

- HAI503I: Algorithmique 4 -

Chap. 2 – Recherche exhaustive et backtrack

L3 informatique
Université de Montpellier

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 Principes du backtrack
- 2.3 Exemple 2 : le Sudoku

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 Principes du backtrack
- 2.3 Exemple 2 : le Sudoku

1. Recherche exhaustive

1.1 L'odomètre ou 'compteur kilométrique'

1.2 Exemple 1 : SAT

1.3 Principes de la recherche exhaustive

1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

2.1 Exemple 1 : le retour de SAT

2.2 Principes du backtrack

2.3 Exemple 2 : le Sudoku

Un outils de base : l'odomètre

Un compteur binaire

- ▶ Parcourt tous les mots de n lettres formés de '0' et de '1'.
- ▶ Le compteur part de 0^n , va jusqu'à 1^n dans '*l'ordre croissant*'.

Un outils de base : l'odomètre

Un compteur binaire

- ▶ Parcourt tous les mots de n lettres formés de '0' et de '1'.
- ▶ Le compteur part de 0^n , va jusqu'à 1^n dans '*l'ordre croissant*'.

MOTSUIVANT(T) :

1. $i \leftarrow n - 1$
2. Tant que $i \geq 0$ et $T[i] = 1$:
3. $T[i] \leftarrow 0$
4. $i \leftarrow i - 1$
5. Si $i = -1$: renvoyer « Fin »
6. $T[i] \leftarrow 1$
7. Renvoyer T

Exemple

- ▶ $n = 4$, $T = [0, 0, 0, 0]$ au départ
- ▶ $\text{MOTSUIVANT}(T) = [0, 0, 0, 1]$
- ▶ $\text{MOTSUIVANT}(T) = [0, 0, 1, 0]$
- ▶ $\text{MOTSUIVANT}(T) = [0, 0, 1, 1]$
- ▶ $\text{MOTSUIVANT}(T) = [0, 1, 0, 0]$
- ▶ ...
- ▶ En tout 2^n appels.

Un outils de base : l'odomètre

Un compteur binaire

- ▶ Parcourt tous les mots de n lettres formés de '0' et de '1'.
- ▶ Le compteur part de 0^n , va jusqu'à 1^n dans '*l'ordre croissant*'.

MOTSUIVANT(T) :

1. $i \leftarrow n - 1$
2. Tant que $i \geq 0$ et $T[i] = 1$:
3. $T[i] \leftarrow 0$
4. $i \leftarrow i - 1$
5. Si $i = -1$: renvoyer « Fin »
6. $T[i] \leftarrow 1$
7. Renvoyer T

Remarques :

- ▶ Nombreuses variantes : parcours ↘, incrément par la gauche, autre base...
- ▶ Sert, en particulier, à **parcourir tous les sous-ensembles d'un ensemble**, par la bijection entre $\mathcal{P}(X)$ et $\{0, 1\}^n$ (pour $|X| = n$).
- ▶ Complexité en $O(n)$, on verra une complexité '*amortie*' en $O(1)$.

1. Recherche exhaustive

1.1 L'odomètre ou 'compteur kilométrique'

1.2 Exemple 1 : SAT

1.3 Principes de la recherche exhaustive

1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

2.1 Exemple 1 : le retour de SAT

2.2 Principes du backtrack

2.3 Exemple 2 : le Sudoku

Le problème SAT

Formule logique : *conjonction de disjonction de littéraux*

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

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

Le problème SAT

Formule logique : *conjonction de disjonction de littéraux*

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

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

Définition du problème SAT

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

Sortie : Une affectation satisfaisant φ ; insatisfiable sinon

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

Le problème SAT

Formule logique : *conjonction de disjonction de littéraux*

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

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

Définition du problème SAT

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

Sortie : Une affectation satisfaisant φ ; insatisfiable sinon

Problème fondamental en informatique théorique, mais aussi d'un point de vue pratique...

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 ?

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 SAT

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

Représentation de φ : tableau de tableaux d'entiers

Exemple

$$\begin{aligned}\varphi(x_1, x_2, x_3) &= (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2 \rightsquigarrow \\ \varphi &= [[-1, 2], [1, 2, -3], [-2]]\end{aligned}$$

SAT : représentation informatique

Représentation d'une formule SAT

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

Représentation de φ : tableau de tableaux d'entiers

Exemple

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

Représentation d'une affectation

- ▶ Tableau de booléens : $(\text{FAUX}, \text{VRAI}, \text{FAUX}) \rightarrow A = [\text{FALSE}, \text{TRUE}, \text{FALSE}]$
- ▶ Plus pratique ici : tableau $\pm 1 \rightarrow \text{VRAI} = 1$; $\text{FAUX} = -1$
 $(\text{FAUX}, \text{VRAI}, \text{FAUX}) \rightarrow A = [-1, 1, -1]$

SAT : tester une affectation

Idée de l'algorithme

- ▶ Parcourir toutes les clauses \rightarrow elles doivent toutes être satisfaites
- ▶ Clause $C = \varphi[i]$ satisfaite : (au moins) un littéral est satisfait
- ▶ Littéral $\ell = \varphi[i][j]$ satisfait :
 - ▶ Littéral non nié : $\ell > 0$ satisfait si $A_{[\ell-1]} = 1$
 - ▶ Littéral nié : $\ell < 0$ satisfait si $A_{[-\ell-1]} = -1$

SAT : tester une affectation

Idée de l'algorithme

- ▶ Parcourir toutes les clauses \rightarrow elles doivent toutes être satisfaites
- ▶ Clause $C = \varphi[i]$ satisfaite : (au moins) un littéral est satisfait
- ▶ Littéral $\ell = \varphi[i][j]$ satisfait :
 - ▶ Littéral non nié : $\ell > 0$ satisfait si $A_{[\ell-1]} = 1$
 - ▶ Littéral nié : $\ell < 0$ satisfait si $A_{[-\ell-1]} = -1$

TESTAFF(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 OK = FAUX : 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* : $1 \rightarrow 1$; $-1 \rightarrow 0$
- ▶ 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* : $1 \rightarrow 1$; $-1 \rightarrow 0$
- ▶ Analogie : parcourir les affectations \Leftrightarrow compter de 0 à $2^n - 1$
- ▶ Opération nécessaire : AFFSUIVANTE \Leftrightarrow incrémenter un compteur binaire

AFFSUIVANTE(A) :

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

Propriétés

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

SAT : algorithme de recherche exhaustive

RECHERCHEEXHAUSTIVESAT(φ) :

1. $A \leftarrow$ tableau de longueur n (nb de variables dans φ), 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

RECHERCHEEXHAUSTIVESAT(φ) :

1. $A \leftarrow$ tableau de longueur n (nb de variables dans φ), 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

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

1. Recherche exhaustive

1.1 L'odomètre ou 'compteur kilométrique'

1.2 Exemple 1 : SAT

1.3 Principes de la recherche exhaustive

1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

2.1 Exemple 1 : le retour de SAT

2.2 Principes du backtrack

2.3 Exemple 2 : le Sudoku

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

$$O(\text{NOMBRESOLUTIONS} \times (\text{COÛTEST} + \text{COÛTPASSAGESUIVANT}))$$

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 $\rightarrow > 20\text{Go}$)

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 $\rightarrow > 20\text{Go}$)

Passer rapidement d'une solution à la suivante

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

↪ Questions complexes, au delà de ce cours, évoquées en TD

1. Recherche exhaustive

1.1 L'odomètre ou 'compteur kilométrique'

1.2 Exemple 1 : SAT

1.3 Principes de la recherche exhaustive

1.4 Exemple 2 : le voyageur de commerce (euclidien)

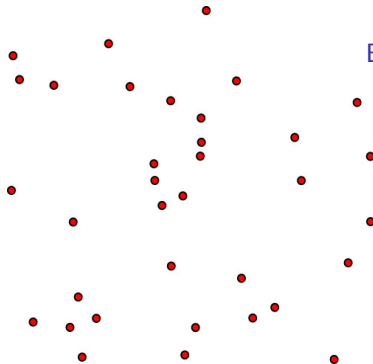
2. *Backtrack* ou retour sur trace

2.1 Exemple 1 : le retour de SAT

2.2 Principes du backtrack

2.3 Exemple 2 : le Sudoku

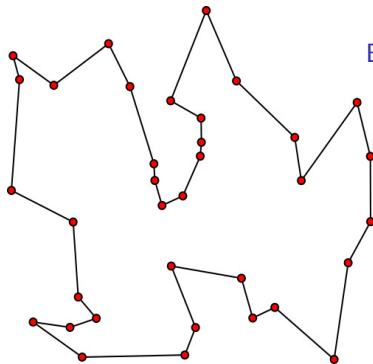
Le voyageur de commerce



Entrée : Un ensemble de points du plan U

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

Le voyageur de commerce



Entrée : Un ensemble de points du plan U

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

Formalisation du problème

Définition : VdC (euclidien)

Entrée : Un ensemble de n points du plan U

Sortie : Une numérotation u_0, \dots, u_{n-1} des points qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$, où $\ell(-, -)$ désigne la distance euclidienne usuelle

Formalisation du problème

Définition : VdC (euclidien)

Entrée : Un ensemble de n points du plan U

Sortie : Une numérotation u_0, \dots, u_{n-1} des points qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$, où $\ell(-, -)$ désigne la distance euclidienne usuelle

Remarques

- ▶ Plus général : on peut se placer sur un graphe, une autre structure discrète, un autre espace métrique,...
- ▶ Numérotation des points = permutation des éléments de U

Formalisation du problème

Définition : VdC (euclidien)

Entrée : Un ensemble de n points du plan U

Sortie : Une numérotation u_0, \dots, u_{n-1} des points qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$, où $\ell(-, -)$ désigne la distance euclidienne usuelle

Remarques

- ▶ Plus général : on peut se placer sur un graphe, une autre structure discrète, un autre espace métrique,...
- ▶ Numérotation des points = permutation des éléments de U

Permutation

Si $U = \{u_0, \dots, u_{n-1}\}$ alors $(u_{\pi(0)}, \dots, u_{\pi(n-1)})$ est une permutation de U avec $\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ bijective.

Ex : $n = 5$ et $\pi = [3, 4, 0, 2, 1] \rightsquigarrow (u_{\pi(0)}, \dots, u_{\pi(4)}) = (u_3, u_4, u_0, u_2, u_1)$

Formalisation du problème

Définition : VdC (euclidien)

Entrée : Un ensemble de n points du plan U

Sortie : Une numérotation u_0, \dots, u_{n-1} des points qui minimise la longueur totale $\sum_{i=0}^{n-1} \ell(u_i, u_{i+1}) + \ell(u_{n-1}, u_0)$, où $\ell(-, -)$ désigne la distance euclidienne usuelle

Remarques

- ▶ Plus général : on peut se placer sur un graphe, une autre structure discrète, un autre espace métrique,...
- ▶ Numérotation des points = permutation des éléments de U

Algorithme par recherche exhaustive

- ▶ Parcours des solutions : permutations de $\{0, \dots, n-1\}$
- ▶ 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

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 $\{0, \dots, n-1\}$: n -uplets d'entiers tous distincts entre 0 et $n-1$
- ▶ Ordre lexicographique : $\pi < \pi'$ s'il existe i tel que :
 $\pi_{[j]} = \pi'_{[j]}$ pour tout $0 \leq j < i$ et $\pi_{[i]} < \pi'_{[i]}$

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 : $\pi \rightarrow \pi'$ 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

Permutations : passer à la suivante

Permutation *suivante*

- ▶ Exemple : quelle permutation π' après $\pi = 431520$?
- ▶ Trois conditions à respecter :
 - ▶ π' est une permutation : $\pi \rightarrow \pi'$ 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 maximal j tq $\pi[j] < \pi[j+1]$
($\pi[j+1] > \pi[j+2] > \dots > \pi[n-1]$)
 - ▶ j est l'indice *le plus à droite* qu'on peut incrémenter
 - ▶ on va incrémenter $\pi[j]$ sans toucher à $\pi[0], \dots, \pi[j-1]$
2. Échanger $\pi[j]$ avec le plus petit $\pi[\ell] > \pi[j]$ pour $\ell > j$
 - ▶ pour incrémenter $\pi[j]$, on l'échange avec le plus petit élément possible
 - ▶ $\ell > j$ car on ne veut pas toucher à $\pi[0], \dots, \pi[j-1]$
3. Retourner la fin $\pi[j+1, n[$
 - ▶ avant retournement : $\pi[j+1] > \pi[j+2] > \dots > \pi[n-1]$
 - ▶ ordre lexicographique commence par $\pi[j+1] < \pi[j+2] < \dots < \pi[n-1]$

Permutations : l'algorithme

PERMSUIVANTE(π) :

0. Si $\pi_{[0]} > \dots > \pi_{[n-1]}$: renvoyer « Fin »
1. Trouver j maximal tel que $\pi_{[j]} < \pi_{[j+1]}$
2. Trouver $\ell > j$ 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

Preuve de l'algo

- ▶ 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

VOYAGEURDECOMMERCE($U = \{u_0, \dots, u_{n-1}\}, \ell$) :

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

Calcul de L

- Boucle sur i , calcul d'une somme $\rightsquigarrow O(n)$

Retour au voyageur de commerce

VOYAGEURDECOMMERCE($U = \{u_0, \dots, u_{n-1}\}, \ell$) :

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

Calcul de L

- Boucle sur i , calcul d'une somme $\rightsquigarrow O(n)$

Preuve

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

Retour au voyageur de commerce

VOYAGEURDECOMMERCE($U = \{u_0, \dots, u_{n-1}\}, \ell$) :

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

Calcul de L

- Boucle sur i , calcul d'une somme $\rightsquigarrow O(n)$

Théorème

L'algorithme VOYAGEURDECOMMERCE résout le problème du voyageur de commerce en temps $O(n \times n!)$

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

Faire mieux ?

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

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 Principes du backtrack
- 2.3 Exemple 2 : le Sudoku

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 Principes du backtrack
- 2.3 Exemple 2 : le Sudoku

Principe de l'algorithme

Sur un exemple

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

Principe de l'algorithme

Sur un exemple

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

Algorithme récursif

- ▶ *Élimination* des variables une à une \rightsquigarrow construction pas à pas d'une affectation
- ▶ Élimination de x_n en premier : $\psi(x_1, \dots, x_{n-1}) = \varphi(x_1, \dots, x_{n-1}, b)$
- ▶ Cas de bases :
 - ▶ Clause vide insatisfiable \rightsquigarrow formule insatisfiable
 - ▶ Formule vide \rightsquigarrow formule satisfiable

Principe de l'algorithme

Sur un exemple

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

Algorithme récursif

- ▶ *Élimination* des variables une à une \rightsquigarrow construction pas à pas d'une affectation
- ▶ Élimination de x_n en premier : $\psi(x_1, \dots, x_{n-1}) = \varphi(x_1, \dots, x_{n-1}, b)$
- ▶ Cas de bases :
 - ▶ Clause vide insatisfiable \rightsquigarrow formule insatisfiable
 - ▶ Formule vide \rightsquigarrow formule satisfiable

Élimination d'une variable

- ▶ Formule ψ obtenue à partir de φ en posant $x_i = \text{VRAI}$ ou $x_i = \text{FAUX}$
- ▶ Exemple $x_i = \text{VRAI}$:
 - ▶ 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 $x_i = \text{FAUX}$ symétrique

Élimination d'une variable

ÉLIMINATION(φ, n, b) :

1. $\psi \leftarrow$ formule vide
2. Pour C dans φ :
3. $C' \leftarrow$ clause vide
4. $sat \leftarrow$ FAUX
5. Pour ℓ dans C :
6. Si $|\ell| = n$ et $\ell \times b > 0$:
7. $sat \leftarrow$ VRAI
8. Sinon si $|\ell| \neq n$:
9. Ajouter ℓ à C'
10. Si NON(sat) : ajouter C' à ψ
11. Renvoyer ψ

Rappel :

- Avec notre encodage, ℓ danc C est un entier relatif. Si $\ell = -i$, le littéral correspondant est $\neg x_i$ et si $\ell = i$, le littéral correspondant est x_i

Élimination d'une variable

ÉLIMINATION(φ, n, b) :

1. $\psi \leftarrow$ formule vide
2. Pour C dans φ :
3. $C' \leftarrow$ clause vide
4. $sat \leftarrow$ FAUX
5. Pour ℓ dans C :
6. Si $|\ell| = n$ et $\ell \times b > 0$:
7. $sat \leftarrow$ VRAI
8. Sinon si $|\ell| \neq n$:
9. Ajouter ℓ à C'
10. Si NON(sat) : ajouter C' à ψ
11. Renvoyer ψ

Rappel :

- ▶ Avec notre encodage, ℓ dans C est un entier relatif. Si $\ell = -i$, le littéral correspondant est $\neg x_i$ et si $\ell = i$, le littéral correspondant est x_i

Propriétés :

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

Algorithme de *backtrack* pour SAT

SATBACKTRACK(φ, n) :

1. Si φ est vide :
2. Renvoyer $A = [1, \dots, 1]$ (de taille n)
3. Si φ possède une clause vide :
4. Renvoyer « Insatisfiable »
5. Pour $b \in \{1, -1\}$:
6. $\psi \leftarrow \text{ÉLIMINATION}(\varphi, n, b)$
7. $A \leftarrow \text{SATBACKTRACK}(\psi, n - 1)$
8. Si ψ n'est pas insatisfiable : Renvoyer $A + [b]$
9. Renvoyer « Insatisfiable »

Propriétés

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

Bilan sur SATBACKTRACK

Théorème

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

Bilan sur SATBACKTRACK

Théorème

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

- ▶ Énoncé *strictement identique* pour RECHERCHEEXHAUSTIVESAT... !

Bilan sur SATBACKTRACK

Théorème

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

- ▶ Énoncé *strictement identique* pour RECHERCHEEXHAUSTIVESAT... !

Lequel des deux algorithmes est meilleur ?

- ▶ Dans le pire cas, aucun des deux
- ▶ Mais dans des cas favorables, SATBACKTRACK peut être *beaucoup* (*beaucoup*) plus rapide
 - ▶ Arbre des solutions très peu exploré
- ▶ Dans des cas défavorables pour SATBACKTRACK
 - ▶ Arbre des solutions exploré (presque) en entier
 - ▶ RECHERCHEEXHAUSTIVESAT peut alors être *légèrement* plus rapide

Bilan sur SATBACKTRACK

Théorème

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

- ▶ Énoncé *strictement identique* pour RECHERCHEEXHAUSTIVESAT... !

Lequel des deux algorithmes est meilleur ?

- ▶ Dans le pire cas, aucun des deux
- ▶ Mais dans des cas favorables, SATBACKTRACK peut être *beaucoup* (*beaucoup*) plus rapide
 - ▶ Arbre des solutions très peu exploré
- ▶ Dans des cas défavorables pour SATBACKTRACK
 - ▶ Arbre des solutions exploré (presque) en entier
 - ▶ RECHERCHEEXHAUSTIVESAT peut alors être *légèrement* plus rapide

Conclusion

L'algorithme SATBACKTRACK est plus efficace *en pratique*.

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 **Principes du backtrack**
- 2.3 Exemple 2 : le Sudoku

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

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 des solutions
 - ▶ test des solutions partielles

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

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 des solutions
 - ▶ test des solutions partielles

Remarque

- ▶ En recherche exhaustive, on construit aussi parfois *récursivement* les solutions
- ▶ La différence majeure : **test de solutions partielles** : permet d'éliminer des branches d'exploration

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 \rightsquigarrow arbre k -aire
 - ▶ vecteur de longueur n \rightsquigarrow arbre de hauteur n
- ▶ Plus général : structure d'arbre parfois complexe à voir
- ▶ L'arbre n'est *pas* représenté en mémoire \rightsquigarrow arbre implicite

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

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 \rightsquigarrow arbre k -aire
 - ▶ vecteur de longueur n \rightsquigarrow arbre de hauteur n
- ▶ Plus général : structure d'arbre parfois complexe à voir
- ▶ L'arbre n'est *pas* représenté en mémoire \rightsquigarrow arbre implicite

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

Parcours partiel

- ▶ Exemple : affectation partielle non satisfaisante \rightsquigarrow retour en arrière
- ▶ *Élagage* de l'arbre :
 - ▶ si solution partielle incorrecte (nécessité d'un algo de test) : pas besoin de continuer
 - ▶ *branches* non explorées de l'arbre

Caractéristiques

Analyse de complexité

- ▶ Algorithme récursif \rightsquigarrow équation de récurrence pour la complexité
- ▶ Arbre des solutions \rightsquigarrow 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

1. Recherche exhaustive

- 1.1 L'odomètre ou 'compteur kilométrique'
- 1.2 Exemple 1 : SAT
- 1.3 Principes de la recherche exhaustive
- 1.4 Exemple 2 : le voyageur de commerce (euclidien)

2. *Backtrack* ou retour sur trace

- 2.1 Exemple 1 : le retour de SAT
- 2.2 Principes du backtrack
- 2.3 Exemple 2 : le Sudoku

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 (GÉNÉRALISÉ)

Entrée : Une grille G de dimensions $n^2 \times n^2$, remplie 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
- ▶ $\lfloor i/n \rfloor = \lfloor k/n \rfloor$ et $\lfloor j/n \rfloor = \lfloor \ell/n \rfloor$ (zone)

Ou 'aucune solution'

Formalisation

SUDOKU (GÉNÉRALISÉ)

Entrée : Une grille G de dimensions $n^2 \times n^2$, remplie 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
- ▶ $\lfloor i/n \rfloor = \lfloor k/n \rfloor$ et $\lfloor j/n \rfloor = \lfloor \ell/n \rfloor$ (zone)

Ou 'aucune solution'

Parcours de la Grille

- ▶ On parcourt la grille ligne par ligne
- ▶ Primitive utile :
 $\text{CASESUIVANTE}(u, v)$ avec
 $0 \leq u, v \leq n^2$

CASESUIVANTE(u, v)

1. Si $u = n^2 - 1$ et $v = n^2 - 1$,
Renvoyer **Fin**
2. Si $v = n^2 - 1$, Renvoyer $(u + 1, 0)$
3. Renvoyer $(u, v + 1)$

Plan de bataille

Algorithme récursif

- ▶ Pour chaque case initialement vide :
 - ▶ Essayer toutes les valeurs possibles dans la case
 - ▶ Vérifier qu'on ne crée aucune incohérence
- ▶ Passer à la CASESUIVANTE...

Plan de bataille

Algorithme récursif

- ▶ Pour chaque case initialement vide :
 - ▶ Essayer toutes les valeurs possibles dans la case
 - ▶ Vérifier qu'on ne crée aucune incohérence
- ▶ Passer à la CASE SUIVANTE...

Test de solutions partielles

- ▶ Tester si une grille (partielle) est incohérente
- ▶ Test des n^2 lignes, n^2 colonnes et n^2 zones?
- ▶ **Non !** Uniquement la nouvelle case → une ligne, une colonne, une zone

Plan de bataille

Algorithme récursif

- ▶ Pour chaque case initialement vide :
 - ▶ Essayer toutes les valeurs possibles dans la case
 - ▶ Vérifier qu'on ne crée aucune incohérence
- ▶ Passer à la CASE SUIVANTE...

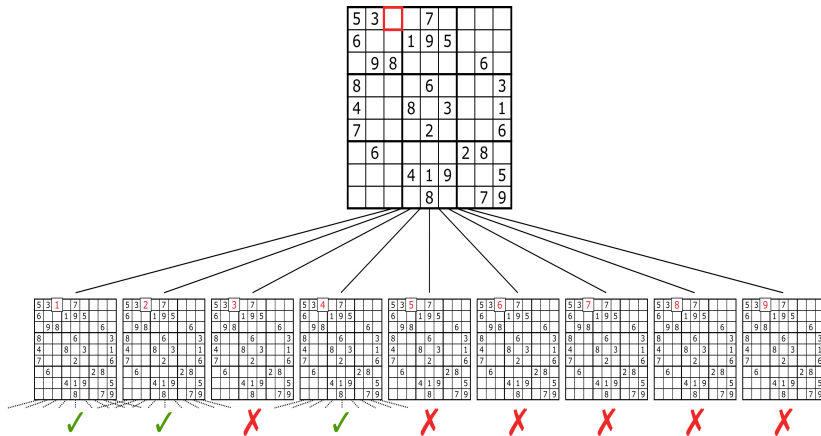
Test de solutions partielles

- ▶ Tester si une grille (partielle) est incohérente
- ▶ Test des n^2 lignes, n^2 colonnes et n^2 zones?
- ▶ **Non !** Uniquement la nouvelle case → une ligne, une colonne, une zone

Remarque

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

Arbre des solutions



Test de solution partielle

$\text{VALIDE}(G, n, (u, v), x)$ (avec (u, v) coordonnées de la case à remplir et x valeur à tester)

1. Pour $k = 0$ à $n^2 - 1$:
2. Si $(k \neq v \text{ et } G[u, k] = x)$: Renvoyer **Faux** *#Test de ligne*
3. Si $(k \neq u \text{ et } G[k, v] = x)$: Renvoyer **Faux** *#Test de colonne*
4. $(z_u, z_v) \leftarrow (n \lfloor u/n \rfloor, n \lfloor v/n \rfloor)$ *# 'Début' de la zone de (u, v)*
5. Pour $k = 0$ à $n - 1$:
6. Pour $\ell = 0$ à $n - 1$: *# Test de la zone*
7. Si $(z_u + k, z_v + \ell) \neq (u, v) \text{ et } G[z_u + k, z_v + \ell] = x$:
 Renvoyer **Faux**
8. Renvoyer **Vrai**

Propriétés

- ▶ Renvoie **Vrai** si et seulement si $G[u, v] \leftarrow x$ ne crée pas de conflit
- ▶ Sa complexité est $O(n^2)$ Rq : taille de l'entrée : $O(n^4)$

Algorithme de *backtrack* pour le Sudoku

SUDOKUBACKTRACK($G, n, (u, v)$) :

1. Tant que $(u, v) \neq \mathbf{Fin}$ et $G[u, v] \neq 0$: $(u, v) \leftarrow \text{CaseSuivante}(u, v)$
2. Si $(u, v) = \mathbf{Fin}$: renvoyer **Vrai**
3. Pour x de 1 à n^2 :
4. Si VALIDE($G, n, (u, v), x$) :
5. $G[u, v] \leftarrow x$
6. Si SUDOKUBACKTRACK($G, n, \text{CaseSuivante}(u, v)$) :
renvoyer **Vrai**
7. $G[u, v] \leftarrow 0$ et renvoyer **Faux**

Propriétés

- ▶ Se lance par SUDOKUBACKTRACK($G, n, (0, 0)$)
- ▶ L'algo renvoie **Vrai** et G contient une solution ssi il en existe une
- ▶ Sa complexité vérifie : $T(m) \leq n^2 \cdot T(m-1) + O(n^4)$ où m est le nombre cases vides $\rightsquigarrow T(m) = O(n^{4m})$ $O(n^{4n^4})$ si $m \simeq n^4$

Conclusion générale

Deux techniques proches

- ▶ *Recherche exhaustive* et *backtrack* sont très proches
- ▶ 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 (issu de la recherche opérationnelle) :
 - ▶ 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 des techniques d'IA