Algorithmes de recherche informés

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

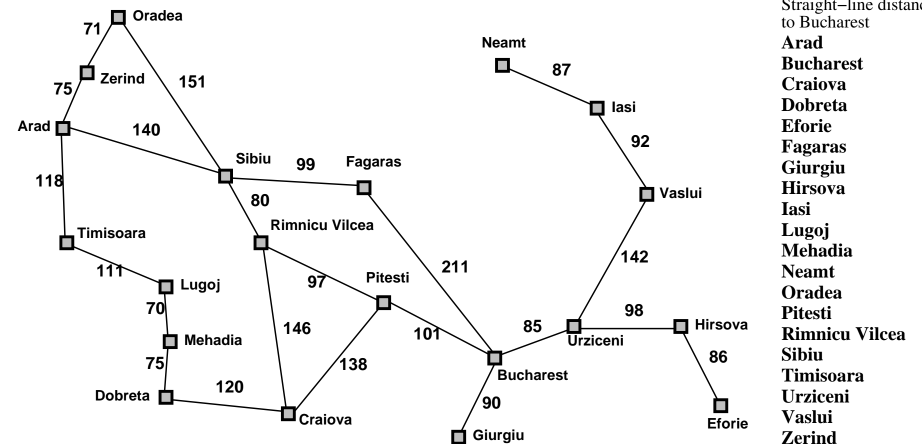
Recherche meilleur d'abord

- Rappel: Une stratégie est définie en choisissant un ordre dans lequel les états sont développés.
- Idée: Utiliser une fonction d'évaluation pour chaque noeud
  - mesure l'utilité d'un noeud
- QUEING-FN: insérer le noeud en ordre décroissant d'utilité
- Cas spéciaux:
  - Recherche gloutonne
  - A*

Recherche meilleur d'abord

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```


Roumanie avec coûts de chaque pas



Recherche gloutonne

- Fonction d'évaluation (heuristique) $h(n)$
- $h(n) =$ estimation du coût de n vers l'état final
- Par exemple $h_{dd}(n) =$ distance directe entre la ville n et Bucharest.
- La recherche gloutonne développe le noeud qui paraît être le plus proche de l'état final.
-
```
function GREEDY-SEARCH(problem) returns a solution or failure
   return BEST-FIRST-SEARCH(problem,h)
```

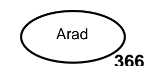

Recherche gloutonne

- Incomplet (peut aller dans des boucles)
  - complet si on ajoute test d'état répété.
- Temps: $O(b^m)$ (bonne heuristique peut améliorer beaucoup)
- Espace: $O(b^m)$
- non optimale

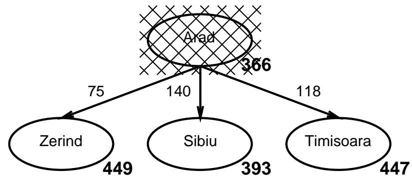
L'algorithme A*

- Idée: Éviter de développer des chemins qui sont déjà chers
- Fonction d'évaluation: $f(n) = g(n) + h(n)$
  - $g(n) =$ coût jusqu'à présent pour atteindre n
  - $h(n) =$ coût estimé pour aller vers un état final
  - $f(n) =$ coût total estimé pour aller vers un état final en passant par n
- A* utilise une heuristique admissible:
  $h(n) \leq h^*(n)$ où $h^*(n)$ est le vrai coût pour aller de n vers un état final.
- Par exemple h_{dd} ne sur-estime jamais la vrai distance.
- Théorème: A* est optimale.

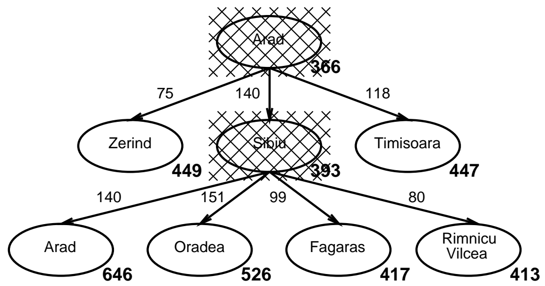
Exemple: A*



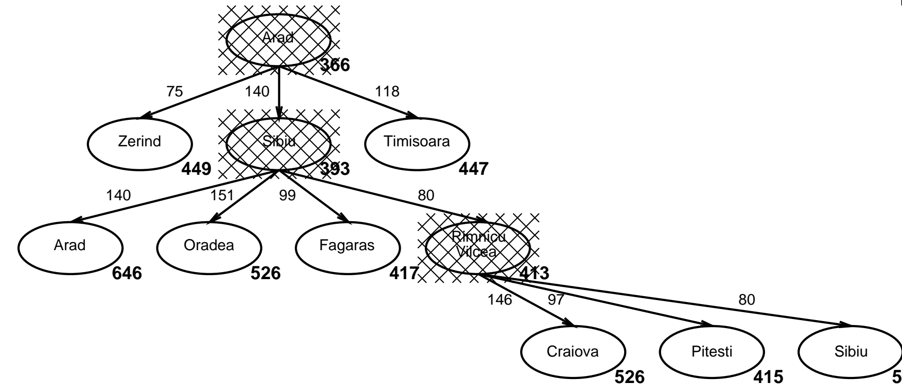
Exemple: A^*



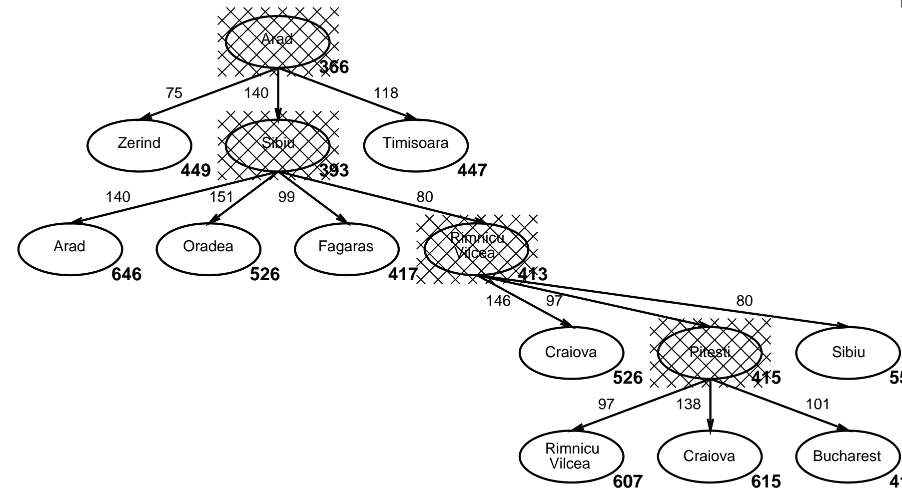
Exemple: A^*



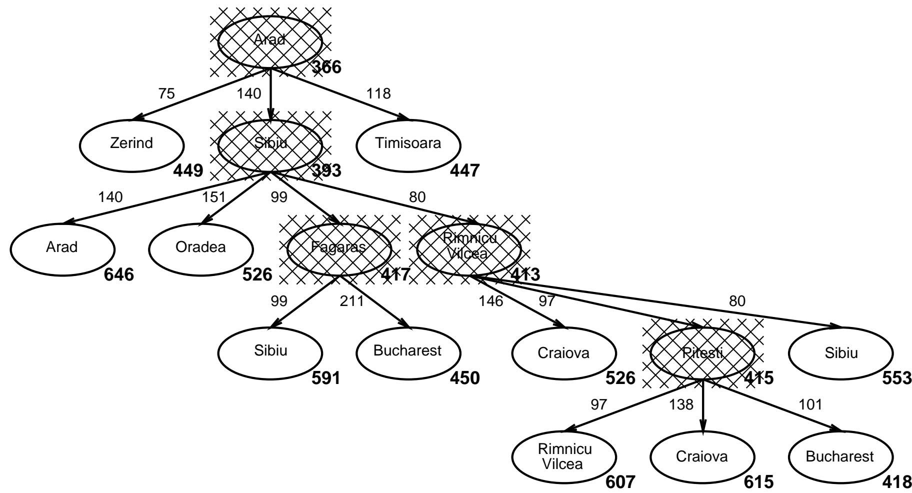
Exemple: A^*



Exemple: A^*



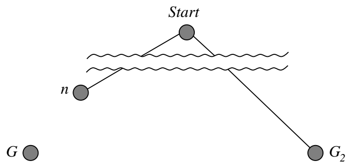
Exemple: A*



Preuve que A* est optimal

- Supposons qu'il y a un état final non optimal G_2 généré dans la liste des noeuds à traiter.

- Soit n un noeud non développé sur le chemin le plus court vers un état final optimal G.

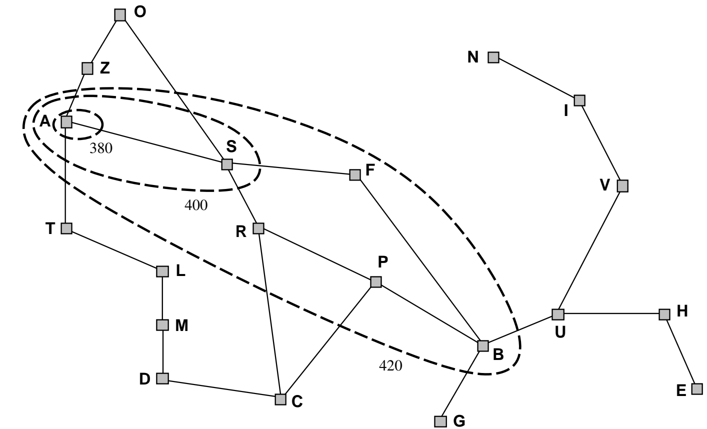


$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{car } h(G_2) = 0 \\
&> g(G) && \text{car } G_2 \text{ n'est pas optimale} \\
&\geq f(n) && \text{car } h \text{ est admissible}
\end{aligned}
$$

$f(G_2) > f(n)$, donc A* ne va pas choisir G_2.

Autre explication

Supposons que A* développe les noeuds dans l'ordre de f croissant. On ajoute pas à pas des f-régions. La région i contient tous les noeuds avec $f = f_i$, où $f_i < f_{i+1}$



Analyse de l'algorithme A*

- complet, sauf s'il y a une infinité de noeuds avec $f \leq f(G)$

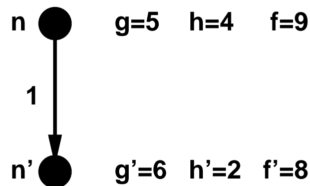
- Temps: exponentiel

- Espace: tous les noeuds !

- optimale

Quoi faire si f décroît ?

- Pour une heuristique admissible on peut avoir, que f décroît le long d'un chemin
- Par exemple, supposons que n' est successeur de n



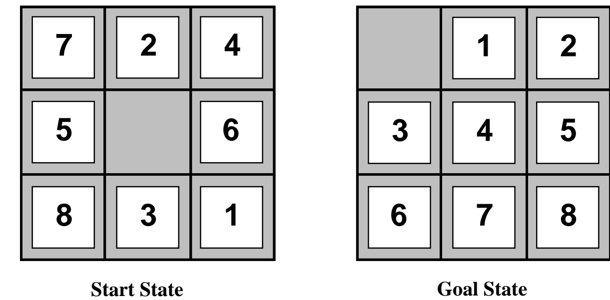
- On perd de l'information:
  - $f(n) = 9 \Rightarrow$ le vrai coût d'un chemin à travers n est ≥ 9
  - Donc le vrai coût d'un chemin à travers n' est aussi ≥ 9

Modification de A*

- Modification pathmax pour A*
- Au lieu de $f(n') = g(n')+h(n')$, on utilise $f(n') = max(g(n')+h(n'), f(n))$
- Avec pathmax, f ne décroît jamais le long d'un chemin.

Exemples d'heuristiques admissibles: 8-puzzle

- $h_1(n)$ = le nombre de pièces mal placées
- $h_2(n)$ = la distance de Manhattan globale (la distance de chaque pièce en nombre de places de sa position finale)



- $h_1(S)$ = 8, $h_2(S)$ = 3+1+2+2+2+3+3+2 = 18

Domination

Si $h_2(n) \geq h_1(n)$ pour tout n (les deux sont admissibles)
alors h_2 domine h_1 et est meilleure pour la recherche. (Pourquoi ?)

Exemple

$d = 14$	RPI = 3,473,941 noeuds
	A*(h_1) = 539 noeuds
	A*(h_2) = 113 noeuds
$d = 24$	RPI = trop de noeuds
	A*(h_1) = 39,135 noeuds
	A*(h_2) = 1,641 noeuds

Comment trouver des heuristiques ?

- Des heuristiques admissibles peuvent être obtenues en considérant le coût exact d'une version simplifiée du problème.
- Si les règles du 8-puzzle sont simplifiées de sorte qu'une pièce peut être déplacée partout, alors $h_1(n)$ donne la plus petite solution.
- Si les règles du 8-puzzle sont simplifiées de sorte qu'une pièce peut être déplacée vers chaque place adjacente, alors $h_2(n)$ donne la plus petite solution.

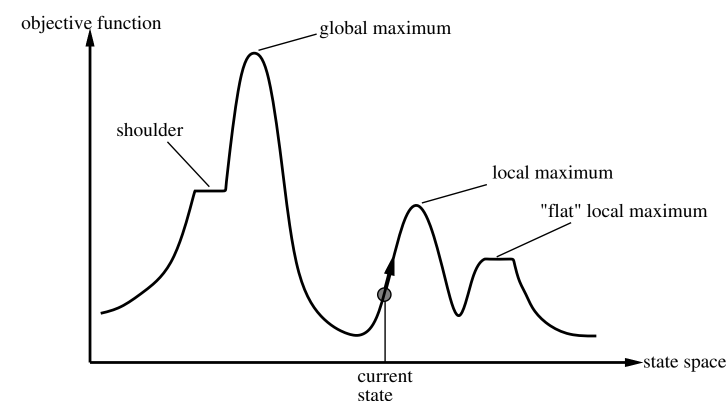
Réduire la mémoire utilisée par A*

- Au pire des cas A* doit mémoriser tous les noeuds.
- En adapte l'idée de la recherche en profondeur itérative et on obtient IDA*
- L'idée est de rechercher comme avec A* tant que la valeur de $f = g + h$ est plus petite que la valeur d'un noeud qui dépassait la limite dans l'itération précédente.
- On recherche avec une limite de plus en plus grande.
- Problèmes si les valeurs de f sont réelles.
- Il y a d'autres améliorations de A*

Algorithmes de recherche locale

- Souvent, le chemin qui mène vers une solution n'est pas important
- L'état lui-même est la solution
- utiles pour des problèmes d'optimisation
- Idée: Modifier l'état en l'améliorant au fur et à mesure
- On doit définir une fonction qui mesure l'utilité d'un état

Le paysage d'états



- On cherche un maximum global (le "meilleur" état)
- Plateau: épaule ou maximum local plat

Ascension du gradient

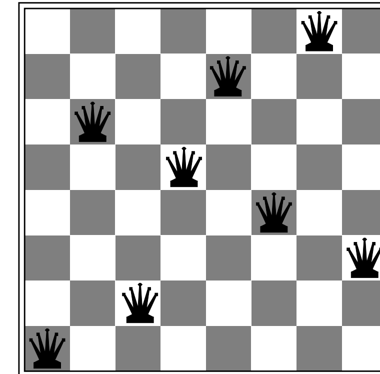
```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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


Ascension du gradient

- 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 coté.
- Variantes de Descente/Ascension du gradient:
  - Ascension stochastique
  - Ascension premier choix
  - Ascension avec reset aléatoire

Exemple: 8-reines

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18



- On choisit comme fonction d'utilité le nombre de pairs de reines qui s'attaquent et on minimise (Descente du gradient). À gauche c'est 17. Pour chaque mouvement possible les nouvelles valeur de la fonction sont indiquées. À droite la fonction est 1 (minimum local).

Recuit simulé

- Idée: fuir les maxima locaux en autorisant des changements "mauvais" mais on diminue leur fréquence

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
           schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability 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^(ΔE/T)
```

Recherche local à faisceau

- Au lieu de garder un seul état en mémoire, on en garde k
- Tous les successeurs des k états sont générées
- Si on n'a pas trouvé la solution, on prend les k meilleurs successeurs et on continue.
- Attention: Faire k recherches indépendemment est différent.
- Recherche local à faisceau stochastique
- se rapproche des algorithmes génétiques

Algorithme génétique

- Idée: s'inspirer de l'évolution naturelle pour résoudre des problèmes d'optimisation
- Ingrédients de base:
  - Une fonction de codage de la ou des données en entrée sous forme d'une séquence de bits
  - Une fonction d'utilité $U(x)$ qui donne l'adaptation d'une séquence de bits x (une valeur en entrée) donnée. Par exemple, $U(x)$ peut prendre des valeurs entre 0 et 1. Elle vaudra 1 si x est parfaitement adaptée et 0 si x est inadaptée.

L'algorithme génétique

1. Génération aléatoire d'un nombre de séquences de bits (la "soupe" initiale)
2. Mesure de l'adaptation de chacune des séquences
3. Reproduction de chaque séquence en fonction de son adaptation. Les séquences les mieux adaptées se reproduisent mieux que les séquences inadaptées. La nouvelle soupe est composée des séquences après reproduction.
4. On remplace un certain nombre de paires de séquences tirées aléatoirement par le croisement de ces paires (le lieu du croisement est choisi aléatoirement). Chaque nouvelle paire est constitué comme suit:
   - la première (seconde) séquence nouvelle est composée de la première partie de la première (seconde) séquence et de la deuxième partie de la seconde (première) séquence.
5. Mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort. Retour à l'étape 2.

Un exemple

- Soit la fonction $f(x) = 4x(1-x)$ prenant ses valeurs sur $[0, 1[$.
- On veut trouver le maximum de cette fonction
- On code les données sur 8 bits en prenant les 8 premiers bits de la représentation binaire de x
- La fonction $U(x)$ est donné par $f(x)$

Exemple

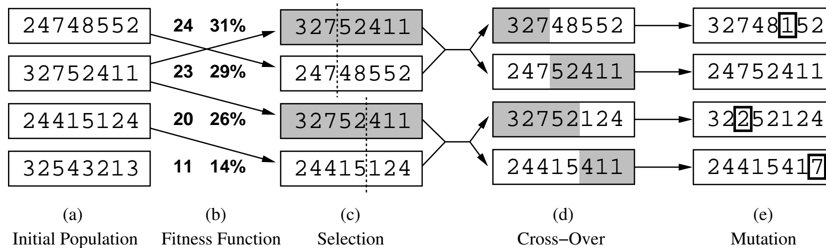
On applique l'algorithme génétique:

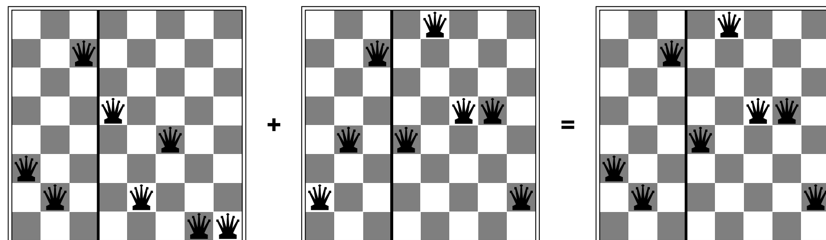
Séquence	Valeur	$U(x)$	chance de reproduction	Après reproduction
10111010	0.7265625	0.794678	0.31	11011110
11011110	0.8671875	0.460693	0.18	10111010
00011010	0.1015625	0.364990	0.14	01101100
01101100	0.4218750	0.975586	0.37	01101100

On choisi au hasard séquence 1 et 3 et la position 5 pour le croisement. On obtient la nouvelle population:

Après croisement	Valeur	U(x)
11011100	0.8593750	0.483398
10111010	0.7265625	0.794678
01101110	0.4296875	0.980225
01101100	0.4218750	0.975586

Exemple: 8-reines



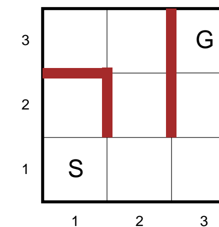


Algorithmes de recherche "online"

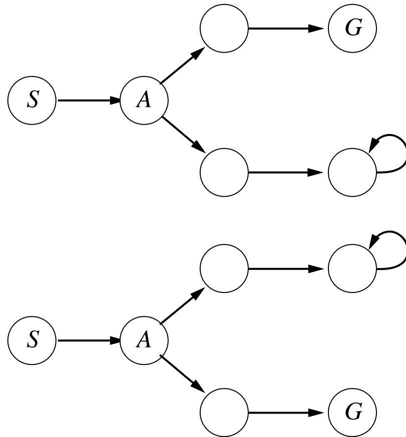
- Les algorithmes présentés jusqu'à présent sont "offline":
  - Les agents cherchent une solution avant de faire une action
  - Ils exécutent la solution (une séquence d'action) sans utiliser leur perception
- recherche "online":
  - entrelacement d'action et calcul
  - utile pour des environnements dynamiques ou semi-dynamiques
  - très utile pour des environnements stochastiques
  - nécessaire pour le problème d'exploration
    - ∗ un robot qui est mis dans un environnement inconnu et doit construire une carte pour savoir comment aller de A à B.

Problème de recherche online

- résolu par un agent qui fait des actions (déterministes) et connaît (sait)
  - la liste d'actions autorisées dans un état
  - le coût d'une action a posteriori
  - si le but est atteint
  - s'il a déjà visité un état
- Robot qui est dans un labyrinthe:

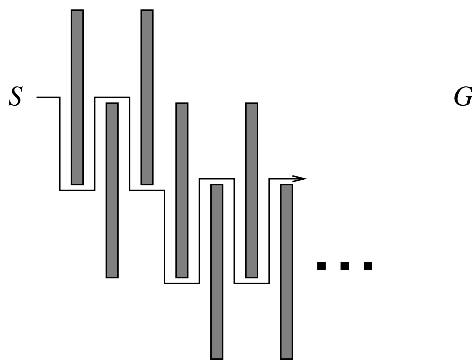


Recherche online



- Impasses (Il n'y en a pas dans les espaces explorables)

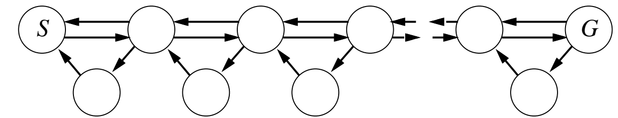
Recherche online



- Algorithmes sont nécessairement inefficaces dans certaines situations

Recherche online

- Recherche en profondeur d'abord
- Ascension du gradient
- Exploration aléatoire



- Ascension du gradient avec mémoire (LRTA*)

LRTA*

- On utilise une heuristique h (estimant le coût de l'état jusqu'à un état final) qui change au fur et à mesure de l'exploration par l'agent: Par exemple, dans l'étape b on change la valeur d'un état de 2 vers 3, puisque pour aller à un état final, il faut passer par un état qui "coûte" déjà 2.