IFT 615 – Intelligence artificielle

Recherche heuristique

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

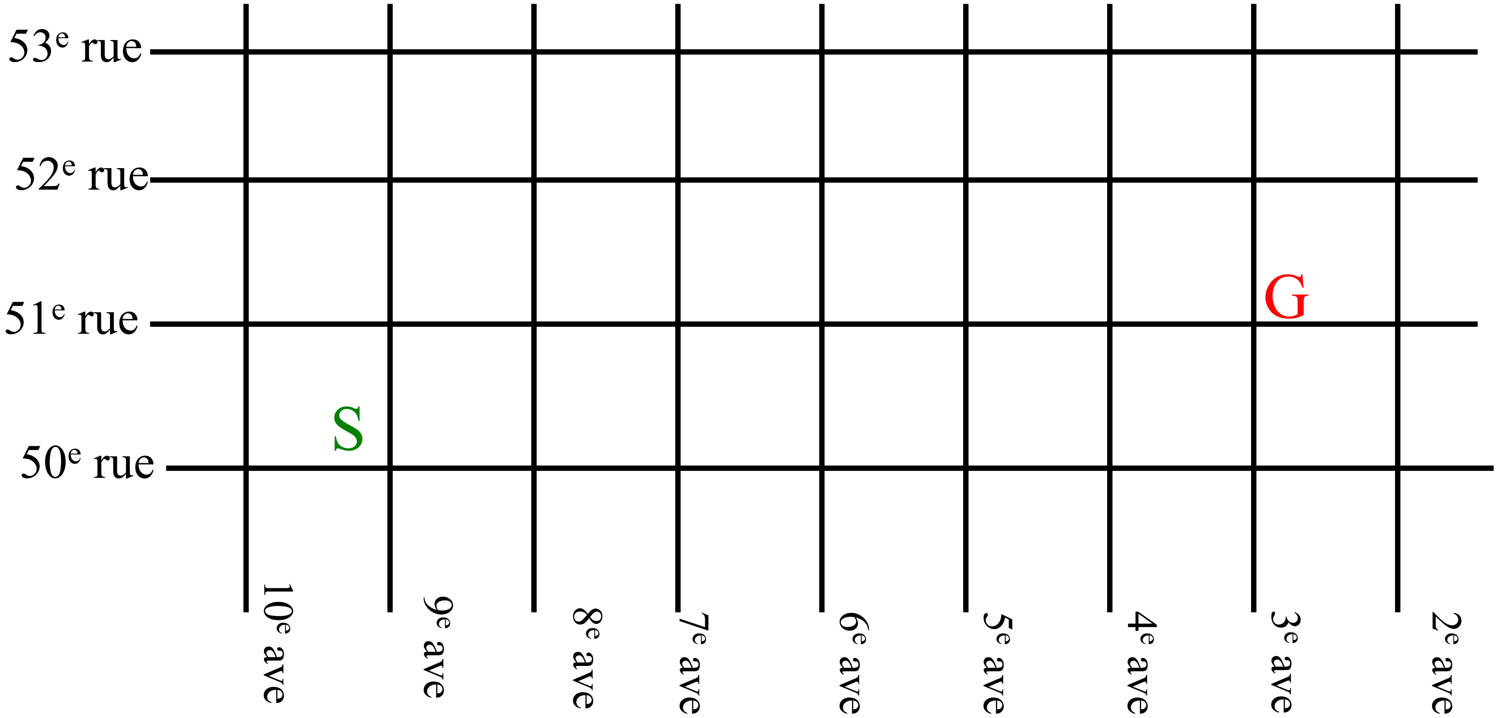
Objectifs

- Résolution de problème par recherche
- Rappel de A* vu en IFT436
- Comprendre A*
- Implémenter A*
- Appliquer A* à un problème donné
- Comprendre la notion d'espace de solutions
- Comprendre la notion d'espace d'états
- Comprendre la notion d'heuristique

Exemple: trouver chemin dans ville

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

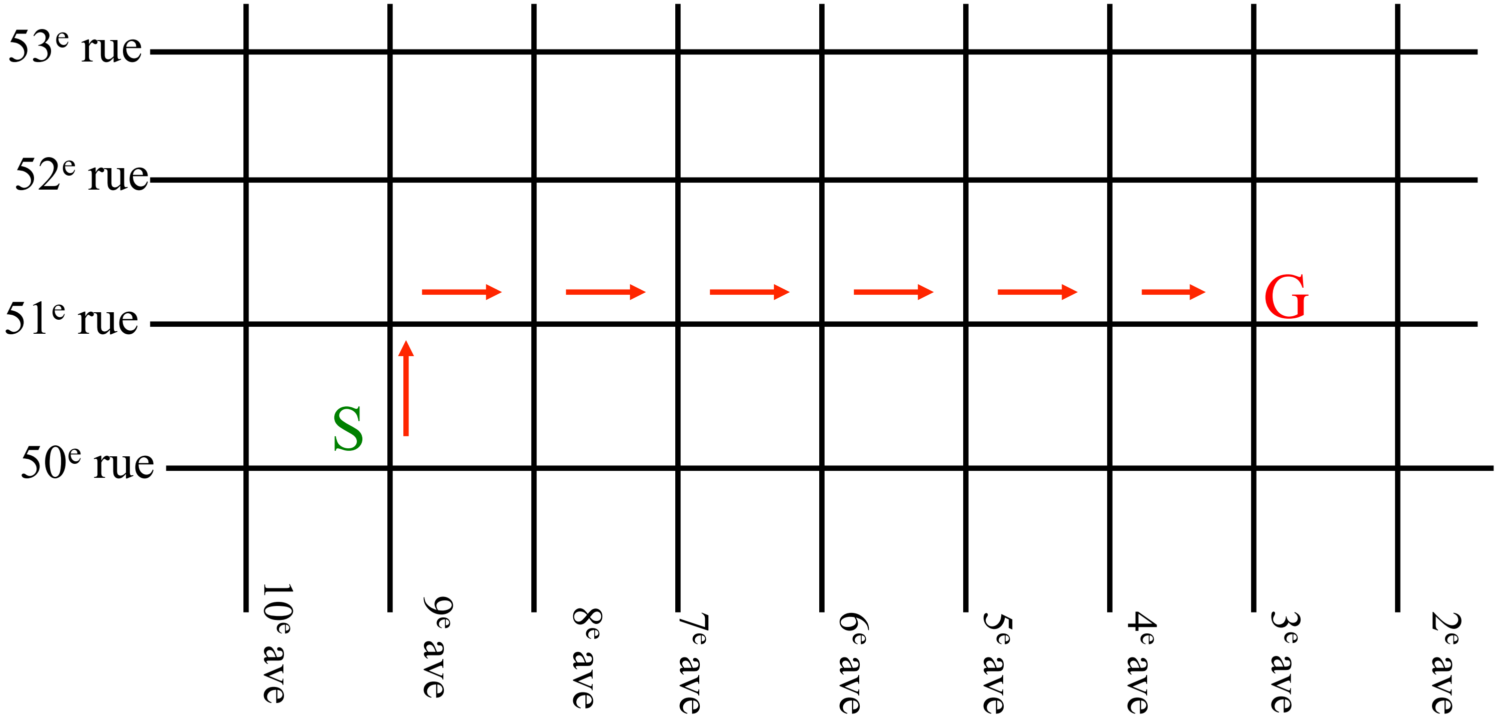
(Illustration par Henry Kautz, U. of Washington)



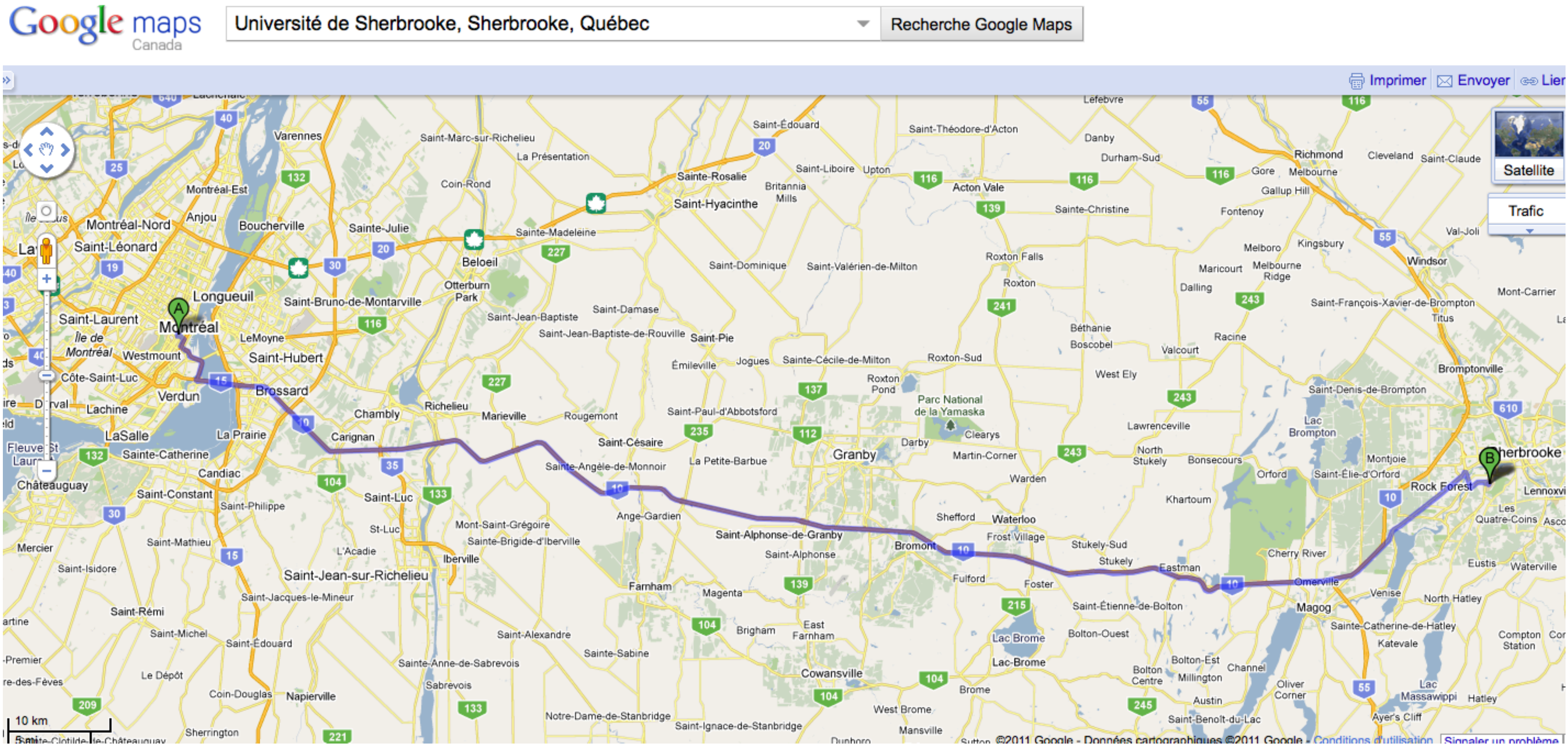
Exemple: trouver chemin dans ville

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

(Illustration par Henry Kautz, U. of Washington)

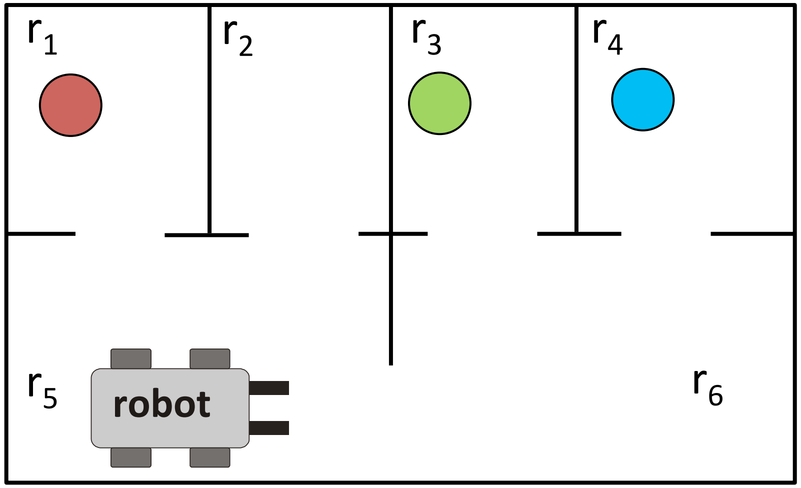


Exemple: Google Maps

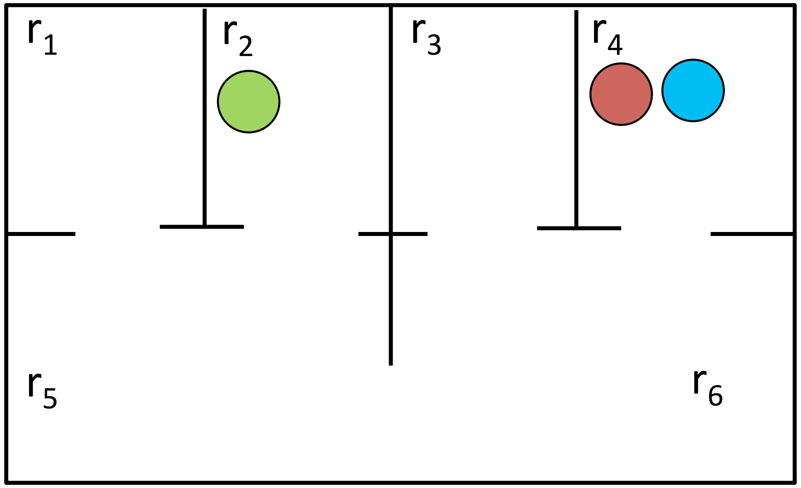


Exemple: livrer des colis

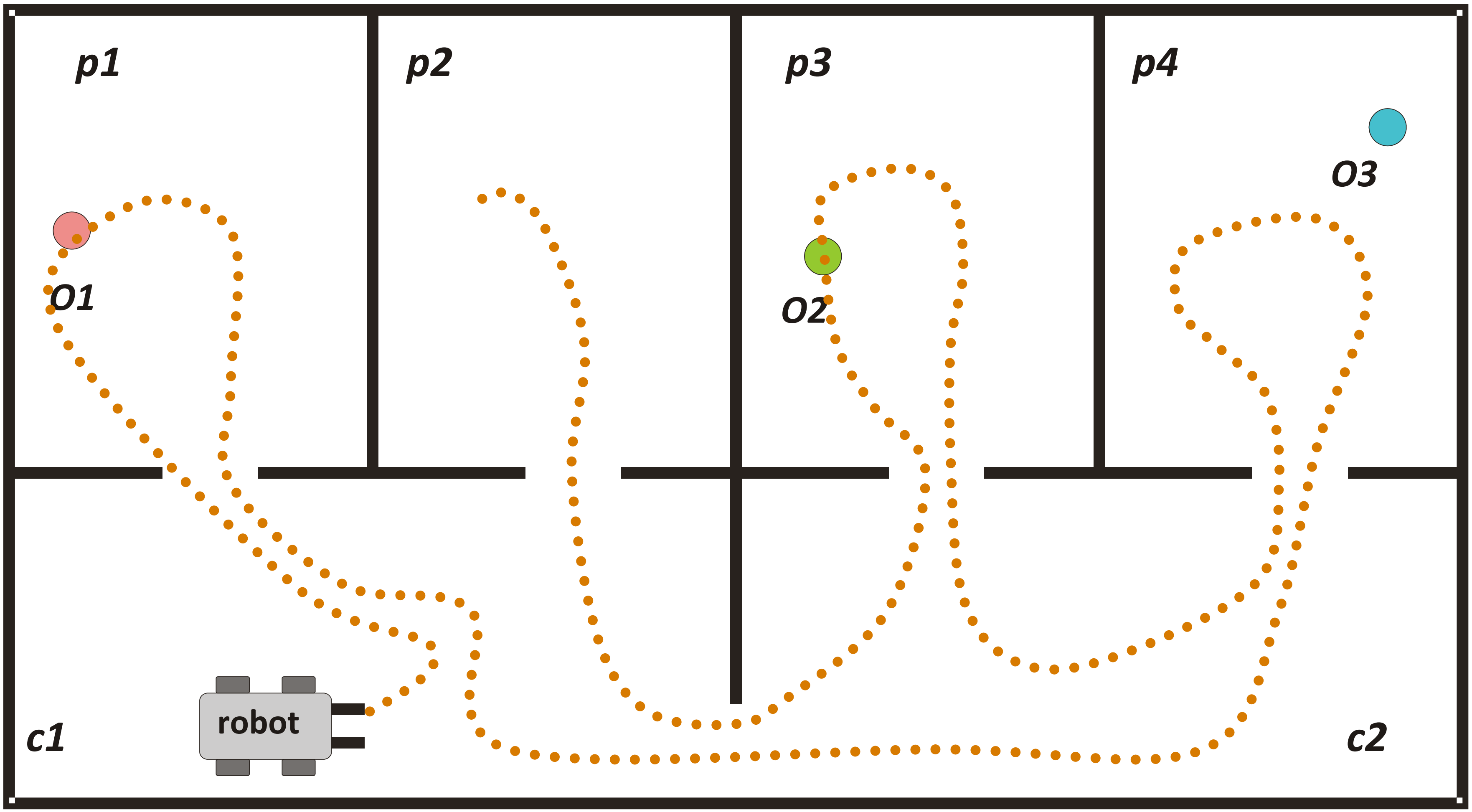
État initial



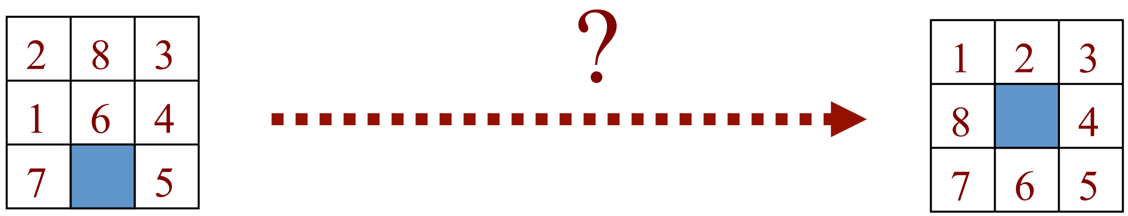
But

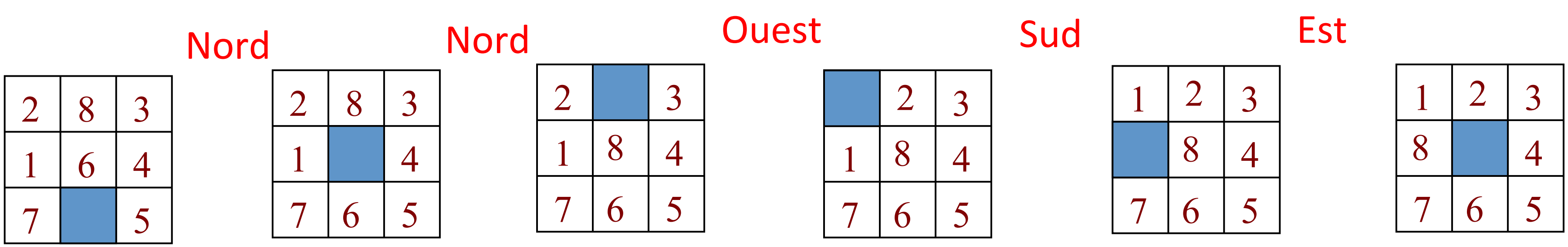


Exemple: livrer des colis



Exemple: N-Puzzle





Résolution de problèmes

- Étapes intuitives par un humain
  1. Modéliser la situation (état) actuelle
  2. Énumérer les options possibles
  3. Évaluer les conséquences des options
  4. Retenir la meilleure option possible satisfaisant le but

- La résolution de beaucoup de problèmes peut être faite par une recherche dans un graphe
- Le graphe peut être un espace de solutions (espaces d'états, espace d'assignations, espace de plans, …)

Résolution de problème par une recherche heuristique dans un graphe

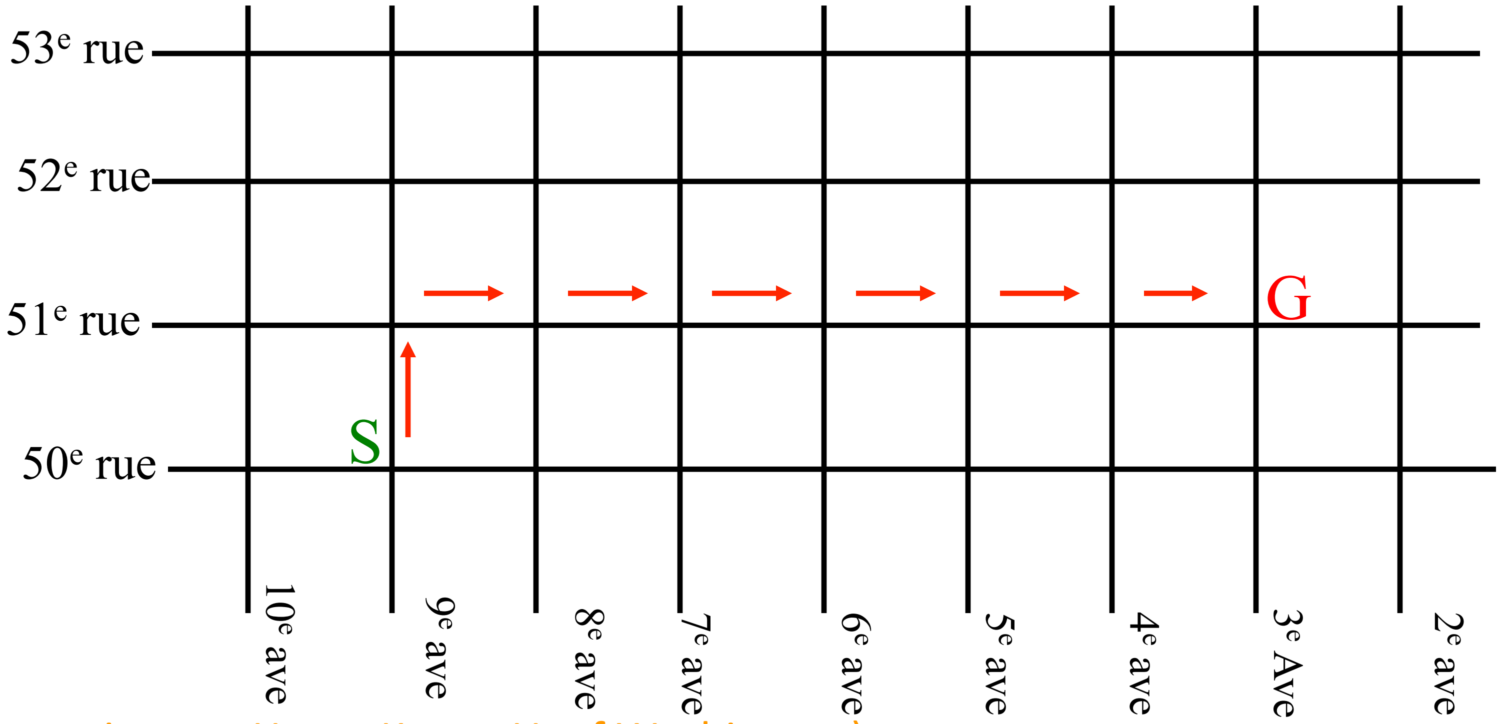
- La recherche heuristique est à la base de beaucoup d'approches en IA
- Approche générale:
  - pour une application donnée, l'**espace des solutions** est représenté à l'aide d'un **graphe**
  - un **problème particulier** pour une application donnée est résolu par **une recherche dans le graphe**
- 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.
  - 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
  - trouver un chemin optimal
  - trouver rapidement un chemin (optimalité pas importante)

Exemple: graphe d'une ville

- Nœuds = intersections
- Arrêtes = segments de rue



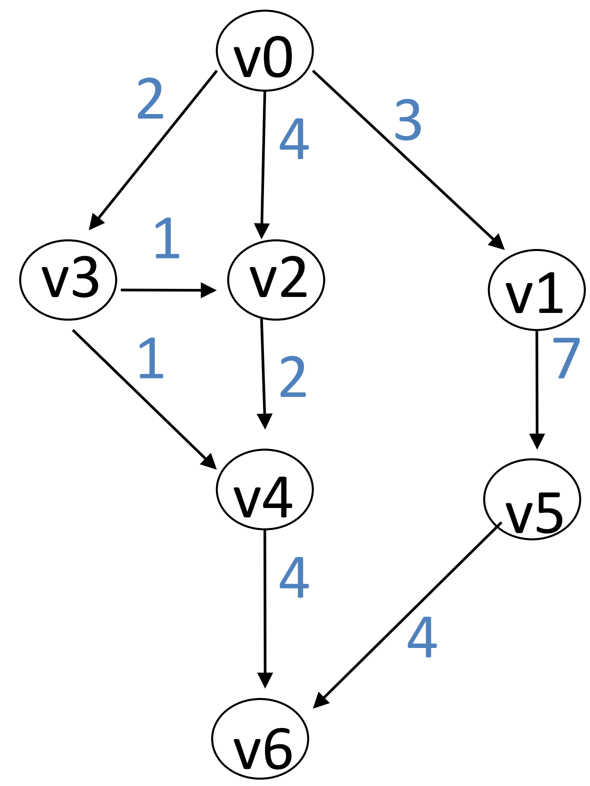
(Illustration par Henry Kautz, U. of Washington)

Exemple: trouver chemin dans une ville

Domaine:

Routes entre les villes

transitions(v0):
((2,v3), (4,v2), (3, v1))



Problème posé (initNode, goal):

v0: ville de départ (état initial)
v6: destination (but)

En d'autres termes:
goal(v): vrai si v=v6

Rappel sur les algorithmes de recherche dans des graphes

- Recherche en profondeur (Depth-First-Search)
- Recherche en largeur (Breadth-First-Search)
- Algorithme de Dijkstra
- Recherche heuristiques:
  - 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
  - par l'ajout des heuristiques
- Une heuristique est une fonction d'estimation du coût entre un nœud d'un graphe et le but (le nœud à atteindre)
- Les heuristiques sont à la base de beaucoup de travaux en IA:
  - recherche de meilleurs 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 dans *open* sont triés selon l'estimé de leur proximité au but
- À chaque nœud *n* est associé une valeur *f(n)* mesurant la qualité de la meilleure solution passant par ce nœud
- Pour chaque nœud *n*, *f(n)* est un nombre réel positif ou nul, estimant le coût pour un 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*

Définition de *f*

- La **fonction d'évaluation** *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 la racine et un nœud *déjà exploré*
- Il est pratique de séparer *f(n)* en deux parties:
  - *g(n)*: coût réel du chemin optimal partant de la racine à *n* dans la partie déjà explorée
  - *h(n):* coût estimé du reste du chemin partant de *n* jusqu'au but

    h(n) est une **fonction heuristique**

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

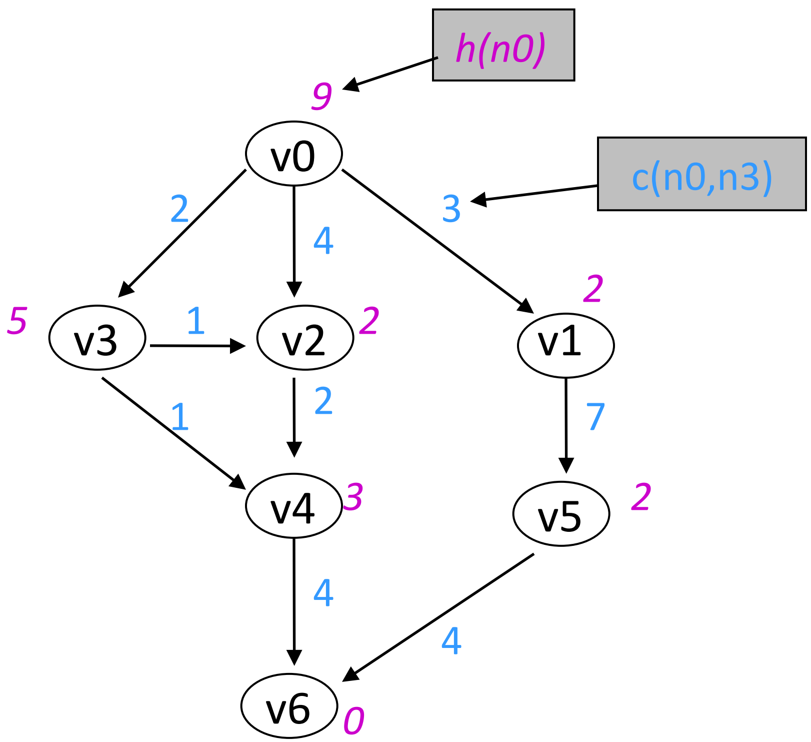
Algorithme rechercheDansGraphe(*noeudInitial*)

1. Déclarer deux nœuds: *n1*, *n2*
2. Déclarer deux listes: *open*, *closed* // *toutes les deux sont vides au départ*
3. Insèrer *noeudInitial* dans *open*
4. while (1) // *la condition de sortie (exit) est déterminée dans la boucle*
    5. si *open* est vide, sortir de la boucle avec échec
    6. *n1* = noeud au début de open*;*
    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 successeur n2 de n1
        10. Initialiser la valeur *g* de *n2 à: g(n1) + le coût de la transition (n1,n2)*
        11. mettre le parent de *n2* à *n1*
        12. Si *closed* ou *open* contient un nœud n3 égal à n2 avec *f(n2) ≤f(n3)*, enlèver *n3* de *closed* ou *open* et insérer *n2* dans *open (ordre croissant selon f(n))*
        13. Sinon (c-à-d., n2 n'est ni dans *open* ni dans *closed)*
            14. insérer *n2* dans open *en triant les nœuds en ordre croissant selon f(n)*

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):

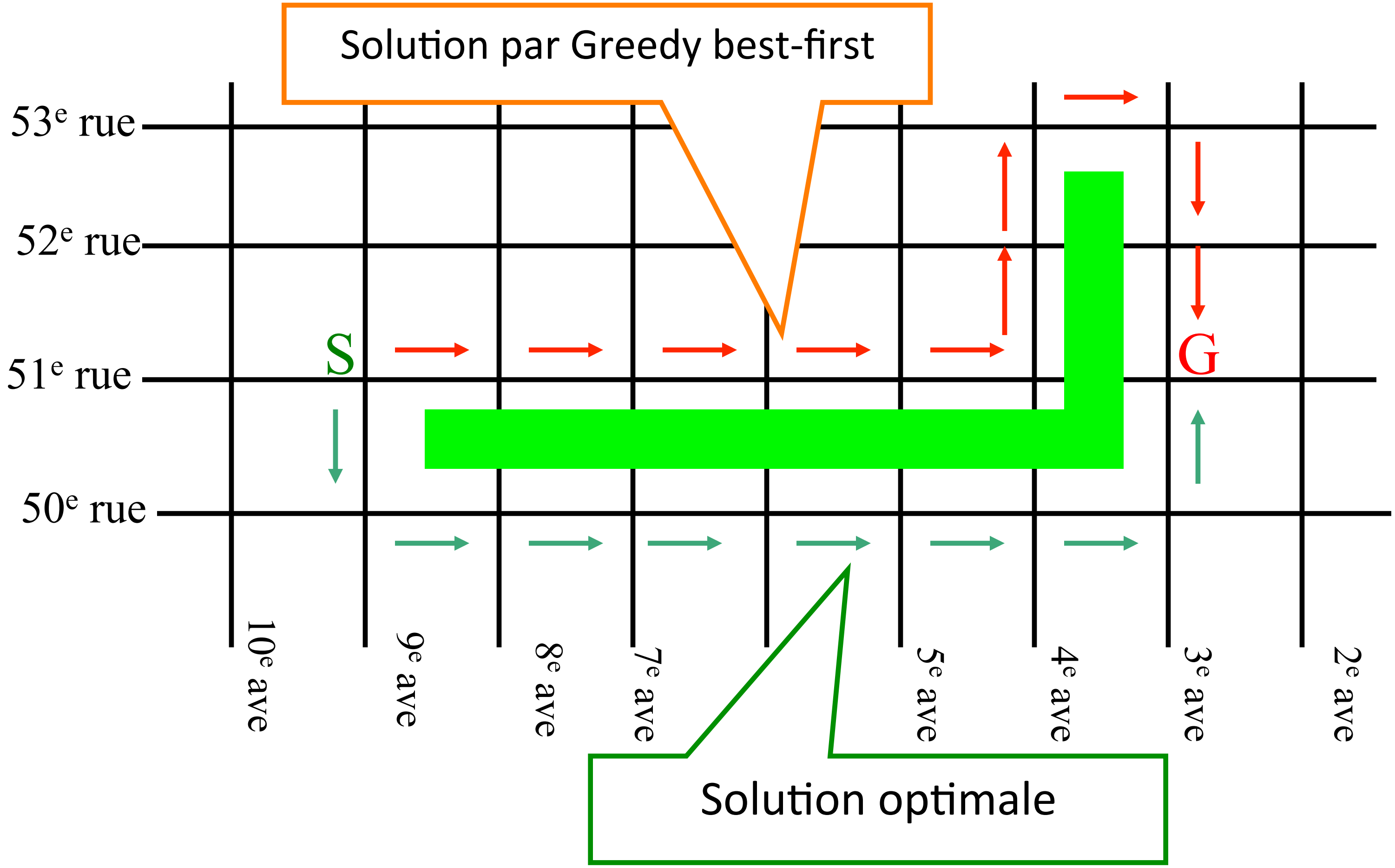
1. (v0, 9, void)
2. (v1,5,v0) (v2,6,v0), (v3,7,v0)
3. (v2,6,v0) (v3,7,v0), (v5,12,v1)
4. (v3,7,v0),(v4,9,v2),(v5,12,v1)
5. (v2,5,v3),(v4,6,v3),(v5,12,v1)
6. (v4,6,v3),(v5,12,v1)
7. (v6,7,v4), (v5,12,v1)
8. Solution: **v0,v3,v4,v6**

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

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

Non-optimalité de Greedy best-First Search

(Illustration par Henry Kautz, U. of Washington)



Propriétés de A*

- Si le graphe est fini, A* termine toujours
- Si une solution existe A* la trouve toujours (elle pourrait ne pas être optimale)
- 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 une solution optimale**
- *Parfois, on entend par A* la version de l'algorithme avec la condition additionnelle que h soit admissible*
- A* est une sorte de 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 retournant toujours 0 quelque soit le nœud, 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 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 $h(n1) \leq c(n1,n2) + h(n2)$, où $c(n1,n2)$ est le coût de l'arc $(n1,n2)$, on dit 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* $h1$ et $h2$, tel que $h1(n) < h2(n)$, alors $h2(n)$ conduit plus vite au but: avec $h2$, A* explore moins ou autant de nœuds avant d'arriver au but qu'avec $h1$

- Si h n'est pas admissible, soit x la borne supérieure 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 de plus que le coût optimal, c-à-d., A* ne se trompe pas plus que x 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 $0 \leq h_1(s) < h_2(s) \leq h^*(s)$, 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 de nœuds
  - Vrai.
- Si h(s)=h*(s), pour tout état *s,* 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 0, 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

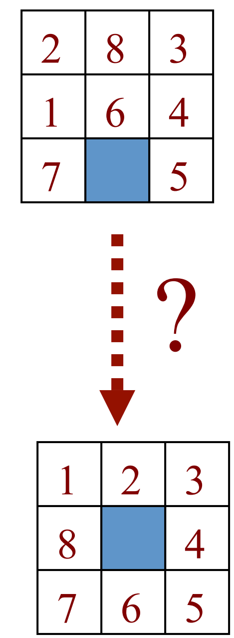
Variations de A*

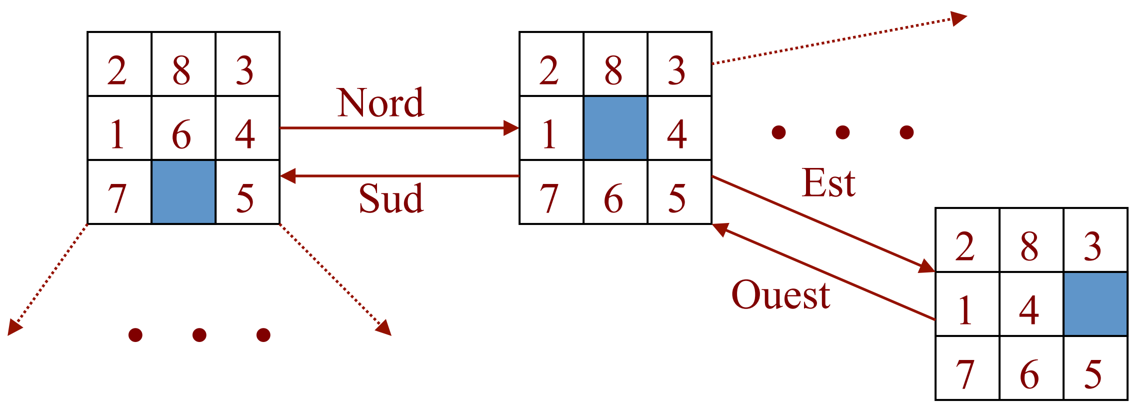
- Iterative deepening
  - on met une limite sur la profondeur
  - on lance A* jusqu'à la limite de profondeur spécifiée.
  - si pas de solution on augmente la profondeur et on recommence A*
  - ainsi de suite jusqu'à trouver une solution.

- 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.

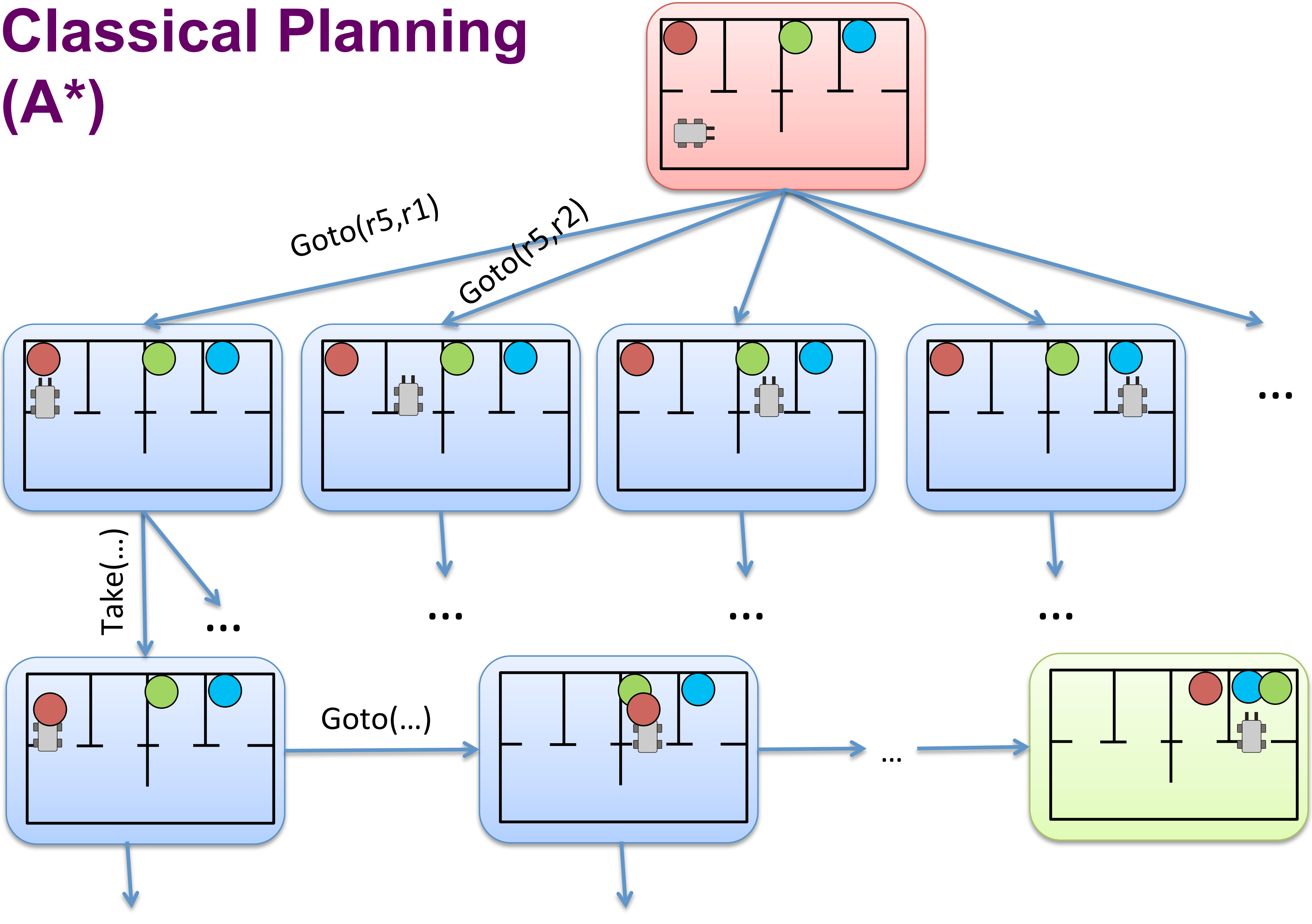
Exemple académique

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

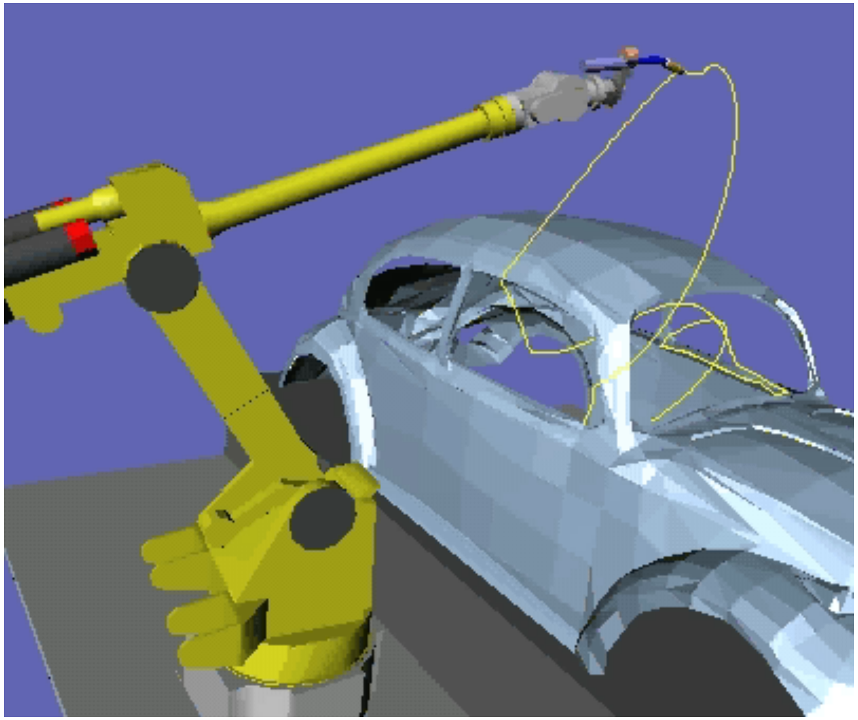


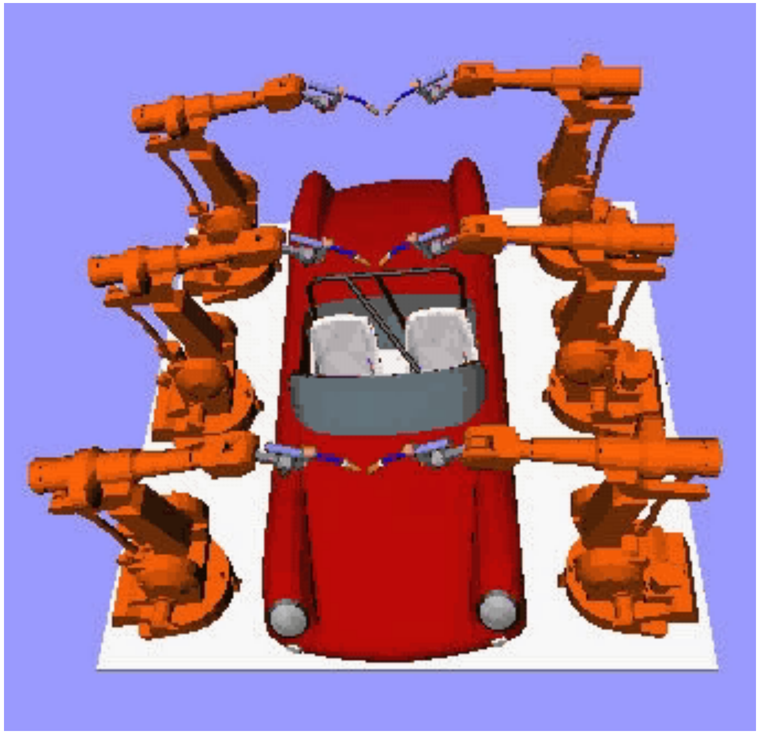


Classical Planning (A*)



Application: industrie automobile





Démos du Motion Planning Kit (Jean-Claude Latombe)

Application: jeux vidéos et cinéma

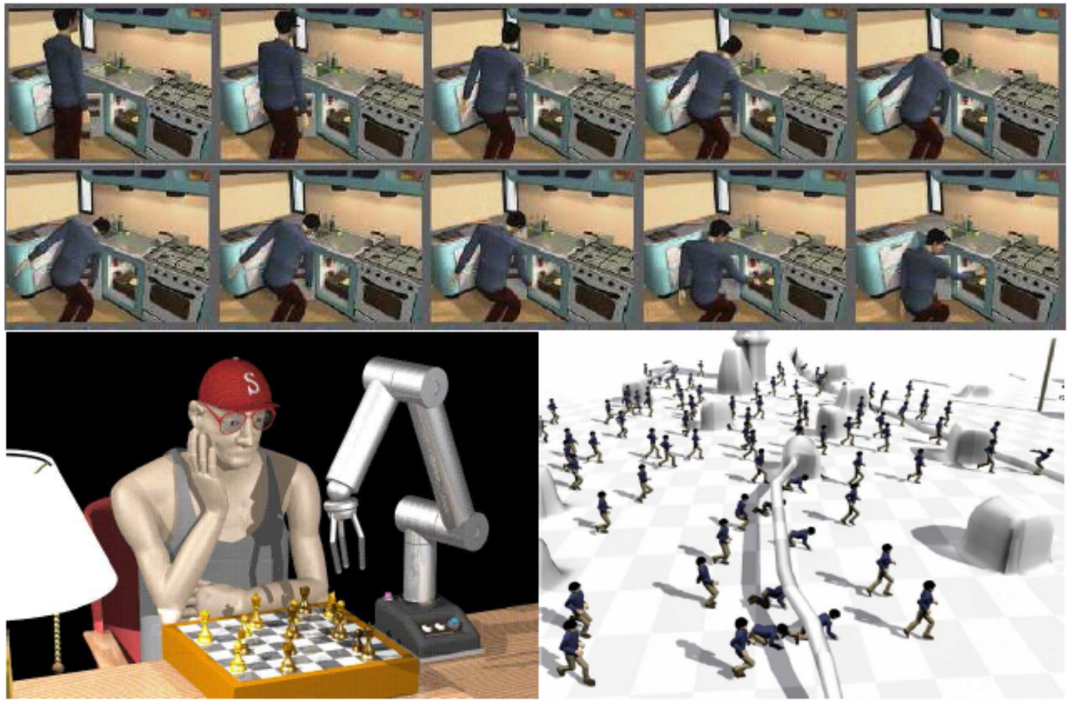


Figure 1.8: Across the top, a motion computed by a planning algorithm, for a digital actor to reach into a refrigerator [499]. In the lower left, a digital actor plays chess with a virtual robot [545]. In the lower right, a planning algorithm computes the motions of 100 digital actors moving across terrain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

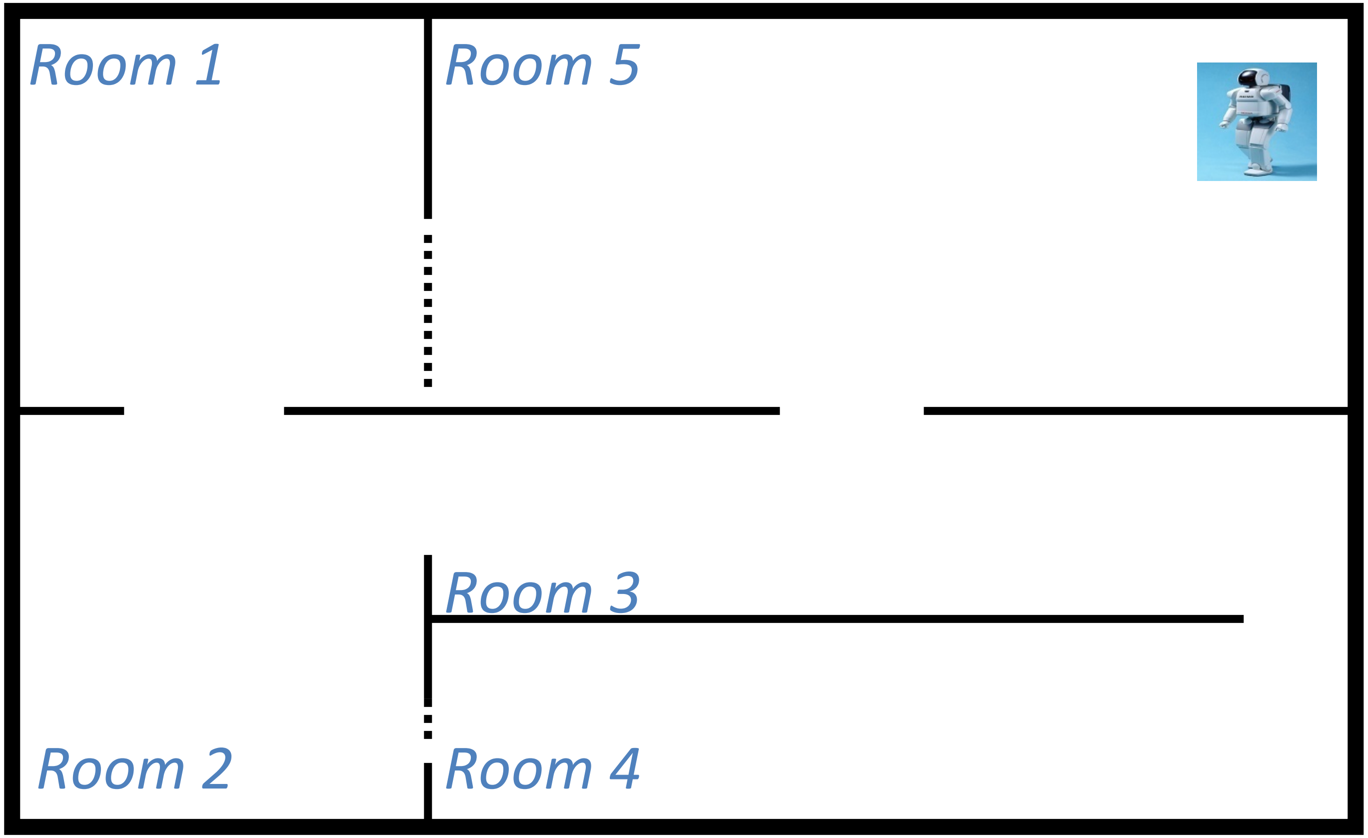
(décomposition, échantillonage)

↓

Recherche heuristique dans un graphe

(A* ou similaire)

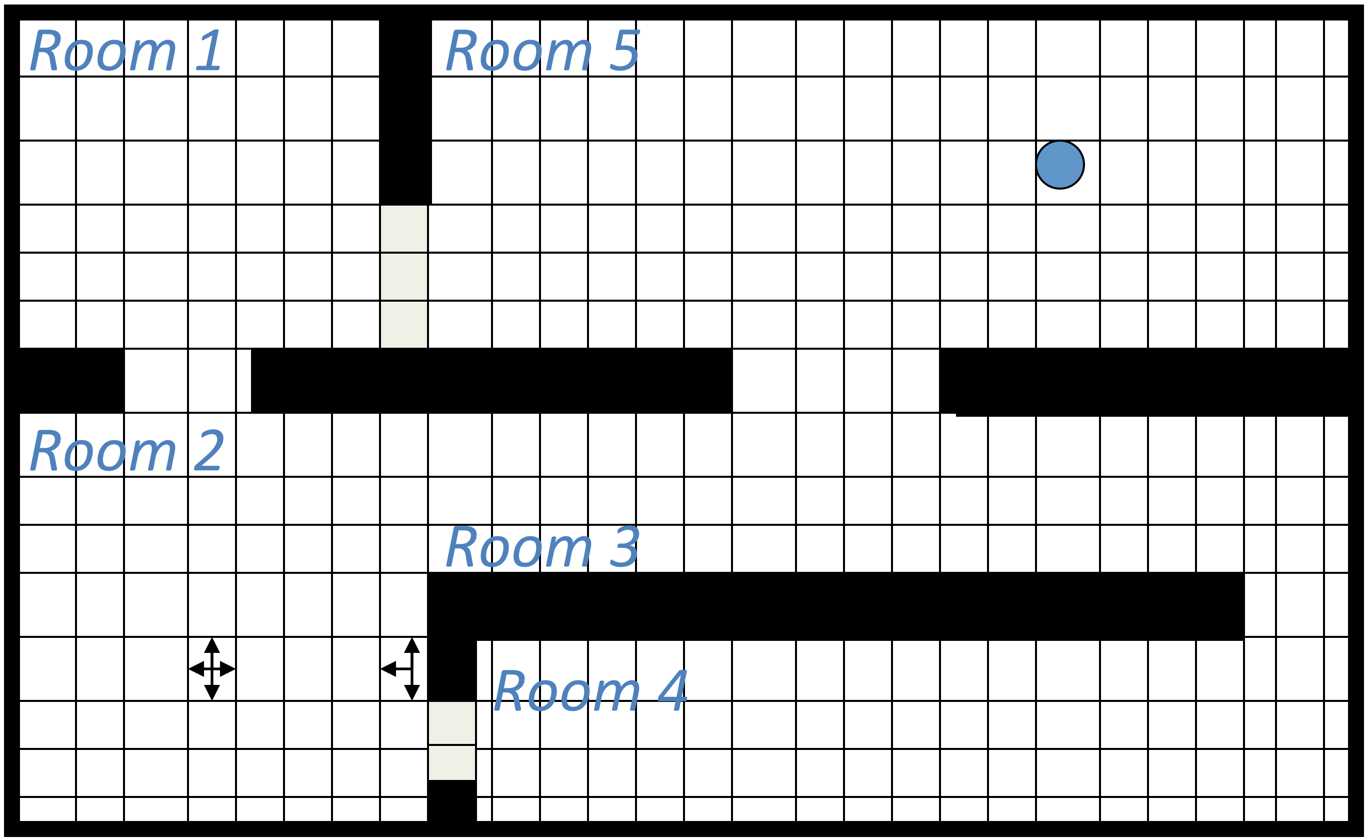
Approche combinatoire par décomposition en cellules



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