

INF4230 – Intelligence Artificielle

Recherche heuristique / Algorithme A*

Hiver 2017

Sommaire

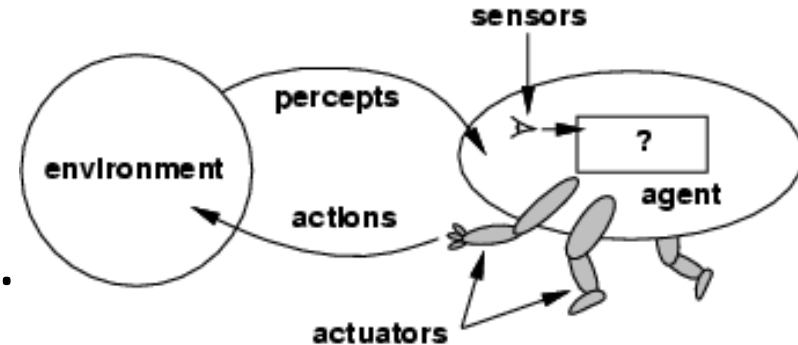
- Rappels :
 - Agents.
 - Résolution de problème par recherche.
 - Espace d'états.
- Recherche informée.
- Recherche «meilleur en premier».
- Algorithme A^* .
- Heuristiques.
- Propriétés de A^* .
- Exemples d'heuristiques.

Rappel : Agents rationnels

- **Agent :**

- Perçoit son environnement.
- Agit dans son environnement
- Se fait une représentation du monde (modèle).
- Mesure de performance.
- (Modèle PEAS).

- Un agent a une **fonction** $f: \mathcal{P}^* \rightarrow \mathcal{A}$.



Rappel : Environnements

- Caractéristiques d'un environnement :
 - Complètement observable vs partiellement observable.
 - Déterministe vs stochastique.
 - Épisodique vs séquentiel.
 - Statique vs dynamique.
 - Discret vs continu.
 - Agent unique vs multi-agent.
- Quasi synonymes : environnement et monde (*world*).

Rappel : Paradigme de résolution de problème

- La fonction f d'un agent peut être implémentée à l'aide du paradigme «résolution de problèmes par recherche».
- Construction d'un modèle du monde à l'aide des données sensorielles provenant des capteurs.
- Un graphe est étendu à partir de l'état initial (actuel) en simulant les actions de l'agent.
- Les décisions séquentielles sont sélectionnées à l'aide d'une recherche dans un graphe.

Rappel : Types de problème

- **Déterministe + Complètement observable** → **problème à partir d'un état unique.**
 - L'agent sait tout et peut simuler ses actions.
 - Solution = séquence d'actions
- **Non déterministe et/ou observabilité partielle** → **problème de contingence.**
 - À chaque itération, on doit percevoir l'environnement.
 - Solution = Plan contingent; Alternance recherche et exécution.
- **Non observable** → **problème sans capteurs ou problème de conformance (*conformant planning*).**
 - L'agent n'a aucune idée de son environnement.
 - Solution = séquence d'actions.
- **Espace d'états inconnu** → **problème d'exploration.**

Rappel : Agent basé sur des buts

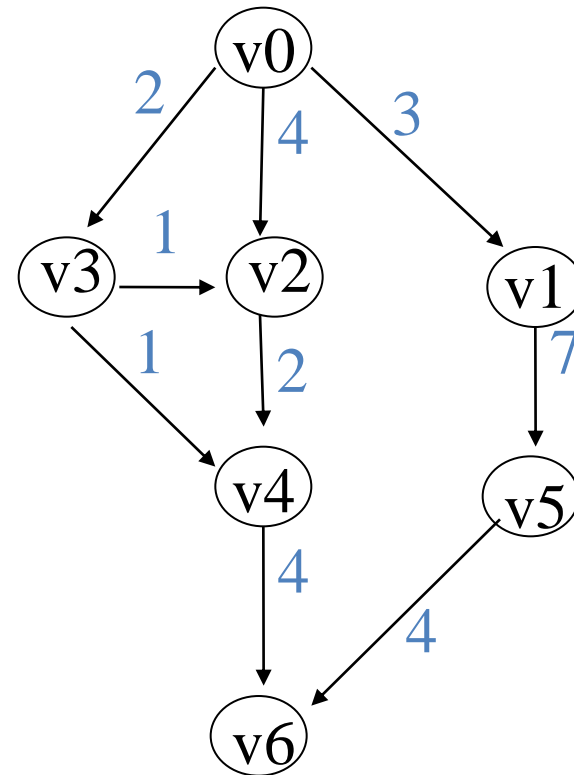
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Exemple 1 – Agent sur une carte

Monde:

Villes et routes.

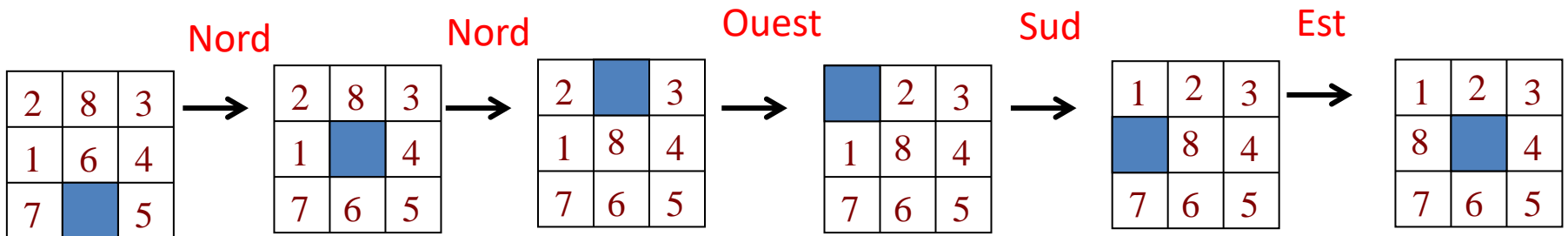
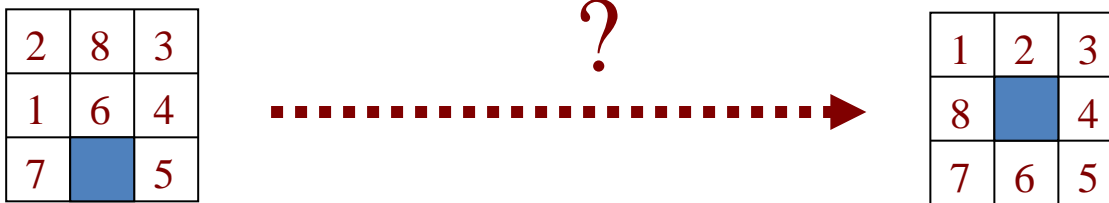


Problème posé (état initial, but):

v0: ville de départ (état initial)

v6: destination (but)

Exemple 2 – Jeu de taquin (puzzle)



RECHERCHE INFORMÉE

Algorithme Meilleur en premier (*Best-First-Search*)

- La définition varie selon les auteurs :
 - Dans le livre de Norvig et Russell :
 - «**Greedy Best-First-Search**» \neq «**Best-First-Search**»
- «**Greedy Best-First-Search**» (GBFS) est une recherche locale \approx *hill climbing* (chapitre 4).
- «**Best-First-Search**» est une recherche globale.
- Idée = choisir le prochain état qui «semble» le plus près du but (meilleur).
- Ce choix est fait par une **(fonction) heuristique**.

Qu'est-ce qu'une heuristique

- Dans divers domaines, dont en informatique, une heuristique est une méthode (~algorithme) qui calcule rapidement (ex: en temps constant, linéaire ou polynomiale) une solution pouvant être approximative et incomplète à un problème généralement trop complexe.

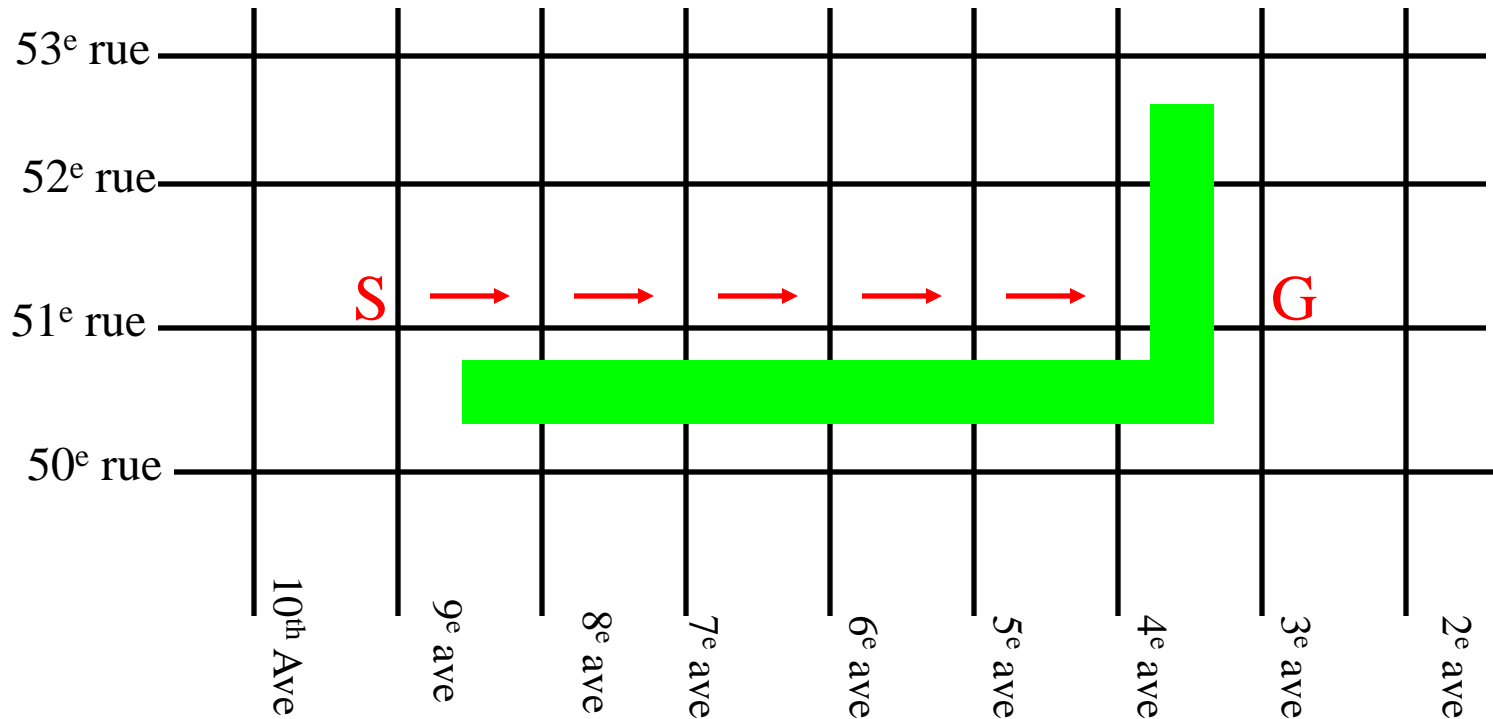
Algorithme de recherche en IA / Fonction heuristique

- Estimation de la distance (coût restant) entre un état n et un but g .
- Le but g peut être implicite.
- Généralement notée h ou $h(n)$.
 - $h(n)$: estime le coût restant pour atteindre le but implicite g à partir du nœud (état) n .
- Exemple de fonctions heuristiques pour la navigation dans sur une carte :
 - Distance euclidienne (aussi appelée à vol d'oiseau).
 - Distance de Manhattan (ville quadrillée).

Greedy Best-First-Search

- Utilisation d'une heuristique pour guider la recherche.
- Algorithme :
 1. GBFS(n) :
 2. tant que n ne satisfait pas le but
 3. $S \leftarrow \text{Successeurs}(n)$
 4. $n' \leftarrow$ choisir n' dans S ayant le plus petit $h(n')$
 5. si $n' = n$ alors ÉCHEC
 6. $n \leftarrow n'$

Exemple de GBFS sur une carte

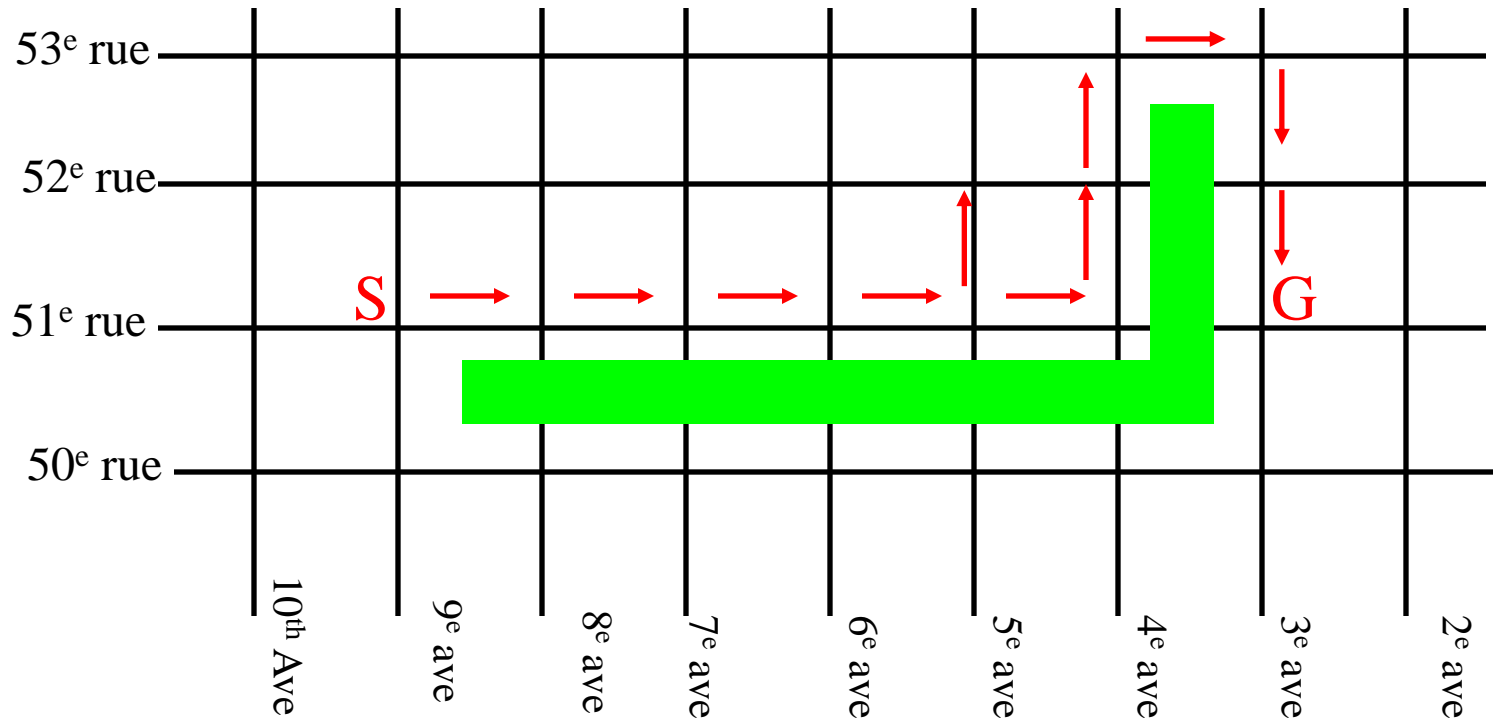


(Adaptée d'une illustration par Henry Kautz, U. of Washington)

Best-First-Search

- Algorithmme :
 1. BFS(n_0) :
 2. $open \leftarrow \{n\}$
 3. tant que open n'est pas vide
 4. $n \leftarrow$ choisir dans open qui minimise $h(n)$
 5. si n satisfait le but, alors retourner solution
 6. ajouter successeurs(n) dans open

Exemple de GBFS sur une carte



(Adaptée d'une illustration par Henry Kautz, U. of Washington)

Algorithme A*

- **A*** est une extension de l'algorithme de **Dijkstra**.
 - Ajout d'une heuristique.
- A* et les heuristiques sont à la base de beaucoup de travaux en IA:
 - Recherche de meilleures heuristiques.
 - Heuristiques indépendantes du problème.
 - Apprentissage automatique d'heuristiques.
- Pour décrire A*, on décrit un algorithme générique très simple, dont A* est un cas particulier.

VARIABLES IMPORTANTES : *open* et *closed*

- ***open*** :
 - contient les nœuds à traiter;
 - c'est à dire à la **frontière** de la partie explorée jusqu'à présent dans le graphe.
- ***closed*** :
 - contient les nœuds déjà traités;
 - c'est à dire à l'intérieur de la frontière délimitée par *open*.

Insertion des nœuds dans *open*

- Les nœuds dans *open* sont ordonnés selon une estimation $f(n)$ de la qualité d'une solution passant par ce nœud.
- Une fonction $f(n)$ donne ou estime la qualité de la meilleure solution passant par le nœud n .
- Pour chaque nœud n , $f(n)$ est un nombre réel positif ou nul, estimant le coût pour un chemin partant de l'état initial n_0 , passant par n , et arrivant dans un état n' satisfaisant le but g .

Définition de la fonction $f(n)$

- La fonction f désigne la distance entre le nœud initial et le but.
- En pratique on ne connaît pas cette distance : c'est ce qu'on cherche !
- Par contre on connaît la distance optimale *dans la partie explorée* entre le nœud initial n_0 et un nœud *déjà exploré*.
- Ainsi, on peut séparer $f(n)$ en deux parties: $f(n) = g(n) + h(n)$
 - $g(n)$: coût réel du chemin optimal partant du nœud initial n_0 à n dans la partie déjà explorée.
 - $h(n)$: coût estimé du reste du chemin partant de n jusqu'à un état satisfaisant le but.
 - $h(n)$ est une **fonction heuristique**.

Algorithme générique de recherche dans un graphe

Algorithme rechercheDansGraphe(n_0)

1. $open \leftarrow$ Créer un ensemble ordonnées par $f(n)$ // vide au départ
2. $closed \leftarrow$ Créer un ensemble // vide au départ
3. insérer n_0 dans $open$
4. while (true) // *la condition de sortie (exit) est déterminée dans la boucle*
 5. si $open$ est vide, sortir de la boucle avec échec (aucune solution n'existe)
 6. $n1 =$ noeud au début de $open$ avec le plus petit $f(n)$
 7. enlever $n1$ de $open$ et l'ajouter dans $closed$
 8. si $n1$ est le but, sortir de la boucle avec succès en retournant le chemin;
 9. pour chaque noeud successeur $n2$ de $n1$
 10. Initialiser la valeur $g(n2) = g(n1) + \text{coût de la transition } (n1, n2)$
 11. mettre $\text{parent}(n2) = n1$
 12. si $open$ ou $closed^*$ contient un nœud $n3$ équivalent à $n2$ (même état) avec $f(n2) < f(n3)$, enlever $n3$ de $open$ ou $closed^*$ et insérer $n2$ dans $open$.
 13. sinon (c-à-d., $n2$ n'est est ni dans $open$ ni dans $closed$)
 14. insérer $n2$ dans $open$ en triant les nœuds en ordre croissant selon $f(n)$

* Le test dans $closed$ à la ligne 12 est uniquement nécessaire en cas d'heuristique inadmissible ou **inconsistante**.

Exemple A* avec recherche dans une ville

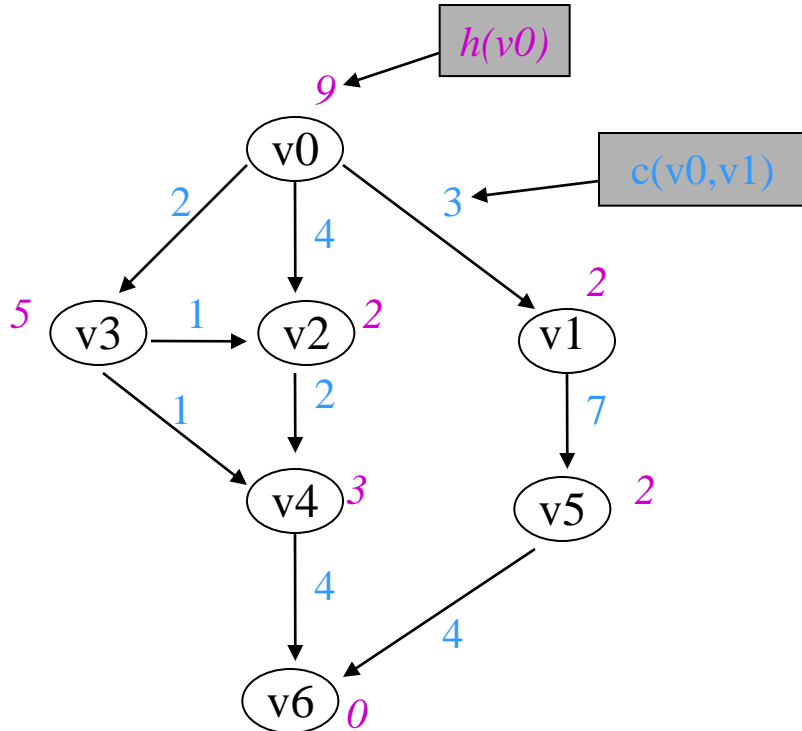
Routes entre les villes :

v0: ville de départ

v6: destination

h: distance à vol d'oiseau

C: distance réelle entre deux ville



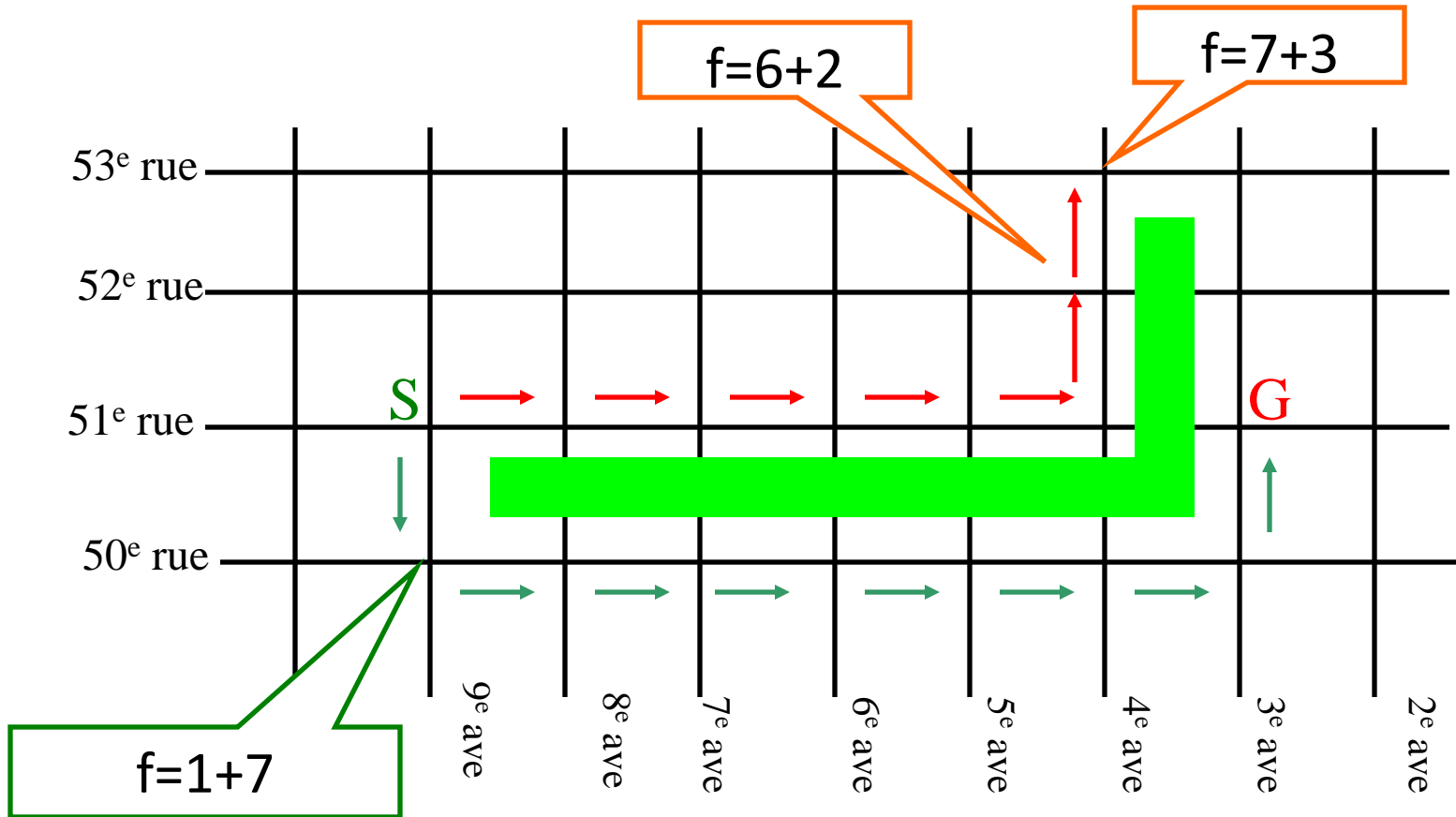
Contenu de *open* à chaque itération (état, f, parent) :

1. (v0, 9, void)
2. (v1,5,v0) (v2,6,v0), (v3,7,v0)
3. (v2,6,v0) (v3,7,v0), (v5,12,v1)
4. (v3,7,v0),(v4,9,v2),(v5,12,v1)
5. (v2,5,v3),(v4,6,v3),(v5,12,v1)
6. (v4,6,v3),(v5,12,v1)
7. (v6,7,v4), (v5,12,v1)
8. Solution: **v0,v3,v4,v6**

Contenu de *closed* à la sortie (noeud, f) :

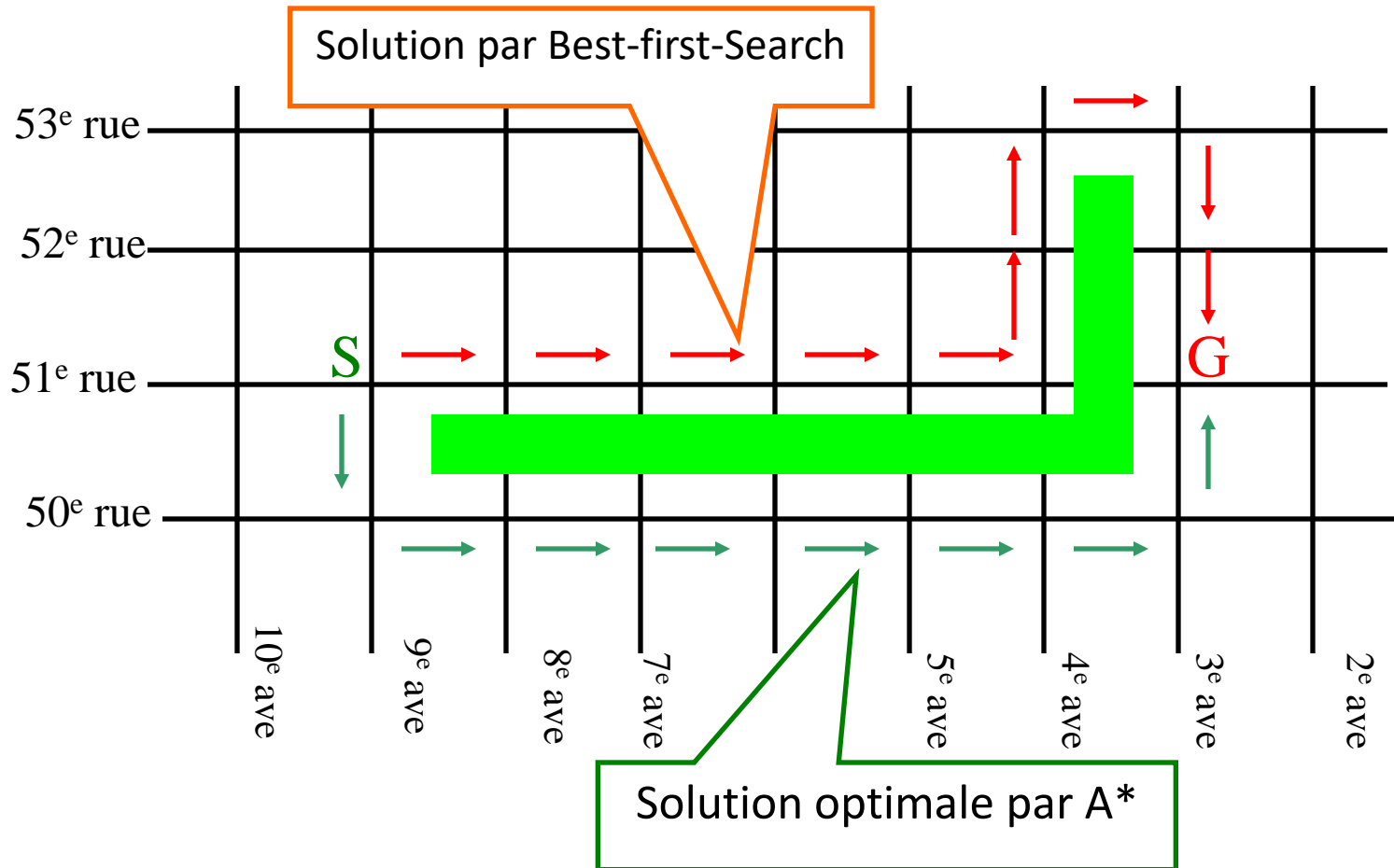
(v4,6), (v3,7), (v2,5), (v1,5), (v0,9)

Exemple de recherche avec A*



(Illustration par Henry Kautz, U. of Washington)

Non-optimalité de Best-First Search



(Illustration par Henry Kautz, U. of Washington)

Démos

Simulation des algorithmes :

A*, Recherche en profondeur, largeur, meilleur en premier

<http://ericbeaudry.ca/INF4230/demos/search/> (Applet Java)

<http://qiao.github.io/PathFinding.js/visual/>

Exemples d'heuristiques

JEU DE TAQUIN

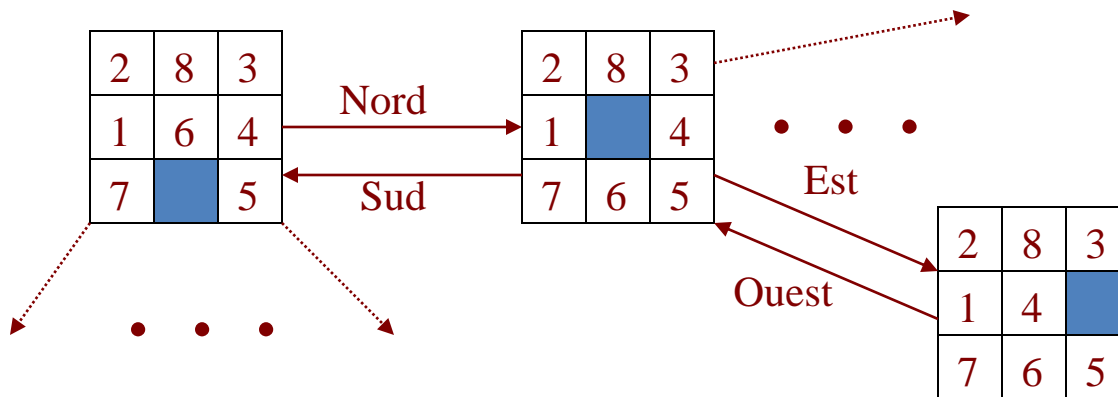
Jeu de taquin

- *État*: configuration légale du jeu
- *État initial*: configuration initiale
- *État final (but)*: configuration gagnante
- *Transitions (fonction successeur)*

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

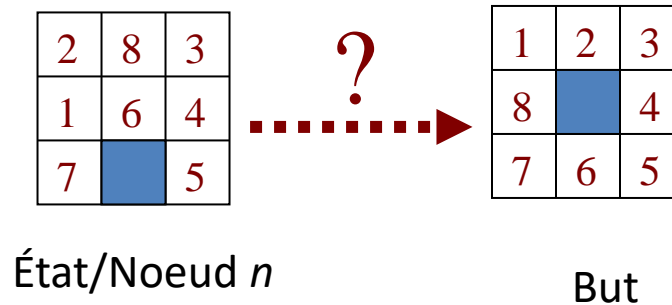


| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |



Heuristiques pour jeu de taquin

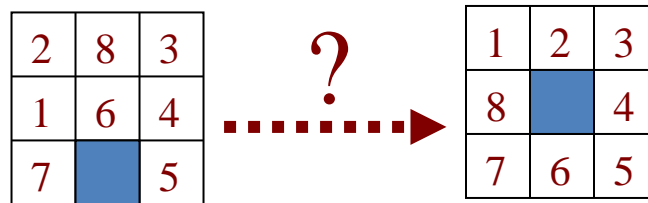
- Estimer un nombre minimal de coups (actions)



- Nombre de tuiles mal placées :
 - Tuiles 2, 8, 1, 6
 - Donc, $h = 4$

Heuristiques pour jeu de taquin

- Estimer un nombre minimal de coups (actions)



État/Noeud n

But

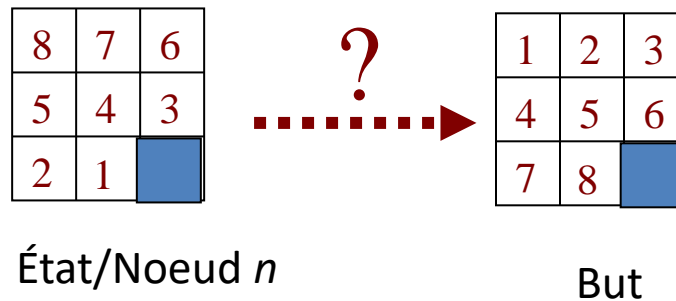
- Sommes des distances de Manhattan des tuiles avec leur position à atteindre :

| Tuile | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|
| Distance | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 2 |

$$h = 5$$

Heuristiques pour jeu de taquin

- Estimer un nombre minimal de coups (actions)



- Sommes des distances avec buts:

| Tuile | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|---|---|---|---|---|---|---|---|
| Distance | 3 | 3 | 1 | 1 | 1 | 1 | 3 | 3 |

$$h = 16$$

PROPRIÉTÉS DE L'ALGORITHME A*

Propriétés de l'algorithme A*

- Si un problème n'a pas de solution :
 - Tout l'espace d'états accessibles depuis l'état initial sera visité.
 - Une fois open vide, le constat d'absence de solution est confirmée.
- Si une solution existe, l'algorithme A* la trouve toujours.

Propriétés de l'algorithme A*

- Si la fonction heuristique h retourne toujours une estimation inférieure au coût réel, on dit que h est une **fonction heuristique admissible**.
- Lors qu'utilisé avec une heuristique admissible, l'algorithme A* retourne toujours une solution optimale si elle existe.
- Attention : il peut y avoir plusieurs solutions optimales (mais de même coût).

Propriétés de l'algorithme A*

- Complet? **Oui**, à condition d'avoir un nombre fini d'états.
- Complexité temporelle?
 - Dépend de l'heuristique.
 - Dans le pire cas, il faut visiter n nœuds ...
 - Mais, la taille de l'espace d'états (n) est généralement exponentielle à la taille du problème!
- Complexité spatiale? **O(n)** où n est le nombre de nœuds explorés. Dans le pire cas $n=|S|$.
- Optimal? **Oui**, à condition que l'heuristique soit admissible.

Propriétés de l'algorithme A*

- Équivalence A* et recherche en largeur
 - En utilisant des coûts des arcs uniformément égaux et strictement positifs (par exemple, tous égaux à 1) et h retournant toujours 0 quelque soit le nœud, A* devient une recherche en largeur.
 - *Open* devient une queue LIFO (*last in, last out*), en d'autres termes « dernier entré, dernier sorti ».

Propriétés de l'algorithme A*

- Soit $f^*(n)$, le coût d'un chemin optimal passant par n . Pour chaque nœud exploré par A*, on a toujours $f(n) \leq f^*(n)$.
- Si quelque soit un nœud $n1$ et son successeur $n2$, nous avons toujours $h(n1) \leq \text{coût}(n1, n2) + h(n2)$, où $\text{coût}(n1, n2)$ est le coût de l'arc $(n1, n2)$, on dit que h est une heuristique **consistante** (on dit aussi parfois **monotone** – mais c'est en réalité f qui devient monotone). Dans ce cas,
 - h est aussi admissible.
 - Chaque fois que A* choisit un nœud au début de open, cela veut dire que A* a déjà trouvé un chemin optimal vers ce nœud: le nœud ne sera plus jamais revisité!
 - Si une heuristique est non consistante, pour que A* soit optimal, il faut vérifier la présence des nœuds générés dans closed et les réinsérer dans open au besoin.

Propriétés de l'algorithme A*

- Si on a deux heuristiques *admissibles* h_1 et h_2 , tel que $h_1(n) < h_2(n)$, alors $h_2(n)$ conduit plus vite au but: avec h_2 , A* explore moins ou autant de nœuds avant d'arriver au but qu'avec h_1 .
- Si h n'est pas admissible, soit x la borne supérieur sur la surestimation du coût. C-à-d., on a toujours $h(n) \leq h^*(n) + x$. Dans ce cas A* retournera une solution dont le coût est au plus x plus que le coût optimal, c-à-d., A* ne se trompe pas plus que x sur l'optimalité.

Mini-quiz sur A*

- Étant donné une fonction heuristique non admissible, l'algorithme A* donne toujours une solution lorsqu'elle existe, mais il n'y a pas de certitude qu'elle soit optimale.
- Si les coûts des arcs sont tous égaux à 1 et la fonction heuristique retourne tout le temps 0, alors A* retourne toujours une solution optimale lorsqu'elle existe.
- Lorsque la fonction de transition contient des boucles et que la fonction heuristique n'est pas admissible, A* peut boucler indéfiniment même si l'espace d'états est fini.
- Avec une heuristique monotone, A* n'explore jamais le même état deux fois.

Mini-quiz sur A*

- Étant donné deux fonctions heuristiques h_1 et h_2 telles que $0 \leq h_1(s) < h_2(s) \leq h^*(s)$, pour tout état s , h_2 est plus efficace que h_1 dans la mesure où les deux mènent à une solution optimale, mais h_2 le fait en explorant moins ou autant de nœuds.
- Si $h(s) = h^*(s)$, pour tout état s , l'optimalité de A* est garantie.

Variations de A*

- Selon le poids que l'on veut donner à l'une ou l'autre partie, on définit f comme suit:

$$f(n) = (1-w)*g(n) + w*h(n)$$

où w est un nombre réel supérieur ou égal à 0 et inférieur ou égal à 1.

- Selon les valeurs qu'on donne à w , on obtient des algorithmes de recherche classique:
 - **Dijkstra** : $w = 0$ c'est-à-dire $f(n) = g(n)$
 - **Best-first-search** : $w = 1$ c'est-à-dire $f(n) = h(n)$
 - **A*** : $w = 0.5$ équivalent à $f(n) = g(n) + h(n)$

Variations de A*

- **Beam search**
 - On met une limite sur le contenu de OPEN et CLOSED
 - Recommandé lorsque pas assez d'espace mémoire.
- **Bit-state hashing**
 - **closed** est implémenté par une table hash et on ignore les collisions
 - Utilisé dans la vérification des protocoles de communication, mais avec une recherche en profondeur classique (pas A*).
 - *Exemple*: outil SPIN

Variations de A*

- **Iterative deepening**
 - On met une limite sur la profondeur
 - On lance A* jusqu'à la limite de profondeur spécifiée.
 - Si pas de solution on augmente la profondeur et on recommence A*
 - Ainsi de suite jusqu'à trouver une solution.
- **And-Or A***
 - Fait pour les graphes ET-OU
- **D*** (proposée par Stenz et al.).
 - A* Dynamique. Évite de refaire certains calculs lorsqu'il est appelé plusieurs fois pour atteindre le même but, suite à des changements de l'environnement.

Variations de A*

- **Anytime Dynamic A***

- Similaire au *iterative deepening*, en variant le poids de l'heuristique.

- Référence : Maxim Likhachev, David Ferguson, Geoffrey Gordon, Anthony (Tony) Stentz, and Sebastian Thrun. **Anytime Dynamic A*: An Anytime, Replanning Algorithm**. Dans Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), June, 2005.