Intelligence Artificielle

Recherche Heuristique
(*Informed Search*)

Stephane Marchand-Maillet

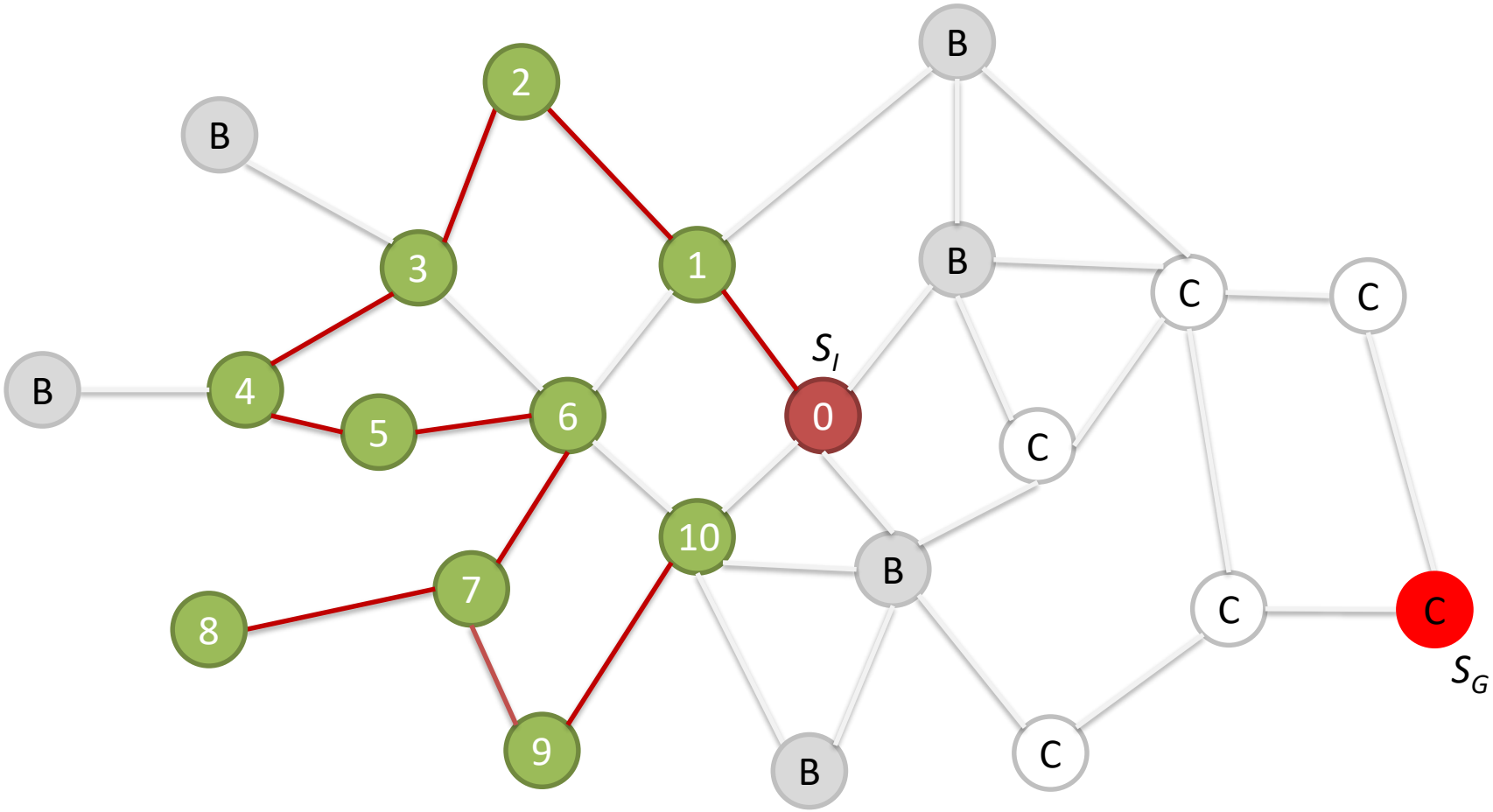
Recherche aveugle: aléa

Dans la recherche aveugle, certaines opérations sont “aléatoires”:

- Elles résultent de l’ordre dans lequel le problème est présenté
  - Utilisation du voisinage, des propriétés, …

- Un choix existe auquel on a répondu de facon aléatoire, en l’absence de meilleure alternative
  - Direction d’exploration, …

Recherche en Profondeur (DFS) - Rappel

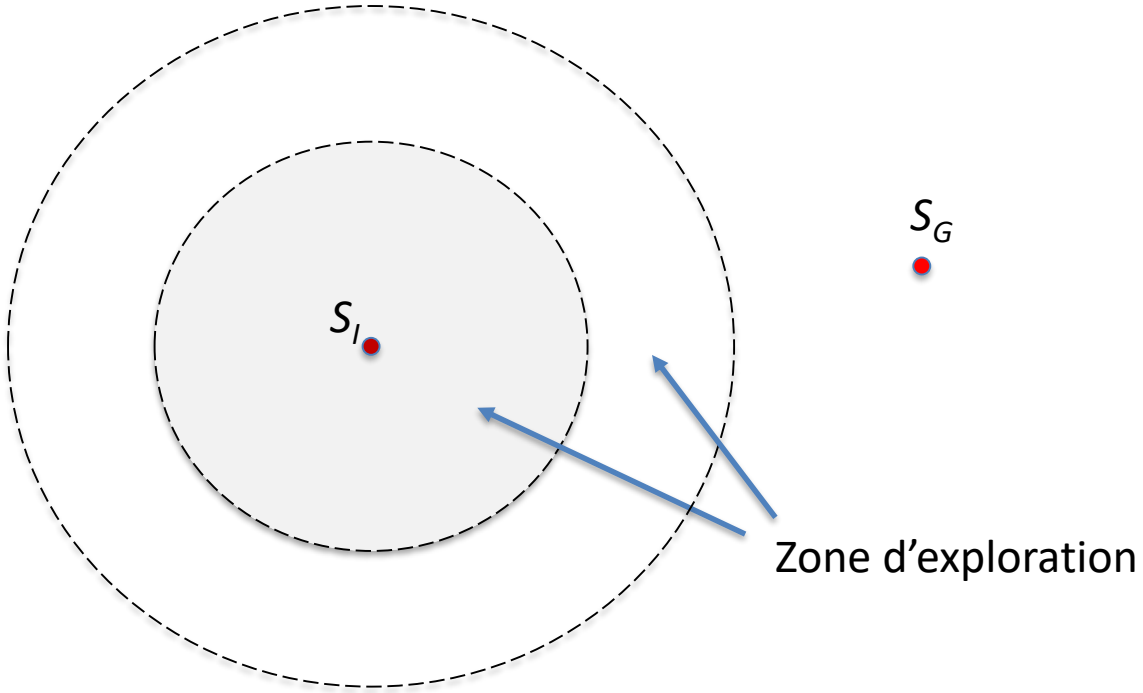


Voir aussi: https://visualgo.net/en/dfsbfs

Recherche aveugle: aléa

Exple: Navigation

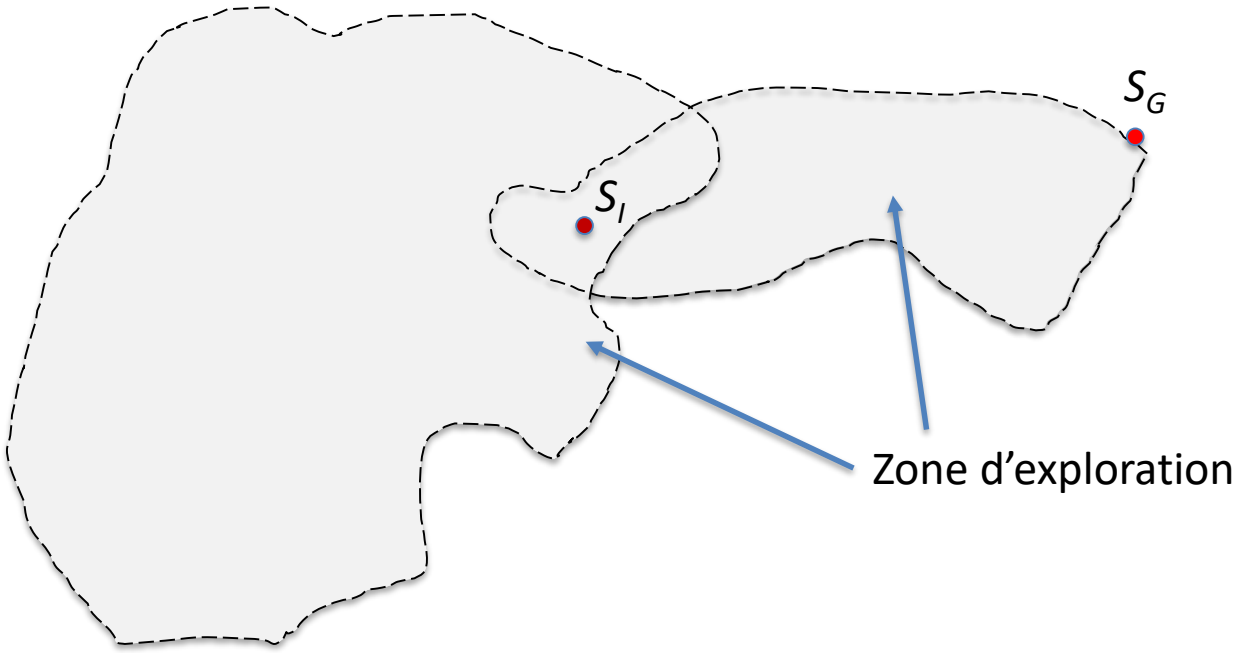
- On utilise le BFS pour chercher le chemin optimal d'un état (S_I) à un autre (S_G)



Recherche aveugle: aléa

Exple: Navigation

- On utilise le DFS pour chercher le chemin optimal d'un état (S_I) à un autre (S_G)



Recherche aveugle: aléa

Exple: Navigation

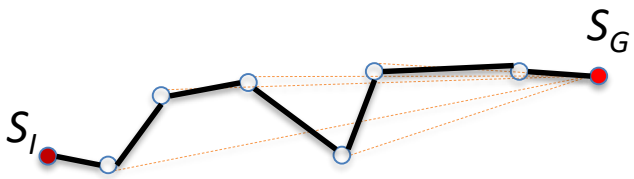
- On utilise *xxx* pour chercher le chemin optimal d'un état (S_I) à un autre (S_G)

→ Y aurait-il un choix plus judicieux qui fasse converger plus vite l'algorithme (trouve la solution plus rapidement)?

… quitte à quand même, au pire, explorer des solutions inutiles

Recherche aveugle: aléa

Idée: navigation “orientée”



Toujours choisir l’état suivant *<u>le plus proche de</u>* S_G

$$s_{t+1} = \text{argmin}(\text{dist}(s\text{- } s_G) \text{ t.q } s \text{ voisin de } s_t)$$

→ C est une *heuristique*

Informed search

- On applique le principe de recherche de solution en injectant de la connaissance *a priori* (*informed search*)

- On applique un classement particulier (heuristique, basée sur une connaissance a priori) à la liste (catégorie B) pour piloter l'exploration et la diriger vers la solution s_G plus rapidement

Heuristique: définition

Def: une heuristique est une fonction h d'estimation de coût telle que:

$$h: V \longrightarrow \mathbf{R}^+ \qquad \text{avec} \quad h(v)=0 \text{ si } state(v)=s_G$$

$h(v)$ représente une estimation du coût du chemin de l'état actuel à l'état final s_G passant par le nœud v (ayant v comme première étape)

Heuristique: exemples

- Problème du Sac à Dos: mettre les gros objets d'abord

(on s'assure d'avoir de la place pour les gros objets)

- Problème du Sac à Dos: mettre un gros objet et à chaque fois « compléter » par des petits

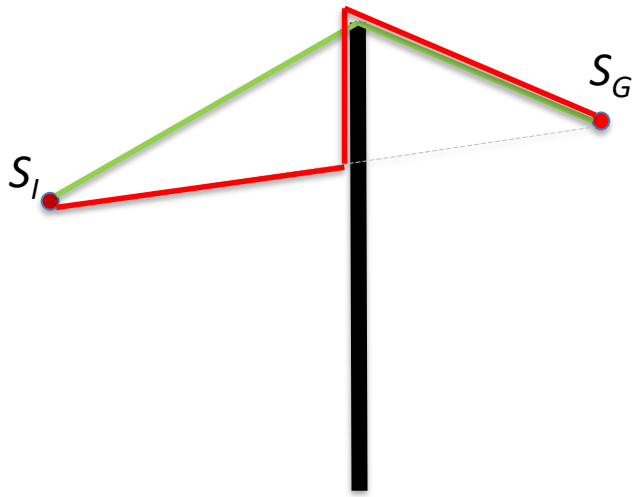
(on ne perd pas de place)

- Voyageur de commerce: faire un DFS et parcourir cet arbre (2-OPT)

(heuristique prouvée)

Heuristique: exemples

Navigation:



Une heuristique ne doit pas changer la complétude de l'algorithme: elle n'empêche pas l'algorithme de trouver la solution si elle existe et si il la trouvait sans heuristique

Heuristique: exemples

- Navigation: on essaie les directions aléatoirement

→ On revient à la recherche aveugle

- Navigation: toujours aller dans la direction la plus opposée au but

→On trouvera le but après avoir exploré tout l'espace

→L'heuristique est mauvaise mais n'empêche pas de trouver la solution

La qualité d'une heuristique peut ne pas être garantie.

Elle dépend de la compréhension du problème et de la variabilité des situations (heuristique efficace « en moyenne »)

Fonction d'évaluation

Def: la fonction d'évaluation *f* fournit le coût estimé d'une solution (chemin) passant par le nœud *v*

→On trie les nœuds *v* de la catégorie "B" (nœuds dans la liste) par valeurs croissantes de *f*(*v*)

→La liste est un Tas-Min

→On explore d'abord les solutions qu'on estime être moins coûteuses (*Best First Search*)

Best First Search (moins coûteux d'abord)

liste ← vide ; liste.push(s_I)
repeat
    $s_{courant}$ ← liste.pop()
    if ($s_{courant}$ == s_G)
        break
    list.push($sort_f(\Gamma(s_{courant}))$)
until liste.len() == 0

if ($s_{courant}$ == s_G)
    backtrack solution
else
    pas de solution

Recherche

Explicitation de la solution

→ Tout est dans la stratégie de gestion de la liste et l'expansion des noeuds (états)

Recherche en coût uniforme (RAPPEL)

Similaire à la recherche en Largueur

$g(v)$: coût du chemin de la racine au nœud v

$g(v)$ représente la somme des coûts de transition entre les états $c(s,s')$ le long du chemin

→Expansion du nœud le moins coûteux de la liste

→La liste est un Tas-min des coûts

Analogue au plus court chemin de Dijkstra

Fonction d'évaluation $f(v)$

- C'est le cœur de l'algorithme *Best First Search*
- Peut prendre toutes les formes.

Exemples:

- $f(v) = h(v)$: heuristique pure

  →*Greedy Best First Search*

- $f(v) = g(v) + h(v)$: prise en compte de la connaissance actuelle

Propriétés de *Greedy Best First Search*

- Complet: Préserve la complétude des algorithmes de recherche (complet si l'espace de recherche est fini)
- Optimal: non

Complexité:

- Temps: $O(b^m)$ avec m profondeur maximale
- Espace: $O(b^m)$ conserve les nœuds pour $f(v)$

La complexité du «pire des cas» (*worst case*) ne s'améliore pas en général (car elle considère l'heuristique inefficace)

Il faut regarder la complexité empirique (en moyenne) qui illustre mieux l'efficacité de l'heuristique à éviter le pire des cas

Heuristique admissible

Def: Une heuristique est admissible si elle sous-estime le coût du chemin vers la solution

Si $h^*(v)$ est le coût réel du nœud v à la solution (heuristique idéale), alors si h est admissible:

$0 <= h(v) <= h^*(v)$ pour tout nœud v

Une heuristique admissible est optimiste

Heuristique admissible: exemple

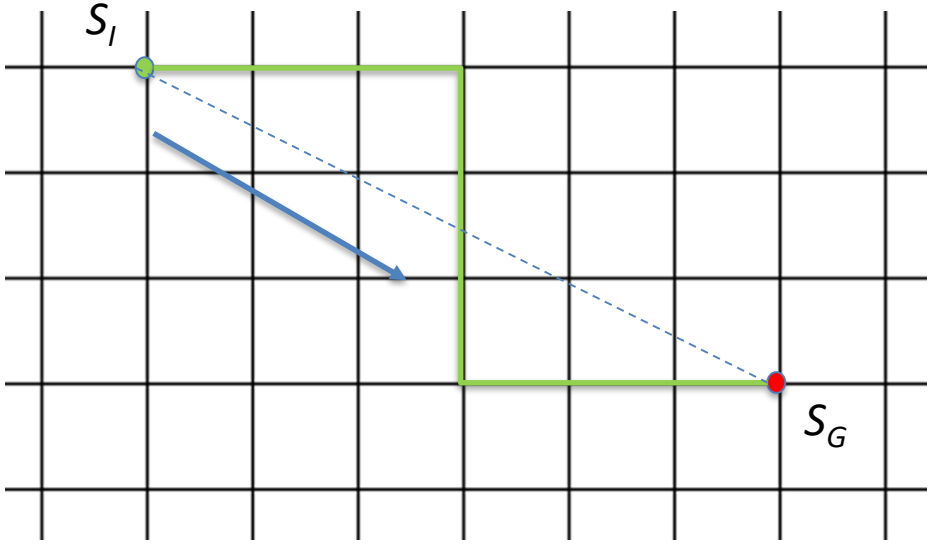
- Distance Euclidienne sur un espace de navigation discrétisé: admissible

- Distance de Manhattan: pas admissible si les mouvements diagonaux sont permis (cf illustration)

→Une heuristique admissible est une stratégie qui ne prend pas en compte certaines contraintes (obstacles) du problème

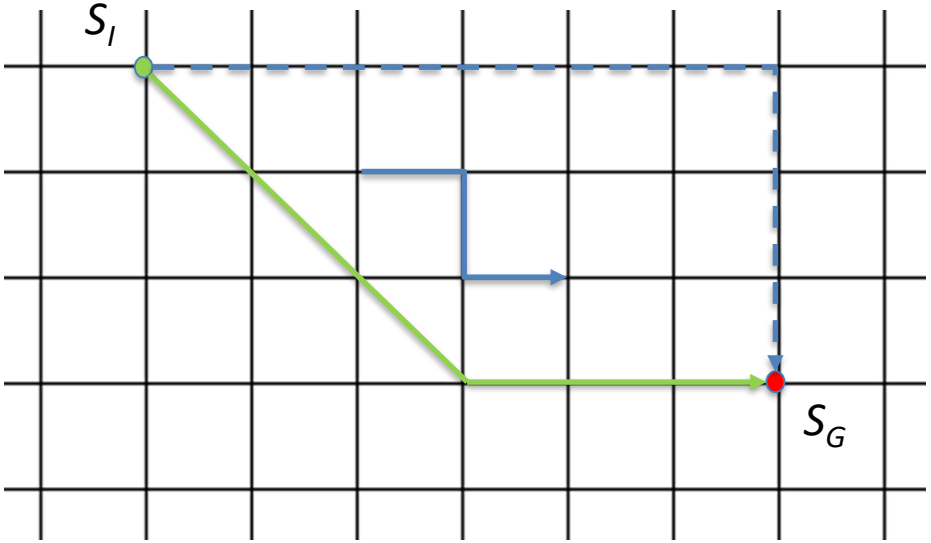
→Une heuristique admissible ne rajoute pas de contrainte

Heuristique admissible: exemple



Heuristique admissible

Heuristique non-admissible



Heuristique consistante

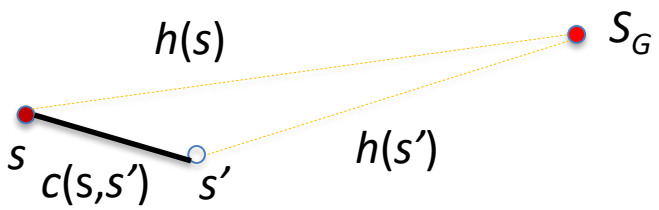
Si on veut éviter de répéter l'exploration d'un même état:

- On veut que le coût de ré-accéder à cet état soit au moins aussi élevé que la première fois
- Sinon le coût d'accès aux états suivants devra être mis à jour en partant de ce nouveau coût

Une heuristique qui maintient cet ordre est dite consistante:

Si il existe une transition de s à s' de coût c(s,s') alors

$$h(s) \leq h(s') + c(s,s')$$



Similaire à l'inégalité triangulaire

Preuve (par l'absurde)

- Supposons que les nœuds *v* and *v'* correspondent au même état (state(*v*)=state(*v'*)) avec $g(v') < g(v)$ mais *v* est atteint en premier
- *v'* ne pouvait être en catégorie B (frange) quand v a été atteint car $g(v') < g(v)$, donc

$$g(v') + h(v') < g(v) + h(v)$$

- Donc *v'* est l'enfant d'un autre nœud *v''* de la frange, après que *v* soit étendu

$$g(v) + h(v) \leq g(v'') + h(v'')$$

- En rassemblant, on a

$$g(v') + h(v') < g(v'') + h(v'')$$

ou

$$h(v'') > h(v') + c(v'',v')$$

→ pas de consistance

Précision heuristique

La précision heuristique évalue la qualité d'une heuristique:

Si h_1 et h_2 sont 2 heuristiques admissibles avec

$$h_1(v) >= h_2(v) \text{ pour tout nœud } v$$

alors h_2 est plus précise que h_1 et on a

$$0 <= h_1 <= h_2 <= h^*$$

(h_2 est plus proche de h^* que h_1)

La précision d'une heuristique ne tient pas compte de sa complexité

Best First Search (moins coûteux d'abord)

liste ← vide ; liste.push(s_I)
repeat
    $s_{courant}$ ← liste.pop()
    if ($s_{courant}$ == s_G)
        break
    list.push($sort_f(\Gamma(s_{courant}))$)
until liste.len() == 0

if ($s_{courant}$ == s_G)
    backtrack solution
else
    pas de solution

Recherche

Explicitation de la solution

→ Tout est dans la stratégie de gestion de la liste et l'expansion des noeuds (états)

Approximation: *Beam search*

C est le *Best First Search* avec une limite de longueur pour la liste:

- Consommation mémoire limitée
- Temps de recherché réduit

On limite à $B>0$ le nombre d'états suivants à explorer:

Au lieu de:

list.push($sort_f(\Gamma(s_{courant}))$)

on a:

list.push($prune_B(sort_f(\Gamma(s_{courant})))$)

On ne développe que les *B* états les plus prometteurs

Algorithme A* (*A-star*)

C'est l'algorithme *Best First Search* avec la fonction d'évaluation:

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

ou h est une heuristique admissible et consistante, et

$$c(s,s') >= \varepsilon > 0 \qquad \text{pour tout } s \text{ et } s'$$

Sous ces conditions:

l'algorithme A* est complet et optimal

Complétude de A*

Sur un graphe fini:

- A* explore le graphe complètement au pire des cas et donc trouvera la solution si elle existe

Sur un graphe infini avec:

- Un facteur de branchement fini
- Un cout par arête strictement positif

$$c(s,s') \geq \varepsilon > 0$$

l'algorithme trouve la solution en temps fini si elle existe

Optimalité de A*

- Si l'heuristique est admissible, A* est optimal
  - La solution trouvée est optimale
- Preuve (par l'absurde):
  1. Supposons que l'on est sur le point d'étendre un nœud solution v (state(v)=s=s_G) sous-optimal ($g(v)=c > c^*$, ou le coût c^* est optimal)
  2. Soit v^* un nœud solution optimal (à venir)
  3. Il doit y avoir dans la frange un nœud v' qui est sur le chemin vers v^*
  4. Donc $g(v) = c > c^* = g(v^*) \geq g(v') + h(v')$
  5. Mais donc, v' devrait être étendu en premier (si h est admissible) → contradiction

Optimalité de A* (efficacité)

L'efficacité de A* est optimale

- Tout autre algorithme (utilisant la même heuristique h) doit développer au moins autant de nœuds que A*

Preuve :

1. En dehors de la solution, A* étend tous (et seulement) les nœuds v tels que $g(v) + h(v) < c^*$
   - Si il n'y a pas de nœud non-solution t.q $g(v) + h(v) = c^*$

2. Tout autre algorithme (utilisant la même heuristique h) doit aussi étendre ces nœuds (pour chercher une solution)

Propriétés de A*

- Si h_2 est plus précise que h_1 alors tous les nœuds produits par A* avec h_2 (fonction f_2) seront produits par A* basé sur h_1 (fonction f_1), sauf certains nœuds tels que $f_1(v) = f_2(v) = f^*(v)$ (solutions optimales pour lesquelles h_1 et h_2 choisiront différents ex-aequo)

- A* peut explorer indéfiniment si il existe une infinité de nœuds v tels que

$$f(v) <= f(v^*) \text{ avec } \text{state}(v^*) = s_G$$

A* les explorera avant de trouver v^* (s_G)

Stratégie IDA* (*Iterative deepening* A*)

IDS limite la profondeur → évite que la recherche en profondeur explore des chemins indéfiniment

Même principe: IDA* limite $f(v)$

→ Ne pas approfondir des chemins de coût excessif par rapport à:

- une connaissance à priori
- une contrainte de l'environnement (temps, mémoire, budget,….)
- …

IDA*

Algorithme:

$f_{max} = f(v_l)$ $(state(v_l) = s_l)$

répéter

- expansion des nœuds v pour lesquels $f(v) <= f_{max}$
- fixer $f_{max} = min(f(v)$ t.q v est une feuille)

jusqu'à solution

→ alterne l'exploration par niveaux

Propriétés de IDA*

IDA* :

- Complet
- Optimal
- Utilise moins de mémoire que A*
- Ne nécessite pas de tri explicite de la liste (le tri se fait par le choix du noeud suivant à developper)

- Revisite des chemins hors du chemin courant
- N'offre pas une utilisation optimale de la memoire (AIMA p 101-107)

SMA*

Simplified Memory-bounded A*

Idée simple:

- On fixe la limite mémoire
- si la limite est atteinte, on enlève le noeud le moins interessant

Propriétés:

- Comme A* tant que la limite n'est pas atteinte
- Complet si la mémoire est suffisante
- Optimal si la mémoire est suffisante pour contenir le chemin solution le moins profond
- Optimal dans les contraintes mémoire données

Génération d'heuristiques

Une stratégie de création d'heuristiques est la relaxation du problème:

- On rend le problème plus simple
- On enlève des contraintes

Exemple:

- Le « nombre de pièces mal placées » relaxe le problème en imaginant que les pièces peuvent se déplacer librement (1 coup par pièce)

Un problème idéalement relaxé est:

- Simple a résoudre
- Bien moins couteux à résoudre que l'original

On peut générer des heuristiques automatiquement

Alternatives

Stratégies de recherche:

- Locale: pas d'arbre conservé, juste une connaissance locale (seule la solution est importante, pas le chemin)
- Steepest: on développe seulement le noeud de coût minimal

Autres strategies:

- Monte Carlo
- Beam search
- Algorithmes génétiques

Quand utiliser la recherche?

- Quand l'espace d'états est de taille "raisonnable" (pour une énumération "systématique")
- Quand il existe de bonnes heuristiques
- Quand on ne peut pas calculer de gradient vers la solution (peut être vu comme une heuristique pour améliorer la solution courante)

Résumé

- Une heuristique permet d'exploiter l'aléa dans les recherche (en le remplaçant par une connaissance a priori)
- Une heuristique ne doit pas changer la complétude
- Une heuristique peut être admissible et/ou consistante
- L'algorithme A* donne des garanties d'optimalité et de complétude
- On peut lui associer des variantes en fonction des contraintes (temps et mémoire)