Algorithmique — 2. Techniques algorithmiques

6. Recherche exhaustive

Bruno Grenet

https://membres-ljk.imag.fr/Bruno.Grenet/Algorithmique.html

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Table des matières

Table des matières

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

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

Représentation d'une affectation

- Tableau de booléens — [FAUX, VRAI, FAUX]
- Plus pratique : tableau $\pm 1 \to$ VRAI $= 1$; FAUX $= -1$ — $[-1, 1, -1]$

SAT : tester une affectation

Idée de l'algorithme

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

SATISFAIT(φ, A) :

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

Complexité

Linéaire en la taille de 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* : $1 \mapsto 1$, $-1 \mapsto 0$
- Analogie : parcourir les affectations $\Leftrightarrow$ compter de 0 à $2^n - 1$
- Opération nécessaire : AffSuivante $\Leftrightarrow$ incrémenter un compteur binaire

AffSuivante(A) :

$i \leftarrow 0$
Tant que $i < n$ et $A_{[i]} = 1$:
    $A_{[i]} \leftarrow -1$
    $i \leftarrow i + 1$
Si $i = n$: renvoyer « Fin »
$A_{[i]} \leftarrow 1$
Renvoyer A

Propriétés

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

$[1, 1, -1, 1, -1, 1, 1, 1]$

$[-1, -1, 1, 1, -1, 1, 1, 1]$

11101011

11101100

SAT : algorithme de recherche exhaustive

RECHERCHEEXHAUSTIVE(φ) :

1. $A \leftarrow$ tableau de longueur n, initialisé à -1
2. Tant que NON(SATISFAIT(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 SATISFAIT et AFFSUIVANTE

Complexité : nombre d'itérations $\leq 2^n$; coût d'une itération : $O(|\varphi| + n) = O(|\varphi|)$

Remarque : permet d'obtenir toutes les affectations satisfaisantes, ou leur nombre, …

Théorème

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

Table des matières

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

Objectifs

- Trouver une solution correcte parmi un ensemble de solutions possibles
- Compter le nombre de solutions, trouver toutes les solutions
- Trouver la meilleure solution, la plus petite, au contraire la plus grande, …

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

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 !)

L'exemple de base : n-uplets d'entiers entre 0 et $k-1$

Interprétation

- Entiers écrits en base k
  - de $(0,\ldots,0)$ à $(k-1,\ldots,k-1) \to k^n$ valeurs
  - passage au suivant : *incrément* d'un compteur k-aire

UpletSuivant(U, k) :

```
n ← #U
i ← 0
Tant que i < n et U[i] = k − 1 :
    U[i] ← 0
    i ← i + 1
Si i = n : renvoyer « Fin »
U[i] ← U[i] + 1
Renvoyer U
```

Complexité

- $O(n)$ dans le pire cas
- $O(1)$ amortie

i

$(9, 8, 9, 1, 0)$

$\hookrightarrow (0, 9, 9, 1, 0)$

$(9, 9, 9, 4)$

$\hookrightarrow (0, 0, 0, 5)$

01989

01990

Applications immédiates

Mots de longueur n sur un alphabet Σ à s lettres

- Numérotation des lettres de Σ de 0 à $s-1$ *bijection* $\Sigma \simeq \{0, \ldots, s-1\}$
- Mot de longueur n sur Σ = n-uplet d'entiers entre 0 et $s-1$

$S = \{a, b, c\}$ avec $a \mapsto 0$, $b \mapsto 1$, $c \mapsto 2$

000 → $\emptyset$
001 → $\{c\}$
010 → $\{b\}$
011 → $\{b,c\}$
100 → $\{a\}$
101 → $\{a,c\}$
110 → $\{a,b\}$
111 → $\{a,b,c\}$

Sous-ensembles d'un ensemble fini S

- Numérotation des éléments de S de 0 à $\#S - 1$
- Sous-ensemble = mot binaire de longueur $\#S$
- Parcours des sous-ensembles = parcours des mots binaires de longueur $\#S$

Uplets de longueur $\leq n$ d'entiers entre 0 et $k-1$

- Parcours des t-uplets, pour $t = 0$ à n
- Nombre d'éléments : $\sum_{t=0}^{n} k^t = (k^{n+1}-1)/(k-1)$

$$\frac{k^{\#\text{termes}} - k^{\text{indice init}}}{k-1}$$

En base k : $k^t \rightsquigarrow 1\underbrace{0\cdots0}_{t}$ donc $\sum_{t=0}^{n} k^t \rightsquigarrow \underbrace{1\cdots\cdots1}_{n+1}$

$(k-1)\sum_{t=0}^{n} k^t \rightsquigarrow (k-1)(k-1)\cdots(k-1)$ donc $(k-1)\sum_{t=0}^{n} k^t + 1 \rightsquigarrow 1\underbrace{0\cdots0}_{n+1} \rightsquigarrow k^{n+1}$

$$\sum_{t=0}^{n} k^t = \frac{(k^{n+1}-1)}{k-1}$$

Application non-immédiate : modification de l'algorithme

n-uplets *décroissants* d'entiers entre 0 et $k-1$

- Idée : partir de UpletSuivant et ne garder que certains n-uplets

UpletDécroissantSuivant(U, k) :

```
n ← #U
i ← 0
Tant que i < n et U[i] = k − 1 :
    i ← i + 1
Si i = n : renvoyer « Fin »
U[i] ← U[i] + 1
Pour j = 0 à i − 1 :
    U[j] ← U[i]
Renvoyer U
```

Théorème

UpletDécroissantSuivant parcourt les $\binom{n+k-1}{n}$ n-uplets décroissants

$(9, 9, 7, 5, 2) \to (0, 0, 8, 5, 2)$

$(8, 8, 8, 5, 2)$

$(1, 0, 8, 5, 2)$

$(9, 0, 8, 5, 2)$

$(8, 1, 8, 5, 2) \leftarrow (0, 1, 8, 5, 2)$

Application non-immédiate : modification de l'algorithme

n-uplets *décroissants* d'entiers entre 0 et $k-1$

- Idée : partir de UpletSuivant et ne garder que certains n-uplets

UpletDécroissantSuivant(U, k) :

$n \leftarrow \#U$
$i \leftarrow 0$
Tant que $i < n$ et $U_{[i]} = k-1$:
  $i \leftarrow i+1$
Si $i = n$: renvoyer « Fin »
$U_{[i]} \leftarrow U_{[i]} + 1$
Pour $j = 0$ à $i-1$:
  $U_{[j]} \leftarrow U_{[i]}$
Renvoyer U

Théorème

UpletDécroissantSuivant parcourt les $\binom{n+k-1}{n}$ n-uplets décroissants

$k=7$ $(5,2,1)$ • | • • • | • | •

$(6,3,3)$ | • • • | | • • •

$(4,2,0)$ • • | • • | • • |

- Dessin : $k-1$ points + n barres $\rightsquigarrow$ $k+n-1$ objets

$\#$ dessins = $\#$ positions possibles des n barres = $\binom{k+n-1}{n}$

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 $\rightarrow >$ 20Go)

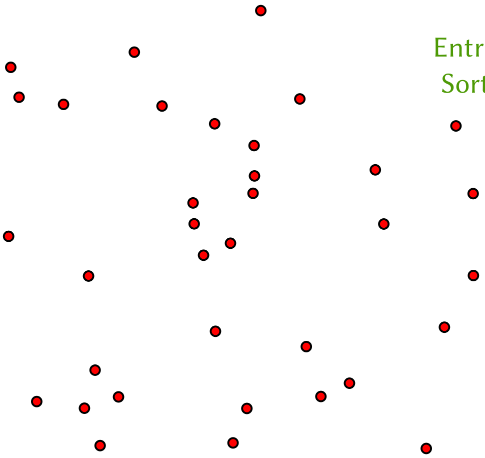
Passer rapidement d'une solution à la suivante

- Ordre optimisé d'énumération des solutions
- Réutilisation du test d'une solution pour la suivante

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

Table des matières

Le voyageur de commerce

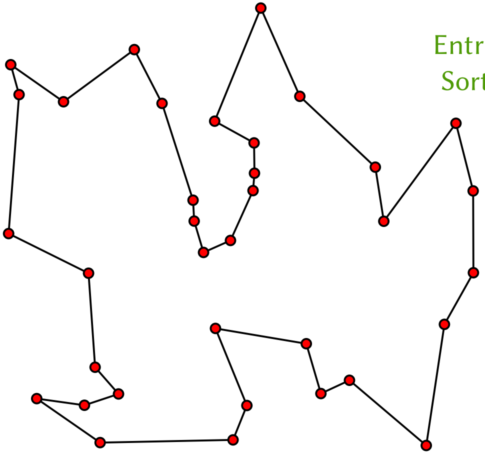


Entrée : Un ensemble de points du plan

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 : Wikimedia commons (public domain)

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 : Wikimedia commons (public domain)

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 $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$

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 $\to \{0, \ldots, n-1\}$
- Test d'une solution : calcul de la longueur totale $\to$ simple boucle

Générer les permutations d'un ensemble

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

Définitions

- Permutation de $\{0, \dots, n-1\}$: 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 $\longrightarrow$ 0132 $\longrightarrow$ 0213 $\longrightarrow$ 0231 $\longrightarrow$ 0312 $\longrightarrow$ 0321

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

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

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

Permutations : passer à la suivante

Trois conditions à respecter

1. π' est une permutation : $\pi \to \pi'$ en échangeant des valeurs
2. $\pi' > \pi$: début de π' égal à π, puis valeur plus grande
3. π' suit π dans l'ordre : début égal à π le plus long possible

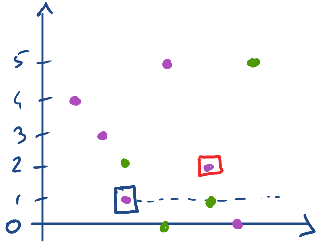
Exemple : permutation suivant $\pi = 431520$

4 3 1 5 2 0 → impossible

4 3 1 5 2 0 → permutation + petite

4 3 1 5 2 0 → ______

4 3 1 5 2 0 → 4 3 2 0 1 5



Permutations : principe de l'algorithme

Idée de l'algorithme

1. Trouver l'indice maximal j tq $\pi_{[j]} < \pi_{[j+1]}$
   - j est l'indice *le plus à droite* qu'on peut incrémenter
2. Échanger $\pi_{[j]}$ avec le plus petit $\pi_{[\ell]} > \pi_{[j]}$ pour $\ell > j$
   - ne pas toucher à $\pi_{[0]}, \ldots, \pi_{[j-1]}$
   - incrément de $\pi_{[j]}$ le plus petit possible
3. *Retourner* la fin $\pi_{[j+1,n[}$
   - avant retournement : $\pi_{[j+1]} > \pi_{[j+2]} > \cdots > \pi_{[n-1]}$
   - ordre lexicographique commence par $\pi_{[j+1]} < \pi_{[j+2]} < \cdots < \pi_{[n-1]}$

Exemple : permutation suivant $\pi = 431520$

$$4\,3\,\boxed{1}\,5\,\boxed{2}\,0 \to 4\,3\,2\,\underbrace{5\,1\,0} \to 4\,3\,2\,0\,1\,5$$

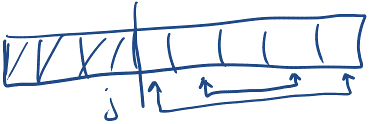
Permutations : l'algorithme

PermSuivante(π) :

0. Si $\pi_{[0]} > \cdots > \pi_{[n-1]}$: renvoyer « Fin »
1. $j \leftarrow n-2$; $\ell \leftarrow n-1$
2. Tant que $\pi_{[j]} > \pi_{[j+1]}$: $j \leftarrow j-1$
3. Tant que $\pi_{[\ell]} < \pi_{[j]}$: $\ell \leftarrow \ell - 1$
4. $\pi_{[j]} \leftrightarrow \pi_{[\ell]}$
5. Pour $k = 1$ à $\lfloor \frac{n-1-j}{2} \rfloor$: $\pi_{[j+k]} \leftrightarrow \pi_{[n-k]}$
6. Renvoyer π

j max tq $\pi_{[j]} < \pi_{[j+1]}$

ℓ max tq $\pi_{[\ell]} > \pi_{[j]}$



Lemme

La complexité de PermSuivante est $O(n)$

Preuve

0. $\Theta(n)$

2. 3. $\Theta(n)$

5. $\Theta(n)$

Correction de PermSuivante

Lemme

Si π est une permutation, PermSuivante(π) renvoie la permutation suivant π dans l'ordre lexicographique (ou « Fin »)

Preuve π^0 la permutation avant et $\pi^1 = $ PermSuivante (π^0)

(0) π^1 est une permutation : OK car on échange des valeurs

(1) $\pi^1 > \pi^0$: $\pi^0_{[0,j[} = \pi^1_{[0,j[}$ et $\pi^1_{[j]} > \pi^0_{[j]}$ donc OK.

(2) π^1 suit π^0 :

- π^0 est la plus grande perm. qui commence par $\pi^0_{[0,j]}$ car $\pi^0_{[j+1,n]} \searrow$
- π^1 —— petite —————— $\pi^1_{[0,j]}$ car $\pi^1_{[j+1,n]} \nearrow$
- Pour être entre π^0 et π^1, il faudrait commencer par $\pi^0_{[0,j[} = \pi^1_{[0,j[}$ puis avoir soit $\pi^0_{[j]}$ soit $\pi^1_{[j]}$ en case j.

Retour au voyageur de commerce

Formalisation du graphe

- G : matrice $n \times n \to G_{[i,j]} =$ longueur entre sommets i et j (symétrique)
- Remarque : si graphe non complet $\to G_{[i,j]} = +\infty$ si pas d'arête entre i et j

VoyageurDeCommerce(G) :

1. $\pi \leftarrow$ tableau de taille n, initialisé à $[0, 1, \dots, n-1]$
2. $L_{\text{min}} \leftarrow +\infty$; $\pi_{\text{min}} \leftarrow \pi$
3. Répéter :
4. $\quad L \leftarrow G[\pi_{[n-1]}, \pi_{[0]}] + \sum_{i=0}^{n-2} G[\pi_{[i]}, \pi_{[i+1]}]$
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}

Propriétés

- 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
  - *Algorithmes d'énumération*

Pour aller plus loin

Backtrack et *Branch-and-bound*

- Parcours *récursif* des solutions, avec test de *solutions partielles* $\rightarrow$ *arbre des solutions*
  - Si la solution partielle est *contradictoire* $\rightarrow$ pas d'exploration des sous-arbres
- Ex. des problèmes de minimisation *(trouver la plus petite solution)*
  - *Backtrack* : arrêt si la solution partielle est déjà trop grande
  - *Branch-and-bound* : arrêt si la solution partielle ne peut mener qu'à une solution trop grande

Exemple de SAT

- Parcours récursif : pour chaque x_i, essayer $x_i = \text{VRAI}$ et $x_i = \text{FAUX}$
- *Backtrack* : est-ce qu'il existe une clause (déjà) insatisfaite ?

Exemple du Voyageur de Commerce

- Parcours récursif : $(n-k)$ possibilités pour $\pi_{[k]}$, si $\pi_{[0]}, \dots, \pi_{[k-1]}$ fixés
- *Backtrack* : si $\sum_{i=0}^{k-1} G[\pi_{[i]}, \pi_{[i+1]}] > L_{\text{min}} \rightarrow$ stop
- *Branch-and-bound* : si longueur partielle + coût du retour $> L_{\text{min}} \rightarrow$ stop