Heuristiques de Recherche 2020 MPRI

Résoudre des formules SAT

Christoph Dürr

1 Les algorithmes exponentiels

Nous allons voir des algorithmes pour résoudre des formules SAT. Ils seront toujours des algorithmes exponentiels, mais l'amélioration peut être importante. Par exemple pour $n = 80$, un algorithme qui effectue 2^n opérations à raison de 10^8 opérations par seconde, mettrait 390 millions d'années, alors qu'un algorithme de complexité $(4/3)^n$ mettrait seulement une minute et demi.

2 Simplification

Un ingrédient important aux algorithmes de cette section est la simplification d'une formule. Elle consiste simplement en l'application répétée des règles suivants — qui sont plutôt de bon sens.

1. Si une clause contient une variable et à la fois sa négation, alors remplacer cette clause par *Vrai*. Si une clause contient deux fois le même littéral, alors remplacer les deux par un seul.
2. Si les littéraux d'une clause C_1 sont un sous-ensemble des littéraux d'une clause C_2 alors supprimer C_2.
3. Si une clause contient la constante *Faux*, alors supprimer la constante de la clause. Si la clause devient alors vide, alors remplacer la formule entière par *Faux*.
4. Si une clause est un unique littéral x ($\overline{x}$), alors remplacer x par *Vrai* (*Faux*) partout dans la formule.
5. Si la formule contient un littéral mais pas sa négation, alors le replacer par *Vrai* partout dans la formule.

La propriété importante de la procédure de simplification est qu'elle préserve la faisabilité de la formule et qu'elle réduit le nombre de clauses, sinon le nombre de variables. Elle peut être implémentée en temps polynomial en la taille de la formule.

3 Davis-Putnam

Dans l'algorithme suivant $F|_{x=c}$ représente le résultat du remplacement de chaque occurrence de x par la constante $c \in \{0, 1\}$ dans F. Ici Simplification$_{1-5}(F)$ fait référence aux 5 règles ci-haut.

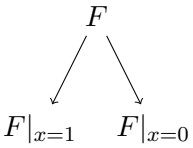

Figure 1: L'arbre de récursion de l'algorithme de Davis-Putnam

Data: Une formule F en forme normale conjonctive
Result: teste si F est satisfiable
F := Simplification$_{1-5}(F)$;
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
Soit x une variable de F;
if *DavisPutnam*$(F|_{x=1})$ **then**
  retourner Vrai;
else
  retourner DavisPutnam$(F|_{x=0})$
end

Algorithm 1: DavisPutnam

Notation

n nombre de variables

m nombre de clauses

nm borne sur la taille du codage de la formule

Cet algorithme a une complexité de l'ordre de $O(2^n \text{poly}(nm))$. Pour obtenir un meilleur algorithme en terme de n, il faudrait restreindre la taille des clauses, ou d'exprimer la complexité en terme de m, le nombre de clauses. Un ingrédient important est la procédure de simplification que nous décrivons maintenant.

Exercice 1 *Montrer que la complexité de Davis-Putnam est $O(2^m poly(nm))$.*

4 Résolution

On va ajouter une règle supplémentaire à la procédure de simplification.

6. Soit x une variable apparaissant p fois positif dans F et q fois négatif dans F, avec $p+q \geq pq$. On distingue dans F les clauses contenant x, les clauses contenant $\overline{x}$ et ceux ne contenant ni x ni $\overline{x}$. Alors F a la forme

$$\begin{aligned} &(x \vee C_1) \wedge \ldots \wedge (x \vee C_p) \\ \wedge\, &(\overline{x} \vee D_1) \wedge \ldots \wedge (\overline{x} \vee D_q) \\ \wedge\, &U, \end{aligned}$$

et on la remplace par

$$\begin{aligned}&(C_1 \vee D_1) \wedge \ldots \wedge (C_1 \vee D_q)\\ \wedge &\vdots\\ \wedge &(C_p \vee D_1) \wedge \ldots \wedge (C_p \vee D_q)\\ \wedge &U.\end{aligned}$$

Exercice 2 *La règle de résolution préserve la faisabilité de la formule.*

Exercice 3 *La procédure de simplification avec les règles 1–6, fonctionne en temps polynomial en n, m, nm.*

Théorème 1 *La procédure Davis-Putnam avec les règles de simplification 1–6, a la complexité $O(1,325^m poly(nm))$.*

Preuve : Après simplification, chaque variable de la formule x qui apparait p fois positivement et q fois négativement satisfait $\min\{p,q\} \geq 2$ et $\max\{p,q\} \geq 3$. Donc sur les formules $F|_{x=0}$ et $F|_{x=1}$, une a au plus $m-2$ clauses et l'autre au plus $m-3$ clauses. La complexité satisfait donc la récursion

$$T(m,n) \leq T(m-2,n) + T(m-3,n) + \text{poly}(nm),$$

pour $m > 0$ et $T(0,n) = O(\text{poly}(nm))$ sinon. La preuve est complétée par l'analyse standard des récursions linéaires. □

5 Récursions linéaires

Voici un outil important pour l'analyse de la complexité des algorithmes de cette section.

Théorème 2 *Si*

$$f(n) \leq a_1 f(n-1) + \ldots + a_k f(n-k),$$

pour tout $n \geq k$ et des constantes réels non-négatives $a_1, \ldots, a_k$, alors $f(n) = O(v^n)$ pour v l'unique racine réelle positive de

$$g(t) = 1 - \frac{a_1}{t} - \ldots - \frac{a_k}{t^k}.$$

Preuve : La fonction g est monotone croissante, et tend vers $-\infty$ quand t tend vers 0 et tend vers 1 quand t tend vers $+\infty$. Elle a donc une unique racine positive.

On montre par récurrence que $f(n) \leq cv^n$ pour une constante c. Pour le cas de base on choisit c suffisamment grand afin que ceci soit vrai pour tout $0 \leq n \leq k$.

Maintenant $n > k$ et et on fait l'hypothèse d'induction que $f(i) \leq cv^i$ pour tout $0 \leq i < n$. Alors

$$\begin{aligned}f(n) &\leq a_1 f(n-1) + \ldots + a_k f(n-k)\\ &\leq a_1 cv^{n-1} + \ldots + a_k cv^{n-k}\\ &= cv^n(a_1 v^{-1} + \ldots + a_k v^{-k})\\ &= cv^n(1 - g(v))\\ &= cv^n.\end{aligned}$$

□

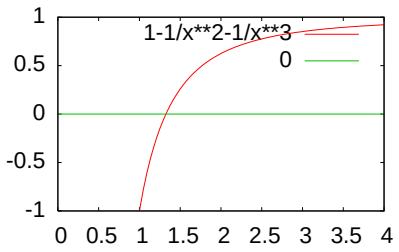

Figure 2: La racine de $t \mapsto 1 - 1/t^2 - 1/t^3$ est $1,3247$ ce qui permet d'affirmer le Théorème 1.

6 Backtracking de Monien, Speckenmeyer

Pour améliorer la procédure Davis-Putnam, on introduit une opération supplémentaire. Une affectation associe à chaque variable de la formule une valeur booléenne. L'affectation *majoritaire* associe à une variable x la valeur *Vrai* si x apparaît plus souvent dans la formule que $\overline{x}$, et associe la valeur *Faux* dans le cas contraire. Quelles sont les clauses qui ne sont *pas* validées par l'affectation majoritaires ? Mais tout simplement les clauses dont tous les littéraux sont dans leur forme minoritaire. Ces clauses vont jouer un rôle important dans la suite.

La procédure de Monien Speckmeyer commence par tester si l'affectation majoritaire valide la formule. Mais elle fait plus encore. Elle contrôle aussi le choix de la variable choisie pour brancher. Le but est de choisir une variable qui permet lors d'une récursion de diminuer le nombre de clauses d'au moins 3.

$$\begin{aligned} F = &(x \vee y \vee \overline{z}) \\ &\wedge (x \vee \overline{y} \vee z) \\ &\wedge (\overline{x} \vee y \vee z) \\ &\wedge (\overline{x} \vee \overline{y} \vee z) \\ &\wedge (\overline{x} \vee \overline{y} \vee \overline{z}) \end{aligned}$$

Figure 3: L'affectation majoritaire est $x = 0, y = 0, z = 1$ qui ne satisfait pas la première clause. Les variables de cette clause sont permises. La formule $F|_{x=T}$ n'a que 2 clauses de moins que F, mais heureusement la règle 6 (résolution) appliquée à la variable y va diminuer strictement le nombre de clauses, car elle n'apparaît qu'une fois positivement dans $F|_{x=1}$ Elle n'est pas bien faite la vie ?

Data: Une formule SAT F sur n variables
Result: teste si F est satisfiable
F := Simplification$_{1-6}(F)$;
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
if *l'affectation majoritaire valide F* **then**
  annoncer la valeur *Vrai*;
end
S'il existe une variable x qui apparaît au moins 3 fois positivement et au moins 3 fois négativement, alors choisir cette variable. Dans le cas échéant toutes les variables apparaissent 2 fois dans une forme et au moins 3 fois dans la forme opposée. Comme l'affectation majoritaire n'a pas validé toutes les clauses, il existe des clauses contentant seulement des variables dans leur forme minoritaire. Soit x une telle variable.;
if *MonienSpeckenmeyer*$(F|_{x=1})$ **then**
  retourner Vrai;
else
  retourner MonienSpeckenmeyer$(F|_{x=0})$
end

Algorithm 2: Monien-Speckenmeyer

Théorème 3 *La complexité de l'algorithme de Monien-Speckenmeyer est* $O(2^{m/3}poly(nm)) = O(1.26^m poly(nm))$.

Preuve : On va tout simplement monter que la complexité satisfait la récurrence

$$T(m,n) \leq T(m-3,n) + T(m-3,n) + \text{poly}(nm).$$

Le seul cas qui pose problème est quand toutes les variables apparaissent 2 fois dans une forme et au moins 3 fois dans la forme opposé. Soit x la variable choisie, et C une clause contenant x dans sa forme minoritaire, disons $\bar{x}$ (le cas $x \in C$ est symétrique). Alors par la règle 4, C contient aussi une autre variable y, et dans sa forme minoritaire. On pourrait penser que lors du branchement sur $F|_{x=0}$, seul les deux clauses contenant $\bar{x}$ disparaissent. Mais la cette formule y apparaît désormais 0 ou 1 fois dans sa forme minoritaire et la règle 5 ou 6 diminue strictement le nombre de clauses. En résultat les simplifications des formules $F|_{x=0}$ et $F|_{x=1}$ contiennent au moins 3 clauses de moins que F. □

7 Résoudre des formules 3-SAT

On a vu la semaine dernière que pour espérer des algorithmes qui résolvent SAT dans un temps meilleur que $\Omega(2^n)$, il faudrait restreindre la taille des clauses. C'est ce que nous faisons dans ce cours, qui présente des algorithmes pour 3-SAT, les formules donc chaque clauses contient au plus 3 littéraux.

Le tableau 1 résume la complexité de certains algorithmes de résolution de 3-SAT, illustrant ainsi la course que se font les chercheurs depuis des décennies sur ce problème. Ici on note $g(n) \in O^*(f(n))$ s'il existe une constante c tel que $g(n) \in O(f(n)\log^c f(n))$. Ainsi dans l'étoile on cache des facteurs polynomiaux de la complexité exponentielle.

	Algorithmes déterministes
$O^*(2^n)$	recherche exhaustive
$O^*(1.618^n)$	Monien,Speckenmeyer (1985)
$O^*(1.505^n)$	Kullmann (1999)
$O^*(1.481^n)$	Dantsin, Goerdt, Hirsch, Schöning (2000)
$O^*(1.439^n)$	Kutzkov, Scheder (2010)
$O^*(1.333^n)$	Moser, Scheder (2011)
	Algorithmes probabilistes
$O^*(1.587^n)$	Paturi, Pudlák, Zane (1997)
$O^*(1.5^n)$	Schöning (1998)
$O^*(1.447^n)$	Paturi, Pudlák, Saks, Zane (1998)
$O^*(1.333^n)$	Schöning (1999)
$O^*(1.32113^n)$	Iwama, Seto, Takai, Tamaki (2010)
$O^*(1.32065^n)$	Hertli, Moser, Scheder (2011)

Table 1: Complexité de certains algorithmes pour 3-SAT. (La bibliographie reste à être complétée avec des résultats depuis 2011.)

8 Algorithmes probabilistes

Pour ce cours on distingue les algorithmes déterministes pour 3-SAT, des algorithmes probabilistes. Les algorithmes probabilistes de ce cours, tournent en temps polynomial en n et soit affichent *échec*, soit trouvent une assignation satisfaisant la formule. L'analyse de l'algorithme va montrer que dans le cas où la formule est satisfiable, l'algorithme succède avec une petite probabilité p au moins. Pour l'amplifier, on peut alors répéter l'algorithme par exemple $20/p$ fois. En effet la probabilité que l'algorithme échoue à chaque fois est au plus

$$(1-p)^{20/p} \leq e^{-20} = 0,000000002.$$

9 Monien, Speckenmeyer pour 3-SAT

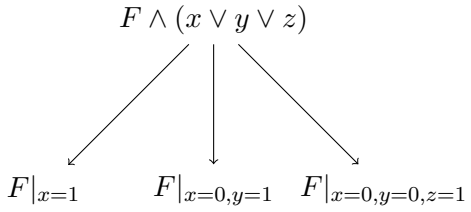

Figure 4: L'arbre de récursion de l'algorithme de Monien-Speckmeyer

Data: Une formule 3-SAT F sur n variables
Result: teste si F est satisfiable
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
Soit C une clause arbitraire de F avec un nombre minimal de littéraux, disons
$C = x \vee y \vee z$;
if *backtracking*$(F|_{x=1})$ **then**
  annoncer Vrai;
end
if *backtracking*$(F|_{x=0,y=1})$ **then**
  annoncer Vrai;
end
if *backtracking*$(F|_{x=0,y=0,z=1})$ **then**
  annoncer Vrai;
end
annoncer Faux;

Algorithm 3: Monien-Speckenmeyer-3

La complexité de cette procédure est la solution à la récursion

$$T(m,n) \leq T(m,n-1) + T(m,n-2) + T(m,n-3) + \text{poly}(nm),$$

qui est bornée par $T(m,n) = O(1.84^n \text{poly}(nm))$.

Exercice 4 *Considérez le problème NP-complet de 3-colorabilité d'un graphe de n sommets. Modélisez par une formule 3-SAT et montrez que l'algorithme Monien-Speckenmeyer-3 a une complexité $O(1.618^n poly(nm))$, où 1.618... est le nombre d'or.*

10 Recherche locale dans un voisinage de rayon k

Soient a, b deux affectations. La *distance de Hamming* entre a et b est définie comme le nombre de variables auxquelles a et b donnent des valeurs différentes. Voici un algorithme qui cherche une solution parmi toutes les affectations à distance au plus k d'une affectation donnée.

Ici a_x représente l'affectation qui diffère de a seulement dans la variable correspondant au littéral x.

Data: Une formule 3-SAT F sur n variables, une affectation a et un entier $k \geq 0$
Result: cherche s'il existe une assignation validant F et qui soit de distance au plus k de a
if $F(a)$ *est vrai* **then**
  retourner *Vrai*;
end
if $k = 0$ **then**
  retourner *Faux*;
end
Soit $C = x \vee y \vee z$ une clause de F qui n'est pas satisfaite par a;
if *RechercheBoule*$(F, a_x, k-1)$ **then**
  retourner *Vrai*;
end
if *RechercheBoule* $(F, a_y, k-1)$ **then**
  retourner *Vrai*;
end
if *RechercheBoule* $(F, a_z, k-1)$ **then**
  retourner *Vrai*;
end
retourner *Faux*;

Algorithm 4: RechercheBoule

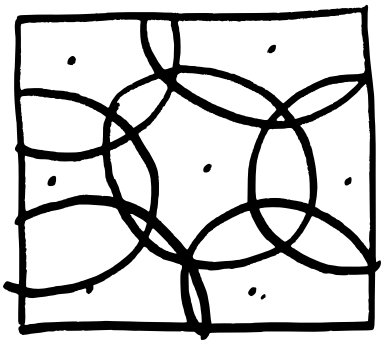

Figure 5: Couvrir l'espace de recherche avec des boules dans lesquelles on cherche une solution

La complexité de cet algorithme est clairement en $O^*(3^k)$, ignorant les facteurs polynomiaux. Alors un appel à *RechercheBoule*$(F, 0^n, n)$ succède car l'algorithme explore toutes les affectations possibles. Et ceci en temps $O^*(3^n)$, ce qui n'est pas terrible, avouez.

Une petite amélioration est d'appeler plutôt *RechercheBoule*$(F, 0^n, n/2)$ puis *RechercheBoule*$(F, 1^n, n/2)$ qui succède en temps $O(3^{n/2}) = O(1.732^n)$.

On a alors envie d'initier plusieurs recherches qui couvrirons tout l'espace de recherche pour obtenir une complexité encore plus petite. Vous n'avez pas envie ?

Ceci a été fait en analysant la probabilité de succès de *RechercheBoule*$(F, a, n/4)$ pour une affectation uniforme aléatoire a, ce qui donnait un algorithme dont la complexité en espérance est $O^*(1.5^n)$. Plus tard cet algorithme a été dérandomisé, c'est à dire qu'on a su remplacer le choix de l'affectation aléatoire par une série d'affectations dont le voisinage couvre toutes les affectations.

11 Codes couvrants

Ce qu'on cherche, c'est un ensemble $C \subseteq \{0,1\}^n$ — appelé *code* — tel que chaque mot $a \in \{0,1\}^n$ est à distance au plus d d'un mot du code $b \in C$. Et de plus on veut qu'on puisse générer en temps $O^*(|C|)$ tous les mots du code. On appèle alors C un *code couvrant de rayon d*.

Un tel code donnerait un algorithme de résolution 3-SAT en temps

$$O^*(|C|3^d).$$

Data: Une formule 3-SAT F sur n variables
Result: cherche s'il existe une assignation validant F
Soit C un code couvrant de rayon d;
for *tout mot a du code C* **do**
  if *RechercheBoule(F, a, d)* **then**
    retourner *Vrai*;
  end
end
retourner *Faux*;

Algorithm 5: RechercheCodeCouvrant

On a de la chance, car il existe de bons codes : pour chaque $0 < \rho, \epsilon < 1/2$ et un n assez grand, il existe un code couvrant de rayon $(\rho + \epsilon)n$ et de taille $2^{(1-h(\rho)+\epsilon)n}$. Ici $h : [0,1] \to [0,1]$ est la fonction d'*entropie*, définie par

$$h(\rho) = -\rho \log_2(\rho) - (1-\rho)\log_2(1-\rho).$$

Théorème 4 *Pour chaque $0 < \epsilon < 0.1$ fixé il existe un algorithme de résolution pour 3-SAT de complexité $O^*((1.5 + 3\epsilon)^n)$.*

Preuve : On va utiliser un code couvrant C de rayon $(1/4 + \epsilon)n$ et de taille au plus

$$2^{(1-h(1/4)+\epsilon)n} = (2(1/4)^{1/4}(3/4)^{3/4}2^{\epsilon})^n.$$

La complexité de l'algorithme 5 est alors de l'ordre

$$\begin{aligned} |C|3^d &\leq (2(1/4)^{1/4}(3/4)^{3/4}2^{\epsilon})^n 3^{n/4} 3^{\epsilon n} \\ &= (2(1/4)^{1/4}(3/4)^{3/4}3^{1/4}6^{\epsilon})^n \\ &= (2(3/4)6^{\epsilon})^n \\ &\leq (1.5 + 3\epsilon)^n \end{aligned}$$

où la dernière inégalité est valide pour $\epsilon < 0.1$. □

12 Recherche locale d'Uwe Schöning

L'algorithme suivant, remplace la recherche systématique d'un voisinage par une procédure aléatoire.

```
Data: Une formule 3-SAT F sur n variables, un entier positif d
Result: produit une assignation a qui valide la formule, s'il en existe une
for ever do
    Soit a une assignation choisie uniformément au hasard;
    for 3n fois au plus do
        if F(a) =Vrai then
            retourner Vrai;
        end
        Soit C une clause arbitraire de F, tel que C(a) =Faux;
        Soit x un littéral aléatoire de C;
        Changer la valeur a[x] affectée à x
    end
end
```

Algorithm 6: Recherche locale de Uwe Schöning

Quand on change la valeur du littéral x, on valide cette clause, mais au risque d'invalider d'autres. Comment s'assurer qu'il y aura du progrès, au moins en espérance ? Cet algorithme ne s'arrête jamais si la formule n'est pas satisfiable, mais sinon il trouve une assignation valide a^* au bout d'un certain temps, et on va analyser l'espérance de temps.

Pour cela on fixe une assignation valide arbitraire a^* et on mesure la *distance de Hamming* avec a, c'est-à-dire le nombre de variables où a et a^* diffèrent.

Lemme 1 *À chaque étape, la distance de Hamming* $d(a, a^*)$ *diminue de* 1 *avec probabilité au moins* $1/3$ *et augmente de* 1 *avec une probabilité au plus* $2/3$.

Preuve : Soit k le nombre de différences entre a et a^* sur les trois littéraux de la clause C choisie dans l'étape.

cas $k = 0$ impossible, car C est valide dans a^* et l'algorithme a choisit une clause qui n'est pas valide dans a.

cas $k = 1$ alors avec probabilité $1/3$ l'algorithme inverse l'unique variable où a et a^* différent dans C, et la distance diminue. Dans l'autre cas, donc avec probabilité $2/3$ la distance augmente.

cas $k = 2$ **ou** $k = 3$ alors la probabilité que la distance diminue est encore plus grande.

□

Le comportement de l'algorithme décrit une marche dans le graphe de la figure 6. L'étiquette dans les sommets représente la distance de Hamming avec l'assignation valide a^*.

Pour une analyse pessimiste on peut associer une probabilité $1/3$ aux arcs vers la gauche et $2/3$ aux arcs vers la droite.

On va analyser une marche aléatoire sur le graphe ci-haut, mais de manière pessimiste. Pour rendre les choses plus difficiles à l'algorithme, on considère le graphe dont les sommets sont les entiers $\mathbb{Z}$, et on cherche à déterminer la probabilité que l'algorithme atteint le sommet 0 du sommet j au bout de $3j$ étapes. Pire que ça, on veut déterminer la probabilité d'un tel chemin qui utilise exactement j pas du type $i \to i + 1$ et $2j$ pas de type $j \to j - 1$. Cette probabilité est au moins

$$q_j \geq \binom{3j}{j} \left(\frac{2}{3}\right)^j \left(\frac{1}{3}\right)^{2j}.$$

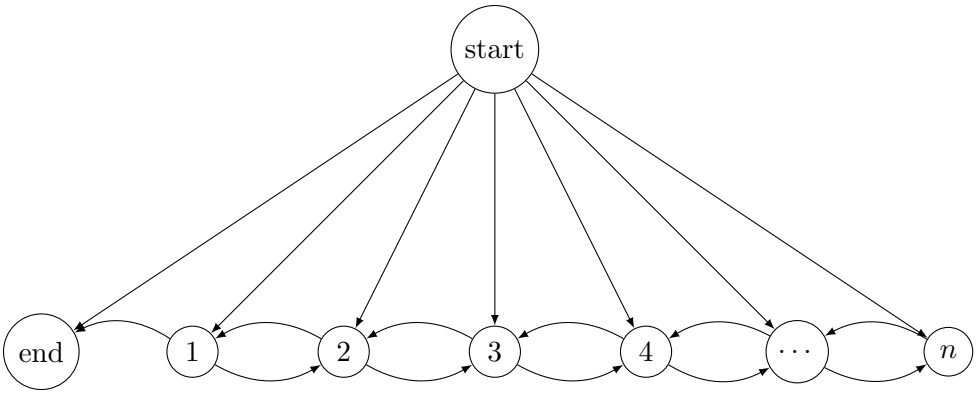

Figure 6: Le comportement de l'algorithme correspond à une marche aléatoire dans ce graphe.

Utilisant la formule d'approximation de Stirling $n! = \Theta(\sqrt{n} \cdot (n/e)^n)$, on obtient

$$\binom{3j}{j} = \Theta\left(\frac{1}{\sqrt{j}} \frac{3^{3j}}{2^{2j}}\right).$$

Et ainsi la probabilité de succès de la boucle intérieure de l'algorithme de Schöning est pour une constante c au moins

$$\begin{aligned} & c \cdot \sum_{j=0}^{n} \frac{1}{\sqrt{j}} \cdot \frac{1}{2^j} \binom{n}{j} \cdot \frac{1}{2^n} \\ \geq & \frac{c}{\sqrt{n}} \sum_{j=0}^{n} \binom{n}{j} \frac{1}{2^{n+j}} \\ = & \frac{c}{\sqrt{n}} \left(\frac{3}{4}\right)^n, \end{aligned}$$

où la dernière égalité utilise l'expansion binomiale de $(1/2 + 1/4)^n$.

Pour conclure on utilise l'observation de la section 8 pour montrer que la complexité de l'algorithme de Schöning est $O^*((4/3)^n)$ et qu'il succède avec une grande probabilité.