IFT 615 – Intelligence artificielle

Recherche heuristique

Hugo Larochelle
Département d'informatique
Université de Sherbrooke
http ://www.dmi.usherb.ca/~larocheh/cours/ift615.html

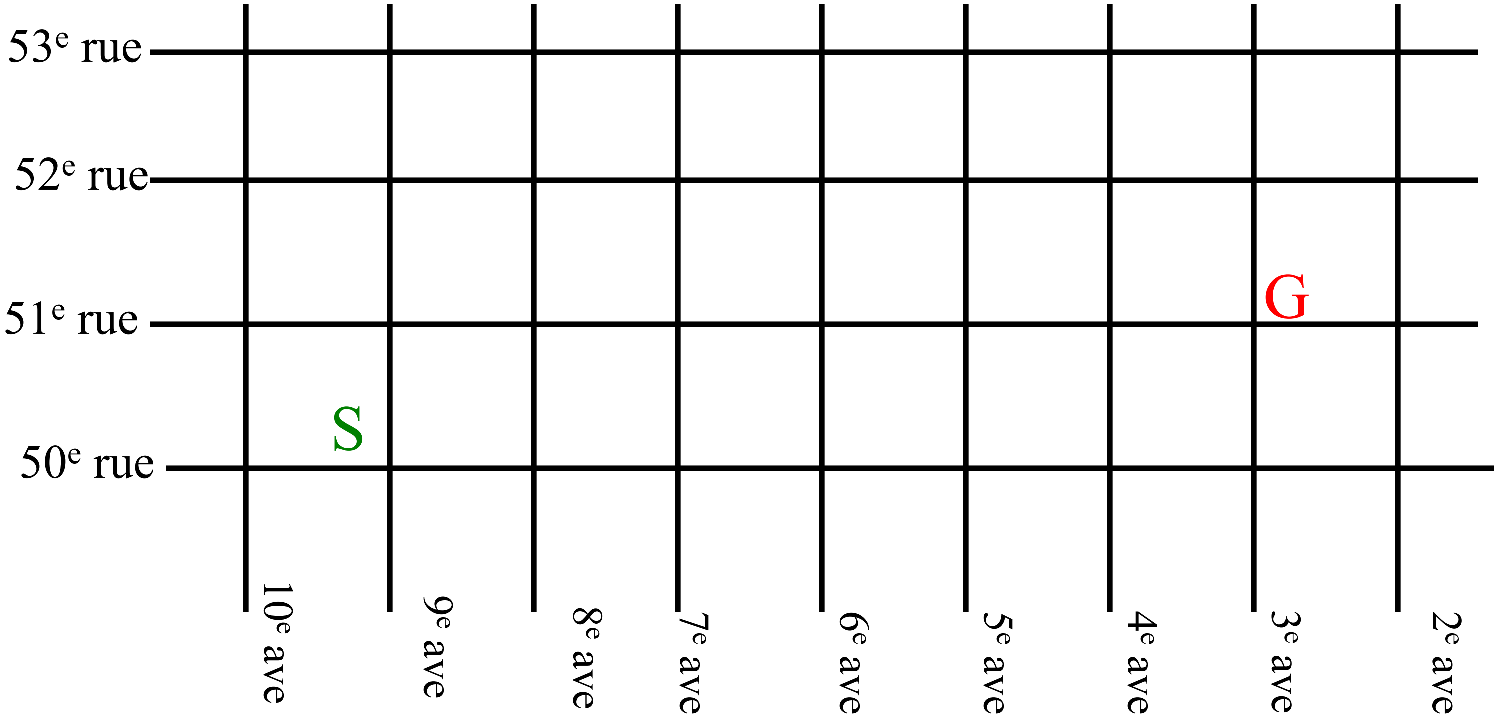
Objectifs

- Résolution de problème par recherche
- Rappel de A* vu en IFT 436
- Comprendre A*
- Implémenter A*
- Appliquer A* à un problème donné
- Comprendre la notion d'heuristique

Exemple : trouver chemin dans ville

Trouver un chemin de la 9e ave & 50e rue à la 3e ave et 51e rue

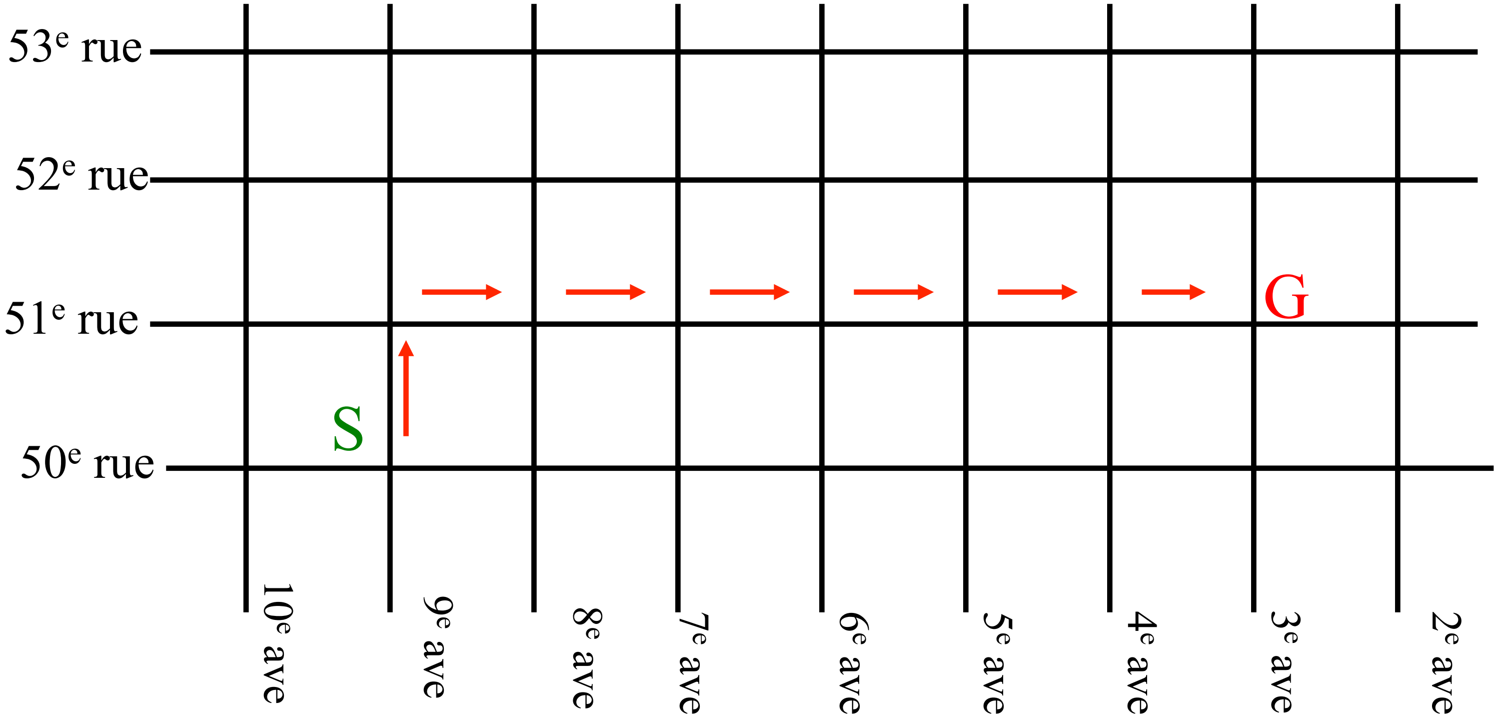
(Illustration par Henry Kautz, U. of Washington)



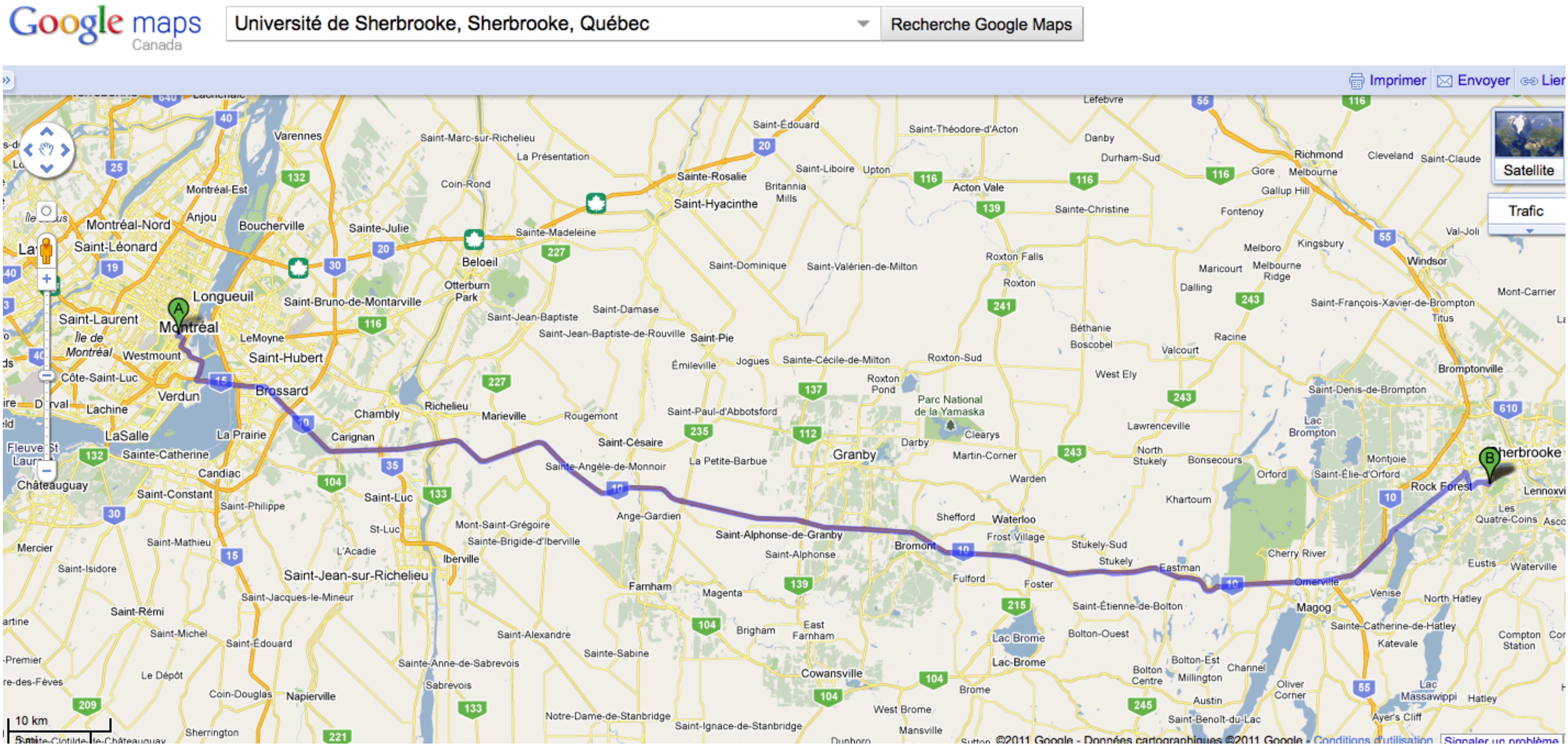
Exemple : trouver chemin dans ville

Trouver un chemin de la 9e ave & 50e rue à la 3e ave et 51e rue

(Illustration par Henry Kautz, U. of Washington)

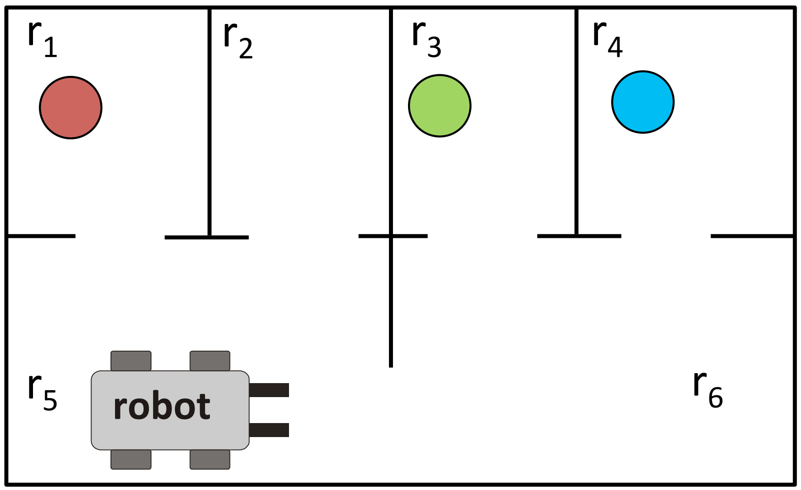


Exemple : Google Maps

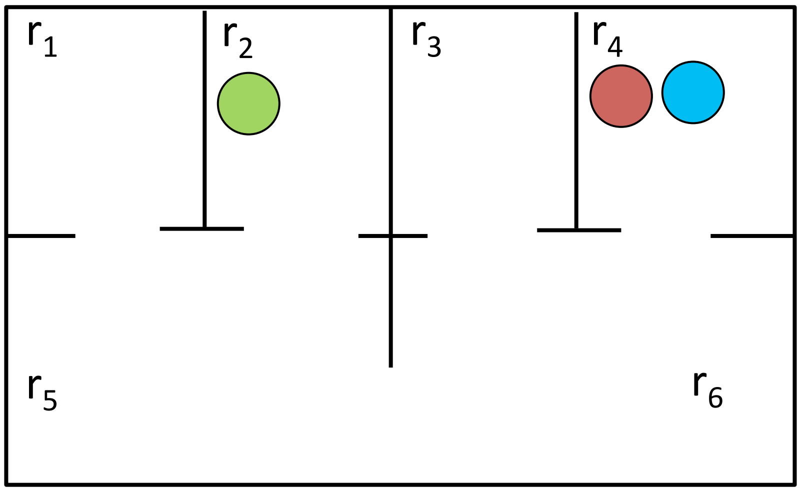


Exemple : livrer des colis

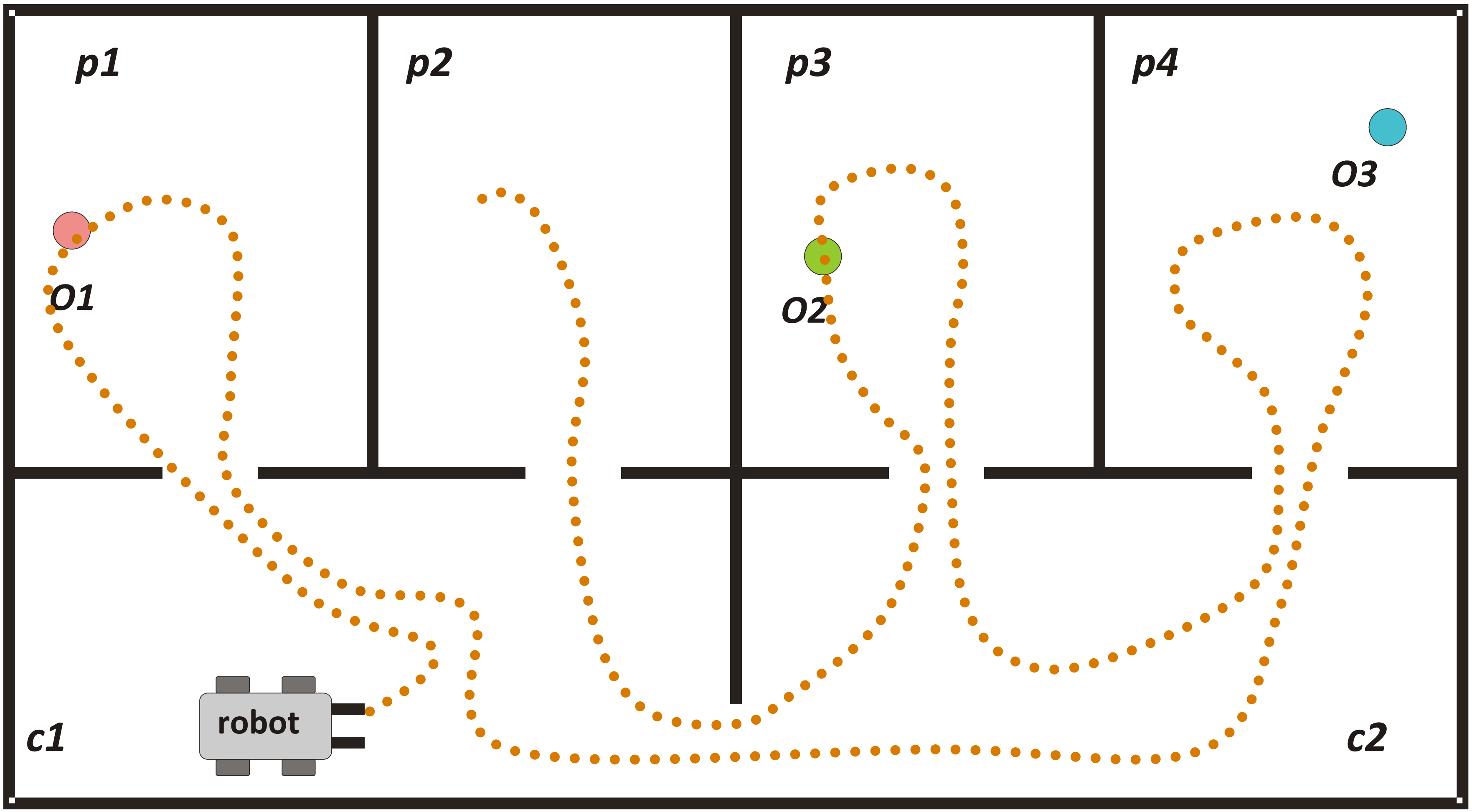
État initial



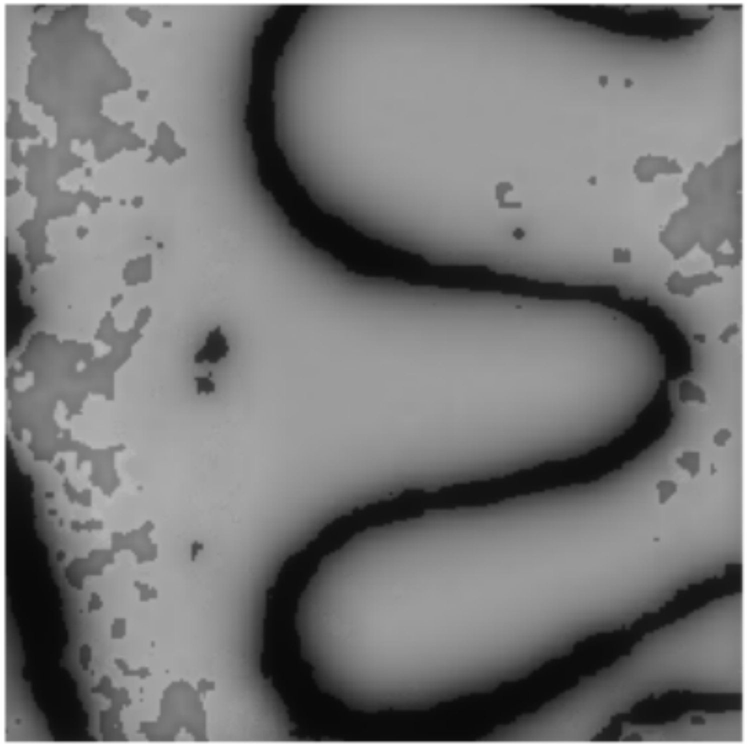
But



Exemple : livrer des colis

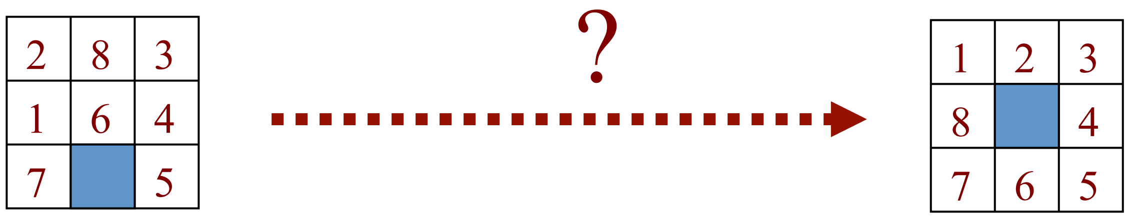


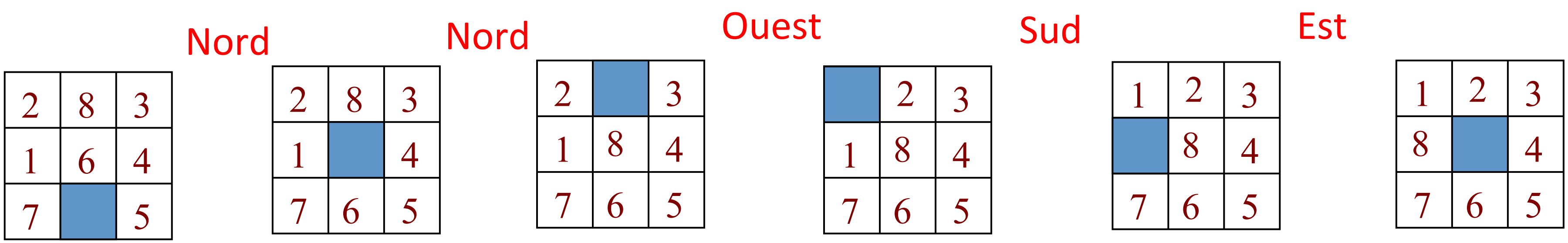
Exemple : navigation d'un robot



(Ratliff, Bagnell et Zinkevich, 2006)

Exemple : N-Puzzle





Résolution de problèmes

- Étapes intuitives par un humain
  1. modéliser la situation actuelle
  2. énumérer les solutions possibles
  3. évaluer la valeur des solutions
  4. retenir la meilleure option possible satisfaisant le but
- Mais comment parcourir efficacement la liste des solutions?

- La résolution de beaucoup de problèmes peut être faite par **une recherche dans un graphe**
  - chaque noeud correspond à un état de l'environnement
  - chaque chemin à travers un graphe représente alors une suite d'actions prises par l'agent
  - pour résoudre notre problème, suffit de chercher le chemin qui satisfait le mieux notre mesure de performance

Résolution de problème par une recherche heuristique dans un graphe

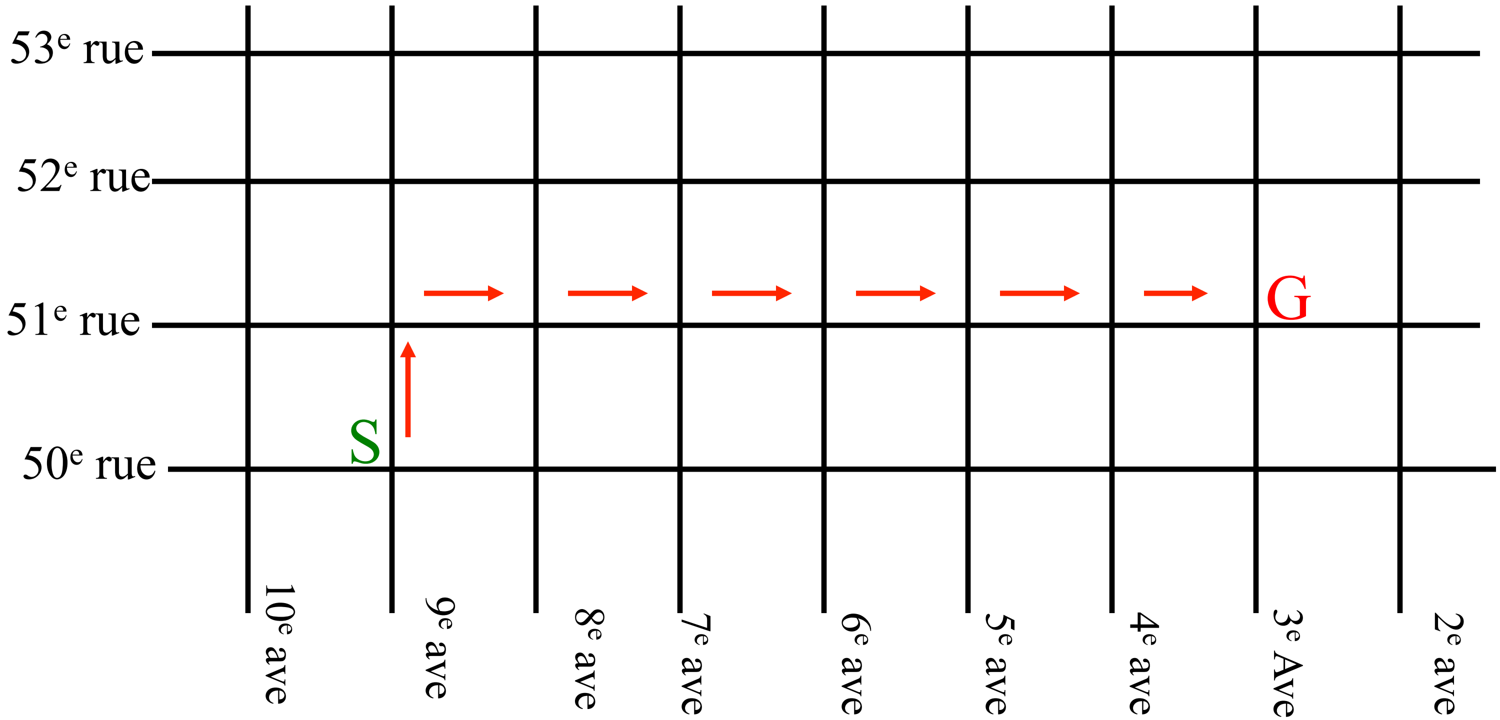
- La recherche heuristique est à la base de beaucoup d'approches en IA
- Le graphe est défini récursivement (plutôt qu'explicitement)
- Une heuristique est utilisée pour guider la recherche :
  - les heuristiques exploitent les connaissances du domaine d'application

Problème de recherche dans un graphe

- Algorithme de recherche dans un graphe
  - Entrées :
    - un nœud initial
    - une fonction *goal*(*n*) qui retourne *true* si le but est atteint
    - une fonction de transition *transitions*(*n*) qui retourne les nœuds successeurs de *n*
    - une fonction *c*(*n*,*n*') strictement positive, qui retourne le coût de passer de *n* à *n*' (permet de considérer le cas avec coûts variables)
  - Sortie :
    - un chemin dans un graphe (séquence nœuds / arrêtes)
  - Le **coût d'un chemin** est la **somme des coûts des arrêtes** dans le graphe
  - Il peut y avoir plusieurs nœuds qui satisfont le but
- Enjeux :
  - trouver un chemin solution, ou
  - trouver un chemin optimal, ou
  - trouver rapidement un chemin (optimalité pas importante)

Exemple : graphe d'une ville

- Nœuds = intersections
- Arrêtes = segments de rue



(Illustration par Henry Kautz, U. of Washington)

Exemple : trouver chemin dans une ville

Domaine :

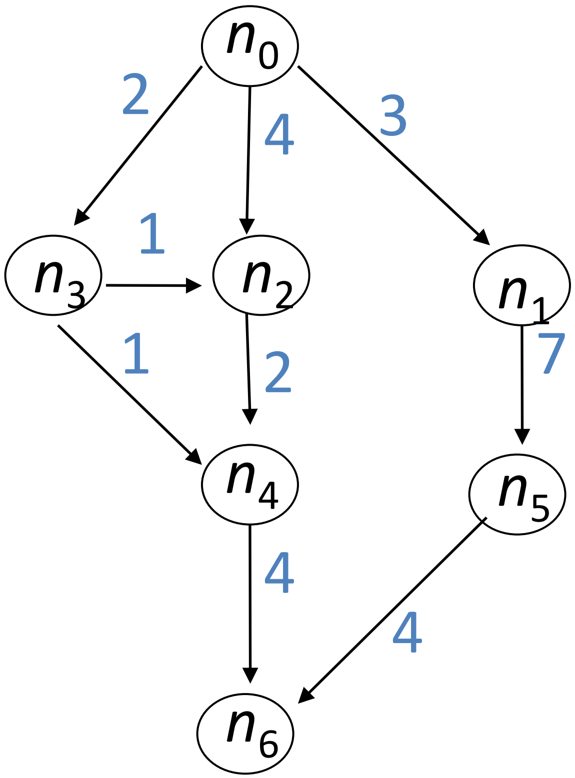
Routes entre les villes :
transitions(n_0) = (n_3, n_2, n_1)

Distance entre les villes :
$c(n_0,n_2) = 4$

Problème posé (initNode, goal) :

n_0 : ville de départ (état initial)
n_6 : destination (but)

En d'autres termes :
goal(n) : vrai si $n=n_6$



Rappel sur les algorithmes de recherche dans des graphes

- Recherche sans heuristique et coût uniforme
  - Recherche en profondeur (*Depth-First-Search*)
    - » pour un noeud donné, explore le premier enfant avant d'explorer un noeud frère
  - Recherche en largeur (*Breadth-First-Search*)
    - » pour un noeud donné, explore les noeuds frères avant leurs enfants

- Recherche sans heuristique et coût variable
  - Algorithme de Dijkstra
    - » trouve le chemin le plus court entre un noeud source et tous les autres noeuds

- Recherche avec heuristique et coût variable :
  - *Best-First-Search*
  - *Greedy Best-First-Search*
  - **A***

Algorithme A*

- A* est une extension de l'algorithme de Dijkstra
  - utilisé pour trouver un chemin optimal dans un graphe via l'**ajout d'une heuristique**
- Une **heuristique** $h(n)$ est une fonction d'**estimation du coût entre un nœud** ***n*** **d'un graphe et le but** (le nœud à atteindre)
- Les heuristiques sont à la base de beaucoup de travaux en IA :
  - recherche de meilleures heuristiques
  - apprentissage automatique d'heuristiques
- Pour décrire A*, il est pratique de décrire un algorithme générique très simple, dont A* est un cas particulier

Variables importantes : *open* et *closed*

- *Open* contient les nœuds non encore traités, c'est à dire à la frontière de la partie du graphe explorée jusque là
- *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 *n* dans *open* sont triés selon l'estimé *f*(*n*) de leur « valeur »
  - on appelle *f*(*n*) une **fonction d'évaluation**

- Pour chaque nœud *n*, *f(n)* est un nombre réel positif ou nul, **estimant le coût du meilleur chemin partant de la racine, passant par *n*, et arrivant au but**

- Dans *open*, les nœuds se suivent en ordre croissant selon les valeurs *f(n).*
  - le tri se fait par insertion : on s'assure que le nouveau nœud va au bon endroit
  - on explore donc les noeuds les plus « prometteurs » en premier

Définition de *f*

- La **fonction d'évaluation** *f*(*n*) tente d'estimer le coût du chemin optimal entre le nœud initial et le but, et qui passe par *n*
- En pratique on ne connaît pas ce coût : c'est ce qu'on cherche !
- À tout moment, on connaît seulement le coût optimal **pour la partie explorée** entre la racine et un nœud **déjà exploré**
- Dans A*, on sépare le calcul de *f(n)* en deux parties :
  - *g(n)* : coût du meilleur chemin ayant mené au noeud *n* depuis la racine
    - » c'est le coût du meilleur chemin **trouvé jusqu'à maintenant** qui se rend à *n*
  - *h(n) :* coût **estimé** du reste du chemin optimal partant de *n* jusqu'au but. *h(n)* est la **fonction heuristique**
    - » on suppose que *h*(*n*) est non négative et *h*(*n*) = 0 si n est le noeud but

Exemples de fonctions heuristiques

- Chemin dans une ville
  - distance **Euclidienne** ou distance de **Manhattan** pour un chemin sur une carte
  - éventuellement pondéré par la qualité des routes, le prix du billet, etc.
- Probabilité d'atteindre l'objectif en passant par le nœud
- Qualité de la configuration d'un jeu par rapport à une configuration gagnante
- N-Puzzle
  - nombre de tuiles mal placées
  - somme des distances des tuiles

Algorithme générique de recherche dans un graphe

Algorithme rechercheDansGraphe(*noeudInitial*)

1. déclarer deux nœuds : *n*, *n'*
2. déclarer deux listes : *open*, *closed* // *toutes les deux sont vides au départ*
3. insèrer *noeudInitial* dans *open*
4. tant que (1) // *la condition de sortie (exit) est déterminée dans la boucle*
    5. si *open* est vide, sortir de la boucle avec échec
    6. *n* = noeud au début de open*;*
    7. enlever *n* de *open* et l'ajouter dans *closed*
    8. si *n* est le but, sortir de la boucle avec succès en retournant le chemin;
    9. pour chaque successeur n' de n
        10. initialiser la valeur *g(n') à : g(n) + le coût de la transition (n,n')*
        11. mettre le parent de *n'* à *n*
        12. si *closed* ou *open* contient un nœud *n''* égal à *n'* avec *f(n') ≤f(n'')*
            13. enlever *n''* de *closed* ou *open* et insérer *n'* dans *open* (ordre croissant selon *f*(*n*))
        14. si *n'* n'est ni dans *open* ni dans *closed*
            15. insérer *n'* dans *open* en triant les nœuds en ordre croissant selon *f*(*n*)

Exemple A* avec recherche dans une ville

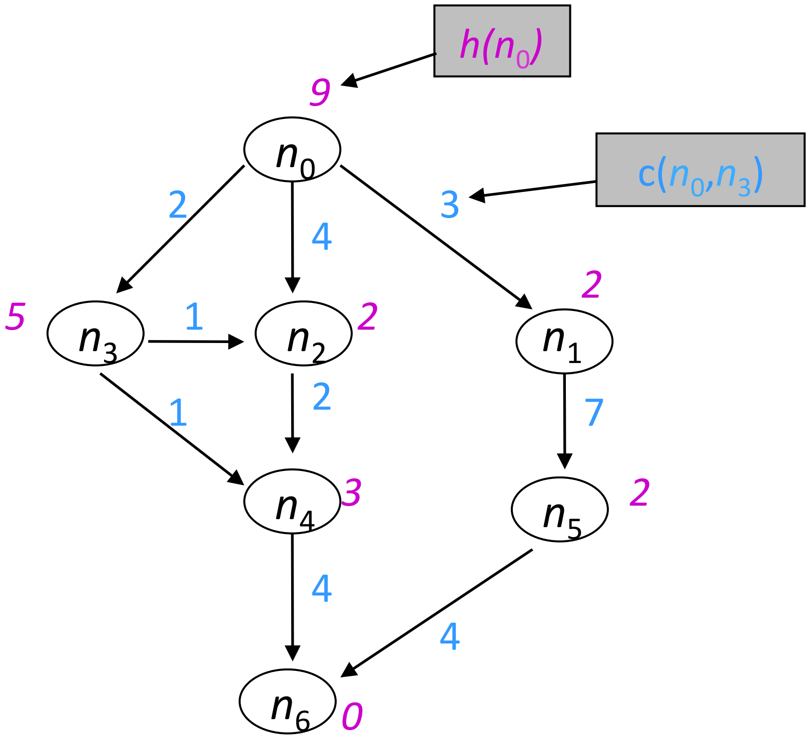
Routes entre les villes :

n_0 : ville de départ

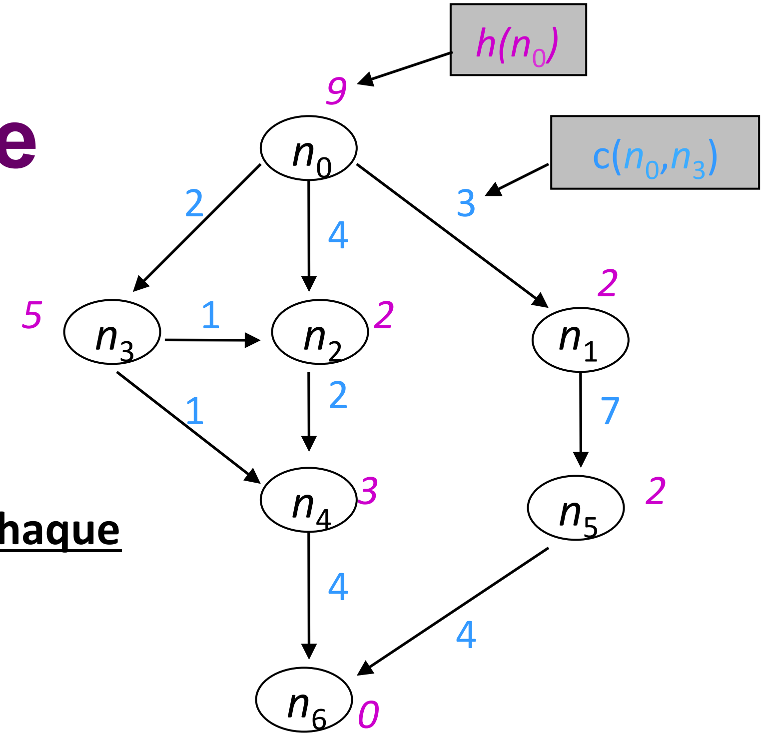
n_6 : destination

h : distance à vol d'oiseau

c : distance réelle entre deux ville



Exemple A* avec recherche dans une ville



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

1. (n_0, 9, void)
2. (n_1,5,n_0), (n_2,6,n_0), (n_3,7,n_0)
3. (n_2,6,n_0), (n_3,7,n_0), (n_5,12,n_1)
4. (n_3,7,n_0), (n_4,9,n_2), (n_5,12,n_1)
5. (n_2,5,n_3), (n_4,6,n_3), (n_5,12,n_1)
6. (n_4,6,n_3), (n_5,12,n_1)
7. (n_6,7,n_4), (n_5,12,n_1)
8. Solution : $\boldsymbol{n_0,n_3,n_4,n_6}$

Contenu de *closed* à chaque itération :

1. Vide
2. (n_0, 9, void)
3. (n_0, 9, void), (n_1,5,n_0)
4. (n_0, 9, void), (n_1,5,n_0), (n_2,6,n_0)
5. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0)
6. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3)
7. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3)
8. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3), (n_6,7,n_4)

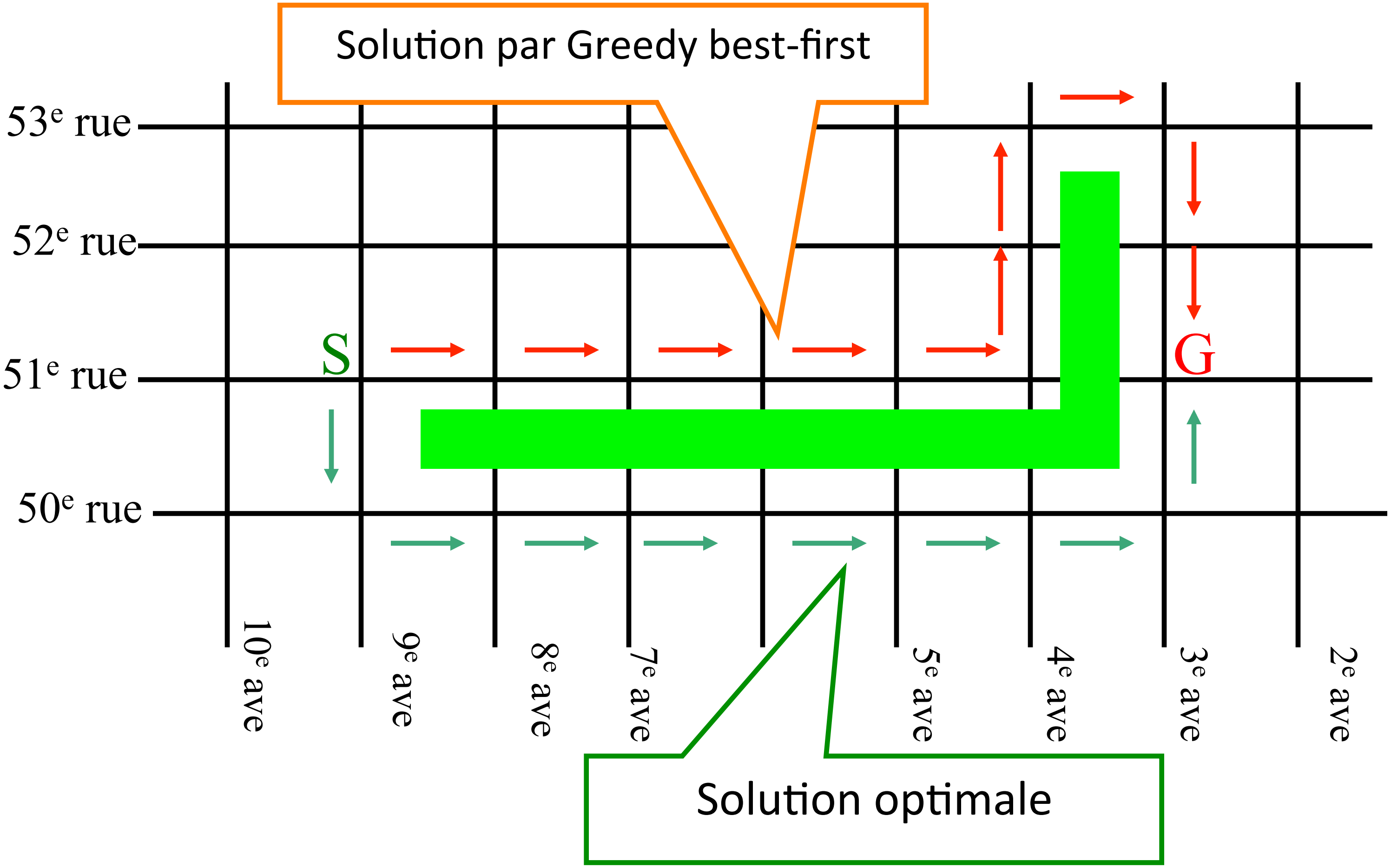
D’autres algorithmes de recherche heuristique

- *Best-First-Search*
  - variante plus générale où f peut prendre une forme quelconque
  - A* est un cas spécial de *Best-First-Search*, où $f(n) = g(n) + h(n)$

- *Greedy Best-First-Search*
  - c’est un *Best-First-Search* où $f(n) = h(n)$
  - n’est pas garanti de trouver un chemin qui est optimal, mais marche parfois bien en pratique

Non-optimalité de Greedy best-First Search

(Illustration par Henry Kautz, U. of Washington)



Démo d'algorithmes de recherche dans un espace d'états

A*, Profondeur, Largeur, Best-First

http ://planiart.usherbrooke.ca/~eric/ift615/demos/search/search.html

Propriétés de A*

- Si le graphe est fini, A* termine toujours
- Si un chemin vers le but existe, A* va en trouver un
- Si la fonction heuristique *h* retourne toujours un **estimé inférieur ou égal au coût réel à venir**, on dit que *h* est **admissible** :
  - dans ce cas, **A* retourne toujours un chemin optimal**
- *Parfois, on entend par A* la version de l'algorithme avec la condition additionnelle que h soit admissible*
  - A* est alors un *Best-First-Search* où *f(n) = g(n) + h(n)* **et** *h(n)* est admissible

Propriétés de A* : recherche en largeur

- En utilisant des coûts des arcs uniformément égaux et strictement positifs (par exemple, tous égaux à 1) et $h(n)$ retournant toujours 0 quelque soit le nœud n, A* devient une recherche en largeur

- *Open* devient une queue LILO (*last in, last out),* en d'autre termes « dernier entré, dernier sorti »

Propriétés de A*

- Soit $f^*(n)$ le coût exact (pas un estimé) **du chemin optimal** du noeud initial au noeud but, **passant par *n***

- Soit $g^*(n)$ le coût exact du **chemin optimal du noeud initial au noeud *n***

- Soit $h^*(n)$ le coût exact du **chemin optimal du noeud *n* au noeud but**

- On a donc que $f^*(n) = g^*(n) + h^*(n)$

- Si l'heuristique est admissible, pour chaque nœud n exploré par A*, on peut montrer que l'on a toujours $f(n) \leq f^*(n)$

Propriétés de A*

- Si quelque soit un nœud n_1 et son successeur n_2, nous avons toujours

$$h(n_1) \leq c(n_1,n_2) + h(n_2)$$

où $c(n_1,n_2)$ est le coût de l'arc (n_1,n_2).

On dit alors que h est **cohérente** (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!

Propriétés de 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 b la borne supérieure sur la surestimation du coût, c-à-d. on a toujours $h(n) \leq h^*(n) + b$:
  - A* retournera une solution dont le coût est au plus b de plus que le coût optimal, c-à-d., A* ne se trompe pas plus que b sur l'optimalité.

Test sur la compréhension de 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
  - Vrai
- 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
  - Vrai
- 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
  - Faux

Test sur la compréhension de A*

- Avec une heuristique monotone, A* n'explore jamais le même état deux fois.
  - Vrai
- Étant donné deux fonctions heuristiques h_1 et h_2 telles que $0 \leq h_1(n) < h_2(n) \leq h^*(n)$, pour tout état n, 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 de nœuds
  - Vrai.
- Si $h(n) = h^*(n)$, pour tout état n, l'optimalité de A* est garantie
  - Vrai

Définition générique de f

- Selon le poids que l'on veut donner à l'une ou l'autre partie, on définie 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*** : $w = 0$ ($f(n) = g(n)$)
  - ***Greedy best-first search*** : $w = 1$ ($f(n) = h(n)$)
  - ***A**** : $w = 0.5$ ($f(n) = g(n) + h(n)$)

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*

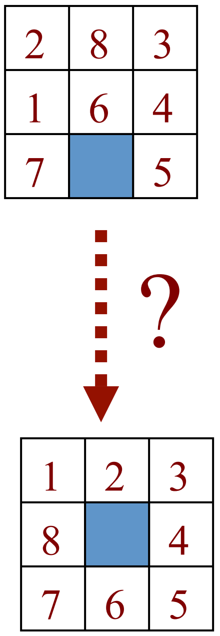
- Iterative deepening
  - on met une limite sur la profondeur
  - on lance A* jusqu'à la limite de profondeur spécifiée.
  - si pas de solution on augmente la profondeur et on recommence A*
  - ainsi de suite jusqu'à trouver une solution.

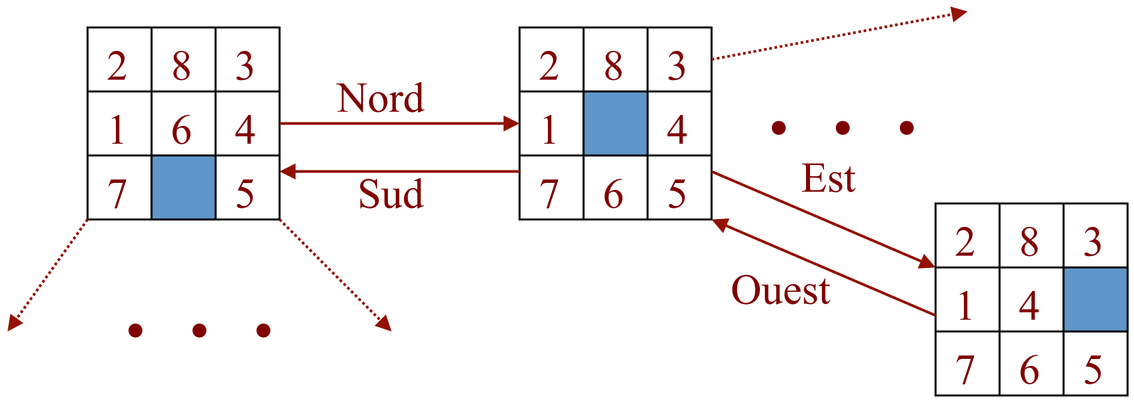
- Recursive best-first search (RBFS) et simplified memory-bounded A* (SMA*)
  - variantes de A* qui utilisent moins de mémoire mais peuvent être plus lentes

- D* (inventé par Stenz et ses collègues).
  - A* dynamique, où le coût des arrêtes peut changer durant l'exécution. Évite de refaire certains calculs lorsqu'il est appelé plusieurs fois pour atteindre le même but, suite à des changements de l'environnement.

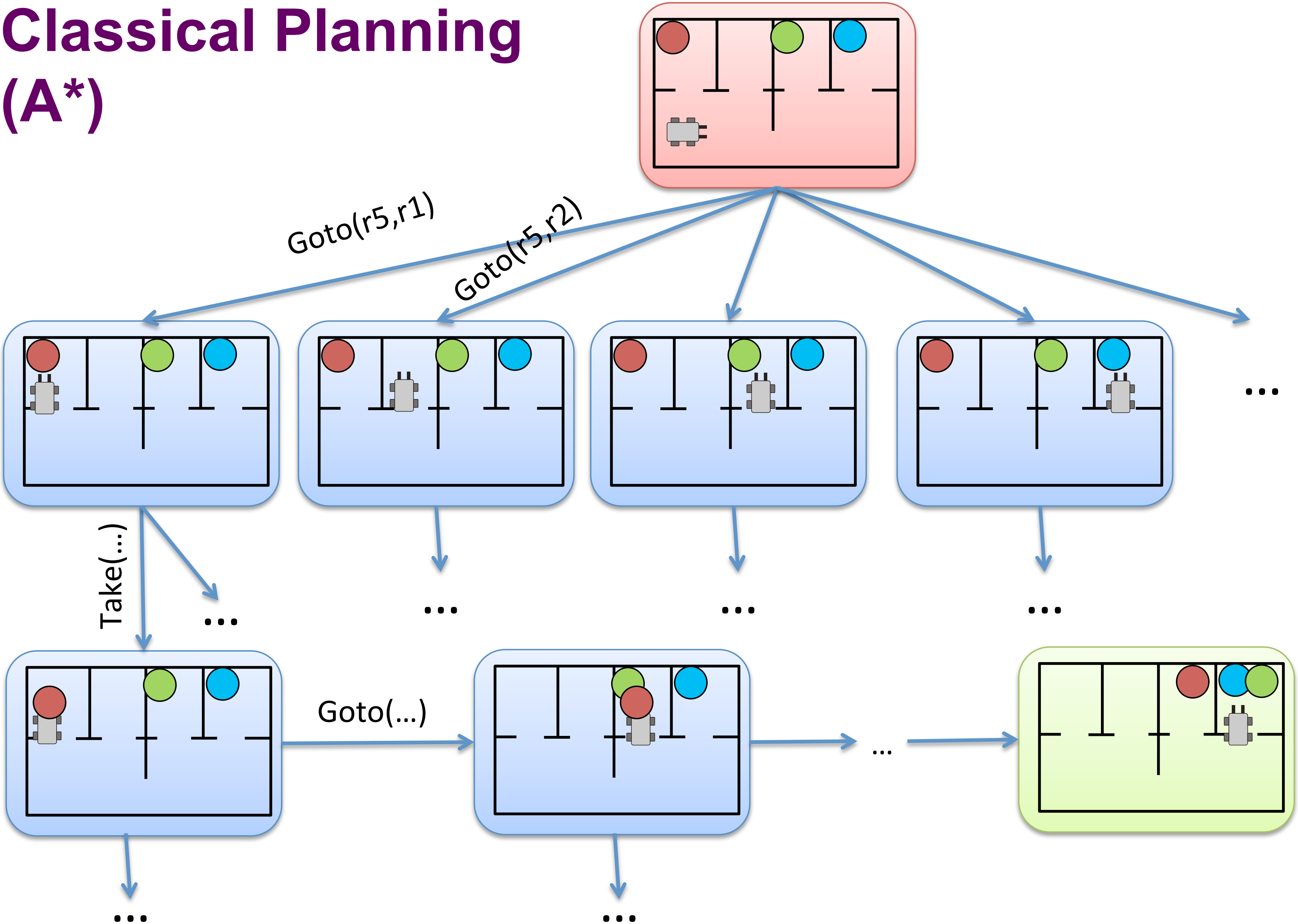
Exemple académique

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

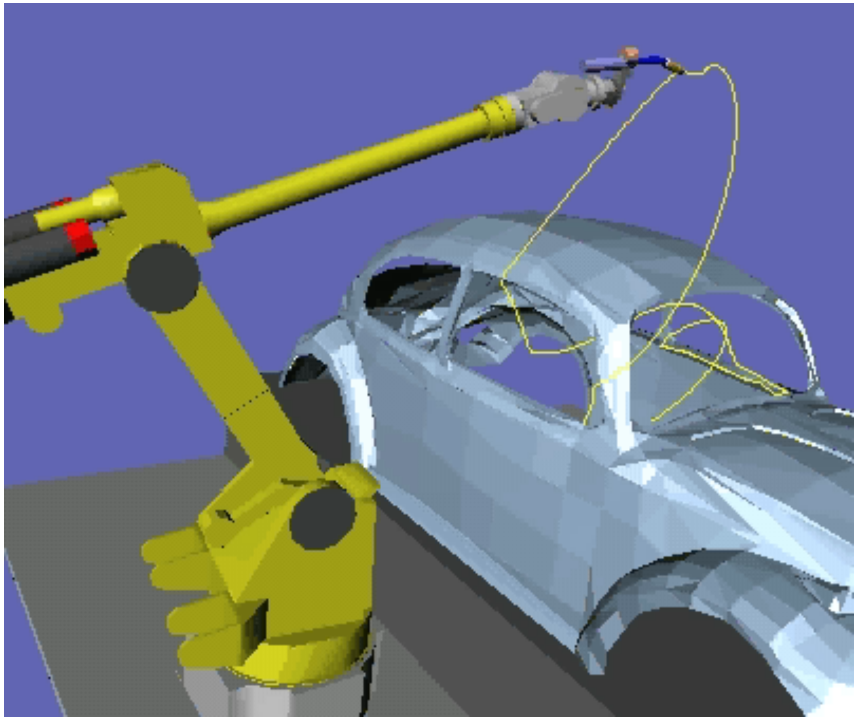


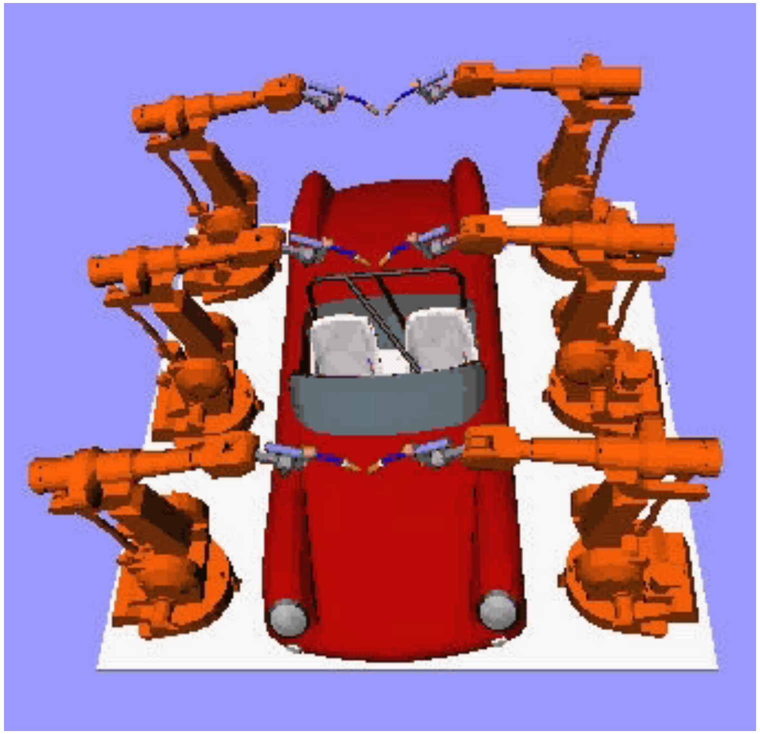


Classical Planning (A*)



Application : industrie automobile





Démos du Motion Planning Kit (Jean-Claude Latombe)

Application : jeux vidéos et cinéma

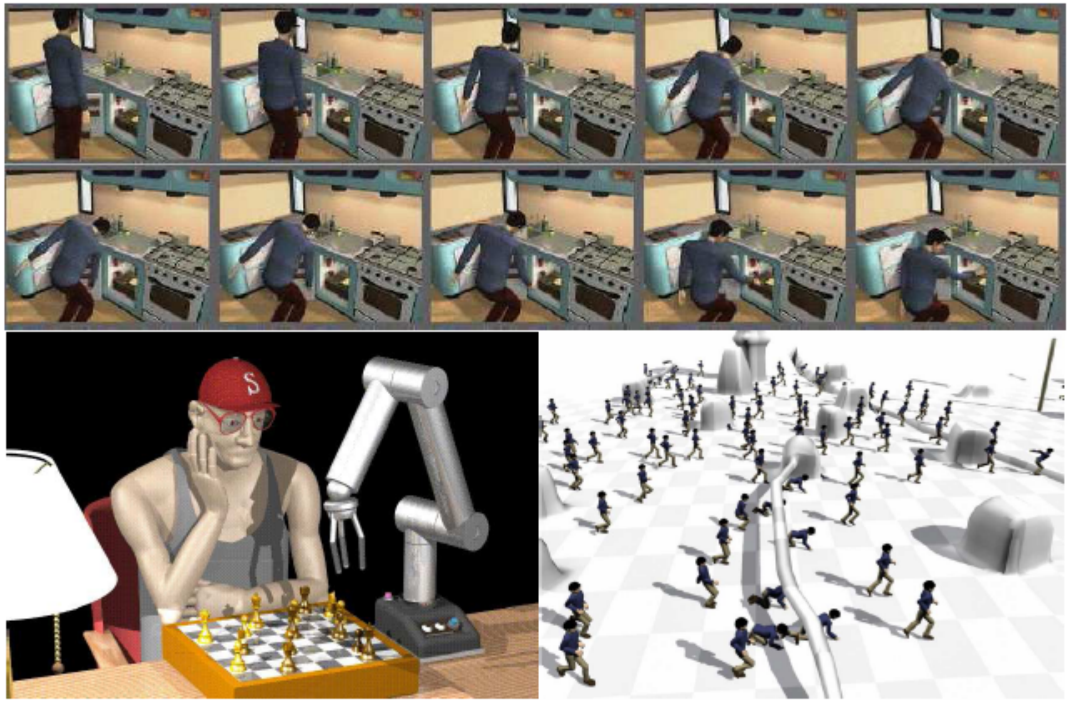


Figure 1.8: Across the top, a motion computed by a planning algorithm, for a digital actor to reach into a refrigerator [499]. In the lower left, a digital actor plays chess with a virtual robot [545]. In the lower right, a planning algorithm computes the motions of 100 digital actors moving across terrain with obstacles [592]. [Steven LaValle. *Planning Algorithms*]

Énoncé du problème

- Calculer une trajectoire géométrique d'un solide articulé sans collision avec des obstacles statiques.

Entrée :

- Géométrie du robot et des obstacles
- Cinétique du robot (degrés de liberté)
- Configurations initiale et finale

↓

Planificateur de trajectoires

↓

Sortie :

- Une séquence continue de configurations rapprochées, sans collision, joignant la configuration initiale à la configuration finale

Cadre générale de résolution du problème

Problème continu

(espace de configuration + contraintes)

↓

Discrétisation

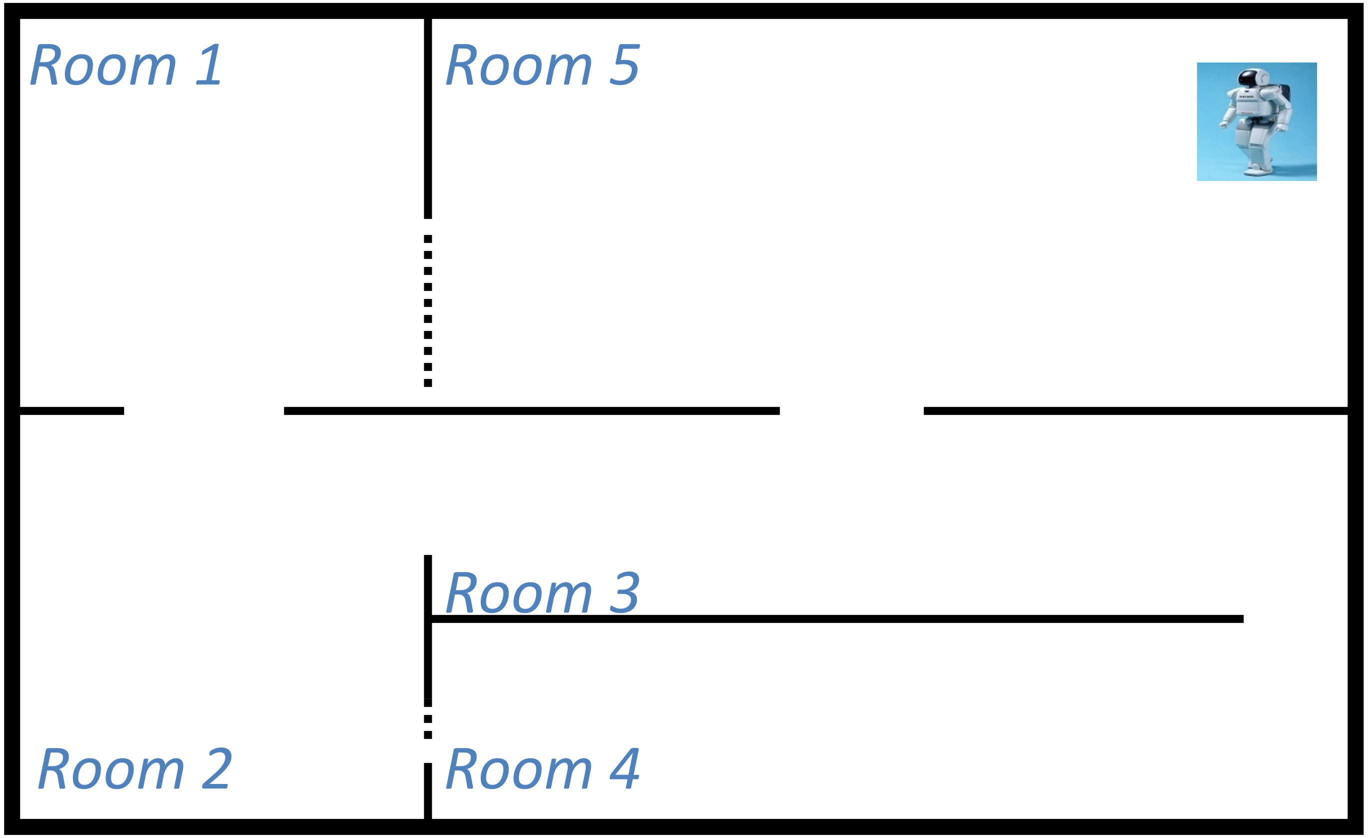
(décomposition, échantillonage)

↓

Recherche heuristique dans un graphe

(A* ou similaire)

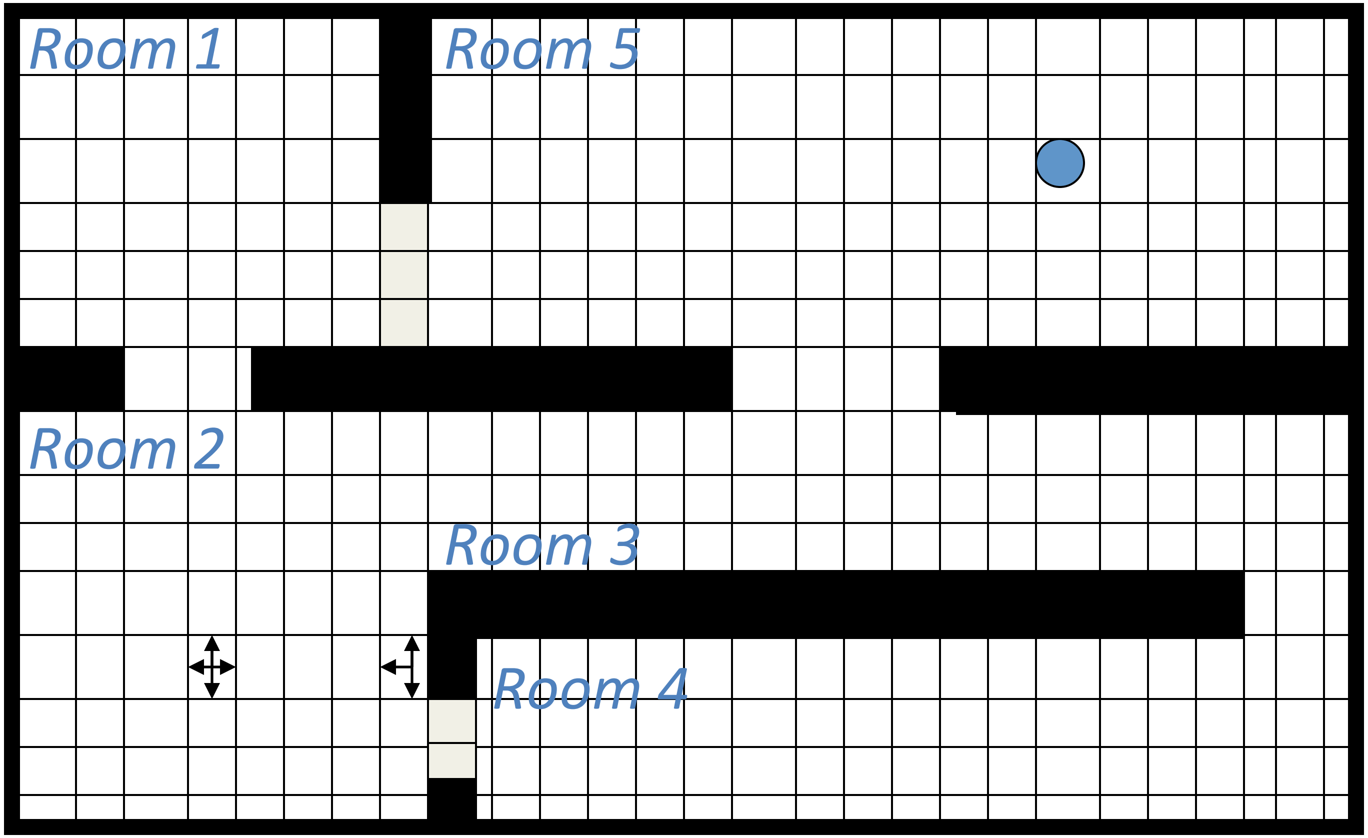
Approche combinatoire par décomposition en cellules



Décomposer la carte en grille (*occupancy grid*) :
4-connected (illustré ici) ou 8-connected.

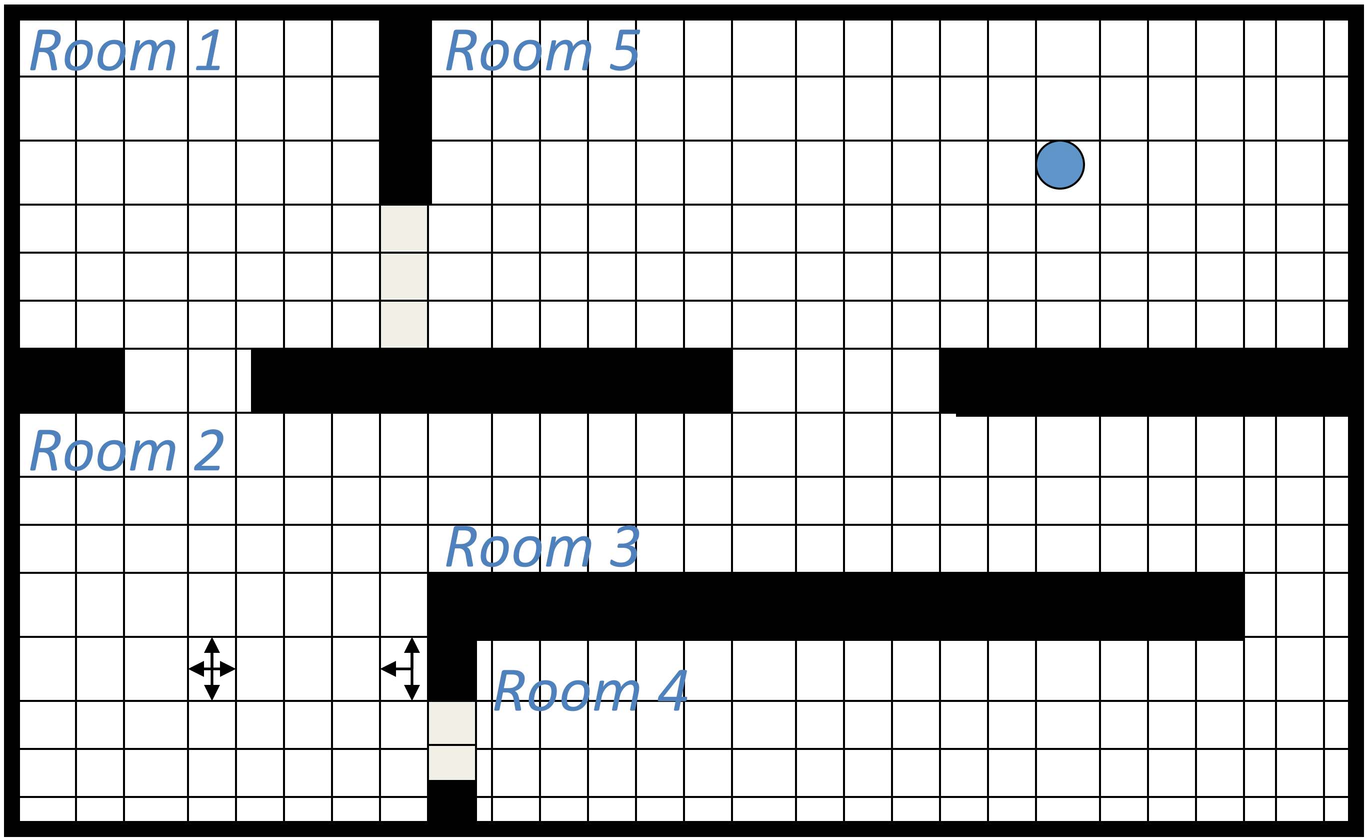
noeud : case occupée par le robot + orientation du robot

Transitions :
- Turn left →
- Turn right ←
- Go straight ahead



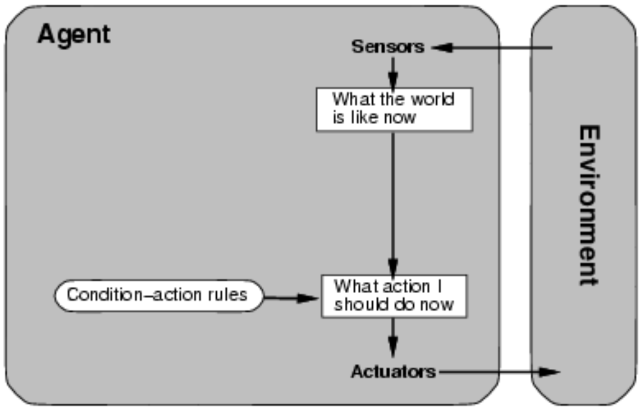
Heuristiques :
- Distance euclidienne, durée du voyage
- Consommation d'énergie ou coût du billet
- Degré de danger (chemin près des escaliers, des ennemis).

Go east =
(Turn right) +
Go straight ahead

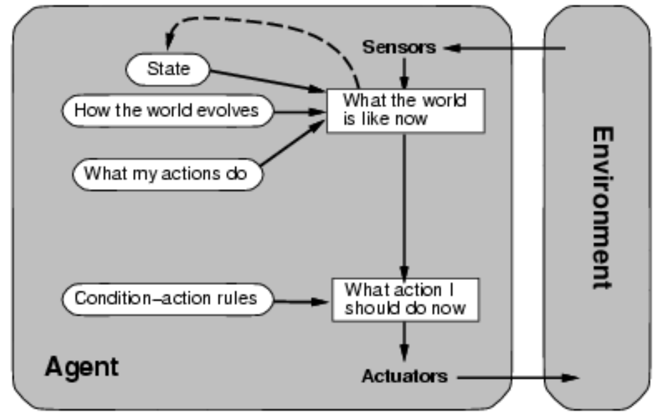


Recherche heuristique: pour quel type d'agent?

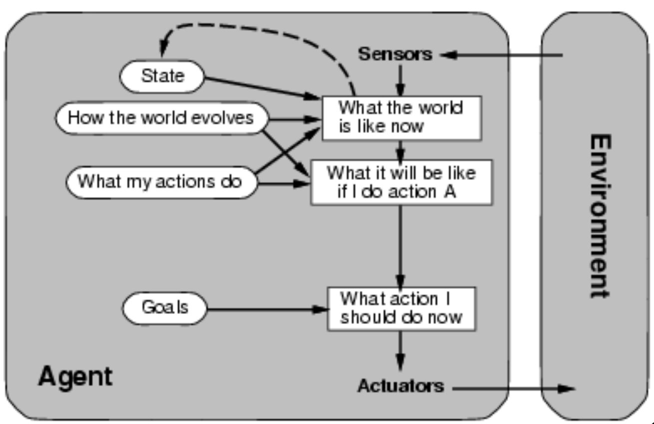
Simple reflex



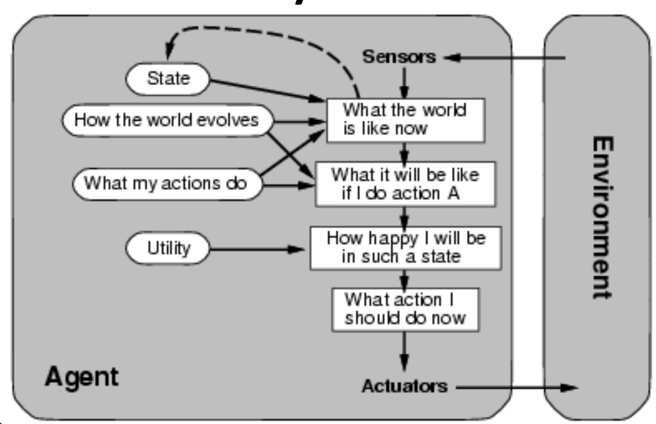
Model-based reflex



Goal-based



Utiliy-based



Conclusion

- La recherche heuristique est une approche fondamentale en IA
  - elle est assez flexible pour être appliquée à plusieurs problèmes

- A* est l'algorithme de recherche heuristique le plus connu et répandu

- Il a l'avantage d'avoir de garanties théoriques potentiellement intéressantes

- Par contre, le succès de A* dépend beaucoup de la qualité de l'heuristique $h(n)$ que l'on définit
  - une mauvaise heuristique peut augmenter considérablement les temps de calcul et l'espace mémoire nécessaire

Vous devriez être capable de...

- Comprendre le concept de recherche heuristique
  - qu'est-ce qu'une heuristique?
- Comprendre les différents concepts derrière A*
  - fonctions $f(n)$, $g(n)$ et $h(n)$, ainsi que $f^*(n)$, $g^*(n)$ et $h^*(n)$
- Identifier une heuristique admissible ou monotone
- Décrire les propriétés théoriques de A*
- Programmer/simuler l'exécution de A*