INF581: Du calcul probabiliste au calcul quantique **Hiver 2011**

Cours 2 — 17 janvier

Enseignant : Frédéric Magniez *Rédacteur : DO Quoc Khanh*

2.1 Test de commutativité sur les structures de groupe

Exemple 1 : Soient A_1, A_2, ..., A_k k matrices $n \times n$ inversibles. On cherche un algorithme qui décide si $\forall i, j \in \{1, 2, ..., k\}$ $A_i A_j = A_j A_i$. **(1)** On va montrer qu'il existe un algorithme probabiliste one-sided error qui accepte avec la probabilité 1 dans le cas où **(1)** est valable, est rejette avec la probabilité $\geqslant \frac{1}{8}$ si non, et dont la complexité sera $O(k \times n^2)$

Algorithme : Choisir au hasard $u = \{u_1, u_2, ..., u_k\}$ et $v = \{v_1, v_2, ..., v_k\}$ $in\{0,1\}^k$, puis on calcule

$$U = {A_1}^{u_1} {A_2}^{u_2} ... {A_k}^{u_k}$$
$$V = {A_1}^{u_1} {A_2}^{u_2} ... {A_k}^{u_k}$$

Si $UV = VU$ **(2)**, on accepte **(1)**. Sinon, on rejette.

Pour vérifier **(2)** (et généralement pour vérifier $X = Y$ pour matrices X, Y quelconques), on choisit au hasard une vecteur $r \in \{0,1\}^n$, puis si $UV.r = VU.r$, on accepte **(2)** et on rejette sinon.

Multiplication : La multiplication de r avec UV et VU se fait par les multiplications de r avec les éléments ${A_i}^{u_i}, {A_i}^{v_i}$ de U et V de droit à gauche :

$$UV.r = ({A_1}^{u_1} ... ({A_k}^{u_k} ({A_1}^{v_1} ... ({A_k}^{v_k}.r) ...)) ...)$$
$$VU.r = ({A_1}^{v_1} ... ({A_k}^{v_k} ({A_1}^{u_1} ... ({A_k}^{u_k}.r) ...)) ...)$$

donc par $4k$ multiplications de r avec des matrices $n \times n$ qui nécessitent une complexité de l'ordre $4k \times n^2$. La complexité de cet algorithmes est donc $O(k \times n^2)$.

Analyse de l'algorithme : C'est un algorithme one-sided error :

Si **(1)** est valable, l'algorithme accepte toujours (avec probabilité 1).

Si **(1)** n'est pas valable, ça veut dire que $\exists i \neq j$ tel que $A_i A_j \neq A_j A_i$ **(3)**. On va montrer que dans ce cas

$$Pr[rejette] \geq \frac{1}{8}$$

En effet, dans le cours 1, on a montré (pour une structure d'un groupe général) que si **(3)** est valable, alors

$$1)\ Pr_{u,v}[UV \neq VU] \geq \frac{1}{4}\ \textbf{(4)}$$

En plus, on a montré aussi (pour le cas général de la comparaison de deux matrices de dimension n) que

$$Pr_r[UVr \neq VUr | UV \neq VU] \geq \frac{1}{2}$$

de 1) et 2) on déduit :

$Pr_{u,v,r}[rejette|(3)]$
$= Pr[UVr \neq VUr \wedge UV = VU|(3)] + Pr[UVr \neq VUr \wedge UV \neq VU|(3)]$
$= 0 + Pr[UVr \neq VUr|UV \neq VU \wedge (3)] \times Pr[UV \neq VU|(3)]$
$\geq \frac{1}{2} \times \frac{1}{4} = \frac{1}{8}.$

Donc

$$Pr[X = UVr = VUr = Y|(3)] \leq \frac{7}{8}$$

Pour améliorer ce résultat, on va simplement répéter plusieurs fois l'essai en rejettant si un des essais rejette. Alors après T essais :

$$Pr[accepte] = Pr[X_1 = Y_1 \wedge X_2 = Y_2 \wedge ... \wedge X_T = Y_T] \leq \left(\frac{7}{8}\right)^T \leq \frac{1}{2} \text{ (pour T assez grand).}$$

Pour que **(4)** soit vrai, d'après le cours 1, il faut assurer qu'on travaille sur une structure de groupe, c'est pour ça qu'on a supposé l'inversibilité des matrices $A_1, A_2, ..., A_k$.

Exemple 2 : Groupe libre
Soit $\Sigma = \{a, b, a^{-1}, b^{-1}\}$ un alphabet
L'ensemble Σ^* qui contient tous les mots créés de l'alphabet Σ dont le mot vide ε :

$$\Sigma^* = \{\varepsilon, a, b, aa^{-1}, ba, bab, abab^{-1}, ...\}$$

Sur l'ensemble Σ^* on définit une relation d'équivalence par transitivité :

$$aa^{-1} \equiv a^{-1}a \equiv \varepsilon$$
$$bb^{-1} \equiv b^{-1}b \equiv \varepsilon$$

Exemple : $a^{-1}baa^{-1}b^{-1}a \equiv \varepsilon$
Problème : Soit $\omega \in \Sigma^*$, trouver un algorithme probabiliste en temps $O(n)$ et en espace $O(logn)$ qui décide si $\omega = \varepsilon$ dans l'ensemble Σ^* muni de cette relation d'équivalence.
Indication : On correspond chaque a et b à une matrice :

$$a \longrightarrow A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, a^{-1} \rightarrow A^{-1}$$
$$b \longrightarrow B = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, b^{-1} \rightarrow B^{-1}$$

Théorème 2.1. *Soit W la matrice obtenue de ω en substituant a par A et b par B, alors*

$$\omega = \varepsilon \Leftrightarrow W = Id_2$$

L'algorithme naïf consiste à calculer W et puis le comparer à Id_2. En remarquant que $a^n \longrightarrow A^n = \begin{bmatrix} 1 & 2^n \\ 0 & 1 \end{bmatrix}$ et $b^n \longrightarrow B^n = \begin{bmatrix} 1 & 0 \\ 2^n & 1 \end{bmatrix}$ on constate que si $|\omega| = n$, alors W a des entrées en $O(2^n)$ et l'algorithme nécessite l'espace $O(n)$ bits pour encoder W.

L'idée de l'algorithme : Soit p un premier entier choisi au hasard entre 1 et n^4 (ici $n = |\omega|$. Au lieu de calculer W et la comparer à Id_2 on calcule W module p et on accepte quand $W = Id_2$ mod p. On a besoin de stocker 4 nombres entiers au plus p de la matrice W, donc a besoin un espace $O(logp)$ pour l'encoder. Si $n = |\omega|$ alors algorithme consiste à calculer, à chaque étape, le module p de la matrice W^k avant de passer à l'étape suivante jusqu'à ce que k soit n, donc nécessite $O(n)$ multiplications et opérations module p.
Analyse de l'algorithme : L'algorithme probabiliste est de type "one-sided error" :
Si $\omega = \varepsilon$ **(5)** algorithme accepte avec probabilité 1.
Si $\omega \neq \varepsilon$ on a besoin d'évaluer la probabilité que l'algorithme accepte quand même. On a besoin de deux théorèmes :

Théorème 2.2. *Soit N un entier, il y a au plus logN diviseurs premiers.*

Preuve: Ecrivez N sous la forme d'un produit de tous ses diviseurs premiers : $N = p_1 p_2 ... p_m$ (p_i et p_j peut être égaux). Parce que $p_i \geq 2\ \forall i = 1, 2, ..., m$ on a $N \geq 2^m$ donc le nombre de diviseurs premiers de $N \leq m \leq logN$. □

Théorème 2.3. *Dans l'ensemble* $\{1, 2, ..., M\}$ *il y a au moins* $\frac{M}{logM}.C$ *avec une constante* C *quelconque.*

Preuve: Cette théorème est une conséquence du **théorème des nombres premiers** : Soit $\pi(x)$ le nombre de nombres premiers inférieurs à x, alors lors que $x \to \infty$ on a

$$\pi(x) \sim \frac{x}{logx}$$

□

Supposons que $\omega \neq \varepsilon$, alors $W - Id_2 \neq 0$ ça veut dire qu'il existe un paramètre T non nul de cette matrice. L'algorithme accepte indique que le nombre premier p qu'on a choisi est un diviseur premier de T. Dans $[1, n^4]$ il existe au moins $C.\frac{n^4}{log(n^4)}$ nombres premiers et il existe au plus $logT \leq log(2^n) = n$ diviseurs premiers de T. Donc la probabilté que l'algorithme accepte est inférieure à :

$$Pr[accepte|(5)invalid] \leq \frac{n}{C.\frac{n^4}{4.logn}} \leq \frac{4}{Cn^2} \leq \frac{1}{2}$$

L'algorithme nécessite un générateur aléatoire qui génère uniformément un nombre premier entre 1 et n^4.

L'ensemble de classes d'équivalence dans Σ^* constitue le groupe libre de a et b.

2.2 Algorithmes du type "fingerprint" polynomial

Dans cette partie, on cherche, pour chaque objet, un "fingerprint" polynomial afin de faciliter la comparaison entre ces objets. Comme dans l'exemple du groupe libre, les "fingerprints" doivent satisfaire le fait que les objets qui ont un même "fingerprint" seraient probablement égaux. On rappelle cette lemme qui a été prouvée dans le cours précédent :

Lemme 2.4. *Soit $P(x_1, x_2, ..., x_n)$ un polynomial de degré d sur un corps fini $\Re$. Si P est non nul sur $\Re$ alors*

$$Pr_{a_1,...,a_n \in \Re}[P(a_1, ..., a_n) \neq 0] \geq 1 - \frac{d}{|\Re|}$$

Pattern Matching :
Problème : Soit un pattern $u \in \{0,1\}^m$ et un mot $\omega \in \{0,1\}^n$ avec $n \geq m$. Le but et de trouver s'il existe i tel que u correspond à la partie de ω à partir de i, ça veut dire $\omega[i]\omega[i+1]...\omega[i+m-1] = u$.
Il existe bien sur un algorithme glouton et naïf qui consiste à vérifier à chaque position i si la partie de ω de i est u. C'est un algorithme déterministe avec la complexité $O(m.n)$. On peut aussi appliquer la théorie d'automates et obtenir un autre algorithme déterministe avec la complexité $O(m+n)$. Ici sans chercher à construire un tel système d'automates, on obtient la même complexité avec un algorithme probabiliste.

Définition 2.5. *Pour chaque mot $u \in \{0,1\}^n$, on correspond à un polynomial*

$$P_u = \sum_{i=1}^{n} u[i]X^{i-1}.$$

Théorème 2.6. *Soient $u, v \in \{0,1\}^n$, et p nombre premier tel que $p \geq n^3$.*
Si $u = v$ alors $\forall a \in \{0, 1, ..., p-1\}$ on a $P_u(a) = P_v(a) \mod p$.
Si $u \neq v$ alors $Pr_{a \in \{0,1,...,p-1\}}[P_u(a) \neq P_v(a) \mod p] \geq 1 - \frac{1}{n^2}$.

Preuve: On ne considère que le cas $u \neq v$. Alors $P_u \neq P_v$ ou $P_u - P_v \neq 0$.
Par la lemme de Schartz-Zippel :

$$Pr_{a \in \{0,1,...,p-1\}}[(P_u - P_v)(a) \neq 0 \mod p] \geq 1 - \frac{n-1}{n^3} \geq 1 - \frac{1}{n^2}.$$

□

$P_u(a) \mod p$ est le "fingerprint" de u.

Algorithme : Soit p un nombre premier entre n^3 et $2n^3$ quelconque.
On choisit au hasard $a \in \{0, 1, ..., p-1\}$, calcule :
- $b \leftarrow a^{m-1}$
- $f_u = P_u(a)$
- $f_v = P_{\omega[0,m-1]}(a)$
- $i \leftarrow m-1$.

Faire : {

si $f_u = f_v$ affiche i
$f_v \leftarrow \dfrac{f_v - \omega[i-m+1]}{a} + \omega[i+1] \times b$
$i \leftarrow i+1$
} Tant que $i \leq n-1$;
Renvoyer $\emptyset$
Analyse de l'algoritme : La complexité est $O(m+n)$.
Pour tout i, si u est un motif de ω à i, alors l'algorithme affiche i
Si u n'est pas un motif de ω à i, ça veut dire $u \neq \omega[i-m+1, i]$ alors l'algorithme affiche i avec la probabilité $\leq \dfrac{n-1}{p} \leq \dfrac{1}{n^2}$.
Alors l'algorithme est one-sided error. Si u apparait dans ω, 'algorithme affiche le résultat. Par contre, si u n'apparait pas dans ω, alors
$Pr[\exists i$ tel que algo affiche$[i-m+1, i]]$
$= Pr[$Algo affiche $[1, m]$ ou $[2, m+1]$ ou ...ou$[n-m+1, n]]$
$\leq (m-n) \times \dfrac{n}{p} \sim \dfrac{1}{n}$ car $p \sim n^3$
Arbres et sous-arbres :
Problème : Soient deux arbres T_1, T_2 connectes, non ordonnées, de hauteur h et de taille n. Le but est de décider si T_1 et T_2 sont isomorphes. On cherche un algorithme de temps $O(n)$ et de coût d'espace $O(n \times logn)$.

Définition 2.7. *Pour chaque arbre T avec la racine u, on définit le polynomial $f_{T,u}$ correspondant à T de manière récursive comme suivant :*
- Si T a un seul point u, alors $f_{T,u} = X_0$
- Si la hauteur de l'arbre $h \geq 1$ et T a k sous-arbres $T_i, 1 \leq i \leq k$ avec les racines u_i, alors

$$f_{T,u} = \prod_{i=1}^{k} (X_h - f_{T_i,u_i})$$

Théorème 2.8. *Les deux arbres T_1, T_2 avec deux racines u_1, u_2 sont isomorphes si et seulement si*

$$f_{T_1,u_1} = f_{T_2,u_2}$$

Preuve: Par la définition de polynomiaux "fingerprint", si les deux arbres sont isomorphes, ils ont un même polynomial f correspondant. Par contre, supposons que $f_{T_1,u_1} = f_{T_2,u_2}$. Par la construction, le degré d de ces deux polynomiaux est égal à la hauteur h de deux arbres plus un. On prouve que deux arbres sont isomorphes par récursif en h, la hauteur des arbres qui est clairement la même pour ces deux arbres.
Pour $h = 0$, l'assertion est valable.
Supposons que l'assertion est vraie pour toute hauteur $< h$. Considérons $f_{T_1,u_1} = f_{T_2,u_2}$ comme un polynomial de la variable X_h. Car ce polynomial a une unique factorisation de degré 1 en X_h, alors T_1 et T_2 doivent avoir un même nombre de sous-arbres (qui est k) et en plus, si on appelle $T_{i,j}$ les sous-arbres de T_i, il existe une permutation π de $\{1, 2, ..., k\}$ tel que

$$\forall i = 1, 2, ..., k, f_{T_{1,i},u_{1,i}} = f_{T_{2,\pi(i)},u_{2,\pi(i)}}$$

D'après l'hypothèse de récursion, le sous-arbre $T_{1,i}$ est isomorphe au sous-arbre $T_{2,\pi(i)}$ pour tout i, et donc T_1 est isomorphe à T_2. □

Algorithme : On obtient des polynomiaux $f_{T,u}$ pour les arbres T qui ont la même propriété avec les polynomiaux P_u obtenus à l'exemple précédent : les deux arbres isomorphes se caractérisent par un même polynomial. Le problème se ramène à la comparaison de polynomiaux. En utilisant le théorème de Schwartz-Zippel et la même stratégie que celle utilisée à l'exemple précédent, on obtient un algorithme probabiliste one-sided error qui satisfait les contraintes du problème.

2.3 Problème SAT

Définition 2.9. *Soient $X_1, X_2, ..., X_n$ les variables logiques. Un litéral l est sous la forme X_i ou $\overline{X_i}$.*
Une clause C est une union de litéraux, par ex : $C = X_1 \vee X_2 \vee \overline{X_3}$
La clause C contenant au plus k variables est aussi appelée une k-clause. Une formule SAT θ est sous la forme $\theta = C_1 \wedge C_2 \wedge ... \wedge C_m$ dans laquelle C_i sont des clauses. Une formule SAT θ est appelée une formule k-SAT si θ contient seulement des k-clauses.
Le problème k-SAT consiste à décider s'il existe, pour une formule k-SAT quelconque, une affectation a des variables de cette formule, tel que la valeur de cette formule est 1.

Théorème 2.10. *2-SAT $\in P$*
k-SAT est NP-complet pout tout $k \geq 3$

Théorème 2.11. *Il existe un algorithme déterministe en temps $O(n+m)$ pour le problème 2-SAT, ici n est le nombre de variables et m est le nombre de clauses.*

On peut supposer que la formule ne contient que des 2-clauses. En effet, la présence de chaque 1-clause dans la formule implique tout de suite la valeur de la variable correspondante. On ne considère ici les variables qui ne sont pas encore affectées.

Définition 2.12. *Chaque formule 2-SAT θ correspond à un graphe $G(\theta)$ orienté défini comme suivant :*
• *$G(\theta)$ a les commets qui sont les litéraux de toutes les variables ; $X_1, X_2, ..., X_n, \overline{X_1}, \overline{X_2}, ..., \overline{X_n}$*
• *Si θ contient la clause $C = \alpha \vee \beta$ alors les arêtes $\overline{\alpha} \to \beta$ et $\overline{\beta} \to \alpha$ sont dans $G(\theta)$.*

Théorème 2.13. *θ est satisfaisable si et seulement si $\forall i \in \{1, 2, ..., n\}$, les sommets X_i et $\overline{X_i}$ ne sont pas dans une même composante fortement connexe dans $G(\theta)$.*

Preuve: 1) Supposons que $\exists i$ tel que X_i et $\overline{X_i}$ sont dans une même composante connexe, donc il existe deux chemins $X_i \to \overline{X_i}$ et $\overline{X_i} \to X_i$ dans G, et la valeur de la formule θ est 1. On a deux cas :
• Si $X_i = 1$, le chemin de $X_i \to \overline{X_i}$ est $l_0 = X_i, l_1, ..., l_t = \overline{X_i}$. En remarquant que $l_t = 0$ et $l_0 = 1$, il existe k, $0 \leq k \leq t-1$ tel que $l_k = 1$ et $l_{k+1} = 0$. D'après la construction du graphe $G(\theta)$, la clause $C = \overline{l_k} \vee l_{k+1}$ est dans θ. On constate que la valeur de C est 0, ce qui rend faux toute la formule θ, contradictoire.

• Si $X_i = 0$, on considère le chemin $\overline{X_i} \to X_i$ et on recoit la contradiction similaire.
2) Supposons que $\exists i, X_i$ et $\overline{X_i}$ ne sont pas dans une même composante connexe. On construit une affectation comme suivant :
Construction d'une affectation possible :
Tant que $\exists X_i$ non affectée :
a) Choisir un litéral l parmi les litéraux qui ne sont pas affectés, tel que $\nexists$ chemin de l à $\bar{l}$ dans G. De l, on considère tous les chemins sortant de l et rend 1 tous les sommets passés par ces chemins. Cela veut dire que s'il existe un chemin de l à l', alors on rend $l' = 1$. b) Pour tous les litéraux l_1 de la formule, si $\overline{l_1}$ est déjà affecté, alors on affecte l_1 tel que $l_1 = 1 - \overline{l_1}$. c) Pour tous les litéraux l_2 dans la formule, si il existe un chemin entre l_2 et l_3 et puis $l_3 = 0$, alors on rend $l_2 = 0$. d) Répéter b) et c) certaines fois si nécessaire.
Ce processus se termine car à chaque étape le nombre de litéraux affectés est augmenté au moins d'un. Le nombre d'opérations a), b), c) qu'on doit effectuer ne peut pas dépasser $2n$ qui est le nombre total de litéraux. □

Algorithme probabiliste pour 2-SAT :
• prendre une affectation $a \in \{0,1\}^n$ quelconque.
• Tant qu'il existe une clause C tel que $C(a) = 0$:
(6) Soit X_i une des variables de C prise au hasard, on inverse la valeur de X_i et $\overline{X_i}$
Renvoyer satisfaisable.
Cet algorithme est appelé WalkSAT.

Théorème 2.14. *Si θ non satisfaisable, WalkSAT ne s'arrête jamais.*
Si θ est satisfaisable, alors WalkSAT trouve une affectation positive (qui rend positive la formule) en temps moyen $\leq n^2$.

Dans le cas où θ est satisfaisable, si T est le nombre d'itérations et si on a $E(T) \leq n^2$, alors

$$Pr[T \geq k.n^2] \leq \frac{1}{k.n^2} \times E(T) \leq \frac{1}{k}, \forall k > 0$$

Alors on peut prendre $k = 2$ et mettre $2n^2$ comme une borne de nombre d'itérations, au delà de $2n^2$ itération, si on ne trouve pas une affectation positive, on rejette la formule θ. Un tel algorithme probabiliste est one-sided error du fait que si θ est non satisfaisable, il le rejette toujours, et si θ est satisfaisable, l'algorithme trouve une affectation avec la probabilité $\geq \frac{1}{2}$. Donc il ne nous reste qu'à démontrer le théorème.

Preuve: On ne considère que le cas où θ est satisfaisable, donc il existe une affectation u satisfaisant toutes les clauses.
Soit $E(x)$ le nombre moyen d'itérations avant de trouver une affectation positive en choisissant $a = x$ au premier étape, et soit $E(i)$ le maximum de $E(x)$ sur toutes les assignations x tel que la distance Hamming $d(x,u) = i$. On va montrer que :

$$\begin{cases} E(0) = 0, (*) \\ E(n) \leq 1 + E(n-1), (**) \\ E(i) \leq 1 + \dfrac{E(i-1) + E(i+1)}{2}, \text{ pour } 1 \leq i \leq n-1, (***) \end{cases}$$

($*$) En effet, si $x = u$ alors l'algorithme se termine et donc $E(x) = 0$, la première assertation est vérifiée.

($**$) Soit x une assignation quelconque tel que $d(x, u) = n$. Si x est aussi une affectation positive comme u, alors l'algorithme se termine et $E(x) = 0$. Si x n'est pas une solution et $x = \overline{u}$, l'algorithme va inverser une variable quelconque de x et l'affectation obtenue x' a exactement $n - 1$ variables qui se diffèrent de celles de u. A partir de x' l'algorithme a besoin en moyenne au plus $E(n - 1)$ itérations pour trouver une affectation positive, donc $E(x) \leq 1 + E(n - 1)$. On voit bien que dans tous les cas $E(x) \leq 1 + E(n - 1)$ $\forall$ assignation x tel que $d(x, u) = n$, donc ($**$) est vérifié.

($***$) Soit x une assignation quelconque tel que $d(x, u) = i$. Si x est aussi une affectation positive comme u, alors l'algorithme se termine et $E(x) = 0$. Si x n'est pas encore une solution, alors soit C la clause non satisfaite choisie. Parce que $C(u) = 1$, au moins une des deux variables de C a une valeur différente de celle qu'elle a dans u ; donc avec la probabilité p au moins $\frac{1}{2}$, le changement de x en x' entraîne $d(x', u) = d(x, u) - 1$, et avec la probabilité $1 - p$ au plus $\frac{1}{2}$, on a $d(x'', u) = d(x, u) + 1$ avec l'assignation x'' obtenue de x après la première itération. Alors :

$$E(x) \leq p \times E(x') + (1 - p) \times E(x'') + 1$$

Mais on remarque que $d(x', u) = i - 1$ et $d(x'', u) = i + 1$ et $E(i)$ est une fonction croissante en i, donc

$E(x) \leq p \times E(i - 1) + (1 - p) \times E(i + 1) + 1$

$\leq 1 + \dfrac{E(i - 1) + E(i + 1)}{2}$

Cette inégalité est aussi vérifiée quand x est déjà une solution (donc $E(x) = 0$), donc se vérifie pour tout x tel que $d(x, u) = i$. En prenant cette inégalité sur tout x tel que $d(x, u) = i$, on déduit la troisième assertation.

A partir de ces trois assertations, on déduit que $E(i) \leq n^2 - (n - i) \leq n^2$ pour tout i $\square$

Algorithme probabiliste pour 3-SAT : On essaie d'appliquer l'algorithme au-dessus au problème 3-SAT et évalue sa complexité. En gardant les meêm notations, on a toujours, dans ce cas, $E(0) = 0$ et $E(n) = 1 + E(n - 1)$. En revanche, à chaque fois qu'on modifie la valeur d'une variable dans une clause invalide, la probabilité qu'on ajoute encore une différence entre x et u est au plus $\frac{2}{3}$ et seulement avec la probabilité au moins $\frac{1}{3}$ qu'on diminue $d(x, u)$, donc si on répète la même analyse que précédent, on a :

$$E(i) \leq 1 + \frac{E(i - 1) + 2E(i + 1)}{3}$$

ce qui rend $E(k) \sim 2^k$, et la complexité de l'algorithme ne peut pas rester à la classe polynomiale.

 Une amélioration de cet algorithme est, au lieu de choisir l'assignation a quelconque au début, on choisit a uniformément aléatoire. Le fait qu'on prend a quelconque nous force à analyser la performance dans le pire cas parmi toutes les assignations a possibles, alors que en randomizant ce choix, on évalue la performance en moyenne et non plus dans le pire cas.

Algorithme amélioré :
- Prendre une assignation a aléatoirement.
- Quand on n'a pas trouvé une affectation positive, on répète **(6)**.
- On répète **(6)** au plus $3n$ fois avant de conclure.

Lemme 2.15. *La probabilité qu'une solution soit trouvée après $3n$ itérations **(6)** est $\geq \theta(1)\frac{1}{\sqrt{n}}\left(\frac{3}{4}\right)^n$*

Corollaire 2.16. *Dans le cas où la formule est satisfaisable, en répétant l'algorithme amélioré en $\theta\left(\sqrt{n}\left(\frac{4}{3}\right)^n\right)$ fois on trouve une solution avec probabilité au moins $\frac{1}{2}$.*