Quatrième partie IV

Algorithmes classiques de recherche heuristique

En bref ...

Recherche meilleur d'abord

L'algorithme A*

Algorithmes de recherche locale

Algorithmes génétiques

Algorithmes en "online"

Plan

Recherche meilleur d'abord

Recherche meilleur d'abord

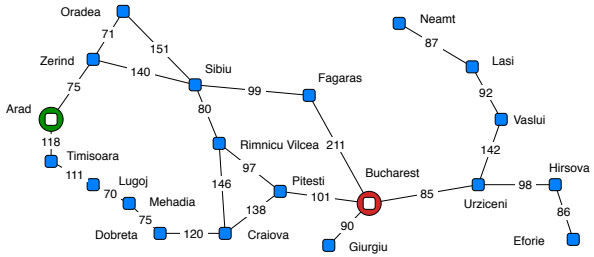
- Rappel : Une stratégie est définie en choisissant un ordre dans lequel les états sont développés.
- Idée : Utiliser une fonction d'évaluation pour chaque nœud
  - mesure l'utilité d'un nœud
- INSERT-FN() insérer le nœud en ordre décroissant d'utilité
- Cas spéciaux :
  - Recherche gloutonne (un choix n'est jamais remis en cause)
  - A*

Recherche meilleur d'abord

Algorithme

funtion BEST-FIRST-SEARCH(*problem*,EVAL-FN) **returns** *a solution sequence*
  input: *problem a problem*
    EVAL-FN, *an evaluation function*
  INSERT-FN ← *a function that orders nodes by* EVAL-FN
  return GENERAL-SEARCH(*problem*,INSERT-FN)

Le voyage en Roumanie



Straight-line distance to Bucharest (km)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Luhoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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 nœud qui paraît être le plus proche de l'état final

Algorithme

funtion GREEDY-SEACRH(*problem*) **returns** *a solution or failure*
  return BEST-FIRST-SEARCH(*problem*, h)

Recherche gloutonne

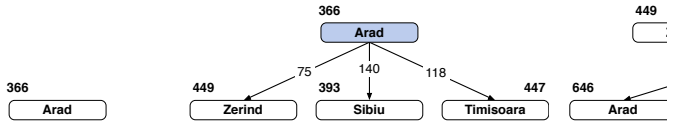
- Complétude : incomplet (peut rester bloquer dans des boucles)
  - complet si on ajoute un test pour éviter les états répétés
- Temps : $O(b^m)$
  - une bonne heuristique peut améliorer grandement les performances
- Espace : $O(bm)$
- Optimalité : non optimale

L'algorithme A*

L'algorithme A*

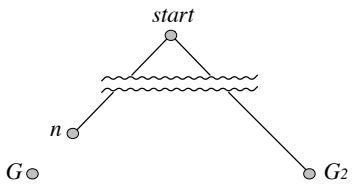
- Idée : Éviter de développer des chemins qui sont déjà chers
- Fonction d'évaluation : $f(n) = g(n) + h(n)$
  - $g(n) =$ coût jusqu'à présent pour atteindre n
  - $h(n) =$ coût estimé pour aller vers l'état final
  - $f(n) =$ coût total estimé pour aller vers l'état final en passant par n
- A* utilise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

L'algorithme A*



L'algorithme A*

- Supposons qu'il y a un état final non optimal G_2 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

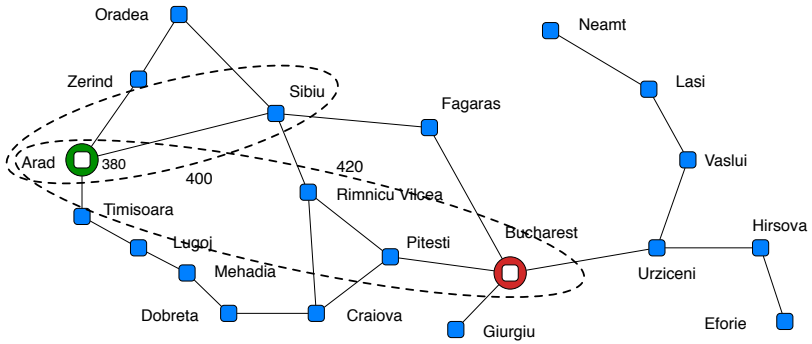


$$\begin{aligned} f(G_2) &= g(G_2) \text{ car } h(G_2) = 0 \\ &> g(G) \text{ car } G_2 \text{ n'est pas optimale} \\ &\geq f(n) \text{ car } h \text{ est admissible} \end{aligned}$$

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

L'algorithme A*

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

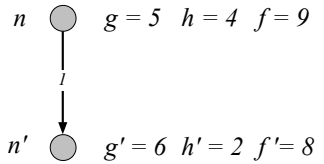


L'algorithme A*

- Complétude : complet sauf s'il y a une infinité de nœuds avec $f \leq f(G)$
- Temps : exponentiel
- Espace : tous les nœuds !
- Optimalité : optimale

L'algorithme A*

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



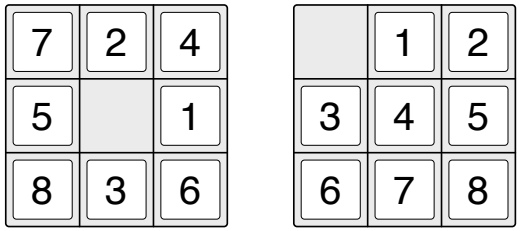
- On perd de l'information
  - $f(n) = 9 \Rightarrow$ le vrai coût d'un chemin à travers n est ≥ 9
  - Donc le vrai coût d'un chemin à travers n' est aussi ≥ 9

L'algorithme 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

L'algorithme A*

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



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

L'algorithme A*

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

$d = 14$ IDS $= 3,473,971$ nœuds
A*(h_1) $= 539$ nœuds
A*(h_2) $= 113$ nœuds
$d = 24$ IDS $=$ trop de nœuds
A*(h_1) $= 39{,}136$ nœuds
A*(h_2) $= 1{,}641$ nœuds

L'algorithme A*

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

L'algorithme A*

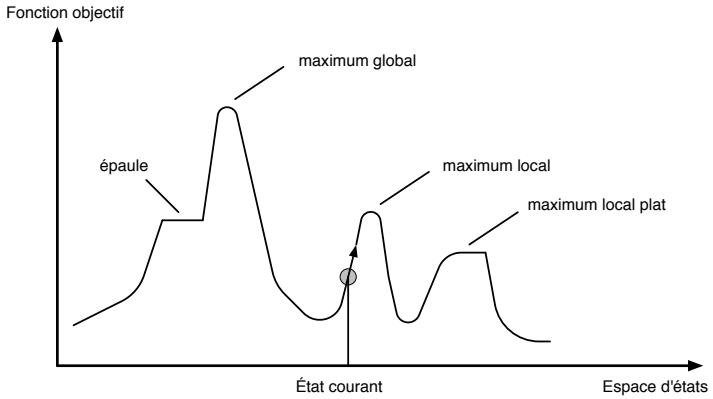
- Au pire des cas A* doit mémoriser tous les nœuds
- En adaptant l'idée de la recherche en profondeur itérative et on obtient IDA*
- L'idée est de rechercher comme avec A* tant que la valeur de $f = g + h$ est plus petite que la valeur d'un nœud qui dépassait la limite dans l'itération précédente
- On 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*, e.g.,
  - MA* – *Memory Bound A**
  - SMA* – *Simplified Memory Bound A**
  - Weighted-A*

Algorithmes de recherche locale

Algorithmes de recherche locale

- Souvent, le chemin qui méne vers une solution n'est pas important
- L'état lui-même est la solution
- Utile 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

Algorithmes de recherche locale



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

Algorithmes de recherche locale

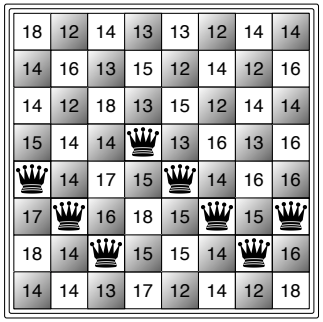
Algorithme

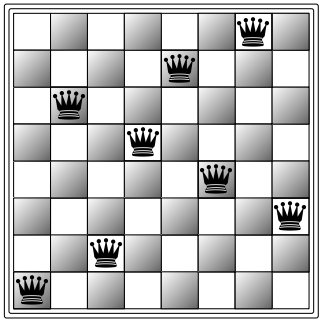
funtion HILL-CLIMBING(*problem*) **returns** *a solution state*
- **input** : *problem a problem*
- **static**: *current a node*
  next a node
- *current* ← MAKE-NODE(INITIAL-STATE[*problem*])
- **loop do**
  - *next* ← *a highest-value successor of current*
  - **if** VALUE[*next*] < VALUE[*current*] **then return** *current*
  - *current* ← *next*

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
- Variantes de Descente/Ascension du gradient :
  - Ascension stochastique : choisir aléatoirement un nœud parmi les nœuds qui améliore h
  - Ascension premier choix : choisir aléatoirement le meilleur nœud qui améliore h
  - Ascension avec reset aléatoire : lance plusieurs séries de recherche avec génération aléatoire de l'état initial

Algorithmes de recherche locale





- 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 valeurs de la fonction sont indiquées. À droite la fonction est 1 (minimum local).

Algorithmes de recherche locale

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

Algorithme

funtion SIMULATED-ANNEALING(*problem*, *schedule*) **returns** *a solution state*
- **input** : *problem, a problem ; schedule, a mapping from time to "temperatue"*
- **static**: *current, a node ; next, a node ;*
  T, a "temperature" controlling the probability of downward steps
- *current* ← MAKE-NODE(INITIAL-STATE[*problem*])
- **for** $t \leftarrow 1$ *to* ∞ **do**
  - $T \leftarrow schedule[t]$
  - **if** $T = 0$ **then return** *current*
  - *next* ← *a randomly selected successor of current*
  - $\Delta E \leftarrow$ VALUE[*next*] − VALUE[*current*]
  - **if** $\Delta E > 0$ **then** *current* ← *next*
  - **else** *current* ← *next only with probability* $e^{\Delta E / T}$

Algorithmes de recherche locale

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

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.

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 constituée comme suit :
   - la première (seconde) séquence nouvelle est composée de la première partie de la premiére (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.

Algorithme génétique

- 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 donnée par $f(x)$

Algorithme génétique

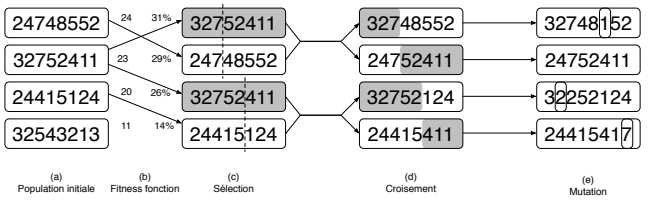
- On applique l'algorithme génétique :

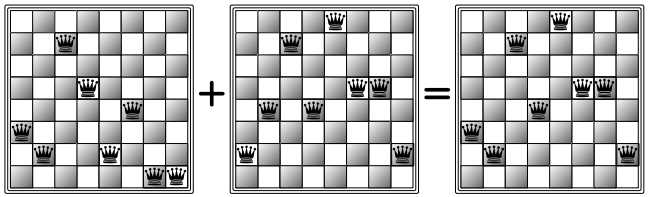
Séquences	Valeurs	$U(x)$	Chance de reproduction	Après reproduction
10111010	0.7265625	0.794678	0.31	11011110
11011110	0.8671875	0.460693	0.18	10111010
10111010	0.1015625	0.364990	0.14	01101100
01101100	0.4218750	0.975586	0.37	01101100

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

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

Algorithme génétique





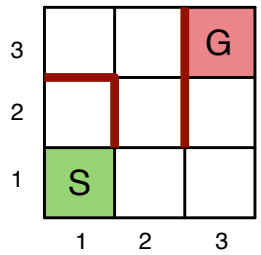
Algorithmes en "online"

Algorithmes de recherche en "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'actions) sans utiliser leurs perceptions
- Principe de la recherche en ligne :
  - entrelacement d'actions et de calculs
  - utile pour des environnements dynamiques ou semi-dynamiques
  - très utile pour des environnements stochastiques
  - nécessaire pour traiter 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.

Algorithmes de recherche en "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 :



Algorithmes de recherche en "online"

Algorithme

funtion LRTA*(s') **returns** *an action*
  input : s', *a percept that identifies the current state*
  static: *result*, *a table indexed by action and state, initially empty*
    H, *a table of cost estimates indexed by state, initially empty*
    s, a, *the previous state and action, initially null*
  if GOAL-TEST(s') **then return** *stop*
  if s' *is a new state (not in* H*)* **then** $H[s'] \leftarrow h(s')$
  if s *is not null* **then**
    $result[a, s] \leftarrow s'$
    $H[s] \leftarrow min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[b, s], H$)
  $a \leftarrow$ *an action* b *in* ACTIONS(s') *that minimizes* LRTA*-COST($s', b, result[b, s'], H$)
  $s \leftarrow s'$ **return** a

funtion LRTA*-COST(s, a, s', H) **returns** *a cost estimate*
  if s' *is undefined* **then return** $h(s)$
  else return $c(s, a, s') + H[s']$

Algorithmes de recherche en "online"

- 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.