Intelligence Artificielle Heuristique

Bruno Bouzy

http://web.mi.parisdescartes.fr/~bouzy
bruno.bouzy@parisdescartes.fr

Licence 3 Informatique

UFR Mathématiques et Informatique
Université Paris Descartes

Algorithmes et recherches heuristiques

- Recherche meilleur d'abord
- Recherche gloutonne
- L'algorithme A*
- Algorithmes de recherche locale

Algorithmes et recherches heuristiques

- Recherche meilleur d'abord
- Recherche gloutonne
- L'algorithme A*
- Algorithmes de recherche locale

Recherche meilleur d'abord

- **Rappel** : Une stratégie est définie en choisissant un ordre dans lequel les états sont développés
- **Idée** : Utiliser une fonction d'évaluation f pour chaque noeud
  - → mesure l'utilité d'un noeud
  - → introduction d'une fonction heuristique $h(n)$ qui **estime** le coût du chemin le plus court pour se rendre au but
- `InsertAll` insère le nœud par ordre décroissant d'utilité
- **Cas spéciaux** :
  - Recherche gloutonne (un choix n'est jamais remis en cause)
  - A*

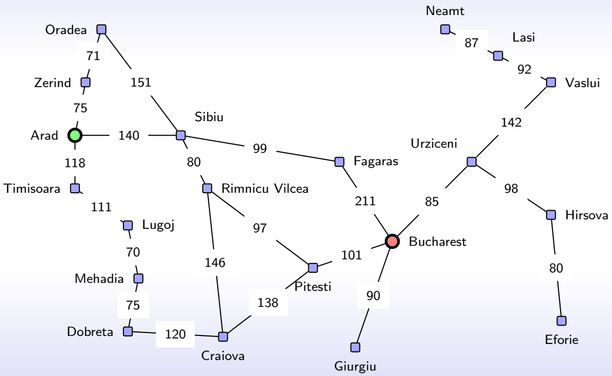
Algorithmes et recherches heuristiques

- Recherche meilleur d'abord
- Recherche gloutonne
- L'algorithme A*
- Algorithmes de recherche locale

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

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

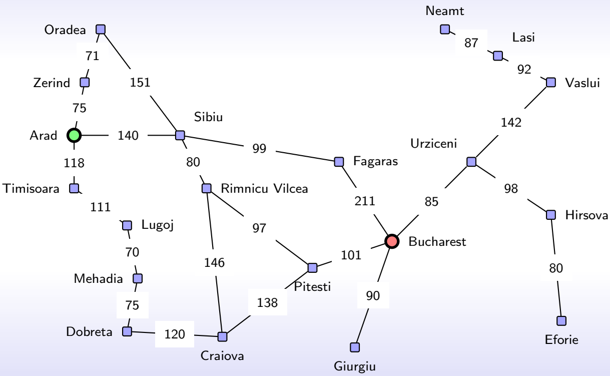
Algorithmes et recherches heuristiques

- Recherche meilleur d'abord
- Recherche gloutonne
- L'algorithme A*
- Algorithmes de recherche locale

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**

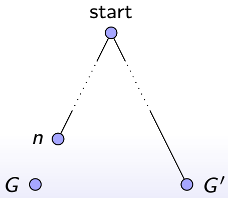
Le voyage en Roumanie



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

$$\begin{array}{ll} n & g = 4,\ h = 8,\ f = 12 \\ \downarrow & \\ p & g = 5,\ h = 4,\ f = 9 \end{array}$$

- 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

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

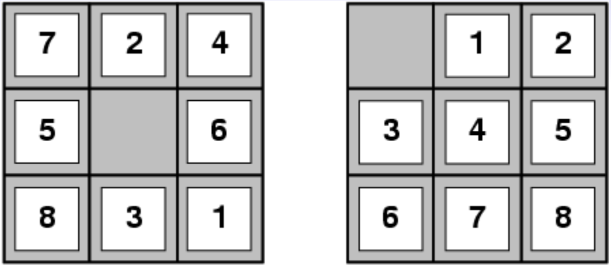
$$\begin{array}{ll} n & g = 4,\ h = 8,\ f = 12 \\ \downarrow & \\ p & g = 5,\ h = 4,\ f = 9 \end{array}$$

- 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



Etat initial — Etat final

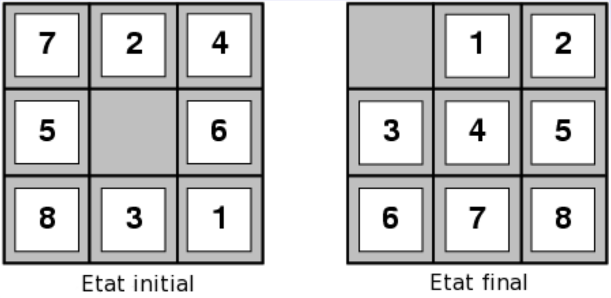
- $h_1(n)$ = le nombre de pièces mal placées

  $\rightarrow$ $h_1(S) = 8$

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

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

Exemple d'heuristiques admissibles: le taquin



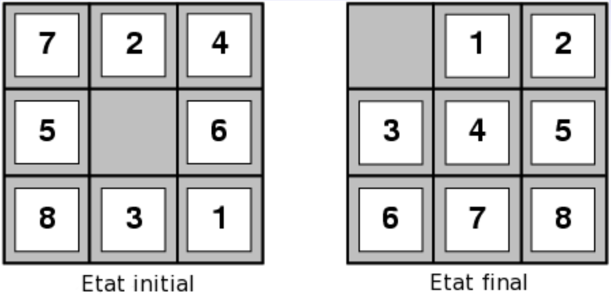
- $h_1(n)$ = le nombre de pièces mal placées

  $\rightarrow h_1(S) = 8$

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

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

Exemple d'heuristiques admissibles: le taquin



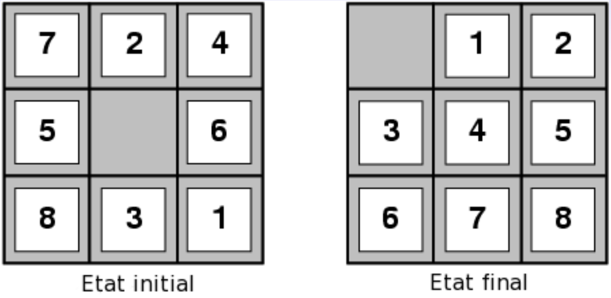
- $h_1(n)$ = le nombre de pièces mal placées

  $\rightarrow$ $h_1(S) = 8$

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

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

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
	A$^*(h_1)$: 227 nœuds
	A$^*(h_2)$: 73 nœuds
$d = 24$	IDS: trop de nœuds
	A$^*(h_1)$: $39,135$ nœuds
	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

Algorithmes et recherches heuristiques

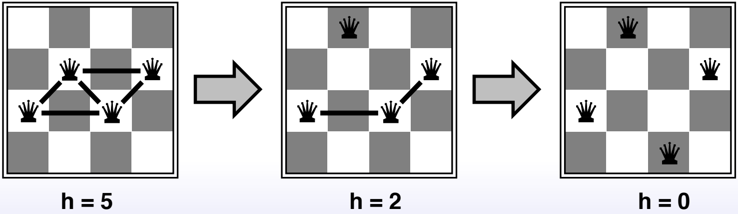
- Recherche meilleur d'abord
- Recherche gloutonne
- L'algorithme A*
- Algorithmes de recherche locale

Algorithmes de recherche locale

- Dans de nombreux problèmes d'optimisation, le chemin qui mène vers une solution n'est pas important
- 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

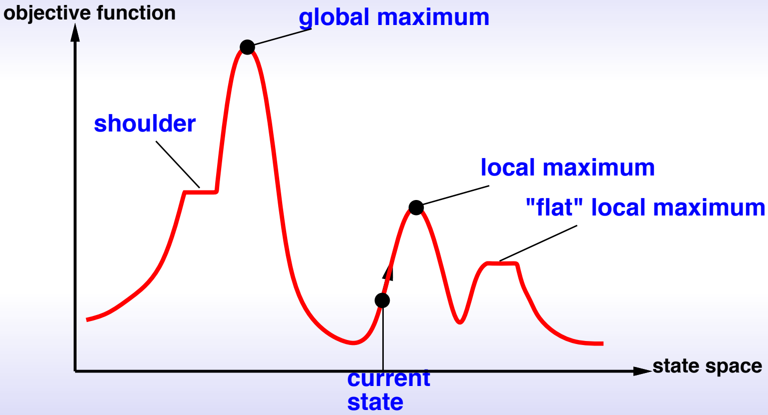
Exemple : les n reines

- 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



Algorithmes de recherche locale

On cherche un maximum global



Algorithmes de recherche locale

Algorithme d'ascension du gradient:

```
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
```

Algorithmes de recherche locale

- On peut aussi considérer la descente du gradient
- On peut être bloqué dans un maximum local
- Problème : les plateaux
- Solution : on admet des mouvements de côté