Chap. 2 – Recherche exhaustive et *backtrack*

HAI503I – Algorithmique 4

Bruno Grenet

Université de Montpellier – Faculté des Sciences

Le problème Sat

Définition

Entrée : une formule logique φ à n variables booléennes, sous *forme normale conjonctive* (CNF)

Sortie : une affectation des variables qui satisfasse φ ; « insatisfiable » sinon

Formule logique CNF : *conjonction de disjonction de littéraux*

- Littéraux : $x_1, \neg x_1, \ldots, x_n, \neg x_n$
- Disjonction : $C = x_1 \vee \neg x_3 \vee \neg x_4$ (clause)
- Conjonction : $C_1 \wedge C_2 \wedge \cdots \wedge C_k$

$$\varphi(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2$$

Affectation satisfaisante ou non

- $(x_1, x_2, x_3) = (\text{FAUX}, \text{FAUX}, \text{FAUX})$ satisfait φ
- $(x_1, x_2, x_3) = (\text{VRAI}, \text{FAUX}, \text{VRAI})$ ne satisfait pas φ

Sat : résolution par recherche exhaustive

Algorithme: tester toutes les affectations possibles

Questions

- Comment parcourir toutes les affectations possibles ?
- Comment tester si une affectation satisfait la formule ?
- Quelle est la complexité de cet algorithme ?

Question préalable

- Quelle représentation informatique pour les formules et les affectations ?

Sat : représentation informatique

Représentation d'une formule CNF

- Conjonction $C_1 \wedge \cdots \wedge C_k \to$ tableau de clauses
- Clause $C = \ell_1 \vee \cdots \vee \ell_t \to$ tableau de littéraux
- Littéral $x_i \to$ entier i ; $\neg x_i \to$ entier $-i$

> Représentation de φ : tableau de tableaux d'entiers

Exemple

$\varphi(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2 \to$ `phi = [[-1,2],[1,2,-3],[-2]]`

Représentation d'une affectation

- Tableau de booléens : (FAUX, VRAI, FAUX) $\to$ `[False, True, False]`
- Plus pratique : tableau $\pm 1 \to$ VRAI $= 1$; FAUX $= -1$

$\neg x_3$ $\quad x_3 =$ FAUX

$-3 \times -1 = 3$

Sat : tester une affectation

Idée de l'algorithme

- Parcourir toutes les clauses $\rightarrow$ elles doivent toutes être satisfaites
- Clause satisfaite : (au moins) un littéral est satisfait
- Littéral satisfait :
  - Littéral non nié : affectation vrai $\rightarrow \ell > 0$ et $A_{[\ell-1]} = 1$
  - Littéral nié : affectation faux $\rightarrow \ell < 0$ et $A_{[-\ell-1]} = -1$

SAT : tester une affectation

Idée de l'algorithme

- Parcourir toutes les clauses $\rightarrow$ elles doivent toutes être satisfaites
- Clause satisfaite : (au moins) un littéral est satisfait
- Littéral satisfait :
  - Littéral non nié : affectation VRAI $\rightarrow \ell > 0$ et $A_{[\ell-1]} = 1$
  - Littéral nié : affectation FAUX $\rightarrow \ell < 0$ et $A_{[-\ell-1]} = -1$

TESTAFF(φ, A) :

Pour C dans φ :
    OK $\leftarrow$ FAUX
    Pour ℓ dans C :
        Si $\ell \times A_{[|\ell|-1]} > 0$:
            OK $\leftarrow$ VRAI
    Si NON(OK) : Renvoyer FAUX
Renvoyer VRAI

Complexité

Linéaire en la taille de $\cancel{C}$ φ
= somme des tailles des clauses

SAT : parcourir les affectations

Affectations = mots binaires

- Affectation : tableau de n valeurs ± 1
- *Bijection* avec les *mots binaires de longueur n* : $1 \to 1$; $-1 \to 0$
- Analogie : parcourir les affectations $\Leftrightarrow$ compter de 0 à $2^n - 1$
- Opération nécessaire : AFFSUIVANTE $\Leftrightarrow$ incrémenter un compteur binaire

Sat : parcourir les affectations

Affectations = mots binaires

- Affectation : tableau de n valeurs ± 1
- *Bijection* avec les *mots binaires de longueur n* : $1 \to 1$; $-1 \to 0$
- Analogie : parcourir les affectations $\Leftrightarrow$ compter de 0 à $2^n - 1$
- Opération nécessaire : AffSuivante $\Leftrightarrow$ incrémenter un compteur binaire

AffSuivante(A) :

1. $i \leftarrow 0$
2. Tant que $i < n$ et $A_{[i]} = 1$:
3. $\quad A_{[i]} \leftarrow -1$
4. $\quad i \leftarrow i + 1$
5. Si $i = n$: renvoyer « Fin »
6. $A_{[i]} \leftarrow 1$
7. Renvoyer A

10010111
+1
10011000

- 0111111

Propriétés

- Si on part de [-1, . . . , -1], AffSuivante parcourt toutes les affectations
- Complexité :
  - $O(n)$ dans le pire cas
  - $O(1)$ *amortie* (chap. 3)

SAT : algorithme de recherche exhaustive

RECHERCHEEXHAUSTIVE(φ) :

\# variables dans φ

1. $A \leftarrow$ tableau de longueur n, initialisé à -1
2. Tant que NON(TESTAFF(φ, A)):
3. $A \leftarrow$ AFFSUIVANTE(A)
4. Si AFFSUIVANTE a renvoyé « Fin » : Renvoyer « Insatisfiable »
5. Renvoyer A

Sat : algorithme de recherche exhaustive

RechercheExhaustive(φ) :

1. $A \leftarrow$ tableau de longueur n, initialisé à -1
2. Tant que non(TestAff(φ, A)):
3. $\quad A \leftarrow$ AffSuivante(A)
4. $\quad$ Si AffSuivante a renvoyé « Fin » : Renvoyer « Insatisfiable »
5. Renvoyer A

Propriétés

- Correction : conséquence de la correction de TestAff et AffSuivante
- Complexité : nombre d'itérations $\leq 2^n$; coût d'une itération : $O(|\varphi| + n) = O(|\varphi|)$

Théorème

L'algorithme RechercheExhaustive trouve une affectation satisfaisante s'il en existe une, et renvoie « Insatisfiable » sinon, en temps $O(|\varphi| 2^n)$.

Recherche exhaustive

Deux ingrédients

- Parcourir toutes les solutions possibles
- Tester chaque solution

En pratique

- Tester une solution est souvent *facile*
- Parcourir toutes les solutions peut être complexe

Analyse de complexité

$$O\left(\textsc{NombreSolutions} \times (\textsc{CoûtTest} + \textsc{CoûtPassageSuivant})\right)$$

Ensembles de solutions

Les ensembles de solutions ont souvent une structure mathématique à exploiter

- Exemple : affectations $\leftrightarrow$ mots binaires
- Concevoir un algorithme de parcours des solutions demande de :
  - exhiber la structure mathématique
  - trouver une façon de parcourir la structure

Ensembles de solutions

Les ensembles de solutions ont souvent une structure mathématique à exploiter

- Exemple : affectations $\leftrightarrow$ mots binaires
- Concevoir un algorithme de parcours des solutions demande de :
  - exhiber la structure mathématique
  - trouver une façon de parcourir la structure

Quelques exemples de structures

- Mots binaires, mots k-aires
- Suites d'entiers, suites *croissantes* d'entiers
- Sous-ensembles, combinaisons (k *parmi* n), permutations
- Arbres binaires, arbres plus généraux
- …

(ça peut être difficile !)

Problèmes d'efficacité

> La recherche exhaustive est en général exponentielle : soyons efficaces !

Deux façons de produire les solutions

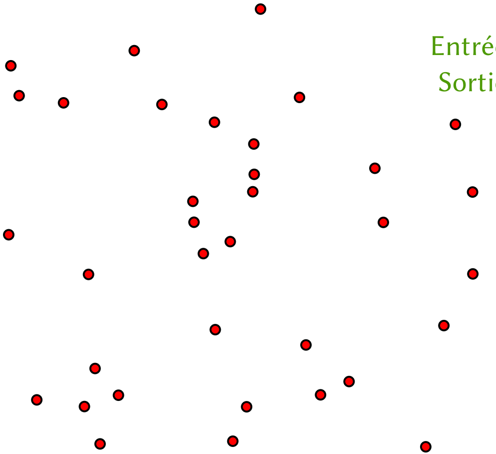
- Algorithme pour passer d'une solution à la suivante
  - situation favorable si algorithme efficace
  - complexité en espace réduite (stockage d'une seule solution)
- Algorithme pour produire la liste de toutes les solutions, puis parcours
  - par exemple *via* un algorithme récursif
  - problèmes de mémoire (ex.: permutations à 12 éléments $\rightarrow > 20$Go)

Passer rapidement d'une solution à la suivante

- Ordre d'énumération des solutions
- Test d'une solution utilisant le test de la précédente

$\rightarrow$ Questions complexes, au delà de ce cours, évoquées en TD

Le voyageur de commerce

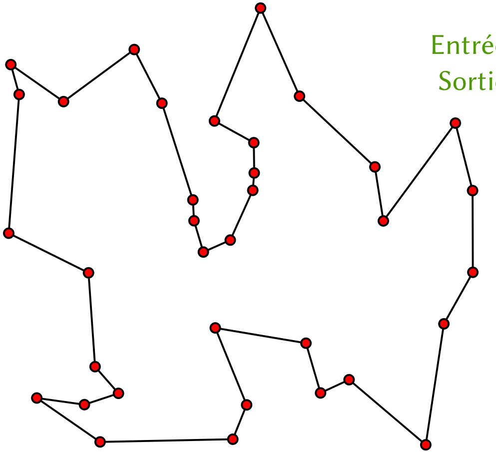


Entrée : Un ensemble de points du plan

Sortie : Un ordre de parcours des points $u_0 \to u_1 \to \cdots \to u_{n-1} \to u_0$ qui minimise la distance totale

Source : Wikipédia

Le voyageur de commerce



Entrée : Un ensemble de points du plan

Sortie : Un ordre de parcours des points $u_0 \to u_1 \to \cdots \to u_{n-1} \to u_0$ qui minimise la distance totale

Source : Wikipédia

Formalisation du problème

Définition

Entrée : Graphe $G = (S, A)$ avec une longueur $\ell(u, v)$ pour chaque arête

Sortie : Une numérotation $u_0, \ldots, u_{n-1}$ des sommets qui minimise la longueur totale $\sum_{k=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$

Formalisation du problème

Définition

Entrée : Graphe $G = (S, A)$ avec une longueur $\ell(u, v)$ pour chaque arête

Sortie : Une numérotation $u_0, \ldots, u_{n-1}$ des sommets qui minimise la longueur totale $\sum_{k=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$

Remarques

- Plus général : $\ell(u, v)$ n'est pas forcément une distance
- Numérotation des sommets = permutation des éléments de S

Algorithme par recherche exhaustive

- Parcours des solutions : permutations d'un ensemble $\rightarrow \{0, \ldots, n-1\}$
- Test d'une solution : calcul de la longueur totale $\rightarrow$ simple boucle

Générer les permutations d'un ensemble

- Comment passer d'une permutation à la suivante ?
- Comment définir « la suivante » ? $\rightarrow$ ordre sur les permutations

Définitions

- Permutation de $\{0, \ldots, n-1\}$: n-uplets d'entiers tous distincts entre 0 et $n-1$
- Ordre lexicographique : $\pi^0 < \pi^1$ s'il existe j tq $\pi^0_{[i]} = \pi^1_{[i]}$ pour $i < j$ et $\pi^0_{[j]} < \pi^1_{[j]}$

Exemple : permutations de $\{0, 1, 2, 3\}$ dans l'ordre lexicographique

0123 $\rightarrow$ 0132 $\rightarrow$ 0213 $\rightarrow$ 0231 $\rightarrow$ 0312 $\rightarrow$ 0321

$\rightarrow$ 1023 $\rightarrow$ 1032 $\rightarrow$ 1203 $\rightarrow$ 1230 $\rightarrow$ 1302 $\rightarrow$ 1320

$\rightarrow$ 2013 $\rightarrow$ 2031 $\rightarrow$ 2103 $\rightarrow$ 2130 $\rightarrow$ 2301 $\rightarrow$ 2310

$\rightarrow$ 3012 $\rightarrow$ 3021 $\rightarrow$ 3102 $\rightarrow$ 3120 $\rightarrow$ 3201 $\rightarrow$ 3210

Permutations : passer à la suivante

Permutation *suivante*

- Exemple : quelle permutation π' après $\pi = 431520$?
- Trois conditions à respecter :
  - π' est une permutation : $\pi \to \pi'$ en échangeant des valeurs
  - $\pi' > \pi$: début de π' égal à π, puis valeur plus grande
  - π' suit π dans l'ordre : début égal à π le plus long possible

$431502 < 431520$

$431 \times$

$432510 > \boxed{432015}$

$4\cancel{35}\ldots$

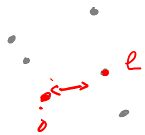
Permutations : passer à la suivante

Permutation *suivante*

- Exemple : quelle permutation π' après $\pi = 431520$?
- Trois conditions à respecter :
  - π' est une permutation : $\pi \to \pi'$ en échangeant des valeurs
  - $\pi' > \pi$: début de π' égal à π, puis valeur plus grande
  - π' suit π dans l'ordre : début égal à π le plus long possible

Idée de l'algorithme

1. Trouver l'indice j tq $\pi_{[j+1]} > \pi_{[j+2]} > \cdots > \pi_{[n-1]}$
   - π est la dernière permutation commençant par $\pi_{[0]}, \ldots, \pi_{[j]}$
   - il faut incrémenter $\pi_{[j]}$, sans toucher $\pi_{[1]}, \ldots, \pi_{[j-1]}$
2. Trouver $\pi_{[\ell]} > \pi_{[j]}$ avec $\ell > j$, minimal, et l'échanger avec $\pi_{[j]}$
   - pour incrémenter $\pi_{[j]}$, on choisit le plus petit élément possible
   - $\ell > j$ car on utilise déjà $\pi_{[1]}, \ldots, \pi_{[j-1]}$
3. *Retourner* la fin $\pi_{[j+1,n[}$
   - avant retournement : $\pi_{[j+1]} > \pi_{[j+2]} > \cdots > \pi_{[n-1]}$
   - ordre lexicographique commence par $\pi_{[j+1]} < \pi_{[j+2]} < \cdots < \pi_{[n-1]}$



Permutations : l'algorithme

PermSuivante(π) :

0. Si $\pi_{[0]} > \cdots > \pi_{[n-1]}$: renvoyer « Fin »
1. Trouver j maximal tel que $\pi_{[j]} < \pi_{[j+1]}$
2. Trouver ℓ maximal tel que $\pi_{[j]} < \pi_{[\ell]}$
3. Échanger $\pi_{[j]}$ et $\pi_{[\ell]}$
4. Retourner $\pi_{[j+1,n[}$: $\pi_{[j+k]} \leftrightarrow \pi_{[n-k]}$ pour $0 < k < \frac{n-j}{2}$
5. Renvoyer π

Propriété

- Complexité $O(n)$:
  - « Trouver j max. tq $\pi_{[j]} < \pi_{[j+1]}$ » : $j \leftarrow n-2$; Tant que $\pi_{[j]} > \pi_{[j+1]}$: $j \leftarrow j-1$
  - « Retourner $\pi_{[j+1,n[}$ » : parcours avec deux indices en sens inverses
- Correction : justifiée précédemment

Retour au voyageur de commerce

sommets du graphe

VoyageurDeCommerce(S, A, ℓ) :

1. $\pi \leftarrow$ tableau de taille n, initialisé à $[0, 1, \ldots, n-1]$
2. $L_{\text{min}} \leftarrow +\infty$; $\pi_{\text{min}} \leftarrow \pi$
3. Répéter :
4. $\quad L \leftarrow \sum_{i=0}^{n-1} \ell(S[\pi_{[i]}], S[\pi_{[i+1 \bmod n]}])$
5. $\quad$ Si $L < L_{\text{min}}$: $(L_{\text{min}}, \pi_{\text{min}}) \leftarrow (L, \pi)$
6. $\quad \pi \leftarrow$ PermSuivante(π)
7. $\quad$ Si PermSuivante a renvoyé « Fin » : renvoyer π_{min}

Calcul de L

- Écriture simplifiée, supposant $\ell(u, v) = +\infty$ si $uv \notin A$
- Algorithmiquement : boucle sur i, avec test $uv \in A$

- Calcul de L
- PERM SUIVANTE

Propriétés

permutations

- Complexité $O(n \times n!)$
- Correction : déduite de celle de PermSuivante

Conclusion sur la recherche exhaustive

Atouts

- Technique algorithmique conceptuellement simple : on teste toutes les possibilités
- Analyse de complexité simple : essentiellement le nombre de solutions
- Parfois le mieux qu'on sache faire !
- Point de départ d'algorithmes plus sophistiqués (*backtrack*, …)

Limites

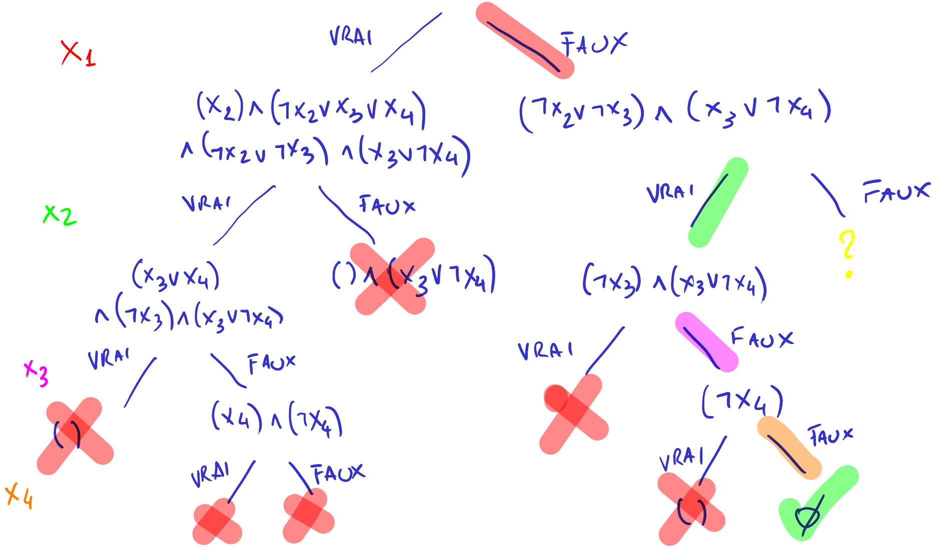
- Solution algorithmiquement coûteuse (quasiment toujours exponentiel)
- Écriture en détail et implantations parfois difficiles
- Problèmes éventuels de mémoire

Pour aller plus loin

- Techniques d'*élagage* de l'ensemble des solutions (dont *backtrack*)
- Optimisation du passage d'une solution à la suivante

Sur un exemple

$$\varphi(x_1, x_2, x_3, x_4) = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_4)$$



x_1 = FAUX
x_2 = VRAI
x_3 = FAUX
x_4 = FAUX

Principe de l'algorithme

Algorithme récursif

- *Élimination* des variables une à une
- $\Leftrightarrow$ construction pas à pas d'une affectation
- Cas de bases :
  - Clause vide insatisfiable $\rightarrow$ formule insatisfiable
  - Formule vide satisfiable

Élimination d'une variable

- Formule ψ obtenue à partir de φ en posant $x_i = \textsc{vrai}$ ou $x_i = \textsc{faux}$
- Exemple $x_i = \textsc{vrai}$:
  - Si une clause contient $x_i \rightarrow$ suppression de la clause car satisfaite
  - Si une clause contient $\neg x_i \rightarrow$ suppression du littéral, maintien de la clause
- Cas $x_i = \textsc{faux}$ symétrique

Remarque

- Élimination de x_n en premier : $\psi(x_1, \ldots, x_{n-1}) = \varphi(x_1, \ldots, x_{n-1}, b)$

Élimination d'une variable

Élimination(φ, n, b) :

- $\psi \leftarrow$ formule vide
- Pour C dans φ :
  - $C' \leftarrow$ clause vide
  - SAT $\leftarrow$ FAUX
  - Pour ℓ dans C:
    - Si $|\ell| = n$ et $\ell \times b > 0$:
      - SAT $\leftarrow$ VRAI
    - Sinon si $|\ell| \neq n$:
      - Ajouter ℓ à C'
  - Si NON(SAT) : ajouter C' à ψ
- Renvoyer ψ

Propriétés

- ψ a une affectation satisfaisante $(b_1, \ldots, b_{n-1})$
  $\Leftrightarrow$ φ est satisfaite par $(b_1, \ldots, b_{n-1}, b)$
- L'algorithme a une complexité $O(|\varphi|)$

Algorithme de *backtrack* pour SAT

SATBACKTRACK(φ, n) :

```
Si φ est vide :
    Renvoyer A = [1, . . . , 1]      taille n
Si φ possède une clause vide :
    Renvoyer « Insatisfiable »
Pour b ∈ {1, −1} :
    ψ ← ÉLIMINATION(φ, n, b)
    A ← SATBACKTRACK(ψ, n − 1)
    Si ψ n'est pas insatisfiable : Renvoyer A + [b]
Renvoyer « Insatisfiable »
```

Propriétés

- Complexité : $T(n) \leq 2T(n-1) + O(|\varphi|) \rightarrow T(n) = O(2^n\,|\varphi|)$
- Correction : par récurrence

Preuve de correction de SatBacktrack

<u>Thm</u> SatBacktrack renvoie une affectation satisfaisante s'il en existe une, et "insatisfiable" sinon

- <u>$n=0$</u> : φ est soit vide, soit constituée de clauses vides. ✓
- <u>$n>0$</u> : On suppose que SatBacktrack est correct pour les formules à $n-1$ variables. Donc le résultat de SatBacktrack$(\varphi, n-1)$ est correct (pour $b=1$ et pour $b=-1$). La correction de l'algo Elimination permet de conclure :

φ est satisfiable $\Leftrightarrow$ ψ est satisfiable $\Leftrightarrow$ SatBacktrack$(\psi, n-1)$ renvoie une affectation

↑ Elimination ↑ Hyp. de récurrence.

$\Rightarrow$ SatBacktrack est correct pour les formules à n variables. □

Bilan sur SatBacktrack

L'algorithme SatBacktrack trouve une affectation satisfaisante de φ s'il en existe une, et renvoie « Insatisfiable » sinon, en temps $O(|\varphi|\, 2^n)$.

Bilan sur SatBacktrack

L'algorithme SatBacktrack trouve une affectation satisfaisante de φ s'il en existe une, et renvoie « Insatisfiable » sinon, en temps $O(|\varphi|\, 2^n)$.

- Énoncé *strictement identique* pour RechercheExhaustive !

Lequel des deux algorithmes est meilleur ?

- Dans le pire cas, aucun des deux
- Mais dans des cas favorables, SatBacktrack peut être *beaucoup* plus rapide
  - Arbre des solutions très peu exploré
- Cas les pires pour SatBacktrack
  - Arbre à explorer (quasiment) en entier
  - RechercheExhaustive peut alors être *légèrement* plus rapide

Conclusion

L'algorithme SatBacktrack est efficace *en pratique*

Qu'est-ce qu'un algorithme de *backtrack* ?

Le *backtrack* est de la recherche exhaustive récursive

Comparaison avec la recherche exhaustive

- Recherche exhaustive :
  - parcours itératif des solutions possibles
  - test de chaque solution
- *Backtrack* :
  - construction récursive progressive des solutions
  - test de solutions partielles

Remarque

- En recherche exhaustive, on construit parfois *récursivement* les solutions
  $\rightarrow$ différence majeure : test de solutions *partielles*

Fonctionnement général

Arbres des solutions

- Exemple : affectations comme arbre binaire de hauteur n
- Cas (presque) général : solutions décrites comme vecteurs
  - chaque composante du vecteur peut prendre k valeurs $\rightarrow$ arbre k-aire
  - vecteur de longueur $n \rightarrow$ arbre de hauteur n
- Plus général : structure d'arbre parfois complexe à voir
- L'arbre n'est *pas* représenté en mémoire $\rightarrow$ arbre implicite

> L'algorithme récursif de *backtrack* est un parcours en profondeur de l'arbre des solutions

Parcours partiel

- Exemple : affectation partielle non satisfaisante $\rightarrow$ retour en arrière
- *Élagage* de l'arbre :
  - si solution partielle incorrecte : pas besoin de continuer
  - *branches* non explorées de l'arbre
- Algorithme de test de solutions partielles

Caractéristiques

Analyse de complexité

- Algorithme récursif $\rightarrow$ équation de récurrence pour la complexité
- Arbre des solutions $\rightarrow$ complexité proportionnelle au nombre de solutions

Pourquoi *backtrack* ou « retour sur trace » ?

- Construction d'une solution pas à pas…
- … et retour sur nos pas si échec
- Géré par les appels récursifs

Remarque

- Généralisation de la recherche exhaustive
- Si pas de test de solutions partielles : recherche exhaustive

Une grille de Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Une grille de Sudoku remplie !

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Formalisation

Sudoku

Entrée : Une grille G de dimensions $n^2 \times n^2$, remplis d'entiers de 0 (= vide) à n^2

Sortie : La même grille G sans 0, tel que $G_{[i,j]} \neq G_{[k,\ell]}$ dès que :

- $i = k$ (ligne), ou
- $j = \ell$ (colonne), ou
- $\lfloor i/3 \rfloor = \lfloor k/3 \rfloor$ et $\lfloor j/3 \rfloor = \lfloor \ell/3 \rfloor$ (zone)

Ou « aucune solution »

0	1	2
3	4	5
6	7	8

0	1	2	3	4	5	6	7	8

Représentation *linéaire* de la grille

- Grille G de dimensions $n^2 \times n^2 \to$ tableau T de taille n^4
- Case $G_{[i,j]} \to$ case $T_{[in^2+j]}$ (stockage *ligne par ligne*)
- Case $T_{[u]} \to$ case $G_{[u \operatorname{div} n^2, u \bmod n^2]}$ où $u \operatorname{div} n^2 = \lfloor u/n^2 \rfloor$
- Représentation *pratique* pour parcourir toutes les solutions

Plan de bataille

Algorithme récursif

- Pour chaque case initialement vide :
  - Essayer toutes les solutions
  - Vérifier qu'on ne crée aucune incohérence
- Représentation linéaire : « case suivante » évidente

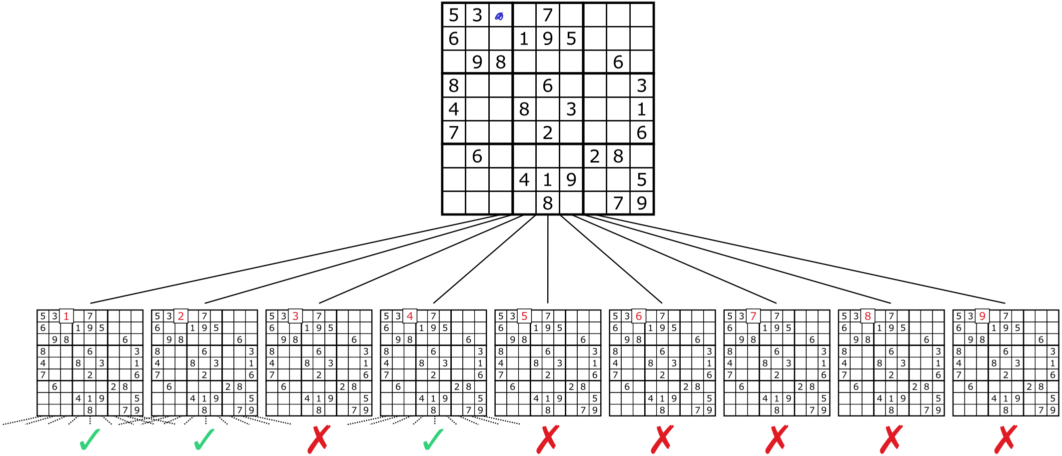
Test de solutions partielles

- Tester si une grille (partielle) est incohérente
- ~~Test des n^2 lignes, n^2 colonnes et n^2 *zones*~~
- Uniquement la nouvelle case $\rightarrow$ une ligne, une colonne, une zone

Remarque

- Complexité : au pire n^4 cases à remplir avec n^2 valeurs possibles
  $\rightarrow (n^2)^{n^4} = n^{2n^4}$ possibilités !
- Mais cases déjà remplies, conflits rapidement obtenus
  $\rightarrow$ bien plus rapide en pratique (on espère !)

Arbre des solutions



Test de solution partielle

$\textsc{Valide}(G, n, u, x)$:

$(i, j) \leftarrow (u \operatorname{div} n^2, u \overset{\text{mod}}{\cancel{\operatorname{div}}} n^2)$
Pour $k = 0$ à $n^2 - 1$:
    Si $k \neq i$ ET $G_{[kn^2+j]} = x$: Renvoyer FAUX
    Si $k \neq j$ ET $G_{[in^2+k]} = x$: Renvoyer FAUX
$(z_i, z_j) \leftarrow (3\lfloor i/3 \rfloor, 3\lfloor j/3 \rfloor)$
Pour $k = 0$ à $n - 1$:
    Pour $\ell = 0$ à $n - 1$:
        Si $(z_i + k)n^2 + (z_j + \ell) \neq u$ ET $G_{[(z_i+k)n^2+(z_j+\ell)]} = x$: Renvoyer FAUX
Renvoyer VRAI

Propriétés

- $\textsc{Valide}$ renvoie VRAI si et seulement si $G_{[u]} \leftarrow x$ ne crée pas de conflit
- Sa complexité est $O(n^2)$ — taille de l'entrée : $O(n^4)$

Algorithme de *backtrack* pour le Sudoku

SudokuBacktrack(G, n, u) :

1. Tant que $u < n^4$ ET $G_{[u]} \neq 0$: $u \leftarrow u + 1$
2. Si $u = n^4$: renvoyer VRAI
3. Pour x de 1 à n^2 :
4. Si Valide(G, n, u, x) :
5. $G_{[u]} \leftarrow x$
6. Si SudokuBacktrack$(G, n, u+1)$: renvoyer VRAI
7. $G_{[u]} \leftarrow 0$
8. Renvoyer FAUX

Propriétés

- L'algorithme renvoie VRAI et remplit correctement G s'il existe une solution
- L'algorithme renvoie FAUX et laisse G non modifiée sinon
- Sa complexité vérifie : $T(m) \leq n^2 \cdot T(m-1) + O(n^2)$ où m est le nombre cases vides
  $\rightarrow T(m) = O(n^{2m})$ $O(n^{2n^4})$ si $m \simeq n^4$

Conclusion

Deux techniques proches

- Recherche exhaustive et *backtrack* sont deux faces d'une même pièce
- Algorithme souvent itératif pour la recherche exhaustive (mais parfois récursif)
- Algorithme quasiment toujours récursif pour le *backtrack*
- Principale différence : test de solutions partielles pour le *backtrack*

Principales difficultés

- Produire toutes les solutions possibles
  - Itérativement : ordre sur les solutions et parcours
  - Récursivement : solution à partir d'une « solution incomplète »
- Pour le *backtrack* : peut-on tester une solution partielle ?

Pour aller plus loin

Branch-and-bound

- Problèmes d'optimisation :
  - Objectif : trouver la solution de plus grande valeur (ou plus petite valeur)
- Idées :
  - À chaque nœud de l'arbre, borner les valeurs des solutions *en dessous*
  - Ne pas explorer les branches dont les solutions seront moins bonnes

Algorithmes d'intelligence artificielle

- *Backtrack* et *branch-and-bound* sont considérés comme techniques d'IA
- Plus généralement :
  - Autres idées pour explorer *rapidement* l'arbre des solutions
  - Éventuellement : accepter une solution qui n'est pas la meilleure $\rightarrow$ algorithmes heuristiques