Introduction à l'intelligence artificielle

Wieslaw Zielonka

ITERATIVE DEEPENING DEPTH-FIRST SEARCH

limited_depth_first_search(k) — fait la recherche en profondeur jusqu'au niveau k et retourne soit une solution soit échec

```
deepening_depth_first_search()
     for k=1 to infinity
     do
          solution = limited_depth_first_search(k)
          si solution != échec alors
                              retourner solution
     done

limited_depth_first_search(k) – effectue recherche
 en profondeur mais ne dépasse
jamais le niveau k.
```

Recherche locale

Dans le problème de 8 reines nous cherchons à trouver une solution (l'état destination) et non pas le chemin qui mène vers une solution.

Même situation pour d'autres problèmes, par exemple le voyageur de commerce.

Pour ce type de problème : la **recherche locale :**

- on maintient juste le noeud courant, pas de chemin qui mène vers ce noeud,

- on se déplace vers un voisin du noeud courant

Recherche locale

- fonction de coût définie pour chaque noeud
- solution - le noeud de coût minimal
- Exemple :

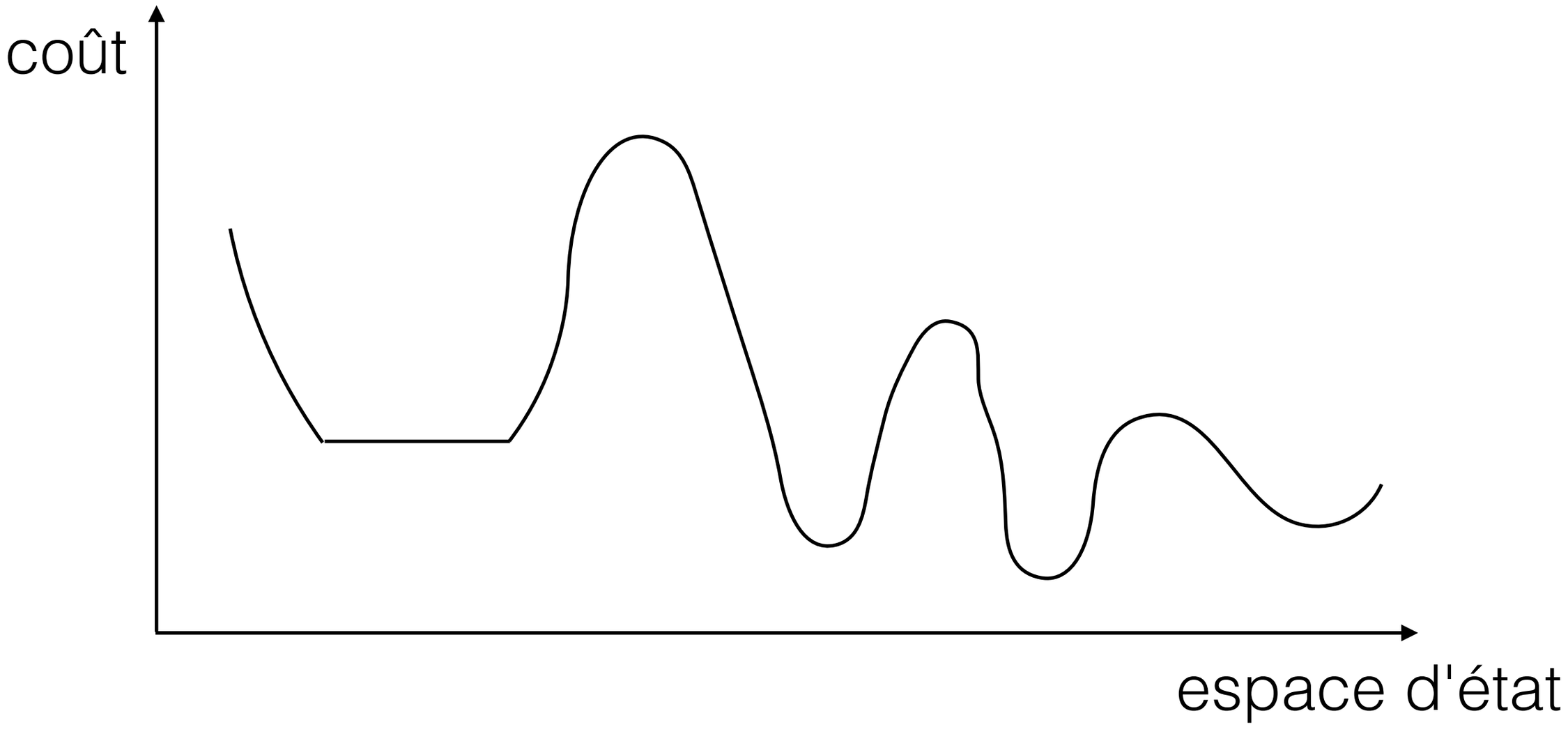
1. 8 reines, le coût d'une configuration de 8 reines = le nombre de couples de reines en conflit

2. la solution : la configuration avec le coût minimal 0

Recherche locale

Problème d'optimisation de la fonction du coût.

Difficulté : il faut trouver le minimum global



- Algorithme complet - trouve une solution si elle existe

- algorithme optimal - trouve une solution optimale

Steepest-descent (descente gradient)

```
fonction steepest-descent(problème)
                                        retourne le minimum local
courant = new noeud(problème.état_initial)
loop do
    voisin = le voisin avec le coût minimal
    if voisin.coût >= courant.coût return courant
    courant=voisin
done
```

steepest-descent appliquée à 8 reines (avec initialement 8 reines, une par ligne mais sur une colonne quelconque):

- bloqué dans 86% de cas sans qu'une solution soit trouvée
- 4 itérations en moyenne si une solution est trouvée
- 3 itérations en moyenne si une solution n'est pas trouvée et l'algorithme steepest-descent bloque
- le nombre d'états 8

Steepest descent

Steepest-descent s'arrête si on se trouve sur un plateau (pas de voisins avec le coût inférieur mais il existe des voisins avec le coût égal).

Pour éviter le blocage dans ce cas : permettre les mouvement latéraux (s'il n'y a pas de voisin de coût inférieur mais il en existe avec le coût égal au coût du courant alors déplacer le courant vers un tel voisin) .

Limiter le nombre de mouvement latéraux consécutifs (pour éviter un long parcours inutile sur un plateau).

Steepest descent

Pour 8 reines : steepest descent avec au plus 100 mouvements latéraux consécutifs trouve une solution dans 94% de cas.

Mais le nombre moyen de mouvement avant que steepest-descent trouve une solution passe à 21 et le nombre moyen de mouvement avant que l'algorithme soit bloqué (sans avoir trouvé une solution) passe à 64.

Steepest descent stochastique

A chaque étape on sélectionne un voisin avec la probabilité plus élevée pour les voisins avec le coût plus petit (la probabilité proportionnelle à 1/coût). Cela permet de passer les collines avoisinante et sortir d'un minimum local entouré de collines.

Random restart steepest descent

Si l'algorithme bloque alors redémarrer avec un nouveau état initial choisi de manière aléatoire.

Si p la probabilité de trouver une solution par l'algorithme steepest descent alors le nombre moyen de " restart" est 1/p et une solution sera trouvée avec la probabilité 1.

Pour 8 reines, p=0.14, le nombre moyen de pas dans l'algorithme random restart steppest descend =

$$\sum_{n\geq 0}(1-p)^n p(3n+4) = 3\sum(1-p)^n pn + 4\sum(1-p)^n p =$$

$$4\times 1+(\frac{1}{p}-1)\times 3\approx 22$$

Pour 8 reines avec les mouvements latéraux: p=0.94, 1/p≈1.06,

le nombre moyen de mouvements =

$$21\times 1+(\frac{1}{p}-1)\times 61\approx 25$$

Pour 1.000.000 reines random restart deepest descend trouve une solution en moins d'une minute.

Simulated annealing - recuit simulé

Ingredients :

- un ensemble fini *S* d'états
- une fonction de coût $J : S \to \mathbb{R}$
- pour chaque $i \in S$, l'ensemble $S(i) \subset S \setminus \{i\}$de voisins de i
- pour chaque $i \in S$et $j \in S(i)$ un coefficient $q_{ij} > 0$ tel que $\sum_{j \in S(i)} q_{ij} = 1$
- une fonction non-croissante $T : \mathbb{N} \to (0, \infty)$, plan de refroidissement (cooling schedule), $T(t)$ - la température au moment t
- l'état initial $x(0)$

Simulated annealing

Soit $x(t) = i$ l'état au moment t .

Choisir l'état j avec la probabilité q_{ij}

- si $J(j) \leq J(i)$ alors $x(t+1) = j$
- si $J(j) > J(i)$ alors $x(t+1) = j$ avec la probabilité $\exp[-\frac{J(j)-J(i)}{T(t)}]$ sinon $x(t+1) = i$

où $\exp(z) = e^z$.

Simulated annealing

On peut démontrer sous certaines conditions que si la température $T(t) \to 0$ quand $t \to \infty$ alors à la limite la distribution de probabilité se concentre dans les états où J prend la valeur minimale.

La fonction de refroidissement souvent utilisée est

$$T(t) = \frac{d}{\log(t)}$$

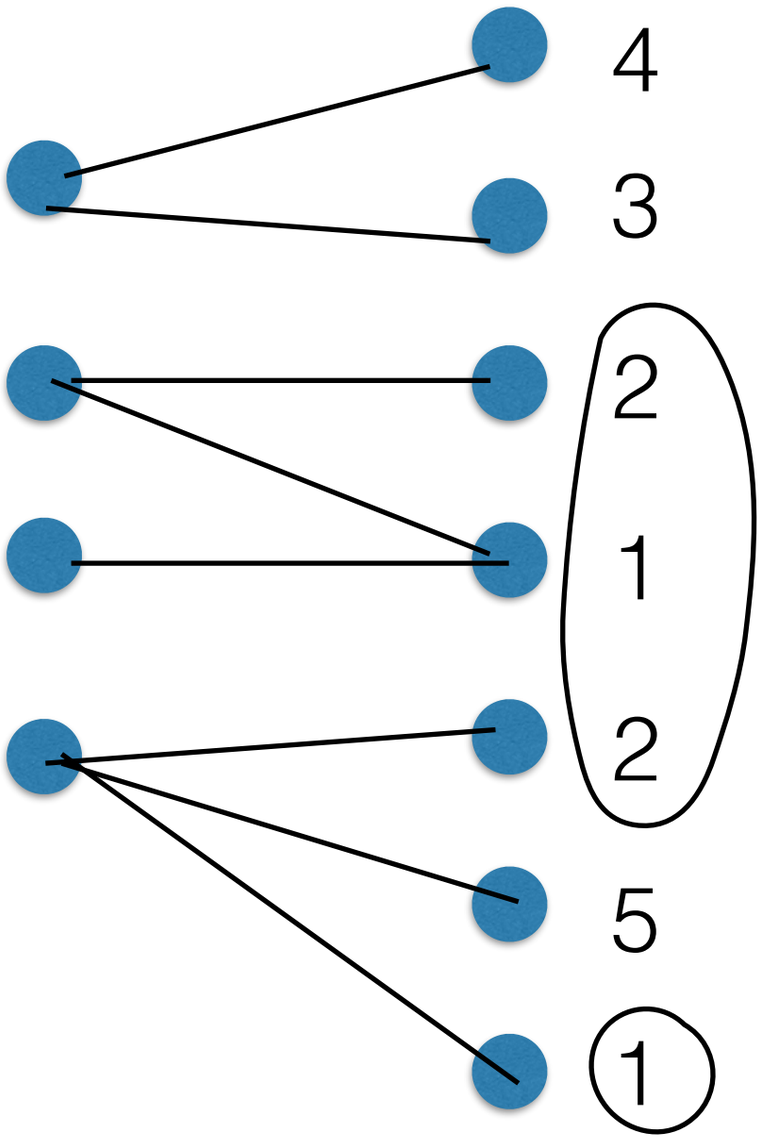
Local beam search
recherche locale à faisceau

- A chaque moment on maintient k noeuds n1,…,nk
- On génère tous les voisins V de ces noeuds
- Si parmi ces voisins il y a une solution alors on termine
- Sinon pour l'étape suivant on choisit dans V k noeuds dont le coût est minimal

Noter qu'il est possible que pour l'étape suivant on garde plusieurs voisin d'un noeud et aucun voisin d'un autre (on choisit k noeud d'un ensemble de tous les voisins de {n1,…,nk}).

Local beam search
recherche locale à faisceau

k=4



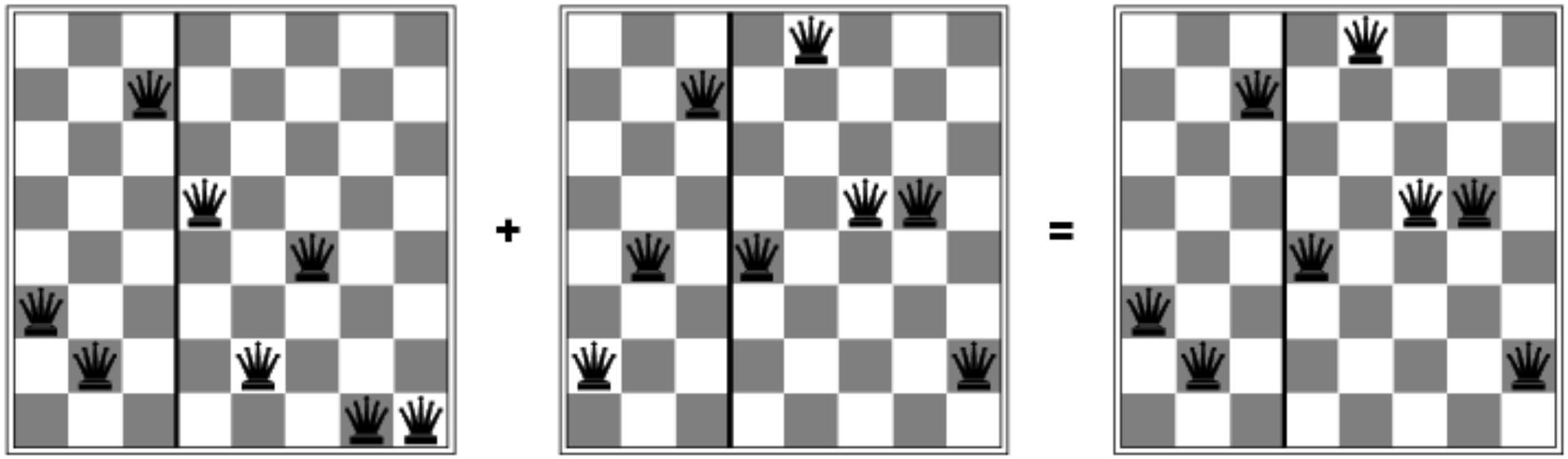
Algorithmes génétiques

- Un état successeur obtenu en combinant deux états parents
- Initialement k états générés aléatoirement (population)
- Un état représenté par une chaine de caractères sur un alphabet fini (souvent 0,1)
- La fonction d'évaluation (fitness fonction) permet d'évaluer la qualité d'un état. Plus la valeur de la fonction est élevée plus l'état est mieux "adapté".
- Produire la nouvelle génération d'états par la sélection, croisement et mutation.

Algorithmes génétiques

- 24748552 -- la position de chaque reines, colonne par colonne, par exemple dans la première colonne la reine se trouve sur la deuxième ligne, dans la deuxième colonne sur la quatrième ligne etc.)
- Fitness function: le nombre de couples de reines qui ne sont pas en conflit (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Algorithmes génétiques (croisement de deux configurations dans le problème de 8 reines)



Recherche "online"

Les algorithmes de recherche précédents était "offline", le calcul de la solution avant d'exécuter la solution.

Les algorithmes "online" :

- le calcul entrelacé avec l'exécution
- à chaque moment on peut visiter uniquement un voisin du sommet courant (recherche d'un chemin dans un labyrinthe)
- adapté à l'environnement dynamique qui change avec le temps ou un environnement inconnu (on ne connait pas quels sont les états)
- l'exploration -- il faut utiliser les actions, même non optimales, pour découvrir l'environnement

Recherche "online"

La qualité de la recherche online mesurée par

competitive ratio=(la complexité en temps de l'algorithme online) / (la complexité en temps de l'algorithme le plus efficace si l'environnement était complètement connu)

S'il y a des états sans issu l'algorithme peut rester bloqué (concerne le graphe orienté).

online depth-first search

delta(s,a) - l'état obtenu en appliquant l'action a dans l'état s

Pour chaque action a et chaque état s si delta(s,a)=t alors il existe une action b telle que delta(t,b)=s (pour chaque action a il existe une action b qui permet de "défaire" a). De plus b peut être trouvé si on connait s, a, t.

online depth-first search est la recherche c'est un algorithme naturel de recherche dans un labyrinthe

online depth-first search

s - un variable globale qui donne l'état précédent

a - l'action exécutée dans s

result[état, action] une table indexées par l'état et l'action qui donne la valeur de delta(état,action),
initialement la table est vide

untried[état] qui donne pour un état l'ensemble d'action qu'on a pas encore exécuté dans cet état

unbacktracked[état] donne pour chaque état visité la liste des actions "back" qu'on a pas encore exécuté, initialement vide

online depth-first search

function online-DFS(état t) retourne une action{

- si test-destination(t) alors retourner stop
- si t n'est pas dans `untried alors untried[t]=Actions(t)`
- `si s <> null alors`
  - `result[s,a] = t`
  - `ajouter s sur la pile de unbacktracked[t]`

online depth-first search (suite)

- si untried[t] est vide alors
    - ✦ si unbacktracked[t] est vide alors retourner stop
    - ✦ sinon a <-- action b telle que result[t,b]=POP(unbacktracked(t))
- sinon a <-- POP(untried[t])
- s <-- t
- retourner a

Recherche "online"
LRTA* - learning real time A*

Pour chaque sommet s, une heuristique admissible h(s).

L'idée : on améliore la valeur de la heuristique en cours de l'exploration.

Si l'environnement sûr (il existe une solution accessible à partir de chaque sommet) alors LRTA* est complet.

Recherche "online"
LRTA* - learning real time A*

variables globales:

résultat — un tableau dynamique (un map) initialement vide indexé par les couples [état,action]

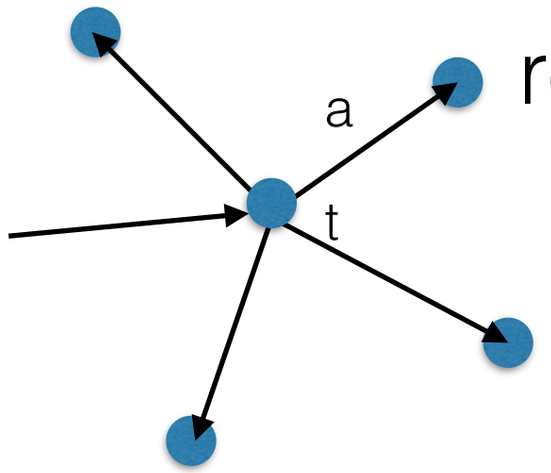
Si résultat[s,a] défini alors on a déjà visité s, on a exécuté a dans s et résultat[s,a]
donne l'état obtenu en appliquant a dans s.

H — un tableau dynamique (un map) indexé par les états, initialement vide.

H[s] donne la valeur de heuristique pour l'état s. Pendant la première visite dans s on initialise H[s]=h(s), où h une fonction heuristique admissible connue.

Recherche "online"
LRTA* - learning real time A*

résultat[t,a] inconnu pour une action a de Actions(t)

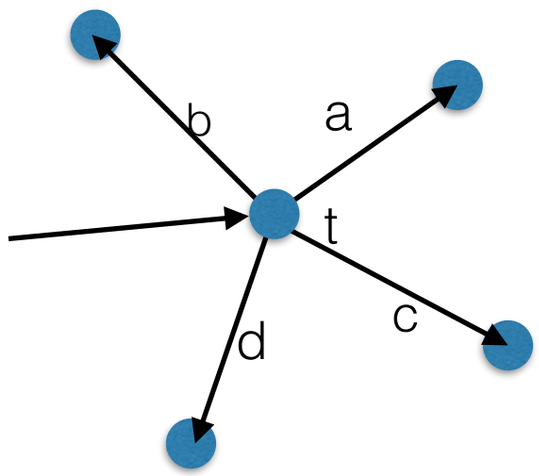


resultat[t,a] = ?

H[t] = h(t)

---

résultat[t,a] connu pour toute action a de Actions(t)



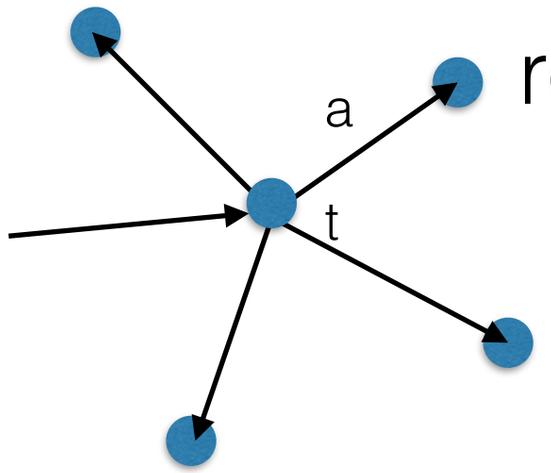
mise à jour de H[t]

$$w = \min_{a \in \text{Actions}(t)} \text{cout}(t, a, \texttt{resultat}[t, a]) + H[\texttt{resultat}(t, a)]$$

$$H[t] = \max\{H[t], w\}$$

Recherche "online"
LRTA* - learning real time A*

résultat[t,a] inconnu pour une action a de Actions(t)

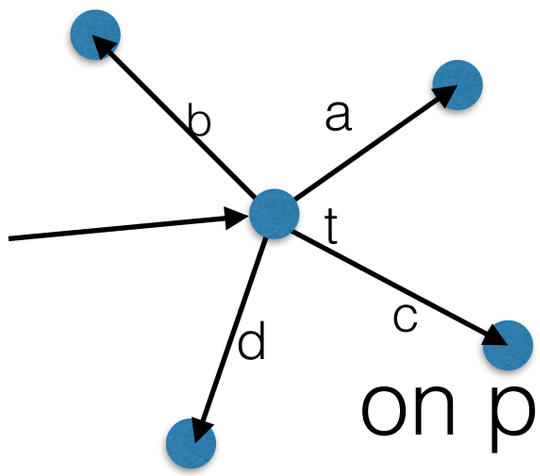


resultat[t,a] = ?

on préfère une action a
telle que resultat[s,a] inconnu
(exploration)

---

résultat[t,a] connu pour toute action a de Actions(t)



on préfère l'action a telle que
cout(t,a,resultat[t,a])+H[resultat[t,a]]
soit minimal

Recherche "online"
LRTA* - learning real time A*

```
variable globales : s – état précédent, a – action exécutée dans
s pour aller vers l'état courant. Initialement s et a sont null.

La fonction suivante met à jour l'estimation du coût
minimal :

fonction LRTA*coût(s,a,t) retourne réel
   if t undefined then return h(s)
   else return coût(s,a,t)+H[t]
```

Recherche "online"
LRTA* - learning real time A*

```
variable globales : s – état précédent, a – action exécutée dans
s pour aller vers l'état courant t. Initialement s et a sont
null.


Faire tourner en boucle jusqu'à stop:


fonction LRTA*(état t) retourne action
  if t état final then stop
  if H[t] undefined then H[t]=h(t)
  if s is not null then  /*mis à jour de H[s]*/
       resultat[s,a]=t
       w=min { LRTA*coût(s,b,résultat[s,b]) | b in Actions(s) }
       H[s] = max { H[s], w }
   a = action b qui minimise LRTA*coût(t,b,résultat[t,b])
   s=t
   retourner a
```

Intuition :

- quand on entre dans un état t et si H[t] non défini alors c'est la première visite dans t et on initialise H[t]=h(t)
- quand on sort d'un état s alors et si tous les voisins de s ont déjà été visités alors on calcule

w = min coût(s,b,résultat[s,b])+H[resultat[s,b]] pour tout b dans Actions(s)

- si H[s]<w alors H[s]=w