Intelligence artificielle

Algorithmes et recherches heuristiques

Elise Bonzon
elise.bonzon@u-paris.fr

LIPADE - Université de Paris
http://www.math-info.univ-paris5.fr/∽bonzon/

Algorithmes et recherches heuristiques

Recherche meilleur d’abord

Recherche meilleur d'abord

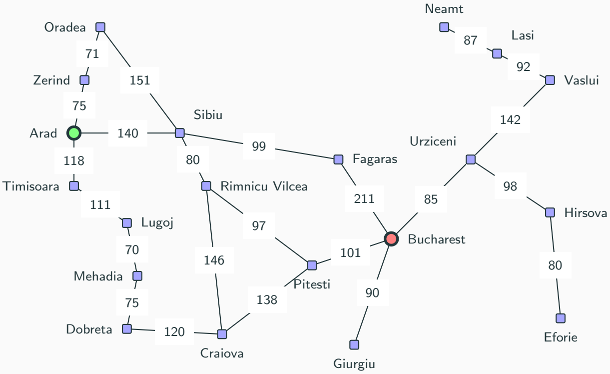
- **Rappel** : Une stratégie de recherche permet de choisir l'ordre dans lequel les états sont développés
- **Idée** : Utiliser une fonction d'évaluation f pour chaque noeud
  - $\rightarrow$ mesure l'utilité d'un noeud
  - $\rightarrow$ introduction d'une fonction heuristique $h(n)$ qui **estime** le coût du chemin le plus court pour se rendre au but
  - Comme dans les algorithmes de recherche aveugle, $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- `InsertAll` insère le nœud par ordre décroissant de la valeur de la fonction d'évaluation
- **Cas spéciaux** :
  - Recherche gloutonne (un choix n'est jamais remis en cause)
  - A*

Recherche gloutonne

Recherche gloutonne

- Fonction d'évaluation $f(n) = h(n)$ (heuristique)
- $h(n)$: estimation du coût de n vers l'état final
- Par exemple, $h_{dd}(n)$ est la distance à vol d'oiseau entre la ville n et Bucharest
- La **recherche gloutonne** développe le nœud qui paraît le plus proche de l'état final, sans prendre en compte le coût du chemin déjà parcouru

Le voyage en Roumanie



Ligne droite jusqu'à Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Lasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recherche gloutonne

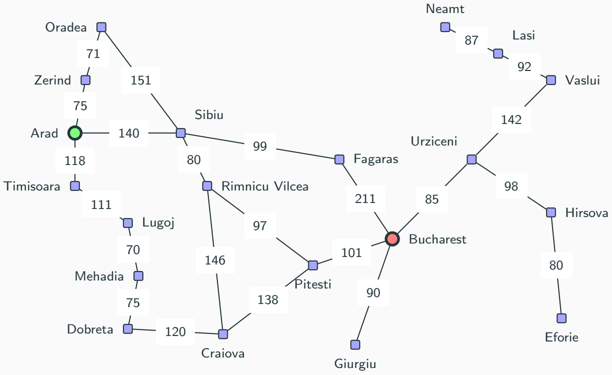
- **Complétude** : Incomplet (peut rester bloqué dans des boucles)
  - Exemple : Arad $\rightarrow$ Zerind $\rightarrow$ Arad $\rightarrow \ldots$
  - Complet si on ajoute un test pour éviter les états répétés
- **Temps** : $O(b^m)$
  - Une bonne heuristique peut améliorer grandement les performances
- **Espace** : $O(b^m)$: Garde tous les nœuds en mémoire
- **Optimale** : Non

L'algorithme A^*

Algorithme A*

- **Idée** : Eviter de développer des chemins qui sont déjà chers
- **Fonction d'évaluation** : $f(n) = g(n) + h(n)$
  - $g(n)$ est le coût de l'état initial à l'état n
  - $h(n)$ est le coût estimé pour atteindre l'état final
  - $f(n)$ est le coût total estimé pour aller de l'état initial à l'état final en passant par n
- A* utilise une heuristique admissible
  - $h(n) \leq h^*(n)$ où $h^*(n)$ est le coût réel pour aller de n jusqu'à l'état final
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste
  - Par exemple h_{dd} ne surestime jamais la vraie distance
- Si $h(n) = 0$ pour tout n, alors A* est équivalent à l'algorithme de Dijkstra de calcul du plus court chemin
- Théorème : **A* est optimale**

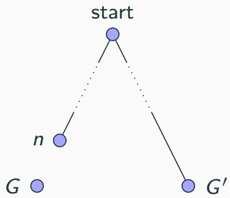
Le voyage en Roumanie



Ligne droite jusqu'à Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Lasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Preuve d'optimalité de A*

- Supposons qu'il y ait un état final non optimal G' généré dans la liste des nœuds à traiter
- Soit n un nœud non développé sur le chemin le plus court vers un état final optimal G



$$\begin{array}{rcl} f(G') & = & g(G') \text{ car } h(G') = 0 \\ & > & g(G) \text{ car } G' \text{ n'est pas optimale} \\ & \geq & f(n) \text{ car } h \text{ est admissible} \end{array}$$

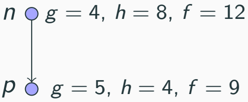
- $f(G') > f(n)$, donc A* ne va pas choisir G'

Algorithme A*

- **Complétude** : Oui, sauf s'il y a une infinité de nœuds tels que $f \leq f(G)$
- **Temps** : exponentielle selon la longueur de la solution
- **Espace** : exponentielle (garde tous les nœuds en mémoire)
    - Habituellement, on manque d'espace bien avant de manquer de temps
- **Optimale** : Oui

Que faire si f décroît ?

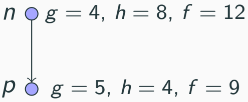
- Avec une heuristique admissible, f peut **décroître** au cours du chemin
- Par exemple, si p est un successeur de n, il est possible d'avoir



- On perd de l'information
  - $f(n) = 12$, donc le vrai coût d'un chemin à travers n est ≥ 12
  - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12

Que faire si f décroît ?

- Avec une heuristique admissible, f peut **décroître** au cours du chemin
- Par exemple, si p est un successeur de n, il est possible d'avoir

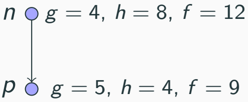


- On perd de l'information
  - $f(n) = 12$, donc le vrai coût d'un chemin à travers n est ≥ 12
  - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12

$\Rightarrow$ Au lieu de $f(p) = g(p) + h(p)$, on utilise
$f(p) = max(g(p) + h(p), f(n))$

Que faire si f décroît ?

- Avec une heuristique admissible, f peut **décroître** au cours du chemin
- Par exemple, si p est un successeur de n, il est possible d'avoir

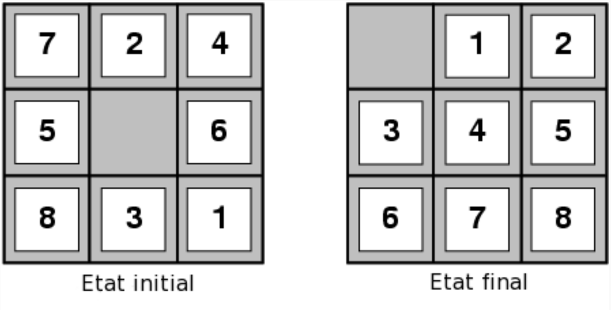


- On perd de l'information
  - $f(n) = 12$, donc le vrai coût d'un chemin à travers n est ≥ 12
  - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12

$\Rightarrow$ Au lieu de $f(p) = g(p) + h(p)$, on utilise $f(p) = max(g(p) + h(p), f(n))$

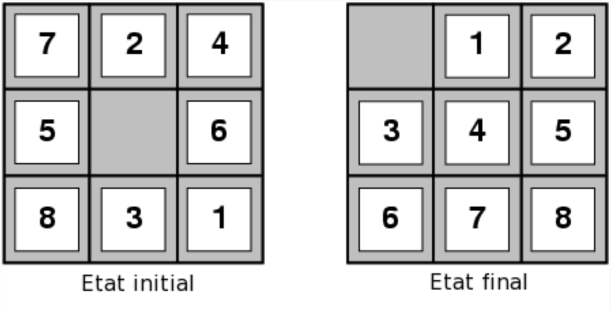
$\rightarrow$ f ne décroît jamais le long du chemin

Exemple d'heuristiques admissibles : le taquin



- $h_1(n) =$ le nombre de pièces mal placées

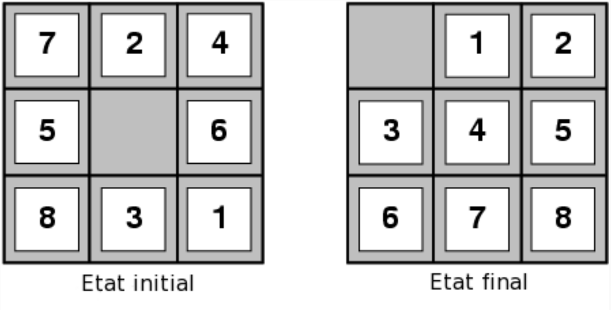
Exemple d'heuristiques admissibles : le taquin



- $h_1(n) =$ le nombre de pièces mal placées

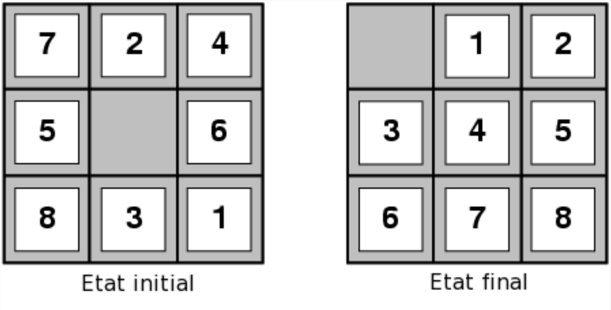
  $\rightarrow\ h_1(S) = 8$

Exemple d'heuristiques admissibles : le taquin



- $h_1(n)$ = le nombre de pièces mal placées
  $\rightarrow\ h_1(S) = 8$
- $h_2(n)$ = la distance de Manhattan totale (la distance de chaque pièce entre sa place actuelle et sa position finale en nombre de places)

Exemple d'heuristiques admissibles : le taquin



- $h_1(n) =$ le nombre de pièces mal placées

  $\rightarrow\ h_1(S) = 8$

- $h_2(n) =$ la distance de Manhattan totale (la distance de chaque pièce entre sa place actuelle et sa position finale en nombre de places)

  $\rightarrow\ h_2(S) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

Dominance

- h_1 domine h_2 si h_1 et h_2 sont admissibles et que $h_1(n) \geq h_2(n)$ pour tout n
- h_1 est alors meilleure pour la recherche
- Exemple :

$d = 12$	IDS : $3,644,035$ nœuds
	$\text{A}^*(h_1)$: 227 nœuds
	$\text{A}^*(h_2)$: 73 nœuds
$d = 24$	IDS : trop de nœuds
	$\text{A}^*(h_1)$: $39,135$ nœuds
	$\text{A}^*(h_2)$: $1,641$ nœuds

Comment trouver des heuristiques admissibles ?

- Considérer une **version simplifiée du problème**
- Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
- Exemple : simplification des règles du taquin
  - une pièce peut être déplacée partout
    $\rightarrow$ $h_1(n)$ donne la plus petite solution
  - une pièce peut être déplacée vers toutes les places adjacentes
    $\rightarrow$ $h_2(n)$ donne la plus petite solution

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père
    - $\rightarrow$ permet de garder en mémoire la valeur du chemin passant par ce nœud oublié

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père

    → permet de garder en mémoire la valeur du chemin passant par ce nœud oublié

    - SMA* parcourt ce sous-arbre seulement si tous les autres chemins étudiés se sont montrés comme étant pires que celui oublié

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père
    - $\rightarrow$ permet de garder en mémoire la valeur du chemin passant par ce nœud oublié
    - SMA* parcourt ce sous-arbre seulement si tous les autres chemins étudiés se sont montrés comme étant pires que celui oublié
  - Si tous les nœuds ont la même valeur f, SMA* étend le nœud le plus récent, et oublie le plus ancien

Réduire la mémoire utilisée par A* :SMA*

- Au pire des cas, A* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A*
  - Procède comme A* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père
    - $\rightarrow$ permet de garder en mémoire la valeur du chemin passant par ce nœud oublié
    - SMA* parcourt ce sous-arbre seulement si tous les autres chemins étudiés se sont montrés comme étant pires que celui oublié
  - Si tous les nœuds ont la même valeur f, SMA* étend le nœud le plus récent, et oublie le plus ancien
  - Complet si une solution est à une profondeur inférieure à la taille de la mémoire

Réduire la mémoire utilisée par A^* :SMA*

- Au pire des cas, A^* doit mémoriser tous les nœuds
- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- SMA* : Simplified memory-bounded A^*
  - Procède comme A^* : étend les meilleurs nœud en fonction de la valeur de f, jusqu'à ce que la mémoire soit pleine
  - Eliminer le nœud ayant la plus grande valeur f, et rappatrier la valeur de ce nœud à son père
    - $\rightarrow$ permet de garder en mémoire la valeur du chemin passant par ce nœud oublié
    - SMA* parcourt ce sous-arbre seulement si tous les autres chemins étudiés se sont montrés comme étant pires que celui oublié
  - Si tous les nœuds ont la même valeur f, SMA* étend le nœud le plus récent, et oublie le plus ancien
  - Complet si une solution est à une profondeur inférieure à la taille de la mémoire
  - Optimal si d est inférieur à la taille de la mémoire

Algorithmes de recherche locale

Algorithmes de recherche locale

- Dans de nombreux problèmes d'optimisation, soit
  - La solution recherchée est juste l'état optimal (ou proche de l'optimalité) et non le chemin qui y mène ;
  - Il y a une fonction objective à optimiser ;
  - L'espace d'états est trop grand pour être enregistré.
- L'état lui-même est la solution
- **Idée** : Modifier l'état en l'améliorant au fur et à mesure
- **Espace d'états** : ensemble des configurations possible des états
- Besoin de définir une fonction qui mesure l'utilité d'un état

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
  - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
  - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
  - Le cas le plus élaboré est celui des algorithmes génétiques qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
    - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
    - Le cas le plus élaboré est celui des algorithmes génétiques qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.
- Il y a souvent une fonction objective à optimiser (maximiser ou minimiser)

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
  - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
  - Le cas le plus élaboré est celui des algorithmes génétiques qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.
- Il y a souvent une fonction objective à optimiser (maximiser ou minimiser)
  - Dans le cas de *hill-climbing*, elle permet de déterminer l'état successeur.

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
  - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
  - Le cas le plus élaboré est celui des algorithmes génétiques qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.
- Il y a souvent une fonction objective à optimiser (maximiser ou minimiser)
  - Dans le cas de *hill-climbing*, elle permet de déterminer l'état successeur.
  - Dans le cas des algorithmes génétiques, on l'appelle la fonction de *fitness*. Elle intervient dans le calcul de l'ensemble des états successeurs de l'état courant.

Algorithmes de recherche locale

- Une recherche locale garde juste certains états visités en mémoire :
  - Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
  - Le cas le plus élaboré est celui des algorithmes génétiques qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.
- Il y a souvent une fonction objective à optimiser (maximiser ou minimiser)
  - Dans le cas de *hill-climbing*, elle permet de déterminer l'état successeur.
  - Dans le cas des algorithmes génétiques, on l'appelle la fonction de *fitness*. Elle intervient dans le calcul de l'ensemble des états successeurs de l'état courant.
- En général, une recherche locale ne garantie pas de solution optimale, mais elle permet de trouver une solution acceptable rapidement

Algorithmes de recherche locale

Les algorithmes dédiés

- Ascension/descente de gradient (*hill climbing*)
- Descente de gradient stochastique
- Recuit simulé (*Simulated annealing*)
- Recherche en faisceau (*Beam search*)
- Recherche en faisceau stochastique
- Algorithmes génétiques

Algorithmes de recherche locale

Ascension du gradient

Algorithme d'ascension du gradient (Hill Climbing)

- Le nœud courant est initialisé à l'état initial.
- Itérativement, le nœud courant est comparé à ses successeurs immédiats.
- Le meilleur voisin immédiat et ayant la plus grande valeur que le nœud courant, devient le nœud courant
- Si un tel voisin n'existe pas, on arrête et on retourne le nœud courant comme solution.

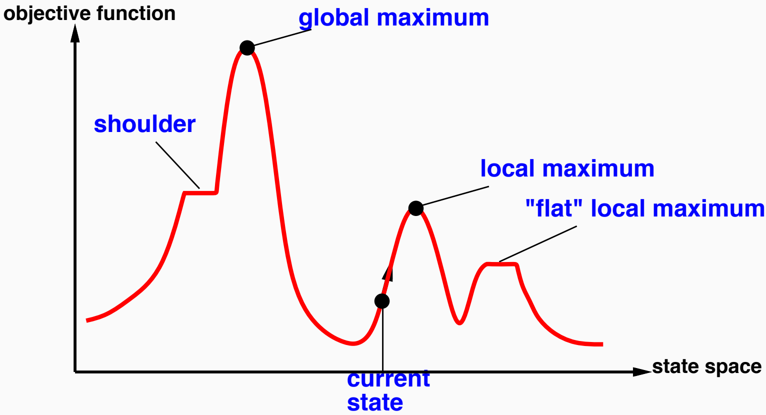
Algorithme d'ascension du gradient (Hill Climbing)

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

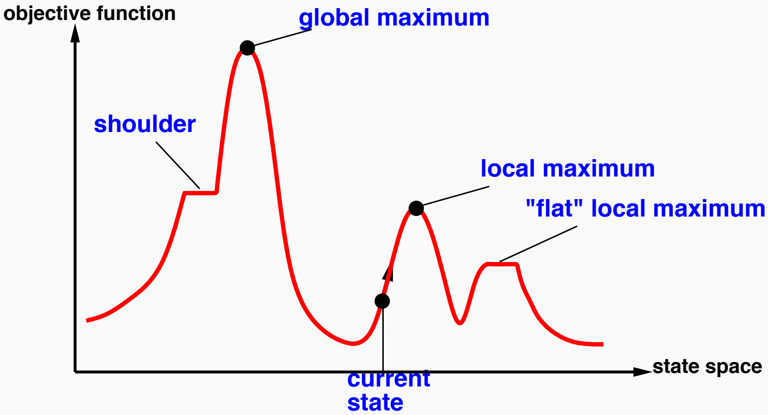
Algorithme d'ascension du gradient (Hill Climbing)

On cherche un maximum global



Algorithme d'ascension du gradient (Hill Climbing)

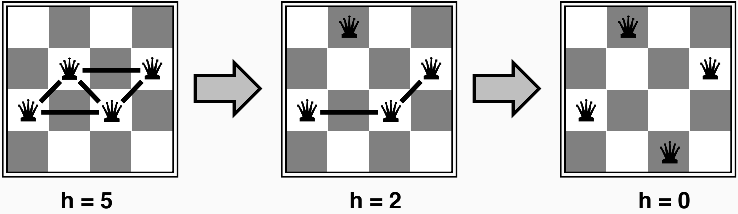
On cherche un maximum global



Comme monter l'Everest dans un épais brouillard, en étant amnésique

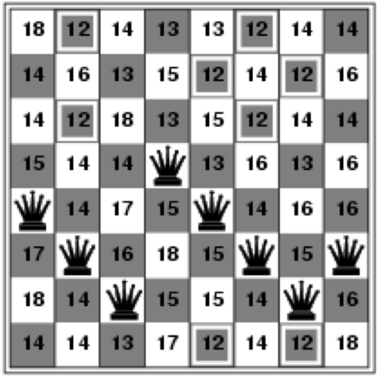
Exemple : les n reines

- Placer n reines sur un plateau de taille $n \times n$, sans que deux reines se trouvent sur la même ligne, colonne ou diagonale
- Déplacer une reine pour réduire le nombre de conflits



Exemple : les n reines

- VALUE (ou h) : nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement
- On cherche à minimiser cette valeur
- Etat actuel : $h = 17$
- Les chiffres dans chaque case indiquent le nombre de cas de conflits en mettant la reine de la colonne sur cette case
- Encadrés : les meilleurs successeurs



Ascension du gradient

- On peut aussi considérer la descente du gradient
- On peut être bloqué dans un maximum (ou un minimum) local, ou sur un plateau
- Solution : on admet des mouvements de côté

Algorithmes de recherche locale

Recuit similé

Recuit similé (simulated annealing)

- Amélioration de l'algorithme hill-climbing pour minimiser le risque d'être piégé dans des maxima/minima locaux
- Au lieu de regarder le meilleur voisin immédiat du nœud courant, avec une certaine probabilité on va regarder un moins bon voisin immédiat
- On espère ainsi s'échapper des optima locaux
- La probabilité de prendre un moins bon voisin diminue graduellement

Recuit similé (simulated annealing)

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

Algorithmes de recherche locale

Recherche taboue

Tabu Search

- L'algorithme de *recuit simulé* minimise le risque d'être piégé dans des optima locaux
- Mais il n'élimine pas la possibilité d'osciller indéfiniment en revenant à un nœud antérieurement visité
- **Idée** : On pourrait enregistrer les nœuds visités
    - Impraticable si l'espace d'états est trop grand
- L'algorithme Tabu Search (recherche taboue) enregistre seulement les k **derniers nœuds visités**
    - L'ensemble Tabou est l'ensemble contenant les k nœuds
    - Le paramètre k est choisi empiriquement
    - Cela n'élimine pas les oscillations, mais les réduit

Algorithmes de recherche locale

Local beam search

Beam Search

- **Idée** : plutôt que maintenir un seul nœud solution n, en pourrait maintenir **un ensemble de k nœuds** différents
  - Ensemble de k nœuds choisis initialement aléatoirement
  - A chaque itération, tous les successeurs des k nœuds sont générés
  - On choisit les k meilleurs parmi ces nœuds et on recommence
- Cet algorithme est appelé Local Beam Search (exploration locale par faisceau)
  - A ne pas confondre avec *Tabu Search*
  - Stochastic Beam Search : plutôt que prendre les k meilleurs, on assigne une probabilité de choisir chaque nœud, même s'il n'est pas parmi les k meilleurs (comme dans *Simulated Annealing*)

Algorithmes de recherche locale

Algorithme génétique

Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))

Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences

Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences
3. Créer une nouvelle population de taille n

Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences
3. Créer une nouvelle population de taille n
   3.1 Croisement : Sélection de 2 séquences parents (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité

Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences
3. Créer une nouvelle population de taille n
    3.1 Croisement : Sélection de 2 séquences parents (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité
    3.2 Mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort.

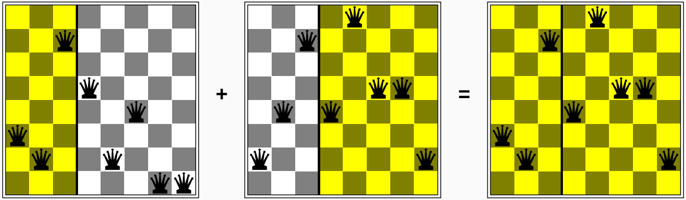
Algorithme génétique

1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences
3. Créer une nouvelle population de taille n
    3.1 Croisement : Sélection de 2 séquences parents (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité
    3.2 Mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort.
    3.3 Recommencer jusqu'à avoir une population de taille n

Algorithme génétique

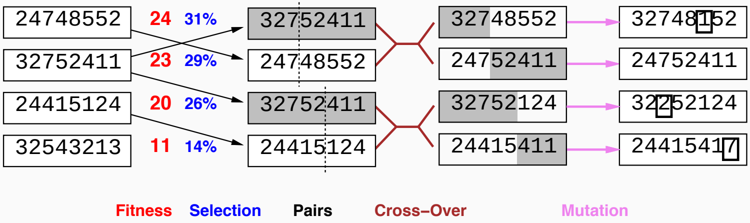
1. Génération aléatoire de n de séquences de bits (la **population** initiale (appelée aussi *soupe*))
2. Mesure de l'adaptation (*fitness*) de chacune des séquences
3. Créer une nouvelle population de taille n
   3.1 Croisement : Sélection de 2 séquences parents (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité
   3.2 Mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort.
   3.3 Recommencer jusqu'à avoir une population de taille n
4. Si la population satisfait le critère d'arrêt, arrêter. Sinon, retour à l'étape 2.

Croisement : Exemple avec 8 reines



$$\mathbf{672}47588 + 752\mathbf{51447} = \mathbf{67251447}$$

Algorithme génétique : Exemple avec 8 reines



- Fonction de fitness : nombre de paires de reines qui ne s'attaquent pas ($min = 0$, $max = (8x7)/2 = 28$)
- Pourcentage de fitness (c-à-d., probabilité de sélection de la séquence) :

$$24/(24+23+20+11) = 31\%$$
$$23/(24+23+20+11) = 29\%$$
$$20/(24+23+20+11) = 26\%$$
$$11/(24+23+20+11) = 14\%$$