A l'intérieur d'un solveur SAT

François Schwarzentruber

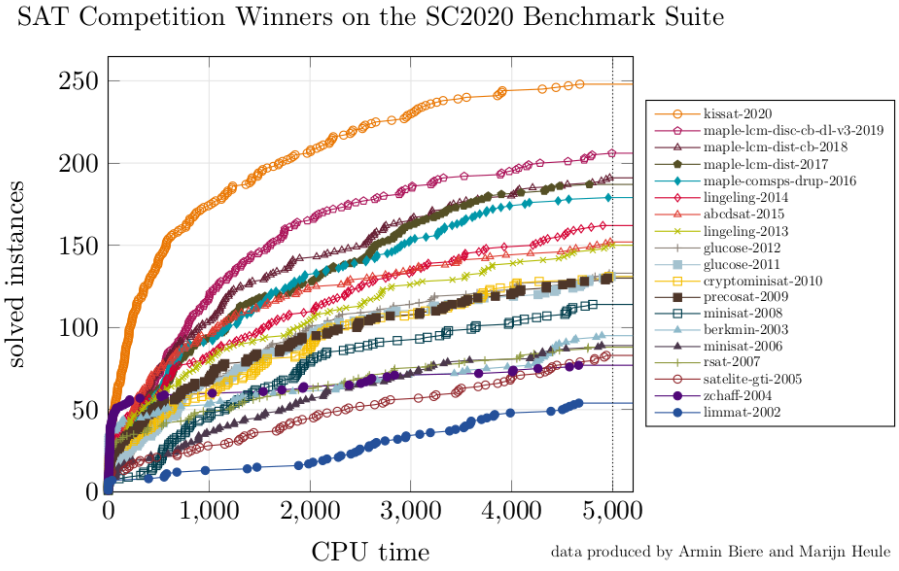
20 mars 2024

SAT
entrée : une formule φ en forme normale conjonctive
sortie : une valuation qui satisfait φ si φ est satisfiable ; unsat si φ est insatisfiable.

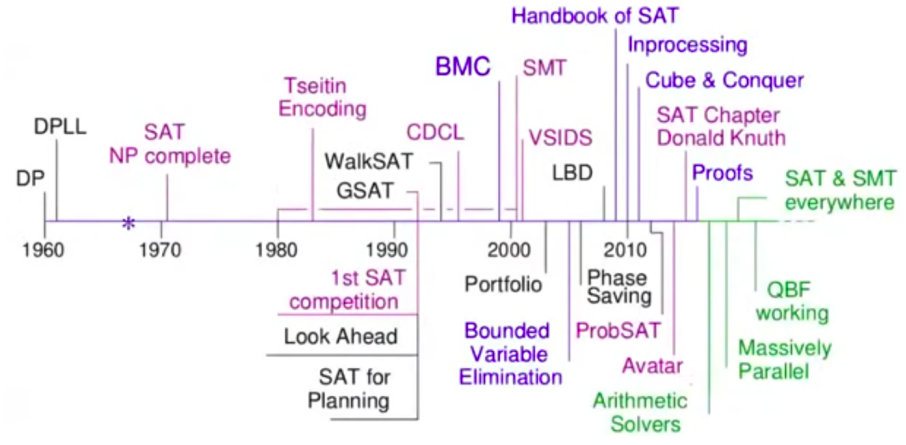
1 Applications

— bounded model checking
— planification automatique
— sous-routine pour des solveurs d'autres logiques (SMT, QBF, logique du premier ordre, ASP, etc.)
— résoudre des problèmes mathématiques (problème des triplets pythagoriciens booléens) [HKM16]

2 Performances des solveurs SAT



3 Historique



par Armin Biere

4 Notations

Soit AP un ensemble dénombrable de variables propositionnelles.

Définition 1 (valuation partielle) Une valuation partielle est une fonction partielle de AP dans $\{0,1\}$.

Exemple 1
$\nu = \emptyset$
$\nu = \{(p,0),(q,1)\}$.

Remarque 2 Soit ℓ un littéral. $\neg\ell = \begin{cases}\neg p & \text{si } \ell = p \\ p & \text{si } \ell = \neg p\end{cases}$

Définition 3 Pour tout $i \in \{0,1\}$, on note $\begin{cases}\nu[p := i] & = \nu \cup \{(p,i)\} \\ \nu[\neg p := i] & = \nu \cup \{(p,1-i)\}.\end{cases}$

Définition 4 Soit ℓ un littéral. On définit la notation $\nu \models \ell$ pour dire que $\begin{cases}(p,1) \in \nu & \text{si } \ell = p \\ (p,0) \in \nu & \text{si } \ell = \neg p\end{cases}$

Définition 5 (littéral assigné) ℓ est ν-assigné si $\nu \models \ell$ or $\nu \models \neg\ell$.

Définition 6 (condition de vérité d'une forme normale conjonctive)
— $\nu \models \bigwedge_{i\in\{1,\dots,n\}} \bigvee_{j\in\{1,\dots,k_i\}} \ell_{i,j}$ si pour tout $i \in \{1,\dots,n\}$, il existe $j \in \{1,\dots,k_i\}$ tel que $\nu \models \ell_{i,j}$.
— $\nu \models \neg \bigwedge_{i\in\{1,\dots,n\}} \bigvee_{j\in\{1,\dots,k_i\}} \ell_{i,j}$ si il existe $i \in \{1,\dots,n\}$ pour tout $j \in \{1,\dots,k_i\}$, on a $\nu \models \neg\ell_{i,j}$.

5 Backtracking

— entrée : une valuation partielle ν, une formule φ sous forme normale conjonctive
— sortie : *true* si ν peut être étendue en une valuation totale qui satisfait φ

fonction $dpll(\nu,\varphi)$
  si $\nu \models \neg\varphi$ **alors renvoyer** $false$
  si $\nu \models \varphi$ **alors renvoyer** $true$
  sinon
    choisir une variable p non ν-assignée
    renvoyer $dpll(\nu[p := \bot],\varphi)$ ou $dpll(\nu[p := 1],\varphi)$

6 Davis–Putnam–Logemann–Loveland (DPLL)

6.1 Clause unitaire

Définition 7 (clause unitaire) Une clause $\bigvee_j \ell_j$ est ν-unitaire s'il existe j_0 tel que pour tout $j \neq j_0$, $\nu \models \neg\ell_j$ et ℓ_{j_0} non ν-assigné.

$$\not{p} \vee \not{q} \vee \not{\neg s} \vee \ell.$$

Exemple 2 (élimination des littéraux purs) Suppose the formula φ is

$$(p \vee q \vee \ell) \wedge (r \vee s \vee \neg u \vee \ell) \wedge (s \vee \neg t).$$

If ℓ is ν-unassigned, we replace ν by $\nu[\ell := 1]$.

6.2 Pseudo-code standard

fonction $dpll(\nu,\varphi)$
  ν := propagationsUnitaires(ν, φ)
  si $\nu \models \neg\varphi$ **alors renvoyer** $false$
  si $\nu \models \varphi$ **alors renvoyer** $true$
  sinon
    choisir une variable p non ν-assignée
    renvoyer $dpll(\nu[p := \bot],\varphi)$ ou $dpll(\nu[p := 1],\varphi)$

```
— entrée : une valuation partielle ν, une CNF φ
— sortie : ν′ qui étend ν telle que ν peut être étendue en une valuation satisfaisant φ
    ssi ν′ peut être étendue en une valuation satisfaisant φ
fonction propagationsUnitaires(ν, φ)
    tant que il y a une clause ν-unitaire dans φ avec ℓ non ν-assigné
        ν := ν[ℓ := 1]
    renvoyer ν
```

Exemple 8 $\overbrace{(p \vee q)}^{\alpha} \wedge \overbrace{(p \vee \neg q)}^{\beta} \wedge \overbrace{(\neg p \vee q)}^{\gamma} \wedge \overbrace{(\neg p \vee \neg q)}^{\delta} \wedge \overbrace{(a \vee b)}^{\epsilon}$.

Supposons que les choix faits en premier sont sur les variables a et b. Dès lors que $a \vee b$ est vraie, on doit maintenant étudier les choix pour p et q.

Supposons p vrai. On infère q oar γ et il y a une contraction avec δ. Ensuite, si nous supposons que p est faux, nous déduisons q par δ et il y a une contraction avec β. (*)

Mais le raisonnement ennuyeux (*) se produit également pour $a, \neg b$, $\neg a, b$ et $\neg a, \neg b$!

6.3 Pseudo-code avec backtracking explicite

```
fonction dpll(ν, φ)
    si ν = × alors renvoyer false
    sinon
        ν := propagationsUnitaires(ν, φ)
        si ν ⊨ ¬φ alors renvoyer dpll( backtrack(ν, φ), φ )
        si ν ⊨ φ alors renvoyer true
        sinon
            choisir une variable p non ν-assignée
            renvoyer dpll(ν[ℓ := 1], φ)
```

Pour faire fonctionner cet algorithme, la structure ν est augmentée comme suit. Nous marquons chaque littéral choisi par :

p	:	p a été dérivé par propagation unitaire
$\neg p$	:	$\neg p$ a été choisi
p	:	p a été dérivé par backtracking

Exemple 9 Voici des exemples de valeurs renvoyées pour $backtrack(\nu, \varphi)$.

ν	$backtrack(\nu, \varphi)$
$\neg p\ q\ r$	p
$p\ q$	×
$\neg p\ q\ r\ \neg s\ t$	$\neg p\ q\ r\ s$
$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ e\ b$	$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ \neg e$
$\neg p\ q\ r\ s$	$\neg p$

Pour tester si φ est satisfiable ou non, nous appelons $dpll(\epsilon, \varphi)$ où ϵ est la valorisation vide. La valorisation vide est différente de $\bullet$ qui dénote la contradiction.

7 Conflict-driven clause learning (CDCL)

```
fonction cdcl(ν, φ)
    si ν = × alors renvoyer false
    sinon
        ν := propagationsUnitaires(ν, φ)
        si ν ⊨ ¬φ alors renvoyer cdcl( backtrack − learning(ν, φ) )
        si ν ⊨ φ alors renvoyer true ;
        sinon
            choisir une variable p non ν-assignée
            renvoyer cdcl(ν[ℓ := 1], φ)
```

Exemple 10 (cet exemple vient d'une présentation à IAF2007 de Pascal Nicolas et Laurent Simon)

$$\overbrace{(x1 \vee x2)}^{\alpha} \wedge \overbrace{(\neg x2 \vee \neg x3)}^{\beta} \wedge \overbrace{(x3 \vee \neg x12)}^{\gamma} \wedge \overbrace{(\neg x2 \vee \neg x4 \vee \neg x5)}^{\delta} \wedge \overbrace{(x3 \vee x5 \vee x6 \vee x7)}^{\epsilon} \wedge$$
$$\overbrace{(\neg x7 \vee x8 \vee \neg x9)}^{\theta} \wedge \overbrace{(x6 \vee \neg x7 \vee x9 \vee x10)}^{\lambda} \wedge \overbrace{(\neg x7 \vee \neg x10 \vee x8 \vee x11)}^{\mu} \wedge \overbrace{(x13 \vee \neg x14 < \vee \neg x15)}^{\nu} \wedge$$
$$\overbrace{(x8 \vee \neg x7 \vee x11)}^{\rho} \wedge \overbrace{(\neg x11 \vee \neg x13)}^{\sigma} \wedge \overbrace{(\neg x11 \vee x14)}^{\tau} \wedge \overbrace{(x12 \vee x15)}^{\chi}$$

Le problème du backtracking est que l'ordre dans lequel nous choisissons les littéraux peut être loin d'être optimal ! Dans DPLL, comme on le voit, nous pouvons avoir :

avant	après
$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ e\ b$	$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ \neg e$

Voici notre rêve :

avant	après	
$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ e\ b$	$\underbrace{\neg p\ q\ r\ \neg s\ t}_{\text{un nombre minimum de choix}}$	$\underbrace{a}_{\text{déduit par propagation unitaire}}$

C'est ce qu'on appelle le *backjumping*. Tout d'abord, nous rappelons l'algorithme que nous souhaitons concevoir avec la fonction backtrack-learning Ensuite nous verrons que nous représentons la valorisation partielle ν avec un graphe G. Ce graphe est appelé *graphe d'implication*. Nous verrons ensuite comment fonctionne la fonction backtrack-learning.

7.1 Graphe d'implication

Une valuation partielle ν est désormais représentée par un graphe d'implication qui contient également des informations sur la propagation unitaire.

7.1.1 Definition

Définition 11 (graphe d'implication) Un graphe d'implication est un graphe $G = (V, E)$ tel que :

— Un nœud dans V est :

— soit un *nœud littéral choisi* représenté en jaune :

où ℓ est un littéral et $n \in \mathbb{N}$ est appelé le niveau ;

— un *nœud littéral déduit* représenté en bleu :

où ℓ est un littéral et $n \in \mathbb{N}$ est appelé le niveau ;

— ou un *nœud de contradiction* représenté en orange :

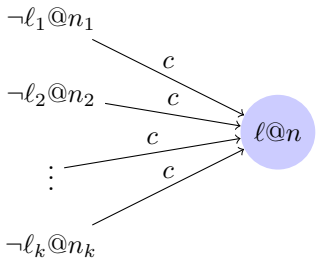
— Les arcs dans E sont orientés et étiquetés par des clauses.

7.1.2 Opérations

Au cours du processus, les opérations suivantes peuvent avoir lieu :

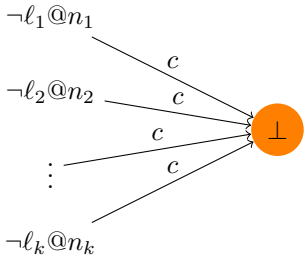
— Choisir un littéral consiste à ajouter un nouveau nœud littéral choisi de niveau $n+1$ où n est le niveau courant (c'est-à-dire le niveau maximum apparaissant dans le graphe courant) ;

— Déduire un littéral par propagation unitaire. Pour toute clause unitaire de la forme $c = \ell \vee \ell_1 \vee \cdots \vee \ell_k$, si les littéraux $\neg\ell_1, \ldots \neg\ell_k$ sont dans le graphe, alors on y ajoute ℓ :



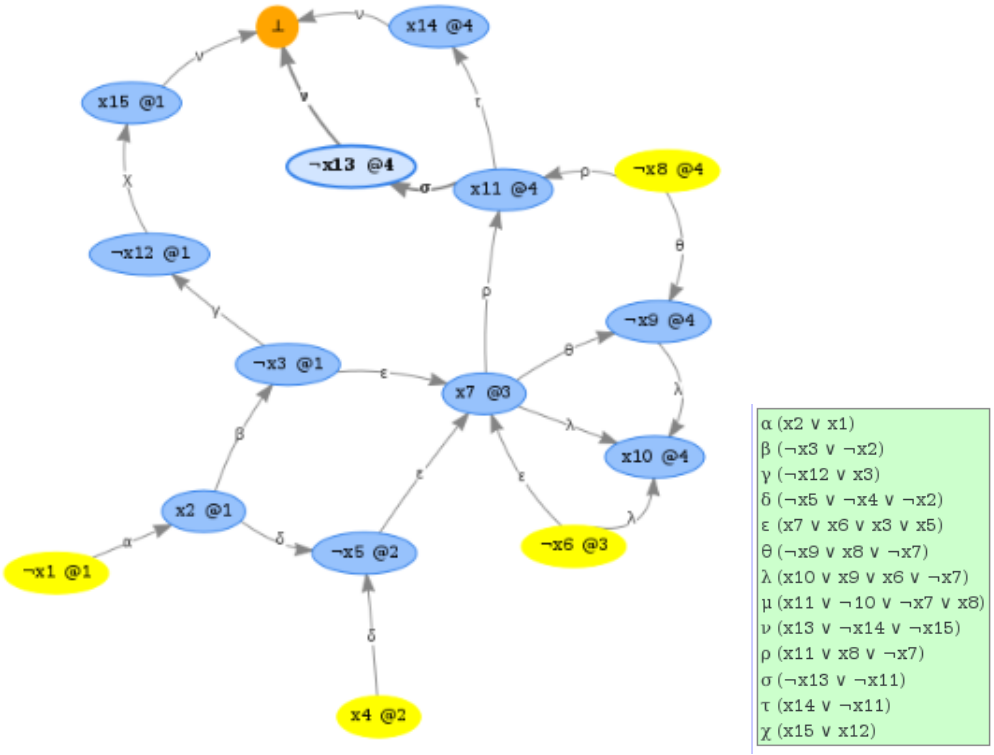
où $n = max_i n_i$. On note $reason(\ell) = c$.

— Aboutir à une contradiction. Si une clause conflictuelle $c = \ell_1 \vee \cdots \vee \ell_k$ est trouvée, i.e. les littéraux $\neg\ell_1, \ldots \neg\ell_k$ sont dans le graphe, alors on ajoute le nœud contradiction.



Yellow nodes are literals that have been selected. For instance, $\neg x_6$ has been chosen at decision level 1. $x1$ has been derived from clause ϵ because $\neg x5$ and $x2$ were true.

Exemple 12 On reprend l'exemple 9. Voici le graphe d'implication correspondant en choisissant les littéraux dans l'ordre $\neg x_1, x_4, \neg x_6, \neg x_8$:



7.2 Backtracking vu comme apprentissage d'une clause

On voit que

$$\neg x1@1 \qquad x4@2 \qquad \neg x6@3 \qquad \neg x8@4$$

aboutit à une contradiction.

Backtracking correspond à :

— Apprendre la clause $(x1@1 \vee \neg x4@2 \vee x6@3 \vee x8@4)$; (on indique ici les niveaux des littéraux dans la clause pour plus de lisibilité)

— Supprimer les nœuds de niveau 4

— Déduire $x8@3$ par propagation unitaire depuis $(x1 \vee \neg x4 \vee x6 \vee x8)$.

7.3 Notre objectif

Définition 13 Une clause est de la bonne forme ressemble à

$$c_L := \ell@n \vee \underbrace{\ell_1 \vee \ell_2 \vee ...}_{\text{de niveau inférieur à } n'}$$

où n est le niveau courant et $n' < n$ le plus petit possible.

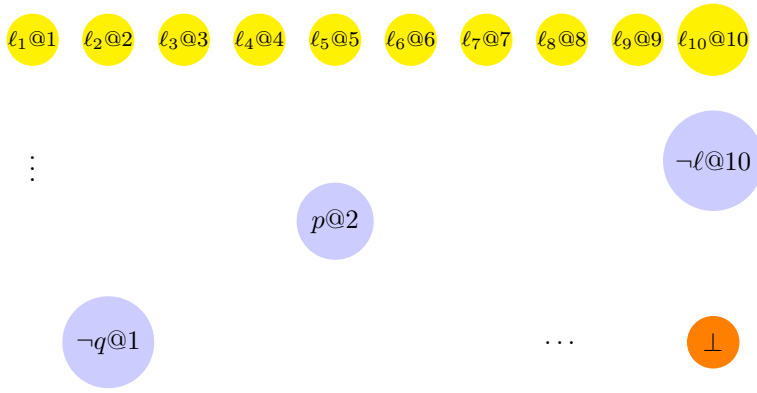
Ainsi le backjumping consiste à :

1. Apprendre la clause c_L ;
2. Supprimer tous les noeuds de niveau $> n'$;
3. Inférer $\ell@n'$ par propagation unitaire depuis c_L.

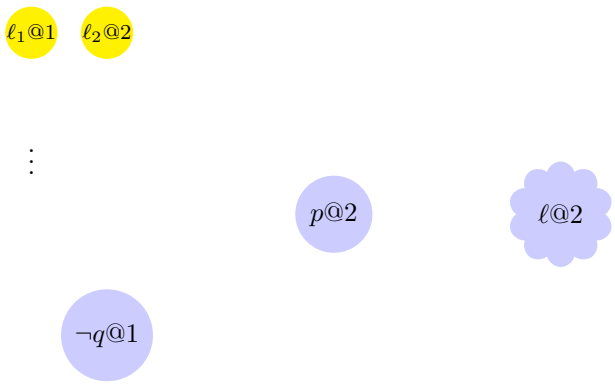
Supposons que la clause apprise soit

$$\ell@10 \vee \underbrace{\neg p}_{\text{level 2}} \vee \underbrace{q}_{\text{level 1}}$$

et que les littéraux choisis sont $\ell_1@1, \ldots, \ell_{10}@10$. Voici une vue schématique de ce qu'il se passe :



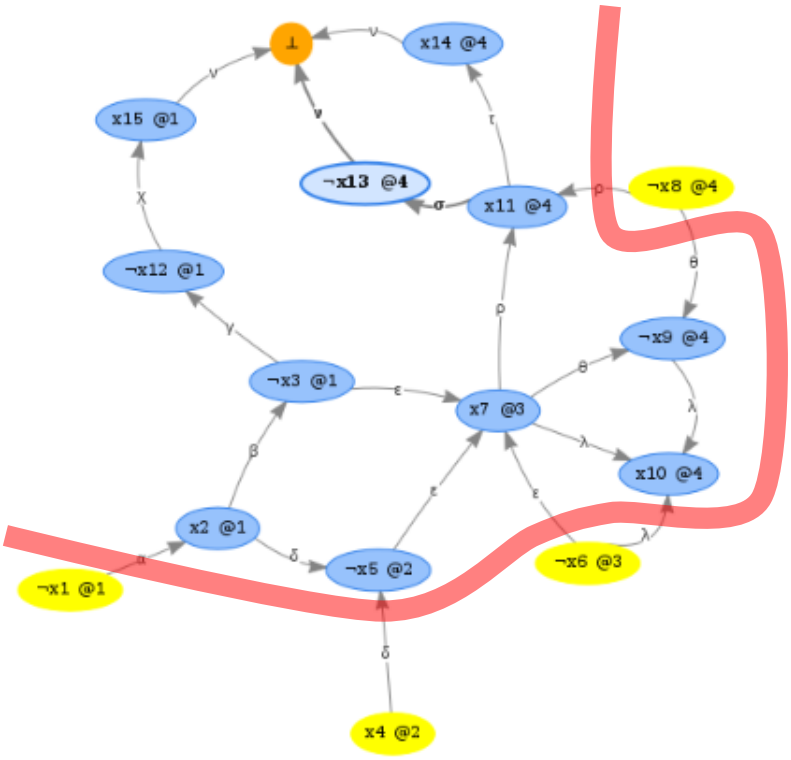
Après le backjumping, on oublie tout après le niveau > 2. Les littéraux choisis restants sont ℓ_1, ℓ_2. Comme p et $\neg q$ sont toujours dans le graphe d'implication, on en déduit $\ell@2$ via la clause apprise :



7.4 Clause apprise est une coupe

Afin, d'apprendre une clause de la bonne forme, on calcule une coupe dans le graphe. On lit la clause à apprendre (plus précisément ce sont les négations des littéraux de la clause qu'on lit) le long de la coupe.

Exemple 14 La coupe correspond à la clause d'apprentissage pour le backtracking :



Le long de la coupe on lit :

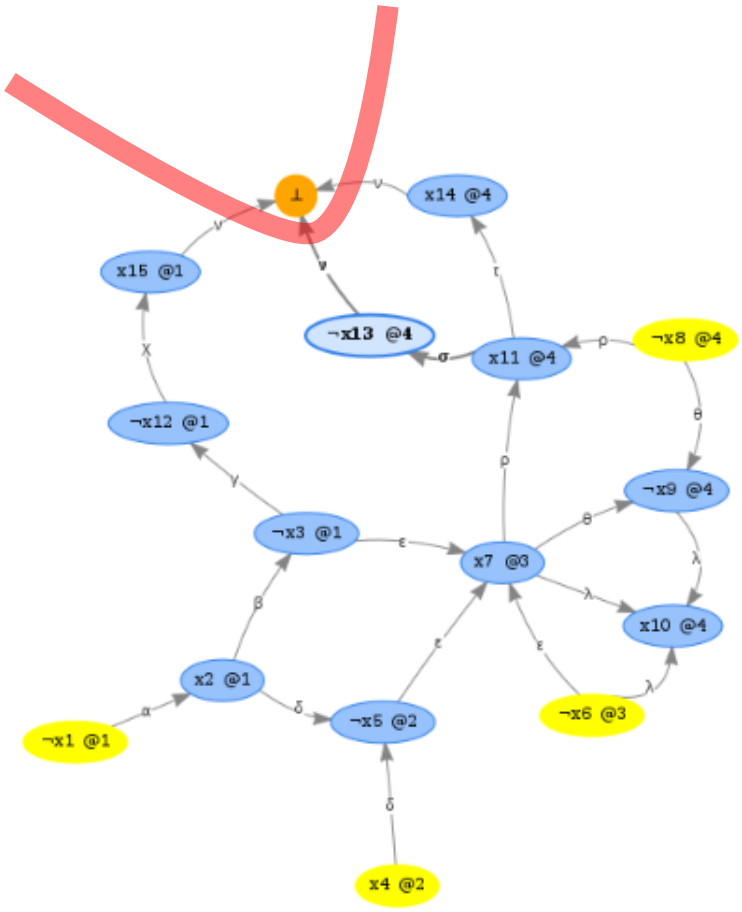
$$\neg x1@1 \qquad x4@2 \qquad \neg x6@3 \qquad \neg x8@4$$

Autrement dit, on peut :

1. apprendre la clause $x_1@1 \vee \neg x_4@2 \vee x_6@3 \vee x_8@4$.
2. oublier tout le niveau 4
3. déduire x_8 au niveau 3.

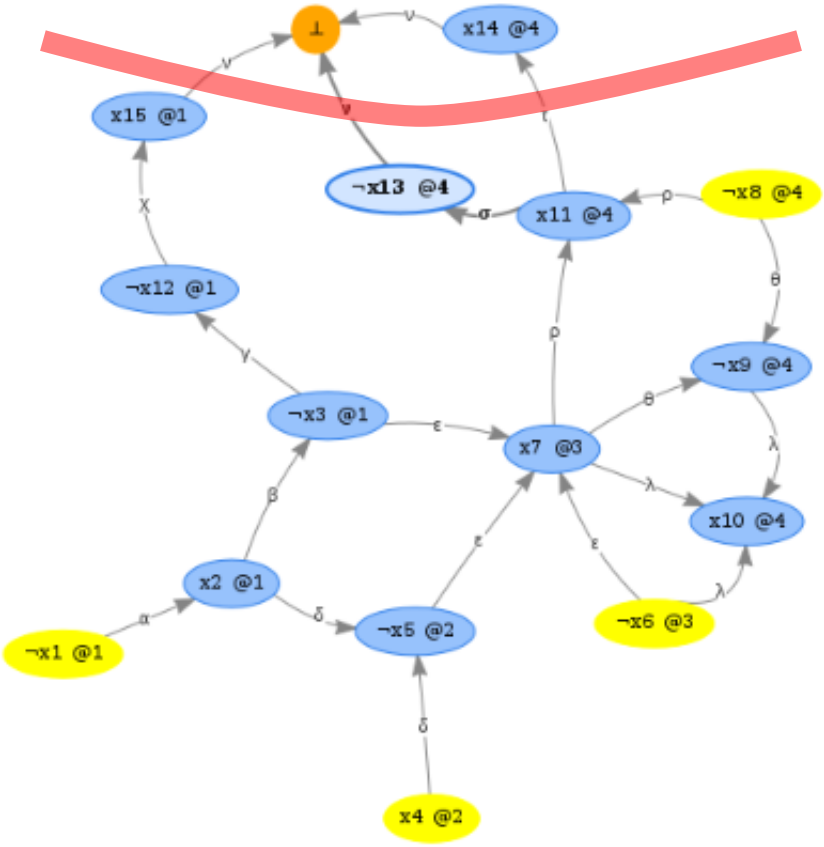
On espère apprendre une clause qui nous fait oublier plus de niveau que la clause correspondante au backtracking. Pour cela, l'idée est de partir de la clause conflictuelle (i.e. la coupe toute proche de $\bot$). Puis d'avancer, comme des fantassins romains, jusqu'à obtenir une clause de la bonne forme. C'est une manière d'espérer avoir mieux que la clause correspondante au backtracking.

Exemple 15 Dans l'exemple, la clause conflictuelle est $x13@4 \vee \neg x14@4 \vee \neg x15@1$. Elle n'est pas de la bonne forme car il y a deux littéraux de niveau 4.

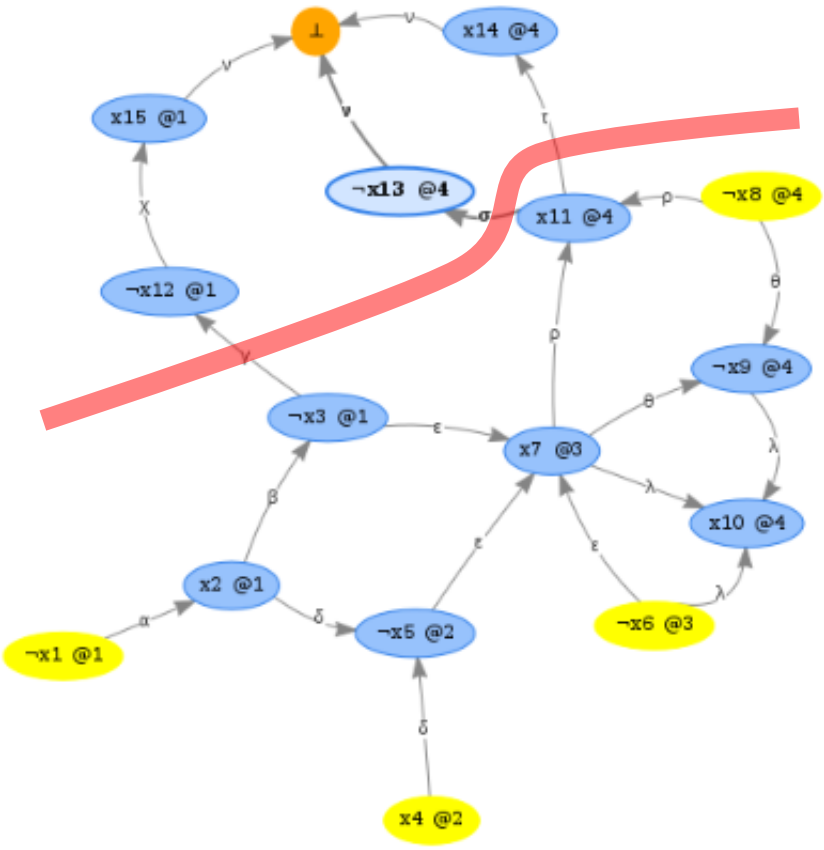


De manière intéressante, en avançant comme des fantassins, on a d'autres clauses.

Exemple 16 Voici la coupe correspondant à $\neg x_{15}@1 \vee x_{13}@4 \vee x_{14}@4$.



Exemple 17 Si on avance un peu plus, on a une clause de la bonne forme $(\neg x3@4 \wedge x11@1)$.



On peut :
— l'apprendre
— Supprimer les noeuds de niveau > 1 ;
— Déduire $\neg x11@1$ par propagation unitaire avec $(\neg x11 \wedge x3@1)$.

8 Backjumping et learning : algorithme

On construit $backtrack - learning(\nu, \varphi)$ qui renvoie la nouvelle valuation ν (un graphe d'implication) et la nouvelle formule φ (avec la clause apprise ajoutée). On en décrit une implémentation possible.

8.1 Learning as resolution steps

Définition 18 (condition d'application de la résolution) Soit c_1 et c_2 deux clauses. On dit que l'on peut appliquer la résolution quand il existe une variable r telle qu'une clause a un littéral r et l'autre a un littéral $\neg r$, $c_1 \bigodot c_2$.

Définition 19 (résolution) Soit c_1 et c_2 deux clauses. Quand on peut appliquer la résolution sur c_1 et c_2 pour une certaine variable r. Alors on note $c_1 \bigodot c_2$ la clause qui contient tous les littéraux de c_1 et c_2 à l'exception de r et $\neg r$.

Exemple 20 $(p \vee q \vee \neg r) \bigodot (r \vee s \vee u) = (p \vee q \vee s \vee u)$.

Proposition 21 Soit ν une valuation complète. Soit c_1 et c_2 sur lesquelles on peut appliquer la résolution.

$$(\nu \models c_1 \text{ and } \nu \models c_2) \text{ implies } \nu \models (c_1 \bigodot c_2).$$

On définit maintenant une suite de clauses qui correspond à l'avancement de fantassins dans le graphe d'implication.

Définition 22 (calcul de la clause apprise) Soit n le niveau actuel. Nous définissons la séquence de clauses suivante :

$c_0 =$ la clause conflictuelle

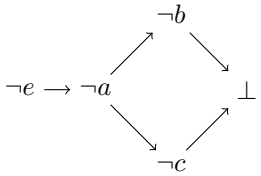
$$c_{i+1} = \begin{cases} c_i & \text{si } c_i \text{ est de la bonne forme} \\ c_i \bigodot reason(\ell) & \text{où } \ell@n, \neg\ell \in c_i \text{ et } reason(\ell) \text{ est défini} \end{cases}$$

Remarque 23 Si c_i n'est pas de la bonne forme, ce n'est pas la clause de backtracking donc on trouve toujours un ℓ avec $\ell@n$, $\neg\ell \in c_i$ et $reason(\ell)$ est défini.

Exemple 24 On reprend l'exemple 16.

i	c_i	
0	$\neg x15@1 \vee x13@4 \vee \neg x14@4$	
		résolution avec $reason(x14) = (x14 \vee \neg x11)$
1	$\neg x15@1 \vee x13@4 \vee \neg x11@4$	
		résolution avec $reason(\neg x13) = (\neg x13 \vee \neg x11)$
2	$\neg x15@1 \vee \neg x11@4$	

Exemple 25 Considérons ce graphe d'implication :



La clause conflictuelle est $b \vee c$.
On peut faire : $(b \vee c) \bigodot (a \vee \neg b) = a \vee c$
$(a \vee c) \bigodot (\neg a \vee e) = e \vee c$
$(e \vee c) \bigodot (\neg c \vee a) = e \vee a$
$(e \vee a) \bigodot (\neg a \vee e) = e$

ou alors :
$(b \vee c) \bigodot (a \vee \neg b) = a \vee c$
$(a \vee c) \bigodot (\neg c \vee a) = a$

Proposition 26 Il existe i_0 tel que c_{i_0} soit de la bonne forme.

DÉMONSTRATION. On note $G = (V, E)$ le graphe d'implication. Nous avons l'invariant suivant : **le graphe d'implication est acyclique**. En effet, le graphe est initialement vide donc acyclique, puis le graphe reste acyclique si on ajoute un noeud de décision, et faire une propagation unitaire. Nous considérons alors un tri topologique, i.e. on peut ordonner les sommets comme suit : $v_1, \ldots, v_{|V|}$ avec $v_k \to v_{k'}$ implique $k < k'$.

Par l'absurde, supposons qu'il n'y a pas de tel i_0. Mais alors on a pour tout $i \in \mathbb{N}$, $c_{i+1} = c_i \bigodot reason(\ell_i)$ pour un certain noeud ℓ_i.

On représente une clause c par un vecteur $(\beta_{|V|}, \ldots, \beta_1) \in \{0,1\}^{|V|}$ où $\beta_k = \begin{cases} 1 \text{ si } \neg v_k \text{ apparaît dans } c \\ 0 \text{ sinon} \end{cases}$. On dira que $c < c'$ si le vecteur de c est plus petit que le vecteur de c' pour l'ordre lexicographique sur les mots de longueur $|V|$ sur l'alphabet $\{0,1\}$.

Exemple 27 $011000 < 011100$.

Fait 28 On a $c_{i+1} < c_i$.

DÉMONSTRATION. $c_{i+1} := c_i \bigodot reason(\ell_i)$. En notant $c_i := \neg\ell_i \vee c_i'$ et $reason(\ell_i) := \ell_i \vee r$ où c_i' et r sont des clauses, on a

$$c_{i+1} = c_i' \vee r.$$

ℓ_i est un certain v_k. Les négations des littéraux dans r sont des $v_{k'}$ avec $k' < k$. Ainsi, $c_{i+1} < c_i$:

$$\begin{array}{lccc} & & v_k & \\ c_i & = (xxxxxxx & 1 & yyyyyyy) \\ c_{i+1} & = (xxxxxxx & 0 & zzzzzzz) \end{array}$$

■

Contradiction car nous avons la suite $(c_i)_{i \geq 0}$ qui est strictement décroissante dans l'ensemble $\{0,1\}^N$. ■

8.2 UIP (*Unit Implication Points*)

Définition 29 Un UIP est un sommet u tel que tout chemin du nombre de décision au niveau n à $\perp$ contient u.

8.3 Pseudo-code

fonction *backtrack* − *learning*(ν, φ)
  si pas de noeuds de décision dans ν **alors**
    renvoyer $\times$
  c_L = calculer la limite de $(c_i)_{i\in\mathbb{N}}$
  n := le niveau maximum en c_L
  n' := le deuxième niveau maximum dans c_L
  ν' := ν où on a oublié tous les nœuds du niveau $> n'$
  renvoyer $\nu', \varphi \wedge c_L$

Voir demo ici : `https://francoisschwarzentruber.github.io/dpll/`

9 Démonstrations

This subsection is inspired from Chapter 4 of [ZM03].

Proposition 30 $\models (\varphi \to c_L)$.

Démonstration. We prove that $\models \varphi \to c_n$ for all $n \in \mathbb{N}$. Let us do it by recurrence on n. Let P_n be the following property :

$$\models \varphi \to c_n.$$

— the conflicting clause c_0 is already a clause of φ so P_0 is true ;
— Suppose P_n. Let us prove that P_{n+1}. We have $\models \varphi \to c_n$. If $c_{n+1} = c_n$, P_{n+1} is true. Otherwise, $c_{n+1} = c_n \odot reason(\ell)$ where $\neg\ell$ is in c_n and ℓ is a deduced literal at the current decision level.
Let ν be a (total) valuation such that $\nu \models \varphi$. We have $\nu \models c_n$ by P_n. We have $\nu \models reason(\ell)$ as $reason(\ell)$ is a clause of φ. Therefore, by Proposition 20, $\nu \models c_{n+1}$. Hence, we proved that $\models \varphi \to c_{n+1}$. The property P_{n+1} is true.
We proved the proposition by recurrence.
■

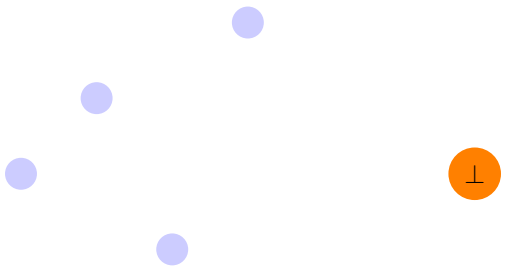
Corollaire 31 $\models (\varphi \leftrightarrow (\varphi \wedge c_L))$.

Proposition 32 If the procedure returns *true*, then the initial formula φ is satisfiable.

Démonstration. The algorithm returns true when a (partial) valuation ν such that $\nu \models \varphi$ is found. ν can be extended into a total valuation. Therefore, the formula φ is satisfiable. ■

Proposition 33 If the procedure returns $false$, then the initial formula φ is unsatisfiable.

Démonstration. If the procedure returns $false$ then $\nu = \times$ is reached. $\nu = \times$ is reached because $backtrack - learning(\nu, \varphi)$ returns a $\nu = \times$. This means that a conflict was detected and no literals are chosen. At this stage, the implication graph is full of deduced literals at level 0 and a $\bot$ node.



By contradiction, suppose that the initial formula φ^{ini} is satisfiable. Therefore, by proposition 27, the current formula φ, that is the conjunction of φ^{ini} and all learned clauses is also satisfiable. Let ν be a (total) valuation such that $\nu \models \varphi$.

Lemme 34 All added literals ℓ in the implication graph are such that $\nu \models \ell$.

Démonstration. They are added via unit propagation. For the first added literal ℓ, ℓ itself should be a clause in φ. Thus, $\nu \models \ell$. Then at each step, such that $\nu \models \ell'$ for all ℓ' already in the implication graph. Let ℓ be the next added literal. We should have a clause $\ell_1 \vee \cdots \vee \ell_k \vee \ell$ of φ such that $\neg\ell_1, \ldots, \neg\ell_k$ are in the implication graph. As $\nu \models \varphi$ and $\nu \models \neg\ell_i$ we have $\nu \models \ell$. ■
Therefore, the conflict clause c is such that $\nu \models \neg c$. So $\nu \not\models \varphi$. There is a contradiction. ■

Proposition 35 The call $cdcl(\epsilon, \varphi)$ terminates.

DÉMONSTRATION. By contradiction. Suppose that $cdcl(\epsilon, \varphi)$ does not terminate. We exhibit a strictly increasing sequence $(x^{(t)})_{t\in\mathbb{N}}$ of elements in a finite set with a strict total order. This leads to a contradiction.

Definition of $(x^{(t)})_t$. Given a partial valuation ν (where assigned litterals have levels), let $k(\nu, i)$ be the number of assigned (chosen or deduced) literals at level i in ν. Let us consider the t^{th} call of $cdcl$. Let

$$x^{(t)} = (k(\nu_t, 0), \ldots, k(\nu_t, n)) \quad \in \quad \{0, \ldots, n\}^{n+1}$$

where ν_t is the valuation at t^{th} call of $cdcl$ after saturation by unit propagation.

Evolution of $x^{(t)}$. There are two reasons for a $t+1^{th}$ call of $cdcl$:

— Either we chose a new literal. For instance :

before	after
$\neg p$ q r $\neg s$ t u v w a	$\neg p$ q r $\neg s$ t u v w a e

In the example :

$x^{(t)} = (0, 3, 2, 1, 1, 2, 0, 0, \ldots)$;
$x^{(t+1)} = (0, 3, 2, 1, 1, 2, \mathbf{1}, 0, \ldots)$.

Another example with some unit propagation after the choice :

before	after
$\neg p$ q r $\neg s$ t u v w a	$\neg p$ q r $\neg s$ t u v w a e z b

In the example :

$x^{(t)} = (0, 3, 2, 1, 1, 2, 0, 0, \ldots)$;
$x^{(t+1)} = (0, 3, 2, 1, 1, 2, \mathbf{3}, 0, \ldots)$.

Generally speaking,

$$\begin{aligned} x^{(t+1)} &= (k(\nu_{t+1}, 0), \ldots, k(\nu_{t+1}, n)) \\ &= (k(\nu_{t+1}, 0), \ldots, k(\nu_{t+1}, i-1), k(\nu_{t+1}, i), 0, \ldots, 0) \text{ where } i \text{ is the level in } \nu_{t+1} \\ &= (k(\nu_t, 0), \ldots, k(\nu_t, i-1), \underbrace{k(\nu_{t+1}, i)}_{>0}, \ldots, 0) \\ &> (k(\nu_t, 0), \ldots, k(\nu_t, i-1), 0, \ldots, 0) \\ &= x^{(t)} \end{aligned}$$

— Or we performed backjumping. For instance :

before	after
$\neg p$ q r $\neg s$ t u v w a e b	$\neg p$ q r $\neg s$ t for sure a $\ldots$

In the example :

$x^{(t)} = (0, 3, 2, 1, 1, 2, 0, \ldots)$;
$x^{(t+1)} = (0, 3, > \mathbf{2}, 0, 0, \ldots)$.

We suppose we were at level i at time t and we reach level $i' < i$ at time $t+1$.

$$\begin{aligned} x^{(t)} &= (k(\nu_t, 0), \ldots, \quad , k(\nu_t, i'-1), k(\nu_t, i'), \ldots, k(\nu_t, i), 0, \ldots, 0) \\ x^{(t+1)} &= (k(\nu_t, 0), \ldots, \quad , k(\nu_t, i'-1), k(\nu_t, i') + \alpha, 0, \ldots, 0) \text{ where } \alpha > 0 \\ &> x^{(t)} \end{aligned}$$

Lexicographical order. That is why we introduce the lexicographic order $>$ on $\{0, \ldots, n\}^{n+1}$. Given $x, y \in \{0, \ldots, n\}^{n+1}$, we have $x > y$ iff there exists an integer $i \in \{0, \ldots, n\}$ such that $x_i > y_i$ and for all $j < i$, we have $x_j = y_j$.

Conclusion. The sequence $(x^{(t)})_t$ is strictly increasing and takes only a finite number of values. Contradiction.

■

10 Implementation details

10.1 Two literal watching

Le problème est de savoir efficacement quelles clauses sont unitaires. Notez que les clauses de la formule φ ne sont *pas* modifiées au cours de l'algorithme (les clauses sont simplement ajoutées). Mais les clauses elles-mêmes

ne sont pas modifiées (sinon le retour en arrière est trop difficile à mettre en œuvre). En particulier, les clauses unitaires sont des clauses telles que tous les littéraux sauf un ont été affectés à false mais ce ne sont pas des clauses avec un seul littéral.

Exemple 36 If we consider the clause $p \vee q \vee \neg s$ and we set p to false, the clause is *not* replaced by $q \vee \neg s$.

In each clause c, two unassigned literals ℓ_1, ℓ_2 are selected. We say that the two unassigned literals (occurences of literals) are *watched*. The implementation works as follows. For each atomic proposition p, we maintain a list of clauses in which some occurrence of p is watched and a list of clauses in which some occurrence of $\neg p$ is watched.

- When some clause is added, we select randomly two unassigned literals in it ;
- When some atomic proposition p is assigned, we browse all the clauses c in which p or $\neg p$ is watched :
  - if all literals except one in c are assigned (to false), then the last litteral is assigned to true ;
  - if all literals in c are assigned to false, then we backtrack ;
  - if one litteral in c is assigned to true, we continue ;
  - otherwise, let ℓ be a unassigned litteral in the clause c, make ℓ to be watched and the litteral about p unwatched.

	Naive implementation with counters	VS	2 literal watching
When a clause is added	We update counter for that clause		We select randomly two unassigned literals in it
When a literal is set	We update counters for all clauses		We update the watched literals
When backtrack	We recompute the counters		We do nothing !

Proposition 37 (invariant) For all clauses c, if c contains more than 2 unassigned literals, then the two watched literals are unassigned.

In practice, if c is a clause, watched literals are $c[0]$ and $c[1]$ (and we swap literals to garantee this property). If p is an atomic proposition, $p.positiveWatches$ is an array of clauses where p appears positively and $p.negativeWatches$ is an array of clauses where p appears negatively.

Exemple 38

Operations	Valuations	My clause				
		👁 p	👁 q	r	s	
We set r	r	👁 p	👁 q	r	s	
We set $\neg p$	$r, \neg p$	👁 q	👁 r	p	s	
We set $\neg q$	$r, \neg p, \neg q$	👁 r	👁 s	p	q	
Backtrack 2 steps	r	👁 r	👁 s	p	q	(we do nothing)

10.2 Efficient algorithm for computing FUIP

See Javascript code of `sat.js`.

11 Améliorations

11.1 Redémarrage

- redémarrer après un certain nombre de backtracking, avec un seuil qui augmente (sinon on ne serait pas complet !)

11.2 Heuristiques

- Choix de la variable à décider, qui apparaît le plus fréquemment dans les clauses (Chaff)
- priorité aux variables qui apparaissent dans les clauses les plus récentes
- utilisation du machine learning

11.3 Suppression de clauses apprises

On peut apprendre un nombre exponentiel de clauses...
— On apprend que des clauses assez courtes (moins de n littéraux où n est fixé)
— On oublie une clause dès qu'au moins m littéraux sont non assignés

12 Recherche locale

fonction GSAT(φ)
  ν := valuation au hasard
  pour $i := 1$ à N **faire**
    si $\nu \models \varphi$ **alors renvoyer** ν
    v := variable parmi lesquelles, l'échange de la valeur $\nu(v)$ donne le plus de clauses satisfaites
    $\nu(v) := 1 - \nu(v)$
  renvoyer échec

Définition 39 Le break-count d'une variable est le nombre de clauses sont actuellement satisfaites mais qui deviennent insatisfaites si la variable est flipée.

fonction walkSAT(φ)
  ν := valuation au hasard
  pour $i := 1$ à N **faire**
    si $\nu \models \varphi$ **alors renvoyer** ν
    C := une clause non satisfaite choisie au hasard
    si il existe une variable x dans C avec break-count = 0 **alors** $v := x$
    sinon
      avec probabilité p : v := une variable de C au hasard
      avec probabilité $1 - p$: v := une variable de C avec break-count le plus petit
    $\nu(v) := 1 - \nu(v)$
  renvoyer échec

13 Notes bibliographiques

Les éléments techniques de ce cours viennent de [KS08] et [Har09]. La terminaison de l'algorithme CDCL est très bien expliquée dans la thèse [ZM03].

Références

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures*, volume 5. Springer, 2008.

[ZM03] Lintao Zhang and Sharad Malik. *Searching for truth : techniques for satisfiability of boolean formulas.* PhD thesis, Princeton University Princeton, 2003.

TD1 : solveurs SAT

1. Clause apprise

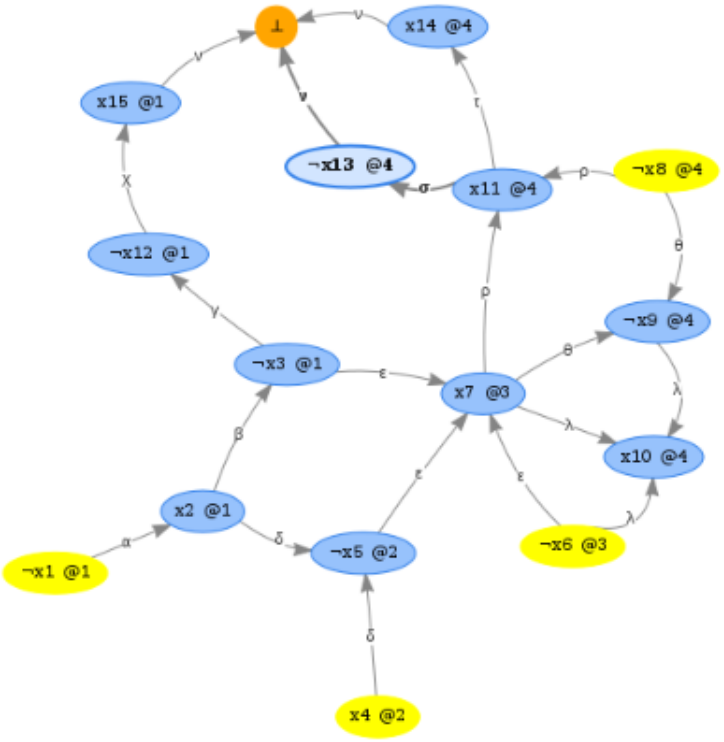
Soit n le niveau actuel. On considère un graphe d'implication $G = (V, E)$ avec une contradiction. On dit qu'une clause est de la bonne forme quand un littéral est de niveau n et les autres sont de niveau strictement inférieur.
On note $reason(\ell)$ la raison du littéral ℓ, autrement dit la clause qui a fait que le littéral a été ajouté. On note $c_1 \bigodot c_2$ une clause obtenue par résolution appliquée à c_1 et c_2.
Nous définissons la suite $(c_i)_{i\in\mathbb{N}}$ de clauses suivante :
$c_0 =$ la clause conflictuelle, i.e. la clause qui a donné la contradiction.

$$c_{i+1} = \begin{cases} c_i & \text{si } c_i \text{ est de la bonne forme} \\ c_i \bigodot reason(\ell) & \text{où } \ell@n,\ \neg\ell \in c_i \text{ et } reason(\ell) \text{ est défini} \end{cases}$$

La clause apprise est la limite de c_i.

1.1. Expliquer pourquoi $(c_i)_{i\in\mathbb{N}}$ est bien définie.

1.2. Appliquer l'algorithme sur le graphe d'implication suivant (le même que celui en cours) en explicitant les résolutions :



Le reste de l'exercice consiste à montrer que l'algorithme réussit, autrement que la suite c_i admet une limite. On procède par l'absurde.

1.3. Expliquer pourquoi le graphe d'implication est acyclique.

1.4. Expliquer pourquoi on peut ordonner les sommets de V comme suit : $v_1, \ldots, v_{|V|}$ avec $v_k \to v_{k'}$ implique $k < k'$.

On représente une clause c par un vecteur $(\beta_{|V|}, \ldots, \beta_1) \in \{0,1\}^{|V|}$
où $\beta_k = \begin{cases} 1 \text{ si } \neg v_k \text{ apparaît dans } c \\ 0 \text{ sinon} \end{cases}$. On dira que $c < c'$ si le vecteur de c est plus petit que le vecteur de c' pour l'ordre lexicographique sur les mots de longueur $|V|$ sur l'alphabet $\{0,1\}$.

1.5. Montrer que $c_{i+1} < c_i$.

1.6. Conclure.

1.7. Ecrire un pseudo-code pour la fonction $backjumping(\nu, \phi)$.

2. Correction de l'algorithme CDCL

2.1. Montrer que $\phi \to c_L$ est valide, où c_L est la clause apprise.
Indice : utiliser la définition de $(c_i)_{i\in\mathbb{N}}$.

2.2. Expliquer que si *cdcl* renvoie vraie, alors la formule ϕ initiale est satisfiable.

Le reste de l'exercice consiste à démontrer que si *cdcl* renvoie faux, alors la formule ϕ_{ini} initiale n'est pas satisfiable. On procède par l'absurde. Supposons que la formule initiale ϕ_{ini} soit satisfiable. Elle admet donc une valuation totale ν qui la satisfait. Regardons maintenant le graphe d'implication juste avant de renvoyer faux.

2.3. Y-a-t-il des littéraux choisis ? De quel niveau sont les littéraux présents ?

2.4. Montrer que pour tout littéral ℓ présent, on a $\nu \models \ell$.

2.5. Conclure.

3. Terminaison de CDCL

Supposons par l'absurde que l'appel $cdcl(\epsilon, \phi)$ ne termine pas. Etant donné une valuation partielle ν (i.e. un graphe d'implication), soit $k(\nu, i)$ le nombre de littéraux (choisis et déduits par propagation unitaire) de niveau i dans ν.
Considérons le t-ième appel de *cdcl*. On pose :

$$x^{(t)} = (k(\nu,0), \ldots, k(\nu,n)) \quad \in \quad \{0,\ldots,n\}^{n+1}$$

où ν est la valuation, après les propagations unitaires du t-ième appel de *cdcl*.

3.1. Étudier l'évolution de la suite $(x^{(t)})_{t\geq 0}$. En particulier exhiber une propriété en faisant apparaître un ordre $>$ pertinent sur $\{0,\ldots,n\}^{n+1}$.

3.2. Conclure.

Dualité en programmation linéaire

François Schwarzentruber

20 mars 2024

1 Certificat d'optimalité (15min)

Est-ce que vous aimez appliquer l'algorithme du simplexe ? Bof. Un ordinateur aime bien mais pas tant que ça. Ca prend quand même du temps. Regardons cet exemple.

Exemple 1 Considérons le programme linéaire suivant :

$$\text{maximiser } x_1 + 6x_2$$
$$\begin{cases} x_1 \leq 200 & (1) \\ x_2 \leq 300 & (2) \\ x_1 + x_2 \leq 400 & (3) \\ x_1 + x_2 \geq 0 \end{cases}$$

Question. Si je vous dis que $(x_1, x_2) = (100, 300)$ est une solution optimale avec un objectif de 1900. Me croyez-vous ?

Certes, vous pouvez me croire que $(100, 300)$ fait partie de l'espace des solutions : on vérifie que $(100, 300)$ satisfait les contraintes et que l'objectif vaut 1900. Mais comment être sûr que $(100, 300)$ est une solution optimale ? Certes, vous pouvez lancer l'algorithme du simplexe que vous aimez tant, mais je crois que je ne fais pas des heureux.ses.

Solution. On peut additionner des contraintes pour obtenir l'objectif $x_1 + 6x_2$. La partie droite, elle, donne alors une borne supérieure de $x_1 + 6x_2$.

- Par exemple : $(1) + 6(2)$ donne $x_1 + 6x_2 \leq 200 + 300 \times 6 = 2000$.
- Mieux : $0(1) + 5(2) + 1(3)$ donne $x_1 + 6x_2 \leq 5 \times 300 + 400 = 1900$. Le fait que $x_1 + 6x_2 \leq 1900$ me certifie que 1900 qui est atteint pour $(100, 300)$ est l'optimum.

Idée de généralisation. On cherche calculer une combinaison des contraintes $y1(1) + y_2(2) + y_3(3)$ telle que :

- la partie de gauche soit égal (ou supérieure) à $x_1 + 6x_2$;
- la partie de droite $200y_1 + 300y_2 + 400y_3$ soit minimale.

Autrement dit : S'occuper de minimiser $200y_1 + 300y_2 + 400y_3$ sous la contrainte

$$\underbrace{(y_1 + y_3)x_1 + (y_2 + y_3)x_2}_{\geq x_1 + 6x_2 ?} \leq x_1y_1 + x_2y_2 + (x_1 + x_2)y_3 \leq 200y_1 + 300y_2 + 400y_3$$

Mais du coup, quid de formuler ce problème en le programme linéaire suivant ?

$$\text{minimiser } 200y_1 + 300y_2 + 400y_3$$
$$\begin{cases} y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \\ y_1, y_2, y_3 \geq 0 \end{cases}$$

Le programme linéaire obtenu s'appelle le *dual*. Le programme initial s'appelle le *primal*.

programme primal

$$\text{maximiser } x_1 + 6x_2$$
$$\begin{cases} x_1 \leq 200 & (1) \\ x_2 \leq 300 & (2) \\ x_1 + x_2 \leq 400 & (3) \\ x_1 + x_2 \geq 0 \end{cases}$$

programme dual

$$\text{minimiser } 200y_1 + 300y_2 + 400y_3$$
$$\begin{cases} y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \\ y_1, y_2, y_3 \geq 0 \end{cases}$$

Quand une solution y du dual a un objectif qui est égal à l'objectif pour une solution x du primal, on a dit que la solution y est un *certificat d'optimalité* de la solution x. Ici, la solution du programme dual $y = (0, 5, 1)$, est un *certificat d'optimalité* pour $x = (100, 300)$.

Avec une écriture matricielle, on se rend compte de quelque chose de magique :

programme primal

$$\text{maximiser } \begin{pmatrix}1 & 6\end{pmatrix}^t \begin{pmatrix}x_1 & x_2\end{pmatrix}$$
$$\left\{ \begin{pmatrix}1 & 0\\ 0 & 1\\ 1 & 1\end{pmatrix} \begin{pmatrix}x_1\\ x_2\end{pmatrix} \leq \begin{pmatrix}200\\ 300\\ 400\end{pmatrix} \right.$$

programme dual

$$\text{minimiser } \begin{pmatrix}200 & 300 & 400\end{pmatrix} \begin{pmatrix}y_1 & y_2 & y_3\end{pmatrix}$$
$$\left\{ \begin{pmatrix}1 & 0 & 1\\ 0 & 0 & 1\end{pmatrix} \begin{pmatrix}y_1\\ y_2\\ y_3\end{pmatrix} \geq 1 \begin{pmatrix}1\\ 6\\ 0\end{pmatrix} \right.$$

TODO: ajouter une interprétation économique ?

TODO: expliquer pourquoi $y_1 = 0$?

Les coefficients de l'objectif du primal	deviennent	la partie droite des contraintes
la partie droite des contraintes	devient	les coefficients de l'objectif du dual
La matrice des contraintes	se transpose !	

2 Problème dual (5min)

Définition 2 (programme dual) A tout programme linéaire, appelé *programme primal*, avec n variables et m contraintes, on associe un programme linéaire, appelé le *programme dual*, en utilisant cette recette de cuisine :

	Programme primal	Programme dual
Variables	$x_1, \ldots, x_n$	$y_1, \ldots, y_m$
Matrice	A	A^t
Vecteur de droite	b	c
Fonction objectif	**maximiser** $c^t x$	**minimiser** $b^t y$
Constraintes	la i-ème contrainte a $\leq$	$y_i \geq 0$
	la i-ème contrainte a $\geq$	$y_i \leq 0$
	la i-ème contrainte a $=$	$y_i \in \mathbb{R}$
	$x_j \geq 0$	j-ème contrainte a $\geq$
	$x_j \leq 0$	j-ème contrainte a $\leq$
	$x_j \in \mathbb{R}$	j-ème contrainte a $=$

Proposition 3 Le dual du dual de P est P.

Exercice 1 Donner le programme dual de

$$\text{maximiser } 5x + 3y$$
$$\left\{ \begin{array}{l} -5x + 2y \leq 0 \\ x + y \leq 7 \\ x \leq 5 \\ x, y \geq 0 \end{array} \right.$$

3 Théorèmes de dualité (10min)

Considérons un programme primal P et son programme dual D :

Programme primal P
maximiser $c^t x$
$$\left\{ \begin{array}{l} Ax \leq b \\ x \geq 0 \end{array} \right.$$

Programme dual D
minimiser $b^t y$
$$\left\{ \begin{array}{l} A^t y \geq c \\ y \geq 0 \end{array} \right.$$

3.1 Dualité faible

Théorème 4 (de dualité faible) Pour toute solution x de P, pour toute solution y de D, $c^t x \leq b^t y$.

Démonstration.

$$\begin{array}{lll} c^t x & = \sum_{j=1}^n c_j x_j & \\ & \leq \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij} y_i\right) x_j & \text{par } A^t y \geq c \\ & = \sum_{i=1}^m \sum_{j=1}^m a_{ij} x_j y_i & \\ & \leq \sum_{i=1}^m b_i y_i & \text{par } Ax \leq b \\ & = b^t y & \end{array}$$

■

3.2 Dualité forte

Théorème 5 (de dualité forte) On est dans l'une des quatre situations suivantes :

1. P et D n'admettent pas de solutions ;
2. P est non borné et D n'admet pas de solutions ;
3. P n'admet pas de solutions et D est non borné ;
4. P possède une solution optimale x^*, D possède une solution optimale y^* et $c^t x^* = b^t y^*$.

1. 2. 3. 4.

Exemple 6 Le cas peuvent arriver ! Par exemple :

maximiser 1
$\begin{cases} 0 \leq -1 \end{cases}$

minimiser -1
$\begin{cases} 0 \geq 1 \end{cases}$

Démonstration. Voici le tableau des notations utilisées dans la démonstration :

P	programme primal
D	programme dual
n	nombre de variables dans le programme primal
m	nombre de contraintes dans le programme primal
$c \in \mathbb{R}^n$	coefficient de l'objectif du primal dans le tableau initial
$b \in \mathbb{R}^m$	vecteur des biais dans le primal
$A \in \mathbb{R}^{m,n} = (a_{ij})_{i=1..m,j=1..n}$	matrice du primal
$x = (x_1, \ldots, x_n) \in \mathbb{R}^n$	solution du programme primal
$x_{n+1}, \ldots, x_{n+m}$	variables d'écart
$\bar{x} = (x_1, \ldots, x_{n+m}) \in \mathbb{R}^{n+m}$	solution du programme primal avec en plus les variables d'écart
$y^* \in \mathbb{R}^m$	candidat pour être solution optimale du dual
obj	fonction objectif du primal
obj^*	valeur maximale de la fonction objectif

On se ramène à ces quatre cas via le théorème de dualité faible. Par exemple, le cas P non borné et D possède une solution optimale est impossible. Il ne reste plus qu'à montrer le quatrième cas.

On considère le problème P suivant :

$$\text{maximiser } c^t x \quad \begin{cases} Ax \leq b \\ x \geq 0 \end{cases}$$

Supposons que P possède une solution optimale x^*, montrer que D possède une solution optimale y^* et que $c^t x^* = b^t y^*$. Considérons l'exécution de l'algorithme du simplexe sur le problème primal P. L'algorithme commence par mettre sous forme équationnelle. Si $x = (x_1, \ldots, x_n)$ alors on ajoute les variables d'écart $x_{n+1}, \ldots, x_{n+m}$ pour les m contraintes. Autrement dit, on a :

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j.$$

L'algorithme considère alors le tableau associé, puis exécute des pivotages. On considère une exécution qui termine, par exemple celle obtenue via règle de Bland. On atteint donc un tableau final de base B où la fonction objectif s'écrit :

$$obj^* + c'^t \bar{x}_N \text{ où les coordonnées du vecteur } c' \text{ sont négatives.}$$

On pose $\boxed{y_i^* = -c'_{n+i} \text{ pour tout } i \in \{1, \ldots, m\}}$.

On montre maintenant que y^* est solution du dual D et que $b^t y^* = obj^*$. Par le théorème de dualité faible, on aura que y^* est solution optimale du problème D

Pour cela, nous allons comparer deux manières différentes d'écrire la fonction objectif (qui dépend de $\bar{x}$) :

1. La première venant du tableau initial : $obj = c^t x = \sum_{j=1}^{n} c_j x_j$;
2. La deuxième venant du tableau final : $obj = obj^* + \sum_{i=1}^{n+m} c'_i x_i$

On a :

$$\begin{aligned}
\sum_{j=1}^{n} c_j x_j &= obj^* + \sum_{i=1}^{n+m} c'_i x_i \\
&= obj^* + \sum_{i=1}^{n} c'_i x_i + \sum_{i=n+1}^{n+m} c'_i x_i \\
&= obj^* + \sum_{i=1}^{n} c'_i x_i - \sum_{i=1}^{m} y_i^* x_{n+i} \\
&= obj^* + \sum_{i=1}^{n} c'_i x_i - \sum_{i=1}^{m} y_i^* (b_i - \sum_{j=1}^{n} a_{ij} x_j) \\
&= (obj^* - \sum_{i=1}^{m} y_i^* b_i) + \sum_{j=1}^{n} (c'_j + \sum_{j=1}^{n} a_{ij} y_i^*) x_j)
\end{aligned}$$

Comme l'égalité précédente est vraie quelque soit le vecteur x on a égalité du terme constant et des termes devant les x_j :

$$\begin{aligned}
obj^* &= b^t y^* \\
c_j &= c'_j + \sum_{j=1}^{n} a_{ij} y_i^*
\end{aligned}$$

La deuxième équation implique, comme c'_j que $\sum_{j=1}^{n} a_{ij} y_i^* \geq c_j$. Autrement dit y^* est bien solution du dual. ■

Corollaire 7 (coefficients magiques) Voici une solution optimale du problème dual :

exécuter l'algorithme du simplexe dans le problème primal
soit c' le vecteur des coefficients dans l'objectif du tableau final
y^* := opposés des coefficients dans c' des variables d'écart (coefficient nul si dans la base)

3.3 Théorème des écarts complémentaires

On peut résumer par

$$y_i \times (b_i - \sum_j a_{ij} x_j) = 0$$

le fait que soit la i-ème contrainte est une égalité, ou alors il n'y a pas de force appliqué à la bille qui provient de la i-ème face ($y_i = 0$). Le théorème suivant montre que c'est une équivalence.

Théorème 8 (des écarts complémentaires) Soit x une solution du primal P et y une solution du dual D.

x et y sont solutions optimales de respectivement P et D ssi

1. pour tout indice i de contraintes de P, $y_i \times (b_i - \sum_j a_{ij} x_j) = 0$;
2. et pour tout indice j de contraintes de D, $x_j \times (\sum_i a_{ij} y_i - c_j) = 0$.

Démonstration.

$$\begin{array}{c}
x \text{ et } y \text{ solutions optimales de leurs programmes respectifs} \\
\Updownarrow \\
c^t x = b^t y \text{ (dualité forte)} \\
\Updownarrow \\
c^t x = y^t A x = b^t y \text{ (voir démo du th. de dualité faible)}
\end{array}$$

$$\Updownarrow$$
$$(c^t - y^t A)^t x = 0 \text{ et } y^t(b - Ax) = 0 \text{ (réécriture algébrique)}$$
$$\Updownarrow$$
points 1 et 2
(car les coordonnées des vecteurs sont positives et $\sum$ nombres positifs $= 0$ implique que ces nombres $= 0$)

■

TODO: ça serait bien d'avoir une application ici

Corollaire 9 x et y sont solutions optimales de respectivement P et D ssi

1. $a_i x - b_i > 0$ implique $y_i = 0$ — Si ça ne touche pas, alors pas de force
2. $y_i > 0$ implique $a_i x - b_i = 0$ — Si ça touche, alors il y a une force
3. $c_j - y a_{.j} > 0$ implique $x_j = 0$
4. $x_j > 0$ implique $c_j - y a_{.j} = 0$

3.4 Interprétation physique du théorème de dualité forte (15min)

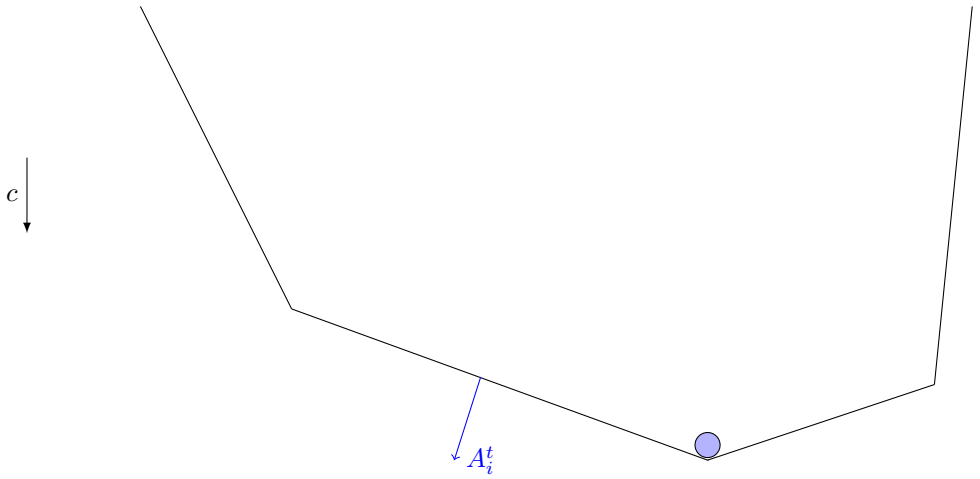
Plutôt que lire la démonstration du théorème de dualité forte, on peut en donner une interprétation physique, qui permet de mieux comprendre. Considérons les programmes primal et dual suivant.

Programme primal P
maximiser $c^t x$
$\begin{cases} Ax \leq b \\ x \geq 0 \end{cases}$

Programme dual D
minimiser $b^t y$
$\begin{cases} A^t y \geq c \\ y \geq 0 \end{cases}$

3.5 Laisser tomber une bille dans un polyèdre

Afin d'être plus concret, considérons que x est de dimension 2 ou 3, et que x est la position d'une bille à l'intérieur du polyèdre défini par $Ax \leq b$. On suppose que c est le vecteur correspondant à la force de gravité.



Ainsi maximiser $c^t x$ revient à laisser agir la gravité. La bille tombe. Une solution optimale x^* est la position de la bille une fois tombée dans un creux (généralement, intersection de 3 faces, même si elle peut aussi rester en plein milieu d'une face si jamais elle est horizontale).

3.6 Vecteurs normaux aux faces

Etant donné une face $i = 1..m$, la i-ème ligne $A_{i.}$ de la matrice A sont les coefficients de la face. Considérons cette ligne $A_{i.}$ comme un vecteur. On peut considérer les coefficients dans $A_{i.}$. Cela donne un vecteur qui est normal à la face. Comme on considère des vecteurs-colonnes, il s'agit de $A_{i.}^t$. Autrement dit, le vecteur $A_{i.}^t$ est normal à la face i.

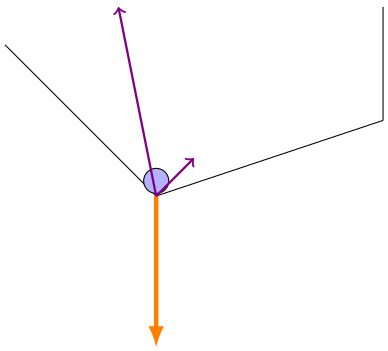
Ce vecteur pointe en dehors. En effet, si on est à l'intérieur $A_i^t x < b$, sur la face $A_i^t x = b$ et dehors $A_i^t x > b$.

3.7 Force

Soit N les indices des faces que la bille touche. Soit $i \in N$. Comme la bille touche la face i, la variable d'écart correspondante est nulle. Dit autrement, il y a égalité dans la i-ème ligne-contrainte du programme primal :

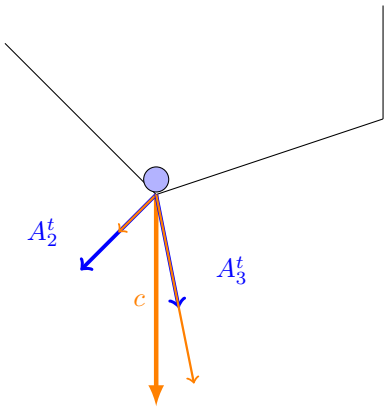
$$(Ax^*)_i = b_i \text{ pour tout } i \in N.$$

Faisons maintenant un peu de physique. La bille subit la gravité, mais aussi les forces qui retiennent la bille dans le polyèdre.



Les flèches violettes sont les forces de réaction de chaque face touchée. En orange, le vecteur de gravité.

La force de gravité F, proportionnelle à c. Elle se décompose sur les vecteurs normaux des faces touchées par la bille. De même pour c : il se décompose sur les vecteurs normaux des faces touchées.



Sur la figure, la bille touche les faces numéro 2 et 3. Donc c est combinaison linéaire de A_2^t et A_3^t.

Il existe donc $(y_i^*)_{i \in N}$ (la notation n'est pas hasardeuse, on verra que c'est un optimum du dual) telle que

$$c = \sum_{i \in N} y_i^* (A_{i.}^t)$$

et $y_i^* \geq 0$.

Si on pose $y_i^* = 0$ pour $i \notin N$ (bah oui, une face non touchée ne participe pas à la force), on a un vecteur $y^* \in \mathbb{R}^m$. On a :

$$c = \sum_{i=1}^{m} y_i^* (A_{i.}^t)$$

Autrement dit, $c = A^t y^*$ et donc y^* est dans l'espace des solutions du dual.

3.8 Bilan

Il nous rester à montrer que

$$c^t x^* = b^t y^*.$$

En effet, on a :

$$c^t x^* = (y^*)^t A x^* = \sum_{i=1}^{m} (y^*)_i (Ax^*)_i = \sum_{i \in N} (y^*)_i (Ax^*)_i + \sum_{i \notin N} \underbrace{(y^*)_i}_{0} (Ax^*)_i = \sum_{i \in N} (y^*)_i b_i + \sum_{i \notin N} \underbrace{(y^*)_i}_{0} b_i = b^t y^*.$$

Bref, on a 'montré manière physique' le théorème de dualité forte.

4 Motivation (20min)

4.1 Théorie de la complexité

Définition 10 Le problème LP est défini par :

- entrée : un programme linéaire $\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \\ x \geq 0 \end{cases}$, un seuil λ
- sortie : oui si le programme linéaire admet une solution dont l'objectif est $\geq \lambda$; non sinon.

Théorème 11 LP est dans NP.

Démonstration.

procédure algoNonDetLP($\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \\ x \geq 0 \end{cases}$, un seuil λ)

mettre sous forme équationnelle
deviner une base B
on considère la solution basique $\bar{x}^*$ en B définie par :

$$\bar{x}^*_B = \bar{A}_B^{-1} b$$
$$\bar{x}^*_N = 0$$

si $\bar{c}^t x^* \geq \lambda$ alors accepter sinon rejeter

■

Théorème 12 LP est dans coNP.

Démonstration. Il s'agit de montrer que le problème $\overline{LP}$ est défini par :

- entrée : un programme linéaire $\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \\ x \geq 0 \end{cases}$, un seuil λ
- sortie : oui si le programme linéaire n'admet pas de solution dont l'objectif est $\geq \lambda$; non sinon.

procédure algoNonDet$\overline{LP}$($\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \\ x \geq 0 \end{cases}$, un seuil λ)

calculer le dual $\begin{cases} \text{minimiser } b^t y \\ A^t y \geq c \\ y \geq 0 \end{cases}$
Il est équivalent à
$\begin{cases} \text{maximiser } -b^t y \\ -A^t y \leq -c \\ y \geq 0 \end{cases}$
faire la première phase de l'algorithme du simplexe pour se ramener à un programme P avec une solution basique
algoNonDetLP(P)

■

4.2 Outil pour démontrer et comprendre

La dualité est utile *conceptuellement*. Elle permet de comprendre que le lien entre deux problèmes algorithmiques (par exemple : flot max et coupe min). Elle permet de *démontrer* des résultats de dualité de manière élégante (deux problèmes sont en duaux l'un de l'autre). Elle offre une *interprétation économique et physique* de la programmation linéaire.

4.3 Dual simplex method

Elle offre aussi la possibilité de construire des algorithmes spécialisés efficaces. Le plus connu est le *dual simplex method*. En gros, il s'agit d'appliquer l'algorithme du simplexe sur le problème dual au lieu du programme primal. Pourquoi est-ce mieux ?

- Dans le primal, il n'y a pas forcément de solution basique. Mais comme souvent dans l'objectif, les coefficients sont positifs, $c \geq 0$, le dual a souvent une solution basique.
- On peut gérer l'ajout de contraintes à la volée. Dans l'algorithme du simplexe classique, l'ajout d'une contrainte, par exemple $x_1 + 3x_3 \leq 10$, peut juste faire que la solution courante n'est plus dans l'espace des solutions. En appliquant l'algorithme du simplexe sur le dual, l'ajout d'une contrainte correspond à ajouter une variable y_{new} et $10y_{new}$ dans l'objectif, et une colonne à la matrice (i.e. des nouveaux termes $cte \times y_{new}$ dans certaines contraintes du dual). Ce n'est pas dangereux car on considère la variable y_{new} comme nulle.

4.4 Autres algorithmes

Elle intervient dans d'autres algorithmes comme : transportation simplex method, l'algorithme primal-dual, l'algorithme hongrois (pour couplage pondéré dans un graphe biparti), network simplex method.

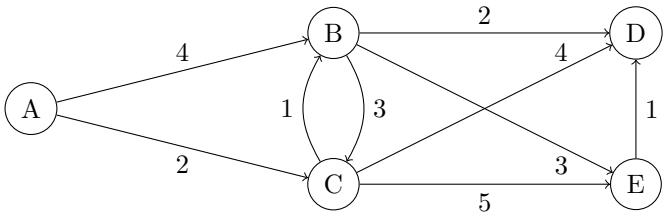
4.5 Algorithme d'approximation

Ca permet de faire des algorithmes d'approximation (cf. Section 12.3 dans le livre *approximation algorithms*).

5 Plus court chemin : dualité et système physique (30min)

Définition 13 **Problème d'un plus court chemin d'une source à une destination**

entrée : graphe $G = (S, A, poids)$ pondéré positivement, deux sommets $s, t \in S$
sortie : le poids d'un plus court chemin depuis s vers t ; $+\infty$ sinon.



5.1 Problème de minimisation

Le problème d'un plus court chemin d'une source à une destination se réduit à la programmation linéaire entière. Pour cela, on introduit des variables x_e qui indique si on prend un arc e ou pas. C'est-à-dire :

$$x_e = \begin{cases} 1 & \text{si } e \text{ est sélectionné comme faisant partie du plus court chemin} \\ 0 & \text{sinon.} \end{cases}$$

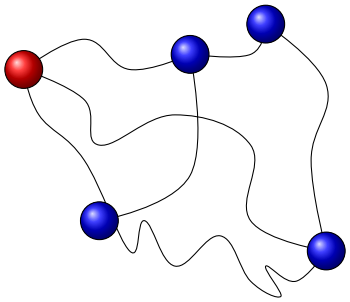
Ce sont des variables entières : un arc ne peut pas à moitié présent !

L'idée est alors de minimiser le poids des arcs sélectionnées, sous contrainte que l'ensemble des arcs sélectionnées constitue un chemin de s à t.

$$\begin{array}{l} \text{minimiser } \sum_{e \in E} x_e poids(e) \\ \begin{cases} \sum_{e \text{ entrant dans } s} x_e - \sum_{e \text{ sortant de } s} x_e & = -1 \\ \sum_{e \text{ entrant dans } t} x_e - \sum_{e \text{ sortant de } t} x_e & = 1 \\ \sum_{e \text{ entrant dans } u} x_e - \sum_{e \text{ sortant de } u} x_e & = 0 \text{ pour tout sommet } u \notin \{s, t\} \\ x_e \in \{0, 1\} \end{cases} \end{array}$$

5.2 Problème de maximisation

Aussi surprenant que cela puisse paraître, le problème du plus court chemin peut aussi se voir de manière naturelle comme un *problème de maximisation*. Pour cela, on construit un mobile. Les sommets du graphe sont des boules et les arcs sont des ficelles. La longueur d'une ficelle entre deux sommets est le poids de l'arc entre ces deux sommets.



L'idée est maintenant de pendouiller le mobile depuis la source. Certaines ficelles se tendent, d'autres non. Étonnamment, les arcs tendus correspondent aux arcs dans un plus court chemin sélectionné. On peut écrire les contraintes sous la forme d'un programme linéaire :

$$\begin{array}{l}\text{maximiser } y_t - y_s \\ \begin{cases} y_v - y_u \leqslant poids(u,v) \text{ pour tout arc } u \to v \\ y_u \in \mathbb{R} \text{ pour tout sommet } u \end{cases}\end{array}$$

5.3 Bilan

On se retrouve avec un programme primal et un dual. Enfin, pas tout à fait. On va voir qu'il y a un léger soucis car le programme de minimisation est un programme entier... Mais en fait, on va aussi parler d'une mini-théorie, la théorie des matrices totalement unimodulaires, qui permet de montrer dans certains cas (et là c'est le cas), que le programme entier est équivalent au programme réel (on parle de programme relaxé ou relâché). Pour le primal, j'ai écrit ici le programme relaxé.

Programme primal

$$\begin{array}{l}\text{minimiser } \sum_{e\in E} x_e poids(e) \\ \begin{cases} \sum_{e \text{ entrant dans } s} x_e - \sum_{e \text{ sortant de } s} x_e & = -1 \\ \sum_{e \text{ entrant dans } t} x_e - \sum_{e \text{ sortant de } t} x_e & = 1 \\ \sum_{e \text{ entrant dans } u} x_e - \sum_{e \text{ sortant de } u} x_e & = 0 \text{ pour tout sommet } u \notin \{s,t\} \\ x_e \geq 0 \text{ pour tout arc } e \end{cases}\end{array}$$

Programme dual

$$\begin{array}{l}\text{maximiser } y_t - y_s \\ \begin{cases} y_v - y_u \leqslant poids(u,v) \text{ pour tout arc } u \to v \\ y_u \in \mathbb{R} \text{ pour tout sommet } u \end{cases}\end{array}$$

Donnons la forme matricielle pour voir que l'on a le même schéma encore et toujours.

Programme primal

$$\begin{array}{l}\text{minimiser } poids^t x \\ \begin{cases} Ax \leq c \\ x_e \geq 0 \text{ pour tout arc } e \end{cases}\end{array}$$

Programme dual

$$\begin{array}{l}\text{maximiser } c^t y \\ \begin{cases} A^t y \leq poids \text{ pour tout arc } u \to v \\ d_u \in \mathbb{R} \text{ pour tout sommet } u \end{cases}\end{array}$$

où $poids$ est vu comme le vecteur colonne des poids $(poids(e))_{e\in E}$ où $c_u = \begin{cases} -1 & \text{si } u = s \\ 1 & \text{si } u = t \\ 0 & \text{sinon} \end{cases}$

$A_{ue} = \begin{cases} 1 & \text{si } e \text{ entre dans } u \\ -1 & \text{si } e \text{ sort de } u \end{cases}$

6 Couverture et packing

La dualité relie des problèmes de couvertures et de packing. Considérons le problème de couverture suivant :

$$\begin{array}{l}\text{minimiser } \sum_{i=1}^{n} cost(i)x_i \\ \{ \text{ pour tout } e \in E, \sum_{i=1..n|e\in S_i} x_i \geq 1, x_i \geq 0\end{array}$$

Intuitivement, $x_i = \begin{cases} 1 \text{ l'ensemble } S_i \text{ est sélectionné} \\ 0 \text{ sinon.} \end{cases}$. La contrainte dit que l'élément e doit être couvert.

Comme le programme n'est pas entier, x_i est la proportion de prendre l'ensemble S_i.

Le dual est :

$$\begin{array}{l}\text{maximiser } \sum_{e\in E} y_e \\ \{ \text{ pour tout } i = 1..n, \sum_{e\in E|e\in S_i} y_e \leq cost(i), y_e \geq 0\end{array}$$

Ce programme a une interprétation en terme de packing. On range des choses dans les éléments e. Intuitivement y_e = quantité de choses rangées dans l'élément e. On veut maximiser les choses rangées. La contrainte dit qu'on ne peut pas ranger plus que $cost(i)$ dans S_i.

7 Matrices totalement unimodulaires

On étudie ici un cadre où un programme linéaire réel à coefficients entiers admet une *solution optimale entière*. Autrement dit résoudre le programme linéaire entier revient à résoudre le programme linéaire réel. En particulier, la dualité fonctionne dans ce cas.

7.1 Définition

Définition 14 (matrice totalement unimodulaire) Une matrice est totalement unimodulaire si toute sous-matrice (en supprimant quelques lignes et/ou colonnes) carrée est de déterminant -1, 0 ou 1.

Proposition 15 Une matrice totalement unimodulaire ne contient que des -1, 0 ou 1.

Exemple 1 $\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$ est totalement unimodulaire.

Exemple 2 $\begin{pmatrix} 42 & -1 \\ -1 & 0 \end{pmatrix}$ et $\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ ne sont pas totalement unimodulaires.

7.2 Stabilité par ajout de vecteur unité

Proposition 16 Soit A une matrice totalement unimodulaire. La matrice $\bar{A}$ obtenue en ajoutant un vecteur unité e_i en dernière colonne est aussi totalement unimodulaire.

Démonstration. Calculer le déterminant d'une sous-matrice carrée en faisant un développement de Laplace selon la dernière colonne. ■

Exemple 3 $\begin{pmatrix} 1 & -1 & 0 & 1 \\ -1 & 1 & 1 & 0 \end{pmatrix}$ est totalement unimodulaire.

7.3 Solution optimale entière

Théorème 17 Considérons un programme linéaire $\begin{array}{l}\text{maximiser } c^t x \\ \begin{cases} Ax \leq b \\ x \geq 0 \end{cases}\end{array}$ où $b \in \mathbb{Z}^m$ et A totalement unimodulaire.
Si le programme admet une solution optimale, alors il admet une solution optimale entière $x^* \in \mathbb{Z}^n$.

Exemple 4 Si $\begin{array}{l}\text{maximiser } x + 3y - 5z \\ \begin{cases} x - y & \leq 30 \\ -x + y + z & \leq 42 \\ x, y, z \geq 0 \end{cases}\end{array}$ est borné, alors il admet une solution maximale entière.

Démonstration. Étudions l'exécution de l'algorithme du simplexe. Il travaille d'abord sur un programme équationnel où les contraintes sont $\bar{A}\bar{x} = b$ avec $\bar{A} = (A \mid Id_m)$ et $\bar{x} \in \mathbb{R}^{n+m}$. L'algorithme du simplexe trouve une base $B \in \{1, \ldots, n+m\}$ telle que la solution basique associée $\bar{x}^*$ soit définie par :

$$\begin{array}{ll} \bar{x}^*_B & = \bar{A}_B^{-1} b \\ \bar{x}^*_N & = 0 \end{array}$$

Les coefficients de $\bar{A}_B^{-1}$ s'écrivent, via les **règles de Cramer**, comme des fractions d'entiers avec au dénominateur $det(\bar{A}_B)$. Comme A est totalement unimodulaire, $\bar{A}$ l'est aussi par la proposition 16. Ainsi, $det(\bar{A}_B) \in \{-1, 0, 1\}$ (la valeur 0 est impossible car la matrice est inversible). ■

7.4 Conditions suffisantes

Voici des conditions suffisantes à ce qu'une matrice soit totalement unimodulaire.

Proposition 18 Soit A une matrice contenant seulement les éléments 0, 1 ou -1 et qui satisfait les 2 conditions suivantes :

1. Chaque colonne contient au plus 2 éléments non nuls ;
2. Les lignes de A peuvent être partitionnées en 2 sous-ensembles S_1 et S_2 tel que pour chaque colonne contenant 2 éléments non nuls :
   - Si les 2 éléments non nuls ont le même signe, alors l'un est dans S_1 et l'autre dans S_2 ;
   - Si les 2 éléments non nuls sont de signe différents, alors ils sont tous deux dans S_1, ou tous deux dans S_2.

Alors A est totalement unimodulaire.

Démonstration. On pose

$$\mathcal{P}(\ell) : \text{le déterminant de toute sous-matrice de taille } \ell \times \ell \text{ de } A \text{ est dans } \{-1, 0, 1\}.$$

que l'on va démontrer par récurrence. Le cas de base $\mathcal{P}(1)$ est ok par définition de A. Supposons que $\mathcal{P}(\ell-1)$. Montrons $\mathcal{P}(\ell)$. Considérons une sous-matrice Q de taille $\ell \times \ell$. Par la condition 1, chaque colonne de A contient au plus deux éléments non nuls. Ainsi, il en est de même pour chaque colonne de Q. S'il y a une colonne avec que des 0, $\det(Q) = 0$. S'il y a une colonne avec un seul élément non nul, on se ramène, à signe près, au calcul du déterminant d'une sous-matrice de A de taille $(\ell-1) \times (\ell-1)$, qui est par récurrence de déterminant dans $\{-1, 0, 1\}$. Enfin, toutes les colonnes de Q contiennent deux éléments non nuls, alors on montre que $det(Q) = 0$. En effet, les lignes sont dépendantes.

1. En sommant toutes les lignes d'indices S_1, on obtient un certain vecteur ;
2. Et en sommant les lignes d'indices S_2 on obtient le vecteur opposé, vu la condition 2.

■

8 Matching pondéré dans un graphe biparti

On avait vu que le problème de matching dans un graphe biparti se résolvait avec les flots max. Ici, pour un matching pondéré on va presque faire pareil, mais réduire au flot max de coût min.

8.1 Définition du matching pondéré dans un graphe biparti

Définition 19 Soit G un graphe biparti. Un couplage M est un sous-ensemble d'arêtes qui touche au plus une fois chaque sommet.

Définition 20 Soit G un graphe biparti où les deux ensembles disjoints de sommets sont de même cardinal n. Un couplage M est parfait s'il est de cardinal n.

Définition 21
- entrée : G un graphe biparti pondéré
- sortie : un couplage parfait de poids **minimal**.

TODO: donner un exemple d'application réel

8.2 Problème du flot de coût minimum

On considère un réseau de flots $G = (V, E, d, c, w, s, t)$ où :
- $d \in \mathbb{R}$ est un demande ;
- $c : E \to \mathbb{R}^+$ est la capacité (comme dans le problème du flot max)
- $w : E \to \mathbb{R}$ est le coût du transport d'une unité de flot pour un arc donné
- s sommet source,
- t somme puits

Le but est de savoir s'il existe un flot $f : E \to \mathbb{R}$ qui minimise le coût total $\sum_{e \in E} w(e)f(e)$ sous les contraintes de capacité $\forall e \in E, 0 \leq f(e) \leq c(e)$, de demande $\sum_{v \in V} f(s, v) = d$, et loi de Kirschoff $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

8.3 Programme linéaire pour le problème du flot de coût min

$$\text{minimiser } \sum_{e\in E} w(e)f(e)$$
$$\begin{cases} \forall e \in E, f(e) & \leq c(e) \\ \forall e \in V \setminus \{s,t\}, \sum_{e\in E e \text{ in } v} f(e) - \sum_{e\in E e \text{ out } v} f(e) & = 0 \\ \forall v \in V, \sum_{v\in V} f(s,v) & = d \\ \forall e \in E, f(e) \geq 0 \end{cases}$$

Proposition 22 Si les capacités sont entières, alors ce programme admet une solution optimale entière.

Démonstration. Car la matrice est totalement unimodulaire. ■

8.4 Réduction de matching pondéré dans flot de coût min

On fait comme on a toujours fait. :)

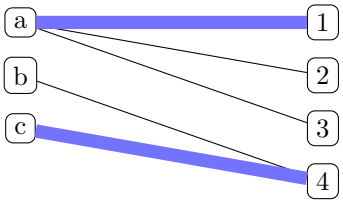
Remarque 23 On peut aussi écrire directement un programme linéaire.

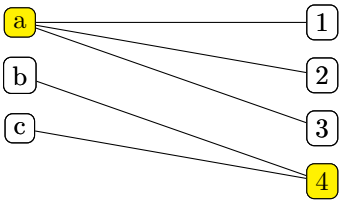
9 Théorème de König

TODO: mettre la motivation SWERC

Théorème 24 Dans un graphe biparti,

cardinal d'un couplage maximal = cardinal d'une couverture minimal de sommets.





Démonstration. Quitte à renommer les objets, on note $\{1,\dots,n\}$ l'ensemble des sommets, et $\{1,\dots,m\}$ l'ensemble des arêtes.

1) Programme linéaire pour le couplage maximal

$$\text{maximiser } \sum_{j=1}^{m} x_j$$
$$\begin{cases} \sum_{j \text{ incident à } i} x_j \leq 1 \text{ pour tout sommet } i = 1..n \\ x \geq 0 \\ x \in \mathbb{Z}^m \end{cases}$$

En posant $A \in \mathbb{R}^{n\times m}$ par $a_{ij} = \begin{cases} 1 & \text{si l'arc } j \text{ incidente au sommet } i \\ 0 & \text{sinon.} \end{cases}$, ce programme se réécrit en :

$$\text{maximiser } \sum_{j=1}^{m} x_j$$
$$\begin{cases} Ax \leq 1 \\ x \geq 0 \\ x \in \mathbb{Z}^m \end{cases}$$

2) Programme linéaire pour la couverture minimal

$$\text{minimiser } \sum_{i=1}^{n} y_i$$
$$\begin{cases} \sum_{j \text{ incident à } i} y_i \geq 1 \text{ pour toute arête } j = 1..m \\ y \geq 0 \\ y \in \mathbb{Z}^n \end{cases}$$

que l'on peut réécrire en

$$\text{minimiser } \sum_{i=1}^{n} y_i$$
$$\begin{cases} A^t y \geq 1 \\ y \geq 0 \\ y \in \mathbb{Z}^n \end{cases}$$

Si on avait "$\in \mathbb{R}^m$" et "$\in \mathbb{R}^n$" à la place "$\in \mathbb{Z}^m$" et "$\in \mathbb{Z}^n$", les programmes seraient duaux l'un de l'autre et cela conclurait la démonstration. Ici, on ne peut pas conclure car il n'y a pas de théorème de dualité pour la programmation linéaire entière. Mais heureusement, la proposition 18 conclut la démonstration ! En effet :

1. Chaque colonne (arête) est incidente à deux sommets ;
2. Les paquets S_1 et S_2 sont pile poil les deux patates de sommets du graphe biparti.

■

Exercice 25 Montrer que la matrice d'incidence signée du problème du plus court chemin (et aussi du problème de flots) est totalement unimodulaire. On notera qu'il n'y a pas besoin ici que le graphe soit biparti.

Notes bibliographiques

La dualité est évoquée dans [DPV08] et [CLRS09]. L'explication avec le certificat provient de [DPV08]. Les démonstrations et les applications proviennent essentiellement de [GM07].

Références

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[GM07] Bernd Gärtner and Jirí Matousek. *Understanding and using linear programming*. Universitext. Springer, 2007.

TD2 : Dualité en programmation linéaire

1. Coopérative viticole

Une coopérative viticole produit des vins par assemblage (en les mélangeant) à partir de 3 cépages. Voici les informations :

	Cabernet-franc	Cabernet sauvignon	Merlot
Stock	5000ℓ	1000ℓ	4000ℓ
Vin L'esperluette à 7€/ℓ	30%	20%	50%
Vin La cédille à 5€/ℓ	50%	10%	40%

1.1. Écrire un programme linéaire pour maximiser le profit, sachant que tout est vendu.

1.2. Écrire le dual.

1.3. Donner une interprétation économique d'un concurrent qui souhaite racheter tous les stocks de la coopérative.

1.4. Donner l'interprétation économique d'un programme quelconque $\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \end{cases}$ avec c, A, b à valeurs réelles positives, et de son dual.

2. Matrices totalement unimodulaires

Dans cet exercice, nous allons discuter de la notion de matrices totalement unimodulaires. Le but est de donner une condition suffisante pour qu'un programme linéaire entier et sa relaxation est le meilleur optimum.
Une matrice est totalement unimodulaire si toute sous-matrice (en supprimant quelques lignes et/ou colonnes) carrée est de déterminant -1, 0 ou 1.

2.1. Montrer que $\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$ est totalement unimodulaire.

2.2. Montrer que $\begin{pmatrix} 42 & -1 \\ -1 & 0 \end{pmatrix}$ et $\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ ne sont pas totalement unimodulaires.

2.3. Soit A une matrice totalement unimodulaire. Montrer que la matrice $\bar{A}$ obtenue en ajoutant un vecteur unité e_i en dernière colonne est aussi totalement unimodulaire.

Considérons un programme linéaire $\begin{cases} \text{maximiser } c^t x \\ Ax \leq b \\ x \geq 0 \end{cases}$ où $b \in \mathbb{Z}^m$ et A totalement unimodulaire.
Nous allons montrer que si le programme admet une solution optimale, alors il admet une solution optimale entière $x^* \in \mathbb{Z}^n$. Étudions l'exécution de l'algorithme du simplexe.

Il travaille d'abord sur un programme équationnel où les contraintes sont $\overline{A}\overline{x} = b$ avec $\overline{A} = (A \mid Id_m)$ et $\overline{x} \in \mathbb{R}^{n+m}$. L'algorithme du simplexe trouve une base $B \in \{1, \ldots, n+m\}$ telle que la solution basique associée $\overline{x}^*$ soit définie par :

$$\begin{aligned} \overline{x}^*_B &= \overline{A}_B^{-1} b \\ \overline{x}^*_N &= 0 \end{aligned}$$

2.4. Montrer que $\overline{x}^* \in \mathbb{Z}^{n+m}$.

3. Condition suffisante pour être une matrice totalement unimodulaire

Voici une condition suffisante pour qu'une matrice soit totalement unimodulaire.

Proposition 1 *Soit A une matrice contenant seulement les éléments 0, 1 ou -1 et qui satisfait les 2 conditions suivantes :*

1. *Chaque colonne contient au plus 2 éléments non nuls ;*
2. *Les lignes de A peuvent être partitionnées en 2 sous-ensembles S_1 et S_2 tel que pour chaque colonne contenant 2 éléments non nuls :*
   - *Si les 2 éléments non nuls ont le même signe, alors l'un est dans S_1 et l'autre dans S_2 ;*
   - *Si les 2 éléments non nuls sont de signe différents, alors ils sont tous deux dans S_1, ou tous deux dans S_2.*

Alors A est totalement unimodulaire.

3.1. Montrer la proposition. On pourra poser la propriété suivante :

$\mathcal{P}(\ell)$: le déterminant de toute sous-matrice de taille $\ell \times \ell$ de A est dans $\{-1, 0, 1\}$.

4. Flot maximum et coupe minimale

Dans un graphe orienté (V, E), un flot de $s \in V$ à $t \in V$ est une fonction $f : E \to \mathbb{R}^+$ vérifiant la condition de flot : le flot entrant et le flot sortant sont égaux dans les sommets de $V \setminus \{s, t\}$. La *valeur* du flot est alors la différence entre flot sortant et flot entrant en s. Le problème d'optimisation MAX-FLOW est le suivant :

MAX-FLOW

Entrée	–	Un graphe orienté $G = (V, E)$, $s, t \in V$, une capacité $c : E \to \mathbb{R}^+$
Sortie	–	Le flot f de s à t de valeur maximale tel que $f(u, v) \leq c(u, v) \, \forall (u, v) \in E$.

4.1. Reformuler le problème MAX-FLOW sous la forme d'un programme linéaire.

4.2. Montrer que, lorsque c est à valeurs dans $\mathbb{N}$, il existe un flot maximal à valeurs dans $\mathbb{N}$.

4.3. Calculer son dual. A quoi correspond-il ?

5. Jeux à somme nulle

On considère un jeu à deux joueurs (Xavier et Yolande), à somme nulle (ce que Xavier gagne Yolande le perd, et inversement) et en forme normale (les deux joueurs choisissent simultanément une action puis récupèrent les récompenses). On modélise un tel jeu par une matrice $G = (G_{ij})_{i,j\in\{1,\dots,n\}}$. Si Xavier choisit l'action i et Yolande choisit l'action j, alors Xavier gagne G_{ij} euros et Yolande perd G_{ij} euros.

5.1. Expliquer comment modéliser pierre-feuille-ciseaux.

Une stratégie mixte est une distribution de probabilité sur $\{1,\dots,n\}$, i.e. un vecteur dans

$$\Delta = \{x \in [0,1]^n \mid \sum_i x_i = 1\}.$$

Par exemple, avec $x = (0.3, 0.2, 0.5)$, Xavier choisit la ligne 1 avec une probabilité de 0.3, la ligne 2 avec une probabilité de 0.2 et la ligne 3 avec une probabilité de 0.5.

5.2. Que représente $\sum_{i,j} G_{ij}x_iy_j$?

Le reste de l'exercice consiste à démontrer le théorème suivant :

Théorème 1 (min-max de Von Neumann, 1926) *Soit $G = (G_{ij})_{i,j\in\{1,\dots,n\}}$ une matrice réelle. On a :*

$$\max_{x\in\Delta}\min_{y\in\Delta}\sum_{i,j} G_{ij}x_iy_j = \min_{y\in\Delta}\max_{x\in\Delta}\sum_{i,j} G_{ij}x_iy_j$$

5.3. Donner une interprétation du théorème.

5.4. Utiliser le théorème de dualité forte pour démontrer le théorème de Von Neumann.

6. Démonstration du théorème de dualité forte

Théorème 2 (de dualité forte) *On est dans l'une des quatre situations suivantes :*

1. *P et D n'admettent pas de solutions ;*
2. *P est non borné et D n'admet pas de solutions ;*
3. *P n'admet pas de solutions et D est non borné ;*
4. *P possède une solution optimale x^*, D possède une solution optimale y^* et $c^tx^* = b^ty^*$.*

6.1. Donner un exemple de programme primal et dual qui vérifie le cas 1 du théorème.

6.2. Montrer que si P est non borné alors D n'admet pas de solutions.

6.3. Montrer que si D est non borné alors P n'admet pas de solutions.

Le reste de l'exercice s'attaque au cas restant : le cas 4. Supposons que P et D admettent des solutions optimales. On utilisera ces notations :

P	programme primal
D	programme dual
n	nombre de variables dans le programme primal
m	nombre de contraintes dans le programme primal
$c \in \mathbb{R}^n$	coefficient de l'objectif du primal dans le tableau initial
$b \in \mathbb{R}^m$	vecteur des biais dans le primal
$A \in \mathbb{R}^{m,n} = (a_{ij})_{i=1..m,j=1..n}$	matrice du primal
$x = (x_1, \ldots, x_n) \in \mathbb{R}^n$	solution du programme primal
$x_{n+1}, \ldots, x_{n+m}$	variables d'écart
$\overline{x} = (x_1, \ldots, x_{n+m}) \in \mathbb{R}^{n+m}$	solution du programme primal avec en plus les variables d'écart
$y^* \in \mathbb{R}^m$	candidat pour être solution optimale du dual
obj	fonction objectif du primal
obj^*	valeur maximale de la fonction objectif

L'algorithme du simplexe exécute des pivotages. On considère une exécution qui termine, par exemple celle obtenue via règle de Bland. On atteint donc un tableau final de base B où la fonction objectif s'écrit :

$$obj^* + c'^t \overline{x}_N \text{ où les coordonnées du vecteur } c' \text{ sont négatives.}$$

On pose $\boxed{y_i^* = -c'_{n+i} \text{ pour tout } i \in \{1, \ldots, m\}}$.
On montre maintenant que y^* est solution du dual D et que $b^t y^* = obj^*$. Par le théorème de dualité faible, on aura que y^* est solution optimale du problème D.
Pour cela, nous allons comparer deux manières différentes d'écrire la fonction objectif (qui dépend de $\overline{x}$) :

1. La première venant du tableau initial : $obj = c^t x = \sum_{j=1}^{n} c_j x_j$;
2. La deuxième venant du tableau final : $obj = obj^* + \sum_{i=1}^{n+m} c'_i x_i$

6.4. Montrer que $\sum_{j=1}^{n} c_j x_j = (obj^* - \sum_{i=1}^{m} y_i^* b_i) + \sum_{j=1}^{n} (c'_j + \sum_{j=1}^{n} a_{ij} y_i^*) x_j)$.

6.5. Conclure.

7. Ecarts complémentaires

7.1. Montrer le théorème suivant :

Théorème 3 (des écarts complémentaires) *Soit x une solution du primal P et y une solution du dual D. On a que x et y sont solutions optimales de respectivement P et D ssi les deux conditions suivantes sont vraies :*

1. *pour tout indice i de contraintes de P, $y_i \times (b_i - \sum_j a_{ij} x_j) = 0$;*
2. *et pour tout indice j de contraintes de D, $x_j \times (\sum_i a_{ij} y_i - c_j) = 0$.*

Algorithmes d'approximation

François Schwarzentruber

Définition 1 Un *problème de minimisation* est de la forme

Problème de minimisation
entrée : instance x
sortie : une solution $sol^* \in Sol_x$ avec $sol^* = argmin_{sol \in Sol_x} c(x, sol)$

Problème de minimisation
entrée : instance x
sortie : $min_{sol \in Sol_x} c(x, sol)$

où Sol_x est l'espace des solutions de l'instance x et c est une fonction de coût.

Quand on dit qu'un problème d'optimisation est NP-complet, cela signifie que le problème de décision associé, est NP-complet :

Problème de décision associé
entrée : instance x, un nombre λ
sortie : oui s'il existe une solution $sol \in Sol_x$ avec $c(x, sol) \leq \lambda$

Un problème de maximisation, c'est la même chose mais avec $argmax$ et max. Un problème d'optimisation désigne à la fois un problème de minimisation ou de maximisation.

Définition 2 Étant donné un problème d'optimisation, un algorithme d'approximation calcule une solution quasi optimale.

Définition 3 Un algorithme d'approximation est de ratio ρ si, sur toute entrée de taille n,

$$\frac{\text{cout(solution calculée)}}{\text{cout(solution optimale)}} \leq \rho(n) \qquad \rho(n) \leq \frac{\text{cout(solution calculée)}}{\text{cout(solution optimale)}}$$

si problème de minimisation — si problème de maximisation

La quantité $\rho(n)$ s'appelle ratio, ou facteur d'approximation : c'est une garantie de performance de l'algorithme d'approximation.

1 Classification

Définition 4 (PTAS) Un *polynomial time approximation scheme* est un algorithme d'approximation qui prend en entrée une instance I et un nombre ϵ, et qui calcule une solution de ratio $\begin{cases} 1+\epsilon & \text{pour pb de min} \\ 1-\epsilon & \text{pour pb de max} \end{cases}$, et tel qu'en fixant ϵ, l'algorithme est en temps $poly(|I|)$.

Définition 5 (FPTAS) Un *fully polynomial time approximation scheme* est un PTAS en temps $poly(|I|, \frac{1}{\epsilon})$.

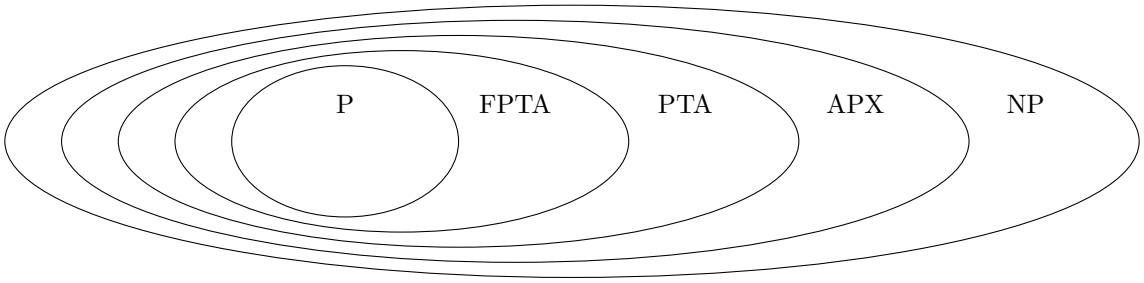
Remarque 6
- Avec PTAS, on peut avoir une complexité $2^{1/\epsilon} n^{1/\epsilon}$.
- Avec FPTAS, on ne peut pas mais on peut avoir $n^2 \frac{1}{\epsilon^3}$.

Définition 7 (Classes de complexité)

APX = problèmes d'optimisation admettant un algo d'approximation poly avec ratio constant

PTA = problèmes d'optimisation admettant un PTAS

FPTA = problème d'optimisation admettant un FPTAS



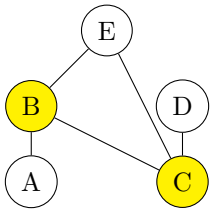
2 Couverture de sommets

Application 8 Placer le minimum de gardiens pour surveiller tous les couloirs (= arêtes).

Toute arête est surveillée.

Définition 9 (couverture de sommets) Soit un graphe $G = (V, E)$. Une couverture de sommet est un ensemble $C \subseteq V$ tel que pour toute arête $(u, v) \in E$, $u \in C$ ou $v \in C$.

Exemple 10



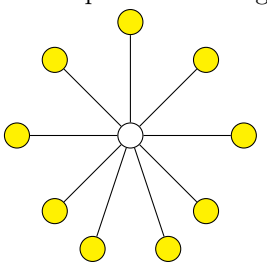
Définition 11 (problème d'optimisation) **VERTEX COVER**

entrée : un graphe $G = (V, E)$ non orienté
sortie : une couverture de sommets de cardinal minimal.

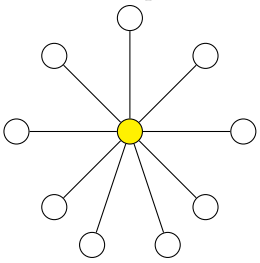
2.1 Algorithme naïf

Algo naïf
Ajouter des sommets
jusqu'à obtenir une couverture

Résultat possible de l'algo



Solution optimale



2.2 Couplage

Un sommet n'est marié à ≤ 1 autre sommet.

Définition 12 (couplage) Soit $G = (V, E)$ un graphe non orienté. Un ensemble $M \subseteq E$ est un couplage si pour tout sommet $u \in V$, il existe au plus une arête $(a, b) \in M$ avec $a = u$ ou $b = u$.

Proposition 13 $|M| \leq |C|$ pour tout couplage M, pour toute couverture C.

Démonstration. Tout couloir (arête) est vu par un certain gardien dans C. En particulier chaque couloir de M est vu par un gardien dans C. On peut donc considérer une fonction f de M dans C qui associe à toute arête $e \in M$ une des extrémités de e, qui est dans C :

$$\begin{array}{rccl} f: & M & \to & C \\ & e & \mapsto & \text{une des extrémités de } e. \end{array}$$

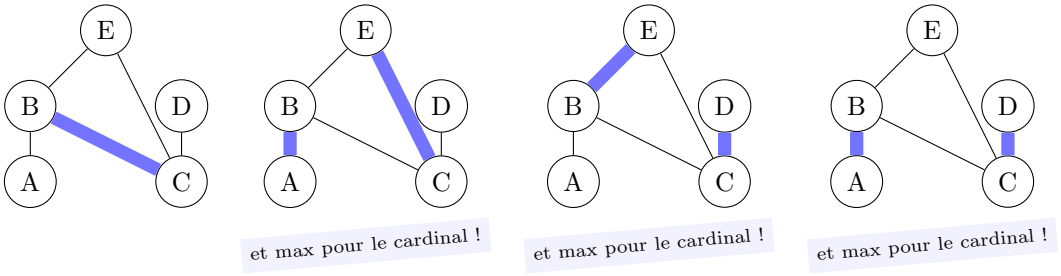
Comme M est un couplage, un gardien ne peut pas surveiller deux couloirs. On conclut car f est injective. En effet, soit e et e' deux arêtes dans M avec $f(e) = f(e')$. Cela signifie que $f(e)$ est une extrémité touchée à la fois par e et e'. Mais comme M est un couplage, on a $e = e'$. ■

2.3 Couplage maximal pour l'inclusion (couplage maximal)

Définition 14 (couplage maximum) Un couplage M est maximum si pour tout couplage M' on a $|M'| \leq |M|$.

Définition 15 (couplage maximal pour l'inclusion) Un couplage maximal pour l'inclusion est un couplage M tel que tout pour tout $M' \subseteq E$, $M \subsetneq M'$ implique M' n'est pas un couplage.

Exemple 16 (couplages maximaux pour l'inclusion)

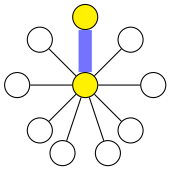


Proposition 17 Un couplage maximum pour le cardinal est maximal pour l'inclusion.

> Algo pour calculer un couplage maximal :
> ajouter des arêtes jusqu'à ne plus pouvoir

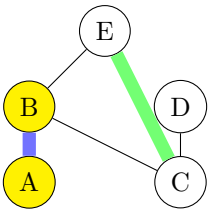
2.4 Algorithme d'approximation pour VERTEX COVER

> Algo d'approximation :
> $M :=$ un couplage max pour l'inclusion
> **renvoyer** $C :=$ l'ensemble des extrémités des arêtes de M



Proposition 18 C est une couverture de sommets.

Démonstration. Par l'absurde, supposons que C ne soit pas une couverture de sommets, i.e. il existe une arête e qui ne touche aucun sommet de C.



Mais alors $M \cup \{e\}$ est aussi un couplage, contredisant la maximalité pour l'inclusion de M. ■

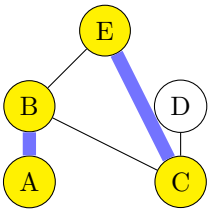
Proposition 19 VERTEX COVER admet un algorithme d'approximation est de ratio 2.

Démonstration. Soit C la couverture renvoyée par l'algorithme, et C^* une couverture minimale.

$$\frac{\dfrac{C = \bigsqcup_{e \in M} extremites(e)}{|C| = 2|M|} \qquad \dfrac{\text{Proposition 13}}{|M| \leq |C^*|}}{|C| \leq 2|C^*|}$$

■

Exemple 1 Le ratio 2 est atteint par cet algorithme, comme par exemple sur les graphes ci-dessous :



2.5 La dualité est affaiblie

Définition 20 (couplage maximum pour le cardinal) Un couplage M est maximum si $|M|$ maximal.

Proposition 21 Dans un graphe biparti, cardinal min d'une couverture = cardinal max d'un couplage.

Proposition 22 Dans un graphe qcq, card couplage maximum $\leq$ card min couverture $\leq 2\times$ card couplage maximum.

Aller plus loin

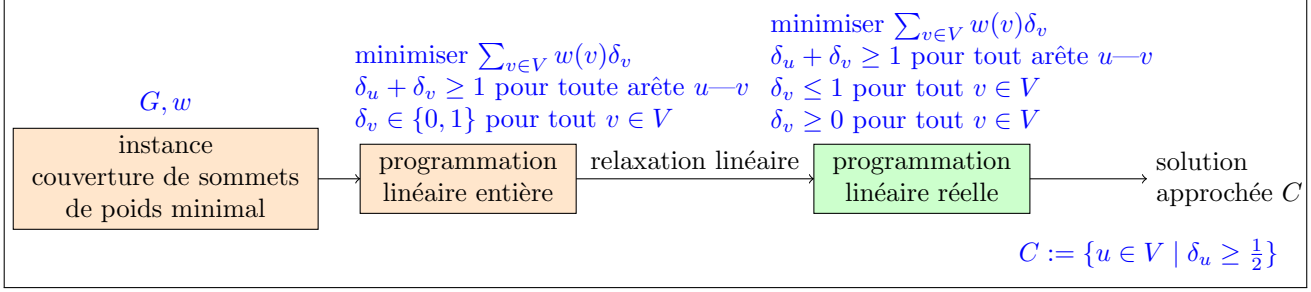
— algorithme de ratio $2 - \Theta\left(\frac{1}{\sqrt{\log |V|}}\right)$ [Kar09].
— Si $P \neq NP$, alors pas d'algo polynomial avec un facteur ≤ 1.3606. [DS05]

3 Couverture de sommets pondérée et relaxation linéaire

Définition 23 **problème de la couverture de sommets pondérée**

entrée : Un graphe non orienté $G = (V, E)$, des poids positifs $w(u)$ à chaque sommet $u \in V$;
sortie : une couverture de sommets $C^* \subseteq V$ tel que son poids $\sum_{v \in C^*} w(v)$ soit minimal.

Algorithme d'approximation



Exercice 1 Donner un exemple pertinent.

Proposition 24 C est une couverture de sommets.

Démonstration. La contrainte $\delta_u + \delta_v \geq 1$ pour tout arête u—v, force que $\delta_u \geq \frac{1}{2}$ ou $\delta_v \geq \frac{1}{2}$.
■

Théorème 25 WEIGHTED VERTEX COVER admet un algorithme 2-approximant.

Démonstration. Soit C la couverture renvoyée par l'algorithme, et C^* une couverture minimale.

$$w(C) = \sum_{u \in C} w(u)$$

Pour tout $u \in C$, $\delta_u \leq \frac{1}{2}$ donc $1 \leq 2\delta_u$.
Ainsi,

$$\begin{aligned}
w(C) &\leq \sum_{u \in C} 2\delta_u w(u) \\
&\leq \sum_{u \in V} 2\delta_u w(u) && \text{car on ne fait que ajouter des termes positifs} \\
&\leq 2 \sum_{u \in V} \delta_u w(u) \\
&\leq 2 \sum_{u \in V} \delta_u w(u) \\
&\leq 2w(C^*) && \text{car la fonction caractéristique de } C^* \text{ est aussi solution du programme linéaire réelle}
\end{aligned}$$

■

4 Relaxation pour indépendent set

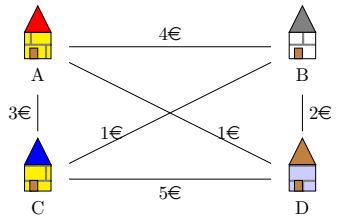
Ca marche pas bien. Section 3.4, p. 46 dans Matousek

5 Inapproximabilité du voyageur de commerce (TSP)

Définition 26 (problème d'optimisation) **TSP**

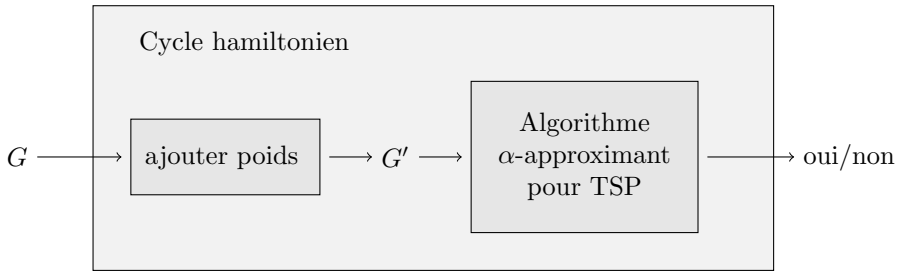
entrée : un graphe non orienté complet pondéré $G = (V, d)$ où $d : V \times V \to \mathbb{N}$ avec $d(i, j) = d(j, i)$
sortie : un tour dans G de poids minimal

Exemple 27



Théorème 28 Si P $\neq$ NP, alors TSP n'admet pas d'algo d'approximation en temps poly de ratio constant.

Démonstration. Par l'absurde, supposons que TSP admette un algo d'approximant en temps poly de ratio constant. Soit α le ratio. Sans perte de généralité, α est un entier. De là, construisons un algorithme polynomial décidant cycle hamiltonien. Comme cycle hamiltonien est NP-complet, on aurait donc P = NP et donc contradiction.



Ajout de poids. Le graphe G' est identique au graphe G mais on met des poids de 1 sur les arêtes de G et un poids artificiellement grand, $\alpha|V|+1$, entre les paires de sommets où il n'y a pas d'arêtes dans G. Formellement, soit $G=(V, E)$ un graphe non orienté. On crée le graphe complet pondéré $G' = (V, d)$ par :

— $d(u, v) = 1$ si $(u, v) \in E$;
— et $d(u, v) = \alpha|V| + 1$ si $(u, v) \notin E$.

```
fonction cycleHam(G)
    G' := ajouter des poids à G
    τ := algo α-approximant pour TSP sur G'
    si |τ| ≤ |V| alors
        renvoyer vrai
    sinon
        renvoyer faux
```

Fait 29 L'algo est en temps polynomial.

Démonstration. L'ajout de poids c'est en temps polynomial et l'algo d'approximant aussi par hypothèse. ■

Fait 30 cycleHam est correct.

Démonstration. Montrons que cycleHam(G) renvoie vrai ssi G admet un cycle hamiltonien.

⇒ Si cycleHam(G) renvoie vrai, alors il y a un tour de poids $|V|$, donc on prend que des arêtes de poids 1, i.e. des arêtes dans E. Donc c'est un cycle hamiltonien : G en admet bien un.

⇐ Supposons que G ait un cycle hamiltonien. C'est aussi un tour optimal de coût $|V|$. L'algorithme approximant renvoie un tour τ de coût $\leq \alpha|V|$. Le tour τ ne peut pas contenir une arête qui n'est pas dans E car sinon son coût est $> \alpha|V|$.

Le tour τ ne contient que des arêtes dans E. Donc $|\tau| = |V|$. Donc cycleHam(G) renvoie vrai.

■

■

6 Voyageur de commerce avec inégalité triangulaire

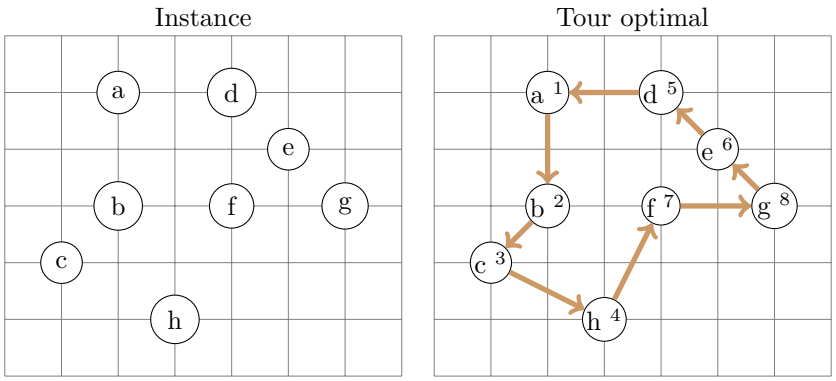
Applications : Transport en bus, faire des trous dans un morceau de bois.

Définition 31 Un graphe non orienté complet pondéré $G = (V, d)$ vérifie l'inégalité triangulaire si $d(u, w) \leq d(u, v) + d(v, w)$ pour tous les sommets $u, v, w \in V$.

Définition 32 **problème du voyageur de commerce avec inégalité triangulaire**

entrée : un graphe non orienté complet pondéré $G = (V, d)$ avec inégalité triangulaire ;
sortie : un tour de poids minimal.

Exemple 33



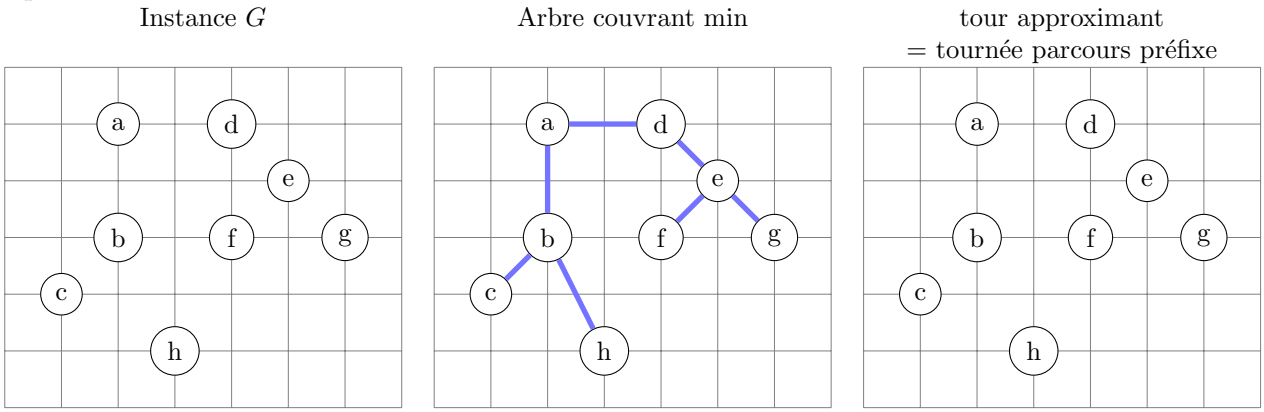
Exercice 2 Montrer que le problème du voyageur de commerce avec inégalité triangulaire est toujours NP-complet.

6.1 Algorithme 2-approximant (super simple à expliquer)

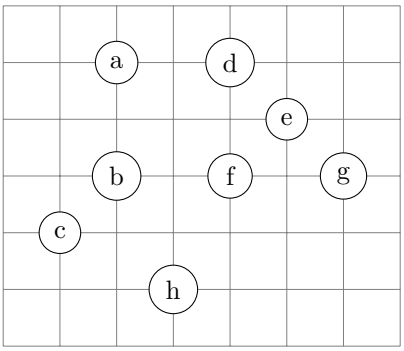
Théorème 34 METRIC TSP admet un algorithme 2-approximant.

fonction algoApprox(G)
  construire T un arbre couvrant minimum de G
  réaliser un parcours préfixe de T
  renvoyer la tournée correspondant au parcours préfixe

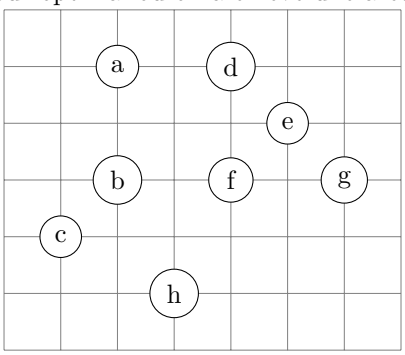
Exemple 35



Arbre couvrant min où on a doublé les arêtes

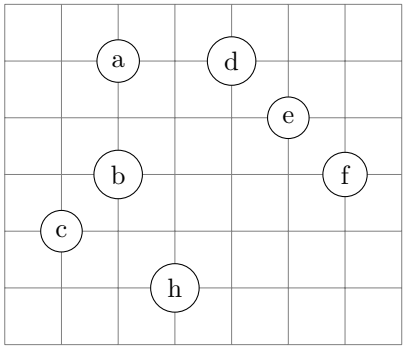


tour optimal où on a enlevé une arête

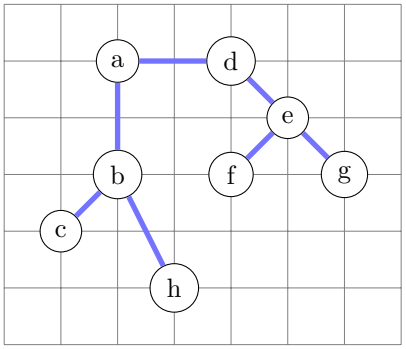


Exemple 2 (réponse)

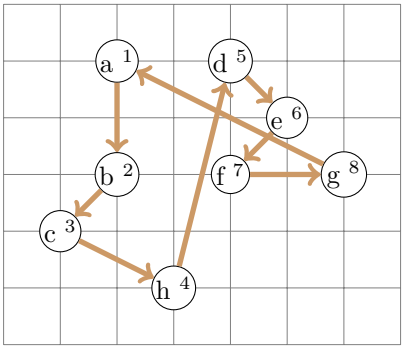
Instance G



Arbre couvrant min

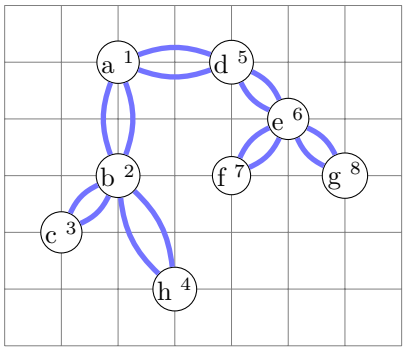


tour approximant
= tournée parcours préfixe



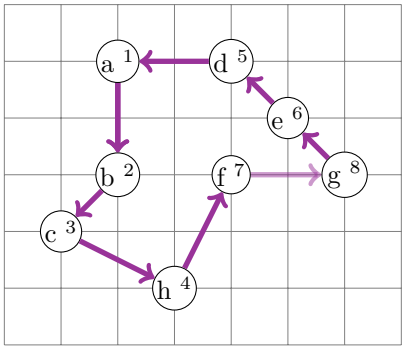
Proposition 36 Poids du tour approximant $\leq 2 \times$ poids d'un tour optimal.

Démonstration. Soit T l'arbre couvrant de poids min calculée pendant l'algo.



+ inégalité triangulaire

$w(\text{tour approx}) \leq 2w(T)$



Tour optimal où on a supprimé une arête
= arbre couvrant

$w(T) \leq w(\text{tour opt})$

$w(\text{tour approx}) \leq 2w(\text{tour opt})$

■

6.2 Algorithme de Christofides

On discute ici un algorithme un peu plus compliqué, mais pas pour rien : on obtient un meilleur facteur d'approximation.

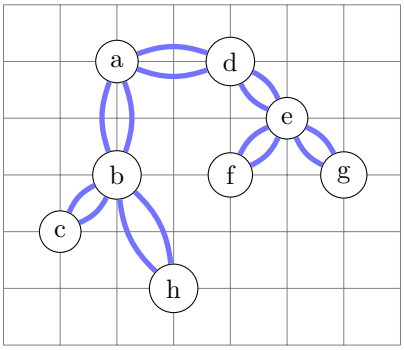
Théorème 37 (algorithme de Christofides) [Vaz04] METRIC TSP admet un algorithme $\frac{3}{2}$-approximant.

6.2.1 Première version de l'algorithme de Christofides

On revisite ici l'algorithme 2-approximant qu'on a vu, mais expliquée avec la notion de tour eulérien. Un tour eulérien est un tour qui passe une et seule fois par chaque arête. Eh oui, le parcours préfixe, c'est comme le tour eulérien où on ne visite les sommets G en ne gardant que les premières occurrences dans le tour.

fonction première version de l'algorithme de Christofides
    calculer un arbre couvrant minimum T
    $T' =$ le multi-graphe obtenu depuis T en doublant chaque arête
    $\epsilon :=$ tour eulérien dans T'
    renvoyer tour qui visite les sommets de G en ne gardant que les premières occurrences dans ϵ

Exemple 38 Voir l'arbre T où on a doublé chaque arête :



Proposition 39 Le ratio de l'algo est de 2.

DÉMONSTRATION. $cout(\text{tour approx}) \leq 2cout(T) \leq 2cout(\text{tour opt})$. ■

6.3 Algorithme de Christofides

Au lieu de doubler toutes les arêtes de l'arbre couvrant minimum T, nous considérons sa partie "problématique" : l'ensemble I des sommets de degrés impairs en T. En effet, c'est à cause d'eux qu'il n'y a pas de tour eulérien dans T.

La solution est donc de calculer un couplage parfait M à coût minimum sur I. Puis de calculer un tour eulérien dans $T \cup M$.

Définition 40 (couplage parfait) Un *couplage parfait* sur I est un couplage $M \subseteq I \times I$ tel que pour tout sommet $u \in I$, il y a exactement une arête $(a, b) \in M$ qui touche u.

Définition 41 (poids d'un couplage parfait) Le poids d'un couplage parfait est la somme des poids des arêtes du couplage.

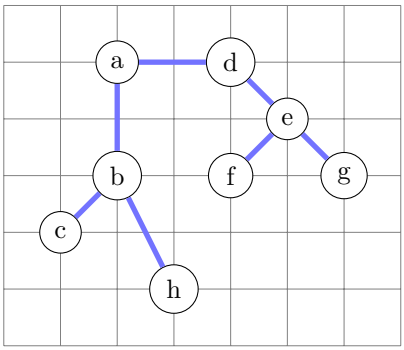
Définition 42 (couplage parfait minimum) Un couplage parfait minimum est un couplage parfait de poids minimum.

Projet 43 Algorithme d'Edmonds pour calculer un couplage parfait minimum. Ce problème est dans P (l'algorithme d'Edmonds est ardu à expliquer).

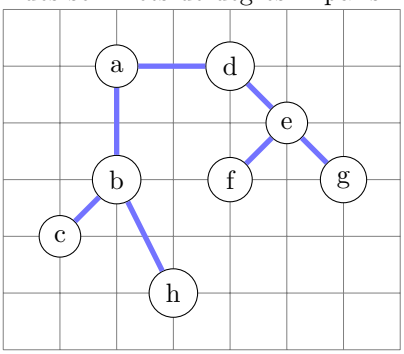
fonction algorithme de Christofides
    $T :=$ un arbre couvrant minimum
    $I :=$ sommets de degrés impairs dans T
    $M :=$ couplage parfait de coût min dans I
    $\epsilon :=$ tour eulérien dans $T \cup M$
    renvoyer $\tau =$ tour obtenu à partir de ϵ en ne gardant que les premières occurrences

Exemple 44

Ensemble des sommets de degrés impairs

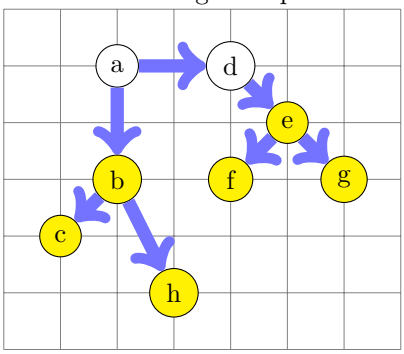


Couplage parfait de coût min des sommets de degrés impairs

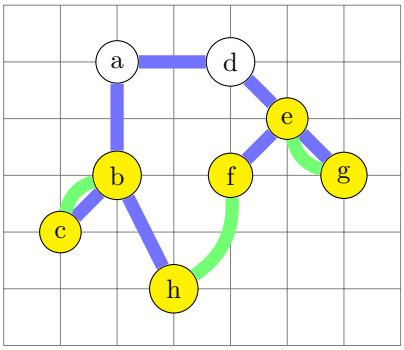


Exemple 45 (réponse)

En jaune, l'ensemble I des sommets de degrés impairs dans T



En vert, un couplage parfait de coût min dans I



Exemple 46 Voici un tour eulérien dans TM :

$$a, b, c, b, h, f, e, g, e, d, a$$

On ne garde que les premières occurrences et on obtient le tour retourné par l'algo :

$$a, b, c, \not{b}, h, f, e, g, \not{e}, d, a \qquad a, b, c, h, f, e, g, d, a.$$

Théorème 47 Soit τ le tour calculée. Soit τ^* un tour optimal. On a :

$$w(\tau) \leq \frac{3}{2} w(\tau^*)$$

.

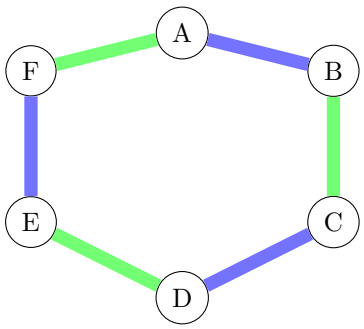
Démonstration. Dans l'algorithme 2-approximant, nous avons montré que $w(\text{tour approx}) \leq 2w(T)$. Là, de la même façon, on a

$$\begin{aligned} w(\tau) &\leq w(\epsilon) && \text{car on prend des raccourcis} \\ &\leq w(T) + w(M) && \text{car le tour est constitué exactement des arêtes de } T \text{ et } M \\ &\leq w(\tau^* \setminus \{\text{une arête}\}) + w(M) && \text{car } \tau^* \setminus \{\text{une arête}\} \text{ est un arbre couvrant} \\ &\leq w(\tau^*) + w(M) \end{aligned}$$

Lemme 48 $w(M) \leq \frac{1}{2} w(\tau^*)$.

Démonstration. Considérons un tour optimal τ^*. Soit $\tau^*_{|I}$ obtenu à partir de τ en ne gardant que les sommets dans I.

$$\dfrac{\dfrac{\text{La somme des degrés dans un graphe est paire.}}{\text{Le nombre de sommets dans } I \text{ est pair.}} \qquad \dfrac{\text{On a juste fait des raccourcis}}{\tau^*_{|I} \text{ est un cycle sur } I.}}{\tau^*_{|I} \text{ est l'union de deux couplages parfaits sur } I}$$



$$\dfrac{\dfrac{}{\text{L'un des deux couplages } C \text{ est de coût} \leq \frac{w(\tau^*_{|I})}{2}} \qquad \dfrac{\text{Inégalité triangulaire}}{w(\tau^*_{|I}) \leq w(\tau^*).}}{w(C) \leq \frac{w(\tau^*)}{2}}$$

$$\frac{\dfrac{M \text{ est un couplage parfait min sur } I}{w(M) \leq w(C)} \qquad w(C) \leq \frac{w(\tau^*)}{2}}{w(M) \leq \frac{w(\tau^*)}{2}}$$

■
■

Il existe des algos avec meilleurs ratio que $\frac{3}{2}$ [PV06].

7 Cycle eulérien (*)

On décrit ici la routine pour calculer un cycle eulérien, utilisé dans l'algorithme de Christofidès.

```
entrée : un graphe G non orienté, chaque sommet est de degré pair, un sommet v
sortie : un cycle eulérien de la composante connexe contenant v
effet : les arêtes utilisées sont supprimées de G
fonction eul(G, v)
    si v est isolé dans G alors
        renvoyer [v]
    sinon
        τ := baladeEffaçante(G, v)
        renvoyer τ.map(u ↦ eul(G, u)).join
```

```
entrée : un graphe G non orienté, chaque sommet est de degré pair, un sommet v
sortie : un cycle contenant v (non trivial)
effet : les arêtes utilisées sont supprimées de G
fonction baladeEffaçante(G, v)
    v := u
    τ := [u]
    tant que v a des voisins
        choisir un voisin w de v
        supprimer l'arête {v, w} de G
        τ := w :: τ
        v := w
```

Proposition 49 *eul*(G, v) s'exécute en temps $O(|E_v| + 1)$.

Démonstration. Par induction sur le nombre d'arêtes dans la composante de v. Si v est isolé (pas d'arêtes), complexité en $O(1)$. Soit k le nombre de sommets dans la balade effaçante. La balade effaçante est en $O(k)$.

Soit $E_1, \ldots, E_k$ les ensemble d'arêtes restantes pour les appels récursifs à *eul* (dans le *map*). La complexité est en

$$k + \sum_{i=1}^{k} |E_i| + 1 = O(|E_v|).$$

On modélise chaque cycle avec une liste doublement chaînée de sorte que

$$\tau.map(u \mapsto eul(G, u)).join$$

est en temps $O(k)$. ■

Proposition 50 *eul*(G, v) est correcte.

Démonstration. Par récurrence sur le nombre d'arêtes dans la composante connexe contenant v.

Cas de base. C'est ok, v est isolé, on renvoie $[v]$.

Cas récursif. Que l'on obtient un cycle eulérien : c'est clair. Ce qui n'est pas évident : c'est que l'on attrape *toute* la composante connexe contenant v. Soit τ le cycle contenant v obtenu via la balade effaçante. Ce cycle est non-trivial : il contient au moins une (deux en fait !) arête. Du coup, le graphe G a perdu des arêtes et on peut appliquer l'hypothèse de récurrence. Ainsi, les appels $eu(G, u)$ à la ligne

$$\tau.map(u \mapsto eul(G, u)).join$$

par hypothèse de récurrence renvoie à chaque fois un cycle eulérien de la composante connexe contenant u dans le graphe G.

Supposons que le cycle eulérien final ne passe pas par un certain sommet u' dans la composante connexe de v dans le graphe G initial. Considérons la balade effaçante dans l'ordre et prenons le premier sommet u sur cette balade d'où u' est accessible. Au moment où $eul(G,u)$, le chemin de u à u' existe toujours dans G. Donc $eul(G,u)$ renvoie un chemin eulérien où u' apparaît. Contradiction. ■

8 Voyage de commerce euclidien

Définition 51 **problème du voyageur de commerce euclidien**

entrée : un ensemble P de n points de $\mathbb{R}^d$ où le poids entre points est donné par la distance euclidienne ;
sortie : un tour sur P de poids minimal.

Théorème 52 (admis) EUCLIDEAN TSP admet un PTAS [Aro98].

9 Set cover : approximation avec un ratio $O(\log n)$

Application 53 On a un ensemble de n villes et on veut placer un nombre minimum d'écoles avec deux contraintes : chaque école est dans une ville et il doit y avoir une école à moins de 10km de chaque village.

Définition 54 **SET COVER**

entrée : U un ensemble fini de n éléments, et des sous-ensembles $S_1, \cdots, S_k$ inclus dans U ;
sortie : Un sous-ensemble $J \subseteq \{1,\dots,k\}$ tel que $\bigcup_{j\in J} S_j = U$ et $|J|$ est minimal (s'il en existe).

Exemple 3 U est un ensemble des n villes et chaque S_i est l'ensemble des villes à moins de 10km de la ville numéro i.

Si on utilise une stratégie gloutonne, on a un algorithme approximatif mais efficace. À chaque fois, on choisit un ensemble S_j avec le maximum d'éléments non couverts.

```
fonction approxSetCover(U, S_1, ··· , S_k)
    J := ensemble vide
    C := ∅
    tant que C ≠ U
        choisir j ∈ {1,...,n} tel que |S_j ∩ U\C| soit maximal.
        si S_j ⊆ C alors
            renvoyer impossible
        C := C ∪ S_j
        ajouter j à J
    renvoyer J
```

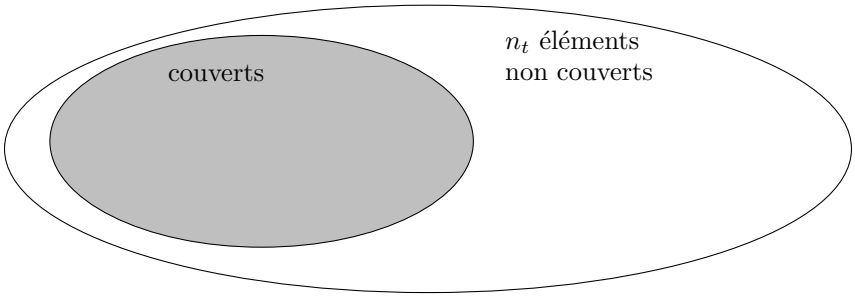
```
fonction approxSetCover(U, S_1, ··· , S_k)
    J := ensemble vide
    tant que ⋃_{j∈J} S_j ≠ U
        choisir j ∈ {1,...,n} tel que |S_j ∩ U\⋃_{j∈J} S_j| soit maximal.
        si S_j ⊆ ⋃_{j∈J} S_j alors
            renvoyer impossible
        ajouter j à J
    renvoyer J
```

Théorème 55 L'algorithme glouton choisit au plus $k\lceil \ln(n)\rceil$ sous-ensembles
où $n = |U|$ et $k =$ nb de sous-ensembles dans une couverture minimale.

Démonstration. Supposons qu'il y a une couverture. Considérons une couverture minimale $\bigcup_{j\in J^*} S_j$ avec $|J^*| = k$. On note n_t le nombre d'éléments non couverts au t-ème tour de boucle, autrement dit il s'agit de $|U\setminus\bigcup_{j\in J} S_j|$ pour le J courant. Au début, pour $t=0$, on a $n_0 = n$.

Lemme 56 Pour tout $t \geq 1$, $n_t \leqslant n\left(1-\frac{1}{k}\right)^t$.

Démonstration. Plaçons nous au t-ème tour de boucle.



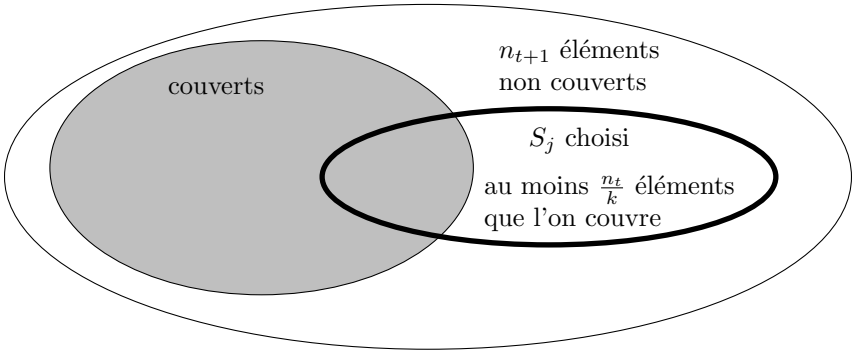
$\bigcup_{j\in J^*} S_j$ couvre tous les éléments

$\bigcup_{j\in J^*} S_j$ couvre les n_t éléments non couverts

il y a un ensemble S_j avec $j \in J^*$ contenant au moins $\frac{n_t}{k}$ éléments non couverts.

À l'étape t l'algorithme va choisir un ensemble S_j et couvrir au moins $\frac{n_t}{k}$ éléments supplémentaires.

$$n_{t+1} \leqslant n_t - \tfrac{n_t}{k} = n_t\left(1 - \tfrac{1}{k}\right).$$



Par récurrence sur $t \in \mathbb{N}$, on montre que $n_t \leqslant n\left(1 - \frac{1}{k}\right)^t$.

■

inégalité de convexité $1 - x \leq e^{-x}$ pour tout x réel, avec égalité uniquement si $x = 0$

$$n_t < ne^{-\frac{t}{k}}$$

Maintenant, à partir de quelle étape t, tous les éléments sont couverts à coup sûr ? Pour $t \geqslant k\lceil\ln(n)\rceil$, on a $ne^{-\frac{t}{k}} \leq ne^{-\lceil\ln(n)\rceil} < 1$; et donc tous les éléments sont couverts avec $k\lceil\ln(n)\rceil$ ensembles. ■

Remarque 57 Si $P \neq NP$, Raz et Safra ont montré qu'on ne faire mieux qu'un ratio de $c \log n$ où $c > 0$ [RS97].

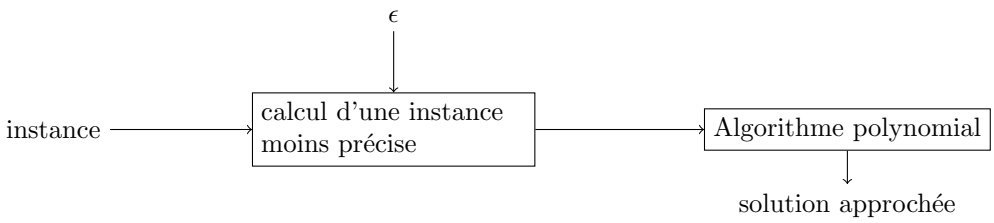
10 Sac à dos

Ref : [TK06][11.8, p. 644], [DPV06][p. 283], [Vaz04][p. 68]

SAC A DOS

entrée : n objets $((w_i, v_i))_{i=1..n}$, un poids W
sortie : un sous-ensemble $S \subseteq \{1, \ldots, n\}$ tel que $\sum_{i\in S} w_i \leq W$ et $\sum_{i\in S} v_i$ maximal.

Théorème 58 SAC A DOS admet un FPTAS.



10.1 Sous-routine basée sur la programmation dynamique

En ALGO1, nous avons donné un algorithme de programmation dynamique en temps $O(nW)$ où W est le poids total. Ici, nous donnons un algorithme en temps $O(n\sum_{i=1}^{n} v_i)$ où $\sum_{i=1}^{n} v_i$ est la somme des valeurs.

On parle d'*algorithme pseudo-polynomial*.

Définition 59 (sous-problèmes) Pour tout $k \in \{1,\dots,n\}$, tout $V \in \{1,\dots,\sum_{i=1}^{n} v_i\}$,

$$\begin{array}{rcl} opt(k,V) & = & \text{le poids minimal d'un sac contenant des objets de } \{1,\dots,k\} \text{ avec une valeur totale} \geq V \\ & = & \min_{S\subseteq\{1,\dots,k\} | \sum_{i\in S} v_i \geq V} \sum_{i\in S} w_i. \\ & & \text{ou } +\infty \text{ quand il n'y a pas de } S \subseteq \{1,\dots,k\} \mid \sum_{i\in S} v_i \geq V. \end{array}$$

Proposition 60 La solution cherchée est la valeur V maximale telle que $opt(n,V) \leq W$.

Exercice 61 Concevoir un algorithme qui calcule une solution à sac à dos en temps $O(n\sum_{i=1}^{n} v_i)$.

Solution :

Cas de base :

$$opt(0,V) = \begin{cases} 0 \text{ pour } V = 0 \\ +\infty \text{ pour tout } V \geq 1 \end{cases}$$

Cas récursif : si $k \geq 1$, on a :

$$opt(k,V) = \begin{cases} min(opt(k-1,V), w_k + opt(k-1, V-v_k)) & \text{si } v_k \leq V \\ min(opt(k-1,V), w_k) & \text{sinon} \end{cases}$$

10.2 Approximation

On remarque que l'algorithme pseudo-polynomial est très efficace si les valeurs des objets sont petites. L'idée est donc de faire que les valeurs des objets soient des entiers dans $[0, \frac{n}{\epsilon}]$. On pose $v_{max} = \max_{i=1..n} v_i$.

fonction *sacadosApprox*$((w_i,v_i)_{i=1..n}, W, \epsilon)$
  $v_{max} = \max(v_1,\dots,v_n)$
  pour $i = 1..n$ **faire**
    $\widetilde{v}_i := \lfloor \frac{n}{\epsilon} \frac{v_i}{v_{max}} \rfloor$
  renvoyer *sacados*$((w_i,\widetilde{v}_i)_i, W)$

Proposition 62 *sacadosApprox* est en temps $O(\frac{n^3}{\epsilon})$.

Démonstration.
Les nouvelles valeurs $\widetilde{v}_k$ sont dans $\{0,\dots,\lfloor \frac{n}{\epsilon} \rfloor\}$. Ainsi $\sum_{i=1}^{n} \widetilde{v}_i = \Theta(\frac{n^2}{\epsilon})$, et la complexité est en $O(n \times \frac{n^2}{\epsilon})$. ■

Théorème 63 Si S^{algo} est la solution trouvé par l'algorithme d'approximation, alors pour toute solution S^* avec $\sum_{i\in S^*} w_i \leq W$ on a

$$\sum_{i\in S^{algo}} v_i \geq \sum_{i\in S^*} v_i(1-\epsilon).$$

Démonstration.
Sans perte de généralité, on suppose que $v_{max} \leq \left(\sum_{i\in S^*} v_i\right)$. Pourquoi ? Parce que l'on peut supprimer les objets i avec $w_i > W$. Ainsi, v_{max} correspond à la valeur de la solution où on prend uniquement l'objet de valeur maximal, et cette valeur est plus petite que la valeur de la solution optimale $\left(\sum_{i\in S^*} v_i\right)$.

$$\begin{array}{lll} \sum_{i\in S^{algo}} v_i & \geq \sum_{i\in S^{algo}} \frac{\epsilon}{n} v_{max} \widetilde{v}_i & \text{par définition de } \widetilde{v}_i, \text{ on a } \widetilde{v}_i \leq \frac{n}{\epsilon}\frac{v_i}{v_{max}} \\ & \geq \sum_{i\in S^*} \widetilde{v}_i \frac{\epsilon}{n} v_{max} & \text{car l'algo est correct par rapport aux valeurs } \widetilde{v}_i \text{ et donc avec } S^* \text{ on a pire} \\ & \geq \sum_{i\in S^*} (\frac{n}{\epsilon}\frac{v_i}{v_{max}} - 1)\frac{\epsilon}{n} v_{max} & \text{par définition de } \widetilde{v}_i, \text{ on a } \frac{n v_i}{\epsilon v_{max}} \leq \widetilde{v}_i + 1 \\ & \geq \sum_{i\in S^*} (v_i - \frac{\epsilon}{n} v_{max}) & \\ & \geq \left(\sum_{i\in S^*} v_i\right) - \epsilon \times v_{max} & \\ & \geq \left(\sum_{i\in S^*} v_i\right)(1-\epsilon) & \text{car } v_{max} \leq \left(\sum_{i\in S^*} v_i\right) \end{array}$$

■

Références

[Aro98] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5) :753–782, 1998.

[DPV06] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. Algorithms. 2006.

[DS05] I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, pages 439–485, 2005.

[Kar09] G. Karakostas. A better approximation ratio for the vertex cover problem. *ACM Transactions on Algorithms (TALG)*, 5(4) :41, 2009.

[PV06] C.H. Papadimitriou and S. Vempala. On the approximability of the traveling salesman problem. *Combinatorica*, 26(1) :101–120, 2006.

[RS97] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability pcp characterization of np. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 475–484. ACM, 1997.

[TK06] É. Tardos and J. Kleinberg. Algorithm design, 2006.

[Vaz04] V.V. Vazirani. *Approximation algorithms*. springer, 2004.

Algorithme de Karger

François Schwarzentruber

19 mars 2024

Réf : [MR95][p. 7, 24, p. 289]

☺ Algorithme de Karger simple à comprendre, alors qu'il faut se lever de bonne heure pour comprendre :

flots, algorithme de Ford-Fulkerson, graphe résiduel, chemin améliorant, théorème de dualité coupe min/flot max, boucle pour toute destination, etc.

☺ Optimisations qui donnent l'algorithme le plus efficace connu à ce jour
☺ Algorithme qui, en le répétant, donne des solutions de plus en plus bonnes
☹ Pas de garantie d'avoir la meilleure solution

1 Définitions

Définition 1 (coupe) Soit $G = (V, E)$ un graphe, connexe, non orienté, avec arêtes multiples mais sans boucles. On appelle coupe de G toute partition $\{X, Y\}$ de G avec $X \neq \emptyset$ et $Y \neq \emptyset$.

Définition 2 (arêtes d'une coupe) L'ensemble des arêtes d'une coupe $\{X, Y\}$ est

$$C_{\{X,Y\}} = \{(x, y) \in E \mid x \in X \text{ et } y \in Y\}.$$

Définition 3 (taille d'un coupe) La taille d'une coupe $\{X, Y\}$ est le cardinal de $C_{\{X,Y\}}$.

Définition 4 (coupe minimale) Une coupe $\{X, Y\}$ est minimale si $C_{\{X,Y\}}$ est de cardinal minimal.

Définition 5 **Problème du calcul de la coupe minimale**

entrée : Un graphe G connexe, non orienté, avec arêtes multiples, sans boucles
sortie : une coupe minimale de G

Remarque 6 Le problème de la coupe maximale est lui NP-complet. De la même façon, si on considère la coupe minimale avec poids, et que les poids peuvent être négatifs, alors on peut y réduire le problème de la coupe maximale et on montre que le problème de la coupe minimale avec poids est NP-complet.

Proposition 7 Le problème du calcul de la coupe minimale est dans P.

Dans la suite, on écrit simplement C au lieu de $C_{\{X,Y\}}$ pour désigner une coupe et on considère que c'est l'ensemble d'arêtes qui vont de X à Y.

2 Applications

On trouve quelques applications dans [KS96] que je résume ici.

Application 8 (réseaux de communication) Nombre minimal d'arêtes à couper pour que deux ordinateurs ne puissent plus communiquer.

Application 9 (documents avec liens hypertextes) La coupe minimale sépare les documents en deux catégories qui sont peu reliées, et donc qui, a priori sont à propos de sujets différents.

Application 10 (génie logiciel) Une coupe minimale dans le graphe de dépendances entre classes permet de découper le logiciel en deux packages.

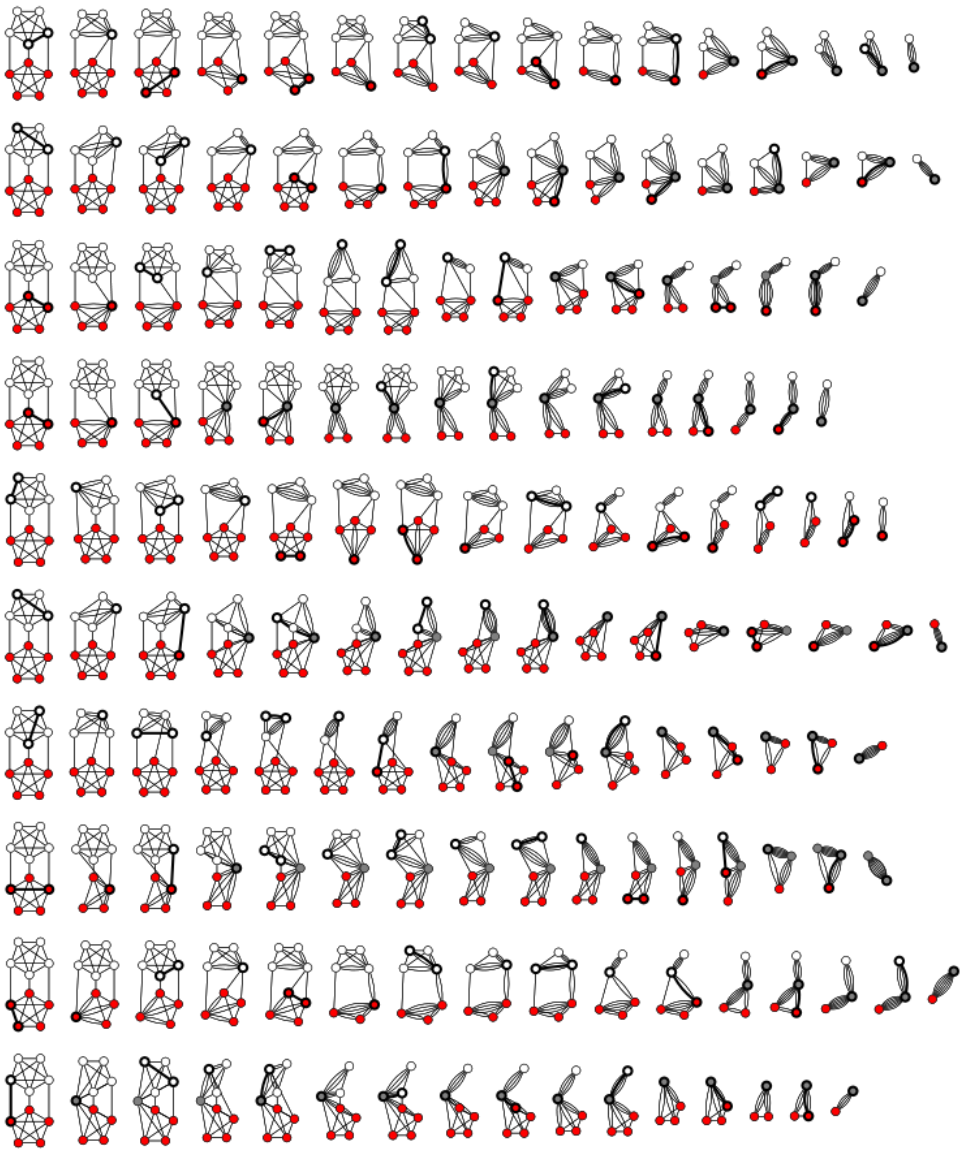
Application 11 (compilation de langages parallèles) Considérons le graphe du programme où les nœuds sont les actions du programme et les arcs sont les communications (échange de message) entre points du programme. Il s'agit de minimiser le nombre d'échanges de message.

3 Première version de l'algorithme de Karger

Animation : `https://playwithalgos.github.io/karger/`

entrée : un multi-graphe G connexe non orienté
sortie : les arêtes d'une coupe
fonction algoKarger(G)
  tant que G a strictement plus de 2 sommets
    sélectionner uniformément au hasard une arête e de G
    contracter l'arête e dans G
  renvoyer l'ensemble C des arêtes de G

Durant l'algorithme, un sommet représente un sous-ensemble de sommets du graphe initial.



Source : wikipedia

3.1 Correction

Proposition 12 Soit G un graphe à n sommets. Soit C^* une coupe minimale. Alors :

$$\mathbb{P}(\text{algoKarger renvoie } C^*) \geq \frac{2}{n(n-1)}.$$

Démonstration. Soit C^* une coupe minimale de G et soit $|C^*|$ sa taille. Soit $e_1, \dots, e_{n-2}$ les arêtes contractés pendant l'algorithme. Ce sont des variables aléatoires (même si je ne les écris pas en majuscule).

$$\begin{aligned}\mathbb{P}(\text{algoKarger renvoie } C^*) &= \mathbb{P}(\bigwedge_{i=1}^{n-2}(e_i \notin C^*)) \\ &= \Pi_{i=1}^{n-2}\mathbb{P}(e_i \notin C^* \mid e_1, e_2, \dots, e_{i-1} \notin C^*)\end{aligned}$$

Lemme 13 $\mathbb{P}(e_i \notin C^* \mid e_1, e_2, \dots, e_{i-1} \notin C^*) \geq 1 - \frac{2}{n-i+1}$.

Démonstration. Démontrons d'abord le cas simple pour $i = 1$.
On a :

$$\dfrac{\mathbb{P}(e_1 \in C^*) = \frac{|C^*|}{|E|} \qquad \dfrac{\dfrac{\sum_{w\in V} deg(w) = 2|E| \qquad deg(w) \geq |C^*|}{2|E| \geq n|C^*|}}{|C^*| \leq \frac{2|E|}{n}}}{\mathbb{P}(e_1 \in C^*) \leq \frac{2}{n}}$$

De manière générale, si $e_1, \dots, e_{i-1} \notin C^*$ alors le graphe contracté possède $n - i + 1$ sommets et possède toujours C^* comme coupe minimale. Ainsi, on a :

$$\mathbb{P}(e_i \in C^* \mid e_1, e_2, \dots, e_{i-1} \notin C^*) \leq \frac{2}{n-i+1}$$

■
Finalement :

$$\begin{aligned}\mathbb{P}(\text{algoKarger renvoie } C^*) &\geq \Pi_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) && \text{par le lemme} \\ &= \Pi_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) \\ &= \frac{(n-2)\times\cdots\times 2\times 1}{n(n-1)\times\cdots\times 3} \\ &= \frac{2}{n(n-1)}\end{aligned}$$

■

Corollaire 14

$$\mathbb{P}(\text{algoKarger renvoie une coupe minimale}) \geq \frac{2}{n(n-1)}$$

Démonstration. Car $\mathbb{P}(\text{algoKarger renvoie une coupe minimale}) \geq \mathbb{P}(\text{algoKarger renvoie } C^*)$. ■

3.2 Implémentation

On considère ici qu'au lieu de choisir les arêtes à contracter à la volée, on choisit une permutation de E dès le début, et on contracte dans l'ordre correspondant.

Du coup, l'algorithme de Karger ressemble beaucoup à l'algorithme de Kruskal avec les changements suivants :
— les supersommets sont les arbres/composantes de la forêt en construction
— on s'arrête quand il y a deux forêts dans l'arbre

Quand on examine une arête, soit elle fait une contraction ; soit c'est une arête au sens d'une composante (on ne fait rien). On rappelle que Kruskal est implémentable en $O(E \log V)$ avec une structure union-find et en $O(E \log^* V)$ en utilisant compression de chemins. Bref, on a la proposition suivante.

Proposition 15 On peut implémenter l'algorithme de Karger *algoKarger* en temps $O(E \log^* V)$.

En fait, la complexité temporelle peut descendre à $O(|E|)$ avec cette astuce. Dans l'algo suivant, on note C_x la composante de x. On note C_e l'arête entre les deux composantes C_x et C_y avec $e = (x, y)$.

entrée : un multi-graphe G connexe non orienté, une permutation $e_1, \dots, e_{|E|}$ des arêtes
sortie : les arêtes d'une coupe
fonction algoKarger($G, e_1, \dots, e_{|E|}$)
    calculer l'ensemble $\mathcal{C}$ des composantes connexes de $(V, \{e_1, \dots, e_{|E|/2}\})$ avec un parcours en profondeur
    si il n'y a qu'une composante dans $\mathcal{C}$ **alors**
        // oups, on a contracté trop d'arêtes !
        renvoyer algoKarger($G, e_1, \dots, e_{|E|/2}$)
    sinon
        construire le graphe $H = (\mathcal{C}, \bigcup_{i=|E|/2+1}^{|E|} \{C_{e_i}\}$
        si il y a deux composantes dans $\mathcal{C}$ **alors**
            renvoyer H
        sinon
            renvoyer algoKarger($H, C_{e_{|E|/2+1}}, \dots, C_{e_{|E|}}$)

En notant m le nombre d'arêtes, la complexité temporelle $T(m) = O(m) + T(m/2)$. Le $O(m)$ vient du calcul des composantes et du calcul du graphe H. Ainsi, $T(m) = O(m)$. Bref, on a la proposition suivante :

Proposition 16 On peut implémenter l'algorithme de Karger *algoKarger* en temps $O(|E|)$, i.e. en $O(n^2)$.

3.3 Amplification

Heureusement, en itérant on peut avoir avec une forte probabilité une coupe minimale. Plus précisément :

Proposition 17 Pour tout $\epsilon \in]0, 1[$,

$$\mathbb{P}(\lceil \frac{n^2}{2} \ln \frac{1}{\epsilon} \rceil \text{ répétitions indépendantes de l'algo donne une coupe min}) \geq 1 - \epsilon.$$

Démonstration.

$$\mathbb{P}(\text{une exécution donne une coupe min}) \geq \frac{2}{n^2}.$$

$$\mathbb{P}(\text{une exécution ne donne pas une coupe min}) \leq 1 - \frac{2}{n^2}.$$

Donc

$$\mathbb{P}(\text{aucune des } k \text{ exécutions ne donne une coupe min}) \leq \left(1 - \frac{2}{n^2}\right)^k.$$

Or pour tout $x > 0$,

$$\left(1 - \frac{1}{x}\right)^x < \frac{1}{e}.$$

Ainsi avec $x = \frac{n^2}{2}$.

$$\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{e}$$

Donc en mettant à la puissance $\ln \frac{1}{\epsilon}$,

$$\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2} \ln\left(\frac{1}{\epsilon}\right)} < e^{-\ln\left(\frac{1}{\epsilon}\right)} = \epsilon.$$

$$\mathbb{P}(\text{aucune des } \lceil \frac{n^2}{2} \ln \frac{1}{\epsilon} \rceil \text{ exécutions ne donne une coupe min}) \leq \left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2} \ln\left(\frac{1}{\epsilon}\right)} < \epsilon.$$

$$\mathbb{P}(\lceil \frac{n^2}{2} \ln \frac{1}{\epsilon} \rceil \text{ répétitions indépendantes de l'algo donne une coupe min}) \geq 1 - \epsilon.$$

■

3.4 Nombre de coupes minimales

Exercice 18 Montrer qu'il y a au plus $\frac{n(n-1)}{2}$ coupes minimales.

Démonstration. Soit $C_1, \dots, C_r$ les différentes coupes minimales. $\mathbb{P}$(l'algorithme retourne C_i) $\geq \frac{2}{n(n-1)}$. Mais

$$1 \geq \mathbb{P}(\text{algo retourne une coupe minimale}) = \sum_i \mathbb{P}(\text{algo retourne } C_i) \geq r\frac{2}{n(n-1)}.$$

D'où $r \leq \frac{n(n-1)}{2}$. ■

Là, pour avoir un algo avec une proba de réussite $\geq \frac{1}{2}$, il faut répéter $O(n^2)$ un algorithme en temps $O(n^2)$. Du coup, en tout, on a une solution en $O(n^4)$. Ce n'est pas raisonnable.

4 Amélioration

Voici les arguments qui permettent de concevoir un meilleur algorithme :

1. il y a peu de chances que l'algorithme contracte une arête de C^* dans les premières itérations ; car il y a beaucoup d'arêtes.
2. il y a malheureusement de plus en plus de chances de contracter une arête de C^* dans les dernières itérations.
3. Heureusement, l'algorithme est plus rapide sur de petits graphes.

Proposition 19 (probabilité de conserver C^* après $n-t$ itérations)

$$\mathbb{P}(\bigwedge_{i=1}^{n-t}(e_i \notin C^*) = \frac{t(t-1)}{n(n-1)}$$

Démonstration.

$$\begin{aligned}
\mathbb{P}(\bigwedge_{i=1}^{n-t}(e_i \notin C^*) &\geq \Pi_{i=1}^{n-t}\left(1 - \frac{2}{n-i+1}\right) \\
&= \Pi_{i=1}^{n-t}\left(\frac{n-i-1}{n-i+1}\right) \\
&= \frac{(n-2)\times\cdots\times(n-t-1)}{n(n-1)\times\cdots\times(n-t+1)} \\
&= \frac{t(t-1)}{n(n-1)}
\end{aligned}$$

■

Corollaire 20 Pour $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$, on a :

$$\mathbb{P}(\bigwedge_{i=1}^{n-t}(e_i \notin C^*) \geq \frac{1}{2}.$$

Démonstration. On veut $\frac{t(t-1)}{n(n-1)} \geq \frac{1}{2}$. Autrement dit, $t(t-1) \geq \frac{1}{2}n(n-1) = \frac{n}{\sqrt{2}}\frac{n-1}{\sqrt{2}}$. C'est le cas quand $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$. ■

```
fonction fastCut(G)
    si G est petit alors
        renvoyer une coupe minimale de G par force brute
    sinon
        t := ⌈1 + n/√2⌉
        exécuter les itérations de contractions pour obtenir un graphe H1 avec t sommets
        exécuter les itérations de contractions pour obtenir un graphe H2 avec t sommets
        C1 := fastcut(H1)
        C2 := fastcut(H2)
        renvoyer la coupe la plus petite entre C1 et C2
```

Proposition 21 L'algorithme est en $O(n^2 \log n)$.

Démonstration. On applique master theorem avec

$$T(n) = 2T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil) + O(n^2).$$

■

Proposition 22

$$\mathbb{P}(\text{fastcut renvoie une coupe minimale}) = \Omega(\frac{1}{\log n})$$

Démonstration.
On pose les événements :

$$\begin{array}{lcl} G \xrightarrow{fc} \checkmark & : & fastcut(G) \text{ renvoie une coupe min de } G \\ G \xrightarrow{\checkmark} H_i & : & H_i \text{ possède une min cut de } G \\ G \xrightarrow{\checkmark} H_i \xrightarrow{fc} \checkmark & : & fastcut(G) \text{ pourrait [1]renvoyer une coupe min de } G \text{ via } H_i \\ & & = (G \xrightarrow{\checkmark} H_i) \text{ et } (H_i \xrightarrow{fc} \checkmark) \end{array}$$

$$G \xrightarrow{fc} \checkmark = (G \xrightarrow{\checkmark} H_1 \xrightarrow{fc} \checkmark) \text{ ou } (G \xrightarrow{\checkmark} H_2 \xrightarrow{fc} \checkmark)$$

$$\begin{aligned} \mathbb{P}(G \xrightarrow{fc} \checkmark) &= 1 - \mathbb{P}(\overline{G \xrightarrow{fc} \checkmark}) \\ &= 1 - \mathbb{P}(\overline{(G \xrightarrow{\checkmark} H_1 \xrightarrow{fc} \checkmark) \text{ ou } (G \xrightarrow{\checkmark} H_2 \xrightarrow{fc} \checkmark)}) \\ &= 1 - \mathbb{P}(\overline{(G \xrightarrow{\checkmark} H_1 \xrightarrow{fc} \checkmark)} \text{ et } \overline{(G \xrightarrow{\checkmark} H_2 \xrightarrow{fc} \checkmark)}) \\ &= 1 - \mathbb{P}(\overline{(G \xrightarrow{\checkmark} H_1 \xrightarrow{fc} \checkmark)}) \times \mathbb{P}(\overline{(G \xrightarrow{\checkmark} H_2 \xrightarrow{fc} \checkmark)}) \\ &= 1 - (1 - \mathbb{P}((G \xrightarrow{\checkmark} H_1 \xrightarrow{fc} \checkmark))) \times (1 - \mathbb{P}((G \xrightarrow{\checkmark} H_2 \xrightarrow{fc} \checkmark))) \end{aligned}$$

Lemme 23 $\mathbb{P}(G \xrightarrow{\checkmark} H_i \xrightarrow{fc} \checkmark) \geq \frac{1}{2}\mathbb{P}(H_i \xrightarrow{fc} \checkmark)$.

Démonstration.

$$\begin{aligned} \mathbb{P}(G \xrightarrow{\checkmark} H_i \xrightarrow{fc} \checkmark) &= \mathbb{P}((G \xrightarrow{\checkmark} H_i) \text{ et } (H_i \xrightarrow{fc} \checkmark)) \\ &= \mathbb{P}((G \xrightarrow{\checkmark} H_i))\mathbb{P}((H_i \xrightarrow{fc} \checkmark)) \\ &\geq \frac{1}{2}\mathbb{P}(H_i \xrightarrow{fc} \checkmark) && \text{par le corollaire 20} \end{aligned}$$

■

Ainsi, on a :

$$\mathbb{P}(G \xrightarrow{fc} \checkmark) \geq 1 - (1 - \frac{1}{2}\mathbb{P}(H_1 \xrightarrow{fc} \checkmark)) \times (1 - \frac{1}{2}\mathbb{P}(H_2 \xrightarrow{fc} \checkmark))$$

Soit $P(k)$ la propriété qui dit : si le graphe G est au niveau de récursion k, alors $\mathbb{P}(G \xrightarrow{fc} \checkmark) \geq \frac{1}{k+1}$. On la démontre par récurrence sur k via l'équation ci-dessus.

Comme le graphe G initial se trouve à un niveau de récursion $O(\log n)$, on a le résultat. ■

Ainsi, on peut amplifier et répéter l'algorithme $\log n$ fois pour avoir une borne inférieure constante de la probabilité de succès.

Remerciements

Merci à Arnaud Jobin pour les discussions sur la démonstration actuelle du temps d'exécution du tri rapide.

1. pourrait car ne le fait pas peut-être car ça marche avec H_1 **et** H_2

Références

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3) :251–280, 1990.

[KS96] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4) :601–640, 1996.

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

Méthode probabiliste et dérandomisation

François Schwarzentruber

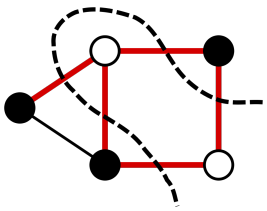
— La *méthode probabiliste* est l'art de démontrer l'existence d'objets par des arguments probabilistes.
— La *dérandomisation* consiste à construire un algorithme déterministe à partir d'un algorithme probabiliste, en gardant les même garanties.

1 MAXCUT

Définition 1 (taille d'une coupe) Soit $G = (V, E)$ un graphe non orienté. Soit $C = \{V_0, V_1\}$ avec $V_0 \sqcup V_1 = V$, une coupe de G. La taille de C, notée $\#C$, est le nombre d'arêtes allant d'un sommet de V_0 à un sommet de V_1.

Définition 2 (coupe maximale) Une coupe maximale est une coupe de taille maximale.

Exemple 3 (coupe maximale de taille 5)



Définition 4 **MAXCUT**
entrée : un graphe G non orienté
sortie : une coupe maximale

1.1 Algorithme naïf

L'idée est de placer chaque sommet de manière aléatoire uniforme dans V_0 ou dans V_1.

```
fonction maxcut(G)
    V_0 := ∅
    V_1 := ∅
    pour u ∈ V faire
        choisir i ∈ {0,1} de manière uniforme
        ajouter u à V_i
    renvoyer (V_0, V_1)
```

1.2 Existence d'une solution

Théorème 5 Soit $G = (V, E)$ une graphe non orienté. Il existe une coupe de G de taille $\geq \frac{|E|}{2}$.

Démonstration. Soit C la coupe calculée par l'algorithme naïf. On note, pour tout sommet u :

$$X_u = \begin{cases} 0 & \text{si } u \in V_0 \\ 1 & \text{si } u \in V_1. \end{cases}$$

🎓

$$\frac{\dfrac{\#C = \sum_{(u,v)\in E} 1_{X_u \neq X_v}}{\mathbb{E}(\#C) = \sum_{(u,v)\in E} \mathbb{E}(1_{X_u \neq X_v})} \qquad \mathbb{E}(1_{X_u \neq X_v}) = \mathbb{P}(X_u \neq X_v) \qquad \dfrac{\mathbb{P}(X_u \neq X_v) = \mathbb{P}(X_u=0 \text{ et } X_v=1) + \mathbb{P}(X_u=1 \text{ et } X_v=0)}{\mathbb{P}(X_u \neq X_v) = \mathbb{P}(X_u=0)\mathbb{P}(X_v=1) + \mathbb{P}(X_u=1)\mathbb{P}(X_v=0)} \qquad \mathbb{P}(X_u \neq X_v) = \frac{1}{2}}{\mathbb{E}(\#C) = \frac{|E|}{2}}$$

Comme l'espérance de la taille de la coupe calculée est plus grande que $\frac{|E|}{2}$, il existe au moins une coupe de cette taille.

■

1.3 Algorithme d'approximation probabiliste en moyenne

Définition 6 Étant donné un problème d'optimisation, un algorithme probabiliste $\mathcal{A}$ est une approximation de facteur $\rho(n)$ *en moyenne* si pour toute entrée de taille n, $\mathcal{A}$ calcule une solution `sol` avec

$$\begin{cases} \mathbb{E}(\text{coût}(\texttt{sol})) \leq \rho(n)\text{coût}(\texttt{sol}^*) & \text{si c'est un problème de minimisation ;} \\ \mathbb{E}(\text{coût}(\texttt{sol})) \geq \rho(n)\text{coût}(\texttt{sol}^*) & \text{si c'est un problème de maximisation.} \end{cases}$$

Proposition 7 L'algorithme probabiliste naïf pour le calcul d'une coupe est une $\frac{1}{2}$-approximation de **MAXCUT** en moyenne.

Démonstration.

$$\frac{\mathbb{E}(\#C) = \frac{|E|}{2} \qquad \#C^* \leq |E|}{\mathbb{E}(\#C) \geq \frac{1}{2}\#C^*}$$

■

Proposition 8 La probabilité que l'algorithme naïf renvoie une coupe de taille $\geq \frac{|E|}{2}$ est d'au moins $\frac{1}{1+\frac{|E|}{2}}$. Autrement dit :

$$\mathbb{P}(\#C \geq \frac{|E|}{2}) \quad \geq \quad \frac{1}{1+\frac{|E|}{2}}.$$

Démonstration. 🎓Afin de minorer la probabilité $p := \mathbb{P}(\#C \geq \frac{|E|}{2})$, calculons $\mathbb{E}(\#C)$:

$$\begin{aligned}
\frac{|E|}{2} = \mathbb{E}(\#C) &= \sum_{i \leq |E|} i\ \mathbb{P}(\#C = i) && \text{Définition de l'espérance} \\
&= \sum_{i \leq \frac{|E|}{2}-1} i\ \mathbb{P}(\#C = i) \quad + \quad \sum_{i \geq \frac{|E|}{2}} i\ \mathbb{P}(\#C = i) && \text{Découpage de la somme} \\
&\leq (\frac{|E|}{2} - 1) \sum_{i \leq \frac{|E|}{2}-1} \mathbb{P}(\#C = i) \quad + \quad |E| \sum_{i \geq \frac{|E|}{2}} \mathbb{P}(\#C = i) && \text{Majoration des } i \\
&\leq (\frac{|E|}{2} - 1)\mathbb{P}(\#C < \frac{|E|}{2}) \quad + \quad |E|\ \mathbb{P}(\#C \geq \frac{|E|}{2}) && \text{Probabilité d'unions disjointes} \\
&= (\frac{|E|}{2} - 1)(1-p) \quad + \quad |E|\ p && \text{Renommage} \\
&= \frac{|E|}{2} - 1 \quad + \quad p(1 + \frac{|E|}{2}). && \text{Regroupement}
\end{aligned}$$

D'où la proposition.

■

Proposition 9 Le nombre moyen de répétitions de l'algorithme naïf pour trouver une coupe de taille au moins $\frac{|E|}{2}$ est $1 + \frac{|E|}{2}$.

Démonstration. Quand on répète un tirage de Bernouilli de probabilité q, alors l'espérance du nombre de répétitions T jusqu'à succès est $\frac{1}{q}$. En effet :

$$
\begin{aligned}
\mathbb{E}(T) &= \sum_{i\geq 1} \mathbb{P}(T \geq i) && \text{Définition alternative de l'espérance}\\
&= \sum_{i\geq 1} (1-q)^{i-1} && \text{au moins } i-1 \text{ échecs}\\
&= \sum_{i\geq 0} (1-q)^{i} && \text{décalage de l'indice}\\
&= \frac{1}{1-(1-q)} && \text{série convergente}\\
&= \frac{1}{q} = 1 + \frac{E}{2} && \text{car ici } q = \frac{1}{1+\frac{|E|}{2}}
\end{aligned}
$$

■

Remarque 10 On obtient donc un algorithme de type Las Vegas pour le calcul d'une coupe contenant au moins la moitié des arêtes.

2 Dérandomisation

Question 11 Les algorithmes probabilistes sont-ils nécessaires ?

2.1 Méthode des probabilités conditionnelles

Cette approche est dûe à Erdös et Selfridge [ES73]. Illustrons la sur l'algorithme naïf probabiliste pour **MAXCUT**. On note C la coupe calculée par l'algorithme probabiliste, et $\#C$ sa taille. Il y a une exécution de l'algo probabiliste naïf avec $\#C \geq \frac{|E|}{2}$.

Idée de l'algorithme déterministe.
Placer de manière déterministe les sommets dans V_0 ou V_1 de façon à conserver l'invariant

$$\mathbb{E}(\#C \mid \text{choix déterministes déjà faits}) \geq \frac{|E|}{2}.$$

Les sommets sont notés $V = \{1, \ldots, n\}$. On note, pour tout sommet u :

$$X_u = \begin{cases} 0 & \text{si } u \in V_0 \\ 1 & \text{si } u \in V_1. \end{cases}$$

Notation 12 On note $X_{1..i} = x_{1..i}$ au lieu de $X_1{=}x_1, \ldots, X_i{=}x_i$.

entrée : un graphe non orienté $G = (V, E)$
sortie : une coupe (V_0, V_1) de taille plus grande que $\frac{|E|}{2}$
fonction gloutonMaxCutApprox($G = (V, E)$)
  pour tout sommet $i = 1..n$ **faire**
    //les $x_1, \ldots, x_{i-1}$ ont déjà été fixés
    soit $x_i \in \{0, 1\}$ avec $\mathbb{E}(\#C \mid X_{1..i}{=}x_{1..i}) \geq \frac{|E|}{2}$
renvoyer $V_0 = \{i \mid X_i = 0\}$, $V_1 = \{i \mid X_i = 1\}$

L'algorithme déterministe consiste à choisir $x_i \in \{0, 1\}$ avec $\mathbb{E}(\#C \mid X_{1..i}{=}x_{1..i}) \geq \frac{|E|}{2}$.

Proposition 13 On a l'invariant suivant. Au i-ème tour de boucle,

$$\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}) \geq \frac{|E|}{2}.$$

Démonstration.
Initialisation. Au début, $\mathbb{E}(\#C) \geq \frac{|E|}{2}$ donc tout va bien.
Induction. Supposons que nous ayons choisi $x_1, \dots, x_{i-1} \in \{0,1\}$ avec

$$\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}) \geq \frac{|E|}{2}.$$

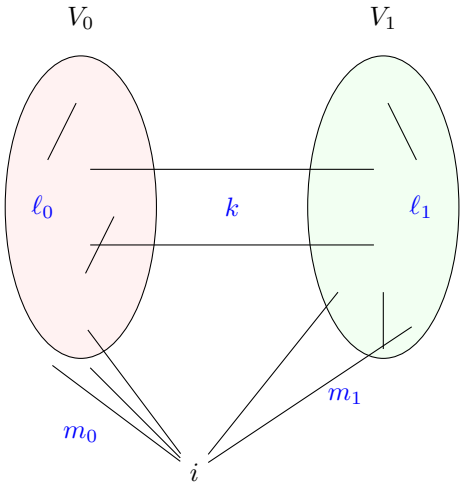
Or

$$\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}) \quad = \quad \frac{1}{2}\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}0) \quad + \quad \frac{1}{2}\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}1)$$

et donc soit $\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}0) \geq \frac{|E|}{2}$ ou $\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}1) \geq \frac{|E|}{2}$.
■
Maintenant, on va chercher à rendre l'algorithme plus concret pour pouvoir décider comment choisir x_i.

Proposition 14

$$\mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}0) = k + m_1 + \frac{\alpha}{2} \quad \text{et} \quad \mathbb{E}(\#C \mid X_{1..i-1}{=}x_{1..i-1}, X_i{=}1) = k + m_0 + \frac{\alpha}{2}$$



avec
- deux ensembles $V_0 := \{j \in \{1,\dots,i-1\} \mid X_j = 0\}$ et $V_1 := \{j \in \{1,\dots,i-1\} \mid X_j = 1\}$;
- ℓ_0 arêtes entre sommets de V_0 ;
- ℓ_1 arêtes entre sommets de V_1 ;
- k arêtes entre un sommet de V_0 et un de V_1 ;
- m_0 arêtes entre sommets de V_0 et le sommet i ;
- m_1 arêtes entre sommets de V_1 et le sommet i. ;
- $\alpha = |E| - m_0 - m_1 - k - \ell_0 - \ell_1$ arêtes touchant un sommet de $\{i+1,\dots,n\}$.

Démonstration. Les arêtes entre sommets de la même patate ne comptent pas. Les arêtes entre les deux patates comptent chacunes pour 1. Enfin, seules les arêtes reliant i à la patate dans laquelle i ne va pas être comptent. Enfin, les arêtes touchant un sommet de $\{i+1,\dots,n\}$ ne sont pas encore choisies donc on a une contribution de $\frac{1}{2}$. ■
🎓écrire le détail en maths

2.2 Algorithme déterministe

🎓écrire l'algo plus joliment

entrée : un graphe non orienté $G = (V, E)$
sortie : une coupe (V_0, V_1) de taille plus grande que $\frac{|E|}{2}$
fonction gloutonMaxCutApprox($G = (V, E)$)
    $V_0 := \emptyset$
    $V_1 := \emptyset$
    pour tout sommet $v \in V$ **faire**
        si v a plus de voisins dans X que dans Y **alors**
            $V_0 := V_0 \cup \{v\}$
        sinon
            $V_0 := V_1 \cup \{v\}$
    renvoyer (V_0, V_1)

2.3 État de l'art

Le meilleur ratio pour **MAXCUT** est environ 1/0.878, voir [GW95]. C'était la première fois que la programmation semidéfinie est utilisée pour concevoir un algo d'approximation. C'est une forme d'optimisation avec des expressions quadratiques. Une instance se transforme en :

$$\begin{array}{l}\text{maximiser } \sum_{(i,j)\in E} \frac{1-v_i v_j}{2} \\ \left\{ \; v_i \in \{-1,1\}\right.\end{array}$$

où $v_i = -1$ signifie que $i \in X$ et $v_i = 1$ que $i \in Y$. Puis ils font de la relaxation, et algo probabiliste (on fera pareil après pour **MAXSAT**). La borne 1/0.878 est optimale si la *conjecture des jeux uniques* est vraie [KKMO07].

3 Théorème d'Adleman

3.1 Énoncé

On suppose qu'un problème de décision L est un langage sur l'alphabet $\{0,1\}$.

Définition 15 *RP* est la classe des problèmes L qui admettent chacun un algorithme probabiliste A en temps polynomial tel que pour tout instance x :
- Si $x \in L$ alors $\mathbb{P}(A(x)$ renvoie 'oui'$) \geq \frac{1}{2}$;
- Si $x \notin L$ alors $\mathbb{P}(A(x)$ renvoie 'non'$) = 1$.

Proposition 16 Un problème L est dans *RP* s'il existe un algorithme déterministe $A(.,.)$ en temps polynomial, et un polynôme Q avec :
- A s'exécute en temps polynomial ;
- Si $x \in L$ alors $\mathbb{P}(A(x,R)$ renvoie 'oui'$) \geq \frac{1}{2}$ où R est une variable aléatoire qui choisit uniformément un mot dans $\{0,1\}^{Q(|x|)}$;
- Si $x \notin L$ alors pour tout mot $r \in \{0,1\}^{Q(|x|)}$, on a $A(x,r)$ renvoie 'non'.

Théorème 17 (Théorème d'Adleman) [Adl78] Tout problème de décision L dans *RP* peut se dérandomiser de manière non-uniforme : il existe un polynôme P, et pour tout entier n, il existe un algorithme déterministe qui décide l'ensemble des instances positives de taille n en temps $P(n)$.

Définition 18 P/*poly* est la classe des problèmes de décision L qui admette un algorithme A en temps polynomial, et une suite de mots $\alpha_0, \alpha_1, \ldots$ de longueur $poly(n)$ tels que pour toute instance de taille n,

$$x \in L \text{ ssi } A(x, \alpha_n) \text{ renvoie 'oui'.}$$

Théorème 19 (théorème d'Adleman) $\mathrm{RP} \subseteq \mathrm{P}/poly$.

3.2 Démonstration

Nous allons exhiber un algorithme déterministe qui décide exactement l'ensemble des instances positives de L de taille n.

Soit $L \in RP$. Il existe donc un algorithme $A(.,.)$ déterministe en temps polynomial et un polynôme Q avec pour toute instance x :
- Si $x \in L$ alors $\mathbb{P}(A(x,R)$ renvoie 'oui'$) \geq \frac{1}{2}$;
- Si $x \notin L$ alors $\mathbb{P}(A(x,R)$ renvoie 'non'$) = 1$.

Fixons n. Considérons maintenant l'algorithme B qui prend en entrée x de taille n ainsi que $n+1$ mots $r_1, \ldots, r_{n+1} \in \{0,1\}^{Q(n)}$:

```
fonction B(x, r_1, ..., r_{n+1})
    pour i := 1..n+1 faire
        si A(x, r_i) renvoie 'oui' alors
            renvoyer 'oui'
    renvoyer 'non'
```

Fait 20 Soit $x \in \overline{L} \cap \{0,1\}^n$. On a pour tout $r_1, \ldots, r_{n+1} \in \{0,1\}^{Q(n)}$, $B(x, r_1, \ldots, r_{n+1})$ renvoie 'non'.

Fait 21 Soit $x \in L \cap \{0,1\}^n$. Soit $R_1, \ldots, R_{n+1}$ des variables aléatoires indépendantes qui donnent un mot dans $\{0,1\}^{Q(n)}$ de manière uniforme.

$$\mathbb{P}(B(x, R_1, \ldots, R_{n+1}) \text{ renvoie 'non'}) \leq \frac{1}{2^{n+1}}$$

Démonstration.

$$\begin{aligned}\mathbb{P}(B(x,R_1,\dots,R_{n+1}) \text{ renvoie 'non'}) &= \mathbb{P}(A(x,R_1) \text{ et } \dots \text{ et } A(x,R_{n+1}) \text{ renvoient non})\\ &= \mathbb{P}(A(x,R_1) \text{ renvoie non}) \times \dots \times \mathbb{P}(A(x,R_{n+1}) \text{ renvoient non})\\ &\qquad \text{car les } R_i \text{ sont des variables indépendantes}\\ &\leq \frac{1}{2} \times \dots \times \frac{1}{2} = \frac{1}{2^{n+1}}.\end{aligned}$$

■

Corollaire 22

$$\mathbb{P}(\text{pour tout } x \in L \cap \{0,1\}^n,\ B(x,R_1,\dots,R_{n+1}) \text{ renvoie 'oui'}) > 0.$$

Démonstration.

$$\begin{aligned}\mathbb{P}(\text{il existe } x \in L \cap \{0,1\}^n,\ B(x,R_1,\dots,R_{n+1}) \text{ renvoie 'non'}) &= \mathbb{P}\Big(\bigcup_{x\in L\cap\{0,1\}^n} \{B(x,R_1,\dots,R_{n+1}) \text{ renvoie 'non'}\}\Big)\\ &\leq \sum_{x\in L\cap\{0,1\}^n} \mathbb{P}(B(x,R_1,\dots,R_{n+1}) \text{ renvoie 'non'})\\ &\leq \sum_{x\in L\cap\{0,1\}^n} \frac{1}{2^{n+1}} \text{ par le Fait 21}\\ &\leq \frac{2^n}{2^{n+1}} = \frac{1}{2} \text{ car } |L\cap\{0,1\}^n| \leq 2^n.\end{aligned}$$

■

Théorème 23 il existe des mots $r_1,\dots,r_{n+1}$ de longueur $Q(n)$ tel que $B(\_,r_1,\dots,r_{n+1})$ décide $L\cap\{0,1\}^n$.

Démonstration.

Par la méthode probabiliste, il existe des mots $r_1,\dots,r_{n+1}$ de longueur $Q(n)$ tel que pour tout $x \in L\cap\{0,1\}^n$, $B(x,r_1,\dots,r_{n+1})$ renvoie 'oui'.

Et on a vu que pour tout mot, donc en particulier pour ces mots $r_1,\dots,r_{n+1}$ là, pour tout $x \in \overline{L}\cap\{0,1\}^n$, $B(x,r_1,\dots,r_{n+1})$ renvoie 'oui'. ■

Références

[Adl78] Leonard M. Adleman. Two theorems on random polynomial time. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 75–83. IEEE Computer Society, 1978.

[ES73] Paul Erdoes and JL Selfridge. On a combinatorial game. *Journal of Combinatorial Theory, Series A*, 14(3) :298–301, 1973.

[GW95] Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6) :1115–1145, 1995.

[KKMO07] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps ? *SIAM Journal on Computing*, 37(1) :319–357, 2007.

ALGO2 – Chaînes de Markov

François Schwarzentruber

3 avril 2024

1 Motivation

Les chaînes de Markov interviennent dans trois grandes types de problèmes algorithmiques que voici.

1.1 Génération aléatoire

Choisir un crayon parmi 5 crayons, c'est facile. Là, on s'intéresse à générer des objets parmi un ensemble de taille exponentielle. Mais choisir un élément de $\{0,1\}^n$ c'est facile. Là, en plus, les objets sont contraints.

On rappelle qu'un ensemble indépendant dans un graphe $G = (V, E)$ est un sous-ensemble $I \subseteq V$ avec pour tout $(u, v) \in E$, $u \notin I$ ou $v \notin I$.

Génération uniforme d'ensembles indépendants
entrée : G graphe non orienté
sortie : un ensemble indépendant dans G, tiré aléatoirement de manière uniforme dans l'ensemble des ensembles indépendants

Génération uniforme d'une q-coloration d'un graphe
entrée : G graphe non orienté
sortie : une q-coloration de G, tirée aléatoirement de manière uniforme dans l'ensemble des q-colorations

Exemple 1 (exemple pédagogique mais ça ne fonctionne pas) Pour générer une carte aléatoire dans un jeu vidéo, on pourrait :

— démarrer avec un terrain vide s_0
— obtenir s_{t+1} depuis s_t en ajoutant une route, supprimant une route, ajoutant une maison, supprimant une maison

Pour t assez grand, on récupère s_t. On peut aussi récupérer plusieurs états : s_t, $s_{t+\tau}$, $s_{t+2\tau}$, etc.

Il faut que t soit grand pour être sûr d'être proche de la distribution limite. Il faut que τ soit grand pour que les états soient indépendants.

$\rightsquigarrow$ Simulation d'une chaîne de Markov

On peut aussi estimer l'espérance d'une quantité. Par exemple, l'espérance de la taille d'un ensemble indépendant tiré uniformément.

1.2 Problème d'optimisation

Ensemble indépendant maximal
entrée : G graphe non orienté
sortie : un ensemble indépendant maximal dans G

Voyageur de commerce
entrée : G graphe complet pondéré
sortie : un tour minimal dans G

Exemple 2 L'idée est :

— démarrer avec un tour s_0 disons aléatoire.
— obtenir s_{t+1} depuis s_t en modifiant un tout petit peu le tour

$\rightsquigarrow$ Recuit simulé

1.3 Problème de comptage

Compter le nombre d'ensembles indépendants
entrée : G graphe non orienté
sortie : le nombre d'ensembles indépendants

Compter le nombre de q-coloration d'un graphe
entrée : G graphe non orienté
sortie : le nombre de q-colorations de G

$\rightsquigarrow$ lire le chapitre 9 de [H+02]. C'est un peu compliqué

2 Chaîne de Markov

Pour pouvoir expliquer comment des algorithmes pour ces problèmes fonctionnent, on est obligé de comprendre la notion de chaîne de Markov. C'est le but de cette section.

Définition 3 Une **chaîne de Markov** est une suite de variables aléatoires $\mathcal{M} = (X_t)_{t\in\mathbb{N}}$ à valeur dans $\mathcal{S}$ telles que pour toute suite $s_0 \cdots s_t, s_{t+1} \in \mathcal{S}$, $\mathbb{P}(X_{t+1} = s_{t+1} \mid X_t = s_t \cdots X_0 = s_0) = \mathbb{P}(X_{t+1} = s_{t+1} \mid X_t = s_t)$.

Définition 4 On dit qu'une chaîne de Markov est **homogène**, si pour tout $(s, s') \in \mathcal{S}$ la valeur $\mathbb{P}(X_{t+1} = s' \mid X_t = s)$ est indépendante de t, c'est-à-dire pour tout $t, t' \in \mathbb{N}$, $\mathbb{P}(X_{t+1} = s' \mid X_t = s) = \mathbb{P}(X_{t'+1} = s' \mid X_{t'} = s)$.

Dans la suite, on ne considère que des **chaîne de Markov homogènes**.

2.1 Représentation matricielle

Si $\mathcal{S}$ est fini (voire dénombrable), une chaîne de Markov homogène $\mathcal{M}$ se représente par la **distribution initiale** μ_0 (celle de X_0) et une matrice $P \in [0,1]^{\mathcal{S}^2}$ avec pour tout $(s, s') \in \mathcal{S}$,

$$P_{s,s'} = \mathbb{P}(X_{t+1} = s' \mid X_t = s).$$

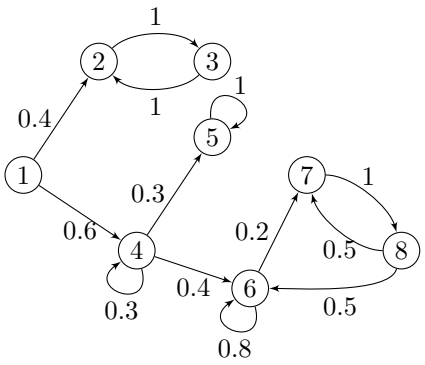
2.2 Représentation avec un graphe orienté

Parfois, on représente la matrice par le graphe dont P est la matrice d'adjacence. Autrement dit on considère le **graphe** orienté pondéré $G = (\mathcal{S}, \to, P)$ qui décrit l'évolution d'une étape. Les sommets du graphe sont les éléments de $\mathcal{S}$. Pour tout $(s, s') \in \mathcal{S}$, si $P_{s,s'} = \mathbb{P}(X_{t+1} = s' \mid X_t = s) > 0$ il y a une arc entre s et s', pondérée par $P_{s,s'}$. Autrement dit :

$$\to = \{(s, s') \in \mathcal{S}^2 \mid P_{s,s'} > 0\}.$$

Le point de l'arc $s \to s'$ est $P_{ss'}$. Dans la suite, si $P_{ss'}$ alors il n'y a pas d'arc de s à s'.

Exemple 5



$$P = \begin{pmatrix} 0 & 0.4 & 0 & 0.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0.3 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.8 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 \end{pmatrix}$$

Dans la matrice P, la ligne pour l'état s donne les probabilités d'être dans un état s' quand on part de l'état s. Ainsi, la somme des nombres sur la ligne s vaut 1. De manière équivalente, la somme des poids des arcs sortants d'un sommet vaut 1.

2.3 Distribution de probabilité

Si μ est une distribution sur les états à l'instant t vu comme un vecteur ligne, alors μP est la distribution sur les états à l'instant $t+1$. En effet :

$$\begin{aligned}\mathbb{P}(X_{t+1}=s') &= \sum_{s\in\mathcal{S}} \underbrace{\mathbb{P}(X_t=s)}_{\mu_s} \times \underbrace{\mathbb{P}(X_{t+1}=s' \mid X_t=s)}_{P_{ss'}} \\ &= (\mu P)_{s'}\end{aligned}$$

En itérant, si μ est une distribution sur les états à l'instant t vu comme un vecteur ligne, alors μP^n est la distribution sur les états à l'instant $t+n$.

Proposition 6 La distribution de probabilité de l'état à l'instant t est $\mu_0 P^t$.

3 Propriétés

La motivation pour générer des objets de manière aléatoire est de dire qu'un état est un objet (par exemple, un état est un ensemble indépendant). Les transitions, elles, on verra... Puis, on regarde la distribution de probabilité sur les états après une longue balade dans la chaîne. On s'intéresse donc à l'éventuelle **distribution limite**.

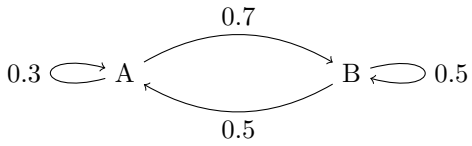
3.1 Irréductibilité

La moindre des choses est d'être sûr d'aller partout (par exemple, dans l'exemple pour générer un ensemble indépendant, il faut que l'on puisse atteindre n'importe quel ensemble indépendant !).

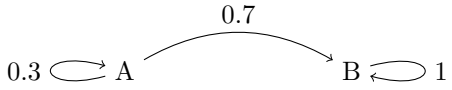
La notion d'irréductibilité veut dire que l'on peut aller partout depuis n'importe où.

Définition 7 $\mathcal{M} = (X_k)_{k\in\mathbb{N}}$ est **irréductible** si pour toute paire d'états $s, t \in \mathcal{S}$, il y a un chemin de s de t, ou de façon équivalente s'il existe $k \in \mathbb{N}$ tel que $(P^k)_{s,t} > 0$.

Exemple 8 (chaîne irréductible)



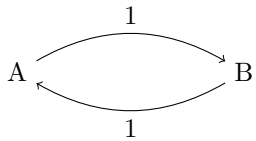
Exemple 9 (chaîne réductible) Cette chaîne est réductible car une fois aller en B on ne peut plus revenir en A :



3.2 Apériodicité

Comme on s'intéresse à une éventuelle distribution limite, il ne faut pas de phénomène périodique comme dans la chaîne suivante :

Exemple 10 (chaîne périodique)

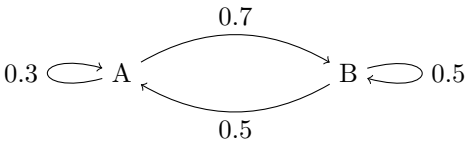


Si $X_0 = A$, alors on est en A à tous les temps pairs, et en B à tous les temps impairs.

L'intuition est donc de fournir une définition pour se débarrasser de phénomènes périodiques.

Définition 11 $\mathcal{M}$ est **apériodique** si pour tout état $s \in \mathcal{S}$, le pgcd de la longueur des cycles autour de s est 1 : $\text{pgcd}\{n \geq 1 \mid (P^n)_{s,s} > 0\} = 1$.

Exemple 12 (chaîne apériodique)



4 Convergence

On peut s'atteindre à ce que la distribution limite soit stationnaire.

Définition 13 Soit $\mu \in Dist(\mathcal{S})$ une distribution. μ est **stationnaire** si $\mu P = \mu$.

Théorème 14 Soit $\mathcal{M}$ une chaîne de Markov **irréductible et apériodique**. Alors :

1. Il existe une **unique distribution stationnaire** μ ;
2. $\mu_0 P^n \xrightarrow{n\to+\infty} \mu$.

Démonstration. Voir chapitre 5 de [H+02]. La démonstration est divisée comme suit. Considérons $\mathcal{M}$ une chaîne de Markov irréductible et apériodique. On montre que :

1. il existe une distribution stationnaire ;
2. si μ est stationnaire, alors $\mu_0 P^n \xrightarrow{n\to+\infty} \mu$;
3. Il y a unicité de la distribution stationnaire (par le point précédent).

■

5 Réversibilité

Définition 15 μ est **réversible** si pour tous états $s, s' \in \mathcal{S}$, $\mu(s) \times P_{s,s'} = \mu(s') \times P_{s',s}$.

Définition 16 On dit que la chaîne de Markov est **réversible** si elle admet une distribution réversible.

Proposition 17 Si μ est réversible, alors μ est stationnaire.

Démonstration. Soit s' un état. On a :

$$\begin{aligned}
\mu_{s'} &= \mu_{s'} \times 1 \\
&= \mu_{s'} \sum_s P_{s',s} && \text{car la somme sur la ligne } s' \text{ vaut } 1 \\
&= \sum_s \mu_{s'} P_{s',s} \\
&= \sum_s \mu_s P_{s,s'} && \text{car } \mu \text{ réversible} \\
&= (\mu P)_{s'}
\end{aligned}$$

Donc $\mu = \mu P$. ■

Corollaire 18 Si $\mathcal{M}$ est **irréductible et apériodique** et μ est **réversible** alors

$$\mu_0 P^n \xrightarrow{n\to+\infty} \mu.$$

6 Échantillonnage d'objets selon une certain loi

Échantillonnage d'un ensemble indépendant

entrée : un graphe G non orienté
sortie : un ensemble indépendant $I \subseteq V$ choisi aléatoirement parmi tous les ensembles indépendants

De manière abstraite, soit π une distribution sur un ensemble $\mathcal{S}$. Objectif : générer un objet $s \sim \pi$.

6.1 Algorithme MCMC

Un algorithme MCMC est un algorithme qui fonctionne comme suit. On définit une chaîne de Markov $\mathcal{M}$ irréductible et apériodique avec π comme unique distribution stationnaire. Autrement dit, chaque état est un objet possible (e.g. un ensemble indépendant).

fonction MCMC()
soit n un entier grand (disons 10000) simuler n étapes de la chaîne de Markov $s_0, s_1, \ldots, s_n$
renvoyer s_n

Il faut espérer que X_{100000} est un objet aléatoire qui suit à peu près π.

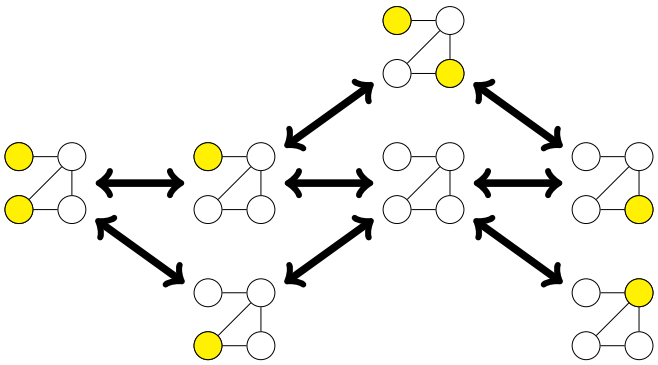
6.2 Exemple : générer un ensemble indépendant selon loi uniforme

Définition de la chaîne de Markov $\mathcal{M}$ pour générer un ensemble indépendant dans G. On pose $\mathcal{S}$ comme la collection des sous-ensembles de V qui sont indépendants.

Posons $X_0 = \emptyset$: la distribution initiale de X_0 est donc concentrée sur l'ensemble vide, qui est un ensemble indépendant. La fonction de transition de cette chaîne de Markov est ensuite définie comme suit. Donnons un algorithme qui à un état $s \in \mathcal{S}$ (un ensemble indépendant donc), en associe un autre de manière probabiliste :

fonction etatSuivant(s)
choisir $v \in V$ au hasard
tirer une pièce à pile ou face
si face et $s \cup \{v\}$ est indépendant
  renvoyer $s \cup \{v\}$
sinon
  renvoyer $s \setminus \{v\}$ Remarque : v n'appartient pas nécessairement à s.

Exemple 19 Avec le graphe $G = (V, E)$ à 4 nœuds  , on obtient la chaîne de Markov suivante à 8 états :



Théorème 20 La chaine converge vers la distribution uniforme sur les ensembles indépendants.

Démonstration. Nous allons utiliser le corollaire 18. Montrons les hypothèses.

Irréductible Cette chaîne de Markov est irréductible. En effet, depuis tout état $S \in \mathcal{S}$, la probabilité est non nulle d'atteindre l'ensemble vide, et depuis l'ensemble vide, tous les ensembles indépendants sont accessibles.

Apériodique La chaîne est apériodique. En effet, pour tout état $s \neq \emptyset$, on a $P_{ss} > 0$. En effet, depuis tout état $s \neq \emptyset$, il y a probabilité au moins $\frac{1}{2} \times \frac{1}{|V|}$ de choisir face et puis de choisir un sommet de V déjà présent, et donc que l'état suivant soit encore s. Ainsi, il y a un cycle de longueur 1 autour de s, et s est apériodique.

Pour $s = \emptyset$, alors il y a probabilité $\frac{1}{2}$ de choisir pile et donc que le successeur soit encore s.

Distribution réversible. Soit μ la distribution uniforme sur les ensembles indépendants : pour tout $S \in \mathcal{S}$, $\mu(S) = \frac{1}{|\mathcal{S}|}$. Montrons que pour tout $(s, s') \in \mathcal{S}$, $P_{s,s'} = P_{s',s}$, en considérant le nombre de sommets dans la différence symétrique Δ de s et s', autrement dit $\Delta := (s \cup s') \setminus (s \cap s')$.

— Si $|\Delta| = 0$, alors $s = s'$, et on a trivialement égalité.
— Si $|\Delta| \geq 2$, alors les deux valeurs sont nulles : on ne peut pas passer de s à s' ou réciproquement en une étape.

— Enfin, si $|\Delta| = 1$, supposons sans perte de généralité que $s' = s \cup \{v\}$. Depuis l'état s, calculons la probabilité que le prochain état soit s'. Pour cela, il faut que v soit choisi, et que le résultat du tirage soit face. Ainsi, $\mathbb{P}(s, s') = \frac{1}{2|V|}$. À présent, calculons la probabilité de passer de s' à s en une étape : il faut que v soit choisi, et que le résultat du tirage au sort soit pile, donc là encore $\mathbb{P}(s', s) = \frac{1}{2|V|}$.

Puisque la distribution μ, uniforme sur $\mathcal{S}$, est réversible, elle est aussi stationnaire, d'après la proposition précédente. Calculer la taille moyenne d'un ensemble indépendant peut donc se faire en évaluant la distribution stationnaire de la chaîne de Markov.

■

6.3 Échantillonnage de Gibbs

L'exemple qu'on vient de voir illustre l'**échantillonage de Gibbs**. L'objectif de l'échantillonage de Gibbs est générer aléatoirement des éléments dans Y^V selon une loi μ.

Exemple 21 Dans l'exemple des ensembles indépendants, on considère $\{0,1\}^V$: la valeur 1 indique que v est dans l'ensemble indépendant courant, 0 qu'il est dehors. La probabilité μ est la loi uniforme sur les ensembles indépendants.

L'échantillonage de Gibbs fonctionne comme cela : à chaque transition dans la chaîne de Markov, on modifie uniquement la valeur d'un seul sommet $v \in V$ choisi aléatoirement. on La valeur $s(v)$ est choisie aléatoirement selon la **probabilité conditionnelle sachant les valeurs des autres coordonnées**. Plus formellement, si X suit la loi μ, on considère la distribution sur l'ensemble Y donnée par le vecteur suivant :

$$(\mathbb{P}(X[v] = y \mid X = s \text{ sauf pour la coordonnée } v))_{y \in Y}$$

Encore plus formellement :

$$(\mathbb{P}(X[v] = y \mid \text{ pour tout } v' \in V \setminus \{v\},\ X_{v'} = s[v']))_{y \in Y}$$

On note $\pi(s_v \mid (s_{v'})_{v' \neq v})$ ce vecteur.

fonction etatSuivant(s)
  choisir $v \in V$ au hasard
  choisir $y \in Y$ selon $\pi(s_v \mid (s_{v'})_{v' \neq v})$
  $s' :=$ une copie de s
  $s'_v := y$
  renvoyer s'

7 Application : résoudre un problème d'optimisation

On utilise toujours MCMC mais avec une distribution stationnaire qui donne beaucoup de chances d'arriver dans une solution optimale.

1. On définit une chaîne de Markov $\mathcal{M}$ où chaque état est une solution possible
2. On se balade dans $\mathcal{M}$.
3. On espère que X_{100000} est une solution optimale (ou quasi-optimale)

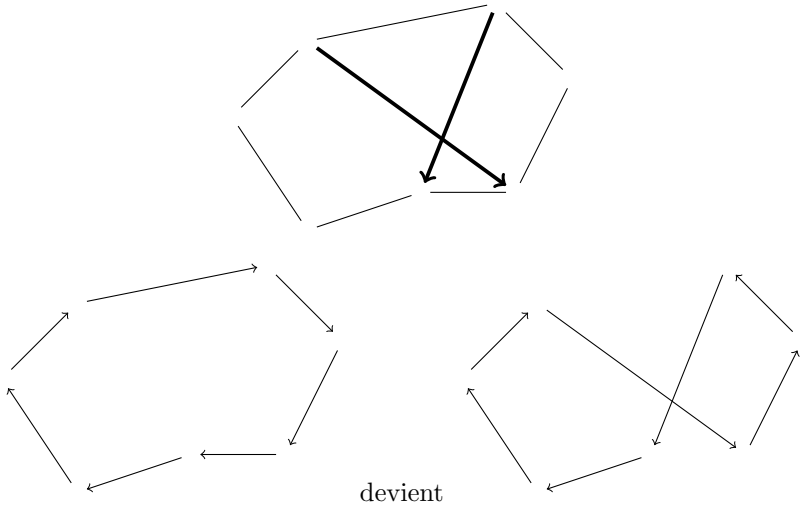
7.1 Algorithme du recuit simulé

Dans cette section, on illustre l'approche sur le voyageur de commerce. On considère une instance du voyageur de commerce où les villes sont : 1, 2, ..., m.

On note $\mathcal{S}$ l'ensemble des tours, c'est aussi l'ensemble des états de la chaîne de Markov (i.e. un tour est un état). Les tours sont notés s, s', etc. Chaque tour est une permutation de $\{1, \ldots, m\}$. On note coût(s) le coût d'un tour s. Autrement dit :

$$\text{coût}(s) := \sum_{i=1}^{m-1} \text{coût}(s_i s_{i+1}) + \text{coût}(s_m s_1).$$

L'idée est de passer d'un tour à un autre en permutant un morceau du tour :



devient

On dit que deux tours s et s' sont voisins s'il existe $i, j \in \{1, \dots, m\}$ avec $i < j$ et si on note

$$s = (s_1, s_2, \dots, s_{i-1}, s_i, s_{i+1}, \dots, s_{j-1}, s_j, s_{j+1}, \dots, s_m)$$

alors

$$s' = (s_1, s_2, \dots, s_{i-1}, s_j, s_{j-1}, \dots, s_{i+1}, s_i, s_{j+1}, \dots, s_m)$$

Chaque voisin de s est déterminé par les indices (i, j) avec $i < j$. On a donc $\frac{m(m-1)}{2}$ voisins pour chaque état.

```
fonction recuitSimulé(G)
    T := température assez élevée
    tant que vous en avez envie
        s' := choisir un voisin de s de manière uniforme
        si coût(s') < coût(s) ou avec une probabilité de e^{-(coût(s')-coût(s))/T} alors
            s' := s
        Faire baisser la température T
    renvoyer s
```

Cet algorithme est proche d'un algorithme de **recherche locale**. On va toujours dans une solution meilleure, mais parfois on s'autorise des sauts. Les sauts sont contrôlés par un paramètre T est appelé **température**. Plus T est petit, plus l'exposant dans l'exponentielle est grand en valeur absolue. Donc la probabilité de sauter devient de plus en plus petite.

Plus coût(s') est grand devant coût(s), plus l'exposant dans l'exponentielle est grand en valeur absolue. Et donc pareil, plus la probabilité de sauter est petite.

```
fonction recuitSimulé(G)
    T := température assez élevée
    tant que vous en avez envie
        s' := choisir un voisin de s de manière uniforme
        avec probabilité min(e^{-(coût(s')-coût(s))/T}, 1) alors
            s' := s
        Faire baisser la température T
    renvoyer s
```

7.2 Distribution stationnaire souhaitée

On fait une analyse pour T fixé.

Définition 22 Soit $T > 0$. La **distribution de Boltzmann** π_T est définie par :

$$\pi_T(s) \propto e^{\frac{-\text{coût}(s)}{T}}.$$

Avec cette distribution, plus un tour est bon, plus il est probable. Le paramètre T est appelé **température**. Plus T est petit, plus cette distribution est fine ; plus T est grand plus elle est étalée.

La distribution π_T est celle que l'on souhaite être stationnaire. Dans la suite, on la note π.

On remarque que la définition de π_T ne requiert pas de connaître les solutions optimales (encore heureux !) mais juste de savoir évaluer la fonction de coût.

7.3 Voir le recuit simulé comme une balade dans une chaîne de Markov

Pour cela, on va appliquer une recette de cuisine pour définir une chaîne de Markov où π est stationnaire : il s'agit de la **chaîne de Metropolis** associée à π (c'et un cas particulier des **chaînes de Metropolis-Hastings**).

On considère une relation symétrique de **voisinage** : on obtient un **graphe non orienté** entre états. On suppose aussi que s est voisin de s.

Exemple 23 Les états sont tous les tours possibles.

Définition de la chaîne de Markov de Metropolis. Donnons quand même le cadre général pour des sommets de degrés quelconques :

$$\mathbb{P}(s,s') := \begin{cases} \frac{1}{d^\circ(s)} \min\left(\frac{\pi(s')d^\circ(s)}{\pi(s)d^\circ(s')}, 1\right) & \text{si } s \text{ et } s' \text{ voisins} \\ 0 & \text{si } s \neq s' \text{ et } s \text{ et } s \text{ ne sont pas voisins} \\ 1 - \sum_{v \text{ voisin de } s} \frac{1}{d^\circ(s)} \min\left(\frac{\pi(v)d^\circ(s)}{\pi(s)d^\circ(v)}, 1\right) & \text{si } s = s' \end{cases}$$

Autrement dit quand on se balade, voici l'algorithme qui retourne l'état suivant :

fonction etatSuivant(s)
  choisir s' de manière uniforme parmi les voisins de s
  avec probabilité $\min\left(\frac{\pi(s')d^\circ(s)}{\pi(s)d^\circ(s')}, 1\right)$
    renvoyer s'
  sinon
    renvoyer s

Exemple 24 Sur l'exemple du voyageur de commerce, cela donne :

fonction etatSuivant(s)
  choisir s' de manière uniforme parmi les voisins de s
  avec probabilité $\min\left(e^{-(\text{coût}(s')-\text{coût}(s))/T‘}, 1\right)$
    renvoyer s'
  sinon
    renvoyer s

On voit que le recuit simulé c'est une chaîne de Metropolis, avec une distribution de Boltzmann comme distribution stationnaire + une température qui décroit.

Théorème 25 Si le graphe non orienté est connexe, alors π est l'unique distribution stationnaire pour la chaîne de Markov ci-dessus. 🎓

Démonstration. La chaîne de Markov est bien irréductible et apériodique. Comme d'habitude, on montre que π est une distribution réversible.

Irréductible Car le graphe est une composante.

Apériodique Car il y a des boucles : on peut toujours aller de s à s' avec une probabilité non nulle.

Réversible Montrons que $\pi_s P_{s,s'} = \pi_{s'} P_{s',s}$.
Si $s = s'$, alors c'est vrai.
Si s et s' ne sont pas voisins, c'est bon aussi car $P_{s,s'} = P_{s',s} = 0$.
Sinon, soit s et s' voisins avec $s \neq s'$.
Si $\frac{\pi_{s'} d_s}{\pi_s d_{s'}} \geq 1$, alors

$$\begin{cases} \pi_s P_{s,s'} = \pi_s \frac{1}{d_s} \\ \pi_{s'} P_{s,s'} = \pi_{s'} \frac{1}{d_{s'}} \frac{\pi_s d_{s'}}{\pi_{s'} d_s} = \pi_s \frac{1}{d_s} \end{cases}$$

Sinon, dans l'autre cas, c'est pareil, symétrique.
■

Algorithme du recuit simulé

Théorème 26

$$\mathbb{P}(Y \text{ choisi selon } \pi_T \text{ soit tel que } f(Y) = min_{s\in\mathcal{S}} f(s)) \xrightarrow{T\to 0} 1.$$

L'algorithme consiste à se balader dans la chaîne de Markov en faisant décroitre la température doucement vers 0. La chaîne de Markov résultante n'est pas donc pas homogène.

8 Simulation d'une chaîne de Markov

Pour estimer la distribution stationnaire, autant simuler la chaîne de Markov. Soit $\mathcal{M} = (\mathcal{S}, \mu_0, P)$ une chaîne de Markov. Voyons comment simuler le comportement de $\mathcal{M}$. On note

$$\mathcal{M} = \{s_1, \dots, s_n\}.$$

Tirer un état initial selon μ_0. Initialement, on tire une valeur aléatoire r_0 uniformément dans $[0, 1]$. La valeur de r détermine l'état initial de la chaîne de Markov dans cette simulation, selon un découpage de $[0, 1]$:

$$[0,1] \quad = \underbrace{[0, \mu_0(s_1)]}_{\text{démarrer en } s_1} \quad \sqcup \quad \underbrace{]\mu_0(s_1), \mu_0(s_1) + \mu_0(s_2)]}_{\text{démarrer en } s_2} \quad \sqcup \quad \begin{matrix}\cdots \\ \cdots\end{matrix} \quad \sqcup \quad \underbrace{]\mu_0(s_1) + \mu_0(s_2) \cdots + \mu_0(s_{n-1}), 1]}_{\text{démarrer en } s_n}$$

Prendre une transition. À partir d'un état s, on tire une valeur aléatoire r uniformément dans $[0, 1]$. La valeur de r obtenue permet de savoir dans quel état, selon un découpage de l'intervalle $[0, 1]$:

$$[0,1] \quad = \underbrace{[0, P_{s,s_1}]}_{\text{aller en } s_1} \quad \sqcup \quad \underbrace{]P_{s,s_1}, P_{s,s_1} + P_{s,s_2}]}_{\text{aller en } s_2} \quad \sqcup \quad \cdots \quad \sqcup \quad \underbrace{]P_{s,s_1} + P_{s,s_2} \cdots + P_{s,s_{n-1}}, 1]}_{\text{aller en } s_n}$$

Typiquement :

$$\varphi(s,u) = \begin{cases} s_1 \text{ si } u \in [0, P_{s,s_1}] \\ s_2 \text{ si } u \in]P_{s,s_1}, P_{s,s_1} + P_{s,s_2}] \\ \vdots \\ s_n \text{ si } u \in]P_{s,s_1} + P_{s,s_2} \cdots + P_{s,s_{n-1}}, 1] \end{cases}$$

Et voici la fonction qui calcule l'état suivant :

fonction etatSuivant(s)
  choisir $u \in [0, 1]$ uniformément
  renvoyer $\varphi(s, u)$

Remarque : le choix de l'ordre des intervalles fait ci-dessus est arbitraire. Aussi, on pourrait pour chaque état utiliser une union d'intervalles. La seule chose qui importe est que leur mesure totale soit la probabilité voulue.

Définition 27 Une **fonction de mise à jour** est une fonction $\varphi : \mathcal{S} \times [0, 1] \to \mathcal{S}$ telle que pour tous états $s, s' \in \mathcal{S}$, la mesure de Lebesgue de $\{u \in [0, 1] \mid \varphi(s, u) = s'\}$ vaut $P_{s,s'}$.

Pour la simulation MCMC, le choix de la **fonction de mise à jour** φ, c'est à dire des valeurs $\varphi(s, u)$ pour $s \in \mathcal{S}$ et $u \in [0, 1]$ n'a pas d'importance. Ce sera différent pour l'algorithme de Propp et Wilson décrit dans la section suivante.

Pour estimer la distribution stationnaire d'une chaîne de Markov, on peut donc procéder de la façon suivante. On choisit n un nombre d'étapes de simulation de la chaîne. On effectue N simulations, chacune de n étapes. Si pour $1 \leq i \leq |\mathcal{S}|$, N_i dénote le nombres de simulations dont le n-ième état est s_i, on définit $\nu_{n,N}(s_i) = \frac{N_i}{N}$. La distribution $\nu_{n,N}$ est une « bonne » approximation de la distribution stationnaire, au sens où $\lim_{n\to\infty} \lim_{N\to\infty} \nu_{n,N} = \mu$.

Limites Il y a des inconvénients à la simulation présentée ci-dessus. Tout d'abord, $\nu_{n,N}$ converge vers la distribution stationnaire μ, mais dans « la plupart des cas » μ n'est jamais atteinte. Par exemple, si $P = \begin{pmatrix} .75 & .25 \\ .25 & .75 \end{pmatrix}$ et $\mu_0 = [1, 0]$, on a $\nu_n = \mu_0 P^n = [\frac{1}{2}(1 + 2^{-n}), \frac{1}{2}(1 - 2^{-n})]$.
Pour contrôler la variation totale entre ν_n et la distribution stationnaire, il faut déterminer quelle valeur de n choisir en fonction de l'erreur admissible ε. En pratique, il est difficile d'obtenir des bornes supérieures sur n qui soient suffisamment petites pour être utilisables.

9 Couplage depuis le passé : algorithme de Propp et Wilson

L'algorithme de Propp et Wilson [PW96], bien que basé sur la simulation, pallie deux inconvénients de la simulation présentée précédemment. Premièrement, il calcule **exactement** la distribution stationnaire, au sens où il produit une sortie dont la distribution est exactement la distribution stationnaire. Deuxièmement, on **sait quand arrêter** la simulation.

9.1 Principe

Le principe est de **simuler k exécutions** de la chaîne de Markov en parallèle, depuis chaque état $s_1, \ldots, s_k$. On arrête la simulation quand les k chaînes ont toutes convergé vers un unique même état.

Plus précisément, on simule les k chaînes depuis le temps -1 jusqu'au temps 0, puis depuis le temps -2 au temps 0, puis du temps -4 au temps 0, etc. On s'arrête dès lors qu'il y a convergence. Toutes ces simulations réutilisent les mêmes résultats des tirages uniformes utilisées dans la fonction de mise à jour.

On se donne une suite

$$N_1, N_2, N_3, ...$$

strictement croissante. Typiquement, on prend $N_i = 2^i$ et c'est ce que l'on fait dans la suite du cours.

entrée : $\mathcal{M}$ une chaîne de Markov décrite avec une fonction de mise à jour
sortie : un état selon la distribution stationnaire de $\mathcal{M}$
fonction Propp-Wilson($\mathcal{M}$)
  Soit $(u_{-i})_{i \in \mathbb{N}^*}$, une suite de nombres dans $[0, 1]$ obtenus par tirages uniformes indépendants
  pour $i := 1, 2, 3, ...$ **faire**
    pour $s \in \mathcal{S}$ **faire**
      $end[s]$: = état après la simulation de $\mathcal{M}$ depuis s du temps $-N_i$ jusqu'au temps 0
        en utilisant $u_{-N_i}, \ldots, u_{-1}$ dans la fonction de mise à jour
    si $\{end[s] \mid s \in \mathcal{S}\}$ est un singleton $\{s'\}$ **alors**
      renvoyer s'

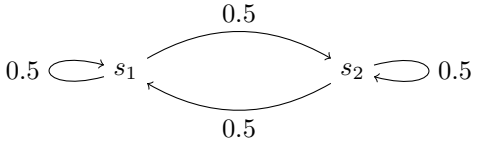
Exemple 1 Donnons un exemple d'exécution de l'algorithme de Propp et Wilson, sur une chaîne de Markov avec $\mathcal{S} = \{s_1, s_2, s_3\}$, pour lequel on suppose les mises à jours suivantes :
— $\varphi(s_1, u_{-1}) = s_1$; $\varphi(s_2, u_{-1}) = s_2$ and $\varphi(s_3, u_{-1}) = s_1$.
— etc.

Remarque : il est crucial, lors de la i-ième simulation de réutiliser les nombres aléatoires déjà utilisés pour la $i-1$-ième, à partir du temps $-N_{j-1}$! 🎓fait en TD

9.2 Terminaison ?

Attention, l'algorithme de Propp-Wilson **ne termine pas forcément**. Le choix de la fonction de mise à jour est important. Voici un exemple où l'algorithme ne termine pas :

Exemple 28 Considérons la chaîne de Markov suivante :



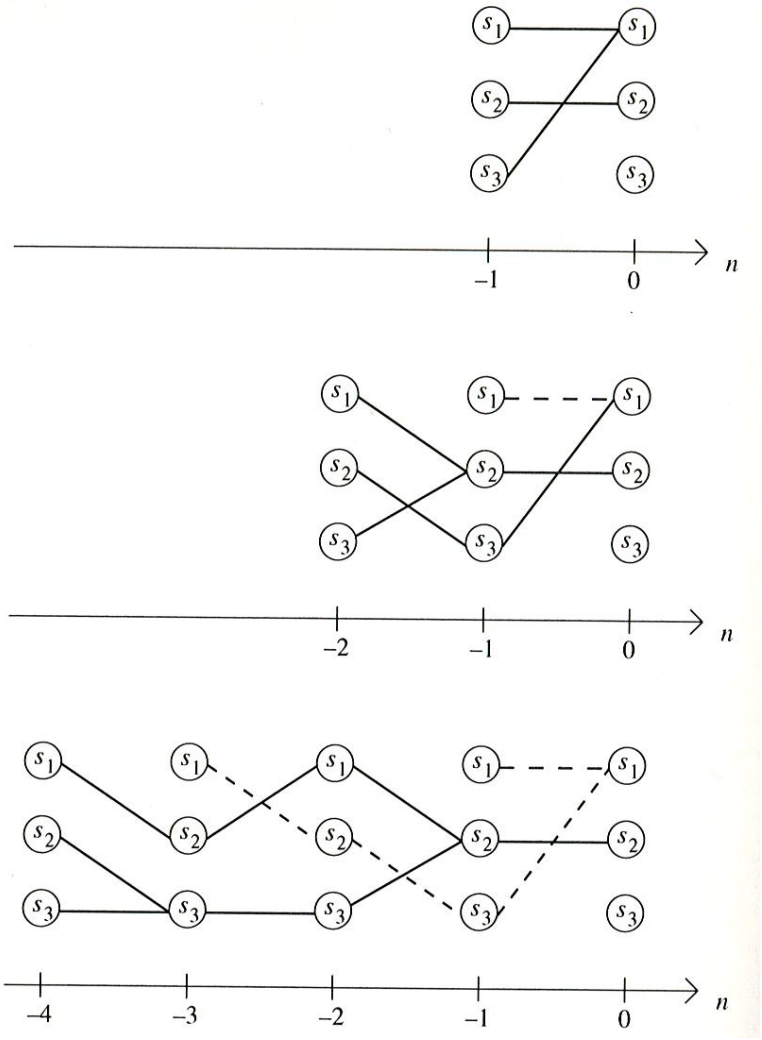


FIGURE 1 – Exemple d'exécution de l'algorithme de Propp et Wilson.

avec la fonction de mise à jour : $\varphi(s_1, r) = \begin{cases} s_1 \text{ si } r < \frac{1}{2} \\ s_2 \text{ sinon} \end{cases}$ et $\varphi(s_2, r) = \begin{cases} s_2 \text{ si } r < \frac{1}{2} \\ s_1 \text{ sinon.} \end{cases}$

Voyez-vous pourquoi l'algorithme ne termine pas ? Voyez-vous comment modifier la fonction de mise à jour pour que l'algorithme termine ?

Théorème 29 Soit $\mathcal{M}$ une description d'une chaîne de Markov :

$$\mathbb{P}(\text{algorithme de Propp-Wilson termine sur } \mathcal{M}) \in \{0,1\}.$$

Démonstration.
Supposons que $\mathbb{P}(\text{algorithme termine sur } \mathcal{M}) > 0$. Montrons que $\mathbb{P}(\text{algorithme termine sur } \mathcal{M}) = 1$.

⇓ Par l'absurde, supposons que pour tout i, on a $\mathbb{P}(\text{l'algorithme termine au rang } i) = 0$. Et donc $\mathbb{P}(\text{l'algorithme termine}) = 0$. Ce qui contredit l'hypothèse.

Ainsi, il existe i tel que $\mathbb{P}(\text{algorithme termine au rang } i \text{ sur } \mathcal{M}) = q > 0$

⇓

Soit $E_{-k}^{-\ell}$ l'événement
$\varphi(\bullet, u_{-k}) \circ \varphi(\bullet, u_{-k+1}) \circ \dots \varphi(\bullet, u_{-\ell-1})$ est une fonction constante,
où les $u_{\dots}$ sont les variables aléatoires qui apparaissent dans l'algorithme.

$\mathbb{P}(E_{-2^i}^{0}) = q > 0$

⇓ Il se passe juste la même chose

$\mathbb{P}(E_{-2^{i+1}}^{-2^i}) = \mathbb{P}(E_{-2^{i+2}}^{-2^{i+1}}) = \dots = q > 0$

algo ne termine pas implique que $\bigcap_{m\geq i} \neg E_{-2^{m+\ell+1}}^{-2^{m+\ell}}$

⇓

$\mathbb{P}(\text{algo ne termine pas}) \leq \mathbb{P}(\bigcap_{m\geq i} \neg E_{-2^{m+\ell+1}}^{-2^{m+\ell}}) = 0$

Pour tout $k \in \mathbb{N}$,

$$\mathbb{P}\Big(\bigcap_{m\geq i} \neg E_{-2^{m+\ell+1}}^{-2^{m+\ell}}\Big) \leq \mathbb{P}\Big(\bigcap_{m=i}^{i+k} \neg E_{-2^{m+\ell+1}}^{-2^{m+\ell}}\Big)$$
$$\leq (1-q)^k \qquad \text{par indépendance}$$

En passant à la limite $k \to +\infty$: $\mathbb{P}(\bigcap_{m\geq i} \neg E_{-2^{m+\ell+1}}^{-2^{m+\ell}}) = 0$.

■

9.3 Correction

Sous l'hypothèse de terminaison avec probabilité 1, l'algorithme de Propp-Wilson est correct.

Théorème 30 Soit $\mathcal{M}$ une chaîne de Markov irréductible apériodique.
Si $\mathbb{P}$(algorithme termine sur $\mathcal{M}$) $= 1$ alors le résultat Y suit la distribution stationnaire.

DÉMONSTRATION. On note μ la distribution stationnaire.
Supposons que $\mathbb{P}$(algorithme termine sur $\mathcal{M}$) $= 1$.

⇓ Par l'absurde, supposons qu'il existe $\varepsilon > 0$, pour tout i tel que $\mathbb{P}$(l'algorithme termine avant l'itération i) $< 1 - \epsilon$. Et donc $\mathbb{P}$(l'algorithme termine) $< 1 - \epsilon$. Ce qui contredit l'hypothèse.

Ainsi, pour tout $\varepsilon > 0$, il existe i tel que $\mathbb{P}$(l'algorithme termine avant l'itération i) $\geq 1 - \epsilon$.

⇓ Considérons une chaîne de Markov $\tilde{\mathcal{M}}$ qui repose sur les même nombres aléatoires que dans l'algorithme, mais qui débute avec un état initial au temps -2^i choisi avec la distribution stationnaire μ. Soit $\tilde{Y}$ l'état de $\tilde{\mathcal{M}}$ au temps 0.

car les nombres aléatoires utilisés sont les même

$\mathbb{P}(\tilde{Y} = Y) \geq 1 - \epsilon$

⇓

$\mathbb{P}(\tilde{Y} \neq Y) \leq \epsilon$

Pour tout état $s \in \mathcal{S}$,

$$\begin{aligned} \mathbb{P}(Y = s) - \mu(s) &\leq \mathbb{P}(Y = s) - \mathbb{P}(\tilde{Y} = s) \\ &\leq \mathbb{P}(Y = s, \tilde{Y} \neq s) \\ &\leq \mathbb{P}(\tilde{Y} \neq Y) \leq \epsilon \end{aligned}$$

$$\begin{aligned} \mu(s_i) - \mathbb{P}(Y = s) &\leq \mathbb{P}(\tilde{Y} = s_i) - \mathbb{P}(Y = s) \\ &\leq \mathbb{P}(Y \neq s, \tilde{Y} = s) \\ &\leq \mathbb{P}(\tilde{Y} \neq Y) \leq \epsilon \end{aligned}$$

$$\Rightarrow |\mathbb{P}(Y = s) - \mu(s)| \leq \epsilon \Leftarrow$$

Et cela pour tout $\epsilon > 0$. Donc pour tout $s \in \mathcal{S}$,

$$\mathbb{P}(Y = s) = \mu(s).$$

■

9.4 Optimisation pour les chaînes monotones

Problème : l'algorithme de Propp-Wilson demande de simuler k chaînes de Markov. Parfois, k est très grand. Pour certaines chaînes de Markov dites **monotones**, on peut utiliser du **sandwiching**.

Exemple 31 On considère une marche aléatoire sur $[\![1, k]\!]$ définie par : $P(1,1) = P(1,2) = \frac{1}{2}$; $P(k,k) = P(k,k-1) = \frac{1}{2}$; et pour $1 < i < k$, $P(i, i-1) = P(i, i+1) = \frac{1}{2}$.

Définissons la fonction de mise à jour naturelle par : « si r est petit, on descend dans la chaîne, sinon, on monte ». Alors, pour l'ordre naturel sur les états, pour toute valeur $r \in [0,1]$, pour tout $i \leq j \in [\![1,k]\!]$, $\varphi(s_i, r) \leq \varphi(s_j, r)$. En conséquence, les k chaînes de Markov partant des états s_2 à s_{k-1} restent toujours « entre » celles partant de s_1 et s_k. Donc, lorsque les chaînes associées à s_1 et s_k convergent vers un même état, c'est le cas également des chaînes intermédiaires $s_2 \cdots s_{k-1}$.

La figure 2 illustre une exécution de l'algorithme de Propp et Wilson sur cette marche aléatoire pour $k = 5$.

Notes bibliographiques

Je remercie Nathalie Bertrand qui avait enseigné ALGO2 il y a quelques années. Le matériel provient principalement de l'excellent livre [H$^+$02] (excellent, car les chapitres sont courts et digestes). L'algorithme de Propp-Wilson provient de [PW96]. Les chaînes de Markov donnent aussi un algorithme d'approximation pour calculer le nombre de modèles d'une formule propositionnelle [WS05]. Pour en savoir plus, sur le comptage de modèles en logique propositionnelle, voir [CMV21].

Références

[CMV21] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Approximate model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1015–1045. IOS Press, 2021.

[H$^+$02] Olle Häggström et al. *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press, 2002.

[PW96] James Gary Propp and David Bruce Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Struct. Algorithms*, 9(1-2) :223–252, 1996.

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2005.

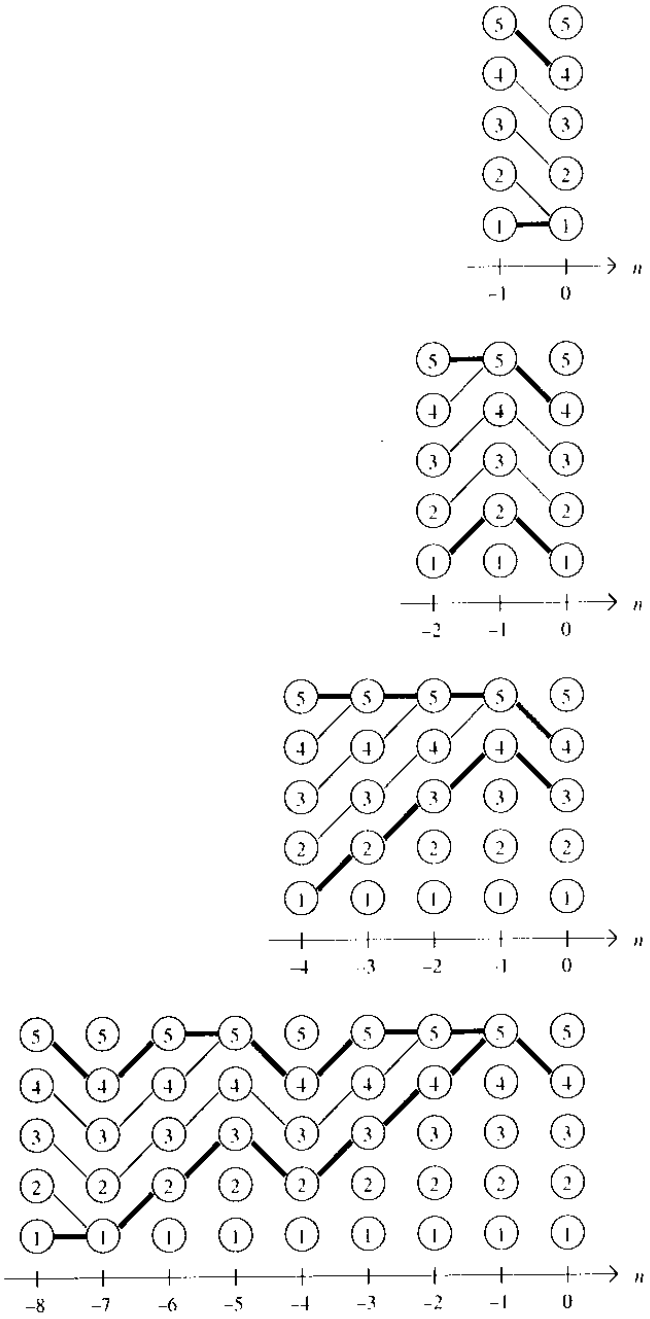


FIGURE 2 – Exemple d'exécution de l'algorithme de Propp et Wilson avec sandwich.

ALGO2 – Largeur arborescente

François Schwarzentruber

May 4, 2024

1 Complexité paramétrée

1.1 Motivation

Pourquoi expliquer la motivation d'étudier la complexité paramétrée, on peut prendre l'exemple de l'évaluation d'une requête dans une base de données.

La base pourrait contenir tous les patients d'un hôpital (des Go de données !) et la requête est 'donner la liste des patients du Dr. Mario'. L'objectif est de calculer le sous-ensemble des données qui répond exactement à la requête. Il existe des classes de requêtes avec un algorithme en $O(n^k)$ où n est la taille de la base et k est la taille de la requête (le nombre de bits pour écrire 'donner la liste des patients du Dr. Mario' dans le langage formel choisi). On voit que la difficulté du problème n'est pas en n (pour k fixé, le problème est polynomial), mais en k.

Mais heureusement, les requêtes sont petites et on s'attend à avoir k petit. L'entier k est un *paramètre*.

Pour d'autres langages logique de requête, on peut avoir une complexité en $O(2^k n)$ et là, le degré du polynôme sur n ne dépend plus de k. C'est ce que l'on appelle fpt pour *fixed-parameter tractable*.

Un autre exemple plus concret :

Exemple 1 pSAT

- entrée : une formule propositionnelle φ
- paramétrisation : nombre de variables dans φ
- sortie : oui si φ est satisfaisable, non sinon.

Voici un algorithme : parcourir toutes les valuations et tester si φ est vraie. Sa complexité est $2^k poly(n)$ où n est la taille de φ et k est le nombre de variables dans φ.

1.2 Fixed-parameter tractable

Donnons ici les définitions formelles.

Définition 1 (paramétrisation) Une *paramétrisation* est une fonction[1] κ qui à toute instance associe un entier :

$$\kappa : Instances \to \mathbb{N}.$$

Définition 2 (problème paramétrée) Un *problème paramétrée* a la forme suivante :
- entrée : une instance x ;
- paramétrisation : κ ;
- sortie : oui si x est une instance positive, non sinon.

Définition 3 (fpt-algorithme) Un algorithme est un fpt-algorithme par rapport à κ s'il existe une fonction calculable f et un polynôme *poly* tel que pour tout entrée x, $A(x)$ s'exécute en temps au plus $f(\kappa(x)) \times poly(|x|)$.

Définition 4 (fixed-parameter tractable) Un problème de paramétrisation κ s'il est décidé par un fpt-algorithme par rapport à κ.

Définition 5 (FPT) FPT est la classe des problèmes paramétrés décidés par un algorithme fpt.

⚠ Attention, FPT contient des problèmes paramétrés, et non des problèmes de décision. Du coup, oui, on peut prendre comme paramétrisation triviale $\kappa(x) = |x|$ et les problèmes paramétrés avec une paramétrisation triviale sont dans FPT. Mais cela n'intéresse personne.

[1] Dans [FG06] p. 4, on la suppose calculable en temps polynomial.

2 Idées générales sur la tree-width

Sur les graphes, il y a paramètre qui connaît un réel succès : la tree-width (largeur arborscente). Pourquoi ? Parce que beaucoup de problèmes sont faciles sur les arbres. Et donc il est naturel de comparer de combien un graphe est proche d'un arbre.

2.1 Sur les arbres = facile

Beaucoup de problèmes sont faciles sur des arbres. En effet, une structure arborescente permet d'appliquer la programmation dynamique. Prenons par exemple le problème de calculer un ensemble indépendant maximum.

Définition 6 **Ensemble indépendant maximum**

entrée : graphe $G = (V, E)$ non orienté
sortie : un ensemble $S \subseteq V$ de sommets deux-à-deux non adjacents, de cardinalité maximale

La version décisionnelle de ce problème est NP-complet en général. Mais si on se restreint à la classe des arbres, le problème est dans P.

Proposition 7 Étant donné un arbre G, on peut calculer un ensemble indépendant maximum en temps polynomial en $|G|$.

Démonstration.

Sous-problèmes On fait de la programmation dynamique en posant les sous-problèmes suivants :

$$I(u) := \text{taille maximale d'un ensemble indépendant dans le sous-arbre enraciné en } u.$$

Le but est de calculer $I(r)$.

Relation de récurrence De deux choses l'une : un ensemble indépendant contient u ou alors ne contient pas u. Ainsi on a :

$$I(u) = \max(1 + \sum_{w \text{ petit-enfant de } u} I(w), \sum_{w \text{ enfant de } u} I(w))$$

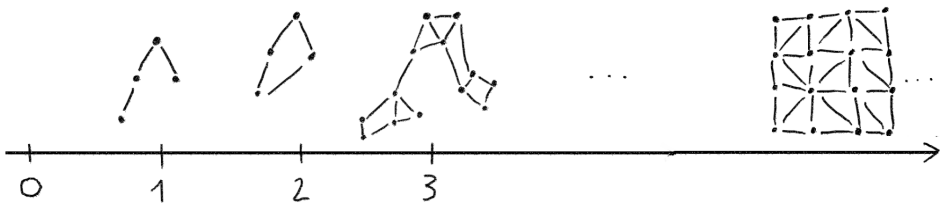
Algorithme

```
fonction calculerI(u)
    si u est une feuille alors
        u.I := 1
    sinon
        pour w enfant de u faire
            calculerI(w)
        u.I = max(1 + Σ_{w petit-enfant de u} w.I, Σ_{w enfant de u} w.I)
```

Je vous laisse le soin d'adapter l'algorithme pour calculer effectivement un ensemble indépendant maximum. ■

2.2 Paramétrisation qui mesure l'arbritude

La dichotomie 'dans P pour les arbres' et 'NP-complet en général'. est hyper grossière... et binaire. Quid d'une paramétrisation qui mesure de combien un graphe ressemble à un arbre ?

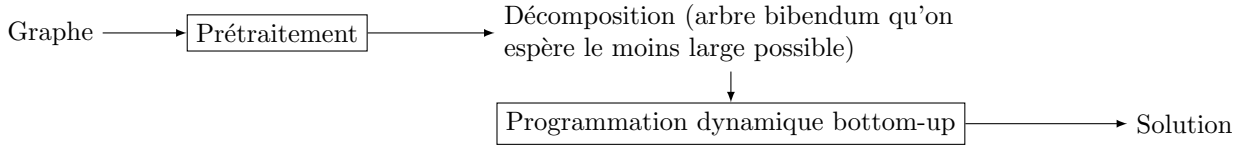


2.3 Principe pour un algorithme efficace

Comme paramètre, on pourrait imaginer compter le nombre de cycles dans le graphe. Un graphe sans cycle est un DAG (proche d'un arbre et sinon, plus il y a de cycle, moins le graphe ressemble à un arbre). Ou alors le *vertex-deletion distance to a forest* (feedback vertex set number). Mais ces solutions ne conviennent pas car elles ne sont pas compatibles avec l'idée de vouloir faire de la programmation dynamique.

L'idée ici est de considérer un 'arbre bibendum' qui entoure le graphe et sur lequel on peut toujours faire de la programmation dynamique, non pas sur le graphe initial, mais sur cet arbre bibendum.

La tree-width est l'épaisseur de l'arbre bibendum.



2.4 En pratique, ça a du sens

En pratique, beaucoup de graphes ressemblent à des arbres, même s'ils n'en sont pas.

Exemple 2
- Graphes des programmes ont une tree-width ≤ 6 généralement
- Graphe de contraintes (CSP)
- Series-parallel graph (ils sont de tree-width 2!)
- outerplanar graphs (tree-width 2 aussi)
- Heuristique pour TSP

3 Définitions

3.1 Définition express

Cette définition de la tree-width permet de comprendre le concept en 5min. Ca vaut le coup de lire, même si ce n'est pas ce que l'on va utiliser dans les algorithmes.

Exemple 8 (1-arbre) Commencer avec une arête. Puis ajoutant itérativement un sommet en le rajoutant à un sommet existant.

Exemple 9 (2-arbre) Commencer avec une 3-clique. Puis ajoutant itérativement un sommet en le rajoutant à une 2-clique existante.

Un *k-arbre* est un graphe obtenu en commençant par une $(k+1)$-clique, puis en ajoutant successivement un sommet en le connectant à une k-clique existante.

Un k-arbre partiel est un sous-graphe d'un k-arbre.

La largeur arborescente d'un graphe G, notée $tw(G)$, est le plus petit entier k tel que G soit un k-arbre partiel.

Mettre un graphe dans un arbre bibendum.

3.2 Définition via décomposition d'un graphe

Cette définition est plus utile algorithmiquement, car c'est ce que l'on va donner en entrée à un algorithme de programmation dynamique.

Soit $G = (V, E)$ un graphe non orienté. Une *décomposition* A de G est un arbre où :

1. les nœuds de A sont (étiquetés par) des *sacs*, i.e. des sous-ensembles de sommets de G ;
2. pour tout sommet x dans G, l'ensemble des nœuds dont les sacs contiennent x forment un sous-arbre connexe de A ;
3. toute arête $\{x, y\}$ de G est incluse dans au moins un sac.

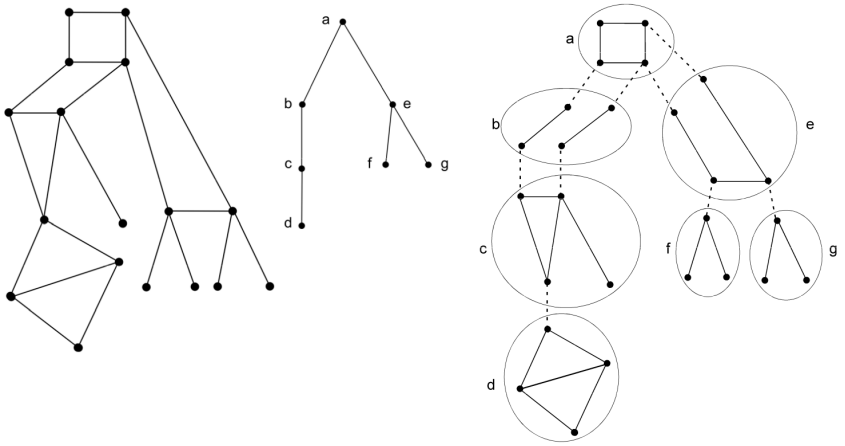
Formellement :

Définition 10 (décomposition d'un graphe) Soit $G = (V, E)$ un graphe non orienté. Une *décomposition* est une paire (T, ∂) où :

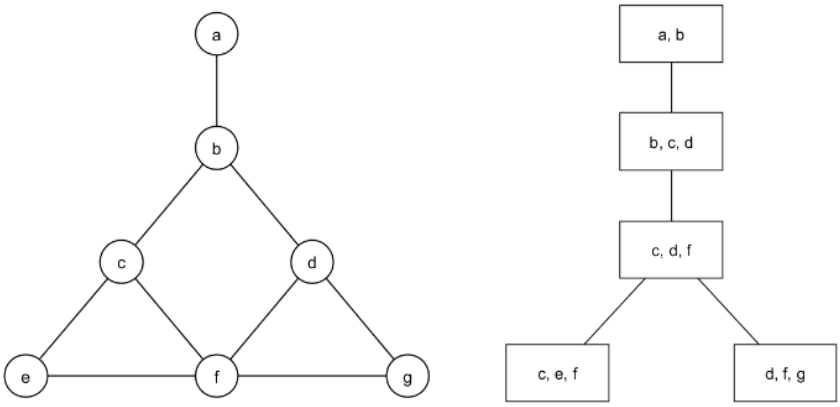
1. $T = (B, R)$ est un arbre, i.e. un graphe non orienté connexe et acyclique ;
2. $\partial : B \to 2^V$ telle que
   (a) Pour tout $x \in V$, en posant $\Gamma_x = \{t \in B \mid x \in \partial_t\}$, Γ_x est non vide et le graphe $(\Gamma_x, \{e \in R \mid e \subseteq \Gamma_x\})$ est connexe ;
   (b) Pour toute arête $e \in E$, il existe $t \in B$ tel que $e \subseteq \partial_t$.

Dans la définition ci-dessus, t désigne un nœud de l'arbre. ∂_t désigne la 'patate' en t, c'est-à-dire le sous-ensemble des sommets du graphe qui étiquette le nœud t. La notation ∂_t fait penser à la notion de frontière. En effet, si on enracine l'arbre, un nœud t est la frontière du sous-arbre enraciné en ce nœud t. Γ_x est l'ensemble des nœuds de l'arbre qui contiennent le sommet x.

Exemple 11



Exemple 3



Notation 12 (sac en un nœud) Étant donné un nœud t d'une décomposition, on note $\partial_t \subseteq V$ le sac du nœud t.

Définition 13 (largeur) La largeur d'une décomposition est le cardinal du plus gros sac moins 1. Autrement dit :

$$largeur(T, \partial) = \max_{t \in B} |\partial t| - 1$$

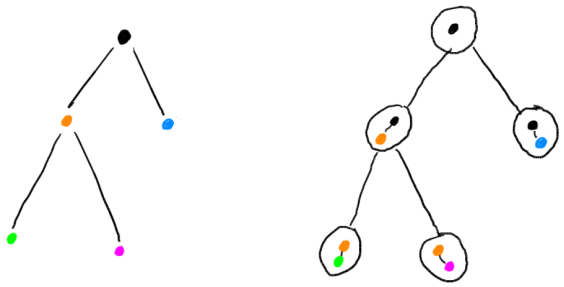
Définition 14 (largeur arborescente − tree-width) La tree-width de G, notée $tw(G)$, est la largeur d'une décomposition la moins large :

$$tw(G) = \min_{(T,\partial) \text{ décomposition de } G} largeur(T, \partial).$$

4 Classes de graphes

Proposition 15 Soit G un graphe connexe. G est un arbre ssi $tw(G) = 1$.

DÉMONSTRATION. Si G est un arbre, alors on peut considérer la décomposition suivante qui est de largeur 1.



Réciproquement, considérons une décomposition de G de largeur 1. TODO: c'est fait en TD (exo)

■

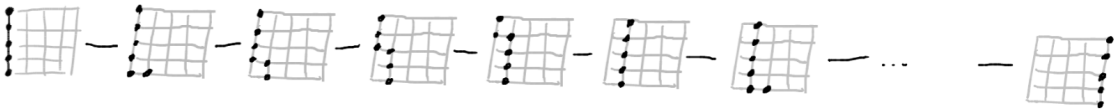
Proposition 16 La largeur arborescente d'un cycle est 2.

DÉMONSTRATION.

■

Proposition 17 La largeur arborescente d'une grille $n \times n$ est n.

DÉMONSTRATION.



■

Proposition 18 La largeur arborescente du graphe complet K_n est $n - 1$.

Proposition 19 Si $tw(G) = k$, alors tout sous-graphe H de G vérifie $tw(H) \leq k$.

Proposition 20 Si G est planaire avec n sommets, alors $tw(G) = O(\sqrt{n})$.

TODO: un jour étudier ça et l'enseigner (est-ce si utile ?)

5 Taille d'une décomposition

Définition 21 Une décomposition d'un graphe G est petite si pour tout nœud x, y, si $x \neq y$ alors $\partial_x \not\subseteq \partial_y$.

Bref, on ne veut pas qu'un sac soit inclus dans un autre.

Proposition 22 Tout graphe G admet une petite décomposition de largeur $tw(G)$. De plus, toute décomposition peut être transformée en temps linéaire en petite décomposition de même largeur.

DÉMONSTRATION. Contracter les arcs (x, y) de la arbre-décomposition pour lesquelles $\partial_x \subseteq \partial_y$ ou $\partial_y \subseteq \partial_x$. Garder le sac le plus grand de ∂_x et ∂_y. Un des étudiants m'a parlé *annexion*, j'aime beaucoup la métaphore. ■

Proposition 23 Une petite décomposition d'un graphe $G = (V, E)$ possède au plus $|V|$ nœuds.

DÉMONSTRATION. Indice : Par induction sur $|V|$ (sachant qu'une feuille contient au moins un sommet qui n'est dans aucun autre sac).

On pose $\mathcal{P}(n)$: 'pour tout graphe $G = (V, E)$ avec $|V| = n$, toute décomposition de G possède au plus n nœuds.'

Montrons que $\mathcal{P}(n)$ pour tout $n \in \mathbb{N}$ par récurrence sur n.

Cas de base. Si le graphe contient un sommet v, alors le seul sac possible est $\{v\}$. Il n'y a qu'une seule petite décomposition et elle est de taille 1. D'où $\mathcal{P}(1)$.

Cas récursif. Supposons $\mathcal{P}(i)$ pour tout $i \leq n-1$. Montrons $\mathcal{P}(n)$. Considérons une petite décomposition A de G. Considérons la comme un arbre enraciné. Considérons une feuille x de A, et y son père. On sait que $\partial_x \not\subseteq \partial_y$. Ainsi l'ensemble $\Gamma := \partial_x \setminus \partial_y$ est non vide.

Fait 24 Les sommets dans Γ n'apparaîssent que dans le sac ∂_x.

DÉMONSTRATION. Soit $v \in \Gamma$. Comme l'ensemble des nœuds dont les sacs contiennent v forment un sous-arbre connexe de A. Comme $v \notin \partial_y$, ∂_x est le seul sac qui contient v. ■

Soit A' la petite décomposition obtenue à partir de A mais en supprimant la feuille x. A' est une petite décomposition de $G \setminus \Gamma$.

Par $\mathcal{P}(n - |\Gamma|)$, on sait que A' contient au plus $n - |\Gamma| \leq n - 1$ nœuds. Ainsi A possède au plus n nœuds.

■

Proposition 25 Dans tout graphe G, il existe un sommet de degré $\leq tw(G)$.

DÉMONSTRATION. 🎓Soit $G = (V, E)$ un graphe et considérons une décomposition petite de largeur $tw(G)$. Deux cas.

- Si G a moins de $tw(G) + 1$ sommets, alors tous les sommets de degré $\leq tw(G)$ et donc c'est gagné.
- Sinon, G a plus de $tw(G) + 2$ sommets. Mais alors la décomposition possède au moins deux nœuds. Soit f une feuille et p son parent. Il y a un sommet $s \in \partial_f \setminus \partial_p$. Alors ∂_f est le sac qui contient s. Ainsi, toutes les arêtes partant de s sont présentes dans ce sac. Or $|\partial_f \setminus \{s\}| \leq tw(G)$. Et donc s est de degré au plus $tw(G)$.

■

6 Algorithmes pour décomposition

Définition 26 (problème de décision du calcul de la largeur arborescente)

tree-width
entrée : un graphe G, un entier k
sortie : oui, si la tree-width de G est k, non sinon.

Théorème 27 (admis) Le calcul de la tree-width est NP-complet.

Théorème 28 (de Bodlaender, admis) Il existe un algorithme qui prend en entrée un graphe G qui renvoie une petite décomposition optimale de largeur $k = tw(G)$ en temps $O(|G| \times 2^{poly(k)})$.

Malheureusement, l'algorithme donné par le théorème de Bodlanender n'est pas efficace en pratique. La partie $2^{poly(k)}$ est $2^{O(k^3)}$ (cf. [FG06], p. 267). Il est aussi difficile à démontrer. En annexe, nous démontrons le résultat suivant où la complexité par rapport au paramètre est plus raisonnable : $2^{O(k)}$. Par contre, il s'agit d'un algorithme d'approximation.

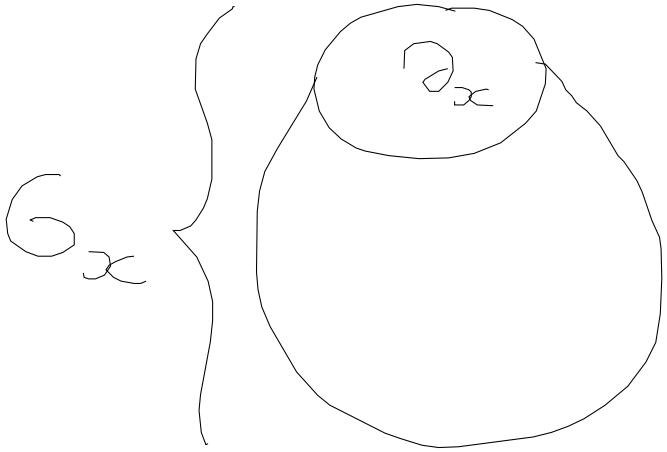
Théorème 29 (démontré en annexe) Il existe un algorithme prenant en entrée un graphe G qui renvoie une petite décomposition de largeur $\leq 4k + 1$ en temps $O(|G| \times 2^{O(k)})$, où $k = tw(G)$.

7 Programmation dynamique

7.1 Décomposition enracinée

Notation 30 (ensemble de sommets d'un sous-arbre) Étant donné un nœud x d'une décomposition, l'ensemble de sommets du sous-arbre enraciné en x est $V_x = \cup_{y \text{ descendant de } x} \partial_y$.

Notation 31 On note $G_x = (V_x, E \cap (V_x \times V_x))$.

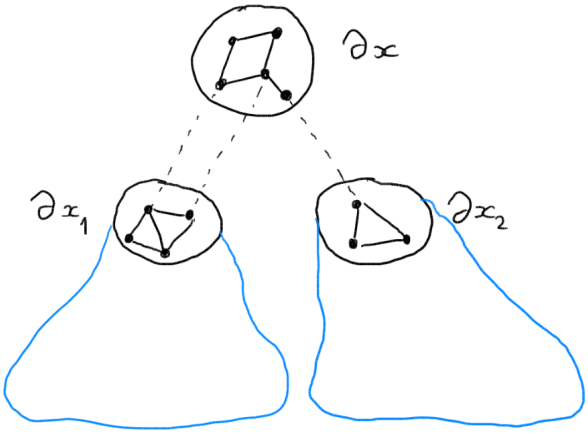


Proposition 32 Le problème Ensemble indépendant avec le paramètre tree-width est dans FPT.

Démonstration. D'abord on calcule une petite décomposition arborescente de G, optimale, ou quasiment optimale. Disons de largeur $k \leq 4tw(G) + 1$. On la considère comme un arbre enraciné. Ensuite, on applique la programmation dynamique bottom-up.

Définition des sous-problèmes. Étant donné un nœud x dans l'arbre, et un sous-ensemble $S \subseteq \partial_x$ qui est indépendant dans G, on note $MIS(x, S)$ la taille d'un plus grand ensemble indépendant I dans G_x avec $I \cap \partial_x = S$.

Relation de récurrence.



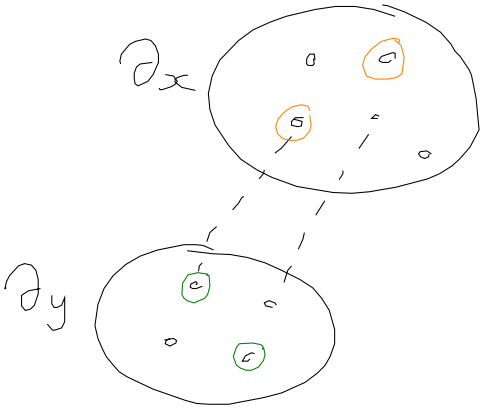
On a

$$MIS(x, S) = |S| + \sum_{y \text{ enfant de } x} \left(\max_{S' \subseteq \partial_y | S' \text{ indépendant sur } \partial_y \text{ et } S' \cap \partial_x = S \cap \partial_y} MIS(y, S') - |S \cap \partial_y| \right).$$

En effet, on comptabilise les sommets choisis dans ∂_x, qui doivent être ceux de S, d'où le $|S|$.

Pour chaque enfant y, on maximise la taille de l'ensemble indépendant de G_y. On ne sait pas pour quel $S' \subseteq \partial_y$ c'est maximal. Donc on maximise sur eux.

Mais il faut que les sommets pris dans S' soient compatible avec S.



La contrainte est $S' \cap \partial_x = S \cap \partial_y$. Il faut aussi retrancher $|S \cap \partial_y|$ car les sommets de S ont déjà tous été comptabilisé dans $|S|$.

Algorithme. On calcule récursivement $MIS(x, \bullet)$.

```
fonction calculerMIS(x)
    pour y enfant de x faire
        calculerMIS(y)
    pour S ⊆ ∂_x avec S indépendant faire
        x.MIS[S] :=...équation récursive...
    renvoyer S qui maximise x.MIS[S]
```

Complexité temporelle. La complexité de calculerMIS(x) est :

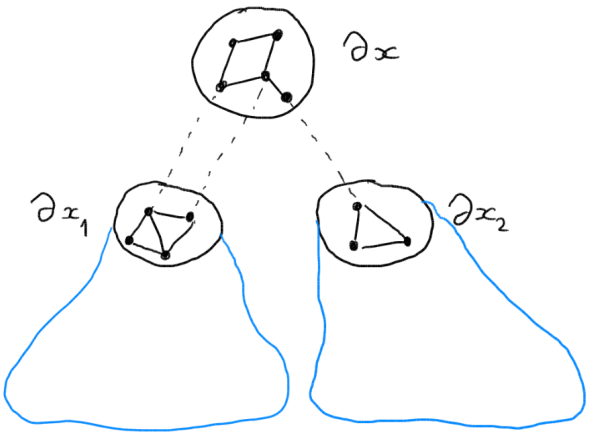
$$T(x) = 4^k k^6 |\text{enfants de } x| + \sum_{y \text{ enfant de } x} T(y).$$

Le 4^k vient de $2^k 2^k$: le premier 2^k de la boucle pour tout S, et le deuxième 2^k de la boucle pour S' (le max). Le k^2 vient du fait qu'il faut vérifier que les S considérés sont indépendants sur ∂_x. Pareil pour S'. Il faut aussi vérifier que $S' \cap \partial_x = S \cap \partial_y$. C'est aussi du k^2. L'accès à $x.MIS[S]$ est en $O(1)$ car $x.MIS$ est un tableau à 2^k cases (on considère S comme un nombre entre 0 et $2^k - 1$). Ainsi on a $T(racine) = 4^k k^6 |V|$. ■

Proposition 33 Le problème de 3-coloration admet un algorithme de complexité $O(|G| \times f(tw(G)))$.

Démonstration. D'abord on calcule une petite décomposition arborescente de G. Ensuite, on applique la programmation dynamique bottom-up. Étant donné un nœud x dans l'arbre, on calcule $extCol(x)$ l'ensemble des colorations licites des sommets dans ∂_x qui peuvent être étendue en une coloration du sous-graphe G_x.

1. La relation de récurrence est, en notant $x_1, \ldots, x_k$ les fils de x :
   $extCol(x) = \{f \text{ coloration de } \partial_x \mid f \text{ coincide avec une coloration de } extCol(x_i) \text{ pour tout } i\}$



2. Le graphe G est coloriable ssi $extCol(racine)$ est non vide.

Exercice 34 Evaluer la complexité de l'algorithme de 3-coloration.

Exercice 35 Écrire le pseudo-code de l'algorithme qui prend en entrée une décomposition d'un graphe G et décide si le graphe est 3-coloriable.

Exercice 36 Implémenter l'algorithme de 3-coloration ci-dessus.

■

8 Théorème de Courcelle

Caractérisation logique d'une classe où l'approche programmation dynamique sur une décomposition fonctionne

Le théorème de Courcelle (ou plutôt les théorèmes de Courcelle) généralise l'approche programmation dynamique vu précédemment, à *n'importe quel problème* que l'on peut décrire en logique monadique du second ordre (MSO pour monadic second-order logic).

8.1 Logique monadique du second ordre

Le théorème fonctionne avec cette variante de MSO.

Définition 37 (syntaxe de MSO) La syntaxe de la *logique monadique du second ordre* sur les graphes est définie par la grammaire suivante :

$$\varphi \ ::= \ u{=}v \ \mid \ u{\in}X \ \mid \ uRv \ \mid \ \neg\varphi \ \mid \ \varphi \vee \varphi \ \mid \ \forall x\varphi \ \mid \ \forall X\varphi$$

Exemple 4 (ensemble indépendant en MSOL) $ind(X) := \forall u, \forall v, (u \in X \wedge v \in X) \rightarrow \neg uRv$

Exemple 5 (3-coloration en MSOL) $\exists A, \exists B, \exists C, \forall u, exact(u, A, B, C) \wedge ind(A) \wedge ind(B) \wedge ind(C)$.

8.2 Énoncé des théorèmes de Courcelle

Définition 38 **Problème de Courcelle**

entrée : une formule φ de MSOL, un graphe G
sortie : oui, si $G \models \varphi$, non sinon.

Définition 39 **Problème d'optimisation de Courcelle**

entrée : une formule $\varphi(X)$ de MSOL, un graphe G
sortie : A de cardinal max/min tel que $G, [X := A] \models \varphi$, non sinon.

Théorème 40 (de Courcelle, admis) Le problème de Courcelle admet un algorithme en temps $O(|G| \times f(|\varphi|, tw(G)))$. Le problème d'optimisation de Courcelle admet un algorithme en temps $O(|G| \times f(|\varphi|, tw(G)))$.

DÉMONSTRATION. (idée) Il y a un nombre fini de sous-graphes de taille k. On fait programmation dynamique bottom-up sur l'arbre. ■

8.3 Généralisation

Les théorèmes de Courcelle fonctionnent aussi avec cette variante de MSO, un peu plus riche. On en a besoin pour montrer que Cycle hamiltonien est FPT par rapport au paramètre tree-width.

Définition 41 (syntaxe de MSO) La syntaxe de la *logique monadique du second ordre* sur les graphes est définie par la grammaire suivante :

$$\varphi \ ::= \ u{=}v \ \mid \ u{\in}X \ \mid \ uRv \ \mid \ inc(u,e) \ \mid \ \neg\varphi \ \mid \ \varphi{\vee}\varphi \ \mid \ \forall x \in V, \varphi \ \mid \ \forall e \in E, \varphi \ \mid \ \forall X \subseteq V, \varphi \ \mid \ \forall X \subseteq E, \varphi$$

Exemple 6 (cycle hamiltonien en MSOL) $\exists C \subseteq E, spanning(C) \wedge \forall u, deg2(u, C)$ avec
$deg2(u, C) := \exists e, f \in E, e \in C \wedge f \in C, e \neq f \wedge inc(u, e) \wedge inc(u, f) \wedge \forall g \in E((g \in C \wedge inc(u, g)) \rightarrow (g = e \vee g = f))$
et $spanning(C) = \forall X \subseteq V, ((X \neq \emptyset) \wedge (X \neq V)) \rightarrow (\exists u, v \in V, \exists e \in E, (u \in X \wedge v \notin X \wedge e \in C \wedge inc(u, e) \wedge inc(v, e)))$.

Notes bibliographiques

Ce cours est adapté du chapitre 11 de [FG06]. Le théorème de Courcelle reste vraie pour MSOL sur les *hypergraphes*, où les variables du premier (deuxième) ordre peuvent dénoter aussi des (ensembles d') arêtes. L'application de programmation dynamique pour ensemble indépendant dans les arbres est présenté dans [DPV08]. Il existe quelques expérimentations récentes sur la tree-width et théorie des bases de données [MSJ19].

References

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

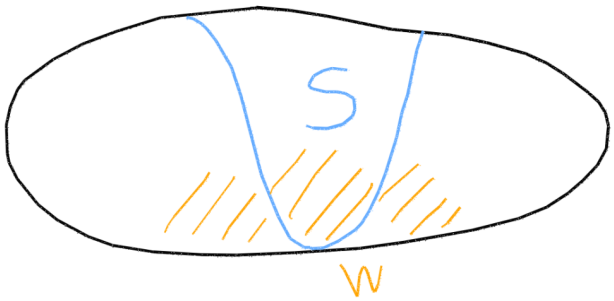
[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[MSJ19] Silviu Maniu, Pierre Senellart, and Suraj Jog. An experimental study of the treewidth of real-world graph data. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

A Algorithme d'approximation pour décomposition

A.1 Séparation balancée

Définition 42 (W-séparateur balancé) Soit $G = (V, E)$ un graphe et $W \subseteq V$. On dit que $S \subseteq V$ est un W-séparateur balancé si toute composante connexe de $G \setminus S$ contienne moins de la moitié des éléments de W. Formellement, si $C_1, \ldots, C_\ell$ sont les composantes de $G \setminus S$, alors pour tout $i = 1..\ell$, $|C_i \cap W| \leq \frac{|W|}{2}$.



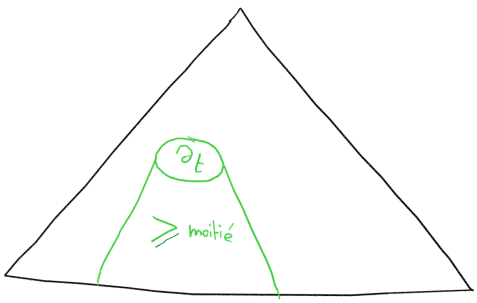
Proposition 43 Soit $G = (V, E)$ un graphe de largeur arborescente au plus k et $W \subseteq V$. Alors il existe un W-séparateur de G de cardinal au plus $k + 1$.

Démonstration. Considérons une décomposition T. On considère un nœud t tels que l'ensemble des sommets sous t contiennent plus de la moitié des éléments de W.

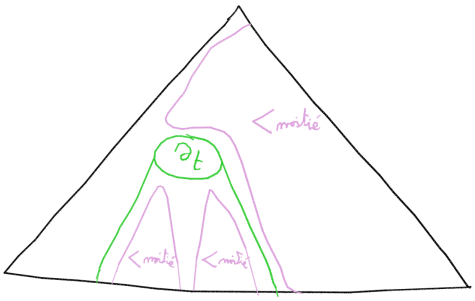
On note V_t l'ensemble des sommets qui sont dans des sacs dans le sous-arbre enraciné en t. Autrement dit on a :

$$|V_t \cap W| \geq \frac{|W|}{2}.$$

De plus, on suppose T_t de hauteur minimale (i.e. au plus profond dans l'arbre T).



On pose $S = \partial_t$ (i.e. le séparateur est le sac du nœud t).
Maintenant on pose :
$C_0 = (V \setminus V_t) \setminus S$ (les sommets qui ne sont pas dans des sacs sous t, et pas dans S) ;
$C_i = V_{u_i} \setminus S$ où u_i est le i-ème fils de t (les sommets qui sont sous u_i mais pas dans S).



Fait 44 C_0 contient strictement moins de la moitié de W.

Démonstration. Comme $|V_t \cap W| \geq \frac{|W|}{2}$, on a $|V \setminus V_t \cap W| < \frac{|W|}{2}$. Comme $C_0 \subseteq V \setminus V_t$, on a $|C_0 \cap W| < \frac{|W|}{2}$. ■

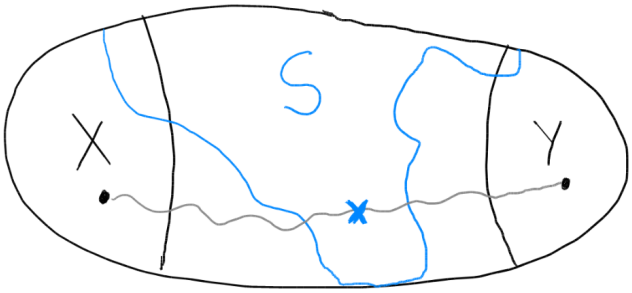
Fait 45 C_i contient strictement moins de la moitié de W.

Démonstration. Car T_t est de hauteur minimale. ■
Chaque composante de $G \setminus S$ est incluse dans un des C_i (exercice). Ainsi chaque composante de $G \setminus S$ contient moins de la moitié des éléments de W. ■

A.2 Séparation

On se donne deux ensemble X et Y, pas forcément disjoints. S est un séparateur de X et Y si en enlevant S on ne peut plus aller de X à Y.

Définition 46 (séparateur) Soit $G = (V, E)$ un graphe. Soit $S, X, Y \subseteq V$. On dit que S sépare X de Y si tout chemin d'un sommet de X à un sommet de Y contient un sommet de S.



Exemple 47 TODO: à faire

Remarque 48 Tout sur-ensemble de X sépare X de Y. Tout sur-ensemble de Y sépare X de Y.

Proposition 49 Le problème suivant admet un algorithme en temps $O(poly(k, G))$:

Séparateur petit
entrée : un graphe $G = (V, E)$, deux ensembles $X, Y \subseteq V$, un entier k
sortie : un ensemble $S \subseteq V$ de cardinal $\leq k$ qui sépare X de Y, s'il en existe un, 'non' sinon.

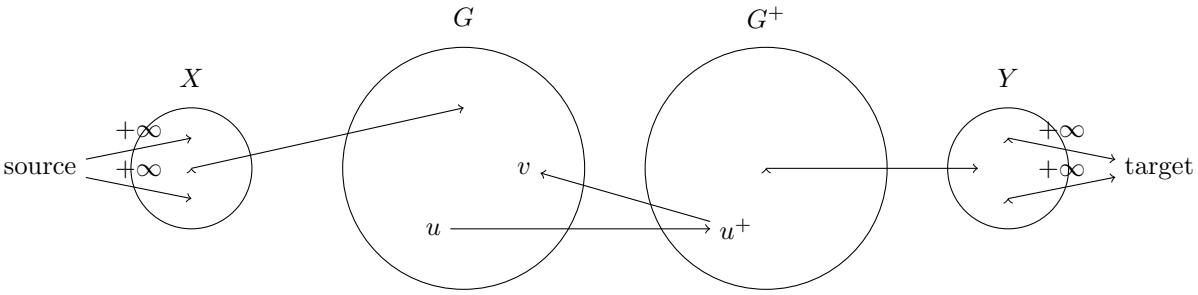
Démonstration. Par réduction aux flots !
Cela provient du théorème Menger :

$$\max_{P \text{ ensemble de chemins disjoints de } X \text{ à } Y} |P| = \min_{S \text{ qui sépare } X \text{ et } Y} |S|.$$

Par chemins disjoints, on entend des chemins tels qu'il n'y ait pas de sommets intermédiaires communs. Pour P un ensemble de chemins disjoints deux à deux, $|P|$ est le nombre de chemins.

$\boxed{\leq}$ Soit S minimal qui sépare X de Y. Soit P. Montrons que $|P| \leq |S|$. Chaque chemin de P vient intersecter un sommet différent de S. Donc $|P| \leq |S|$.

$\boxed{\geq}$ Montrons $|S| \leq |P|$ si P et S sont optimums. Attention, ce n'est pas toujours le cas : on peut mal choisir les chemins (typiquement un exemple avec un chemin qui fait un zizag dans S énorme...). Pour montrer cela, on va montrer qu'il s'agit du théorème max flot/min cut dans un certain réseau de flots. Le réseau de flots est constitué d'une source, d'une copie de X, de deux copies de G, G et G^+, et d'une copie de Y, et d'un puits. On connecte la source avec capacité infinie à tous les sommets de X, et tous les sommets de Y au puits avec capacité infinie. Puis on a des arcs de capacité 1. Les sommets de X sont connectés à leur successeur dans G. Les sommets prédécesseurs de Y dans G^+ sont connectés à leurs successeurs dans Y. Un sommet u est relié juste à u^+. On relie u^+ à v si (u, v) est une arête.



Un flot f dans ce réseau correspond à un ensemble de chemins disjoints. A quoi correspond une coupe minimale ?

Si on a une coupe minimale $C = (S, T)$, elle est de capacité finie, et on peut montrer que les seuls arcs qui vont de la s-partie à la t-partie sont des arcs du type (u, u^+). Par l'absurde, supposons que l'on ait $u^+ \to v$ qui soit séparé, i.e. $u^+ \in S$ et $v \in T$. Dans le graphe résiduel du flot correspondant, ça veut dire que u^+ est accessible mais pas v. Cela, signifie qu'il y a de l'eau de u^+ à v. Mais donc il y a de l'eau qui arrive en u^+, et cette eau ne peut venir que de u. Mais alors, on peut annuler cette eau (il y a un arc u^+u dans le graphe résiduel) et donc u^+ n'est pas accessible dans le graphe résiduel.

On pose S l'ensemble des sommets u tel que (u, u^+) soit un arc séparé par la coupe minimale. Supprimer ces sommets sépare X et Y.

Si P et S sont optimum, On a donc $|P| \geq |f| = |C| \geq |S|$.

Voici donc l'algorithme :

1. construire le réseau de flots
2. appeler l'algorithme Ford-Fulkerson
3. soit S, l'ensemble des sommets u avec (u, u^+) qui est séparé par la coupe minimal.
4. si $|S| \leq k$, alors retourner S sinon retourner "non".

■

Corollaire 50 Le problème suivant admet un algorithme en temps $O(poly(k, G))$:

Séparateur petit'

entrée : un graphe $G = (V, E)$, trois ensembles $X, Y, Z \subseteq V$, un entier k
sortie : un ensemble $S \subseteq V \setminus (X \cup Y)$ de cardinal $\leq k$ et $Z \subseteq S$, qui sépare X de Y, s'il en existe un, 'non' sinon.

Démonstration. S'il y a un arc de X à Y, nous avons un chemin direct de X à Y et il est impossible d'avoir un sommet d'un $S \setminus (X \cup Y)$ dedans. On répond donc non.

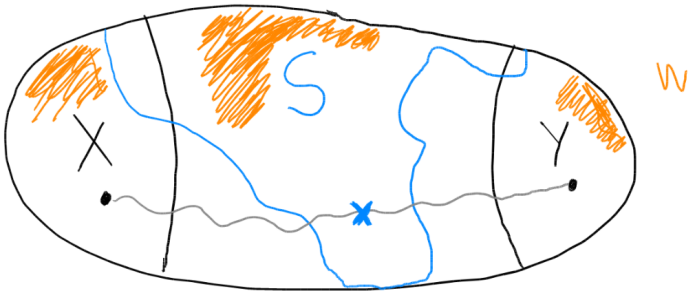
Sinon, soit $S(X)$ les successeurs de X dans $V \setminus X$ (pareil pour $S(Y)$). On considère un nouveau graphe G' qui est le graphe G induit à $V' := V \setminus (X \cup Y)$. On utilise l'algorithme précédent sur l'instance $G', S(X) \cup Z, S(Y) \cup Z, k$. ■

A.3 Séparation faiblement balancé

Définition 51 (W-séparateur faiblement balancé) Soit $G = (V, E)$ un graphe et $W \subseteq V$. On dit que $S \subseteq V$ est un W-séparateur faiblement balancé s'il existe $X, Y \subseteq W$ non vides et disjoints avec :

1. $W = X \sqcup (S \cap W) \sqcup Y$;

2. S sépare X de Y
3. $0 < |X| \leq 2|W|/3$, $0 < |Y| \leq 2|W|/3$.



Exemple 52 TODO: un jour

Proposition 53 Il y a un algorithme en temps $O(3^{3k}poly(k, G))$ pour le problème suivant :

Séparateur faiblement balancé petit

entrée : un graphe $G = (V, E)$, un ensemble $W \subseteq V$ avec $|W| = 3k + 1$, un entier k
sortie : un ensemble $S \subseteq V$ W-séparateur faiblement balancé de cardinal $\leq k + 1$, s'il existe un, 'non' sinon.

Démonstration. On utilise l'algo qui calcule un séparateur comme sous-routine.

pour $X, Y \subseteq W$ non vides et disjoints, de cardinal au plus $2k$ **faire**
  $S :=$ algo pour **séparateur petit'** avec X, Y et $Z := W \setminus (X \cup Y)$ et $k + 1$.
  si S n'est pas "non" **alors renvoyer** S
renvoyer "non"

L'algorithme s'il y a $S \subseteq V \setminus (X \cup Y)$ avec $|S| \leq k + 1$ qui sépare X de Y et avec

$$S \cap W = W \setminus (X \cup Y),$$

en demandant à ce que $W \setminus (X \cup Y) \subseteq S$.

Complexité. Voici comment représenter (X, Y) comme un mot dans $\{0, 1, 2\}^W$ en utilisant le fait que X et Y soient disjoints : pour chaque élément de W, on indique si l'élément n'est ni dans X ni dans Y (0), dans X (1) ou dans Y (2). Ainsi, comme $|W| = 3k + 1$, 3^{3k+1} est une borne supérieure du nombre de (X, Y). D'où la complexité. ■

Proposition 54 Soit $G = (V, E)$ avec $tw(G) = k$. Soit $W \subseteq V$ avec $|W| = 3k + 1$. Alors il y a un W-séparateur faiblement balancé de cardinal au plus $k + 1$.

Démonstration. D'après la proposition 43, il existe un W-séparateur S balancé de cardinal $\leq k + 1$. Soit $C_1, \ldots, C_m$ les composantes connexes de $G \setminus S$, qui ont une intersection non vide avec W. Posons $W_i = C_i \cap W$. On a $W \setminus S = \bigsqcup_{i=1}^{m} W_i$. Par définition d'un W-séparateur balancé, on a $|W_i| \leq |W|/2$.

Fait 55 $m \geq 2$.

Démonstration. Par l'absurde, supposons que $m = 1$. On a : $W \setminus S = \bigsqcup_{i=1}^{m} W_i = \bigsqcup_{i=1}^{1} W_i = W_1$. $|W \setminus S| = |W_1| \leq |W|/2$.

D'autre part, on montre que $|W \setminus S| > |W|/2$. En effet, on a $|W|/2 = (3k + 1)/2 = 1.5k = 0.5$. Et donc :

$$\begin{aligned} |W \setminus S| &\geq |W| - |S| \\ &= 3k + 1 - (k - 1) \\ &= 2k + 2 \\ &> |W|/2 \end{aligned}$$

Contradiction. ■

Sans perte de généralité, supposons que $|W_1| \geq |W_2| \geq \ldots |W_m|$. On rappelle que $W \setminus S = \bigsqcup_{i=1}^{m} W_i$. Soit i minimum tel que $\sum_{j=1}^{i} W_i \geq \frac{|W \setminus S|}{3}$ (on regarde l'union disjointe $\bigsqcup_{i=1}^{m} W_i$ et on s'arrête dès que le cardinal dépasse $\frac{|W \setminus S|}{3}$).

On pose $X = \bigsqcup_{j=1}^{i} W_j$ et $Y = \bigsqcup_{j=i+1}^{m} W_j$.

Fait 56 $|X| > 0$.

Fait 57 $|Y| > 0$.

Fait 58 $|X| \leq 2/3|W|$.

Fait 59 $|Y| \leq 2/3|W|$.

■

A.4 Algorithme calculant une décomposition

entrée : k un entier, $G = (V, E)$ un graphe, $W \subseteq V$ avec $|W| \leq 3k + 1$
sortie : une décomposition arborescente de G de largeur $\leq 4k + 1$ où le sac à la racine inclut W, ou échec s'il n'y a pas de telle décomposition
fonction décomposition(k, G, $[W = \emptyset]$)
  si $|V| \leq 4k + 2$ **alors**
    renvoyer l'arbre-racine $\boxed{V}$
  sinon
    $W :=$ un sous-ensemble de V qui est sur-ensemble de W avec $|W| = 3k + 1$
    $S :=$ un W-séparateur balancée faible avec $|S| \leq k + 1$
                                (s'il n'y a pas de tel S, **échec**)
    $C_1, \ldots, C_m :=$ les composantes connexes de $G \setminus S$
    pour $i := 1$ à m **faire**
      $A_i :=$ décomposition(k, $G_{|C_i \cup S}$, $(W \cap C_i) \cup S$)
    renvoyer

Proposition 60 L'algorithme termine.

Démonstration. On montre que la taille des graphes diminue strictement au cours des appels, c'est-à-dire il faut montrer que $|C_i \cup S| < |V|$.

Comme S est un W-séparateur faiblement balancé, il y a une partition $W = X \sqcup (S \cap W) \sqcup Y$. Ainsi, C_i touche X ou Y mais pas les deux (s'il touchait les deux, on aurait un chemin de X à Y dans $G \setminus S$ et donc S ne serait pas un séparateur). Donc il y a forcément un élément de W qui n'est pas dans $C_i \cup S$ (si C_i touche X, W contient un élément de Y alors C_i non). Ainsi, $C_i \cup S$ ne peut pas être V tout entier. ■

Proposition 61 Sa spécification est bien respectée : on a toujours $|W| \leq 3k + 1$.

Démonstration. Il faut montrer que $|(W \cap C_i) \cup S| \leq 3k + 1$.

Lemme 62 $|C_i \cap W| \leq 2/3|W|$.

Démonstration. Comme S est un W-séparateur faiblement balancé, il y a une partition $W = X \sqcup (S \cap W) \sqcup Y$. On écrit alors :

$$C_i \cap W = (X \cap C_i) \sqcup (S \cap C_i \cap W) \sqcup (Y \cap C_i)$$

Comme C_i est une composante de $G \setminus S$, on a $(S \cap C_i \cap W) = \emptyset$.
Or comme argumenté plus haut, C_i touche X ou Y mais pas les deux, Donc :

$$C_i \cap W = (X \cap C_i) \text{ ou } (Y \cap C_i).$$

Of $|X|, |Y| \leq \frac{2}{3}|W|$. Donc, $|C_i \cap W| \leq 2/3|W|$.
■

$$\begin{aligned} |(W \cap C_i) \cup S| &\leq |W \cap C_i| + |S| \\ &\leq \lfloor 2/3|W| \rfloor + k + 1 \\ &\leq \lfloor 2/3(3k + 1) \rfloor + k + 1 \\ &\leq \lfloor 2k + 2/3 \rfloor + k + 1 \\ &\leq 2k + k + 1 \\ &\leq 3k + 1 \end{aligned}$$

■

Proposition 63 Dans le cas où le calcul réussit, décomposition(k, G, W) renvoie une décomposition arborescente de largeur $\leq 4k + 1$.

DÉMONSTRATION. Soit $i \neq j$. Les composantes C_i et C_j sont disjointes. Donc un sommet qui est à la fois dans $C_i \cup S$ et dans $C_j \cup S$ est en fait dans S, et donc dans le sac $W \cup S$ de la racine. Pour les arêtes : tout va bien.
Et pour la largeur :

$$\begin{aligned} |W \cup S| &\leq |W| + |S| \\ &\leq 3k + 1 + k + 1 \\ &\leq 4k + 2 \end{aligned}$$

D'où une largeur d'au plus $4k + 1$. ■

Proposition 64 Si $tw(G) \leq k$, alors décomposition(k, G, W) réussit.

DÉMONSTRATION. Par induction sur le nombre de sommets dans G. Par la proposition 54, si $tw(G) \leq k$, on montre qu'il existe un W-séparateur balancé faible (et donc pas d'échec !).
■