{A}$, on a toujours $APP \geq m/2 \geq OPT/2$.

On a donc une $\frac{1}{2}$-approximation. □

2.7 Nos analyses peuvent-elles être raffinées ?

Les ratios d'approximation obtenus dépendent de nos capacité à les démontrer. On peut alors se demander si, avec des analyses plus raffinées, on pourrait montrer que nos algorithmes sont meilleurs qu'on le pense. Par

exemple, peut-être que *msat* ci-haut est en fait une $\frac{5}{6}$-approximation, mais que nous sommes incapables de le démontrer.

Lorsqu'on produit un algorithme et une preuve d'approximation, on aime argumenter que notre analyse est *serrée* (*tight* en anglais), c'est-à-dire qu'on ne pourrait pas améliorer le ratio d'approximation démontré. Pour ce faire, il suffit de donner un exemple d'instance (un seul suffit !) dans lequel notre algorithme atteint exactement le ratio démontré.

Reprenons l'algorithme pour 3-SET-COVER.

```
fonction setcover(U, S)
    S* = {}
    pour i = 1..n faire
        Soit u_i le i-ème élément de U
        si u_i n'est pas couvert par S* alors
            Ajouter à S* n'importe quel ensemble qui contient u_i
    fin
    return S*
```

Est-il possible que cet algorithme fasse mieux qu'une 3-approximation ? Non, car il y a certaines instances sur lesquelles cet algorithme donne $APP = 3 \cdot OPT$. Ou plus précisément, on va construire une famille d'instances telles que $APP = (3 - \varepsilon)OPT$, avec ε qui tend vers 0 lorsque n tend vers l'infini. Le théorème suivant formule ceci d'une façon alternative.

Théorème 3. *Pour tout $\varepsilon > 0$, il existe une instance de 3-SET-COVER telle que $APP \geq (3 - \varepsilon)OPT$.*

Démonstration. Soit $U = \{u_1, \ldots, u_n\}$, où n est un multiple de 3. Soient les

ensembles S contenant

$$
\begin{aligned}
&S_1 = \{u_1, u_2, u_3\}\\
&S_2 = \{u_1, u_2, u_4\}\\
&\dots\\
&S_{n-2} = \{u_1, u_2, u_n\}\\
&S'_2 = \{u_4, u_5, u_6\}\\
&S'_3 = \{u_7, u_8, u_9\}\\
&\dots\\
&S'_{n/3} = \{u_{n-1}, u_{n-2}, u_n\}
\end{aligned}
$$

Puisque l'algorithme ne spécifie pas comment choisir l'ensemble pour couvrir un nouvel élément, on peut supposer qu'il fait le pire choix possible. On peut se convaincre que l'algorithme pourrait retourner $S^* = \{S_1, S_2, \dots, S_{n-2}\}$ de taille $n-2$. Pourtant, la solution optimale est $\{S_1, S'_2, S'_3, \dots, S'_{n/3}\}$ de taille $n/3$.

Si on veut connaître le ratio d'approximation, on doit trouver c tel que $APP = c \cdot OPT$, c'est-à-dire le c tel que $n - 2 = c \cdot (n/3)$. On trouve $c = 3 - 6/n$.

$$APP = n - 2 = (3 - 6/n)OPT$$

En faisant tendre n vers l'infini, on obtient le résultat. □

Chapitre 3

Approximation avec l'approche fondamentale

Dans ce chapitre, nous verrons divers exemples d'application de la technique fondamentale. Comme nous le verrons, il faut parfois être créatif dans notre choix de borne sur OPT et sur APP.

3.1 Le problème du commis voyageur, version métrique

Soit $G = (V, E)$ un graphe complet, c'est-à-dire que toute arête possible est présente (et donc $|E| = \binom{n}{2} = n(n-1)/2$). Soit $f : E \to \mathbb{R}^{>0}$ une fonction qui attribue un poids positif à chaque arête. On dit que f satisfait *l'inégalité triangulaire* si, pour tout triplet u, v, w de sommets distincts, on a

$$f(uw) \leq f(uv) + f(vw)$$

On dira aussi que f est une *métrique*. L'inégalité triangulaire est une condition souvent applicable en pratique, et facilite énormément le développement d'algorithme d'approximation.

Un cycle $C = (v_1, v_2, \ldots, v_n, v_1)$ de G est *Hamiltonien* si C parcourt chaque sommet de V exactement une fois pour revenir à v_1. Le poids de C, dénoté $f(C)$, est égal à la somme des poids des arêtes de C, i.e.

$$f(C) = f(v_n v_1) + \sum_{i=1}^{n-1} f(v_i v_{i+1})$$

Dans le problème du commis voyageur, on cherche à visiter chaque ville

et à revenir à la maison de façon à minimiser la somme des distances parcourues.

COMMIS VOYAGEUR
Entrée : Une graphe complet $G = (V, E)$ et une métrique $f : E \to \mathbb{R}^{>0}$
Sortie : Un cycle Hamiltonien de poids minimum.

À priori, il n'est pas évident de trouver une borne *utile* sur OPT. En réfléchissant, on constate qu'un cycle Hamiltonien est un sous-graphe qui doit couvrir chaque sommet. On se rappelle alors qu'on connaît une notion sur les graphes qui vise à couvrir chaque sommet : les arbres couvrants minimums (ACM). Rappelons qu'un arbre couvrant est un sous-graphe de G qui est un arbre (donc connexe et sans cycle), qui contient tous les sommets de G. Cet arbre couvrant est un ACM si la somme des poids des arêtes est minimum parmi toutes les possibilités. L'algorithme de Prim ou encore l'algorithme de Kruskal peut trouver un ACM en temps polynomial.

Lemme 1. *Soit G une instance du COMMIS VOYAGEUR et soit $ACM(G)$ le poids d'un arbre couvrant minimum de G. Alors $OPT(G) \geq ACM(G)$.*

Démonstration. Soit C un cycle Hamiltonien de G de poids minimum. Le poids des arêtes de C est $OPT(G)$. Si on retire une arête de C, on obtient un chemin qui passe par chaque sommet une fois. C'est donc un arbre couvrant, et donc le poids de ses arêtes est au moins aussi grand que $ACM(G)$ (puisque justement, $ACM(G)$ est le minimum possible). Soit k le poids des arêtes de ce chemin. On a $OPT(G) \geq k \geq ACM(G)$. □

On peut donc tenter d'utiliser un ACM comme "point milieu" pour notre approximation, donc trouver un cycle qui n'est pas trop loin du poids d'un ACM. L'idée est de trouver un ACM, de prendre l'ordre de parcours pré-ordre de cet arbre. Cet ordre deviendra l'ordre de visite de nos villes.

En guise de rappel, si on a un arbre enraciné T, le parcours pré-ordre parcours d'abord la racine, puis visite ses enfants.

```
fonction preOrdre(T = (V, E), v, liste)
    //v est le noeud courant
    liste.append(v)
    pour w enfant de v faire
        preOrdre(T, w, liste)
    fin
```

Notre algorithme pour COMMIS VOYAGEUR est le suivant.

```
fonction commisApprox(G = (V, E), f)
    T = getACM(G)     // Obtenir un arbre couvrant minimum
    Enraciner T en un noeud r arbitraire
    C = ()                                          // Liste vide
    preOrdre(T, r, C)
    C.append(C.first)                  // Pour boucler le cycle
    return C
```

On peut montrer que c'est une 2-approximation. L'idée est que le cycle retourné fait un peu comme parcourir chaque arête de T deux fois, grâce à l'inégalité triangulaire.

Théorème 4. *L'algorithme commisApprox est une* 2*-approximation.*

Démonstration. Soit k le poids d'un ACM T de G, et soit $C = (v_1, \ldots, v_n, v_1)$ le cycle retourné par l'algorithme. Soit $f(C)$ le poids des arêtes de C. On veut montrer que $f(C) \leq 2k$, car on sait que $OPT \geq k$.

Soient v_i, v_j deux sommets consécutifs dans C (donc $j = i + 1$, ou bien $i = n, j = 1$). Soit $P_{i,j}$ le chemin de v_i à v_j dans T et soit $f(P_{i,j})$ le poids des arêtes de $P_{i,j}$. Grâce à l'inégalité triangulaire, on peut montrer que

$$f(v_i v_j) \leq f(P_{i,j})$$

Donc,

$$f(C) = \sum_{i=1}^{n-1} f(v_i v_{i+1}) + f(v_n v_1) \leq \sum_{i=1}^{n-1} f(P_{i,i+1}) + f(P_{n,1})$$

On remarque que dans un parcours pré-ordre, les chemins parcourus traversent chaque arête deux fois : une fois en descendant, et une fois en re-

montant. Donc, $\sum_{i=1}^{n-1} f(P_{i,i+1}) + f(P_{n,1}) \leq 2k$. On obtient

$$APP = f(C) \leq \sum_{i=1}^{n-1} f(P_{i,i+1}) + f(P_{n,1}) \leq 2k \leq 2OPT$$

ce qui conclut la preuve. □

On peut se demander comment l'inventeur de cet algorithme a bien pu y penser. Encore une fois, tout se trouve dans l'approche fondamentale. On commence par se demander ce qui pourrait bien donner une borne sur OPT. Une fois que le lien avec les ACM est fait, on se demande comment un ACM pourrait nous donner un cycle. Un fois que cette idée est établi, les calculs viennent de soi.

À noter que si on n'a pas l'inégalité triangulaire, il n'existe *pas* de c-approximation, et ce même si $c = n^k$ pour une constante k, sauf si P = NP. La conjecture P $\neq$ NP stipule que les problèmes NP-complets n'admettent pas d'algorithme en temps polynomial, ce que la plupart des chercheurs croient. Dit autrement, ne cherchez pas d'algorithme d'approximation pour COMMIS VOYAGEUR non-métrique.

3.2 Amélioration à une 3/2-approximation (section optionnelle)

Nous allons faire le sketch d'une 3/2-approximation. Celle-ci n'est pas extrêmement difficile, mais fait intervenir les cycles Eulériens et les matchings parfait, ce qui demande quelques connaissances supplémentaires.

Dans un graphe, un cycle Eulérien est un cycle C dans lequel un même sommet peut être visité plusieurs fois, et dans lequel chaque arête est parcourue exactement une fois. Un résultat classique de la théorie des graphes stipule le théorème suivant. La preuve est laissée en exercice et se trouve facilement en ligne. C'est en fait un des premiers résultats de théorie des graphes, découvert par le célèbre Euler.

Théorème 5. *Un graphe G a un cycle Eulérien si et seulement si chaque sommet a un dégré pair.*

Rappelons que le degré d'un sommet est son nombre de voisins. Le lien avec les cycles Hamiltoniens est le suivant, aussi laissé en exercice.

Lemme 2. *Soit $G = (V, E)$ et f une métrique sur les arêtes. Soit C un cycle Eulérien de G. Alors il existe un cycle Hamiltonien C' de poids égal ou inférieur à celui de C.*

L'idée de la preuve est de prendre l'ordre des sommets de C selon leur première apparition. Si on C contient la suite $v_i, v_{i+1}, \ldots, v_{i+k}$ et que dans C', elle devient v_i, v_{i+k} parce que $v_{i+1}, \ldots, v_{j-1}$ ont déjà été visités, on utilise l'inégalité triangulaire pour argumenter que $f(v_i v_{i+k}) \leq \sum_{j=i}^{i+k-1} f(v_j v_{j+1})$. Au final, on se retrouve avec le même poids que C.

Donc, si notre ACM T avait chaque sommet avec un nombre pair de voisin, il serait Eulérien et on pourrait le transformer en cycle Hamiltonien de même poids, donnant ainsi une 1approximation ! Bien sûr, un arbre a des feuilles, et celles-ci ont 1 voisin. L'idée est d'ajouter à T quelques arêtes aux sommets de degré impair de façon à ce qu'il devienne Eulérien.

Soient X les sommets de degré impair de T. Il est possible de démontrer que $|X|$ est pair (aussi en exercice : le nombre de sommets impairs d'un graphe est toujours pair). On va ajouter un ensemble d'arêtes M à T de façon à augmenter le degré de chaque sommet de X de 1. De cette façon, T avec M sera Eulérien. Pour ce faire, on va prendre un matching parfait dans X. C'est-à-dire, on partitionne X en exactement $|X|/2$ paires de façon à minimiser le poids des arêtes entre les paires choisies. De façon remarquable, ceci peut se faire en temps polynomial. Les paires choisies correspondent à un ensemble M de $|X|/2$ arêtes. L'intérêt de ce matching parfait est le suivant.

Lemme 3. *Soit M un matching parfait entre les éléments de $X \subseteq V(G)$ dans G, avec $|X|$ pair. Alors $OPT(G) \geq 2f(M)$.*

L'idée de la preuve est que puisque f est une métrique, $OPT(G[X]) \leq OPT(G)$, où $G[X]$ est le graphe induit par X. De plus, $OPT(G[X])$ est le poids d'un cycle qui consiste en l'union de deux matchings parfaits, qui sont chacun de poids au moins $f(M)$.

fonction *commisApprox2($G = (V, E), f$)*
  $T = getACM(G)$ `// Obtenir un arbre couvrant minimum`
  Soient X les sommets de degré impair de T
  Soit M un matching parfait minimum entre les éléments de X
  Soit $T' = (V(T), E(T) \cup M)$
  Soit C' un cycle Eulérien de T'
  Soit C un cycle Hamiltonien de G obtenu de C'
  return C

Théorème 6. *commisApprox2 est une* 3/2*-approximation.*

Démonstration. Le poids de C retourné par l'algorithme est $f(T) + f(M)$. On obtient $APP = f(T) + f(M) \leq OPT + OPT/2 = 3/2 \cdot OPT$. □

3.3 Problème des k-centres

Dans le problème des k-centres, on reçoit un ensemble de points $P = (p_1, \ldots, p_n)$ en d-dimensions (donc $p_i = (u_1, u_2, \ldots, u_d)$). Notre but est de trouver les k emplacements parmi les points de P qui sont les plus "centralisés". Pour déterminer ceci, on a une métrique $dist$ entre chaque paire de points, c'est-à-dire que $dist(p_i, p_j)$ est une valeur numérique telle que pour tout p_i, p_j, p_k, on a

$$dist(p_i, p_k) \leq dist(p_i, p_j) + dist(p_j, p_k)$$

Un exemple classique de métrique est la distance Euclidienne, où la distance entre deux points $p = (u_1, \ldots, u_d)$ et $q = (v_1, \ldots, v_d)$ est $dist(p, q) = \sqrt{\sum_{i=1}^{d}(u_i - v_i)^2}$. Notre algorithme fonctionne toutefois sur n'importe quelle métrique.

Notre but est de choisir k centres de façon à minimiser la plus grande distance à parcourir pour parvenir à un de ces centres.

Pour formaliser le problème, soit $P' \subseteq P$ et $p \in P$. On définit

$$dcentre(p, P') = \min_{p' \in P'}(dist(p, p'))$$

Si on interprète P' comme une liste de centres, ceci représente la distance vers le centre le plus près.

Notre problème est le suivant.

k-CENTRES
Entrée : points P, entier k, métrique $dist$
Sortie : $P' \subseteq P$ tel que $|P'| = k$ qui minimise $\max_{p \in P}(dcentre(p, P'))$

On observe d'abord que notre critère d'optimisation concerne une distance entre deux points (un point de p et un centre). Donc, il existe $p, q \in P$ tel que $OPT = dist(p, q)$. Il y a $\binom{|P|}{2} \in O(|P|^2)$ distances possibles, une

pour chaque paire de points. Soit

$$d_1, d_2, \ldots, d_m$$

la liste des distances possibles triées en ordre croissant, où $m = \binom{|P|}{2}$.

On se demande d'abord si $OPT = d_1$. Si oui, tant mieux, sinon, on se demande si $OPT = d_2$, puis si $OPT = d_3$, et ainsi de suite. Pour répondre à ces questions, on utilisera des graphes connexes. Pour $d_i \in \{d_1, d_2, \ldots, d_m\}$, on définit

$$G[d_i] = (P, \{p_1p_2 : dist(p_1, p_2) \leq d_i\})$$

comme le graphe dont les sommets sont les points P, et on ajoute une arête entre deux points si la distance ne dépasse pas d_i.

Il s'avère que déterminer si P admet k centres avec une distance au plus d_i est équivalent à trouver un *ensemble dominant* avec k sommets dans $G[d_i]$. Rappelons que dans un graphe $G = (V, E)$, un ensemble $X \subseteq V$ est *dominant* si $\forall v \in V \setminus X$, v a au moins un voisin dans X. Dit autrement, X domine tout ce qui se trouve à l'extérieur de X.

L'équivalence entre les centres et les ensembles dominants peut être démontrée comme suit.

Lemme 4. *Soit $P' \subseteq P$ un ensemble de k centres. Alors $\max_{p \in P}(dcentre(p, P')) \leq d_i$ si et seulement si P' est un un ensemble dominant dans $G[d_i]$.*

Démonstration. Rappelons que pour démontrer un "si et seulement si", il faut prouver deux directions, une pour chaque côté de l'implication.
($\Rightarrow$) : on prouve que si $\max_{p \in P}(dcentre(p, P')) \leq d_i$, alors P' est un ensemble dominant de $G[d_i]$.

Soit $p \in P \setminus P'$. On sait p est à distance au plus d_i de son centre $p' \in P'$. Par la définition de $G[d_i]$, ceci veut dire qu'il y a une arête entre p et p'. Donc, p est dominé par un élément de P'. Puisque ceci est vrai pour tout $p \in P \setminus P'$, P' est un ensemble dominant de $G[d_i]$.

($\Leftarrow$) : on prouve que si P' est un ensemble dominant de $G[d_i]$, alors tout point p est à distance au plus d_i d'un point de P'.

Si $p \in P'$, alors p est un centre et on n'a pas à le considérer. Sinon, on sait que dans $G[d_i]$, p a un voisin p' dans P' qui le domine. Par la définition de $G[d_i]$, $dist(p, p') \leq d_i$, ce qui implique que $dcentre(p, P') \leq d_i$. Donc tout point est à distance au plus d_i d'un centre. □

Une conséquence du lemme précédent est que OPT est égal au plus petit d_i tel que $G[d_i]$ contient un ensemble dominant de taille k. Une stratégie

serait donc d'itérer à travers les graphes $G[d_1], G[d_2], \ldots, G[d_m]$ et, pour chacun d'entre eux, vérifier s'il admet un ensemble dominant de taille k. Malheureusement, il n'y a pas d'algorithme en temps polynomial connu pour répondre à cette question (sinon, on pourrait avoir un algorithme optimal). Nous allons donc considérer chaque $G[d_i]$ du plus petit au plus grand, et tenter de trouver un ensemble dominant possiblement sous-optimal de façon gloutonne.

Considérons d'abord un algorithme simple pour trouver un ensemble dominant.

```
fonction getDomSet(G = (V, E))
    X = {}
    tant que |V| > 0 faire
        Choisir u ∈ V arbitrairement
        Ajouter u à X
        Retirer de G le sommet v et tous ses voisins
    fin
    return X
```

Ceci retourne un ensemble dominant parce que chaque fois qu'on ajoute un sommet, ses voisins deviennent dominés. Par contre, ce n'est pas nécessairement un ensemble dominant optimal. On peut tout de même utiliser cette sous-routine pour notre approximation.

```
fonction kCentres(P, k, dist)
    D = {}                                          // Les distances
    pour chaque paire de sommets p, q ∈ P faire
        D.append(dist(p, q))
    fin
    Trier D = {d_1, d_2, ..., d_m} en ordre croissant
    pour i = 1 à m faire
        Construire G[d_i]
        P' = getDomSet(G[d_i])
        si |P'| ≤ k alors
            return P'
        fin
    fin
```

Notez qu'on arrête au premier $G[d_i]$ qui admet un ensemble dominant P' tel que $|P'| \leq k$, et non tel que $|P'| = k$. Nous vous laissons le soin de réfléchir à cette technicalité.

Théorème 7. *kCentres est une* 2*-approximation.*

Démonstration. Soit $d_i = OPT$. Pour prouver le théorème, on espère que l'algorithme trouve un ensemble dominant dans un $G[d_j]$ tel que $d_j \leq 2 \cdot d_i$. Pour simplifier, on va supposer que $G[2d_i]$ est construit (en réalité, il faudrait prendre le plus haut d_j qui est inférieur à $2d_i$).

On sait que $G[d_i]$ admet un ensemble dominant $P' = \{p'_1, p'_2, \ldots, p'_k\}$ de taille k. Les centres de P' déterminent des groupes, appelés des clusters. C'est-à-dire, pour tout $p \in P$, on dit que p choisit le centre $p' \in P'$ si p' est le choix le plus proche de p. Pour chaque $j \in \{1, 2, \ldots, k\}$, on définit le cluster

$$C_j = \{p \in P : p \text{ choisit le centre } p'_j \}$$

Donc, tous les points dans un même C_j sont à distance au plus d_i de p'_j. Par l'inégalité triangulaire, si $p, q \in C_h$, alors

$$dist(p, q) \leq dist(p, p'_j) + dist(p'_j, q) \leq 2d_i$$

Une conséquence de ceci est que dans $G[2d_i]$, tous les points d'un même cluster sont voisins l'un de l'autre. Ceci veut dire que l'algorithme *getDomSet*, en recevant $G[2d_i]$ va d'abord choisir un $u \in V$ appartenant à un certain cluster C_j, puis retirer tous les points de C_j. À la 2ème itération, il va retirer tous les points d'un autre cluster, et ainsi de suite. Puisqu'il y a k clusters, *getDomSet* va retourner un ensemble avec au plus k sommets.

On déduit que l'algorithme retourne une ensemble dominant de $G[2d_i]$, ce qui implique que $APP \leq 2d_i = 2OPT$. □

Chapitre 4

Approche gloutonne et recherche locale

Dans le chapitre précédent, nos avons étudié l'approche qui trouvait d'abord une borne sur OPT, et qui développait ensuite un algorithme en fonction de la borne trouvée. Ceci fonctionne bien lorsque c'est possible, mais ce n'est malheureusement pas toujours évident. Une autre façon est de d'abord créer un algorithme, pour ensuite trouver la borne sur OPT par rapport à cet algorithme. Cette approche échoue souvent à trouver un bon facteur d'approximation, disons-le. Par contre, lorsqu'elle fonctionne, l'algorithme qui en résulte est souvent simple à implémenter, même si l'analyse est parfois difficile.

Lorsque nous sommes confrontés à un problème d'optimisation, une solution qui est souvent naturelle est de construire une solution en effectuant des choix basés sur des informations locales. Ceci prend communément l'une des deux formes suivantes :

- **l'approche par amélioration locale**. On démarre avec une solution quelconque, puis on tente de voir si on peut y apporter une petite modification locale pour l'améliorer. Si oui, on le fait et on répète. Sinon, on retourne cette solution.
- **l'approche gloutonne**. On démarre avec une solution vide et on trouve l'élément à y ajouter qui est "le plus prometteur". On l'ajoute, on met à jour notre instance et on continue.
  C'est glouton parce qu'à chaque étape, on prend le choix qui semble meilleur *maintenant* sans se soucier du futur.
  Bien sûr, ceci a tendance à tomber dans des min/max locaux, mais donne parfois des approximation avec de bonnes garanties théoriques.

Dans ce chapitre, nous verrons deux exemples classique, un pour chaque approche, avec MAX-CUT et SET-COVER.

4.1 Recherche locale et MAX-CUT

Soit $G = (V, E)$ un graphe. Une *bipartition* de G est un façon de séparer les sommets en deux parties non-vides. Plus spécifiquement, une bipartition est une paire d'ensembles (V_1, V_2) telle que $V_1 \cup V_2 = V$ avec $V_1 \neq \emptyset, V_2 \neq \emptyset$ et $V_1 \cap V_2 = \emptyset$. Une bipartition est parfois appelée une *coupe*.

Les arêtes d'une bipartition (V_1, V_2) sont dénotées $E(V_1, V_2)$ et correspondent aux arêtes qui traversent d'un côté et de l'autre de la bipartition. En d'autres termes,

$$E(V_1, V_2) = \{uv \in E : u \in V_1, v \in V_2\}$$

Dans le problème MAX-CUT, on cherche une bipartition qui maximise les arêtes qui traversent.

MAX-CUT
Entrée : un graphe $G = (V, E)$.
Sortie : une bipartition (V_1, V_2) qui maximise $|E(V_1, V_2)|$.

Dans l'esprit de la recherche locale, on va démarrer avec une bipartition arbitraire et changer un sommet d'emplacement s'il permet d'augmenter la coupe.

```
fonction maxCutLocal(G = (V, E))
    Soit u un sommet quelconque de V
    V_1 = {u}
    V_2 = V \ {u}
    fini = False
    tant que not fini faire
        fini = True
        pour v ∈ V_1 faire
            si |E(V_1 \ {v}, V_2 ∪ {v})| > |E(V_1, V_2)| alors
                V_1.remove(v)
                V_2.insert(v)
                fini = False
            fin
        fin
        pour v ∈ V_2 faire
            si |E(V_1 ∪ {v}, V_2 \ {v})| > |E(V_1, V_2)| alors
                V_1.insert(v)
                V_2.remove(v)
                fini = False
            fin
        fin
    fin
    return (V_1, V_2)
```

Une première interrogation concerne le temps requis par cet algorithme. Est-il possible qu'il boucle à l'infini ? On ne peut pas exclure qu'il cycle à travers une série de mouvements sans jamais terminer. Un premier élément à démontrer avec ce type d'algorithme est non-seulement qu'il termine, mais aussi qu'il termine en temps polynomial. Ceci est souvent complexe, mais dans notre exemple, une simple preuve suffit.

Lemme 5. *L'algorithme maxCutLocal termine après $O(n^2)$ itérations de la boucle principale.*

Démonstration. Soit (V_1, V_2) la bipartition en mémoire au début d'une itération de la boucle, et soit (V_1', V_2') la bipartition à la fin de la même itération. Si $fini = False$, on a fait au moins un mouvement qui augmente le nombre d'arêtes qui traversent. Ceci implique que $E(V_1', V_2') > E(V_1, V_2)$. Donc la valeur de la coupe augmente d'au moins 1 à chaque tour de boucle. Puisque

le nombre d'arêtes d'une bipartition est au maximum $\binom{n}{2} \in O(n^2)$, le nombre de fois maximum qu'on peut augmenter la valeur de la coupe est $O(n^2)$. □

Ceci implique que l'algorithme prend un temps polynomial puisqu'une itération de la boucle peut se faire en temps $O(n^2)$. Concentrons-nous maintenant sur la performance de *maxCutLocal* en termes d'approximation. Notez que cet algorithme a été développé de façon indépendante à toute borne sur OPT. Ceci complique l'analyse, car nous sommes maintenant contraints à borner OPT en fonction de l'algorithme (alors que dans le chapitre précédent, on était libres d'établir OPT sans contraintes). Il est donc difficile de décrire une technique d'analyse générale pour ce type d'algorithme — il semble que chaque algorithme glouton nécessite une analyse ad hoc.

Pour un sommet $v \in V$, on dénote par $N(v)$ l'ensemble des voisins de v dans G.

Lemme 6. *Soit (V_1, V_2) la solution retournée par maxCutLocal. Pour tout $v \in V_1$, v a au moins la moitié de ses voisins dans V_2, c'est-à-dire*

$$|N(v) \cap V_2| \geq |N(v)|/2$$

De plus, pour tout $v \in V_2, |N(v) \cap V_1| \geq |N(v)|/2$.

Démonstration. On prouve pour $v \in V_1$ seulement. Le cas $v \in V_2$ est identique.

Supposons pour fins de contradiction qu'il existe $v \in V_1$ tel que v a strictement moins de $|N(v)|/2$ voisins dans V_2. Dans ce cas, v a strictement plus de $|N(v)|/2$ voisins dans V_1. La contribution de v à $E(V_1, V_2)$ est plus petite que $|N(v)|/2$. Si on transfère v dans V_2, le nombre d'arêtes de la bipartition est modifié de

$$|N(v) \cap V_1| - |N(v) \cap V_2|$$

car on gagne les arêtes de v vers V_1, mais on perd les arêtes de v vers V_2. Cette différence est strictement plus grande que 0. Donc le transfert de v augmente la coupe. Ceci est une contradiction, car l'algorithme aurait alors effectué ce transfert. □

L'intuition est donc que chaque sommet a la moitié des voisins de l'autre côté. Dans le meilleur des mondes, chaque sommet aurait *tous* ses voisins de l'autre côté, ce qui forme notre borne sur OPT et donne notre 1/2-approximation.

Théorème 8. *maxCutLocal est une* 1/2*-approximation.*

Démonstration. On remarque que le nombre d'arêtes d'une bipartition est égal à la somme du nombre de voisins qui traverse pour chaque sommet, divisé par 2 car on compte chaque arête deux fois. C'est-à-dire, pour toute bipartition (V_1', V_2'), on a

$$|E(V_1', V_2')| = \frac{1}{2}\left(\sum_{v \in V_1'} |N(v) \cap V_2'| + \sum_{v \in V_2'} |N(v) \cap V_1'|\right)$$

Dans le meilleur cas, les intersections contiennent tous les voisins. Donc,

$$OPT \leq \frac{1}{2} \cdot \sum_{v \in V} |N(v)|$$

Dans le cas de l'algorithme *maxCutLocal*, on sait que chaque intersection contient au moins la moitié des voisinages.

$$APP \geq \frac{1}{2} \cdot \sum_{v \in V} \frac{|N(v)|}{2} = \frac{1}{2} \cdot \frac{1}{2} \sum_{v \in V} |N(v)| \geq \frac{1}{2} \cdot OPT$$

□

4.2 Algorithme glouton pour SET-COVER

Il s'avère que la stratégie gloutonne fonctionne bien pour SET-COVER, où chaque ensemble a un nombre arbitraire d'éléments. On cherche à couvrir un univers U avec un nombre minimum d'ensembles.

SET-COVER
Entrée : Univers $U = \{u_1, \ldots, u_n\}$, ensembles $S = \{S_1, \ldots, S_m\}$
Sortie : des ensembles $S^* \subseteq S$ qui couvrent U et qui minimisent $|S^*|$.

Pour fins de précision, on dit que $S^* \subseteq S$ couvre U si $\bigcup_{S' \in S^*} S' = U$. En exercice, nous demanderons également d'étudier la version dans laquelle chaque $S_i \in S$ a un coût différent, et on cherche à minimiser le coût de S^* au lieu de son nombre d'éléments.

Une idée très intuitive est de démarrer avec $S^* = \{\}$, puis de lui ajouter l'ensemble $S_i \in S$ qui couvre un maximum d'éléments. Ensuite, on ajoute le prochain ensemble qui couvre un maximum d'éléments restants, et ainsi de suite. Ceci donne l'algorithme suivant.

fonction *setCoverGlouton*($U = \{u_1, \ldots, u_n\}, S = \{S_1, \ldots, S_m\}$)
    $R = U$ //Éléments restant à couvrir
    tant que $R \neq \emptyset$ **faire**
        Soit $S_i \in S$ qui maximise $|S_i \cap R|$ `// Choix glouton`
        $S^*.append(S_i)$
        $R = R \setminus S_i$
    fin
    return S^*

Comme dans MAX-CUT, cet algorithme a été développé avec aucune borne sur OPT en tête. Il nous faudra donc une analyse ad hoc.

Pour l'algorithme setCoverGlouton, on peut s'attarder au coût qu'il a fallu payer couvrir chaque $u_k \in U$. On va s'en remettre à l'intuition suivante. S'il a fallu un ensemble juste pour couvrir seulement un u_k et aucun autre élément, alors u_k a coûté cher (il a fallu un ensemble à lui seul). Mais si u_k a été couvert par l'ajout d'un S_j, et que S_j a aussi couvert 99 autres nouveaux éléments de U, le coût de u_k est beaucoup moins élevé car il n'a pas été trop nuisible — le coût pour le couvrir est réparti avec 99 autres éléments.

Plus rigoureusement, on dit que chaque $S_i \in S$ a un coût de 1. Donc, le coût d'une solution S^* est $|S^*|$. Au moment où un ensemble S_i est ajouté à S^*, il distribue son coût de 1 à tous les nouveaux éléments couverts. C'est-à-dire, lorsqu'on ajoute S_i à S^*, chaque élément $u_k \in S_i \cap R$ est affecté le coût

$$cout(u_k) = \frac{1}{|S_i \cap R|}$$

où, encore une fois, R est l'ensemble des éléments restant à couvrir **au moment où S_i est ajouté à S^***. Le coût de chaque u_k est donc dépendant de combien d'autres éléments ont été couverts au même moment que lui.

L'intérêt de cette répartition des coût est qu'elle nous donne une autre façon de mesure $APP = |S^*|$.

Lemme 7. *Soit S^* l'ensemble retourné par setCoverGlouton. Alors*

$$\sum_{k=1}^{n} cout(u_k) = |S^*|$$

Démonstration. Chaque fois qu'on ajoute un S_i à S^*, on répartit un total de 1 à travers les éléments de $S_i \cap R$. Puisque chaque élément u_i reçoit un coût une seule fois, chaque $S_i \in S^*$ contribue exactement 1 à la somme des coûts, ce qui explique $\sum_{i=k}^{n} cout(u_k) = |S^*|$. □

Maintenant, les $cout(u_k)$ vont nous servir de "point milieu" entre OPT et APP. Il nous reste à établir un lien entre OPT et ces coûts. Dans l'algorithme, de plus en plus d'éléments sont couverts. On peut donc ordonner U de façon à supposer que u_1 est couvert en premier, u_2 est couvert en deuxième, et ainsi de suite (plusieurs éléments sont couverts dans la même itérations, dans quel cas on détermine un ordre arbitraire).

On suppose donc que $u_1, u_2, \ldots, u_n$ nous donne l'ordre dans lequel les éléments de U sont couverts. Prenons un u_k quelconque.

Lemme 8. *Pour tout $k \in \{1, 2, \ldots, n\}$, on a*

$$OPT/(n - k + 1) \geq cout(u_k)$$

Démonstration. Juste avant que u_k ne devienne couvert, il reste $|R| = n - k + 1$ éléments à couvrir. On sait que la solution optimale est capable de couvrir ces $n - k + 1$ éléments de R avec OPT ensembles. Donc, forcément, il existe un ensemble $S_i \in S^*$ tel que S_i contient $(n - k + 1)/OPT$ éléments de R (car sinon, impossible de couvrir R avec OPT ensembles).

Puisque l'algorithme maximise $S_i \cap R$, il choisira donc d'ajouter à S^* un ensemble S_i tel que $|S_i \cap R| \geq (n - k + 1)/OPT$. Ceci donne

$$OPT/(n - k + 1) \geq 1/|S_i \cap R|$$

Rappelons que ce S_i contient u_k car nous sommes au moins juste avant que u_k ne soit couvert. Ceci conclut la preuve, car $1/|S_i \cap R| = cout(u_k)$. □

On a maintenant tout ce qu'il faut pour comparer $APP = |S^*| = \sum_{k=1}^{n} cout(u_k)$ et OPT. On se retrouvera avec ce qu'on appelle la *série harmonique* dénotée H_n. Plus précisément, on définit

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Il s'avère que notre algorithme est une H_n-approximation.

Théorème 9. *setCoverGlouton est une H_n-approximation.*

Démonstration. On sait que

$$\begin{aligned} APP = \sum_{k=1}^{n} cout(u_k) &\leq \sum_{k=1}^{n} OPT/(n-k+1) \\ &= OPT \cdot \sum_{k=1}^{n} 1/(n-k+1) \\ &= OPT \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right) \\ &= H_n \cdot OPT \end{aligned}$$

ce qui conclut la preuve. □

Il est possible de démontrer que $H_n \in O(\log n)$. On dit donc souvent que *setCoverGlouton* est une $O(\log n)$-approximation. Il s'avère que si P $\neq$ NP, il n'existe pas de meilleur algorithm, c'est-à-dire qu'il n'existe pas de $f(n)$-approximation pour tout $f(n) \in o(\log n)$ (où la notation o réfère aux fonctions qui grandissent strictement moins vite que $\log n$).

Chapitre 5

Algorithmes probabilistes

Nous verrons dans ce chapitre quelques exemples d'algorithmes qui peuvent faire des choix aléatoires. Ces algorithmes ne retournent pas toujours la même solution et sa qualité peut varier selon le hasard. Toutefois, dans plusieurs cas, ces algorithmes peuvent donner un facteur d'approximation raisonnable *en espérance*. Nous verrons ensuite qu'il est parfois possible de *dérandomiser* certains algorithmes probabilistes et de les rendre complètement déterministes avec les mêmes garanties théoriques.

5.1 Notions de base

Pour nos fins, un algorithme probabiliste s'exécute en temps polynomial s'il retourne *toujours* une solution en temps polynomial, et ce peu importe ses choix aléatoires. Noez que ceci diffère de certaines analyses qui demandent un temps polynomial en espérance seulement. Soit A un algorithme d'approximation. Pour définir l'espérance de la valeur, on note que la valeur $APP(A)$ de la solution retournée par A est une variable aléatoire. Pour une valeur numérique K, on écrit

$$Pr[APP(A) = K]$$

pour dénoter la probabilité que A retourne une solution de valeur K. On suppose que les valeurs possibles de K forment un ensemble dénombrable.

L'espérance de la valeur APP de A est dénotée

$$\mathbb{E}[APP(A)] = \sum_K Pr[APP(A) = K] \cdot K$$

où la somme s'applique sur toutes les valeurs possibles de APP. La notion de c-approximation probabiliste se définit comme suit :

— si A résout un problème de minimisation, A est une c-approximation si $\mathbb{E}[APP(A)] \leq c \cdot OPT$;

— si A résout un problème de maximisation, A est une c-approximation si $\mathbb{E}[APP(A)] \geq c \cdot OPT$.

Notez que la plupart du temps, nous n'allons pas calculer l'espérance directement de la façon plus haut et aurons recours à quelques trucs pour simplifier nos calculs. L'un des plus importants est la *linéarité de l'espérance*, bien connue en probabilités et statistiques.

Théorème 10. *Soit $X_1, X_2, \ldots, X_n$ des variables aléatoires. Alors*

$$\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

5.2 1/2-approximation probabiliste pour MAX-CUT

Un des exemples les plus simple d'approximation probabiliste est pour MAX-CUT. On doit construire une bipartition (V_1, V_2) et on va simplement mettre chaque sommet dans un des deux côtés aléatoirement, chacun avec probabilité 1/2.

```
fonction maxCutProba(G = (V, E))
    V1 = ∅
    V2 = ∅
    pour v ∈ V faire
        res = lancerPiece()
        si res = pile alors
            V1.append(v)
        sinon
            V2.append(v)
        fin
    fin
    return (V1, V2)
```

Malgré la simplicité de cet algorithme, la moitié des arêtes se retrouveront dans la coupe en moyenne, car chaque arête a une chance sur deux d'avoir ses extrémités séparées. Nous formalisons cette idée plus bas.

Théorème 11. *L'algorithme maxCutProba est une* 1/2*-approximation probabiliste.*

Démonstration. Dans ce problème, APP est le nombre d'arêtes uv séparées dans (V_1, V_2). Pour $uv \in E$, on définit la variable aléatoire

$$\mathbb{1}_{uv} = \begin{cases} 1 & \text{si } u \in V_1 \text{ et } v \in V_2, \text{ ou si } u \in V_2 \text{ et } v \in V_1 \\ 0 & \text{sinon} \end{cases}$$

Donc a donc $APP = \sum_{uv \in E} \mathbb{1}_{uv}$. L'espérance de APP est donc

$$\mathbb{E}[APP] = \mathbb{E}[\sum_{uv \in E} \mathbb{1}_{uv}] = \sum_{uv \in E} \mathbb{E}[\mathbb{1}_{uv}]$$

Ici, $\mathbb{1}_{uv}$ est la probabilité que u et v se retrouvent séparés dans une solution. On peut argumenter que $\mathbb{1}_{uv} = 1/2$, car il y a 4 combinaisons équiprobables d'emplacements possibles pour u et v, et 2 de ces cas séparent u et v. Donc,

$$\mathbb{E}[APP] = \sum_{uv \in E} \mathbb{E}[\mathbb{1}_{uv}] = \sum_{uv \in E} 1/2 = |E|/2$$

Puisque $OPT \leq |E|$, on obtient $\mathbb{E}[APP] = |E|/2 \geq OPT/2$. □

Notez que cet algorithme fonctionne aussi pour la version avec poids de MAX-CUT, car chaque poids a une chance sur deux de se retrouver dans la coupe. L'avantage de cet algorithme est qu'il s'implémente très facilement et est très rapide. En pratique, une technique commune avec ce type d'algorithme est de l'exécuter plusieurs fois et garder la meilleure solution.

5.3 7/8-approximation pour MAX-3-SAT

MAX-3-SAT est la variante de MAX-SAT dans laquelle chaque clause a exactement 3 variables.

MAX-3-SAT
Entrée : un ensemble de clauses $C_1, \ldots, C_m$ avec 3 variables chacune sur variables $x_1, \ldots, x_n$;
Sortie : une assignation des x_i qui maximise le nombre de clauses satisfaites.

Il existe un algorithme bien simple pour ce problème : choisir une assignation au hasard.

fonction *max3satProba*($C_1, \ldots, C_m$ *sur variables* $x_1, \ldots, x_n$)
  A = assignation vide
  pour $i = 1..n$ **faire**
    $res = lancerPiece()$
    si $res = pile$ **alors**
      Assigner $x_i = True$ dans A
    sinon
      Assigner $x_i = False$ dans A
  fin
  return A

Considérez une clause C_i, par exemple $C_i = (x_i \vee \overline{x}_j \vee x_k)$. La seule façon de ne pas satisfaire C_i est que toutes ses variables évaluent à $false$. Il y a 8 combinaisons d'assignations pour les 3 variables de C_i, et 7 de ces combinaisons satisfont C_i. Puisque chaque combinaison est choisie avec la même probabilité, on a donc une probabilité de $7/8$ de satisfaire C_i. Puisque ceci est vrai pour chaque clause, par la linéarité de l'espérance, on s'attend à ce qu'une proportion $7/8$ des clauses soit satisfaite.

Il en découle le théorème suivant. La preuve ne fait que formaliser le paragraphe ci-haut.

Théorème 12. *L'algorithme max3satProba est une* $7/8$*-approximation probabiliste.*

Démonstration. Soit A l'assignation retournée par l'algorithme. Pour une clause C_i, définissons

$$\mathbb{1}_{C_i} = \begin{cases} 1 & \text{si } A \text{ satisfait } C_i \\ 0 & \text{sinon} \end{cases}$$

On obtient

$$\mathbb{E}[APP] = \mathbb{E}[\sum_{i=1}^{m} \mathbb{1}_{C_i}] = \sum_{i=1}^{m} \mathbb{E}[\mathbb{1}_{C_i}] = \sum_{i=1}^{m} 7/8 = 7m/8$$

On sait que $OPT \leq m$, et donc $\mathbb{APPP}[E] = 7m/8 \geq 7/8 \cdot OPT$.

□

5.4 Dérandomisation

Les puristes diront qu'une c-approximation probabiliste n'est pas convenable, car il se pourrait que par malchance, APP soit très loin de OPT. Quel est le but de fournir des garanties théoriques si elles ne sont garanties qu'avec une certain probabilité ? Une première réponse est qu'il est possible de montrer que si on exécute le même algorithme plusieurs fois et qu'on conserve la meilleure solution, la probabilité de s'éloigner de l'espérance diminue exponentiellement (si ceci vous intéresse, cherchez "Chernoff bound").

Une autre réponse est que parfois, on peut éliminer l'effet du hasard en transformant l'algorithme probabiliste en algorithme déterministe tout en maintenant les mêmes garanties d'approximation. C'est ce qu'on appelle la *dérandomisation*.

L'idée est d'abord calculer l'espérance d'une instance donnée, un peu comme dans l'analyse de MAX-CUT ou MAX-3-SAT. On tente d'ajouter un élément à notre solution et on calcule l'effet de fixer ce choix sur l'espérance. Si elle n'a pas empiré, on fixe le choix. Si elle a empiré, on ne fait pas ce choix et on passe au suivant. Si chacun des choix maintient l'espérance que l'on avait à l'origine, on aura fixé un ensemble de choix déterministes qui donne une solution qui atteint l'espérance originale.

Dérandomisation de MAX-3-SAT

En guise d'exemple, voyons la version "dérandomisée" de MAX-3-SAT. Soit A_j une assignation des variables $x_1, x_2, \ldots, x_j$. C'est une assignation partielle, car A_j assigne la valeur de certaines variables, mais pas toutes (sauf si $j = n$). L'idée est que certaines clauses sont déjà satisfaites par A_j, d'autres ne le seront jamais, et d'autres peut-être.

On suppose qu'on fixe A_j et qu'on assigne $x_{j+1}, \ldots, x_n$ aléatoirement. La probabilité de satisfaire une clause C_i *sachant* A_i peut s'exprimer comme suit :

$$Pr[C_i \text{ est satisfaite } |A_j] = \begin{cases} 1 & \text{si } A_j \text{ satisfait déjà } C_i \\ 1 - \frac{1}{2^k} & \text{sinon, où } k \text{ est le nombre de variables} \\ & \text{de } C_i \text{ non-assignées par } A_j \end{cases}$$

Par exemple, supposons que selon A_2, on a $x_1 = true$ et $x_2 = false$. Soit $C_i = (\overline{x}_1 \vee x_3 \vee \overline{x}_4)$. Alors sachant que A_2 affecte $x_1 = true$, on sait qu'on ne va pas satisfaire C_1 avec x_1. Ceci laisse 2 variables capable de satisfaire C_i avec probabilité $1 - 1/4 = 3/4$. Ceci correspond à 3 chances sur 4 de

satisfaire C_i avec x_3 ou bien x_4.

On peut argumenter que l'espérance de APP sachant A_j est

$$\mathbb{E}[APP|A_j] = \sum_{i=1}^{m} Pr[C_i \text{ est satisfaite } |A_j]$$

L'idée de la dérandomisation est de démarrer avec une assignation A_0 vide. Tel que vu plus haut, $\mathbb{E}[APP] = \mathbb{E}[APP|A_0] = 7m/8$. On va tenter d'assigner $x_1 = true$ dans une assignation temporaire A_1^+ et calculer $\mathbb{E}[APP|A_1^+]$. On regarder aussi l'assignation A_1^- qui assigne $x_1 = false$ et on calcule $\mathbb{E}[APP|A_1^-]$. On pose A_j comme le choix qui maximise l'espérance.

fonction *max3satDerand($C_1, \ldots, C_m$ sur variables $x_1, \ldots, x_n$)*
  A_0 = assignation vide
  pour $j = 1..n$ **faire**
    A_j^+ = copie de A_{j-1} avec $x_j = true$
    $E_j^+ = \mathbb{E}[APP|A_j^+] = \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_j^+]$
    A_j^- = copie de A_{j-1} avec $x_j = false$
    $E_j^- = \mathbb{E}[APP|A_j^-] = \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_j^-]$
    si $E_j^+ \geq E_j^-$ **alors**
      $A_j = A_j^+$;
    sinon
      $A_j = A_j^-$
    fin
  fin
  return A_n

Notez que même s'il fait des calculs de probabilités, cet algorithme est entièrement déterministe. L'idée clé est qu'après chaque tour de boucle, on a fixé x_j dans A_j tel que $\mathbb{E}[APP|A_j] \geq 7m/8$.

Lemme 9. *Après le j-ème tour de boucle de l'algorithme, $j \in \{0, 1, \ldots, n\}$, on a $\mathbb{E}[APP|A_j] \geq 7m/8$.*

Démonstration. Soit $j \in \{0, 1, \ldots, n\}$. On montre par induction sur j que $\mathbb{E}[APP|A_j] \geq 7m/8$.

Comme cas de base, on prend $j = 0$. On a déjà argumenté que $\mathbb{E}[APP|A_0] \geq 7m/8$ parce que a priori, chaque clause a $7/8$ chances d'être satisfaite.

Considérons $j > 0$, en supposant par induction que $\mathbb{E}[APP|A_{j-1}] \geq 7m/8$. L'espérance de A_{j-1} avait été calculée en utilisant le fait qu'on avait une chance sur 2 d'assigner $x_j = true$, et une chance sur 2 d'assigner $x_j = false$. On se retrouve donc avec

$$\begin{aligned}
\mathbb{E}[APP|A_{j-1}] &= \sum_{i=1}^{m} Pr[C_i \text{ est satisfaite } |A_{j-1}] \\
&= \sum_{i=1}^{m} \left(\frac{1}{2} Pr[C_i \text{ est satisfaite } |A_j^+] + \frac{1}{2} Pr[C_i \text{ est satisfaite } |A_j^-] \right) \\
&= \frac{1}{2}\mathbb{E}[APP|A_j^+] + \frac{1}{2}\mathbb{E}[APP|A_j^-] \\
&= \frac{1}{2}E_j^+ + \frac{1}{2}E_j^- \\
&\geq 7m/8
\end{aligned}$$

La dernière inégalité est simplement parce que $\mathbb{E}[APP|A_{j-1}] \geq 7m/8$ par induction. Cette inégalité implique aussi que un de E_j^+ ou E_j^- est au moins $7m/8$. Puisque A_j est choisi en conséquence de $max(E_j^+, E_j^-)$, $\mathbb{E}[APP|A_j] \geq 7m/8$. □

Théorème 13. *L'algorithme max3satDerand est une* 7/8*-approximation (déterministe).*

Démonstration. Soit A_n l'assignation après n tours de boucle. Donc A_n est retourné par l'algorithme et assigne chaque variable. Par le lemme précédent, $\mathbb{E}[APP|A_n] \geq 7m/8$. Notez qu'avec A_n, chaque clause est satisfaite ou non, et il n'y a plus de choix probabiliste possible. Donc $\mathbb{E}[APP|A_n] \geq 7m/8$ implique que A_n satisfait au moins $7m/8$ clauses. Puisque $OPT \leq m$, on déduit qu'on a une 7/8-approximation. □

Chapitre 6

Schémas d'approximation en temps polynomial (et le sac-à-dos)

Quelle est le ratio d'approximation ultime ? Si on résout un problème de minimisation, le mieux qu'on peut espérer pour un problème NP-complet est

$$APP \leq (1 + \varepsilon) \cdot OPT$$

avec ε très petit. De la même manière, l'idéal pour un problème de maximisation est

$$APP \geq (1 - \varepsilon) \cdot OPT$$

Il s'avère que pour certains problèmes, il est possible d'atteindre une précision arbitrairement proche de 1 (et donc un ε arbitrairement proche de 0). Pour ce faire, on donne le ε voulu à l'algorithme, et il se débrouille pour nous garantir notre facteur d'approximation. C'est ce qu'on appelle un *schéma d'approximation en temps polynomial*.

6.1 Schémas d'approximation (PTAS)

La terminologie standard pour désigner nos types d'algorithmes est en fait *Polynomial Time Approximation Scheme* (PTAS). Nous alons donc utiliser le terme PTAS afin de rester consistant avec la littérature.

Une PTAS est un algorithme d'approximation A qui reçoit en entrée une instance *et* un paramètre ε, et tel que :

— A s'exécute en temps polynomial par rapport à la taille n de l'instance, où ε est traité comme une constante ;
— si A résout un problème de minimisation, $APP \leq (1+\varepsilon) \cdot OPT$
— si A résout un problème de maximisation, $APP \geq (1-\varepsilon) \cdot OPT$.

Le premier point précise que la complexité est indépendante de ε, il est fréquent que $1/\varepsilon$ contribue à la complexité. Certaines PTAS s'exécutent par exemple en temps $O(\frac{1}{\varepsilon}n^2)$, ou encore en temps $O(2^{2^{1/\varepsilon}}n)$. Ceci est acceptable si $\varepsilon = 1/2$, mais si $\varepsilon = 1/n$, ceci a une impact sur la complexité. D'autres définition qui considère ε dans la complexité existent. Par exemple, une FPTAS exisge que la complexité soit polynomiale en n *et* $1/\varepsilon$.

Il n'est bien sûr pas toujours possible de trouver une PTAS, ou encore, certaines PTAS ont une complexité absurde et n'ont aucune application pratique. Nous allons donner un exemple de PTAS qui s'avère à être applicable en pratique.

6.2 Problème SAC-A-DOS

Dans le problème SAC-A-DOS, on reçoit la capacité W d'un sac et des objets, qui ont chacun un poids w_i et une valeur v_i. L'objectif est de remplir le sac avec une valeur totale maximum sans dépasser la capacité. Ce problème s'appelle le KNAPSACK en anglais.

Soit $R = \{(w_1, v_1), \ldots, (w_n, v_n)\}$ un ensemble d'objets, où l'objet i a un poids w_i et une valeur v_i. Pour $R' \subseteq R$, on va dénoter $w(R') = \sum_{(w_i,v_i)\in R'} w_i$ et $v(R') = \sum_{(w_i,v_i)\in R'} v_i$.

SAC-A-DOS
Entrée : entier W, objets $R = \{(w_1, v_1), \ldots, (w_n, v_n)\}$
Sortie : un sous-ensemble $R' \subseteq R$ tel que $w(R') \leq W$ et qui maximise $v(R')$.

Il existe un algorithme de programmation dynamique classique pour le SAC-A-DOS. Dénotons d'abord

$$v_{max} = \max\{v_1, \ldots, v_n\}$$

On note que $OPT \leq n \cdot v_{max}$. On va itérer à travers chaque profit possible de 1 à nv_{max} et se demander s'il possible d'atteindre ce profit.

Pour $i \in \{1, 2, \ldots, n\}$ et pour $p \in \{1, 2, \ldots, n \cdot v_{max}\}$, on définit une

fonction $B(i,p)$ comme le poids minimum possible permettant d'atteindre un profit p en utilisant seulement les objets $(w_1,v_1),\ldots,(w_i,v_i)$, c'est-à-dire les i premiers objets. Dit autrement,

$$B(i,p) = \min\{w : \exists R' \subseteq \{(w_1,v_1),\ldots,(w_i,v_i)\} \text{ tel que } w(R') = w \text{ et } v(R') = p\}$$

On cherche la valeur maximum de p telle que $B(n,p) \leq W$, car dans ce cas, $B(n,p)$ correspond à une façon de choisir des objets parmi tous ceux de l'entrée qui a un profit p et un poids dans la capacité.

On a que $B(i,0) = 0$ pour tout i, car on peut toujours avoir un profit 0 avec un poids 0. Pour les autres valeurs, il est possible d'argumenter que

$$B(i,p) = min \begin{cases} B(i-1,p) \\ B(i-1,p-v_i) + w_i \end{cases}$$

Ceci est parce qu'il y a deux façons d'atteindre un profit p. On n'utilise pas (w_i,v_i) et on atteint un profit p avec les $i-1$ premiers éléments. Ceci peut se faire avec un poids $B(i-1,p)$. Sinon, on utilise (w_i,v_i). Dans ce cas, on avait un profit $p-v_i$ avec les $i-1$ premiers éléments et on y a ajouté v_i. On a aussi ajouté un poids de w_i au sac. Le poids dans ce cas est $B(i-1,p-v_i)+w_i$. Il suffit de prendre le mminimum des deux.

```
fonction sacados(W,(w_1,v_1),(w_2,v_2),...,(w_n,v_n))
    B = table de dimension n × nv_max
    Initialiser B(i,p) = ∞ pour tout i,p
    B(i,0) = 0 pour tout i
    pmax = 0
    pour i = 1..n faire
        pour p = 1..nv_max faire
            B(i,p) = min(B(i-1,p), B(i-1,p-v_i) + w_i)
            si B(i,p) ≤ W  et p > pmax alors
                pmax = p
            fin
        fin
    fin
    return pmax
```

Cet algorithme retourne seulement le profit maximum au lieu d'un ensemble d'objets concret. Si on le veut, on peut reconstruire une solution

concrète atteignant un profit $pmax$ à partir de la table B, qui en revient à faire du *backtracking* de programmation dynamique.

fonction
sacadosBacktrack$(W, (w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n), B, pmax)$
  $X = \emptyset$
  $i = n$
  $p = pmax$
  tant que $i > 0$ **faire**
    si $B(i, p) = B(i - 1, p - v_i) + w_i$ **alors**
      $X.append((w_i, v_i))$
      $p = p - v_i$
    $i - -$
  fin
  return X

La complexité de l'algorithme est $O(n^2 v_{max})$, que ce soit pour calculer $pmax$ ou bien une solution concrète X. Ceci n'est pas nécessairement polynomial. Par exemple, il est possible que $v_{max} = 2^n$, dans quel cas le temps est en fait exponentiel (en réalité, la complexité d'un algorithme se mesure selon le nombre de bits de l'entrée. Le fait est que même si $v_{max} = 2^n$, il ajoute $\log(2^n) = n$ bits à l'entrée pour sa représentation et un temps de 2^n est toujours exponentiel). En fait, le problème SAC-A-DOS est NP-complet, et il n'existe probablement pas d'algorithme en temps purement polynomial.

L'algorithme ci-haut est appelé *pseudo-polynomial*, car il est polynomial en les valeurs numériques de l'entrée, mais pas polynomial en le nombre de bits requis pour représenter ces valeurs.

6.3 Une PTAS pour SAC-A-DOS

Puisque v_{max} est un problème, pourquoi ne pas réduire les valeurs des objets pour que v_{max} soit polynomial ? Si on divise toutes les valeurs d'un même facteur K, le problème reste le même. Par contre, puisque tous les v_i doivent être des entiers, on va prendre la valeur plancher de la division.

On ne sait pas trop par quel facteur K diviser nos v_i, alors on va le garder en variable et espérer le déterminer pendant notre analyse. Pour

nous simplifier la vie, on va parfois écrire

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor$$

pour dénoter le poids modifié. On va donc créer une instance avec la même capacité W, mais les objets

$$R' = \{(w_1, v'_1), \ldots, (w_n, v'_n)\}$$

Si par exemple $K = v_{max}/n$, les valeurs v'_i seront toutes polynomiales. Mais rappelons que K est toujours indéterminé (on découvrira plus tard que $K = v_{max}/n \cdot \varepsilon$). Notre stratégie algorithique est simple : modifier les valeurs et utiliser la programmation dynamique.

fonction *sacadosPoly*$(W, (w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n), K)$
  Pour chaque $i \in \{1, \ldots, n\}$, soit $v'_i = \lfloor v_i/K \rfloor$
  $X = sacados(W, (w_1, v'_1), \ldots, (w_n, v'_n))$
  return X

Cet algorithme ne donne pas toujours la solution optimale puisqu'une précision se perd dans les valeurs plancher. Dit autrement, la solution X retournée par l'algorithme est optimale **par rapport aux valeurs** v'_i, mais pas nécessairement par rapport aux valeurs v_i. Il nous faut maintenant quantifier cette perte de précision par rapport à K. Pour quantifier la perte des valeurs plancher, on va utiliser ce qui suit, laissé en exercice.

Lemme 10. *Pour tout v_i, puisque $v'_i = \lfloor v_i/K \rfloor$, on a*

$$\frac{v_i}{K} - 1 \leq v'_i \leq \frac{v_i}{K}$$

Si on isole v_i, on obtient $Kv'_i \leq v_i \leq K(v'_i + 1)$.

Soit X la solution retournée par *sacadosPoly*. Donc, $APP = \sum_{(w_i,v_i)\in X} v_i$ est la somme des valeurs **par rapport aux valeurs originales** v_i.

Comme d'habitude, notre but est de développer une chaîne d'inégalités du style $APP \geq \ldots \geq \ldots \geq c \cdot OPT$. Il suffit de remplir les trois petits points. Commençons avec ce qu'on sait.

$$APP = \sum_{(w_i,v_i)\in X} v_i \geq K \sum_{(w_i,v_i)\in X} v'_i$$

Nous sommes un peu bloqués. Essayons de connecter cette dernière expression avec OPT. Soit X^* une solution optimale par rapport aux valeurs v_i. Si on divise les valeurs des éléments de X^* par K, on peut interpréter X^* comme une solution pour les valeurs v'_i. On sait que X est optimal par rapport à ces valeurs. Donc $\sum_{(v_i,w_i)\in X^*} v'_i \leq \sum_{(v_i,w_i)\in X} v'_i$. Voyons voir ce qu'on peut dire sur OPT.

$$\begin{aligned} OPT = \sum_{(w_i,v_i)\in X^*} v_i &\leq \sum_{(w_i,v_i)\in X^*} K(v'_i + 1) \\ &= K|X^*| + K \sum_{(w_i,v_i)\in X^*} v'_i \\ &\leq K|X^*| + K \sum_{(w_i,v_i)\in X} v'_i \end{aligned}$$

Donc, $K \sum_{(w_i,v_i)\in X} v'_i \geq OPT - K|X^*|$. En reconnectant avec ce qu'on sait sur APP, on a

$$APP \geq K \sum_{(w_i,v_i)\in X} v'_i \geq OPT - K|X^*|$$

Voici enfin le moment de choisir K. Il faut que K contienne v_{max} quelque part pour que les valeurs deviennent polynomiales. De plus, le K devrait nous aider à nous débarasser du $|X^*|$. Avec un peu d'essai et erreur, on peut voir qu'on peut poser $K = \varepsilon \cdot v_{max}/n$ pour tout ε. Puisque $n \leq v_{max}$, et puisque $OPT \geq v_{max}$, on obtient

$$APP \geq OPT - \frac{\varepsilon \cdot v_{max}}{n}|X^*| \geq OPT - \varepsilon \cdot v_{max} \geq OPT - \varepsilon \cdot OPT = (1-\varepsilon) \cdot OPT$$

Si on passe la valeur $K = \varepsilon \cdot v_{max}/n$ à *sacadosPoly*, la complexité de la programmation dynamique devient

$$O(n \cdot nv_{max}/K) = O(n^2 \cdot v_{max} \cdot n/(\varepsilon \cdot v_{max})) = O(\frac{1}{\varepsilon} n^3)$$

Plus on veut une bonne précision, plus ε est près de 0, et plus la valeur de $1/\varepsilon$ augmente la complexité. L'avantage est qu'on peut ajuster ε. On conclue avec le théorème suivant.

Théorème 14. *Pour tout $\varepsilon > 0$, il existe une $(1-\epsilon)$-approximation au problème SAC-A-DOS qui s'exécute en temps $O(\frac{1}{\varepsilon} n^3)$.*

Chapitre 7

Approximation et programmation linéaire

Un programme linéaire est une façon ultra-générique de formuler un problème d'optimisation. On suppose qu'on a un ensemble de variables numérique $x_1, \ldots, x_n$ (possiblement d'autres). On doit spécifier la fonction objective à minimiser ou maximiser, et cette fonction doit être linéaire par rapport aux variables. On spécifie aussi un ensemble de *contraintes linéaires*. Une contrainte linéaire est une inégalité dans laquelle chaque terme ne contient qu'une seule variable, possiblement multiplié par une constante. Voici un exemple :

$$
\begin{aligned}
&\text{Minimiser} \\
&\qquad 10x_1 + 12x_2 + 4x_3 \\
&\text{Sujet à} \\
&\qquad x_1 + 2x_2 \geq 5 \\
&\qquad 3x_2 - x_3 \geq 3 \\
&\qquad x_1 + x_2 \geq 2 \\
&\qquad 0 \leq x_i \leq 1 \qquad\qquad \text{pour chaque } x_i \in \{x_1, x_2, x_3\}
\end{aligned}
$$

Les deux premières lignes indiquent l'objectif, et les autres lignes les contraintes (la dernière ligne représente en fait 6 contraintes différentes).

De façon générale, une programme linéaire est souvent exprimé en forme

matricielle.

$$\begin{aligned} &\text{Minimiser} \\ &\qquad \boldsymbol{c}^T \boldsymbol{x} \\ &\text{Sujet à} \\ &\qquad \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ &\qquad \boldsymbol{x} \geq 0 \end{aligned}$$

où $\boldsymbol{x} = (x_1, \ldots, x_n)$ est le vecteur de nos variables, $\boldsymbol{c}$ représente les coefficients des variables à optimiser, $\boldsymbol{A}$ est une matrice avec n colonnes, et b est un vecteur de constantes. Ici, seul le vecteur $\boldsymbol{x}$ est inconnu, alors que les valeurs de $\boldsymbol{c}, \boldsymbol{A}$ et $\boldsymbol{b}$ ne contiennent que des valeurs constantes connues. On va souvent dénoter par $\boldsymbol{x}^*$ un vecteur de solution, c'est-à-dire un vecteur contenant des valeurs concrètes pour les x_i. On dénotera les valeurs de $\boldsymbol{x}^*$ par $x_1^*, \ldots, x_n^*$.

En supposant que le domaine de x_i est les réels, un résultat remarquable en informatique est le suivant.

Théorème 15. *Il existe un algorithme en temps polynomial qui, étant donné un programme linéaire, trouve une solution optimale aux valeurs des x_i telles que chaque contrainte est satisfaite.*

Un des algorithmes les plus connus est la méthode de l'ellipsoïde. Nous allons nous contenter d'utiliser ce théorème en boîte noire. La résolution de programmes linéaires est l'objet d'un cours de semestre en entier. Nous sommes plutôt intéressés à utiliser ce théorème pour des fins d'approximation.

En anglais, un programme linéaire est appelé *Linear Program*, ou LP pour être bref. Nous allons aussi utiliser LP pour dénoter un programme linéaire.

7.1 LP pour l'approximation

Comme nous l'avons vu, il n'est pas toujours évident de trouver une borne pertinente sur OPT. L'utilité des programmes linéaire réside dans leur capacité à fournir de telles bornes. L'idée est d'exprimer un LP de façon à ce que l'optimal de la fonction objective donne une borne sur OPT. Notez qu'il n'est pas nécessaire que le programme linéaire ne représente pas exactement notre problème — il faut seulement qu'il donne une borne sur

OPT d'une façon ou d'une autre. La façon la plus commune est d'utiliser une *relaxation* à notre problème.

Définition 1. *Soit P un problème sur des variables $\boldsymbol{x}$ et soit L un programme linéaire sur les variables $\boldsymbol{x}$. On suppose que P et L sont tous les deux du même type (minimisation ou maximisation). On dit que L est une* relaxation *à P si, pour toute solution faisable $\boldsymbol{x'}$ de P, $\boldsymbol{x'}$ satisfait toutes les contraintes de L.*

Notez que $\boldsymbol{x'}$ n'est pas nécessairement optimal pour L, seulement pour P. Notez aussi que la faisabilité ne va que dans un sens : une solution faisable pour L n'est pas nécessairement faisable poru P. L'intérêt d'exprimer une relaxation pour P est que l'optimal de L donne automatiquement une borne sur l'optimal de P.

Théorème 16. *Soit L un programme linéaire qui est une relaxation pour un problème de P. Soit OPT_L l'optimal pour L et OPT_P l'optimal pour P. Alors :*

- *si P est un problème de minimisation, $OPT_P \geq OPT_L$;*
- *si P est un problème de maximisation, $OPT_P \leq OPT_L$.*

Démonstration. Supposons que P est un problème de minimisation et soit $\boldsymbol{x'}$ une solution optimale pour P. La valeur de $\boldsymbol{x'}$ pour P est OPT_P. Aussi, $\boldsymbol{x'}$ satisfait toutes les contraintes de L (puisque c'est une relaxation). Donc, $\boldsymbol{x'}$ est une des solutions possibles pour L, mais pas nécessairement l'optimal pour L (il se pourrait que L trouve une meilleure solution $\boldsymbol{x}^*$). On déduit que $OPT_L \leq OPT_P$. La preuve dans le cas maximisation est identique. □

L'approche haut-niveau de l'approximation par LP pour un problème de minimisation P va comme suit.

1. exprimer une relaxation L pour P ;
2. résoudre L en temps polynomial, ce qui résulte en un vecteur $\boldsymbol{x}^*$ de valeur OPT_L par rapport à L ;
3. transformer $\boldsymbol{x}^*$ en une solution faisable $\boldsymbol{x'}$ pour P de valeur APP par rapport à P ;
4. argumenter que $APP \leq c \cdot OPT_L$ ($\leq OPT$).

Les points 3 et 4 sont souvent ceux qui demandent le plus de réflexion. exemples.

7.2 Relaxation par programme linéaire avec entiers

La plupart de nos problèmes algorithmiques peuvent s'exprimer avec un *programme linéaire avec entiers*, ce qui est différent d'un LP. La terminologie anglaise appelle ceci un *Integer Linear Program*, ou ILP pour être bref. Nous allons aussi utiliser cet acronyme. Un ILP est comme un LP, mais avec la possibilité d'ajouter des contraintes du type

$$x_i \in \{0, 1\}$$

Plus généralement, on peut exiger que $x_i \in \{i_1, \ldots, i_k\}$, où les i_j sont des entiers. Ceci est une différence fondamentale. Dans un LP normal, on demande habituellement $x_i \geq 0$, ou encore $0 \leq x_i \leq 1$, ce qui permet aux x_i d'être des réels. La restriction à des entiers rend le problème NP-complet.

Toutefois, les ILP suggèrent des relaxation tout à fait naturelles. Il suffit de transformer les contraintes $x_i \in \{0, 1\}$ en des contraintes $0 \leq x_i \leq 1$. Toute solution faisable au ILP est faisable pour le LP, et c'est donc une relaxation. Le problème est que dans la relaxation, une solution $\boldsymbol{x}^*$ peut être *fractionnelle*.

Avant d'utiliser les LP et ILP pour l'approximation, voici quelques trucs communs lorsqu'on développe un ILP, en supposant des variables $x_i \in \{0, 1\}$:

- exprimer le OU logique $x_i \vee x_j$: contrainte $x_i + x_j \geq 1$;
- exprimer la négation $\overline{x}_i$: écrire $(1 - x_i)$;
- exprimer l'implication $x_i \Rightarrow x_j$: sachant que $x_i \Rightarrow x_j$ est équivalent à $\overline{x}_i \vee x_j$, on peut ajouter la contrainte $1 - x_i + x_j \geq 1$.

Finalement, mettons l'emphase sur des idées qui ne sont *pas* permises :

- il ne faut pas multiplier deux variables dans une même contrainte. Pr exemple, $10x_1 + 3x_2 \cdot x_3 \leq 1$ est interdit ;
- il ne faut pas utiliser des fonctions non-linéaires, mêmes si en apparence simple, telles que $min(x_1, x_2), max(x_1, x_2), |x_i|, \lfloor x_i \rfloor, \lceil x_i \rceil$;

7.2.1 Application à VERTEX-COVER

Soit $G = (V, E)$ un graphe. Rappelons que dans VERTEX-COVER, on cherche un $X \subseteq V$ de taille minimum qui touche chaque arête. Ceci peut s'exprimer avec un ILP. On va supposer que $V = \{v_1, \ldots, v_n\}$. Pour chaque v_i, on va créer une variable $x_i \in \{0, 1\}$. L'idée est que si dans une solution, on a $x_i = 1$, ceci représente $v_i \in X$, et si $x_i = 0$, $v_i \notin X$.

Minimiser

$$\sum_{i=1}^{n} x_i$$

Sujet à

$$\begin{aligned} x_i + x_j &\geq 1 && \text{pour tout } v_i v_j \in E \\ x_i &\in \{0,1\} && \text{pour tout } v_i \in V \end{aligned}$$

Ceci exprime VERTEX-COVER parce qu'on veut minimiser le nombre de variables avec $x_i = 1$ correspondant à ce qu'on mettra dans X. Les contraintes $x_i + x_j \geq 1$ demandent que pour chaque arête $v_i v_j \in E$, on ajoute au moins un de v_i ou v_j à X.

Cet ILP est NP-complet à résoudre. On va donc utiliser la relaxation suivante.

Minimiser

$$\sum_{i=1}^{n} x_i$$

Sujet à

$$\begin{aligned} x_i + x_j &\geq 1 && \text{pour tout } v_i v_j \in E \\ 0 \leq x_i &\leq 1 && \text{pour tout } v_i \in V \end{aligned}$$

Ce LP peut être résolu en temps polynomial. Soit $\boldsymbol{x}^*$ une solution optimale à ce LP. Le problème est que $\boldsymbol{x}^*$ peut exprimer la présence d'un v_i dans X de façon *fractionnelle*. C'est-à-dire, comment interpréter $x_i = 1/4$? Ou encore $x_i = 0.6180339\ldots$? À quoi cela correspond-il dans une solution concrète à VERTEX-COVER ? C'est ici qu'on doit souvent faire de créativité. Il faut trouver une façon de transformer la solution fractionnelle $\boldsymbol{x}^*$ en une solution concrète. Dans le cas du VERTEX-COVER, il s'avère qu'il suffit d'arrondir les x_i vers leur entier le plus près. Ceci en revient à inclure dans notre solution X chaque v_i tel que $x_i^* \geq 1/2$. Il faut ensuite argumenter qu'on ne s'éloigne pas trop de l'optimal $\sum_{i=1}^{n} x_i^*$. L'algorithme peut être résumé comme suit.

```
fonction vcLP(G = (V, E))
    Construire le LP de relaxation L ci-haut pour G
    Obtenir une solution optimale x* pour L
    X = ∅
    pour i = 1..n faire
        si x*_i ≥ 1/2 alors
            X.append(v_i)
    fin
    return X
```

Il faut maintenant argumenter que ceci donne une bonne approximation. Il y a deux éléments ici. Il faut s'assurer que X est bel et bien une solution faisable (i.e. X touche chaque arête), et ensuite il faut analyser le ratio d'approximation.

Théorème 17. *L'algorithme vcLP est une* 2*-approximation.*

Démonstration. On va d'abord argumenter que pour tout $v_iv_j \in E$, on a $v_i \in X$ ou $v_j \in X$ (ou les deux). Supposons que ce n'est pas le cas et qu'il existe $v_iv_j \in E$ tel que $v_i \notin X$ et $v_j \notin X$. Puisqu'aucun des deux n'a été ajouté à X, on avait $x_i^* < 1/2$ et $x_j^* < 1/2$. Ceci est une contradiction car $\boldsymbol{x}^*$ est censé satisfaire chaque contrainte de notre LP L, alors que ici on aurait $x_i^* + x_j^* < 1$. On déduit que X couvre bien chaque arête.

Analysons le ratio d'approximation. Puisque le LP L est une relaxation à VERTEX-COVER, on a $OPT \geq OPT_L = \sum_{i=1}^{n} x_i^*$. Pour chaque v_i qui est dans X, on avait $x_i^* \geq 1/2$. Ceci en revient à dire que $1 \leq 2x_i^*$. Donc,

$$APP = |X| = \sum_{v_i \in X} 1 \leq \sum_{v_i \in X} 2x_i^* = 2 \sum_{v_i \in X} x_i^* \leq 2 \sum_{i=1}^{n} x_i^* \leq 2 \cdot OPT$$

□

On appelle cette approche la *technique de l'arrondissement*. On prend les valeurs de $\boldsymbol{x}^*$ et on les arrondis pour obtenir un vecteur dans les $\{0, 1\}$. Bien sûr, ceci ne fonctionne pas toujours (en fait, rarement). Il y a plusieurs façons d'arrondir, et il faut parfois être créatif.

7.2.2 MAX-INDSET-3

Considérez le problème MAX-INDSET-3.

MAX-INDSET-3
Entrée : un graphe $G = (V, E)$ tel que $|N(v)| \leq 3$ pour tout $v \in V$
Sortie : un ensemble indépendant $X \subseteq V$ de taille maximum.

Le ILP correspondant est similaire à VERTEX-COVER.

Maximiser

$$\sum_{i=1}^{n} x_i$$

Sujet à

$$x_i + x_j \leq 1 \qquad \text{pour tout } v_i v_j \in E$$
$$x_i \in \{0, 1\} \qquad \text{pour tout } v_i \in V$$

La relaxation remplace les contraintes d'intégralité par $0 \leq x_i \leq 1$. Notons qu'on ne peut pas résoudre le LP puis se contenter de retourner $X = \{v_i : x_i^* \geq 1/2\}$ car il n'est pas garanti qu'un tel X est un ensemble indépendant. Par exemple, si G est une clique, le LP posera $x_i^* = 1/2$ pour tout v_i et X serait la clique. Pour garantir que l'on retourne une solution faisable, considérez l'alternative suivante.

fonction *maxIndset3LP($G = (V, E)$)*
  Construire le LP de relaxation L ci-haut
  Obtenir une solution optimale $\boldsymbol{x}^*$ pour L
  $X = \emptyset$
  $fini = false$
  tant que *pas* $fini$ **faire**
    si G *contient un sommet* v_i *tel que* $x_i^* \geq 1/2$ **alors**
      Ajouter v_i à X
      Retirer v_i et $N(v_i)$ de G
    sinon
      $fini = true$
  fin
  return X

Il devrait être clair que le X retourné est un ensemble indépendant.

Nous montrons que ceci est une 1/2-approximation. Nous vous invitons à généraliser les arguments qui suivent au problème MAX-INDSET-k. Pour le reste de cette section, X dénote l'ensemble retourné par l'algorithme. On va dénoter $N(X) = (\bigcup_{x \in X} N(x)) \setminus X$.

Lemme 11. *Pour tout $v_j \in N(X)$, on a $x_j^* \leq 1/2$.*

Démonstration. Pour tout $v_i \in X$, on a $x_i^* \geq 1/2$, ce qui implique selon les contraintes du LP que $x_j^* \leq 1/2$ pour tout $v_j \in N(v_i)$. Donc, chaque membre de $N(X)$ a un poids de 1/2 ou moins. □

Lemme 12. *À la fin de l'algorithme, il ne reste plus de sommet dans G.*

Démonstration. Soit Z l'ensemble des sommets restants à la fin de l'algorithme, et supposons pour fins de contradiction que $Z \neq \emptyset$. Soit $v_j \in Z$. On a $x_j^* < 1/2$ car sinon lalgorithme aurait ajouté v_j. De plus, v_j n'a pas de voisin dans X, car sinon v_j aurait été retiré. Donc, tous les voisins de v_j sont dans $N(X)$ ou dans Z. Tous les éléments de Z ont un poids inférieur à 1/2, et par le Lemme 11, tous les éléments de $N(X)$ ont un poids au plus 1/2. Donc, tous les voisins de v_j ont un poids au plus 1/2. Ceci veut dire qu'on pourrait poser $x_j^* = 1/2$ et satisfaire chaque contrainte du LP. Ceci augementerait la valeur totale du LP, une contradiction car $\boldsymbol{x}^*$ est censé être optimal. On déduit donc que $Z = \emptyset$. □

Notons qu'une déduction que l'on peut faire du lemme précédent est que $X \cup N(X) = V(G)$.

Théorème 18. *L'algorithme maxIndset3LP est une 1/2-approximation.*

Démonstration. Soit $v_i \in X$. On a $x_i^* \geq 1/2$, et donc $x_j^* \leq 1/2$ pour tout $v_j \in N(v_i)$. On argumente que $x_i^* + \sum_{v_j \in N(v_i)} x_j^* \leq 2$. On sait que

$$x_i^* = 1/2 + b$$

pour un certain $b \geq 0$, et que

$$x_j^* \leq 1 - x_i^* = 1 - (1/2 + b) = 1/2 - b$$

pour chaque voisin v_j de v_i (à cause des contraintes $x_i + x_j \leq 1$). Puisque v_i a au plus 3 voisins, on a

$$x_i^* + \sum_{v_j \in N(v_i)} x_j^* \leq 1/2 + b + 3(1/2 - b) = 4/2 - 2b \leq 2$$

Maintenant, l'intuition est que pour chaque sommet de X, OPT_{LP} avait une valeur de 2 ou moins. Pour argumenter plus précisément, par le lemme 12, on a $V(G) = X \cup N(X)$. Ceci mène à

$$OPT_{LP} = \sum_{v_i \in V} x_i^* \leq \sum_{v_i \in X} (x_i^* + \sum_{v_j \in N(v_i)} x_j^*) \leq \sum_{v_i \in X} 2 = 2|X| = 2APP$$

Donc, $APP \geq 1/2 \cdot OPT_{LP} \geq 1/2 \cdot OPT$. □

7.2.3 Livraison de paquets sur réseau circulaire

Soit $G = (V, E)$ un cycle de n sommets. On suppose que $V = \{1, 2, \ldots, n\}$ et $E = \{\{i, i+1\} : i \in \{1, \ldots, n\}\}$. On définit $n + 1 = 1$, ce qui simplifie la notation.

Ici, G représente un réseau circulaire. On a un ensemble de paquet à livrer, ou chaque paquet a une source et une destination. Pour chaque paquet (i, j), on doit choisir de faire circuler dans le sens horaire, ou bien anti-horaire. La charge du'une arête $e \in E$ est le nombre de paquets qui passent par e. Le but est de minimiser la charge maximum.

Pour chaque (i, j), on dénote par $H_{i,j}$ le chemin dans le sens horaire de i à j, et par $A_{i,j}$ le chemin dans le sens anti-horaire.

LIVRAISON-CIRCULAIRE
Entrée : cycle $G = (V, E)$ de n sommets, paquets $P = \{(i_1, j_1), \ldots, (i_m, j_m)\}$
Sortie : un choix de chemins W qui contient $H_{i,j}$ ou $A_{i,j}$ pour chaque $(i, j) \in P$, qui minimise le nombre maximum de paquets sur une arête, c'est-à-dire qui minimise

$$\max_{e \in E} |\{W' \in W : e \text{ est sur } W\}|$$

On va reformuler ce problème en un ILP. Nous avons mentionné plus haut qu'on ne pouvait pas utiliser min et max dans un ILP, alors que notre critère à minimiser contient max. On va utiliser un truc pour "simuler" ce max à l'aide de variables additionnelles. Ceci est possible lorsqu'on minimise un maximum. Nos variables seront :

— h_{ij} qui indique si on prend le chemin de i à j dans le sens horaire ;

— a_{ij} qui indique si on prend le chemin de i à j dans le sens anti-horaire ;
— c_e qui compte le nombre de chemins qui passent par l'arête e ;
— c_{max} qui est le c_e maximum (ce qu'on veut minimiser).

Les variables h_{ij} et a_{ij} devraient être 0 ou 1, mais nous passons directement à la relaxation.

Minimiser

$$c_{max}$$

Sujet à

$$\begin{aligned}
&h_{ij} + a_{ij} \geq 1 && \text{pour tout } (i,j) \in P \\
&c_e = \sum_{h_{ij}:e\in H_{ij}} h_{ij} + \sum_{a_{ij}:e\in A_{ij}} a_{ij} && \text{pour tout } e \in E \\
&c_{max} \geq c_e && \text{pour tout } e \in E \\
&0 \leq a_{ij}, h_{ij} \leq 1 && \text{pour tout } (i,j) \in P
\end{aligned}$$

Imaginons que $h_{ij} \in \{0,1\}$ et $a_{ij} \in \{0,1\}$. La première contrainte s'assure qu'on choisit un chemin pour chaque $(i,j) \in P$. La deuxième s'assure que c_e est le nombre de chemins choisis qui passent par e. La troisième contrainte s'assure que c_{max} est plus grand que tout les c_e. Puisqu'on cherche à minimiser c_{max}, cette variable n'a pas intérêt à dépasser la charge maximum d'une arête et sera donc égale.

Puisque la version ILP modélise le problème, notre LP est une relaxation et donc $OPT \geq OPT_{LP}$.

Nous allons utiliser le même truc d,arrondissement que dans VERTEX-COVER. Pour chaque $(i,j) \in P$, il faut que h^*_{ij} ou a^*_{ij} soit au moins $1/2$. Nous prenons le plus grand des deux, ce qui assure une 2-approximation.

```
fonction ringDeliveryLP(G = (V, E), P)
    Construire le LP de relaxation L ci-haut pour G
    Obtenir une solution optimale pour L, avec valeurs h*_ij et a*_ij
    W = ∅
    pour (i, j) ∈ P faire
        si h_ij ≥ 1/2 alors
            Ajouter H_ij à W
        sinon
            Ajouter A_ij à W
        fin
        return W
    fin
```

Un peu comme VERTEX-COVER, ceci est une 2-approximation.

Théorème 19. *ringDeliveryLP est une* 2*-approximation.*

Démonstration. On dénote par $h^*_{ij}, a^*_{ij}, c^*_e$ et c^*_{max} les valeurs d'une solution optimale au LP. Soit W l'ensemble des chemins retournés par ringDeliveryLP. Soit $e \in E$ et soient W_e les chemins de W qui contiennent e. Soit $W' \in W_e$, et disons que W' va de i à j. Si $W' = H_{ij}$, on a ajouté W' à W parce que $h_{ij} \geq 1/2$. Si $W' = A_{ij}$, c'est parce que $a_{ij} \geq 1/2$. Dans les deux cas, h_{ij} ou a_{ij} contribue au moins $1/2$ à c_e (parce que e est sur le chemin W'). Ceci veut dire que $c_e \geq 1/2 \cdot |W_e|$. La charge maximum de W est APP, ce qui veut dire que $c_{max} \geq 1/2APP$. C'est-à-dire, $APP \leq 2 \cdot c_{max} = 2 \cdot OPT_{LP} \leq 2 \cdot OPT$. □

7.3 Arrondissement aléatoire

Nous allons maintenant voir une technique dans laquelle l'arrondissement est fait de façon probabiliste. Nous utilisons le problème MAX-COVERAGE pour en donner l'exemple. Dans ce problème, on veut couvrir un nombre maximum d'éléments avec k ensembles.

MAX-COVERAGE
Entrée : univers $U = \{u_1, \ldots, u_n\}$, ensembles $S = \{S_1, \ldots, S_m\}$ et entier k.
Sortie : des ensembles $S' \subseteq S$ tels que $|S'| = k$ et qui maximisent $|\bigcup_{S_i \in S'} S_i|$, c'est-à-dire le nombre total d'éléments couverts par S'.

On peut exprimer ce problème avec un ILP — nous donnons directement la relaxation LP ici. On aura deux types de variables : x_i correspond aux choix d'un ensemble S_i, et y_j correspond à couvrir l'élément u_j.

Maximiser

$$\sum_{j=1}^{n} y_j \qquad \text{(maximiser le nombre d'éléments couverts)}$$

Sujet à

$$\sum_{i=1}^{m} x_i = k \qquad \text{(il faut choisir } k \text{ ensembles)}$$

$$y_j \leq \sum_{S_i : u_j \in S_i} x_i \qquad \text{pour tout } u_j \in U$$

$$0 \leq x_i \leq 1 \qquad \text{pour } i = 1..m$$

$$0 \leq y_j \leq 1 \qquad \text{pour } j = 1..n$$

En exercice, vous pouvez démontrer que ceci est bel et bien une relaxation de MAX-COVERAGE et donc que $OPT \leq OPT_{LP}$. Encore une fois, ceci est parce que toute solution faisable à MAX-COVERAGE donne une solution faisable au LP, et donc le LP ne peut que mieux faire.

Soit $\boldsymbol{x}^* = (x_1^*, x_2^*, \ldots, x_m^*)$ les valeurs des $\boldsymbol{x}$ d'une solution optimale à ce LP, et soit $\boldsymbol{y} = (y_1^*, \ldots, y_n^*)$ les valeurs des $\boldsymbol{y}$. La somme de ces x_i^* doit être k et representent des fractions de présence d'ensembles de S. On va interpréter ces x_i^* comme des probabilités. En fait, la somme des x_i^* devrait être 1 pour ce faire, alors qu'elle est présentement k. On va donc interpréter

$$(x_1^*/k, x_2^*/k, \ldots, x_m^*/k)$$

comme une distribution de probabilités. On va tirer k ensembles au hasard

selon cette probabilité, avec remplacement. Donc il se peut qu'on pige le même ensemble plusieurs fois — dans ce cas, tant pis, on retourne moins de k ensembles différents et on couvre moins d'éléments que l'on aurait pu.

fonction *maxcoverLP(U, S, k)*
  Construire le LP de relaxation L ci-haut
  Obtenir une solution optimale $\boldsymbol{x}^*$ pour L
  $S' = \emptyset$
  tant que $|S'| < k$ **faire**
    Choisir un indice i au hasard, où chaque i est choisi avec probabilité x_i^*/k
    Ajouter S_i à S'
  fin
  return S'

En pratique, on voudrait bien sûr éviter de choisir le même ensemble deux fois. On pourrait ajouter un autre ensemble lorsque ceci survient, mais nos calculs seraient beaucoup plus complexes.

Pour analyser la précision de cet algorithme, on doit s'intéresser à la probabilité de couvrir un élément $u_j \in U$. Le LP nous donne déjà une "fraction de couverture" de u_j avec la valeur de y_j^*. La probabilité de couvrir u_j sera donc mise en perspective avec y_j^*. Puisque cette dernière valeur contribue à OPT_{LP}, on arrivera à se comparer à l'optimal.

Lemme 13. *Pour tout $u_j \in U$, la probabilité que S' couvre u_j est au moins $y_j^* \cdot (1 - 1/e)$, où $e \simeq 2.71828$ est la constante d'Euler et S' est l'ensemble retourné par maxCoverageLP.*

Démonstration. Si on choisit un seul ensemble, la probabilité de couvrir u_j est la somme des probabilités des ensembles qui contiennent u_j, qui est $\sum_{S_i:u_j \in S_i} x_i^*/k = \frac{1}{k} \cdot \sum_{S_i:u_j \in S_i} x_i^*$. Il s'avère que le LP spécifie une contrainte avec cette somme. C'est-à-dire, on sait que $\sum_{S_i:u_j \in S_i} x_i^* \geq y_j^*$, et donc la probabilité de couvrir u_j en choisissant un ensemble est au moins

$$\frac{1}{k} \cdot \sum_{S_i:u_j \in S_i} x_i^* \geq \frac{y_j^*}{k}$$

Donc, on a une probabilité $1 - \frac{y_j^*}{k}$ de ne pas couvrir u_j avec un ensemble.

Puisque chaque ensemble est choisi de façon indépendante aux choix

précédents, la probabilité de ne jamais couvrir u_j en choisissant k ensembles est donc $\left(1 - \frac{y_j^*}{k}\right)^k$. Et la probabilité de couvrir u_j est de $1 - \left(1 - \frac{y_j^*}{k}\right)^k$. Il est possible de montrer que

$$1 - \left(1 - \frac{y_j^*}{k}\right)^k \geq (1 - 1/e)y_j^*$$

mais nous le laissons en exercice. Une façon de le voir est que la fonction de gauche, par rapport à y_j^*, est convexe et plus grande ou égale à celle de droite aux points $y_j^* \in \{0, 1\}$, alors que celle à droite est linéaire. Ceci démontre le lemme. □

Théorème 20. *maxcoverLP est une* $(1 - 1/e)$*-approximation probabiliste.*

Démonstration. Soit I_j une variable indicatrice telle que $I_j = 1$ si l'algorithme couvre u_j, et $I_j = 0$ sinon. On a

$$\begin{aligned}
\mathbb{E}[APP] = \mathbb{E}[\sum_{j=1}^{n} I_j] = \sum_{j=1}^{n} \mathbb{E}[I_j] &\geq \sum_{j=1}^{n} (1 - 1/e)y_j^* \\
&= (1 - 1/e) \sum_{j=1}^{n} y_j^* \\
&= (1 - 1/e) \cdot OPT_{LP}
\end{aligned}$$

□

Deuxième partie

Algorithmes à complexité paramétrée

Chapitre 8

La complexité paramétrée

Dans cette deuxième partie, nous allons étudier les algorithmes *exacts*, c'est-à-dire qui garantissent nous retourner une solution optimale. Puisque nous étudions les problèmes NP-complets, nous ne pouvons pas nous attendre à ce que ces algorithmes s'exécutent en temps polynomial. On doit plutôt se diriger vers des algorithmes exponentiels.

Ceci est un changement de paradigme majeur par rapport aux algorithmes d'approximation. Il n'est plus nécessaire de comparer la solution de l'algorithme avec OPT, car cette solution doit toujours être égale. Aussi, nos algorithmes ne sont plus tenus d'être en temps polynomial.

Toutefois, nous souhaitons *limiter* l'ampleur de la complexité exponentielle. Nous allons supposer que nous traitons des données sur lesquelles un certain paramètre k est petit, et ce indépendamment de n. Dans ce cas, un algorithme en temps $O(2^k \cdot n)$ sera beaucoup plus efficace qu'un algorithme en temps $O(2^n)$. Si k reste petit, $O(2^k \cdot n)$ demeure applicable pour n allant des les dizaines de milliers, alors que $O(2^n)$ n'ira jamais vraiment plus loin que $n = 100$. Par exemple, on peut se demander s'il existe un VERTEX-COVER de taille au plus k, et on peut supposer que peu importe le nombre de sommets n, il nous intéresse seulement de savoir si on pouvait bloquer l'entrée à k personnes ou moins.

L'esprit de la complexité paramétrée consiste donc à isoler la partie exponentielle d'un algorithme sur un paramètre k, tandis que n doit rester polynomial. En fait, on va permettre une complexité $f(k)$ pour *n'importe quelle* fonction f en ce qui concerne la partie affectant k. L'important est que k n'est pas traité comme une constante, et donc le polynome qui affecte n doit rester indépendant de k. Donc, une complexité $O(2^k \cdot n^{100})$ est acceptable, mais *pas* une complexité $O(n^k)$.

Les algorithmes "force-brute" essaient toutes les solutions possibles et résultent souvent en un algorithme $O(2^n)$. La compelxité paramétrée peut être vue comme une "force brute intelligente", dans laquelle on tente de limiter l'énumération des possibilités en fonction de k et non en fonction de n.

8.1 Définition d'un algorithme FPT

Soit P un problème algorithmique. Contrairement aux algorithmes d'approximation, il n'y a pas de distinction fondamentale entre un problème de minimisation ou de maximisation (à quelques exceptions près). Soit I une instance de P et soit k une valeur numérique. On dira que (I, k) est une instance de P paramétrée (par k).

Définition 2. *Un problème P est résoluble à paramètre fixe (en anglais,* fixed parameter tractable*) s'il existe un algorithme qui, pour toute instance paramétrée (I, k) de P, retourne une solution optimale en temps $O(f(k)\cdot n^c)$, où f est une fonction arbitraire, n est la taille de l'instance I et c est une constante indépendante de k.*

On dira que P est FPT en le paramètre k.

Notez aussi que l'interprétation de k n'est pas donnée. C'est à nous de choisir ce que représente ce k. Remettons encore une fois l'emphase sur le fait que l'exposant qui affecte n doit être indépendant de k. La fonction f, quand à elle, est arbitraire, ce qui permet parfois des complexités déconnectées de la pratique. Par exemple, un algorithme en temps

$$O(2^{2^{2^{2^k}}} \cdot n^{1000})$$

est admissible. Nous allons éviter ces cas la plupart du temps, et nos algorithmes auront plutôt une forme applicable en pratqieu telle que $O(2^k \cdot n)$.

8.2 L'exemple canonique : VERTEX-COVER

Il existe un algorithme $O(2^k \cdot n)$ très simple pour VERTEX-COVER, où k est le nombre de sommets d'une solution optimale.

Analysons d'abord l'algorithme naïf force-brute.

fonction *vertexCoverBrute*$(G = (V, E))$
  $X = V$
  pour *chaque sous-ensemble* $S \subseteq V$ **faire**
    si S *couvre toutes les arêtes et* $|S| < |X|$ **alors**
      $X = S$
    fin
  fin
  return X

Il y a 2^n sous-ensembles possibles, et vérifier si un sous-ensemble couvre les arêtes prend un temps $O(n + m)$, où $m = |E|$. La complexité de cet algorithme est $O(2^n \cdot (n + m))$.

Étudions la version paramétrée de ce problème.

VERTEX-COVER
Entrée : graphe $G = (V, E)$
Paramètre : k, la taille maximum voulue d'une solution
Sortie : un ensemble $X \subseteq V$ couvrant les arêtes tel que $|X| \leq k$, ou bien *null* si un tel ensemble n'existe pas

Notez qu'en réalité, on ne cherche pas la solution optimale. On veut seulement savoir si un VERTEX-COVER de taille au plus k existe. Ceci entre en contraste avec la notion d'algorithme exact énoncée plus tôt. Le fait est que si u algorithme pour VERTEX-COVER paramétré existe, on peut essayer tous les k en ordre croissant pour trouver celui qui est optimal.

Voici une **mauvaise** tentative pour obtenir un algorithme FPT à ce problème. On tente d'être plus brillant que l'algorithme naïf et on se dit qu'on peut utiliser le paramètre k pour borner notre recherche force-brute. Ceci ne fonctionne pas.

```
fonction vertexCoverSemiBrute(G = (V, E), k)
    pour i = 1..k faire
        pour chaque sous-ensemble S ⊆ V de taille i faire
            si S couvre toutes les arêtes et |S| < |X| alors
                return X
            fin
        fin
    fin
    return null
```

Cet algorithme est correct, mais quelle est sa complexité ? Il y a $\binom{n}{i}$ sous-ensembles possibles de i sommets, et $\binom{n}{i} \in \Omega(n^i)$ (rappel : $\Omega(f(n))$ peut-être vu comme l'inverse de O et veut dire "grandit au moins aussi vite que $f(n)$). Dans le pire cas, l'algorithme va tout essayer et le nombre d'ensembles testés sera au moins

$$\sum_{i=1^k} \binom{n}{i} \geq \binom{n}{k} \in \Omega(n^k)$$

La complexité de cet algorithme est donc $\Omega(n^k)$. Puisque l'exposant de n dépend de k, ce n'est pas de la complexité paramétrée.

Voici un algorithme qui fonctionne. On observe que pour chaque arête uv, il faut inclure u dans notre couverture, ou bien v. On va donc choisir une arête arbitraire uv et brancher sur les deux possibilités. Si on décide d'ajouter u, ses arêtes incidents sont couvertes et on peut les retirer du graphe (même chose avec v). Après avoir choisi d'ajouter u ou v, on a droit à un choix de moins à notre couverture par sommets et on peut décrémenter k. On arrête lorsqu'on a trouvé une couverture, ou bien lorsque k est devenu 0 et qu'on n'a plus le droit de rien ajouter. On va passer en paramètre l'ensemble X qu'on a construit jusqu'à maintenant (à l'appel initial, $X = \emptyset$).

```
fonction vertexCoverFPT(G = (V, E), k, X)
    si X couvre toutes les arêtes alors
        return X
    fin
    si k = 0 alors
        return null
    fin

    Soit uv ∈ E choisie arbitrairement
    Soit G_u obtenu de G en retirant u et ses arêtes
    X_u = vertexCoverFPT(G_u, k − 1, X ∪ {u})
    Soit G_v obtenu de G en retirant v et ses arêtes
    X_v = vertexCoverFPT(G_v, k − 1, X ∪ {v})

    si X_u ≠ null alors
        return X_u
    sinon si X_v ≠ null alors
        return X_v
    sinon
        return null
    fin
```

Cet algorithme est correct parce qu'il teste toutes les possibilités pour l'arête uv, qui doit être couverte peu importe ce qu'on fait. L'algorithme effectue ce qu'on appelle un branchement borné, une technique très répandue en complexité paramétrée. Pour évaluer la complexité de cet algorithme, il suffit de noter qu'il construit un arbre de récursion où chaque sommet représente un appel, avec la racine l'appel initial. Chaque sommet a 2 enfants (2 appels récursifs) et la hauteur de l'arbre est bornée par k. On a donc un arbre binaire de hauteur k, et on sait (n'est-ce pas ?) qu'un tel arbre a $O(2^k)$ sommets. Puisque le temps pour traiter un appel est $O(n)$, la complexité est $O(2^k \cdot n)$.

8.3 Un autre exemple : MAX-CLIQUE

Soit $G = (V, E)$ un graphe. Rappelons qu'une clique est un ensemble de sommets X tel que pour tout $u, v \in X$ distincts, $uv \in E$. On peut tenter de paramétriser le problème de recherche de clique maximum comme suit.

MAX-CLIQUE (paramètre "taille de la clique")
Entrée : graphe $G = (V, E)$
Paramètre : k, la taille voulue d'une clique
Sortie : une clique $X \subseteq V$ de taille k, si elle existe, ou bien $false$ sinon

L'algorithme naïf testerait tous les sous-ensembles de taille k et prendrait un temps au moins $O(n^k)$. Ce n'est donc pas une option viable.

Il s'avère que malgré de années de recherche, personne n'a trouvé d'algorithme FPT pour MAX-CLIQUE paramétrisé par la taille de la clique. En fait, on croit qu'il est impossible qu'il existe un algorithme $O(f(k) \cdot n^c)$ pour ce problème. MAX-CLIQUE est connu comme étant $W[1]$-difficile. Il y a une définition formelle de ce que ceci signifie, mais elle sort du cadre de ces notes. Disons que $W[1]$-difficile est l'analogue de la NP-complétude pour les problèmes FPT, et qu'un problème $W[1]$-dificile veut dire que le problème n'est probablement pas FPT.

On peut toutefois tenter une autre paramétrisation. Disons que le degré de G n'est pas trop grand (le degré est le nombre maximum de voisins d'un noeud). Donc, chaque sommet a au plus k voisins.

MAX-CLIQUE (paramètre "degré maximum")
Entrée : graphe $G = (V, E)$
Paramètre : k, le degré maximum d'un sommet de G
Sortie : une clique $X \subseteq V$ de taille maximum

Notez quelques subtilités ici. Puisque le paramètre k n'a aucun lien (direct) avec la clique maximum, on exige que la valeur de retour soit la clique maximum. On doit plutôt se débrouiller pour isoler la complexité par rapport au degré du graphe. Il devient possible de faire une "force brute intelligente". Il suffit de considérer chaque sommet et de regarder si lui et son voisinage forment une assez grosse clique.

```
fonction maxCliqueDegreFPT(G = (V, E) tel que |N(v)| ≤ k pour
 tout v ∈ V)
   X = ∅
   pour v ∈ V faire
      pour chaque sous-ensemble S ⊆ N(v) faire
         si S ∪ {v} est une clique et |S| + 1 > |X| alors
            X = S ∪ {v}
         fin
      fin
   fin
   return X
```

Lorsqu'on développe un tel algorithme, il faut bien sûr démontrer qu'il est correct, et que sa complexité est FPT.

Théorème 21. *L'algorithme maxCliqueDegreFPT retourne une clique de taille maximum et s'exécute en temps $O(k^2 2^k \cdot n)$. Le problème MAX-CLIQUE est donc FPT en le paramètre "degré du graphe".*

Démonstration. Il est clair que l'algorithme est correct. La clique maximum est formée d'un sommet $v \in V$ et d'un sous-ensemble de son voisinage et l'algorithme teste toutes les possibilités.

Pour la complexité, la boucle principale itère sur n sommets. Pour chaque $v \in V$, on va énumérer les sous-ensembles de $N(v)$. Puisque $|N(v)| \leq k$, il y a au plus 2^k tels sous-ensembles. De plus, chaque sous-ensemble est de taille au plus k, et vérifer que chaque paire de $N(v)$ a une arête peut se fair en temps $O(\binom{k}{2}) = O(k^2)$. Donc, le traitement d'un $v \in V$ se fait en temps $O(k^2 2^k)$ et il y a n sommets à traiter. La complexité est donc $O(k^2 2^k \cdot n)$. □

Peut-on paramétriser VERTEX-COVER par le degré de G ?

Les mêmes idées de paramétrisation par le degré de G échouent sur VERTEX-COVER. En fait, ce n,est probblement pas possible. Ce problème st donc dans la situation inverse à MAX-CLIQUE.

La raison pour laquelle ce n'est pas possible est que VERTEX-COVER est NP-complet même si le graphe en entrée est de degré 3. Si VERTEX-COVER était FPT en le degré de G, on aurait un algorithme $O(f(k) \cdot n^c)$ pour le problème, où k est le degré maximum. Ceci impliquerait qu'en posant $k = 3$, on aurait un algorithme $O(f(3) \cdot n^c)$. Mai ici, $f(3)$ est une constante

peu importe le f (par exemple, si $f(k) = 2^k$, alors $2^3 = 8$ est une constante). Donc avec $k = 3$, on aurait un algorithme $O(n^c)$, donc en temps polynomial, pour VERTEX-COVER avec degré 3. On aurait donc résolu un problème NP-complet en temps polynomial ! Ceci veut dire que si VERTEX-COVER était FPT en le degré, on aurait P = NP. Il faudrait un autre cours pour apprécier la profondeur de cet énoncé. Disons seulement que personne n'a prouvé que ce n'est pas possible, mais il est très improbable que cela se produise.

La façon habituelle de formuler ce genre de résultat est d'utiliser la contraposée, comme suit.

Théorème 22. *Si P $\neq$ NP, alors VERTEX-COVER paramétrisé par le degré du graphe n'est pas FPT.*

Chapitre 9

Algorithmes de branchement

L'idée d'un algorithme de branchement est d'identifier un nombre limité de cas possibles pouvant former une solution optimale, puis de brancher récursivement sur chaque possibilité. Afin d'obtenir un algorithme FPT, deux conditions doivent généralement être satisfaites :

— le nombre de cas doit être borné par une fonction $f(k)$;

— chaque cas doit réduire la valeur de k.

Si ces deux conditions sont remplies, l'arbre de récursion sera tel que chaque noeud a $f(k)$ enfants et tel que la profondeur est bornée par $p(k)$, où f et p sont des fonctions quelconques. Conséquemment, l'arbre aura au plus $f(k)^{p(k)}$ sommets, donnant ainsi un algorithme FPT (en supposant que chaque récursion est en temps polynomial).

Nous avons déjà vu un exemple d'algorithme de branchement avec VERTEX-COVER. Chaque noeud de l'arbre de récursion avait $f(k) = 2$ enfants et la profondeur était bornée par $p(k) = k$, donnant ainsi un algorithme $O(2^k n^c)$.

Beaucoup de problèmes peuvent être résolus avec des algorithmes de branchement. L'analyse de l'arbre de récursion devient parfois complexe, et nous développeront des outils pour ce faire. Mais avant, voyons quelques exemples de base.

9.1 3-HITTING SET

Dans le problème 3-HITTING-SET, on reçoit des ensembles $S_1, S_2, \ldots, S_m$ ayant chacun 3 éléments, sur univers $U = \{u_1, \ldots, u_n\}$. On doit choisir un nombre minimum d'éléments de U afin d'avoir une intersection avec chaque S_i. On peut imaginer une situation dans laquelle des étudiants offrent chacun 3 journées de disponibilités pour un rendez-vous avec un professeur,

et ce dernier doit choisir un nombre minimum de journées de façon à ce que chaque étudiant ait une disponibilité parmi ces journées. Nous allons paramétriser par le nombre d'éléments choisis.

3-HITTING-SET
Entrée : ensembles $S = \{S_1, \ldots, S_m\}$ chacun de taille 3 sur univers $U = \{u_1, \ldots, u_n\}$
Paramètre : k, la taille d'une solution
Sortie : sous-ensemble $X \subseteq U$ tel que $X \cap S_i \neq \emptyset$ pour tout $i = 1..m$ et tel que $|X| \leq k$, ou null si non-existant

L'idée est à peu près la même que VERTEX-COVER (en fait, ce problème est une généralisation de VERTEX-COVER). Pour chaque $S_i = \{a, b, c\}$, il faut inclure a, b ou c. On branche sur les trois possibilités et on arrête quand on a atteint k éléments. Quand on branche sur un des cas, on enlève tous les S_i qui deviennent couverts.

fonction *3hitsetFPT*(S, U, k, X)
  si $k < 0$ **alors**
    return *null*
  fin
  si $S = \emptyset$ **alors**
    return X
  fin
  Choisir $S_i \in S$ arbitrairement
  Soient a, b, c les éléments de S_i
  $X_a = 3hitsetFPT(S \setminus \{S_j : a \in S_j\}, U, k-1, X \cup \{a\})$
  $X_b = 3hitsetFPT(S \setminus \{S_j : b \in S_j\}, U, k-1, X \cup \{b\})$
  $X_c = 3hitsetFPT(S \setminus \{S_j : c \in S_j\}, U, k-1, X \cup \{c\})$
  si *un de* X_a, X_b *ou* X_c *n'est pas null* **alors**
    Retourner celui de X_a, X_b, X_c qui n'est pas *null*
  sinon
    return *null*
  fin

L'algorithme est correct parce qu'il teste toutes les façons d'avoir une intersection avec chaque S_i.

La complexité de cet algorithme s'évalue rapidement. Il faut un temps $O(m)$ pour chaque appel afin de trouver les S_j à retirer. L'algorithme crée un arbre de récursion où chaque noeuds a 3 enfants, et la profondeur est bornée par k. Il y a donc $O(3^k)$ noeuds. La complexité est $O(3^k \cdot m)$.

9.2 Complexité biparamétrée

Nos paramétrisation jusqu'ici ne contenaient qu'un seul paramètre, mais il est en fait possible de définir autant de paramètres que l'on veut. Disons qu'on veut que notre complexité soit exponentielle par rapport à l paramètres $k_1, k_2, \ldots, k_l$. Il suffit de paramétriser selon le paramètre $k = k_1 + k_2 + \ldots + k_l$. Une fonction $f(k)$ sera aussi une fonction $f(k_1, \ldots, k_l)$. Bien sûr, en pratique on souhaite limiter le nombre de paramètres, mais dans certains cas, au moins deux paramètres sont nécessaires.

Prenons par exemple le problème HITTING-SET. Nous avons vu ci-haut la version dans laquelle il y a 3 éléments par ensemble. Dans le cas général, il n'y a pas de borne sur le nombre d'éléments et le problème devient W[2]-difficile (donc, probablement pas d'algorithme FPT en le nombre d'éléments). Par contre, on peut paramétriser par d, le nombre maximum d'éléments dans un ensemble.

d-HITTING-SET
Entrée : ensembles $S = \{S_1, \ldots, S_m\}$ chacun de taile au plus d sur univers $U = \{u_1, \ldots, u_n\}$
Paramètre : $k + d$, où k est la taille d'une solution
Sortie : sous-ensemble $X \subseteq U$ tel que $X \cap S_i \neq \emptyset$ pour tout $i = 1..m$ et tel que $|X| \leq k$, ou null si non-existant

Il suffit de généraliser la technique de branchement vue ci-haut.

fonction *d-hitsetFPT*(S, U, k, X)
  si $k < 0$ **alors**
    return *null*
  fin
  si $S = \emptyset$ **alors**
    return X
  fin
  Choisir $S_i \in S$ arbitrairement
  pour $x \in S_i$ **faire**
    X_s =d-hitsetFPT$(S \setminus \{S_j : x \in S_j\}, U, k-1, X \cup \{x\})$
    si $X_s \neq null$ **alors**
      return X_s
  fin
  return *null*

Sachant que chaque S_i a d éléments ou moins, l'algorithme va brancher sur d cas ou moins. La profondeur de l'arbre est bornée par k, et donc on fera $O(d^k)$ appels récursifs. Chaque appel prend un temps $O(m)$, et donc la complexité est $O(d^k \cdot m)$.

Notez qu'en pratique, on choisirait toujours le S_i avec un nombre minimum d'éléments, question de limiter le nombre d'appels récursifs effectués.

9.3 CLUSTER-EDITING

Dans le CLUSTER-EDITING, on a un graphe et on veut modifier (ajouter/enlever) un nombre minimum d'arêtes pour que chaque composante connexe de G soit une clique, qui forment ce qu'on appelle des clusters.

CLUSTER-EDITING
Entrée : graphe $G = (V, E)$
Paramètre : k, le nombre d'arêtes à modifier
Sortie : une liste d'arêtes E^+ à ajouter et une liste d'arêtes E^- à retirer telles que $|E^+|+|E^-| \leq k$ et telles que $G' = (V, (E \cup E^+) \setminus E^-)$ n'a que des cliqeus comme composantes connexes.

Nous allons développer un arbre de récursion avec $O(3^k)$ noeuds avec la

propriété suivante (à prouver en exercice).

Théorème 23. *Chaque composante connexe d'un graphe G est une clique si et seulement si pour tout $u, v, w \in V$, $uv \in E$ et $vw \in E$ implique que $uw \in E$.*

Notez que si $uv \in E, vw \in E$ mais $uw \notin E$, les sommets u, v, w forment un chemin avec trois sommets, ce qu'on appelle un P_3. Le CLUSTER-EDITING est donc équivalent à modifier un minimum d'arêtes pour que le graphe n'ait pas de P_3.

On va trouver trouver des u, v, w conflictuels (qui forment un P_3), et brancher sur les façons de corriger le problème. On note que si $uv \in E$ et $vw \in E$ mais $uw \notin E$, on a trois façons de satisfaire le théorème :
– retirer uv
– retirer vw
– ajouter uw.

Chaque cas modifie une arête et réduit k de 1.

fonction *clusterEditing*$(G = (V, E), k, E^+, E^-)$
  si $k < 0$ **alors**
    return *null*
  fin
  si G *n'a pas de* P_3 **alors**
    return (E^+, E^-)
  fin

  Trouver u, v, w qui forment un P_3
  Soit G_1 obtenu de G en enlevant uv
  $(E_1^+, E_1^-) = clusterEditing(G_1, k-1, E^+, E^- \cup \{uv\})$
  Soit G_2 obtenu de G en enlevant vw
  $(E_2^+, E_2^-) = clusterEditing(G_2, k-1, E^+, E^- \cup \{vw\})$
  Soit G_2 obtenu de G en ajoutant uw
  $(E_3^+, E_3^-) = clusterEditing(G_3, k-1, E^+ \cup \{uw\}, E^-)$

  si *un des appels n'a pas retournée null* **alors**
    Retourner la solution non *null*
  sinon
    return *null*
  fin

Chaque appel prend un temps $O(n^3)$ car il faut trouver u, v et w (et il n'est pas trivial de faire mieux que regarder chaque triplet). Cet algorithme crée un arbre de récursion où chauqe noeud a trois enfants, et la profondeur est k ou moins. La complexité est donc $O(3^k n^3)$.

9.4 Des branchements plus intelligents

Si on creuse un peu, on peut souvent raffiner nos branchements afin de réduire le nombre d'enfants des noeuds ou bien la profondeur (ironique que creuser réduise la profondeur !). Le prix à payer est qu'il faut toutefois travailler un peu plus fort et analyser des récurrences plus complexes. Revenons à VERTEX-COVER.

Dans l'algorithme vu précédemment, on branchait sur deux cas sur une arête uv :
– inclure u ;
– inclure v.

Une façon alternative de voir ceci est de brancher sur deux cas :
– inclure u ;
– ne pas inclure u.

Mais si on décide que u n'est pas dans la solution, il n'est pas capable de couvrir ses arêtes. Disons que les voisins de u sont $v_1, \ldots, v_l$. Si on n'inclut pas u, il faut inclure v_1 pour couvrir uv_1, et il faut inclure v_2 pour couvrir uv_2, etc. Donc, le choix de ne pas inclure u nous force à inclure $deg(u)$ sommets. Si $deg(u) \geq 2$, ceci nous permettra de réduire k de plus que 1 et limitera ainsi la profondeur de l'arbre de récursion.

Écrivons d'abord l'algorithme. On veut brancher sur u tel que $deg(u) \geq 2$. Si un tel u n'existe pas, tous les sommets ont un degré 0 ou 1 et il est trivial de décider s'il y a une solution de taille k ou moins (exercice !).

```
fonction vc2(G = (V, E), k, X)
    si k < 0 alors
        return null
    fin
    si G n'a pas d'arête alors
        return X
    fin
    si G n'a que des sommets de degré 0 ou 1 alors
        Trouver la couverture optimale X'
        Si |X'| ≤ k, retourner X' ∪ X, sinon retourner null
    fin

    Soit u un sommet de degré au moins 2
    Obtenir G_u en retirant u et ses arêtes de G
    X_1 = vc2(G_u, k − 1, X ∪ {u})

    Obtenir G* en retirant {u} ∪ N(u) et leurs arêtes de G
    X_2 = vc2(G*, k − |N(u)|, X ∪ N(u))

    si un de X_1 ou X_2 n'est pas null alors
        le retourner
    sinon
        return null
    fin
```

Cet algorithme est correct : quand on considère un u, lorsqu'on l'inclut dans notre solution, ses arêtes sont couvertes. Par contre, si on n'inclut pas u, il faut inclure tous ses voisins.

Pour la complexité, on va se concentrer uniquement sur le nombre de noeuds de l'arbre de branchement. Soit $t(k)$ le nombre de tels noeuds sur entrée k. Lorsque des appels récursifs sont faits, on a

$$t(k) = t(k-1) + t(k-|N(u)|)$$

Plus $N(u)$ est grand, plus $t(k)$ sera petit. Puisqu'on a $|N(u)| \geq 2$, on peut supposer que dans le pire cas,

$$t(k) = t(k-1) + t(k-2)$$

Même sans connaître les conditions initiales de cette récurrence, il est possible de déduire que

$$t(k) \in O(1.618^k)$$

En fait, $t(k)$ est la fonction de Fibonacci.

L'algorithme ci-haut prend donc un temps $O(1.618^k \cdot (n+m))$, une amélioration significative sur le 2^k qu'on avait plus tôt.

9.5 Comment résoudre les récurrences ?

La réponse simple à cette question est d'utiliser un logiciel. Rares sont les créateurs d'algorithmes qui analysent les récurrences manuellement, sauf quand elles sont trop compliquées pour un programme. Mais il est important de savoir comment utiliser la sortie des logiciels.

Les récurrences les plus fréquentes sont appelées *homogène linéaire*. Elles ont la forme

$$t(k) = a_1 t(k-1) + a_2 t(k-2) + \ldots + a_d t(k-d)$$

où d est une constante, et les a_i sont aussi des constantes.

L'idée est que ces récurrences sont $O(c^k)$ pour une constante c qu'il faut déterminer. Pour trouver ce c, on va simplement supposer que $t(k) = c^k$ et résoudre. Ceci n'est pas toujours vrai car les constantes sont ignorées, mais en terme de O, tout reste valide.

Donc, en supposant que $t(k) = c^k$, on a

$$\begin{aligned} t(k) &= a_1 t(k-1) + a_2 t(k-2) + \ldots a_d t(k-d) \\ c^k &= a_1 c^{k-1} + a_2 c^{k-2} + \cdots a_d c^{k-d} \end{aligned}$$

En mettant tout d'un même côté, on a

$$c^k - a_1 c^{k-1} - a_2 c^{k-2} - \cdots a_d c^{k-d} = 0$$

On peut tout diviser par c^{k-d} et obtenir

$$c^d - a_1 c^{d-1} - a_2 c^{d-2} - \cdots a_d c^0 = 0$$

Ceci est un polynôme de degré d avec c comme variable. On l'appelle le polynôme caractéristique.

Puisque le degré est d, il y a d racines possibles (i.e. d valeurs de c telles que le polynôme donne 0). Les logiciels d'aujour'hui peuvent sortir toutes les

racines. Certaines sont parfois complexes (dans le sens de nombre complexe), mais il s'avère qu'on peut prendre la plus grande racine réelle pour obtenir c. C'est-à-dire, si les racines réelles du polynôme sont $r_1, r_2, \ldots, r_l$, le nombre de noeuds de notre arbre est $O(\max\{r_1, \ldots, r_l\}^k)$.

Par exemple, le cas VERTEX-COVER ci-haut donnait

$$t(k) = t(k-1) + t(k-2)$$

ce qui devient

$$\begin{aligned} c^k &= c^{k-1} + c^{k-2} \\ c^k - c^{k-1} - c^{k-2} &= 0 \\ c^2 - c - 1 &= 0 \end{aligned}$$

Wolfram nous dit que les racines sont environ 1.618 et -0.618. On prend la plus grande $c = 1.618$ et le nombre de noeuds de l'arbre de récursion est $O(1.618^k)$.

La recette de récurrences homogènes linéaires va donc comme suit :

1. Poser $t(k)$ comme une récurrence ;
2. Supposer que $t(k) = c^k$;
3. Écrire le polynôme caractérisque ;
4. Trouver les racines du polynôme ;
5. Soit r la plus grande racine réelle ;
6. L'arbre de récursion a $O(r^k)$ noeuds.

9.6 Un 3-HITTING-SET amélioré

Nous allons améliorer notre algorithme de 3-HITTING-SET énoncé plus haut en creusant un peu plus sur nos cas de branchement. L'idée est la suivante : prenons deux ensembles S_i et S_j tels que leur intersection $S_i \cap S_j$ est maximum. Supposons que $S_i = \{x, y, z\}$ et $S_j = \{x, y, a\}$. On s'imagine ce qu'il faut inclure à notre solution X pour couvrir S_i et S_j. Il y a trois possibilités : inclure x, inclure y, ou inclure a et b. Les deux premiers cas réduisent k de 1, mais le dernier cas réduit k de 2. On obtient la récurrence

$$t(k) = 2t(k-1) + t(k-2)$$

Il est possible que $S_i = \{x, y, z\}$ et $S_j = \{x, a, b\}$. On peut inclure x, ou bien inclure un de y, z et un de a, b. Les choix possibles sont donc d'inclure

$\{x\}$, $\{y, a\}, \{y, b\}, \{z, a\}, \{z, b\}$. Le premier diminue k de 1, les quatre autres diminuent k de 2. Ce cas mène à la récurrence

$$t(k) = t(k-1) + 4t(k-2)$$

Il est aussi possible que tous les S_i et S_j n'aient aucune intersection. Il faut gérer ce cas séparément, mais ceci est simple. Si les S_i ne partagent pas d'élément, il faut inclure au moins un élément de chaque S_i. Donc, si $|S_i| > k$, on retourne *null*, et sinon on retourne un élément par S_i. Le pseudo-code est décrit ci-bas. Notez qu'au lieu de rendre les appels récursifs explicites, on ne fait qu'énumérer les cas sur lesquels on branche. On ne spécifie pas non plus le comportement de retour (retourner la solution non *null* s'il y en a une, ou *null* sinon). Ceci est commun en FPT — ceci permet d'abréger la présentation d'appels de branchement standards.

fonction *3hitsetAmélioré*(S, U, k, X)
  si $k < 0$ **alors**
    return *null*
  fin
  si S *est vide* **alors**
    return X
  fin
  si $S_i \cap S_j = \emptyset$ *pour tout* $S_i, S_j \in S$ *distincts* **alors**
    si $|S| > k$ **alors**
      return *null*
    sinon
      Ajouter à X un élément de chaque S_i
      return X
    fin
  fin
  Soient S_i, S_j distincts tels que $|S_i \cap S_j|$ est maximum
  si $|S_i \cap S_j| = 2$ **alors**
    Soient $S_i = \{x, y, z\}$ et $S_j = \{x, y, a\}$
    Brancher récursivement sur les cas suivants :
    – Ajouter x à X, réduire k de 1
    – Ajouter y à X, réduire k de 1
    – Ajouter z et a à X, réduire k de 2
  sinon si $|S_i \cap S_j| = 1$ **alors**
    Soient $S_i = \{x, y, z\}$ et $S_j = \{x, a, b\}$
    Brancher récursivement sur les cas suivants :
    – Ajouter x à X, réduire k de 1
    – Ajouter y et a à X, réduire k de 2
    – Ajouter y et b à X, réduire k de 2
    – Ajouter z et a à X, réduire k de 2
    – Ajouter z et b à X, réduire k de 2
  fin

Nous n'allons pas nous attarder sur le fait que cet algorithme est correct. Concentrons-nous sur la complexité. On a deux situations possibles et donc deux récurrences. Laquelle on prend ? Comme on le fait toujours en algorithmique : on prend le pire cas ! Un des deux cas de l'algorithme devrait être pire que l'autre. On va donc supposer que c'est toujours ce pire cas qui survient.

Dans le cas où $|S_i \cap S_j| = 2$, on avait

$$t(k) = 2t(k-1) + t(k-2)$$

avec le polynôme caractéristique

$$c^2 - 2c - 1 = 0$$

et en trouvant les zéros, on a $c \simeq 2.4143$.

dans le cas où $|S_i \cap S_j| = 1$, on avait

$$t(k) = t(k-1) + 4t(k-2)$$

avec le polynôme caractéristique

$$c^2 - c - 4 = 0$$

et on a $c \simeq 2.5616$.

On peut donc supposer que dans le pire cas, l'arbre de branchement a $O(2.5616^k)$ noeuds. Chaque appel peut être implémenté en temps $O(m)$, et en arrondissant, la complexité est donc $O(2.57^k \cdot m)$.

9.7 La séquence consensus

Nous terminons ce chapitre avec un exemple où il n,est pas trivial de borner le nombre de cas sur lesquels brancher.

Soit S une séquence de caractères. On écrit $S[i]$ pour le caractère à la position i de S. La distance Hamming $d(S,T)$ entre deux séquences S et T de même longueur est définie comme le nombre de cararactères différents par position. C'est-à-dire,

$$d(S,T) = |\{i : S[i] \neq T[i]\}|$$

CONSENSUS-SEQUENCE
Entrée : séquences de caractères $S_1, S_2, \ldots, S_n$, chacune de longueur ℓ
Paramètre : distance d
Sortie : une séquence S telle que $d(S, S_i) \leq d$ pour toute S_i, ou *null* si non-existant.

Avant de penser à un algorithme FPT, il faut analyser le problème et en dériver quelques observations. On note que si toutes les séquences ont le même caractère c à la position p, alors la séquence consensus aura le caractère c à la position p. On peut donc ignorer cette position et supposer que pour chaque position p, il existe S_i, S_j tels que $S_i[p] \neq S_j[p]$.

Une autre observation est que si on a des S_i et S_j trop distantes, il ne peut pas y avoir de solution. Une façon de le voir est de constater que d satisfait l'inégalité triangulaire, et qu'une séquence consensus doit être "entre les deux" séquences.

Lemme 14. *S'il existe S_i, S_j tels que $d(S_i, S_j) > 2d$, alors il n'existe pas de séquence consensus à distance d de S_i et S_j.*

Démonstration. Supposons qu'il existe une séquence S telle que $d(S, S_i) \leq d$ et $d(S, S_j) \leq d$. Soient $P = \{p_1, \ldots, p_{2d+1}\}$ un ensemble de $2d + 1$ positions sur lesquelles S_i et S_j diffèrent. Puisque S diffère de S_i à d position ou moins, il doit y avoir au moins $d + 1$ position de P où S et S_i sont identiques. Ceci veut dire que S diffère de S_j à ces $d + 1$ position, une contradiction. □

La stratégie de branchement sera la suivante. On va démarrer avec la séquence S_1 (ce choix est arbitraire). Par le lemme ci-haut, on sait que $d(S_1, S_j) < 2d$, et donc que chaque S_j diffère de S_1 à au plus $2d$ positions. Si $d(S_1, S_j) \leq d$ pour tout S_j, on a terminé. Sinon, prenons S_j tel que $d(S_1, S_j) > d$. Soient P les positions où S_1 et S_j diffèrent. S'il existe un séquence consensus S, il faut qu'au moins une des positions $i \in P$ soit telle que $S[i] = S_j[i]$. On sait qu'il y a au plus $2d$ telles positions, donc on branche sur chacune d'entre elles. Ceci résulte en $2d$ branchements, chacune donnant une séquence S' dont une position diffère de S_1 à 1 endroit. On répète avec S', mais puisque chaque modification nous éloigne de S_1, on ne peut pas répéter plus de d fois. Si on atteint un point où $d(S', S_j) > 2d$ pour un certain S_j, on sait qu'il est impossible d'appliquer d modifications à S' pour atteindre une séquence consensus à distance au plus d de S_j. Dans ce cas, on peut sortir.

Le pseudo-code qui décrit cette procédure se trouve ci-bas. On conserve le d original, mais on retient le nombre de modifications encore permises avec $d\_restant$. On maintient aussi la séquence consensus candidate courante S'. Initialiement, $d\_restant = d$ et $S' = S_1$.

```
fonction seqConsensus(S_1, ..., S_n, d, d_restant, S')
    si d_restant < 0 alors
        return null
    fin
    si d(S', S_j) > 2d pour un certain S_j alors
        return null
    fin
    si d(S', S_j) ≤ d pour tout S_j alors
        return S'
    fin
    Soit S_j tel que d(S', S_j) > d
    Soient P les positions où S' et S_j diffèrent
    pour chaque i ∈ P faire
        S'' = S'
        S''[i] = S_j[i]
        S* = seqConsensus(S_1, ..., S_n, d, d_restant − 1, S'')
        si S* ≠ null alors
            return S*
        fin
    fin
    return null
```

Il n'est pas trivial de garantir que cet algorithme fonctionne, c'est-à-dire qu'il retourne toujours une solution s'il en existe une, et *null* sinon. Les idées principales ont été énoncées ci-haut, et nous vous laissons le soin de démontrer ce fait. La complexité peut être analysée comme suit. On branche sur $|P|$ cas possibles, et on sait que $|P| \leq 2d$. La profondeur de l'arbre est bornée par d, et donc l'arbre de récursion a $O((2d)^d)$ noeuds. Chaque appel prend un temps $O(n^2)$, la majorité du temps étant dans les calculs de distances avec S'. La complexité est donc $O((2d)^d \cdot n^2)$.

Chapitre 10

Kernelisation

La kernelisation, qu'on pourrait traduire par la réduction en un noyau, vise à transformer une instance donnée en une autre instance équivalente, mais plus petite. La taille de l'instance équivalente doit en fait être $O(f(k))$ pour une fonction f arbitraire. Par exemple, supposons qu'on nous donne une instance $G = (V, E)$ de VERTEX-COVER avec n sommets. Disons qu'on arrive à transformer G en un graphe G' tel que :

— $|V(G')| \leq 2k$;
— G a un VERTEX-COVER de taille k si et seulement si G' a un VERTEX-COVER de taille k.

On peut donc travailler sur le graphe G', car l'existence d'une solution sur G' est équivalente à l'existence d'une solution sur G'. De plus, puisque $|V(G')| \leq 2k$, on peut faire une force brute simple sur tous les sous-ensembles de $V(G')$, ce qui énumérerait $O(2^{2k})$ sous-ensembles. En supposant que la transormation prend un temps polynomial, on a donc un algorithme FPT.

Si un tel G' existe, on dira qu'il forme un *noyau* (ou *kernel* en anglais). L'important est qu'un noyau soit équivalent à l'instance originale, et qu'il soit de taille bornée par $f(k)$. Une fois qu'un noyau est trouvé, on sait que notre problème est FPT car une force brute sur le noyau prendra un FPT.

En pratique, il est fréquent de combiner les techniques pour accélérer les algorithmes. On transforme d'abord notre instance en un noyau, puis on exécute par exemple un algorithme de branchement, mais sur le noyau. Lorsque c'est possible, il en résulte des algorithmes beaucoup plus efficaces.

10.1 Définition d'un noyau

Soit P un problème algorihtmique et soit (I, k) une instance paramétrée de P. On dit qu'une autre instance paramétrée (I', k') de P est un *noyau* de (I, k) si les conditions suivantes sont satisfaites :

1. $|I'| \in O(f(k))$ pour une fonction f, où $|I'|$ est la taille de l'instance I' ;
2. $k' \in O(g(k))$ pour une fonction g ;
3. (I, k) admet une solution si et seulement si (I', k') admet une solution ;
4. il est possible de transformer (I, k) en (I', k') en temps $O(|I|^c)$, où c est une constante.

En mots, on veut transformer I en une instance I' de taille bornée par une fonction k, permettant ainsi la force brute sur I' en temps FPT. Notez que le temps requis pour effectuer la transformation ne doit pas être exponentiel en k. L'esprit de la kernelisation est de fournir une procédure de pré-traitement qui peut être suivie d'un algorithme FPT. Avoir un temps exponentiel en k pour créer une instance équivalente donnerait tout de même un algorithme FPT, mais l'instance ne serait pas considérée comme un noyau.

Notez aussi qu'on se permet de modifier le paramètre k' associé à I', dans la mesure où k' est aussi borné par une fonction de k. Ceci peut parfois s'avérer nécessaire lorsque des modifications significatives sont apportées. Finalement, on demande que (I', k') soit équivalente à (I, k). Un fait contre-intuitif est que les solutions (I, k) et (I', k') peuvent n'avoir rien à voir entre elles. La kernelisation permet donc de décider si (I, k) admet une solution ou non, mais une solution pour (I', k') n'est peut-être pas une solution pour (I, k). Il faut parfois appliquer une transformation inverse à la solution de (I', k'). Par contre, ce n'est pas requis, car seulement l'équivalence entre l'existence de solutions est requise.

Comment trouver un noyau ? Avec des règles de réduction !

La construction du noyau dépend du paramètre k. La plupart du temps, on prend notre instance I et on démarre un argument du style

"s'il est vrai qu'il y a une solution de taille k,
alors on ne peut pas avoir plus de [...] dans notre instance"

ou encore

> "s'il est vrai qu'il y a une solution de taille k,
> alors on peut éliminer [...] de notre instance"

et il ne reste qu'à remplir les [...]. Une autre façon de présenter ces arguments est aussi de contraposer :

> "si on a plus de [...] dans notre instance,
> alors il n'y a pas de solution de taille k et on peut retourner *null*"

L'idée générale est donc de supposer qu'il existe une solution de taille k, et d'identifier les propriétés importantes de notre instance. Ces propriétés devraient ensuite nous permettre de réduire notre instance. Ceci prend souvent la forme d'un ensemble de **règles de réduction**, qui sont des énoncés du style "si I a X propriété, alors supprimer Y". Bien sûr, il faut montrer que nos règles de réduction sont valides, i.e. qu,elles préservent l'équivalence.

Tout ceci peut paraître très abstrait, alors passons à des exemples. Comme d'habitude, nous utiliserons VERTEX-COVER pour commencer.

10.2 Kernelisation de VERTEX-COVER

Soit $G = (V, E)$ une instance de VERTEX-COVER avec paramète k, la taille de la solution. On commence par se demander si certaines opérations sont forcées lorsqu'on suppose qu'il existe une couverture de k sommets.

En y réfléchissant assez longtemps, on constate l'observation suivante.

Observation 1. *S'il existe $u \in V$ tel que $|N(u)| > k$, alors tout VERTEX-COVER de taille k ou moins contient u.*

Démonstration. Si une solution X ne contient pas u, alors il faut inclure tous les sommets de $N(u)$ pour couvrir les arêtes incidentes à u. Il est alors impossible que $|X| \leq k$ car $|N(u)| > k$ et $N(u) \subseteq X$. □

On peut en déduire une règle bien simple, qui est complètement déterministe et qui évite de brancher sur plusieurs cas.

Règle 1. S'il existe $u \in V$ tel que $|N(u)| > k$, suprimer u et ses arêtes de G et réduire k de 1.

Cette règle réduit le nombre de sommets de G et réduit aussi k. Il faut bien sûr argumenter qu'en applicant la règle, on obtient une instance équivalente. Lorsque c'est le cas, on dit que la règle est *saine* ("safe" en anglais).

Lemme 15. *La règle 1 est saine.*

Démonstration. Soit G le graphe et soit G' le graphe obtenu après avoir appliqué la règle 1 par la suppression de u. Il faut montrer que G a un VERTEX-COVER de taille k si et seulement si G' a un VERTEX-COVER de taille $k-1$.

Si G a un VERTEX-COVER X de taille k, alors selon l'observation ci-haut, on sait que $u \in X$. Donc, $X \setminus \{u\}$ doit couvrir toutes les arêtes non-incidentes à u, et donc $X \setminus \{u\}$ est un VERTEX-COVER de taille $k-1$ de G'.

Dans l'autre sens, soit X' un VERTEX-COVER de G' de taille $k-1$. Toutes les arêtes de G sont couvertes par X', sauf peut-être certaines incidentes à u. En ajoutant u à X', on obtient un VERTEX-COVER de taille k ou moins. □

Pour créer notre noyau, on va donc appliquer la règle 1 jusqu'à ce que ça ne soit plus possible. Ceci résultatera en un graphe G' dans lequel chaque sommet u satisfait $|N(u)| \leq k'$, où k' est le paramètre résultant après l'application des règles. Pour simplifier, on va supposer que notre graphe s'appelle G et que son paramètre est k, et que la règle 1 n'est plus applicable.

Sachant que chaque somemt $u \in V(G)$ a au plus k voisins, on observe que si on inclut u dans X, alors u peut couvrir seulement k arêtes. Ceci mène à une deuxième observation.

Observation 2. *Supposons que la règle 1 ne s'applique pas sur G. Alors si $|E(G)| > k^2$, il n'y a pas de solution de taille k.*

Démonstration. Si la règle 1 ne s'applique pas, alors chaque u dans une solution X couvre au plus k arêtes. mais si $|E(G)| > k^2$, k sommets ne suffiront pas pour couvrir toutes les arêtes, car ensemble ils couvrent un maximum de k^2 arêtes. □

On peut donc supposer que $|E(G)| \leq k^2$. Qu'en est-il du nombre de sommets ? Il n'y a pas de borne, car il est possible que G contienne beaucoup

de sommets isolés. Il n'est pas difficile de voir qu'on peut dériver une autre règle saine.

Règle 2. Si G contient $u \in V(G)$ tel que $|N(u)| = 0$, alors retirer u de G.

Sachant que G a au plus k^2 arêtes et aucun sommet isolé, on peut borner le nombre de sommets. On peut se convaincre que le nombre maxium de sommets avec k^2 arêtes est atteint si chaque sommet n'a qu'un seul voisin. Ceci se produit lorsque le graphe est en fait un matching, dans quel cas il a $2k^2$ sommets. On en déduit le résultat suivant.

Théorème 24. *VERTEX-COVER admet un noyau avec au plus $2k^2$ sommets et au plus k^2 arêtes.*

L'algorithme qui produit le noyau est le suivant.

```
fonction vcKernel(G = (V, E), k, X)
    si G a un sommet v de degré 0 alors
        return vckernel(G − v, k, X)
    sinon si G a un sommet u tel que |N(u)| > k alors
        return vcKernel(G − u, k − 1, X ∪ {u})
    sinon si |E| > k² alors
        return null
    sinon
        return G
    fin
```

Notez que l'ensemble X n'est pas utilisé. Il contient toutefois tous les sommets qui doivent faire partie d'une couverture et qui ne sont plus dans le noyau de G. L'utilité de X est de reconstruire une solution concrète. Une fois qu'une solution X' pour le noyau a été trouvée, on sait que $X' \cup X$ est une solution pour le G original. Le X est donc optionnel.

10.3 Un noyau trivial pour MAX-3-SAT

Rappelez-vous MAX-3-SAT, où on cherche à maximiser un nombre de clauses satisfaites avec des clauses de taille 3.

MAX-3-SAT
Entrée : clauses $C_1, \ldots, C_m$ avec trois variables chacune, sur variables $x_1, \ldots, x_n$
Paramètre : k, le nombre de clauses satisfaites
Sortie : une assignation des x_i telle que au moins k clauses sont satisfaites, ou *null* si non-existant

On se rappelle de l'algorithme probabiliste pour MAX-3-SAT qui retournait une assignation qui, en espérance, satisfaisait au moins $7m/8$ clauses. Ceci veut dire qu'il *existe* une assignation telle que $7m/8$ clauses sont satisfaites, et notre procédure de dérandomisation en retrouvait une de façon déterministe.

On peut utiliser ceci pour la version paramétrée de MAX-3-SAT. Si $k \leq 7m/8$, on sait automatiquement qu'il existe une assignation et on sait comment en trouver une. Si $k > 7m/8$, alors $m < 8k/7$ et le nombre de clauses est borné par une fonction de k. Il reste à nous assurer que le nombre de variables est aussi borné. Ceci est simple : puisque chaque clause a 3 litéraux, on sait que le nombre de variables est au plus $3m < 24k/7$ (on peut éliminer les variables inutiles). On déduit donc un noyau presque trivial à partir de l'algorithme suivant :

fonction *max3satKernel*$(C_1, \ldots, C_m, k)$
  si $k \leq 7m/8$ **alors**
    Retourner le résultat de la dérandomisation pour MAX-3-SAT
  sinon
    Éliminer les variables qui n'apparaissent dans aucune clause
    return $C_1, \ldots, C_m$
  fin

La discussion précédant l'algorithme nous donne :

Théorème 25. *L'algorithme max3satKernel donne un noyau pour MAX-3-SAT avec au plus* $8k/7$ *clauses et au plus* $24k/7$ *variables.*

10.4 Un noyau pour MAX-SAT

Considérons maintenant la version générale de MAX-SAT, où les nombre de variables par clause est quelconque. On sait qu'en choisissant une assignation aléatoire, chaque clause est satisfaite avec probabilité au moins $1/2$ (le pire cas étant une clause avec une seule variable). On peut donc concevoir un algorithme probabiliste qui retourne une assignation qui satisfait $m/2$ clauses en espérance. Cet algorithme peut être dérandomisé pour retourner une telle assignation de façon déterministe.

On peut donc supposer que si $k \leq m/2$, on retourne toujours une assignation satisfaisant au moins k clauses. On peut exprimer ceci avec une règle.

Règle 1. Si $k \leq m/2$, retourner une assignation qui satisfait $m/2$ clauses (avec un algorithme de dérandomisation, par exemple).

Supposons maintenant que $k > m/2$. On a donc $m < 2k$. Il faut maintenant borner le nombre de variables.

Si chaque clause avait disons k variables, on pourrait borner le nombre de variables utiles par $2k \cdot k$. Par contre, il se peut que certaines clauses aient beaucoup de variables. Soient C^p les clauses avec k variables ou moins, et C^g les clauses avec plus de k variables (p pour petit et g pour gros). On aimerait bien nous débarasser de C^g.

Si $|C^g| > k$, il n'est pas difficile de voir qu'on peut satisfaire k clauses de C^g. Il suffit de choisir k clauses de C^g et de choisir une variable différente pour chaque $C_i \in C^g$ (ceci est possible car elles ont toutes plus de k variables). On assigne chaque variable choisie de façon à satisfaire sa clause correspondante, et voilà !

Règle 2. Si $|C^g| > k$, retourner une assignation qui satisfait chaque clause de C^g (par exemple en choisissant une variable par clause).

On peut donc supposer que $|C^g| < k$. Notons qu'il est possible de satisfaire toutes les $|C^g|$ clauses de C^g avec $|C^g|$ variables (comme plus haut, on choisit une variable par clause). Pourrait-on simplement éliminer les "grosses" clauses en supposant qu'on va les satisfaire avec $|C^g|$ variables ? Il s'avère que oui.

Règle 3. Retirer les clauses C^g et réduire k de $|C^g|$

L'idée de la règle 3 est que si on peut satisfaire $k - |C^g|$ "petites" clauses,

on a besoin d'au plus $k-|C^g|$ variables pour ce faire. Les $|C^g|$ autres variables peuvent être utilisées pour satisfaire C^g. Cette règle est non-triviale, et il faut tout de même démontrer qu'elle fonctionne.

Lemme 16. *Si les règles 1 et 2 ont été appliquées, alors la règle 3 est saine. C'est-à-dire, on peut satisfaire k clauses de $C_1, \ldots, C_m$ si et seulement si on peut satisfaire $k - |C^g|$ clauses de C^p.*

Démonstration. ($\Rightarrow$) supposons qu'on peut satisfaire k clauses parmi $C_1, \ldots, C_m$. Il faut qu'il y ait $k - |C^g|$ clauses parmi C^p qui sont satisfaites.

($\Leftarrow$) supposons qu'on peut satisfaire $l = k - |C^g|$ clauses de C^p. Soient $C_1, \ldots, C_l$ ces clauses satisfaites. On note qu'il est possible de les satisfaire en assignant seulement l variables (car une seule variable suffit pour satisfaire une clause). Il est possible de retrouver l telles variables si on a déjà une assignation, mais ce n'est pas nécessaire pour les fins du lemme. Si on connaît ces $l = k - |C^g|$ variables servant à satisfaire $C_1, \ldots, C_l$, on peut ensuite utiliser $|C^g|$ variables additionnelles pour satisfaire toutes les clauses de C^g. □

On en déduit un noyau de taille quadratique.

Théorème 26. *Les règles 1,2 et 3 mènent à un noyau pour MAX-SAT avec $O(k)$ clauses et $O(k^2)$ variables.*

Démonstration. On suppose que $k > m/2$, sinon on retourne le résultat d'un algorithme d'approximation (règle 1). Donc $m < 2k \in O(k)$. Ensuite, si la règle 2 s'applique, on résout le problème trivialement. Sinon, on applique la règle 3. On a ainsi un ensemble de clauses C^p de taille au plus $2k$, et chaque clause a au plus k variables. le nombre de variables impliquées au total est donc $O(k^2)$. □

10.5 Un noyau pour EDGE-CLIQUE-COVER

Nous verrons que les noyaux peuvent parfois avoir une taille exponentielle. Dans le problème EDGE-CLIQUE-COVER, on veut couvrir totues les arêtes avec des cliques.

EDGE-CLIQUE-COVER
Entrée : un graphe $G = (V, E)$
Paramètre : k, le nombre de cliques
Sortie : un ensemble de k cliques $C_1, \ldots, C_k$ telles que pour tout $uv \in E$, il existe C_i telle que $u \in C_i$ et $v \in C_i$

Notez qu'il pourrait y avoir moins de k cliques qui couvrent chaque arête. Dans ce cas, on peut retourner exactement k cliques en répétant une même clique plusieurs fois.

Commençons par une règle bien simple qui est facilement démontrable comme saine.

Règle 1. S'il existe $u \in V(G)$ tel que $|N(u)| = 0$, alors supprimer u de G.

Une autre règle simple est que si une composante connexe est une clique, on ne peut pas faire mieux que d'inclure cette clique pour couvrir ses arêtes.

Règle 2. S'il existe une composante connexe C de G telle que C est une clique, alors inclure C dans notre solution et décrémenter k de 1.

Deux sommets u et v sont des *jumeaux* si $N(u) \cup \{u\} = N(v) \cup \{v\}$. En particulier, des jumeaux doivent être voisins. À toutes fins pratiques, des jumeaux sont "identiques" et nous pouvons n'en garder qu'un seul.

Règle 3. Si la règle 2 ne s'applique pas à G et que G contient deux jumeaux u et v, alors supprimer u de G.

Pour voir que cette règle est saine, on observe qu'on peut toujours mettre u et v dans les mêmes cliques. La preuve formelle est laissée en exercice. **Tel qu'observé en cours**, on note qu'il est nécessaire d'appliquer la règle 2 avant la règle 3. Sinon, avec une composante connexe qui est une clique, on éliminerait tous ses jumeaux et on se retrouverait avec un seul sommet. Ce dernier serait éliminé et on n'aurait jamais compté la clique.

De façon peut-être étonnante, c'est tout ce qu'il faut pour avoir un noyau.

Théorème 27. *Soit G' le graphe obtenu de G après application des règles 1, 2 et 3 jusqu'à ce que ça ne soit plus possible. Supposons que G' admet une couverture par cliques de taille au plus k. Alors G' a au plus 2^k sommets.*

Démonstration. Soit $C_1, \ldots, C_k$ un ensemble de k cliques qui couvrent chaque arête de G'. Soit $u \in V(G')$. On associe à u un vecteur b_u de k bits, où le

bit i de b_u est 1 si $u \in C_i$, et 0 sinon. On note qu'il y a 2^k vecteurs de bits possibles. On veut montrer que tous les sommets ont un vecteur de bits différent. Soient $u, v \in V(G')$ deux sommets distincts et supposons que $b_u = b_v$. Si b_u (ou b_v) n'a que des 0, alors u n'a aucun voisin et la règle 1 aurait du être appliquée. Sinon, le fait que $b_u = b_v$ implique que u et v sont voisins car ils sont dans une même clique. Aussi, tout voisin de u appraît dans une des cliques et de même pour v. De plus, ces voisins apparaissent dans les mêmes cliques. Il s'ensuit que u et v sont jumeaux et un des deux aurait du être éliminé par la règle 3, une contradiction.

On déduit que chaque sommet de G' a un vecteur de bits différent, et donc il y a au plus 2^k sommets. □

Un corollaire du théorème précédent est que s'il y a plus de 2^k sommets dans $V(G')$, on peut tout de suite retourner *null*. Concrètement, l'algorithme de kernelisation est le suivant.

```
fonction eccKernel(G = (V, E), k)
    si G a un sommet u de degré 0 alors
        return eccKernel(G − u, k)
    sinon si G a une composante connexe C qui est une clique alors
        return eccKernel(G − C, k − 1)
    sinon si G a des jumeaux u et v distincts alors
        return eccKernel(G − u, k)
    sinon si |V| > 2^k alors
        return null
    sinon
        return G
    fin
```

Notez que sous certaines hypothèses de complexité (la Strong Exponential Time Hypothesis, pour les connaisseurs), il a été démontré qu'il était impossible d'obtenir un noyau plus petit que 2^k. Donc il y a de fortes raisons de croire que certains noyaux ne peuvent pas être de taille polynomiale.

10.6 Un noyau pour VERTEX-COVER basé sur les LP

Revenons à notre problème favori, VERTEX-COVER. Nous avons obtenu un noyau de taille $O(k^2)$, mais il s'avère qu'il est possible d'en avoir un de taille $2k$ en utilisant le LP. Rappelons le LP de VERTEX-COVER

Minimiser

$$\sum_{v_i \in V} x_i$$

Sujet à

$$\begin{aligned} x_i + x_j &\geq 1 && \text{pour chaque } v_i v_j \in E \\ 0 \leq x_i &\leq 1 && \text{pour chaque } v_i \in V \end{aligned}$$

Supposons que nous avons résolu ce LP et avons obtenu une solution $\boldsymbol{x}^* = \{x_1^*, \ldots, x_n^*\}$. Puisque le LP donne une borne inférieure sur la taille de la solution, on peut immédiatement utiliser la règle suivante.

Règle 1. Si $\sum_{v_i \in V} x_i^* > k$, retourner *null*.

Pour aller un peu plus loin, on peut séparer nos sommets en trois ensembles :

$$\begin{aligned} V_0 &= \{v_i \in V : x_i^* < 1/2\} \\ V_{\frac{1}{2}} &= \{v_i \in V : x_i^* = 1/2\} \\ V_1 &= \{v_i \in V : x_i^* > 1/2\} \end{aligned}$$

Intuitivement, les sommets de V_1 ont le plus de poids et il est raisonnable de croire qu'ils devraient être dans une couverture optimale. Similairement, V_0 ont un petit poids et on ne devrait pas être obligé de les inclure. Il s'avère que cette intuition est vraie.

Lemme 17. *Il existe un ensemble couvrant $X \subseteq V$ de taille minimum tel que $V_1 \subseteq X \subseteq V_{\frac{1}{2}} \cup V_1$.*

En mots, le lemme dit que nous n'avons pas besoin des sommets de V_0 dans notre solution. De plus, on peut supposer que tous les sommets de V_1 sont dans la solution. Nous n'allons pas prouver ce lemme - une preuve se trouve dans le livre de Cygan & al. On peut toutefois en extraire une règle bien simple.

Règle 2. Retirer V_0 de G, ajouter V_1 à la solution, et réduire k de $|V_1|$.

Lemme 18. *La règle 2 est saine.*

Démonstration. Soit G' le graphe obtenu après avoir appliqué la règle 2. Il faut montrer que G a un ensemble couvrant de taille k si et seulement si G' a un ensemble couvrant de taille $k - |V_1|$.

($\Rightarrow$) Soit X un ensemble couvrant de G de taille k. Par le lemme 17, on peut supposer que $V_1 \subseteq X$. Il doit donc y avoir $k - |V_1|$ sommets dans X pour couvrir les arêtes restantes dans G'.

($\Leftarrow$) Soit X' un ensemble couvrant de G' de taille $k - |V_1|$. Dans G, il ne reste que les arêtes incidentes à V_0 et V_1 à couvrir. Soit $X = X' \cup V_1$. Alors les arêtes incidentes à V_1 sont couvertes. Soit $v_i \in V_0$ et $v_i v_j \in E$. Puisque $v_i \in V_0$, on a $x_i^* < 1/2$ et il faut que $x_j^* > 1/2$ pour satisfaire la contrainte du LP. Donc $v_j \in V_1$ et X couvre $v_i v_j$. □

Avec la bonne analyse, ceci mène directement à un noyau.

Théorème 28. *Soit G' le graphe obtenu après application des règles 1 et 2. Alors G' a au plus $2k$ sommets.*

Démonstration. Puisque la règle 2 ne s'applique pas, $x_i^* = 1/2$ pour tout $v_i \in V(G')$. On a

$$|V(G')| = |V_{\frac{1}{2}}| = \sum_{v_i \in V_{\frac{1}{2}}} 2x_i^* \leq 2 \sum_{v_i \in V(G)} x_i^* \leq 2k$$

où nous avons utilisé la règle 1 pour déduire que $\sum_{v_i \in V(G)} x_i^* \leq k$. □

10.7 Est-ce que tous les problèmes FPT ont un noyau

Il est possible de démontrer qu'un problème est dans FPT si et seulement si il admet un noyau. Pour certains, on aurait même pu définir FPT par l'existence d'un noyau, et l'aspect de complexité paramétrée n,aurait été qu'une conséquence.

Théorème 29. *Un problème P est dans FPT si et seulement si P admet un noyau.*

Démonstration. ($\Leftarrow$) Commençons par la direction facile. Soit (I, k) une instance de P et supposons qu'on peut trouver un noyau de taille $O(f(k))$ avec paramètre $k' \in O(g(k))$ en temps $O(|I|^c)$. Une force brute sur le noyau donnera un algorithme $O(h(k', f(k)))$ pour une certaine fonction h. Peu importe h, on aura $h \in O(h_2(k))$ pour une certaine fonction h_2. L'algorithme qui créé le noyau et exécute la force brute prend un temps $O(h_2(k) + |I|^c)$, ce qui est FPT.

($\Rightarrow$) Supposons que P est FPT et qu'on peut résoudre toute instance (I, k) en temps $O(f(k)|I|^c)$ avec un certain algorithme A. On construit un noyau avec un algorithme alternatif B. Au départ, B exécute A pendant $d|I|^{c+1}$ instructions, où d est la constante caché dans le O de $O(f(k)|I|^c)$, et regarde si A a terminé. Si oui, alors B retourne le résultat de A, ce qui a pris à B un temps polynomial (donc pas besoin de noyau).

Si A n'a pas terminé, alors le temps requis par A, qui est $df(k)|I|^c$, est plus grand que $d|I|^{c+1}$. Donc,

$$df(k)|I|^c > d|I|^{c+1}$$

ce qui implique que

$$f(k) > |I|$$

Puisque $|I| < f(k)$, la taille de notre instance est bornée en fonction de k et est donc un noyau. □

Notez que la construction d'un noyau dans la preuve ci-haut n'est pas très utile en pratique. Elle ne fait qu'exécuter un algorithme FPT connu, donc à quoi sert le noyau de cet algorithme si ce dernier peut déjà résoudre l'instance ? La vraie utilité des noyaux est habituellement de permettre une réduction simple et rapide d'une instance, pour ensuite appliquer un algorithme FPT plus sophistiqué.

Chapitre 11

Décomposition en arbre et *treewidth*

La *treewidth*, que l'on pourrait traduire par la *largeur d'arbre*, est un paramètre sur les graphes déterminant à quel point le graphe est "proche" d'être un arbre. Les arbres admettent des algorithmes en temps polynomial sur une vaste étendue de problèmes. En généralisant, on peut s'attendre à ce qu'un graphe proche d'un arbre admette des algorithmes proches d'être polynomiaux. Ces algorithmes utilisent, pour la plupart, une approche de programmation dynamique. Nous débuterons avec quelques exemples sur les arbres, puis généraliserons à des graphes avec treewidth bornée.

11.1 Programmation dynamique sur les arbres

Soit $T = (V, E)$ un arbre enraciné. Rappelons qu'un arbre est un graphe connexe et sans cycle. Une définition équivalente d'un arbre est un graphe connexe avec exactement $n - 1$ arêtes. Nous dénoterons la racine par $r(T)$. Pour $v \in V$, le sous-arbre enraciné en v est dénoté par $T[v]$.

La programmation dynamique sur un arbre procède habituellement comme suit. On doit optimiser un certain critère, et on détermine un certain ensemble d'informations $I(v)$ à calculer à chaque noeud $v \in V$, où $I(v)$ nous permet de calculer une solution optimale sur T s'il est connu à chaque sommet (ou du moins à la racine).

Les informations $I(v)$ doivent pouvoir se calculer à partir des $I(w_1), \ldots, I(w_k)$ des enfants $w_1, \ldots, w_k$ de v. Souvent, $I(v)$ est exprimé par une fonction de récurrence qui dépend des $I(w_i)$.

La forme générale d'une programmation dynamique sur un arbre va donc

comme suit :
– pour chaque noeud $v \in V$ dans un parcours post-ordre
– – soient $w_1, \ldots, w_k$ les enfants de v
– – calculer $I(v)$ en utilisant les valeurs $I(w_1), \ldots, I(w_k)$

La plupart du temps, calculer $I(v)$ à une feuille v se fait de façon évident. C'est aux noeuds internes qu'il y a parfois complication. Plus concrètement, on a habituellement une procédure récursive :

```
fonction progDynArbre(T = (V, E), v)
    //v est le noeud courant
    si v est une feuille alors
        Calculer I(v) de façon triviale
    sinon
        pour chaque w_i enfant de v faire
            progDynArbre(T, w_i)
        fin
        Calculer I(v) (sachant les I(w_i))
    fin
```

L'appel initial se fait avec $v = r(T)$, la racine. Il faut aussi spécifier comment la solution optimale se calcule à aprtir des $I(v)$ (ce qu'on généralement faire en dehors de l'algorithme).

Voyons quelques exemples.

11.1.1 Ensemble indépendant dans un arbre

Rappelons qu'un ensemble $X \subseteq V$ est indépendant si $uv \notin E$ pour tout $u, v \in X$. On cherche l'ensemble indépendant de taille maximum. Ceci est bien sûr NP-complet, mais pas sur les arbres.

Supposons qu'on reçoit un arbre enraciné $T = (V, E)$. Pour chaque $v \in V$, on va calculer les informations suivantes :

— $M_0[v]$: la taille d'un ensemble indépendant maximum X de $T[v]$, avec la contrainte que v n'est *pas* dans X ;

— $M_1[v]$: la taille d'un ensemble indépendant maximum X de $T[v]$, avec la contrainte que v *doit être* dans X.

Supposons qu'on connaît $M_0[v]$ et $M_1[v]$ pour chaque $v \in V$. Alors la solution optimale sur l'arbre en entier est donnée par $\max(M_0[r(T)], M_1[r(T)])$. Ceci est parce qu'il y a deux possibilités pour l'ensemble indépendant sur $T = T[r(T)]$: soit la racine est dans l'ensemble ou non, et on connaît l'optimal dans les deux cas.

Mais comment calculer $M_0[v]$ et $M_1[v]$? Si v est une feuille, ceci est facile. On a

$$M_0[v] = 0 \qquad M_1[v] = 1$$

qui correspondent à ne pas inclure ou inclure v.

Si v est un noeud interne, si on décide de ne pas inclure v, alors on peut prendre les solutions optimales des sous-arbres enfants et prendre l'union. On aura toujours un ensemble indépendant car on ne va pas inclure deux sommets qui partagent une arête avec l'union. L'optimal du sous-arbre $T[w_i]$ est donné par $\max(M_0[w_i], M_1[w_i])$. On a donc

$$M_0[v] = \sum_{w_i \in enfant(v)} \max(M_0[w_i], M_1[w_i])$$

Si on décide d'inclure v, alors on ne peut *pas* inclure un enfant de v, car sinon on n'aurait pas un ensemble indépendant. On doit donc compter 1 pour l'ajout de v, plus l'optimal des sous-arbres enfants qui n'incluent pas d'enfant. On a donc

$$M_1[v] = 1 + \sum_{w_i \in enfant(v)} M_0[w_i]$$

L'algorithme complet est donc le suivant.

```
fonction maxIndSet(T = (V, E), v)
    //v est le noeud courant
    si v est une feuille alors
        M_0[v] = 0
        M_1[v] = 1
    sinon
        M_0[v] = 0
        M_0[v] = 1
        pour chaque w_i enfant de v faire
            maxIndSet(T, w_i)
            M_0[v]+ = max(M_0[w_i], M_1[w_i])
            M_1[v]+ = M_0[w_i]
        fin
    fin
```

L'appel initial est fait avec $v = r(T)$, et on retourne $\max(M_0[r(T)], M_1[r(T)])$.

Notez que le "si" n'était pas nécessaire, car M_0 et M_1 sont initialisées de la même manière qu'on ait une feuille ou non.

11.1.2 VERTEX-COVER sur un arbre

Le complément d'un ensemble indépendant maximal est un VERTEX-COVER, donc un algorithme simple pour retourner la taille minimum d'une couverture par sommet des arêtes est d'exécuter l'algorithme ci-haut, obtenir une valeur k, puis reoturner $n - k$. On peut aussi formuler une programmation dynamique en utilisant les mêmes principes.

Pour $v \in V$, on écrit $M_0[v]$ pour la taille d'un ensemble couvrant minimum X tel que $v \notin X$, et $M_1[v]$ lorsque $v \in X$. Pour v une feuille, on a encore $M_0[v] = 0$ et $M_1[v] = 1$.

Si v est un noeud interne, si on n'inclut pas v, il faut inclure chaque enfant de v pour couvrir les arêtes. Donc

$$M_0[v] = \sum_{w_i \in enfant(v)} M_1[w_i]$$

Si on inclut v, on fait ce qu'on veut avec les enfants, et donc

$$M_1[v] = 1 + \sum_{w_i \in enfant(v)} \min(M_0[w_i], M_1[w_i])$$

(on prend le min car ici on minimise). La valeur recherchée est $\min(M_0[r(T)], M_1[r(T)])$.

L'algorithme est presque identique à celui ci-haut. En fait, ce type d'algorithme est souvent donné seulement par les récurrences, car elles peuvent être traduites directement en algorithme de façon standard. Pour la suite, nous allons donc donner les récurrences seulement. Voyons un dernier exemple.

11.1.3 Assignation de caractères dans une phylogénie

On a un arbre $T = (V, E)$ dans lequel chaque feuille l est assignée à un caractère $c(l)$. Le caractère fait partie d'un alphabet Σ. On a une fonction de coût de transformation $f(a, b)$ qui représente le coût de transformer le caractère a en b, où $a, b \in \Sigma$. On suppose que $f(a, b) = f(b, a)$. Le but est d'assigner à chaque noeud interne un caractère de façon à minimiser la somme des coûts sur chaque arête. Dit autrement, étant donné $c(l)$ aux feuilles, on cherche à assigner $c(v)$ pour chaque noeud interne, de façon à minimiser $\sum_{uv \in E} f(u, v)$. Ceci est utile en bioinformatique pour reconstruire l'évolution d'espèces ancestrales (les noeuds internes) lorsqu'on connaît seulement les espèces d'aujourd'hui (les feuilles).

Pour chaque noeud $v \in V$ et chaque caractère $a \in \Sigma$, on calcule $M_a[v]$ qui est le coût minimum possible dans $T[v]$ sachant que $c(v) = a$.

Si v est une feuille, on a

$$M_a[v] = \begin{cases} 1 \text{ si } a = c(v) \\ \infty \text{ si } a \neq c(v) \end{cases}$$

Sinon, si v est un noeud interne, on va prendre le minimum dans les sous-arbres enfants en ajoutant le coût des arêtes. L'observation principales est qu'on n'a pas besoin de tester chaque combinaison possible chez les enfants. On peut prendre le minimum chez chaque enfant indépendamment. Bref, on a

$$M_a[v] = \sum_{w_i \in enfant(v)} \min_{b \in \Sigma} (f(a, b) + M_b[w_i])$$

Au final, on retourne $\min_{a \in \Sigma} M_a[r(T)]$. En exercice, écrivez le pseudo-code qui implémente cette récurrence.

11.2 Décomposition en arbre et *treewidth*

Nous aimerions bien utiliser les techniques décrites ci-haut sur un graphe $G = (V, E)$, et pas seulement sur un arbre. Pour ce faire, il faudrait trouver

une sorte de représentation en arbre de G. Après beaucoup d'efforts de recherche, une représentation qui s'adapte très bien à la programmation dynamique a été trouvée.

Étant donné un graphe $G = (V, E)$, l'idée est de

— définir des sous-ensembles de sommets $B_1, B_2, \ldots, B_p \subseteq V$, appelés des *sacs*. Ces ensembles peuvent avoir une intersection non-vide, et il faut que p soit polynomial en n. On peut même supposer que $p \in O(n)$.

— construire un arbre $T = (V_T, E_T)$ dans lequel $V_T = \{B_1, \ldots, B_p\}$.

L'arbre T doit satisfaire certaines propriétés pour que la programmation dynamique soit applicable.

Soit $G = (V, E)$ un graphe. On suppose que G n'a pas de sommet isolé. Soit $T = (V_T, E_T)$ un arbre avec $V_T = \{B_1, \ldots, B_p\}$. On dit que T est une *décomposition en arbre de* $G = (V, E)$ si toutes les conditions suivantes sont satisfaites :

1. pour chaque $B_i \in V_T$, $B_i \subseteq V$ (les B_i sont des sacs) ;
2. pour chaque $uv \in E$, il existe $B_i \in V_T$ tel que $u \in B_i$ et $v \in B_i$ (chaque arête est dans au moins un sac) ;
3. pour chaque $v \in V$, alors les sacs contenant v forment un sous-graphe connexe de T. Plus formellement, soient $B_1^v, \ldots, B_q^v$ les sacs de T qui contiennent v. Alors le sous-graphe de T sur les sommets $B_1^v, \ldots, B_q^v$ est connexe.

La *taille* de la décomposition T est $\max_{B_i \in V_T} |B_i| - 1$. La *treewidth* de G est la taille minimum d'une décomposition de G.

Autrement dit, il existe beaucoup de décompositions en arbre pour un graphe G. On cherche celle dans laquelle le plus gros sac est de taille minimum.

Ces conditions peuvent sembler abstraites pour le moment. Il est difficile de donner une intuition quant à leur raison d'être. L'idée est que ces conditions permettent de dire qu'un noeud de l'arbre de décomposition T est un sac B_i qui sépare le graphe en deux parties indépendantes ou plus, où les parties ne partagent par d'arêtes à part celles dans B_i. Ces conditions sont nécessaires afin d'assurer ce fait, et jouent un rôle dans la programmation dynamique.

11.3 Quelques exemples

Il est généralement difficile d'obtenir une décomposition minimum d'un graphe. C'est en fait NP-complet de la calculer. On peut tout de même

s'attarder à quelques exemples simples.

11.3.1 Décomposition en arbre d'un arbre

Si $G = (V, E)$ est un arbre, alors $tw(G) = 1$. Ceci est parce qu'on peut prendre la décomposition $T = (V_T, E_T)$ dans laquelle $V_T = \{\{u, v\} : uv \in E\}$. On suppose que G est enraciné en une feuille ℓ (sinon on réenracine). Ensuite, on ajoute une arête dans E_T entre $\{u, v\}$ et $\{v, w\}$ si et seulement si u est le parent de v et v est le parent de w. La taille de cette décomposition est $\max_{uv \in E} |\{u, v\}| - 1 = 1$.

Il est clair que $\forall uv \in E$, il y a un B_i contenant u et v.

Il faut montrer que pour tout $v \in V$, les sacs contenant v forment un sous-graphe connexe de T. Ceci est évident pour une feuille car un seul sac la contient. Sinon, pour un noeud interne v, ces sacs sont les arêtes contenant v. Il y a le sac-arête $\{u, v\}$ où u est le parent de v, et les sacs-arêtes $\{v, v_i\}$ pour chaque $v_i \in enfant(v)$. Dans T, ces sacs sont liés de façon à ce que les $\{v, v_i\}$ aient une arête avec $\{u, v\}$, et donc c'est connexe.

Pour montrer que T est un arbre, il faut argumenter que T est connexe et n'a pas de cycle. Il est facile de voir que T est connexe parce que G est connexe (à prouver en exercice). De plus, si on suppose que T a un cycle, il a la forme $(\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{k-1}, u_k\}, \{u_k, u_1\})$, où chaque u_i est le parent de u_{i+1}. Ceci veut dire que dans G, on a le cycle $(u_1, u_2, \ldots, u_k, u_1)$, contredisant que G est un arbre.

11.3.2 Décomposition en arbre d'un cycle

Supposons que $G = (V, E)$ est un cycle $(v_1, v_2, \ldots, v_n, v_1)$. On peut montrer que $tw(G) = 2$. On peut obtenir une décomposition en arbre de taille 2 (donc avec des sacs de taille 3 ou moins).

On choisit d'abord v_1 de façon arbitraire. Soit $G - v_1$, c'est-à-dire le graphe obtenu de G en retirant v_1. On voit que $G - v_1$ est un arbre avec les sommets $v_2, \ldots, v_n$. Comme on l'a vu plus plus haut, on peut obtenir une décomposition $T' = (V'_T, E'_T)$ pour $G - v_1$ où les sacs sont tous de taille 2. Maintenant, si on ajoute v_1 à *tous* les sacs de T', on obtient une décomposition pour G. Ceci est parce que les arêtes contenant v_1 vont certainement être dans un des sacs, et parce que les sacs contenant un v_i forment un sous-graphe connexe.

11.3.3 Décomposition en arbre d'une clique

Supposons que $G = (V, E)$ est un clique de taille n. Il s'avère que $tw(G) = n-1$, car essentiellement, la seule façon de décomposer G est d'avoir un seul sac avec V en entier.

Pour le démontrer, supposons que G peut être décomposé en un arbre $T = (V_T, E_T)$ où tous les sacs de V_T ont $n-1$ éléments ou moins. Soit B_r le plus gros sac de T. On peut supposer que B_r est la racine de T. Puisque $|B_r| \leq n-1$, il y a $v \in V$ tel que $v \notin B_r$. Maintenant, soit B_v le descendant le plus près de B_r qui contient v. Puisque les sacs contenant v forment un graphe connexe, tous les sacs contenant v descendent de B_v. De plus, puisque B_r est maximum, il doit y avoir w tel que $w \in B_r$ mais $w \notin B_v$. Puisque les sacs contenant w forment un graphe connexe, aucun descendant de B_v ne contient w. Ceci implique qu'il n'y a aucun sac qui contient à la fois v et w. Ce n'est donc pas une décomposition valide, car vw est une arête de la clique et chaque arête doit être contenue dans un sac.

On déduit qu'il doit y avoir un sac de taille n dans toute décomposition d'une clique.

11.4 Résultats de base

Dans la décomposition en cycle décrite ci-haut, on a enlevé un sommet v, obtenu la décomposition de $G - v$, puis ajouté v à tous les sacs. Cette procédure se généralise à n'importe quel graphe et n'importe quels sommets retirés.

Théorème 30. *Soit $G = (V, E)$ un graphe et $X \subseteq V$. Alors $tw(G) \leq tw(G - X) + |X|$.*

Nous n'allons pas prouver ce théorème. L'idée est simple. Si on prend une décomposition minimum de $G - X$, on peut ajouter X à tous les sacs et démontrer qu'on satisfait toutes les conditions.

Théorème 31. *Soit $G = (V, E)$ et X une clique de taille maximum de G. Alors $tw(G) \geq |X| - 1$.*

Démonstration. Supposons que $tw(G) < |X|-1$. Alors il existe une décomposition $T = (V_T, E_T)$ où chaque sac a au plus $|X| - 1$ éléments. En retirant les sommets qui ne font pas partie de X de ces sacs, on obtient une décomposition pour X de taille $|X| - 1$ ou moins. Ceci contredit le fait qu'une clique X a une treewidth $|X| - 1$. □

Ce fait peut être très utile en pratique. Plusieurs problèmes sur les graphes peuvent être résolus en temps $O(c^{tw(G)} \cdot n^c)$. Si les graphes dans votre application ont une petite treewidth, un tel algorithme peut être très efficace. Par contre, si vous avez des raisons de croire que vos graphes ont de très grosses cliques, vous pouvez immédiatement écarter l'option d'utiliser une décomposition en arbre.

Un dernier résultat, possiblement moins utile en pratique mais tout de même très intéressant, est que $tw(G)$ est équivalent au paramètre des "policiers vs voleurs".

Dans le jeu des **policiers versus voleurs**, on a un graphe $G = (V, E)$ et un voleur placé sur un sommet de G. Il y a aussi k policiers qui sont en hélicoptère, et un hélicoptère peut être soit en vol ou stationné sur un sommet. Les voleurs et policiers jouent chacun leur tour. Dans un tour, un voleur peut se déplacer à n'importe quel sommet accessible sans passer pas un sommet occupé par un policier. Ensuite, un policier peut soit décoller (s'il est stationné), ou atterir à un sommet arbitraire (s'il est en vol). Le but est de savoir s'il est possible que les policiers en arrivent à un point où tous les sommets voisins du voleur sont occupés par des policiers. Le *nombre policier-voleur* est le k minimum tel que k policiers suffisent pour attraper le voleur.

Il s'avère que le nombre policier-voleur d'un graphe G est égal à $tw(G)+1$. Dans un sens, le problème policier-voleur est équivalent à calculer la treewith d'un graphe.

11.5 Algorithmes sur la décomposition en arbre

Pour un designer d'algorithmes, l'élément le plus important de la notion de treewidth est la décomposition en arbre. Un résultat classique en FPT stipule que si un graphe a une treewidth k, alors on peut retrouver une décomposition en arbre en temps $O(k^{O(k^3)})$ (d'autres résultats d'approximation de la treewidth en temps FPT sont aussi connus). De plus, le nombre de noeuds dans la décomposition est linéaire et l'arbre est binaire (rappelons que binaire veut dire que chaque noeud interne a 2 enfants). La preuve de ce théorème est profonde et nous allons l'utiliser en boîte noire pour nos fins.

Théorème 32. *Soit G un graphe tel que $tw(G) = k$. En temps $O(k^{O(k^3)})$, il est possible de trouver une décomposition en arbre binaire de G de largeur k avec $O(|V|)$ noeuds.*

On peut supposer qu'un tel algorithme a été exécuté et qu'une décomposition nous est données. Il ne reste qu'à l'utiliser pour résoudre nos problèmes.

Pour ce faire, on fait un peu comme dans la programmation dynamique sur un arbre, mais en sachant que les sommets représentent des sacs, donc des sous-ensembles de V. Le point fondamental est que la taille des sacs est bornée par $tw(G) + 1$. Lorsqu'on atteint un sac B_i, on peut donc stocker des informations sur chaque sous-ensemble de B_i, ou encore sur chaque permutation de B_i.

11.6 Ensemble indépendant maximum

Comme premier exemple, considérons MAX-INDSET paramétrisé par la treewidth. Rappelons qu'un ensemble X est indépendant si $\forall u, v \in V$, $uv \notin E$.

MAX-INDSET
Entrée : graphe $G = (V, E)$
Paramètre : $tw(G)$
Sortie : un ensemble indépendant $X \subseteq V$ de taille maximum.

Supposons qu'on nous donne une décomposition en arbre $T = (V_T, E_T)$ de G. On peut supposer que T a $O(|V|)$ noeuds et que T est binaire. Pour $B_i \in V_T$ un sac, on rappelons que $B_i \subseteq V$ et que $|B_i| \leq tw(G) + 1$. On dénote par $T[B_i]$ le sous-arbre de T enraciné en B_i. De plus, on dénote par

$$V_i = \bigcup_{B_j \in V(T[B_i])} B_j$$

l'ensemble des sommets de G qui sont présent dans $T[B_i]$.

On voudrait stocker les informations servant à trouver un ensemble indépendant maximum dans $G[V_i]$, le sous-graphe induit par V_i. Une manière classique est de calculer une table $M[S, B_i]$ pour chaque $B_i \in V(T)$ et chaque $S \subseteq B_i$, telle que $M[S, B_i]$ est la taille de l'ensemble indépendant maximum X de $G[V_i]$. On peut exprimer $M[S, B_i]$ avec une récurrence, mais elle devient relativement complexe.

Ici, nous préférons une version plus intuitive qui considère des coloriages en deux couleurs des sommets de G. Pour $S \subseteq V$, un coloriage de S est une fonction $c : S \to \{vert, rouge\}$ qui attribue une couleur *vert* ou *rouge* à chaque sommet de S. On peut voir *vert* comme "est dans l'ensemble

indépendant" et *rouge* comme "n'est pas dans l'ensemble". On dénote par

$$c^{vert}$$

l'ensemble des sommets coloriés en vert.

On peut voir MAX-INDSET comme la recherche d'un coloriage de V tel que c^{vert} est un ensemble indépendant de taille maximum. Il y a 2^n coloriages, ce qui est trop pour un algorithme FPT, mais l'idée ici est d'essayer tous les coloriages possibles pour chaque sac B_i.

Soit $B_i \in V_T$ et soit c_i un coloriage de B_i. On définit

$$M[c_i, B_i]$$

comme la taille maximum d'un ensemble indépendant X de $G[V_i]$, avec la restriction que $X \cap B_i = c_i^{vert}$. On définit $M[c_i, B_i] = -\infty$ si un tel ensemble n'existe pas. Ceci semble représenter beaucoup de concepts, mais notez que ce n'est une généralisation de la programmation dynamique sur un arbre.

Mais comment calculer $M[c_i, B_i]$? Si B_i est une feuille de T, il est facile de voir que

$$M[c_i, B_i] = \begin{cases} |c_i^{vert}| \text{ si } c_i^{vert} \text{ est un ensemble indépendant} \\ -\infty \text{ sinon} \end{cases}$$

Supposons que B_i est un noeud interne. L'idée sera de calculer $M[c_i, B_i]$ à partir des $M[c_j, B_j]$ des enfants de B_i. Soit B_j un enfant de B_i et soit c_j un coloriage de B_j. On dit que c_i et c_j sont *compatibles* si $c_i(u) = c_j(u)$ pour tout $u \in B_i \cap B_j$.

Si c_i^{vert} n'induit pas un ensemble indépendant, on peut mettre $M[c_i, B_i] = -\infty$ immédiatement. Sinon, on peut prouver la récurrence suivante pour $M[c_i, B_i]$:

$$M[c_i, B_i] = |c_i^{vert}| + \sum_{B_j \in enfant(B_i)} \left[\max_{c_j \text{ compatible avec } c_i} (M[c_j, B_j] - |c_i^{vert} \cap c_j^{vert}|) \right]$$

En y réfléchissant assez longtemps, cette récurrence devient intuitive. Pour trouver un ensemble indépendant de $G[V_i]$ qui contient c_i^{vert}, on peut combiner des ensembles indépendants trouvés chez les sous-arbres enfants. Par contre, avant de combiner ces ensembles indépendants, on veut s'assurer que leurs coloriages sont compatibles avec ce qui est demandé par c_i. La soustraction des $|c_i^{vert} \cap c_j^{vert}|$ est faite pour éviter de compter un sommet

vert plus d'une fois.

Ceci n'est qu'une intuition. Il faut habituellement démontrer qu'une telle récurrence est correcte, ce qui demande souvent un travail significatif.

Théorème 33. *La récurrence pour $M[c_i, B_i]$ ci-haut est correcte.*

Démonstration. Pour montrer que cette récurrence est vraie, on va montrer que $M[c_i, B_i]$ est plus petit ou égal à l'expression donnée, et que $M[c_i, B_i]$ est plus grand ou égal à l'expression donnée.

Commençons par la première affirmation. Soit X un ensemble indépendant maximum de $G[V_i]$ tel que $X \cap B_i = c_i^{vert}$. Soit B_j un enfant de B_i dans T. Soit c_{j^*} un coloriage de B_j tel que $c_{j^*}(u) = vert$ si $u \in X$, et $c_{j^*}(u) = rouge$ sinon. Il est clair que c_i et c_{j^*} sont compatibles, car ils sont issus du même ensemble X. Si on considère $X_j = X \cap V_j$, on a un ensemble indépendant de $G[V_j]$ tel que $X_j \cap B_j = c_{j^*}^{vert}$. Par définition, $|X_j| \leq M[c_{j^*}, B_j]$. On a donc

$$\begin{aligned} M[c_i, B_i] &= |c_i^{vert}| + \sum_{B_j \in enfant(B_i)} (|X_j| - |c_i^{vert} \cap c_{j^*}^{vert}|) \\ &\leq |c_i^{vert}| + \sum_{B_j \in enfant(B_i)} (M[c_j, B_j] - |c_i^{vert} \cap c_{j^*}^{vert}|) \\ &\leq |c_i^{vert}| + \sum_{B_j \in enfant(B_i)} \left[\max_{c_j} (M[c_j, B_j] - |c_i^{vert} \cap c_j^{vert}|) \right] \end{aligned}$$

(ou la soustraction est faite pour éviter le double-comptage).

Dans l'autre sens, soient $B_{j_1}, \ldots, B_{j_l}$ les enfants de B_i. Pour chaque B_{j_h}, soit X_{j_h} un ensemble indépendant de $G[V_{j_h}]$ tel que le coloriage c_{j_h} de B_{j_h} correspondant à X_{j_h} est compatible avc c_i, et qui maximise $|X_{j_h}| - |c_i^{vert} \cap c_{j_h}^{vert}|$. On veut montrer que

$$X = c_i^{vert} \cup X_{j_1} \cup \ldots \cup X_{j_l}$$

est un ensemble indépendant. On note que puisque les coloriages c_{j_h} sont tous compatibles avec c_i, $X \cap B_i = c_i^{vert}$. Supposons qu'il existe $u, v \in X$ partagent une arête. Alors $u, v \in B_i$ est impossible car on vérifie avant tout que c_i^{vert} induit un ensemble indépendant et $X \cap B_i = c_i^{vert}$. Donc, $u \in V_{j_h}$ pour un enfant B_{j_h} de B_i et $u \notin B_i$. Puisque les sacs contenant u forment un sous-graphe connexe de T, toutes les occurrences de u sont dans V_{j_h}. De plus, puisque $uv \in E$, il doit y avoir un sac B_{uv} descendant de B_{j_h} tel que $u, v \in B_{uv}$. Donc, u et v ont tous les deux des occurrences dans V_{j_h} Ceci voudrait dire que $u, v \in X_{j_h}$, ce qui est contradictoire car X_{j_h} est censé être

un ensemble indépendant de V_{j_h}. Donc notre X est bel et bien un ensemble indépendant.

On note que la taille de X est donnée par $|c_i^{vert}| + \sum_{B_{j_h}} \left[|X_{j_h}| - |c_i^{vert} \cap c_{j_h}^{vert}| \right]$. Ceci est pour soustraire les intersections des X_{j_h} avec c_i^{vert}, et parce que les X_{j_h} n'ont en commun que des éléments de B_i (à cause de la condition 3 des décompositions). Puisque les $|X_{j_h}| - |c_i^{vert} \cap c_{j_h}^{vert}|$ sont de taille maximum, la taille de X est moins $|c_i^{vert}| + \sum_{B_j \in enfant(B_i)} \left[\max_{c_j} (M[c_j, B_j] - |c_i^{vert} \cap c_j^{vert}|) \right]$, tel que désiré. □

11.7 Jolies décompositions

Comme on l'a vu, les récurrences sur les décompositions peuvent devenir complexes. Il existe une façon de simplifier ces récurrences (un peu) consistant à simplifier la décomposition en soi.

Soit $G = (V, E)$ et soit $T = (V_T, E_T)$ une décomposition en arbre de G. On dit que T est une *jolie* décomposition si :

1. chaque noeud B_i de T a $0, 1$ ou 2 enfants ;
2. si un noeud B_i a 2 enfants B_l et B_r, alors $B_i = B_l = B_r$.
   Dans ce cas, B_i est appelé un NOEUD DE JOINTURE.
3. si un noeud B_i a 1 enfant B_j, alors un de ces deux cas est possible :
   — $B_i = B_j \cup \{v\}$ pour un certain $v \in V$. Dans ce cas, B_i est appelé un NOEUD D'INTRODUCTION.
   — $B_i = B_j \setminus \{v\}$ pour un certain $v \in B_j$. Dans ce cas, B_i est appelé un NOEUD D'OUBLI.

L'avantage des jolies décompositions est que le contenu d'un noeud B_i par rapport à celui de ses enfants est simple, ce qui nous permet de simplifier la définition des récurrences aux noeuds B_i. Le désavantage est qu'il faut gérer trois cas de figure pour nos récurrences : il faut décrire une récurrence selon le type du noeud B_i, qui pourrait être un NOEUD DE JOINTURE, un NOEUD D'INTRODUCTION ou un NOEUD D'OUBLI.

un fait important est qu'on peut toujours trouver une jolie décomposition sans changer la taille de la décomposition.

Théorème 34. *Soit $G = (V, E)$ un graphe. Si on nous donne une décomposition en arbre $T' = (V'_T, E'_T)$ de G de taille $tw(G)$, on peut transformer T' en une **jolie** décomposition de taille $tw(G)$ en temps polynomial.*

11.8 MAX-INDSET et jolie décomposition

Reprenons le problème MAX-INDSET, mais supposons que $T = (V, T)$ est jolie. On reprend les notions de coloriage et de compatibilité entre coloriages.

Rappelons que $M[c_i, B_i]$ dénote la taille maximum d'un ensemble indépendant X de $G[V_i]$ tel que $X \cap B_i = c_i^{vert}$. Si B_i est une feuille, la définition de $M[c_i, B_i]$ ne change pas. Si B_i est un noeud interne, on vérifier d'aobrd si les éléments de c_i^{vert} contiennent une arête. Si oui, alors $M[c_i, B_i] = -\infty$. Sinon, on a trois cas possibles pour $M[c_i, B_i]$:

1. B_i est un NOEUD D'INTRODUCTION. Soit B_j l'enfant de B_i et soit $v \in V$ tel que $B_i = B_j \cup \{v\}$. Puisque $B_j \subseteq B_i$, on note qu'il n'y a qu'un seul coloriage de B_j compatible avec c_i, car on doit reprendre les mêmes couleurs et ce peu importe la couleur de v. Soit c_j^* l'unique coloriage de B_j compatible avec c_i.
   Alors
   $$M[c_i, B_i] = M[c_j^*, B_j] + \begin{cases} 1 \text{ si } c(v) = vert \\ 0 \text{ sinon} \end{cases}$$
   Ceci correspond à reprendre un ensemble indépendant optimal sur V_j, et ajouter v si on l'a colorié en vert. Il est important de comprendre pourquoi l'ajout de v ne peut pas créer d'arête dans l,ensemble indépendant.
2. B_i est un NOEUD D'OUBLI. Soit B_j l'enfant de B_i et soit v tel que $B_i = B_j \setminus \{v\}$. On n'ajoute aucun sommet au niveau B_i, donc on prend une solution optimale au niveau B_j.
   Alors
   $$M[c_i, B_i] = \max_{c_j \text{ compatible avec } c_i} M[c_j, B_j]$$
   Notez qu'ill n'y a que deux coloriages de B_j compatibles avec c_i, dépendamment de si on ajoute v on non.
3. B_i est un NOEUD DE JOINTURE. Soient B_l et B_r les enfants de B_i et rappelons que $B_i = B_l = B_r$. Il suffit de prendre un ensemble indépendant au niveau B_l et d'en faire l'union avec un ensemble indépendant au niveau B_r. dans les deux cas, il n'y a qu'un seul coloriage compatible, et il éviter de double-compter les sommets verts.
   Alors
   $$M[c_i, B_i] = M[c_i, B_l] + M[c_i, B_r] - |c_i^{vert}|$$

Nous n'allons pas démontrer que ces récurrences sont correctes. Il est

toutefois important de se convaincre que le cas des noeuds de jointure fonctionne. Faire la somme $M[c_i, B_l] + M[c_i, B_r]$ correspond à prendre un ensemble indépendant maximum de chaque côté et de les combiner. Comment sait-on qu'on n'a pas inclut deux sommets partageant une arête en ce faisant ? C'est une conséquence des propriétés des décompositions, qui assurent que B_i est un séparateur entre ces deux ensembles.

11.9 MAX-CUT et jolies décompositions

Rappelons que dans MAX-CUT, on a un graphe $G = (V, E)$ et on veut une bipartition (V_1, V_2) de G telle que $|E(V_1, V_2)|$ est maximum. Nous paramétrisons par $tw(G)$ et supposons qu'on reçoit une jolie décomposition $T = (V_T, E_T)$.

L'idée du coloriage est encore utile ici. Chaque sommet aura une couleur 1 ou 2 représentant sa présence dans V_1 ou V_2. Pour un sac B_i, c_i représente un coloriage des sommets de B_i avec 1 ou 2. On écrit $c_i^1 = \{v \in B_i : c(v) = 1\}$ et $c_i^2 = \{v \in B_i : c(v) = 2\}$. La notion de compatibilité entre coloriages demeure la même.

Pour $B_i \in V_T$ et c_i un coloriage de B_i, on dénote par $M[c_i, B_i]$ le nombre maximum d'arêtes traversantes dans une bipartition (V_1', V_2') de $G[V_i]$ telle que $c_i(v) = 1 \Rightarrow v \in V_1'$ et $c_i(v) = 2 \Rightarrow v \in V_2'$.

Si B_i est une feuille, il est facile de voir que

$$M[c_i, B_i] = |E(c_i^1, c_i^2)|$$

Sinon, supposons que B_i est un noeud interne. On a trois cas.

— B_i est un NOEUD D'INTRODUCTION. Soit B_j l'enfant de B_i et v le nouvel élément de B_i. Dénotons par c_j^* l'unique coloriage de B_i compatible avec c_i. Le nombre d'arêtes traversantes est le même qu'avant, en plus des nouvelles arêtes traversantes qui incluent v. Par les propriétés des décompositions, tous les voisins de v dans V_i doivent être dans B_i.
Alors

$$M[c_i, B_i] = M[c_j^*, B_j] + \begin{cases} |N(v) \cap c_i^2| \text{ si } c_i(v) = 1 \\ |N(v) \cap c_i^1| \text{ si } c_i(v) = 2 \end{cases}$$

— B_i est un NOEUD D'OUBLI. Soit B_j l'enfant de B_i. Dans ce cas, on n'ajoute pas de sommet à notre bipartition et on peut prendre la meilleure solution au niveau B_i qui est compatible (notez qu'il n'y en

n'a que deux).
Alors

$$M[c_i, B_i] = \max_{c_j \text{ compatible avec } c_i} M[c_j, B_j]$$

— B_i est un NOEUD DE JOINTURE. Soient B_l et B_r les enfants de B_i, en rappelant que $B_i = B_l = B_r$. Il suffit de combiner les bipartitions au niveau de B_l et B_r, en prenant soin de ne pas compter une même arête deux fois.
Alors

$$M[c_i, B_i] = M[c_i, B_l] + M[c_i, B_r] - |E(c_i^1, c_i^2)|$$

Chapitre 12

Conclusion

Nous avons vu diverses solutions algorithmiques applicables lorsque confrontés à un problème NP-complet. On peut soit tenter de l'approximer rapidement, ou bien d'extraire un paramètre petit duquel dépend la complexité exponentielle. La plupart des problèmes rencontrés sont des problèmes jouets. L'utilité des approches présentées ici consiste à faire des liens entre la théorie et les problèmes informatiques de la vraie vie.

L'approximation est nécessaire lorsque les jeux de données sont très grands, par exemple dans les millions, car dans ce cas il y a très peu de chances qu'on trouve un paramètre petit. Par contre, les jeux de données de taille de l'ordre de milliers peuvent souvent être gérés à l'aide de méthodes exactes FPT. Ce cours a servi à vous présenter les techniques principales de développement d'algorithme. Les défis qui vous attendent sont de savoir **quand** appliquer ces techniques. Et si jamais vous entreprenez des recherches en algorithmiques, un défi de plus grande taille vous attend : créer des *nouvelles* techniques.