Introduction à l'Intelligence Artificielle
et à la Théorie des Jeux

Résoudre un problème par une recherche

Recherche informée — suite

Ahmed Bouajjani

abou@irif.fr

Description Formelle d'un Problème

- **Etat initial** par où l'agent devra commencer
- Ensemble d'**actions** possibles (chacune applicable à un ensemble d'états)
- Fonction de **transition** : état ***s***, action ***a*** —> ***s'*** obtenu en appliquant ***a*** à ***s***
- Fonction de **coût** : transition *s -a-> s'* —> *c*(*s, a, s'*) — distance, temps, …
- Test de l'**état but** où l'agent devra arriver

=> Ensemble des états + transitions = graphe dirigé (étiqueté)

=> Trouver une solution = recherche dans un graphe

Résoudre un problème

- Recherche dans un graphe, schéma générique
- Différentes instances
- Recherche non informée
- **Recherche informée (guidée lors de l’exploration de l’espace d’états)**

RECHERCHE INFORMÉE

- Appelée BEST_FIRST_SEARCH
- Guidée par une évaluation qui inclut une estimation du coût min vers le but
- **heuristique h(n)** : le coût estimé du chemin le plus court de n au but
- **g(n)** : le coût du chemin courant (calculé) du sommet initial à n
- **f(n)** : la fonction d'évaluation, à chaque itération on cherche dans la frontière le sommet n avec f(n) minimal — est une combinaison de g(n) et de h(n)

HEURISTIQUE ?

- **Qui donne une « bonne » estimation du coût réel**
- **Facile à calculer**

Exemple :

Pour le problème du plus court chemin entre les villes

h(n) - la distance à vol d'oiseau de la ville n à la ville destination

=> Que veut dire « bonne » estimation ?

=> Comment définir une heuristique ?

ALGO COÛT UNIFORME

- si **f(n)=g(n)**, c'est-à-dire on n'utilise pas l’heuristique, on obtient alors la recherche à coût uniforme (non informée, déjà étudiée)

RECHERCHE GLOUTONNE (GREEDY BEST-FIRST SEARCH)

- on prend **f(n) = h(n)**
- le coût du chemin de sommet source vers n n'est donc pas pris en compte

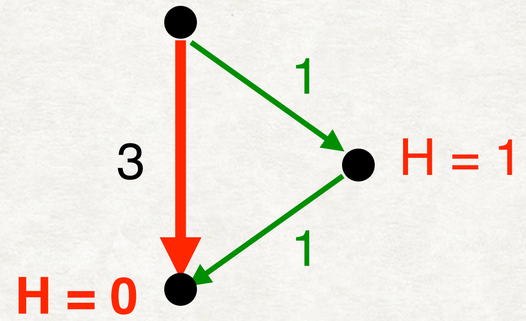
RECHERCHE GLOUTONNE (GREEDY BEST-FIRST SEARCH)

- on prend **f(n) = h(n)**
- le coût du chemin de sommet source vers n n'est donc pas pris en compte
- la version « en arbre » de l'algorithme est-elle complète ?
  - h(n) = coût réel entre n et le but (= infini si but n'est pas atteignable)

    => Complet dans ce cas, mais calcul h(n) pratiquement infaisable
  - h(n) = 0 pour tout n

    => Incomplet dans ce cas, donc incomplet en général
- la version « en arbre » de l'algorithme est-elle optimale ?

  => Non optimale



RECHERCHE GLOUTONNE (GREEDY BEST-FIRST SEARCH)

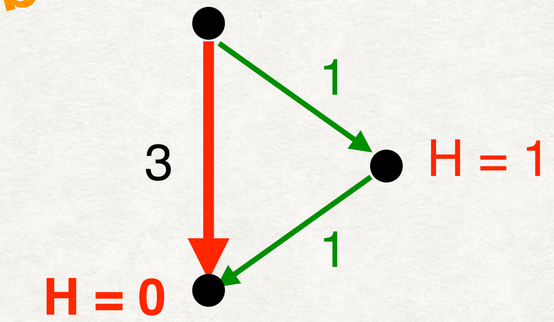
- on prend **f(n) = h(n)**
- le coût du chemin de sommet source vers n n'est donc pas pris en compte
- la version « en graphe » de l'algorithme est-elle complète ?

  OUI, pour toute heuristique

- la version « en graphe » de l'algorithme est-elle optimale ?

  NON

L'algorithme ignore les coûts réels observés



ALGORITHME A* - MINIMISER LE COÛT TOTAL

- dans chaque noeud n on stocke g(n) et **f(n)=g(n)+h(n)**
- à chaque étape on choisit un noeud avec f(n) minimal

ALGORITHME A* (VERSION PARCOURS EN ARBRE)

1. mettre le sommet initial s dans FRONTIERE, g(s)=0, f(s)=h(s)
2. Si FRONTIERE vide, sortir avec échec
3. retirer de FRONTIERE le noeud n pour lequel f(n) est minimal
4. si n est le but alors terminer et retracer la solution, le chemin de s à n
5. développer n pour engendrer tous ses successeurs. Pour tout successeur m de n :
   A. Calculer f(m)=g(m)+h(m), où g(m)=g(n)+coût(n, m)
   B. Insérer m dans FRONTIERE
6. aller à 2

ALGORITHME A* (VERSION PARCOURS EN GRAPHE)

1. mettre le sommet initial s dans FRONTIERE, g(s)=0, f(s)=h(s)
2. Si FRONTIERE vide, sortir avec échec
3. retirer de FRONTIERE et mettre dans FERMES le noeud n pour lequel f(n) est minimal
4. si n est le but alors terminer et retracer la solution, le chemin de s à n
5. développer n pour engendrer tous ses successeurs. Pour tout successeur m de n :
   A. Si m.état n'est pas dans un noeud de FERMES alors calculer f(m)=g(m)+h(m), où g(m)=g(n)+coût(n, m)
   B. Insérer m dans FRONTIERE
6. aller à 2

HEURISTIQUE ADMISSIBLE

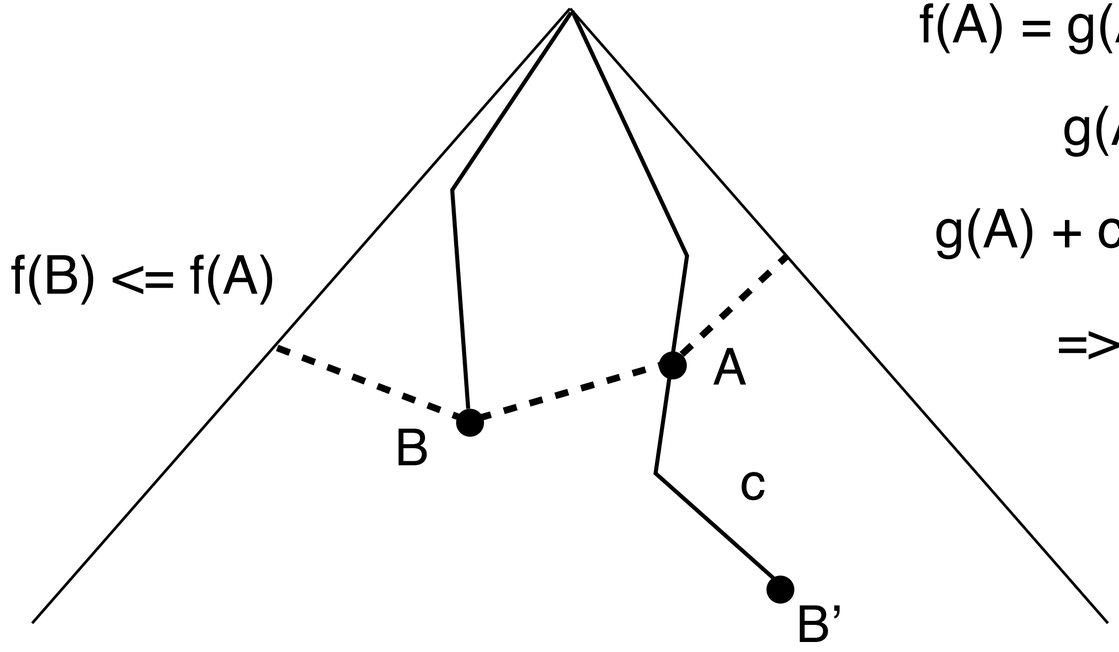
la fonction heuristique h est **admissible** si pour chaque sommet n

h(n) <= coût réel du chemin le plus court de n vers un état but

donc *une heuristique admissible ne surestime jamais le coût réel.*

Algorithme A* parcours en arbre — Optimalité ?

**Si l'heuristique est admissible alors A*
version parcours en arbre est optimal**



$f(A) = g(A) + h(A) <= g(A) + c$ Admissibilité

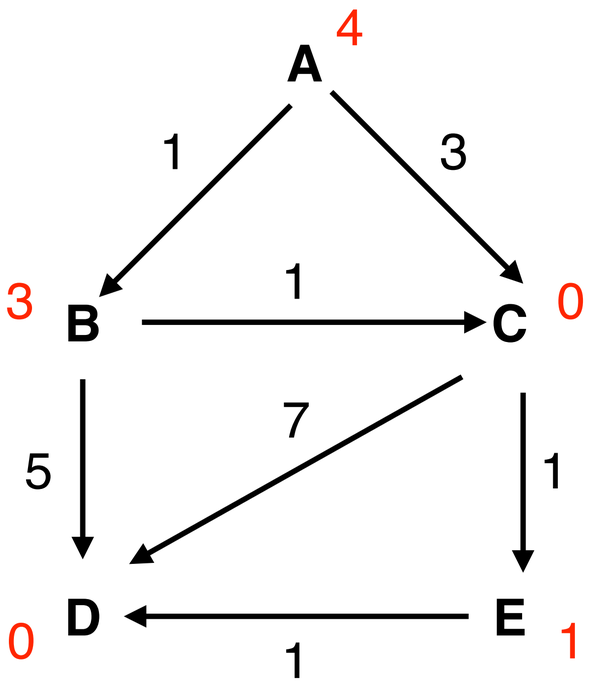
$g(A) + c < g(B)$ Optimalité

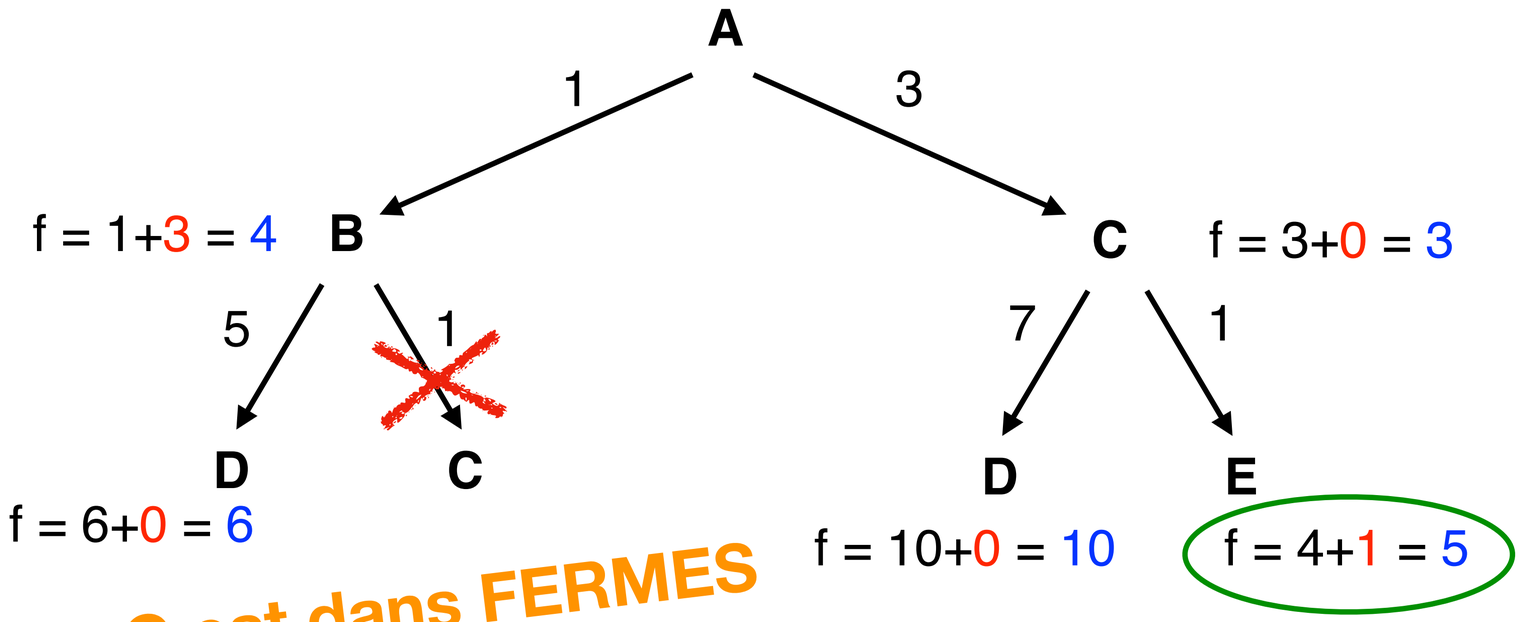
$g(A) + c < g(B) + h(B) = f(B)$ Admissibilité : h(B)=0

$=> f(A) < f(B)$

Contradiction avec f(B) <= f(A)

Algorithme A* parcours en graphe — Optimalité ?





C est dans FERMES

=> L'algorithme rate le chemin optimal !!

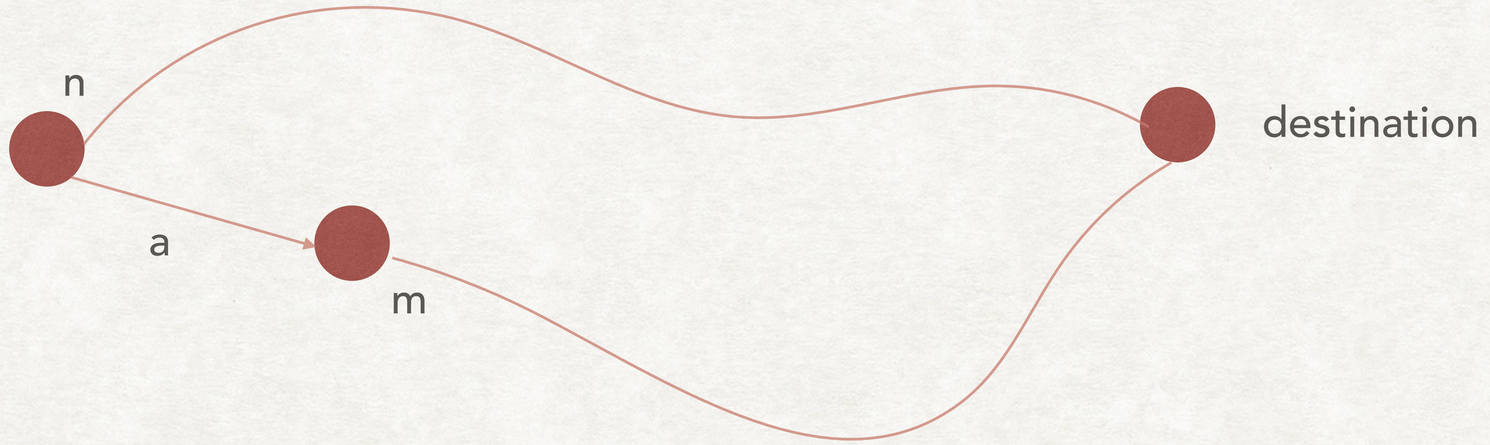
HEURISTIQUE CONSISTANTE

Une heuristique est **consistante (ou cohérente)** si **h(but) =0** et elle satisfait l'inégalité de triangle:

pour chaque couple de sommets n et m et toute action a telle que

m=transition(n,a) , on a :

h(n) <= coût(n,a,m) + h(m)



coût(n,a,m) - coût de l'action a

OPTIMALITÉ DE A*

A. Si la heuristique h est admissible alors la version tree-search de l'algorithme A* est optimale.

B. Si la heuristique h est consistante alors la version graph-search de l'algorithme A* est optimale.

OPTIMALITÉ DE A*

A. Si la heuristique h est admissible alors la version tree-search de l'algorithme A* est optimale.

B. Si la heuristique h est consistante alors la version graph-search de l'algorithme A* est optimale.

Algorithme A* parcours en graphe — Optimalité ?

Si l'heuristique est consistante alors A* version parcours en graphe est optimal

Si l'algorithme choisit un noeud dans Frontière, alors le chemin trouvé à ce moment là, allant de la racine à ce noeud, est optimal

Lemme: f(n) est croissante le long des chemins développés

- Soit n' un successeur de n
- f(n') = g(n') + h(n') = g(n) + c(n,n') + h(n')
- g(n) + c(n,n') + h(n) >= g(n) + h(n) — consistance de h
- Donc f(n') >= g(n) + h(n) = f(n)

- Soit n choisi par l'algorithme mais chemin trouvé non optimal
- Il existe alors n' dans Frontière à ce moment là sur le chemin optimal
- Puisque f est croissante le long de tous les chemins, f(n') aurait été inférieure à f(n) — contradiction car n' aurait été alors choisi

COMMENT TROUVER LES HEURISTIQUES ADMISSIBLES?

HEURISTIQUES POUR LE JEU TAQUIN

- h1 - le nombre de pièces qui ne sont pas à leurs places

- Soit (x1,y1), (x2,y2) deux positions sur la grille.
  Soit la distance

$$Manhattan((x1,y1),(x2,y2)) = |\, x1-x2 \,| + |\, y1 - y2 \,|$$

On définit alors

h2 = somme sur toutes les pièces de la grille de la distance Manhattan entre la position courante de la pièce et sa position finale dans la configuration finale

COMMENT TROUVER LES HEURISTIQUES OPTIMALES?

- En assouplissant les conditions (ajoutant les actions)
- En considérant des sous-problèmes (moins de cas)
- => La résolution du problème de recherche devient moins complexe
- On peut calculer les coûts avec ses « simplifications » et les prendre comme estimations
- On peut les combiner

Combinaison d’heuristique

Soit h1, h2, …, hk des heuristiques admissibles

Max(h1, h2, …, hk) est aussi admissible
et domine toute les heuristiques hi

Heuristique admissible par assouplissement de règles

L'heuristique pour le jeu de taquin obtenue en assouplissant les règles du jeu :

- une pièce peut être déplacée sur n’importe quelle case

- => h = Nb de cases qui ne sont pas à leur place dans la configuration finale

Heuristique admissible par assouplissement de règles

L'heuristique pour le jeu de taquin obtenue en assouplissant les règles du jeu :

- une pièce peut être déplacée sur la case voisine même si la case voisine est occupée

- => heuristique basée sur la distance de Manhattan

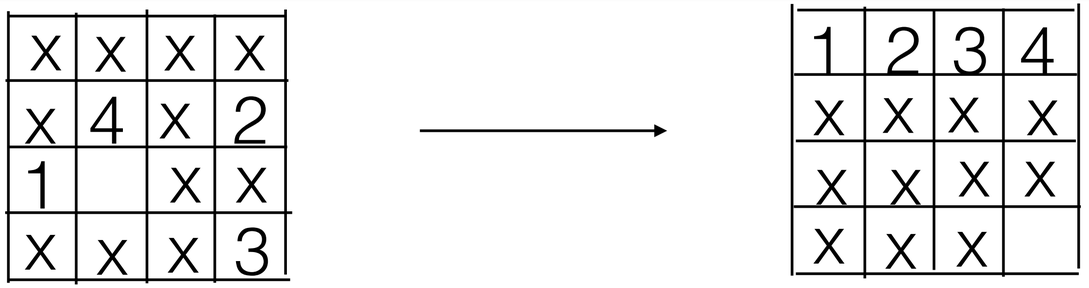
Heuristiques admissibles à partir de sous-problèmes

x	x	x	x
x	4	x	2
1		x	x
x	x	x	3

1	2	3	4
x	x	x	x
x	x	x	x
x	x	x	

h=le nombre de pas pour passer de la configuration gauche à droite avec les règles de taquin habituelles

Heuristiques admissibles à partir de sous-problèmes disjoints



- h1=le nombre de pas pour passer de la configuration gauche à droite avec les règles de taquin habituelles en ne comptant que les déplacement de pièces numérotées

- h2=même chose avec les pièces 5,6,7,8 visibles

- h3= même chose avec les pièces 9,10,11,12 visibles

- h4=même chose avec les pièces 13,14,15 visibles

h = h1+h2+h3+h4

Pour la taille 5 sur 5 accélération de 1 000 000 par rapport à Manhattan

Recherche locale

Dans le problème de 8 reines nous cherchons de trouver une solution (l'état destination) et non pas le chemin qui mène vers une solution.

Même situation pour d'autres problèmes, en optimisation

Recherche locale

Dans le problème de 8 reines nous cherchons de trouver une solution (l'état destination) et non pas le chemin qui mène vers une solution.

Même situation pour d'autres problèmes, en optimisation

Recherche locale

Dans le problème de 8 reines nous cherchons de trouver une solution (l'état destination) et non pas le chemin qui mène vers une solution.

Même situation pour d'autres problèmes, en optimisation

Pour ce type de problème la **recherche locale :**

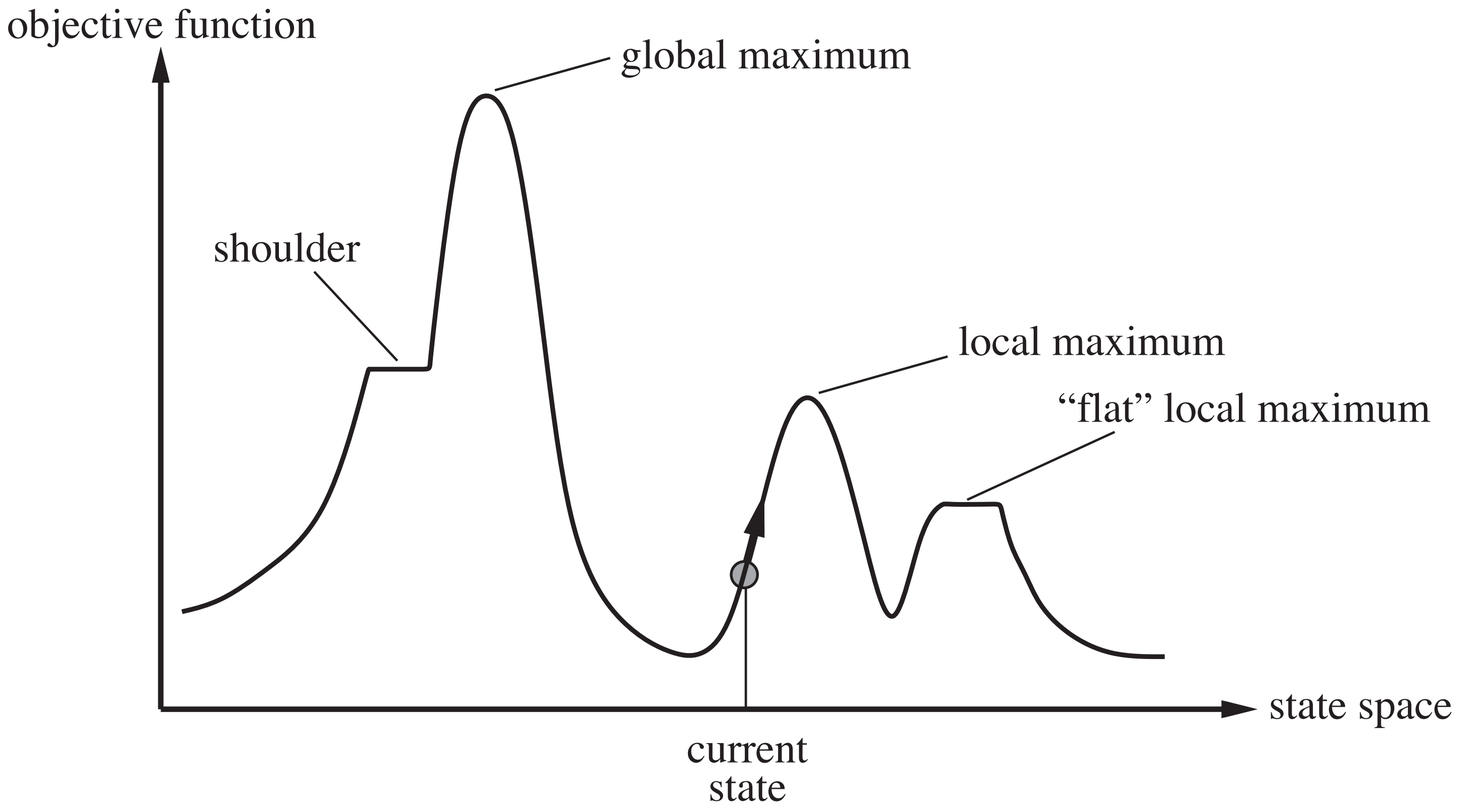
- on maintient juste le noeud courant, pas le chemin qui mène vers ce noeud
- on se déplace vers un voisin du noeud courant

Recherche locale

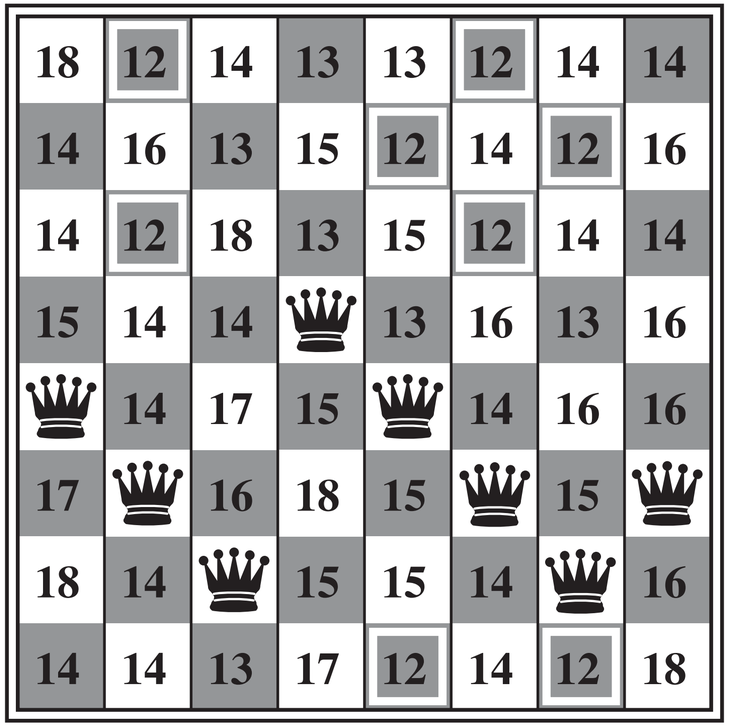
- fonction de coût définie pour chaque noeud
- solution - le noeud de coût minimal
- Problème d'optimisation de la fonction du coût.
- Exemple :

1. 8 reines, le coût d'une configuration de 8 reines = le nombre de couples de reines en conflit
2. la solution : la configuration avec le coût minimal 0

Recherche locale

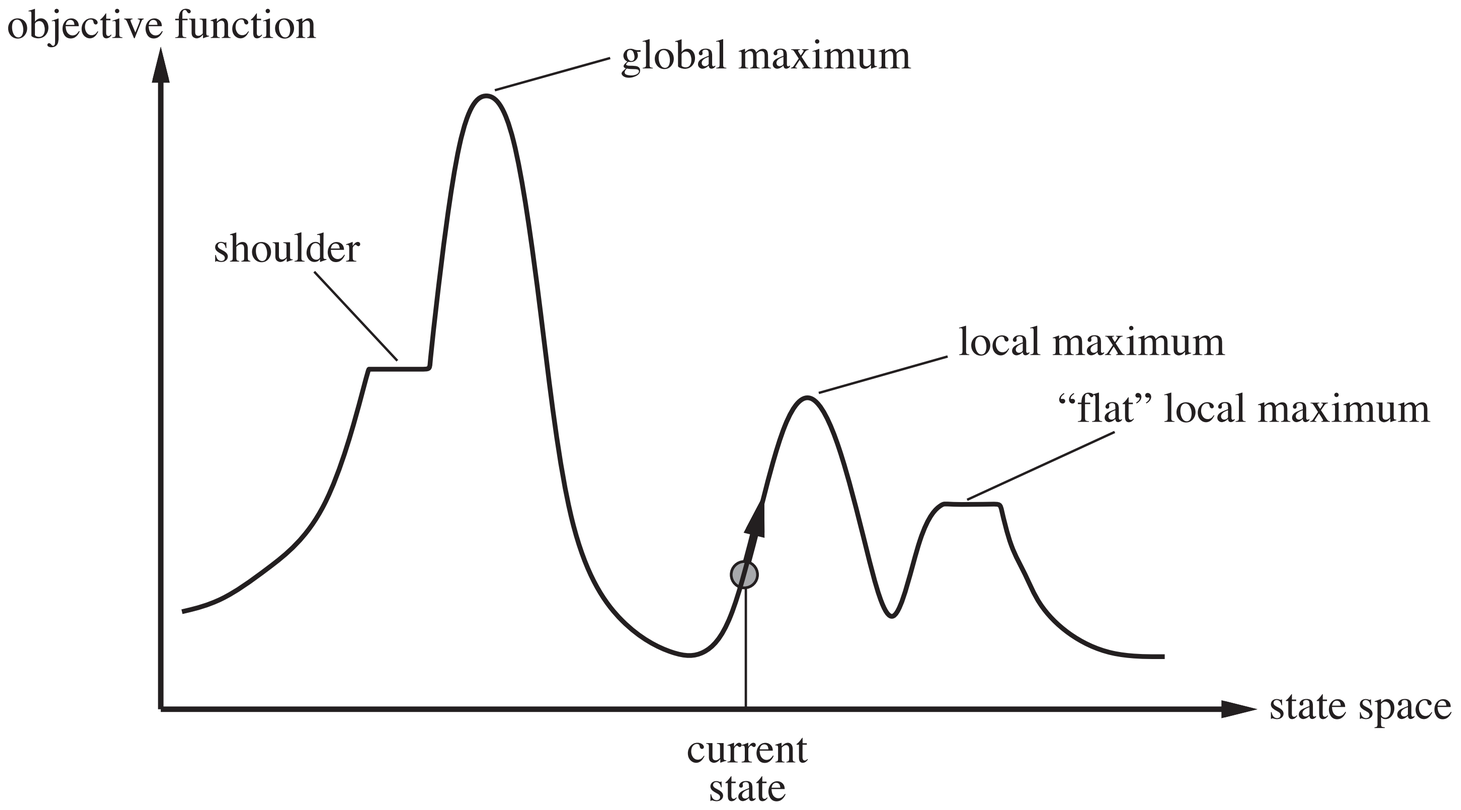


Recherche locale : exemple



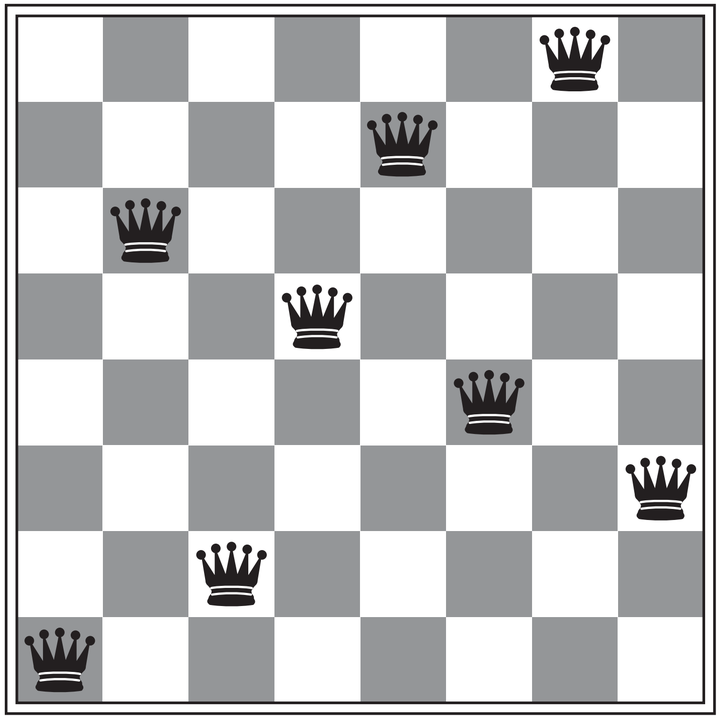
- Une configuration avec coût = 17
- Coût des succeurs en déplaçant une reine sur sa colonne

Recherche locale



- Problème des optimaux locaux
- Problème des plateaux

Miniumum Local



- Coût = 1 et tout successeur a un coût supérieur

Steepest-descent (descente gradient)

```
fonction steepest-descent(problème) retourne le minimum local
courant = new noeud(problème.état_initial)
loop do
    voisin = le voisin avec le coût minimal
    if voisin.coût >= courant.coût return courant
    courant=voisin
done
```

Steepest-descent appliquée à 8 reines (avec initialement 8 reines, une par ligne):

- bloqué dans 86% de cas sans qu'une solution soit trouvée
- 4 itérations en moyenne si une solution est trouvée
- 3 itérations en moyenne si une solution n'est pas trouvée et l'algorithme steepest-descent bloque
- le nombre d'état 8^8

Steepest descent

Steepest-descent s'arrête si on se trouve sur un plateau (pas de voisins avec le coût inférieur mais il existe des voisins avec le coût égal).

Pour éviter le blocage dans ce cas : permettre les mouvement latéraux (s'il n'y a pas de voisin de coût inférieur mais il en existe avec le coût égal au coût du courant alors déplacer le courant vers un tel voisin) .

Limiter le nombre de mouvement latéraux consécutifs (pour éviter un long parcours inutile sur un plateau).

Steepest descent

Pour 8 reines : steepest descent avec au plus 100 mouvements latéraux consécutifs trouve une solution dans 94% de cas.

Mais le nombre moyen de mouvements avant que steepest-descent trouve une solution passe à 21 et le nombre moyen de mouvement avant que l'algorithme soit bloqué passe à 64.

Steepest descent stochastique

- A chaque étape on sélectionne un voisin avec une certaine probabilité — probabilité peu dépendre la pente (le coût) : plus élevée pour les voisins avec le coût plus petit (la probabilité proportionnelle à 1/coût). Cela permet de passer les collines avoisinante et sortir d'un minimum local entouré de collines.

- On peut redémarrer et conduire plusieurs recherche à partir de plusieurs point initiaux choisis aléatoirement

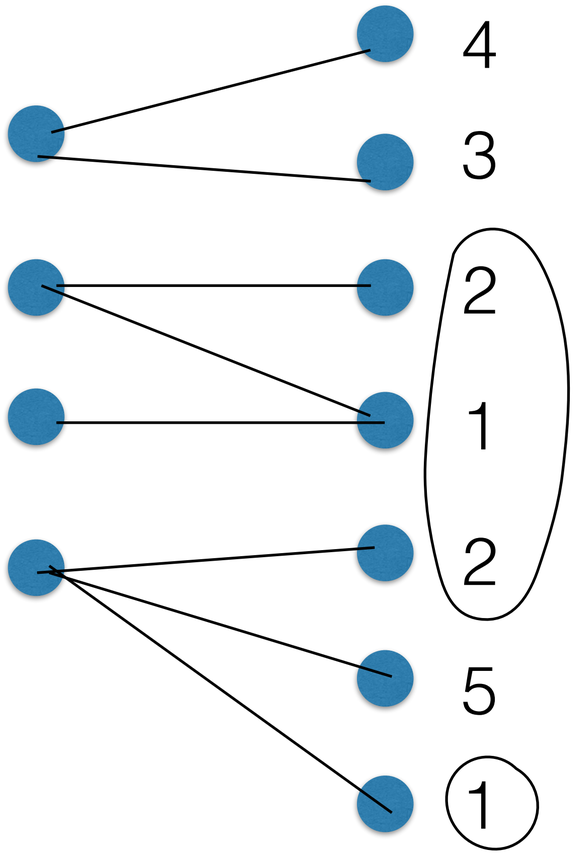
Local beam search
recherche locale à faisceau

- A chaque moment on maintient k noeuds n1,…,nk
- On génère tous les voisins V de ces noeuds
- Si parmi ces voisins il y a une solution alors on termine
- Sinon pour l'étape suivant on choisit dans V k noeuds dont le coût est minimal

Noter que il est possible que pour l'étape suivant on garde plusieurs voisin d'un noeud et aucun voisin d'un autre (on choisit k noeud d'un ensemble de tous les voisins de {n1,…,nk}).

Local beam search
recherche locale à faisceau

k=4



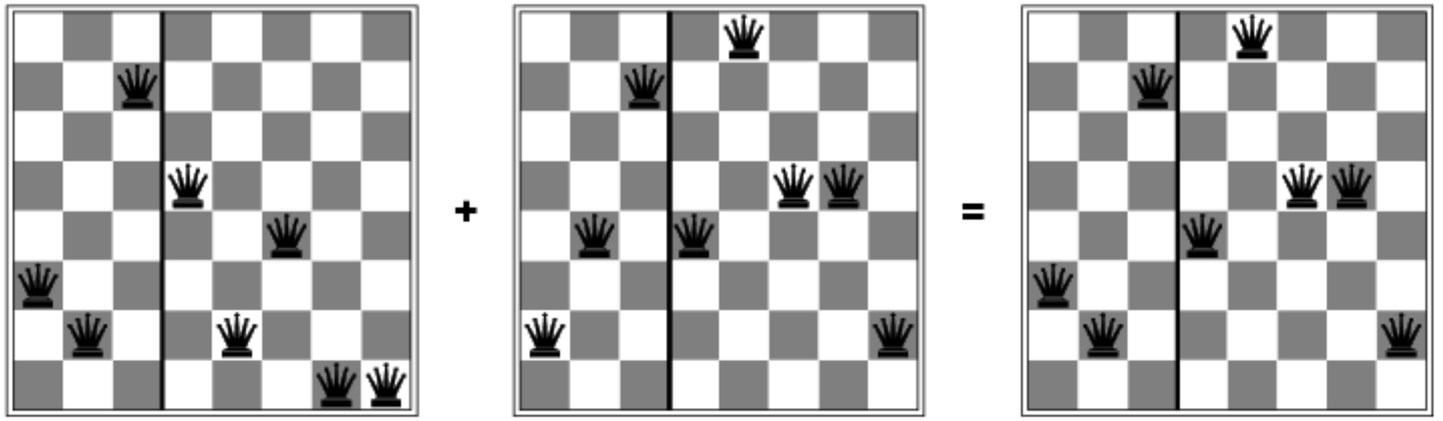
Algorithmes génétiques

- Un état successeur obtenu en combinant deux états parents
- Initialement k états générés aléatoirement (population)
- Un état représenté par une chaine de caractères sur un alphabet fini (souvent 0,1)
- La fonction d'évaluation (fitness fonction) permet d'évaluer la qualité d'un état. Plus la valeur de la fonction est élevée plus l'état est mieux "adapté".
- Produire la nouvelle génération d'états par la sélection, croisement et mutation.

Algorithmes génétiques

- 24748552 -- la position de chaque reines, colonne par colonne, par exemple dans la première colonne la reine se trouve sur la deuxième ligne, dans la deuxième colonne sur la quatrième ligne etc.)
- Fitness function: le nombre de couples de reines qui ne sont pas en conflit (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Algorithmes génétiques (croisement de deux configurations dans le problème de 8 reines)



Algorithme génétique

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 **to** SIZE(*population*) **do**
      x ← RANDOM-SELECTION(*population*, FITNESS-FN)
      y ← RANDOM-SELECTION(*population*, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough, or enough time has elapsed
  return the best individual in *population*, according to FITNESS-FN

---

function REPRODUCE(x, y) **returns** an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, $c+1$, n))