Algorithmes de recherche heuristiques

IA – Chapitre 4a

Master STIC - Université d'Orléans - Février 2012

Aperçu

- Méthodes de recherche heuristiques
- Best-first search
- Greedy search
- A* search
- Heuristics

Méthodes de recherche heuristiques

- Les algorithmes de recherche aveugles n'exploitent aucune information concernant la structure de l'arbre de recherche ou la présence potentielle de nœuds-solution pour optimiser la recherche.
- Recherche "rustique" à travers l'espace jusqu'à trouver une solution.
- La plupart des problèmes réels sont susceptibles de provoquer une explosion combinatoire du nombre d'états possibles.
- Un algorithme de recherche heuristique utilise l'information disponible pour rendre le processus de recherche plus efficace.
- Une information heuristique est une règle ou une méthode qui presque toujours améliore le processus de recherche.

IA – Chapitre 4a Algorithmes de recherche heuristiques

Fonction heuristique

- Une fonction heuristique

$$h : E \to R$$

fait correspondre à un état $s \in E$ (espace d'états) un nombre $h(s) \in R$ qui est (généralement) une estimation du rapport coût/bénéfice qu'il y a à étendre le chemin courant en passant par s.

- Contrainte :

$$h(\text{solution}) = 0$$

- Un algorithme de recherche heuristique se base sur une fonction heuristique pour choisir le prochain nœud à explorer.

Rappel : algorithme de recherche général

```
function TREE-SEARCH( problem, fringe)  returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop  do
      if fringe is empty  then  return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem] applied to STATE(node) succeeds  return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Une stratégie est définie par le choix d'un *ordre des nœuds à explorer*

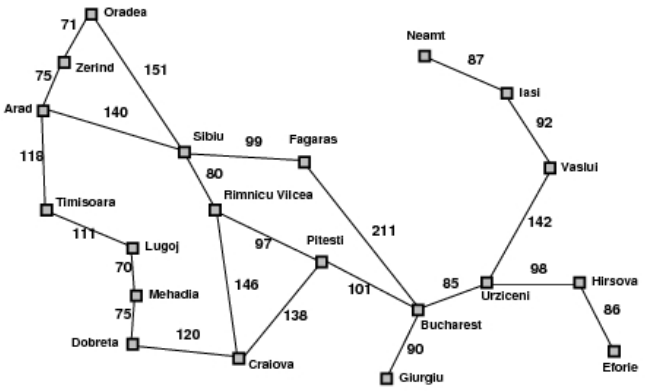
Best-first search

- Idée: utilisation d'une *fonction d'évaluation (heuristique) pour chaque nœud*
- $\Rightarrow$ *Etendre le nœud le plus prometteur selon la fonction d'évaluation*
- *Implantation:*
  les nœuds dans la file d'attente sont ordonnés dans l'ordre décroissant de leur valeur heuristique
- *Cas particuliers :*
  – *recherche glouton (greedy search)*
  – *A**

Greedy search

- Stratégie la plus simple de "best-first search"
- Fonction heuristique $h(n)$ = estimation du coût du nœud n au but
- Greedy search = minimiser le coût estimé pour atteindre le but
- Le nœud qui semble être le plus proche du but sera étendu en priorité
- Fonctions heuristiques classiques
  – distance à vol d'oiseau
  – distance Manhattan : déplacements limités aux directions verticales et horizontales

Exemple : Voyage en Roumanie

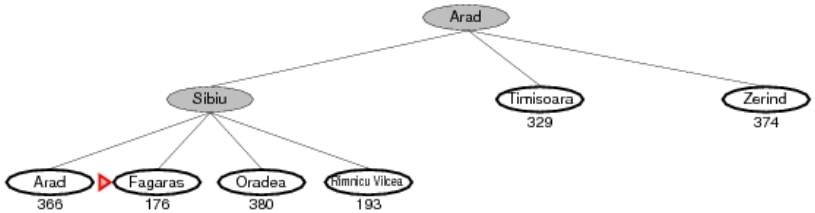


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

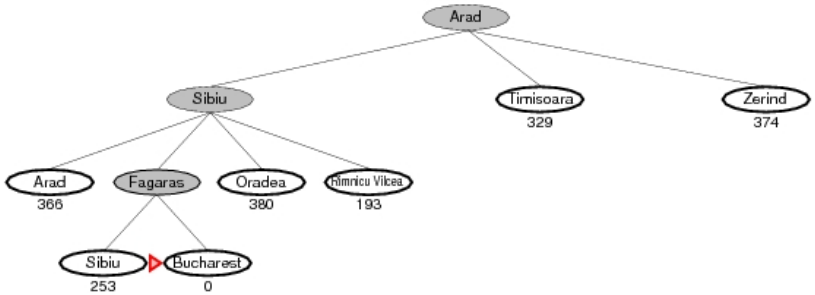
Exemple : Voyage en Roumanie

Exemple : Voyage en Roumanie

Exemple : Voyage en Roumanie



Exemple : Voyage en Roumanie



Propriétés de “greedy search”

- Complétude ?? Non–peut entrer dans une boucle infinie, par exemple, avec le but Oradea,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
  Complet dans un espace fini avec une vérification d’absence de boucle

Propriétés de “greedy search”

- Complétude ?? Non–peut entrer dans une boucle infinie, par exemple, avec le but Oradea,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
  Complet dans un espace fini avec une vérification d’absence de boucle
- Temps ?? $O(b^m)$

Propriétés de “greedy search”

- Complétude ?? Non–peut entrer dans une boucle infinie, par exemple, avec le but Oradea,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
  Complet dans un espace fini avec une vérification d’absence de boucle
- Temps ?? $O(b^m)$
- Espace ?? $O(b^m)$, il faut garder tous les nœuds dans la mémoire

Propriétés de “greedy search”

- Complétude ?? Non–peut entrer dans une boucle infinie, par exemple, avec le but Oradea,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
  Complet dans un espace fini avec une vérification d’absence de boucle
- Temps ?? $O(b^m)$
- Espace ?? $O(b^m)$, il faut garder tous les nœuds dans la mémoire
- Optimalité ?? Non

Propriétés de “greedy search”

- Complétude ?? Non–peut entrer dans une boucle infinie, par exemple, avec le but Oradea,
  Iasi → Neamt → Iasi → Neamt →
  Complet dans un espace fini avec une vérification d'absence de boucle
- Temps ?? $O(b^m)$
- Espace ?? $O(b^m)$, il faut garder tous les nœuds dans la mémoire
- Optimalité ?? Non
- Remarque : la performance de “greedy search” est en fonction de la précision de h, avec une bonne fonction heuristique, les complexités en temps et en espace peuvent être fortement réduites.

Greedy search – commentaires

- La solution
  Arad → Sibiu → Fagaras → Bucarest
  n'est pas optimale, elle est de 32km plus longue que
  Arad → Sibiu → Rimnicu → Pitesti → Bucarest
  car “greedy search” ne considère pas la distance déjà parcourue !
- Stratégie : toujours choisir le plus petit coût restant pour atteindre le but (*greedy* = gourmand), c'est-à-dire minimiser le coût estimé pour atteindre le but
- Susceptible de faux départ :
  pour aller de Iasi à Fagaras, greedy search préfère Neamt à Vaslui

Fonction heuristique admissible

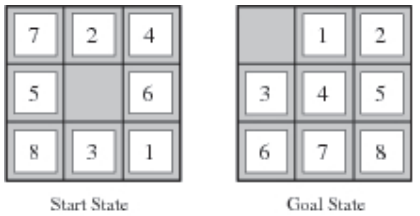
- Une fonction heuristique est admissible (qui ne surestime jamais le coût réel) si

  $\forall n, 0 \leq h(n) \leq h^*(n)$ avec $h^*(n)$ = coût optimal réel de n au but

  Une fonction heuristique admissible est toujours optimiste !
- Quels problèmes poserait une heuristique pessimiste ?

Heuristiques admissibles

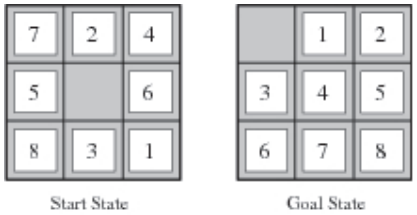
Par exemple, pour le problème de puzzle-8
$h_1(n)$ = nombre de plaquettes mal placées
$h_2(n)$ = nombre total des distances de Manhattan
(nombre de pas pour arriver à la bonne place de chaque plaquette)



$h_1(S)$ = ??
$h_2(S)$ = ??

Heuristiques admissibles

Par exemple, pour le problème de puzzle-8
$h_1(n)$ = nombre de plaquettes mal placées
$h_2(n)$ = nombre total des distances de Manhattan
(nombre de pas pour arriver à la bonne place de chaque plaquette)



$h_1(S)$ = ?? 8
$h_2(S)$ = ?? 4+0+3+3+1+0+2+1 = 14

Vers une heuristique plus efficace...

- Greedy search minimise le coût estimé $h(n)$ du nœud n au but réduisant ainsi considérablement le coût de la recherche, mais il n'est pas optimal et pas complet en général
- L'algorithme de recherche en coût uniforme minimise le coût $g(n)$ depuis l'état initial au nœud n, il est optimal et complet, mais pas très efficace
- Idée : combiner les deux algorithmes et minimiser le coût total $f(n)$ du chemin passant par le nœud n

$$f(n) = g(n) + h(n)$$

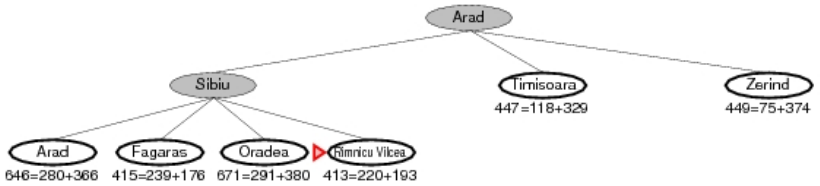
Algorithme A*

Idée: éviter de choisir le chemin qui est déjà coûteux
Fonction d'évaluation $f(n) = g(n) + h(n)$
$g(n)$ = coût du nœud initial à n
$h(n)$ = coût estimé de n au but
$f(n)$ = coût total estimé d'un chemin passant par n au but
A* utilise une heuristique admissible
Théorème: A* est optimal

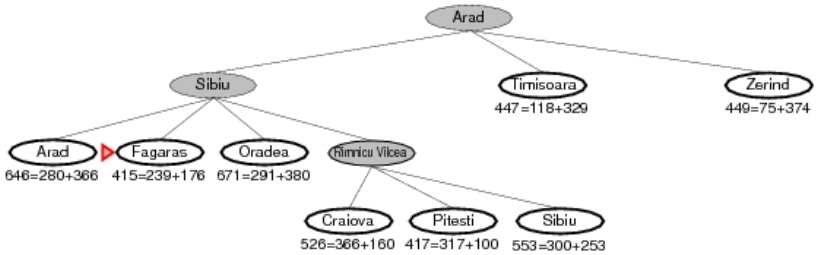
Exemple de l'algorithme A*

Exemple de l'algorithme A*

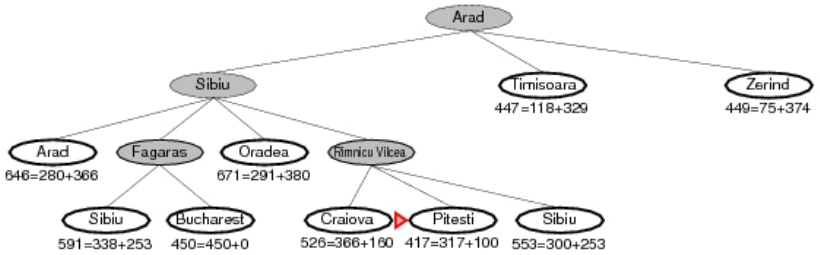
Exemple de l'algorithme A*



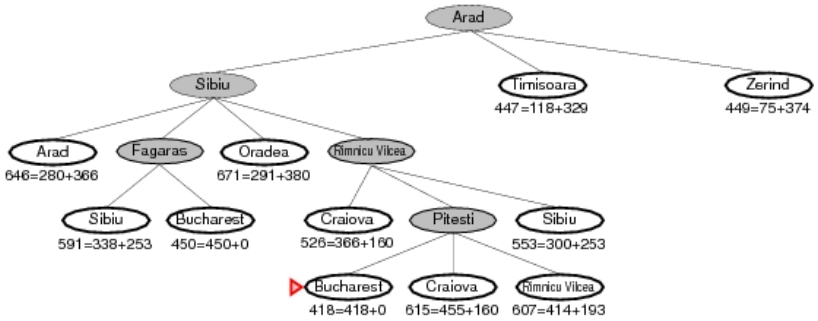
Exemple de l'algorithme A*



Exemple de l'algorithme A*

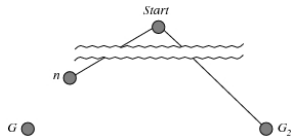


Exemple de l'algorithme A*



Preuve de l'optimalité de A*

Supposons qu'un noeud but sous-optimal G_2 soit généré et soit dans la liste d'attente. Soit n un nœud non exploré sur un chemin menant à un but optimal G.

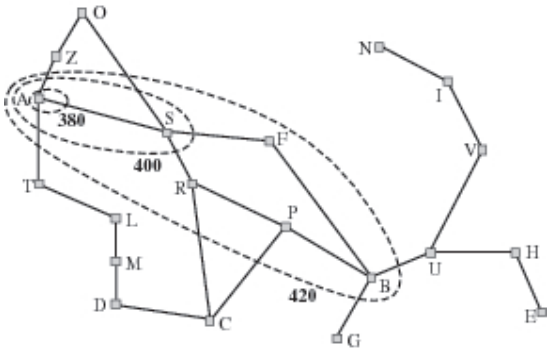


$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{car } h(G_2) = 0 \\
f(G_2) &> f(G) && \text{car } G_2 \text{ est sous optimal} \\
\text{or } f(G) &\geq f(n) && \text{car } h \text{ est admissible} \\
\text{donc } f(G_2) &> f(n)
\end{aligned}
$$

Puisque $f(G_2) > f(n)$, A* ne choisit jamais G_2.

Preuve de l'optimalité de A*

- Lemme: A* étend les nœuds par ordre croissant de valeur de f
- Les coûts sont non décroissants le long de n'importe quel chemin
- On peut définir les f-contours dans l'espace de recheche
  Le contour f_i englobe tous les noeuds tels que $f < f_i$; de plus, $f_i < f_{i+1}$



Preuve du lemme : Consistence

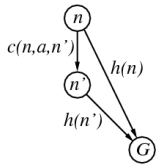
Une heuristique est consistante (monotone) si

$$h(n) \leq c(n, a, n') + h(n')$$

Avec $c(n, a, n')$ = coût de n à n' via l'action a
Si h est consistante, on a

$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n, a, n') + h(n') \\
&\geq g(n) + h(n) \\
&= f(n)
\end{aligned}
$$

Cela signifie que $f(n)$ ne décroît pas le long de chaque chemin (inégalité triangulaire).



Propriétés de A*

- Complétude ?? Oui, sauf dans les cas où il existe un nombre infini de nœuds ayant $f \leq f(G)$

Propriétés de A*

- Complétude ?? Oui, sauf dans les cas où il existe un nombre infini de nœuds ayant $f \leq f(G)$
- Temps ?? Exponentiel

Propriétés de A*

- Complétude ?? Oui, sauf dans les cas où il existe un nombre infini de nœuds ayant $f \leq f(G)$
- Temps ?? Exponentiel
- Espace ?? Exponentiel, il faut garder tous les nœuds dans la mémoire

Propriétés de A*

- Complétude ?? Oui, sauf dans les cas où il existe un nombre infini de nœuds ayant $f \leq f(G)$
- Temps ?? Exponentiel
- Espace ?? Exponentiel, il faut garder tous les nœuds dans la mémoire
- Optimalité ?? Oui, f_{i+1} n'est pas étendu tant que f_i n'est pas fini
  A* étend tous les nœuds ayant $f(n) < C^*$
  A* étend quelques nœuds ayant $f(n) = C^*$
  A* n'étend aucun nœud ayant $f(n) > C^*$

Qualité de fonction heuristique

- Facteur de branchement effectif
  Soit N le nombre total d'états produits pour obtenir la solution
  soit d la profondeur à laquelle la solution a été trouvée
  alors b^* est le facteur de branchement d'un arbre fictif parfaitement équilibré tel que

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

  exemple : si $d = 5$ et $N = 52$ alors $b^* = 1,92$
- Une bonne fonction heuristique aura une valeur de b^* proche de 1 (la fonction heuristique idéale aurait $b^* = 1$)

Dominance

Si $h_2(n) \geq h_1(n)$ pour tout n (les deux étant admissibles) alors h_2 domine h_1 et constitue une meilleure fonction heuristique
Exemple de coûts typiques de recherche (puzzle-8, 100 runs) :

$d = 14$	IDS = 3 473 941 nœuds
	$A^*(h_1)$ = 539 nœuds
	$A^*(h_2)$ = 113 nœuds
$d = 24$	IDS $\approx$ 54 000 000 000 nœuds
	$A^*(h_1)$ = 39 135 nœuds
	$A^*(h_2)$ = 1 641 nœuds

Remarque : Pour chaque fonction heuristique admissible h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

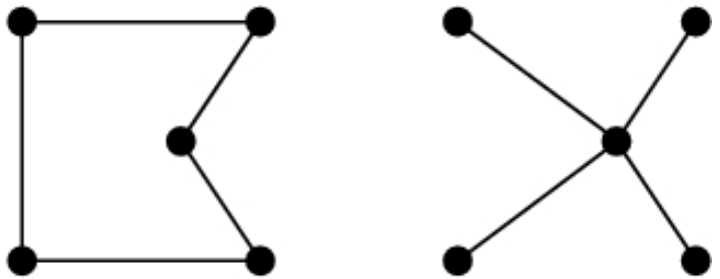
est aussi admissible et domine h_a, h_b

Problème simplifié

- Une heuristique admissible peut être dérivée à partir du coût exact de la solution d'une version "simplifiée" du problème
- Exemple : puzzle-8,
  h_1 = nombre de plaquettes mal placées,
  h_2 = somme des distances de Manhattan
  Versions simplifiées:
  – chaque plaquette peut se déplacer n'importe où, alors $h_1(n)$ donne la solution optimale
  – chaque plaquette peut se déplacer à chaque case adjacente, alors $h_2(n)$ donne la solution optimale
- Point clé : le coût optimal du problème simplifié n'est pas plus grand que le coût optimal du problème original

Exemple de problème simplifié

Exemple typique : problème de voyageur de commerce (TSP)
Trouver le tour de toutes les villes le moins coûteux pour lequel chaque ville est visitée une seule fois



L'arbre couvrant minimal calculé en $O(n^2)$ est une borne inférieure

Variantes de A*

- Les problèmes réels sont souvent très complexes
- L'espace de recherche devient très grand
- Même les méthodes de recherche heuristique deviennent inefficaces
- A* : Complexité en espace exponentielle
- Algorithmes de recherche "économes" en mémoire ?
  2 variantes de A* :
  – IDA* = A* avec approfondissement itératif
  – SMA* = A* avec gestion de mémoire

IDA*

- IDS effectue l'approfondissement par rapport à la profondeur limite de recherche ($l = 0, 1, 2, 3, \ldots$)
- IDA* utilise comme limite de l'horizon de recherche la valeur de la fonction heuristique $f(n)$
  chaque itération de IDA* est une recherche en profondeur.
  La profondeur limite correspond à la plus petite valeur de f observée au delà de la limte de l'étape précédente.
- IDA* est complet et optimal
- Complexité en espace de IDA* : $O(bd)$
- ok pour coût unitaire de étapes, mais pour des coûts réels?

Une alternative : RBFS

- Recursive BFS
- Approche best-first, mais
- Enregistre dans chaque noeud le f du meilleur chemin alternatif
- Remonte si le coût du noeud développé et supérieur à un f mémorisé
- En remontant, mémorise ce coût (pour redescendre si besoin)
- + efficace qu'IDA* mais risque de redévelopper beaucoup de noeuds
- Optimale (si h admissible), espace : O(bd), temps ?
- Pb : la faible complexité en espace est finalement gênante.

SMA*

- SMA* : Simplified Memory-Bounded A*
- SMA* effectue sa propre gestion de mémoire
- principe
  – si la mémoire (file d'attente) est pleine, alors faire de la place en éliminant le nœud le moins intéressant (celui avec une valeur f élevée)
  – retenir dans le nœud-ancêtre la valeur du meilleur descendant oublié

Propriétés de SMA*

- Il évolue dans un espace-mémoire alloué par avance
- Il évite les états multipliés
- Il est complet si l'espace de mémoire alloué est suffisant pour contenir le chemin de solution le moins profond
- Il est optimal si l'espace de mémoire alloué est suffisant pour contenir le chemin de solution optimale le moins profond
- Dans tous les cas, il retourne la meilleure solution accessible étant donné l'espace de mémoire alloué

Résumé

- Une fonction heuristique h estime le coût des solutions
- Une bonne heuristique peut réduire énormément le coût de la recherche
- "Greedy best-first search" étend le nœud n ayant $h(n)$ minimum
  – incomplet et pas toujours optimal
- L'algorithme A* étend le nœud n ayant $g(n) + h(n)$ minimum
  – complet et optimal
- Des heuristiques admissibles peuvent se déduire des solutions exactes des problèmes simplifiés

Exploration en ligne

- Jusque là, tout l'environnement est connu dès le départ
- Que faire si l'environnement évolue suivant les actions ?
- notion d'action : un successeur ne peut être accédé qu'après une action sur le parent.
- notion de ratio de compétitivité : coût du chemin parcouru / coût en offline
- notion d'espace explorable sans risque : actions réversibles

Agent DFS en ligne

Idée :

- recherche de type DFS
- A chaque étape on a une pile de successeurs / prédécesseurs
- On recherche des successeurs jusqu'à atteindre une impasse
- On revient au prédécesseur et on reprend ("récursif").

Agent DFS en ligne

```
fonction ONLINE-DFS(s')
si TEST_BUT(s') alors stop
si s' est un nouvel état alors
  inexploré[s']<-ACTIONS(s')
si s non nul alors
  résultat[a,s]<-s'
  ajouter s au début de nonrevisités[s']
si inexploré[s'] est vide alors
  si nonrevisité[s'] est vide alors stop
  sinon a<-b tq résultat[b,s']=POP(nonrevisité[s'])
sinon a<-POP(inexploré[s'])
s<-s'
retourner a
```

s et a : état et action précédents
résultat : table ; inexplorés : actions non essayées
nonrevisités : états non essayés

Exploration en ligne locale

Idée :

- Combiner A* et exploration en ligne (chapitre suivant)
- Pour chaque état on utilise une fonction heuristique.
- Celle-ci est mise à jour au cours de l'exploration suivant les fonctions heuristiques des voisins
- Pour chaque noeud non solution, on se déplace vers le voisin de plus faible heuristique.
- L'"heuristique" de l'ancien noeud devient égale à h du nouveau noeud + coût de transition.

Exploration en ligne locale

```
fonction LRTA*(s')
si TEST_BUT(s') alors stop
si s' nouvel état alors H[s']<-h(s')
si s non nul alors
  résultat[a,s]<-s'
  H[s]<- min COUT_LRTA*(s,b,résultat[b,s],H)
    (sur b dans ACTIONS(s))
a<-b dans ACTIONS(s') minimisant
   COUT_LRTA*(s',b,résultat[b,s'],H)
s<-s'
retourne a
```

H : table d'estimation des coûts
s,a état et action précédents
COUT_LRTA*(s,a,s',H) retourne h(s) si s' est indéfini,
H[s']+c(s,a,s') sinon