

INF8215 - Intelligence artificielle

Méthodes et algorithmes

Module 1: Stratégies de recherche



POLYTECHNIQUE
MONTRÉAL

Quentin Cappart

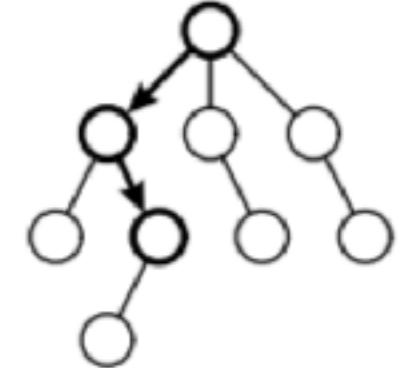
Contenu du cours

Raisonnement par recherche (essais-erreurs avec de l'intuition)

Module 1: Stratégies de recherche

Module 2: Recherche en présence d'adversaires

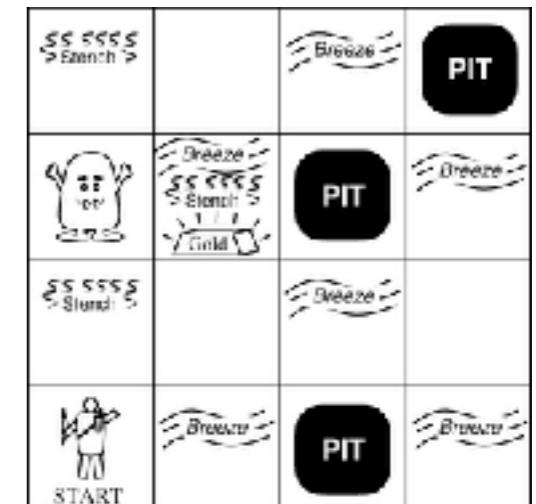
Module 3: Recherche locale



Raisonnement logique

Module 4: Programmation par contraintes

Module 5: Agents logiques



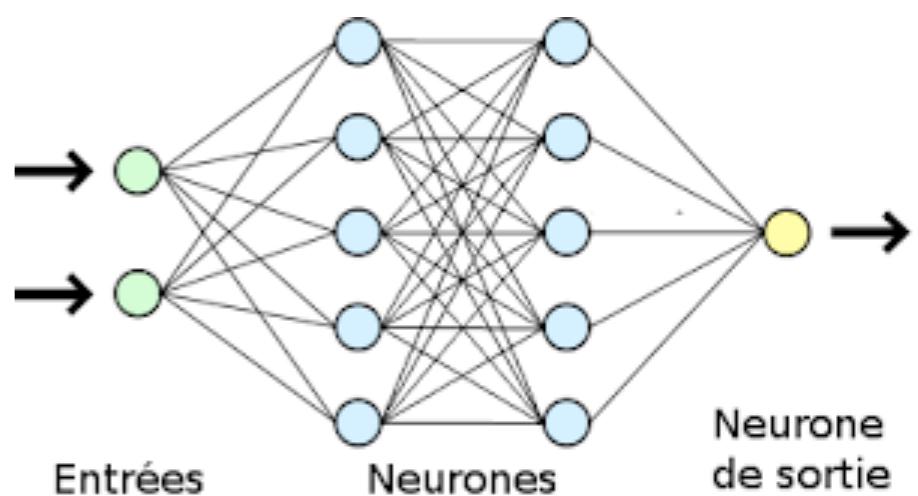
Raisonnement par apprentissage

Module 6: Apprentissage supervisé

Module 7: Réseaux de neurones et apprentissage profond

Module 8: Apprentissage non-supervisé

Module 9: Apprentissage par renforcement



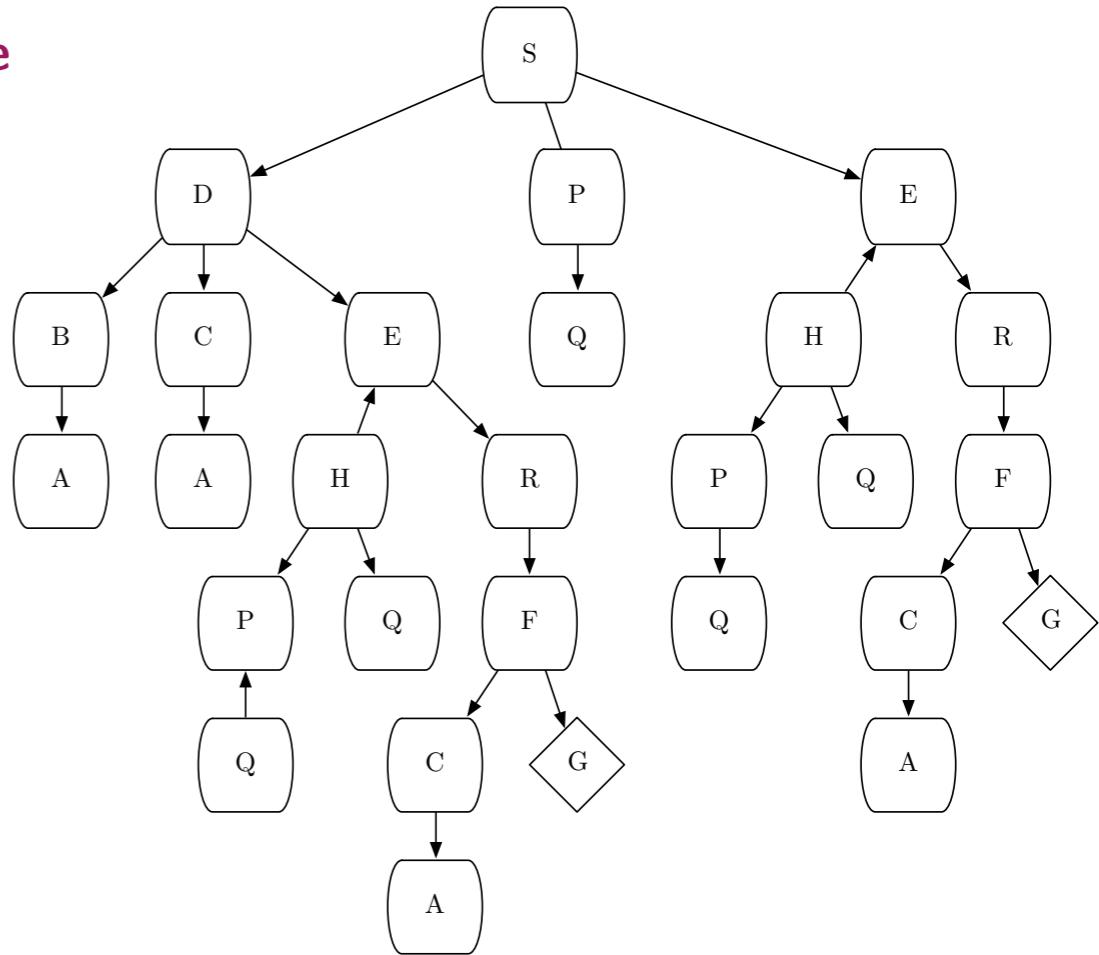
Considérations pratiques et sociétales

Module 10: Utilisation en industrie, éthique, et philosophie

Table des matières

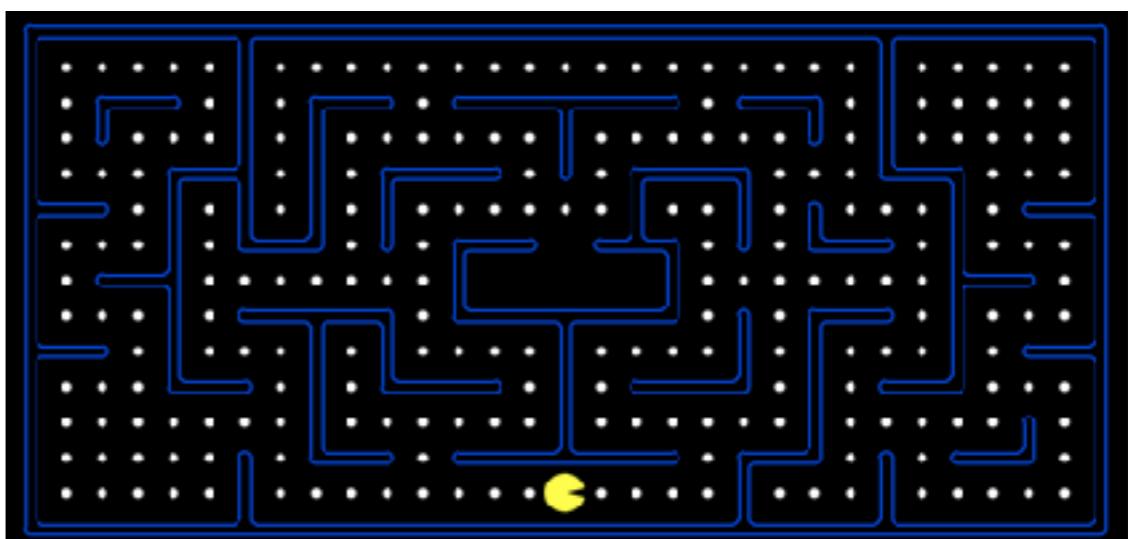
Stratégies de recherche

1. Définition et modélisation d'un problème de recherche
2. Agents réflexes
3. Agents axés sur la recherche
4. Recherche en arbre (*tree search*)
5. Recherche sans information: DFS, BFS, UCS, IDS
6. Recherche avec information: *greedy search*, A*
7. Conception d'heuristiques
8. Recherche en graphe (*graph search*)

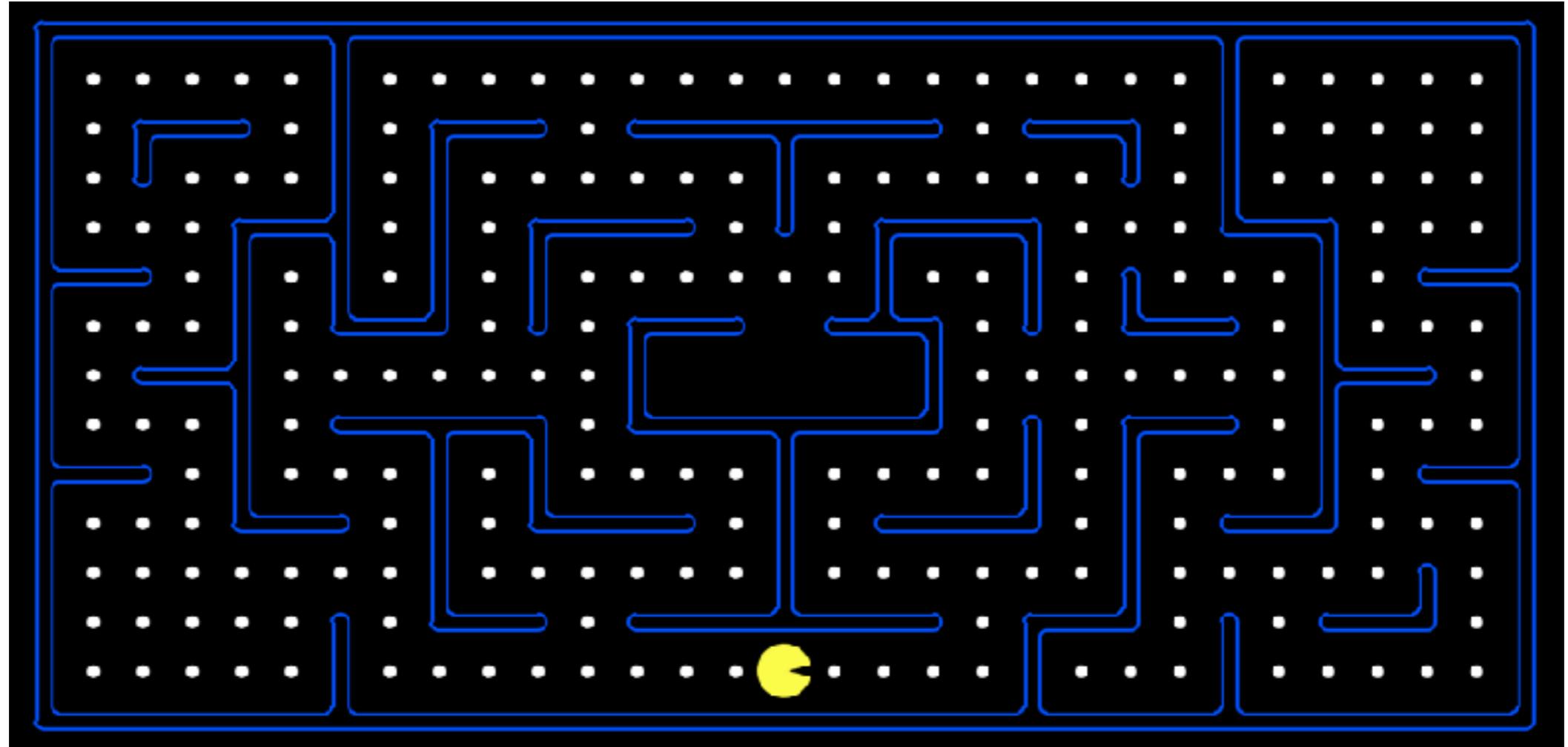


Problèmes abordés

1. *Pacman*
2. *8-puzzle*
3. Planification de routes



Cas d'étude: *Pacman*



Pacman, jeu sorti en 1980

Objectif

Concevoir un agent mangeant tous les points en exécutant le moins de déplacements possibles

Pas de fantômes, ni de bonus

Actions possibles pour Pacman: déplacement à gauche, droite, haut, ou bas

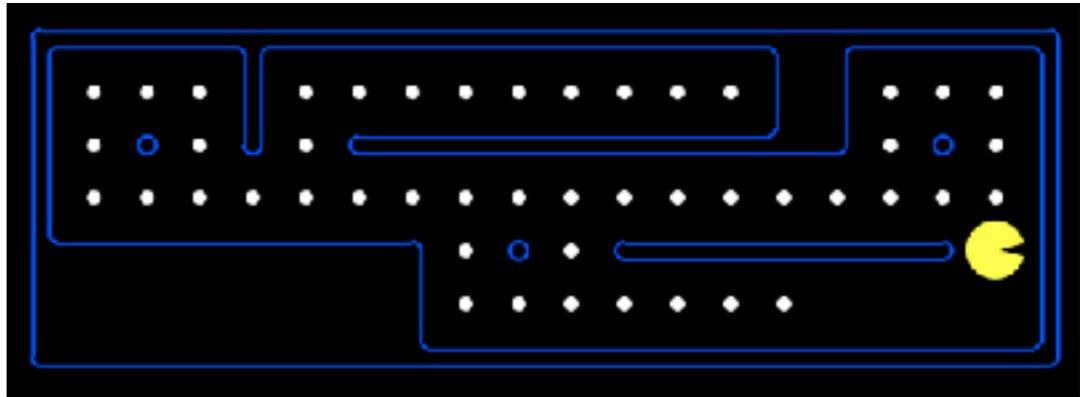


Comment implémenter un agent efficace pour cette tâche ?

Agent réflexe

Première idée

- (1) Observer l'environnement actuel
- (2) Choisir une action en fonction de l'observation
- (3) Effectuer l'action



Agent réflexe (*reflex agent*)

Agent qui agit en fonction de comment le monde lui apparaît (sa perception actuelle)

Intègre éventuellement un mécanisme de mémoire

Retourne une action sur base de l'observation et de la mémoire

Avantage: très rapide à exécuter

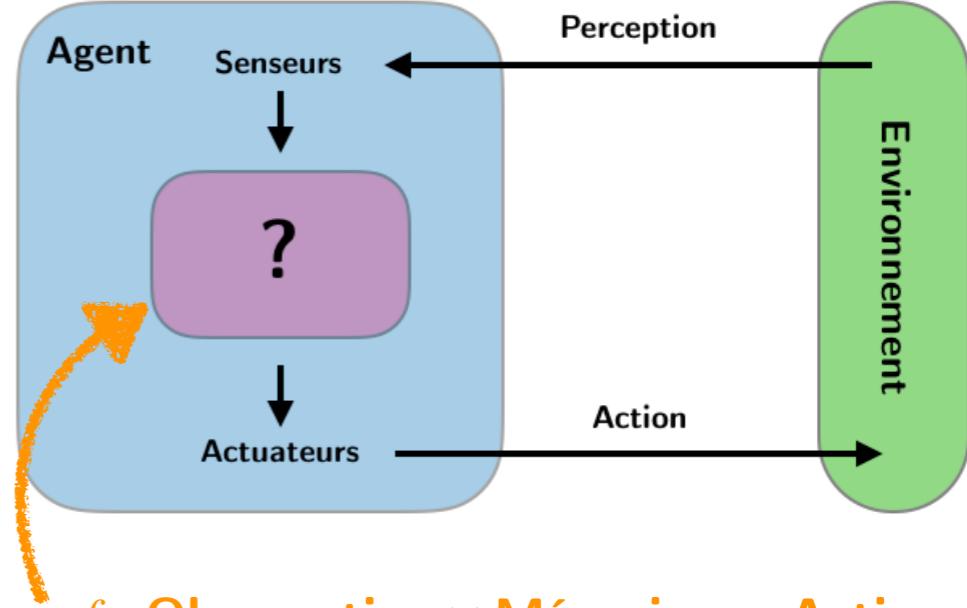
Inconvénient: ne considère pas les conséquences des actions faites

Exemple (Pacman)

Observation (ou perception): la grille du jeu

Mémoire: aucune

Action: prendre la direction du point le plus proche



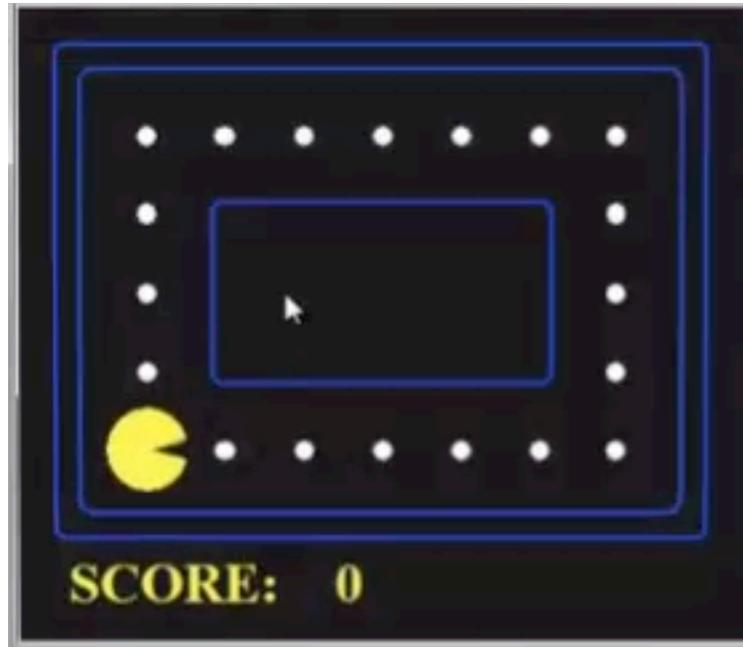
$$f : \text{Observation} \times \text{Mémoire} \rightarrow \text{Action}$$



Est-ce que cet agent est rationnel ?

Agent réflexe: exemples

Situation 1



Comportement rationnel

Situation 2

Comportement non-rationnel



Est-ce qu'on agit parfois comme un agent réflexe ?



(1) L'instinct: sans réfléchir aux conséquences de nos actions

S'éloigner d'une source de douleur (main sur une surface brûlante)

Cligner des yeux lorsqu'une poussière arrive à notre paupière

(2) Par urgence: lorsqu'on a pas le temps de prendre une décision !

Freiner en bloc lorsqu'un piéton apparaît brusquement sur la route

Agent axé sur la recherche (*planning agent*)



Agent axé sur la recherche (*planning agent*)

Agent qui va dérouler une série de scénarios, découlant des actions permises, afin de trouver une séquence d'actions amenant à la réalisation de l'objectif

Cette définition suppose l'**existence d'un objectif**, pouvant être atteint par une séquence d'actions

Différents scénarios sont simulés, jusqu'à ce qu'une séquence favorable soit trouvée

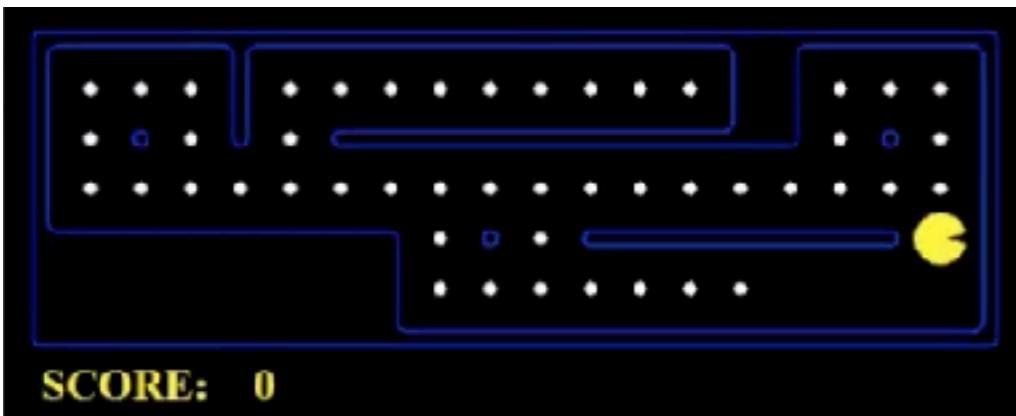
Le défi principal est de dérouler efficacement les différentes séquences d'actions possibles

Avantage: exploite de l'information à long terme, menant à une meilleure stratégie que l'agent réflexe

Inconvénient: la simulation des scénarios demandent plus de ressources

Inconvénient: demande de connaître la dynamique du monde (l'impact de chaque action)

Exemple



Objectif: manger tous les points

Environ 3000 actions ont été évaluées

Environ 30 secondes d'exécution



Comment formaliser ce fonctionnement ?

Formalisation d'un problème de recherche



Problème de recherche

Problème consistant à trouver la meilleure séquence d'action pour atteindre un état final à partir d'un état initial. Il est formellement défini par:

S : un ensemble d'états (contenant un état initial et un/des états finaux)

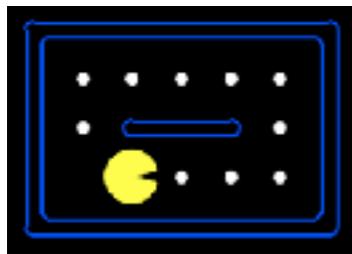
A : un ensemble d'actions

$T : (S \times A \rightarrow S)$: une fonction de transition décrivant le nouvel état émanant d'une action

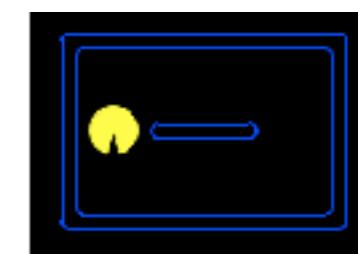
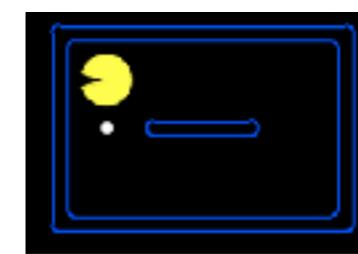
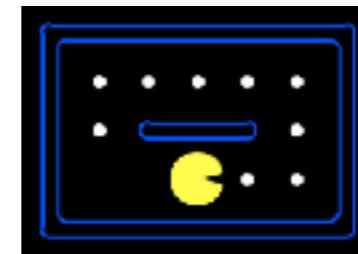
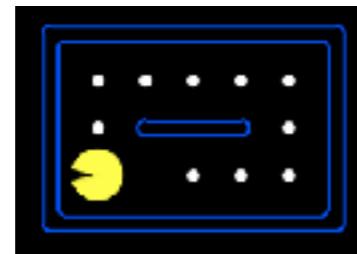
$C : (S \times A \rightarrow \mathbb{R})$: une fonction décrivant le coût d'effectuer une action à partir d'un état

Ensemble d'états

Toutes les configurations possibles que peut prendre l'environnement



Initial



Final

...

Ensemble d'actions

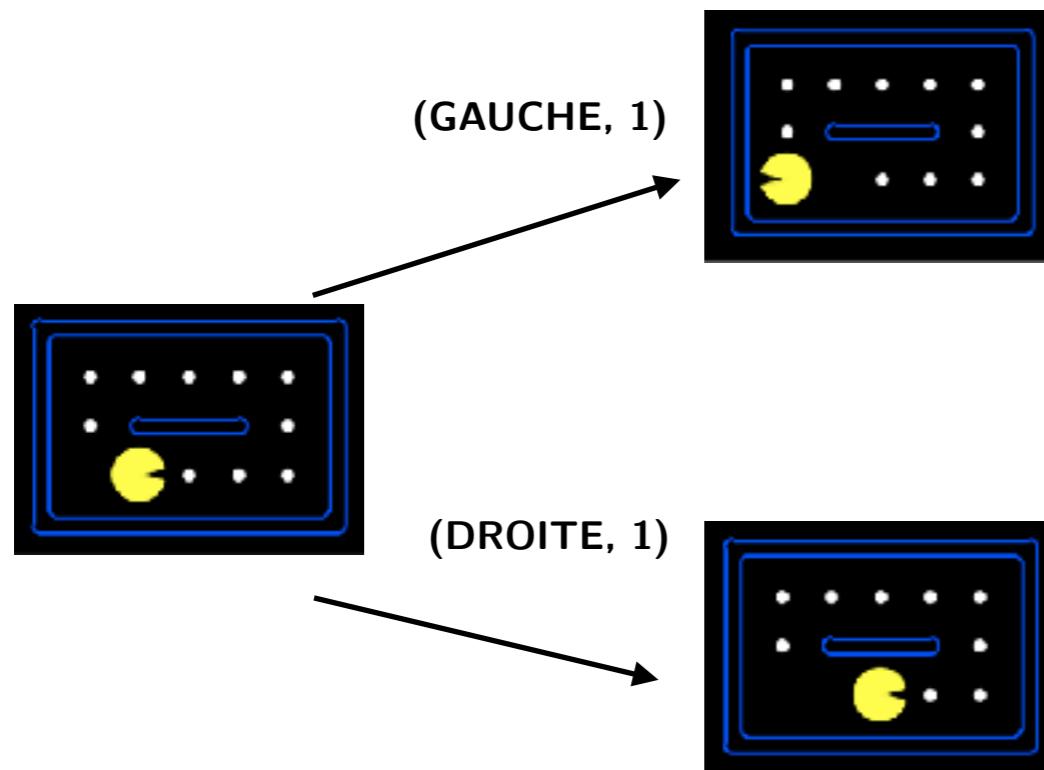
Déplacement à gauche, à droite, en haut, ou en bas

Formalisation d'un problème de recherche

Fonction de transition et de coût

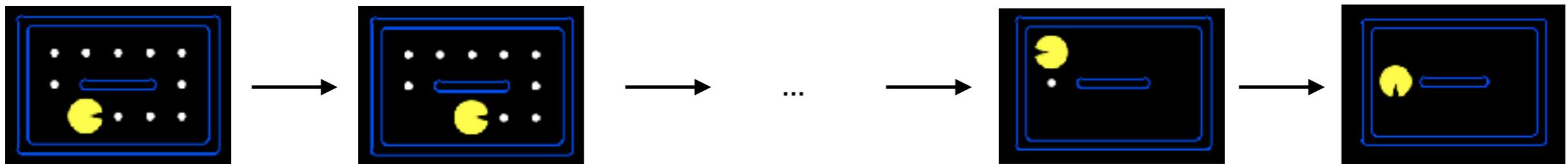
Indique le nouvel état et le coût si une action est faite à partir d'un état spécifique

Cas simple de Pacman: Un coût unitaire est associé à chaque déplacement

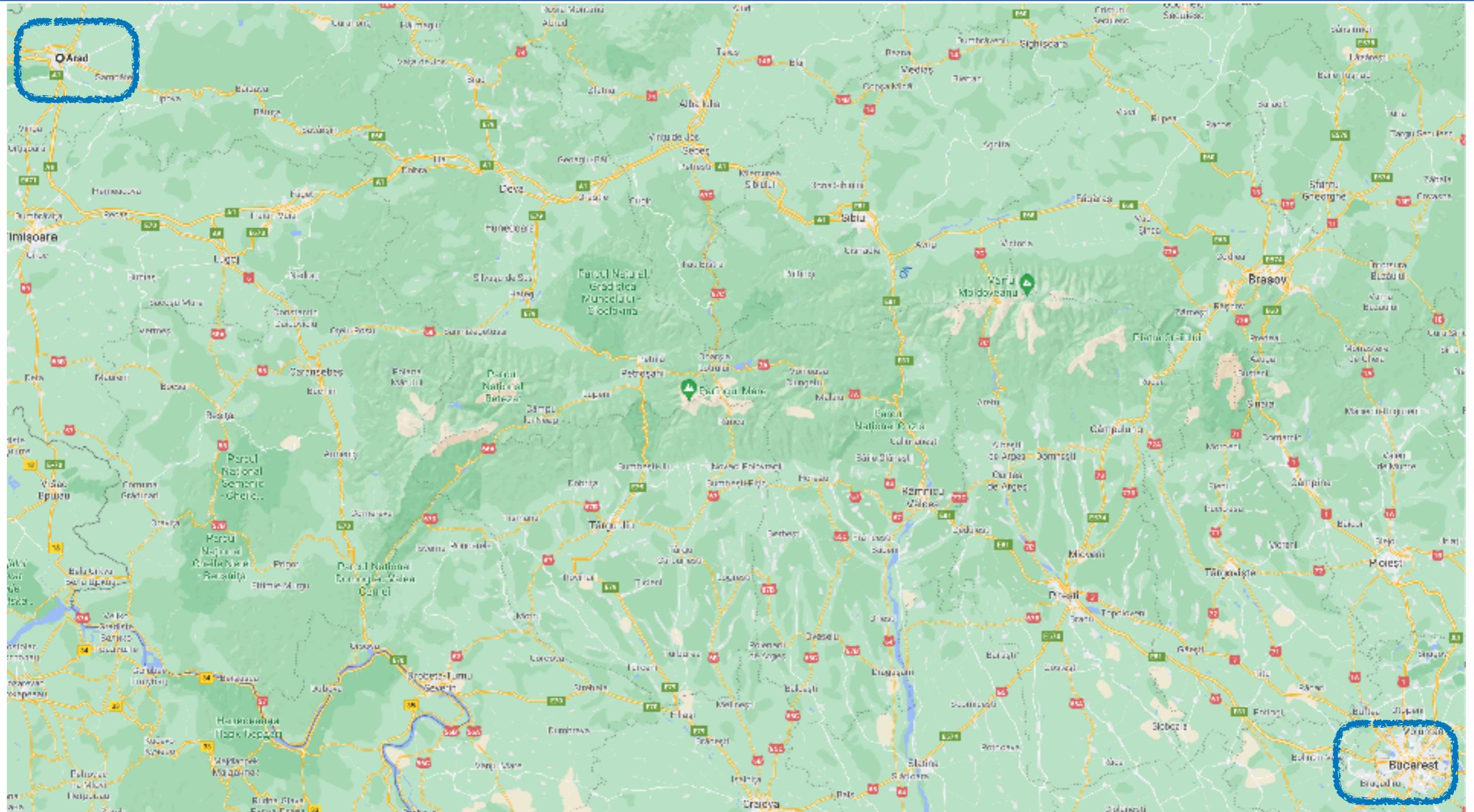


Solution

Séquence d'actions amenant un état initial à un état final



Problème réel de planification de routes



Objectif : trouver le chemin le plus court entre Arad et Bucarest

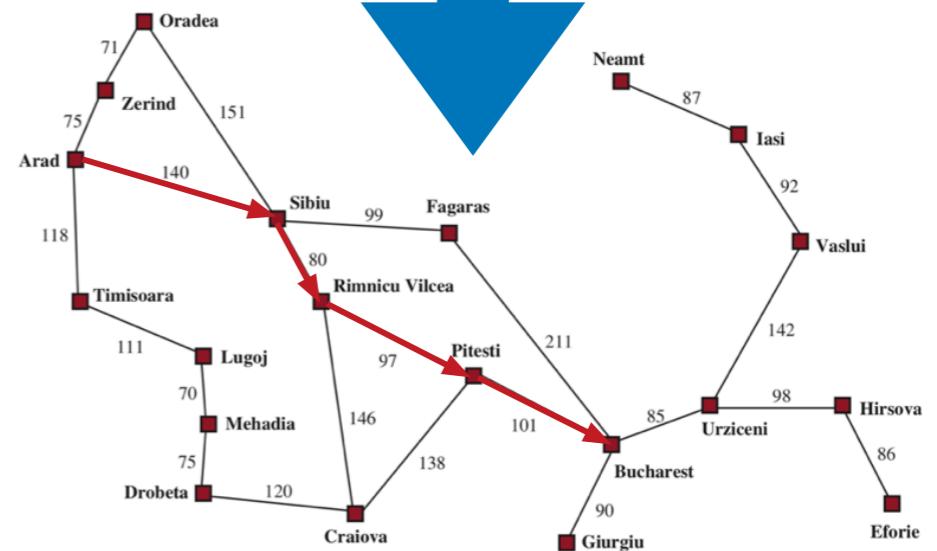
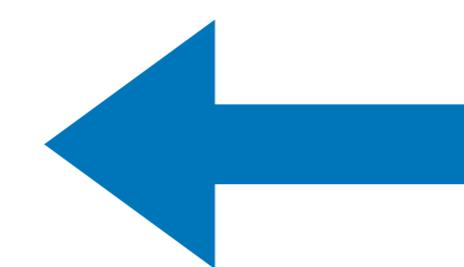
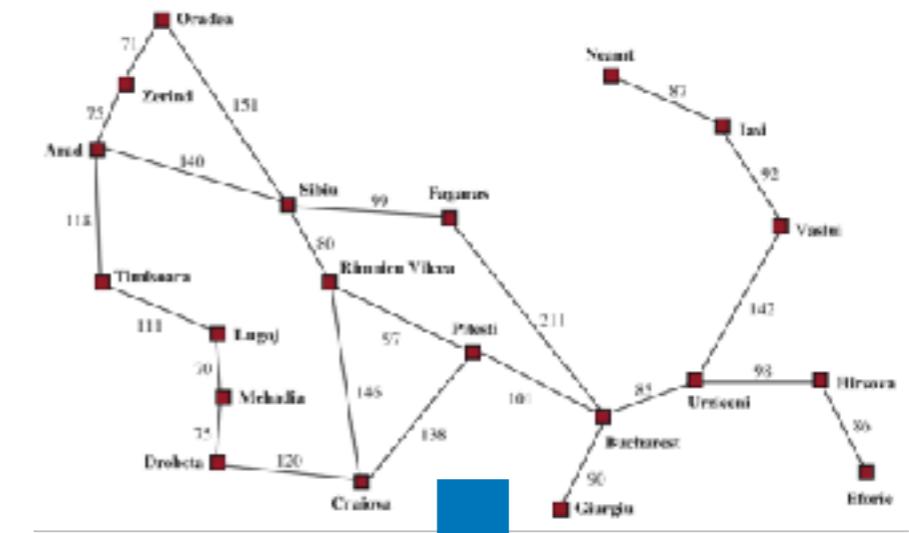
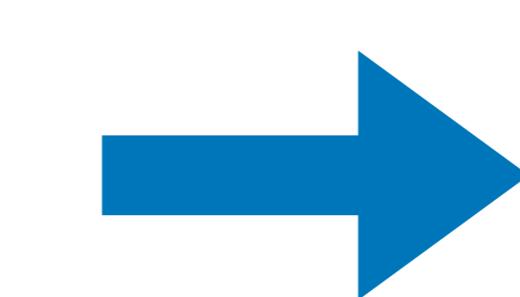
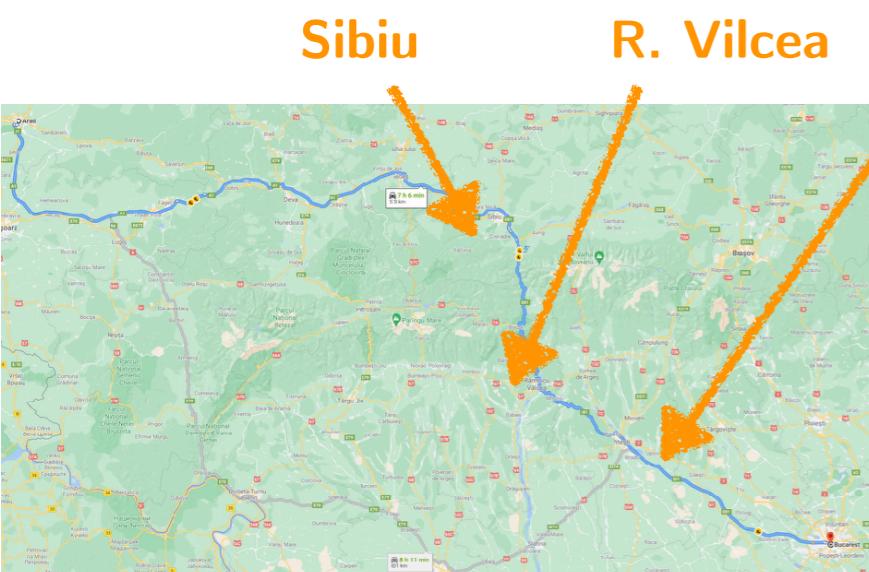
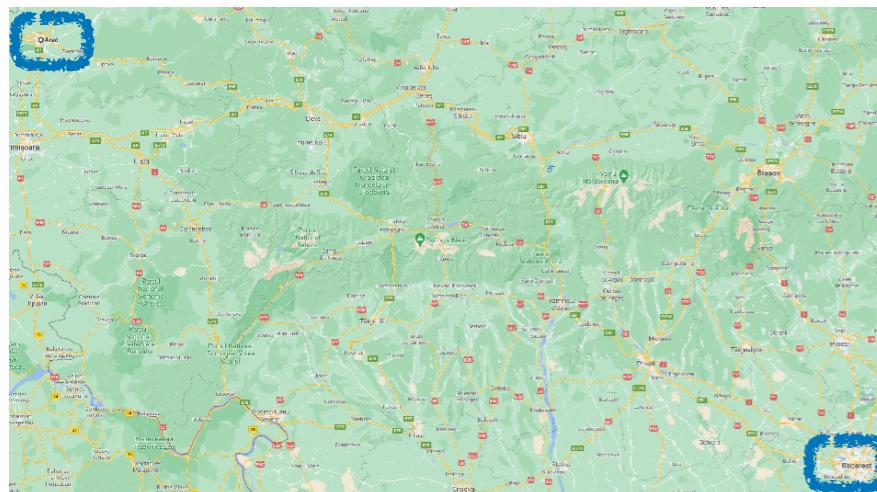


Comment modéliser cette situation en un problème de recherche ?

Modélisation d'un problème

Modélisation

Modéliser un problème de recherche revient à l'encoder sur base de notre formalisation précédente



Le problème modélisé est ensuite résolu, et la solution est ré-exprimée comme solution du problème réel

Abstraction trop forte: le modèle ne reflète pas assez fidèlement la réalité

Abstraction trop faible: le modèle peut être trop difficile à résoudre

Modélisation du monde réel

Niveau d'abstraction

Considérer un sous-ensemble de villes et de liaisons

Objectif

Atteindre Bucharest à partir d'Arad

Ensemble d'états

Toutes les positions possibles

Etat initial: Arad

Etat final: Bucharest

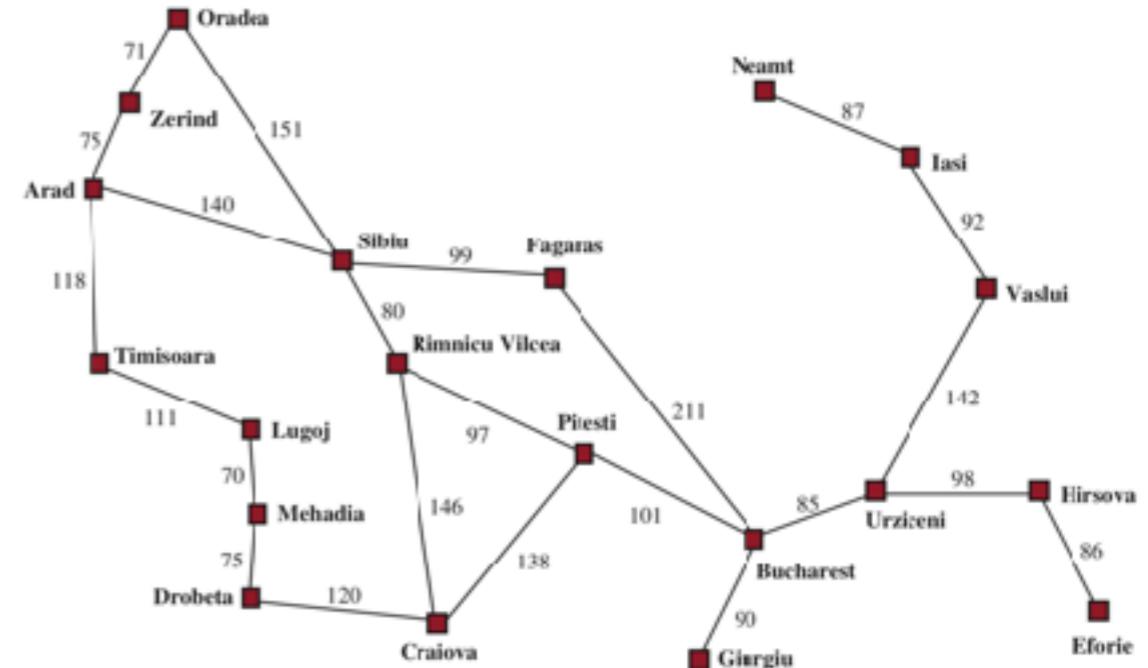
Action

Suivre une arête

Fonction de transition et de coût

Effectuer le déplacement sur l'arête

Pénalisation égale à la distance de l'arête



Comment pourrait-on obtenir une abstraction plus proche de la réalité ?

Plus de points de passages

Heure de la journée (embouteillage ?)

Présence de travaux routiers

...

Améliorer l'abstraction permet de mieux refléter la réalité,
au détriment d'un modèle plus difficile à résoudre

Taille de l'ensemble des états

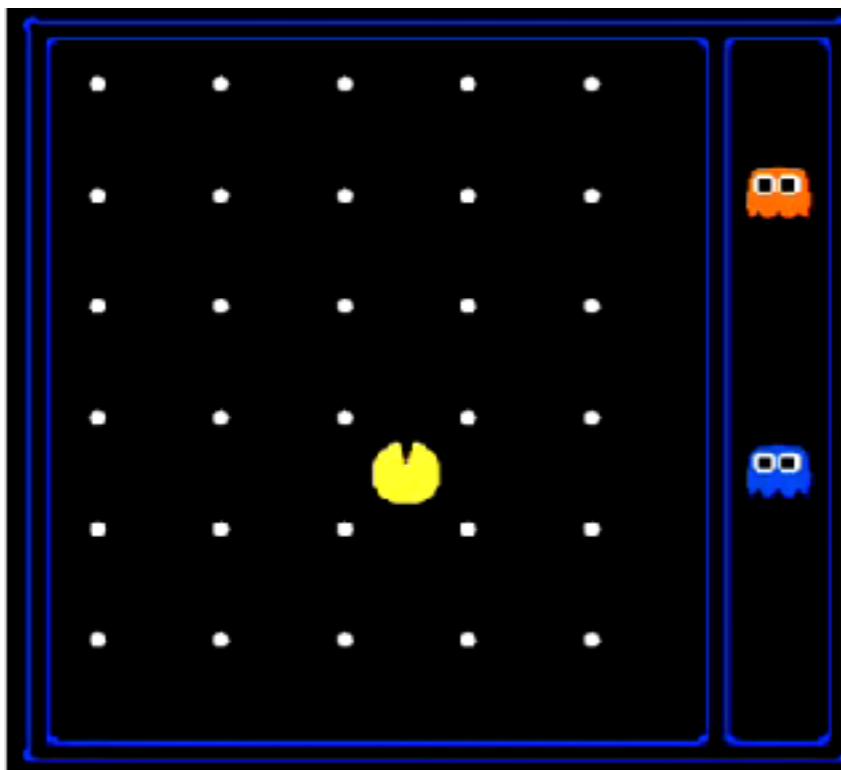
Taille de l'ensemble des états

Un problème de recherche vise à trouver une séquence d'actions amenant un état initial à un état final

Intuitivement, plus le nombre d'états du modèle est grand, plus le problème sera difficile à résoudre

Il est important d'être capable de calculer le nombre d'états d'un modèle

Exemple



Positions possibles pour Pacman: 120 120^1

Nombre de points à manger: 30 2^{30}

Positions possibles pour un fantôme: 12 12^2

?

Combien d'états comporte cette représentation globale ?

Présence d'une nourriture spécifique: valeur binaire (0 ou 1)

$$\# \text{ états} = 120 \times 2^{30} \times 12^2 = 18,554,258,720$$

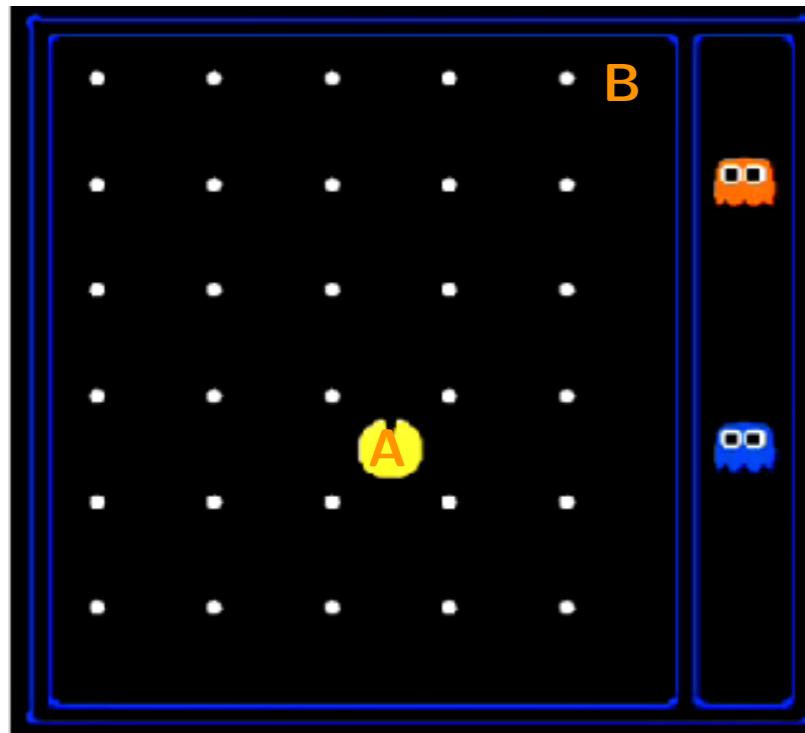
Une difficulté majeure des problèmes de recherche vient de la taille énorme de l'espace à explorer

Bonne nouvelle: en fonction du problème à résoudre, une partie des informations peut être ignorée

Problème de recherche 1: chemin entre deux points

Problème de recherche 1

Objectif: trouver le chemin le plus court pour amener Pacman d'un point A à un point B



? Comment pourrait-on modéliser ce problème ?

Ensemble d'états: positions possibles pour Pacman (x,y)

Etat initial: coordonnées (x,y) de la position A

Etat final: coordonnées (x,y) de la position B

Actions: gauche, droite, haut, bas

Transition et coût: déplacement de Pacman, avec un coût unitaire

Positions de Pacman: 120

Nombre de points: 30

Position par fantôme: 12

? Combien a t-on d'états ? # états = 120

Seulement les positions de Pacman sont utilisées (et non les fantômes ou les nourritures)

Toutes les informations de l'environnement global ne sont pas nécessaires

A ce stade, on ne fait que modéliser le problème (sans essayer de le résoudre)

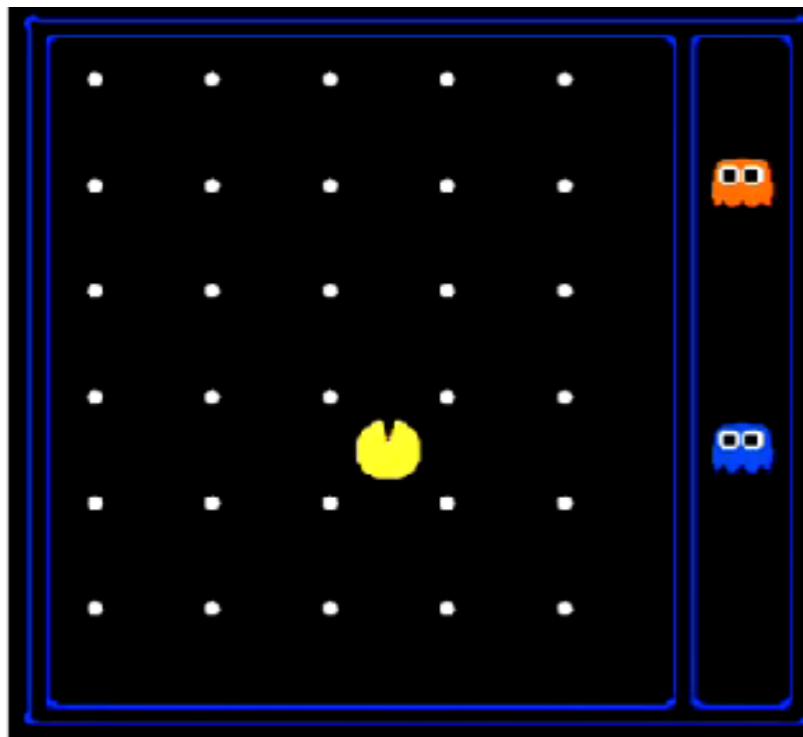
Problème de recherche 2: manger tous les points

Problème de recherche 2

Objectif: trouver le chemin le plus court pour amener Pacman à manger tous les points



Comment pourrait-on modéliser ce problème ?



Ensemble d'états:

- (1) positions possibles pour Pacman (x,y) - celles des points sont fixes
- (2) indication binaire indiquant si les points ont été mangé ou non

Etat initial: coordonnées initiales de Pacman, et 0 pour tous les points

Etat final: valeur de 1 pour tous les points

Actions: gauche, droite, haut, bas

Transition et coût: déplacement et prise de Pacman (coût unitaire)



Combien a t-on d'états ?

états = $120 \times 2^{30} = 128,849,018,880$



Quelles informations devraient être contenues dans un état ?

- (1) Informations nécessaires pour construire la fonction de transition
- (2) Informations nécessaires pour tester si un état est final

Représentation du problème par un graphe des états

Graphe des états

La formalisation d'un problème de recherche permet de représenter le problème comme un graphe dirigé

Noeud du graphe: un état possible

Arête du graphe: une action possible, liant un état avec son successeur

Poids de l'arête: coût de la transition

Propriétés de cette représentation

(1) Représente explicitement le problème à résoudre

(2) Chaque état n'est représenté qu'une seule fois

(3) Le graphe est généralement trop grand pour être mis en mémoire

Va nous servir de base pour la construction d'algorithmes de résolution

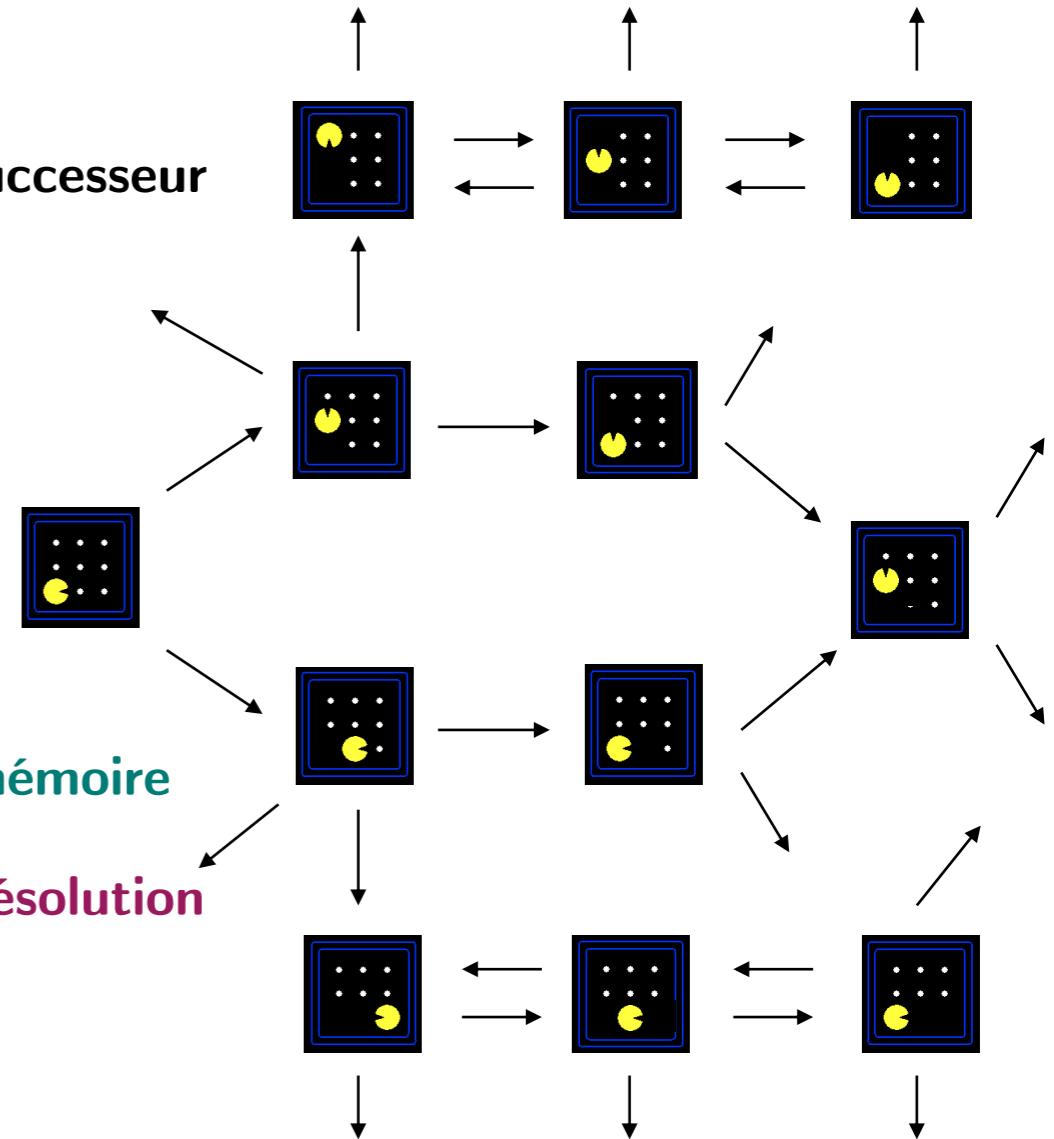
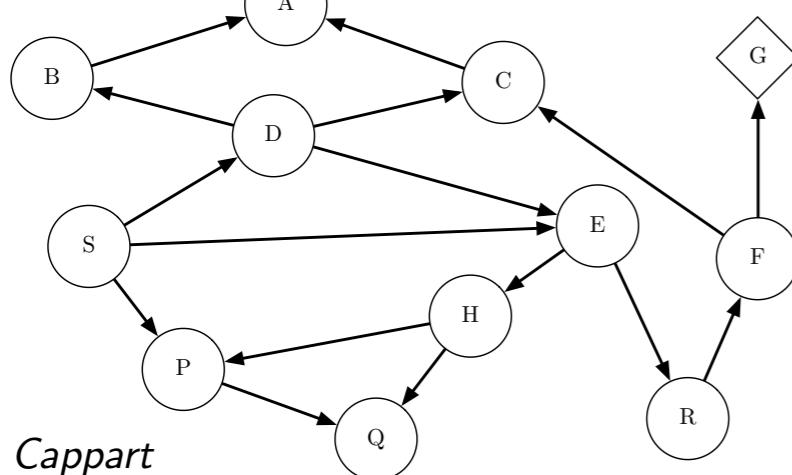


Illustration d'un algorithme de recherche

Procédure de recherche

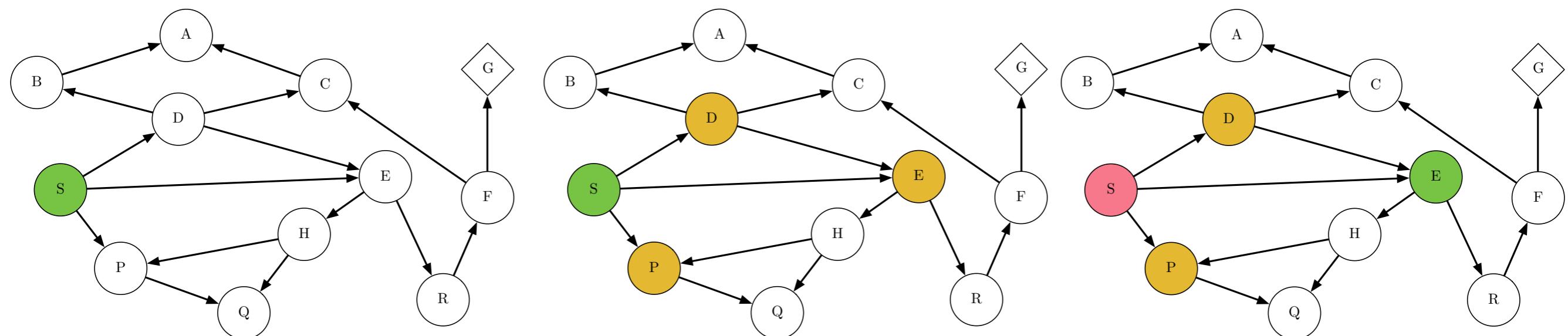
Etape 0: initialiser la recherche en se placant sur l'état initial (S)

Etape 1: maintenir une liste de candidats, correspondant à des voisins non étendus (fringe)

Etape 2: choisir un état dans cette liste, et étendre cet état (succession)

Etape 3: si le nouvel état n'est pas un état final (G), répéter le processus (Etapes 1 à 3)

Illustration



Etat initial

[state : S , fringe : $\langle \rangle$]

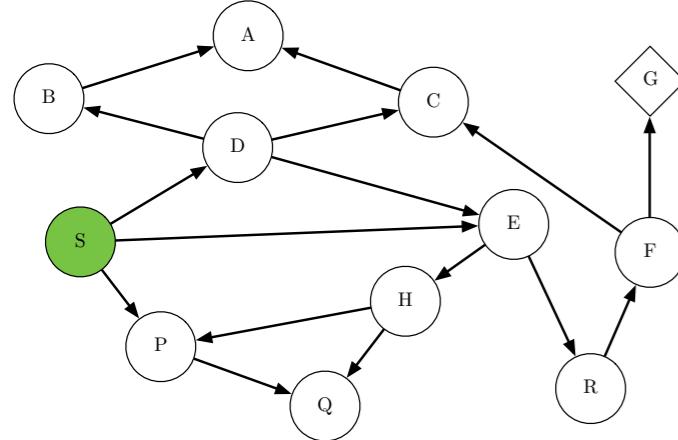
Mise à jour des voisins

[state : S , fringe : $\langle D, E, P \rangle$]

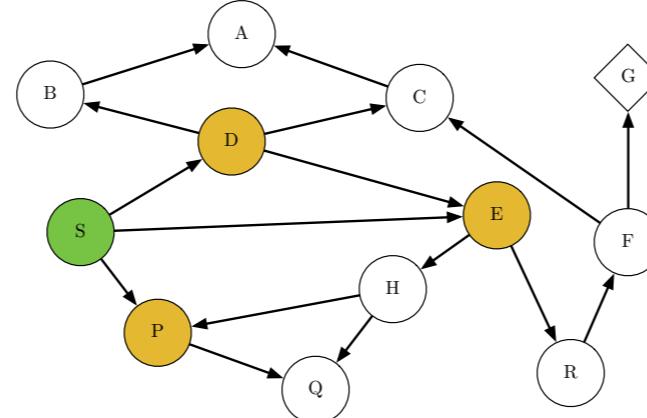
Succession

[state : E , fringe : $\langle D, P \rangle$]

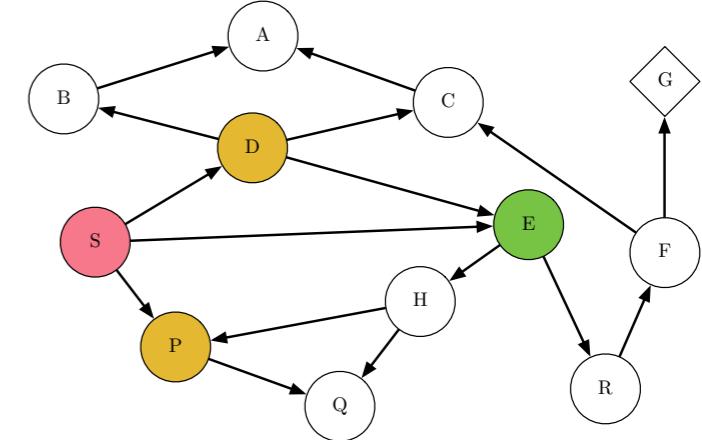
Illustration d'un algorithme de recherche



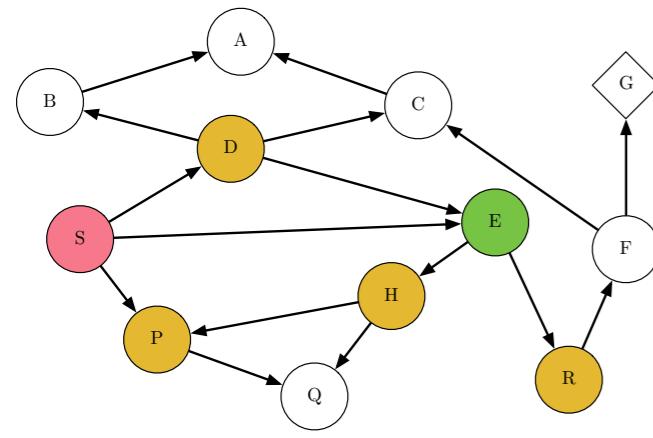
[state : *S*, fringe : $\langle \rangle$]



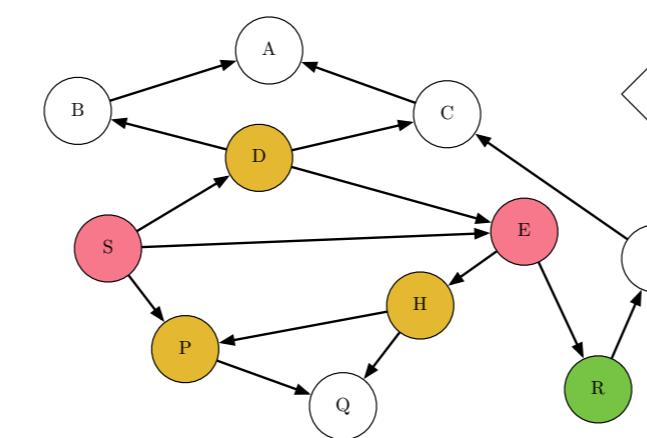
[state : *S*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{E}, \textcolor{red}{P} \rangle$]



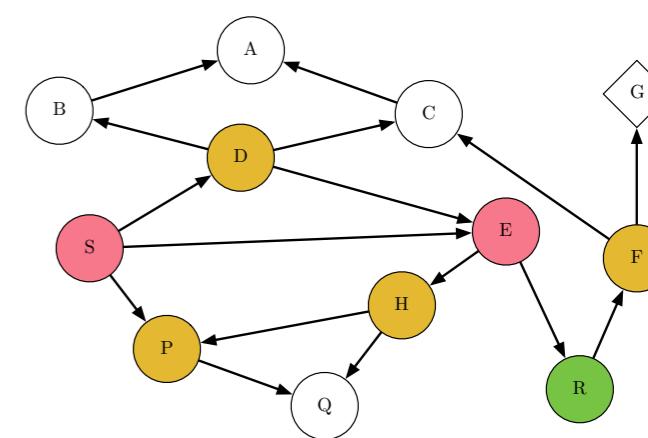
[state : *E*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P} \rangle$]



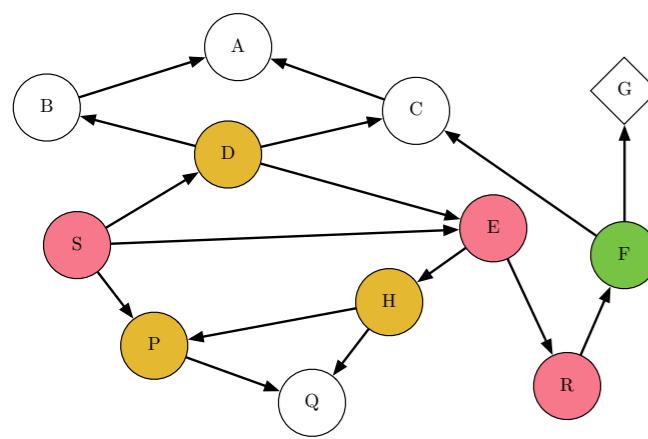
[state : *E*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P}, \textcolor{red}{H}, \textcolor{red}{R} \rangle$]



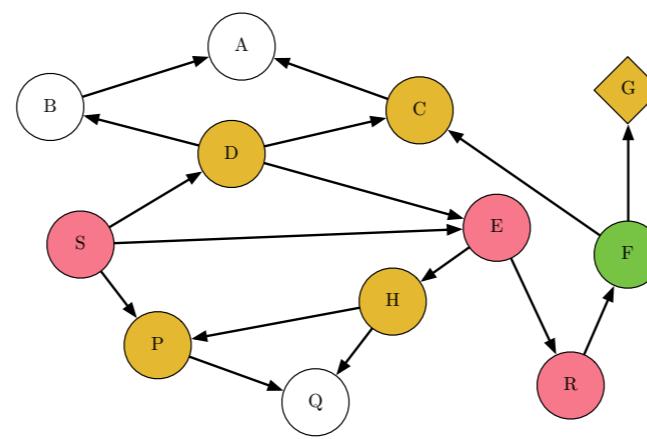
[state : *R*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P}, \textcolor{red}{H} \rangle$]



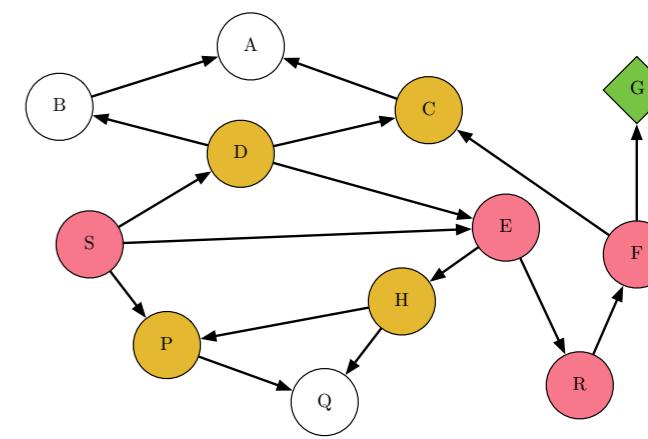
[state : *R*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P}, \textcolor{red}{H}, \textcolor{red}{F} \rangle$]



[state : *F*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P}, \textcolor{red}{H} \rangle$]



[state : *F*, fringe : $\langle \textcolor{red}{D}, \textcolor{red}{P}, \textcolor{red}{H}, \textcolor{red}{G} \rangle$]



[state : *G*, path : *S* → *E* → *R* → *F* → *G*]

Algorithme de recherche en graphe (*graph search*)

Algorithme de recherche en graphe

GraphSearch(P) :

$s = \text{initialState}(P)$

$V = \emptyset$

$L = \{s\}$

while $L \neq \emptyset$:

$s = \text{selectAndRemove}(L)$

 if $s = \text{goalState}(P)$: return solution

 else :

$C = \{c \in \text{succesors}(s, P) \mid c \text{ not in } V\}$

$L = L \cup C$

$V = V \cup s$

return no solution

Commencer à l'état initial du problème à résoudre (P)

Structure qui va garder en mémoire les états déjà visités

Ensemble des états candidats (fringe)

Tant qu'on a des candidats à étendre

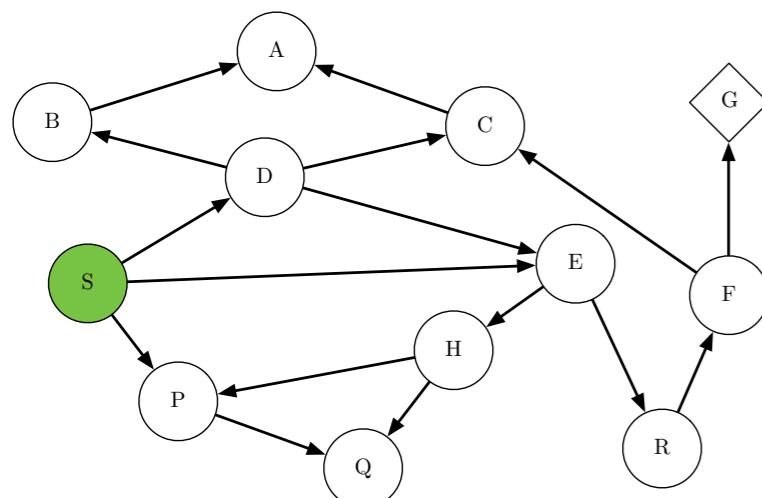
Retirer un noeud candidat de la fringe, et l'étendre

Si l'état est final, retourner une solution (chemin du noeud initial au final)

Considérer les états successeurs comme candidats, s'ils n'ont pas encore été pris...

... et les ajouter dans la fringe

Mettre à jour l'ensemble des états visités



Quel est le point de conception majeur de cet algorithme ?

Comment choisir le prochain noeud à étendre ? (gestion de la Fringe)

Cette question va donner lieu à différents algorithmes de recherches

Algorithme de recherche en graphe (*graph search*)

Algorithme de recherche en graphe

GraphSearch(P) :

$s = \text{initialState}(P)$

$V = \emptyset$

$L = \{s\}$

while $L \neq \emptyset$:

$s = \text{selectAndRemove}(L)$

if $s = \text{goalState}(P)$: return solution

else :

$C = \{c \in \text{succesors}(s, P) \mid c \text{ not in } V\}$

$L = L \cup C$

$V = V \cup s$

return no solution

Très bonne base pour nos futurs algorithmes

?

Quelle est la faiblesse majeure de cet algorithme

Il demande de retenir tous les états visités (V)

Or le nombre d'états est souvent extrêmement grand

états Pacman = $120 \times 2^{30} = 128,849,018,880$

La mémoire va vite être saturée

?

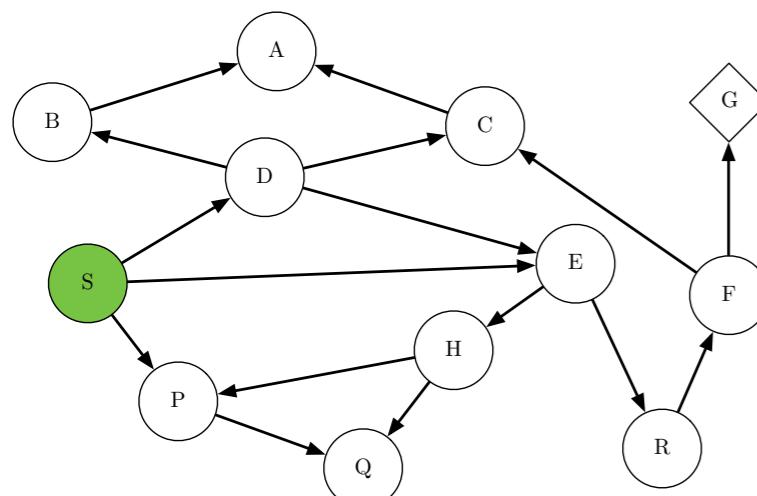
Peut-on régler facilement ce problème ?

Oui, en ne retenant pas les états déjà visités

Permet de réduire drastiquement la consommation mémoire

Engendre un risque de retomber sur un état déjà vu

Idée de base de la recherche en arbre (*tree search*)



Algorithme de recherche en arbre (tree search)

Algorithme de recherche en arbre

Le problème est représenté comme un arbre, sur base du graphe des états

Racine de l'arbre: l'état initial du problème

Noeud de l'arbre: un état possible

Successseurs d'un noeud: tous les états autorisés par la transition

Poids de l'arête: coût de la transition

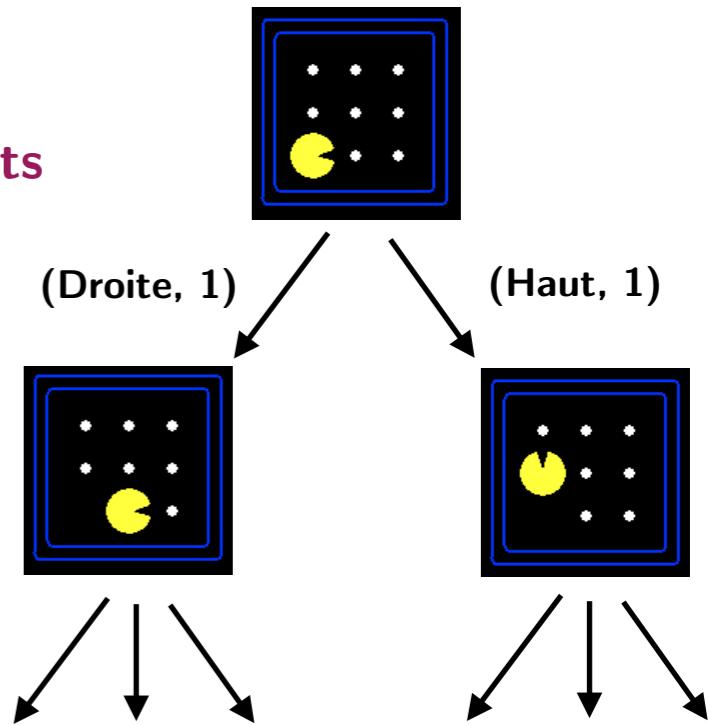
Propriétés

- (1) Les états déjà visités ne sont pas stockés
- (2) Algorithme similaire à la recherche en graphe
- (3) Chaque état peut apparaître plusieurs fois dans l'arbre
- (4) L'arbre complet est généralement trop grand pour la mémoire
- (5) Cependant, un stockage complet n'est pas nécessaire

Avantage: permet une meilleure efficacité mémoire

Inconvénient: recherche ralentie par la répétition des états

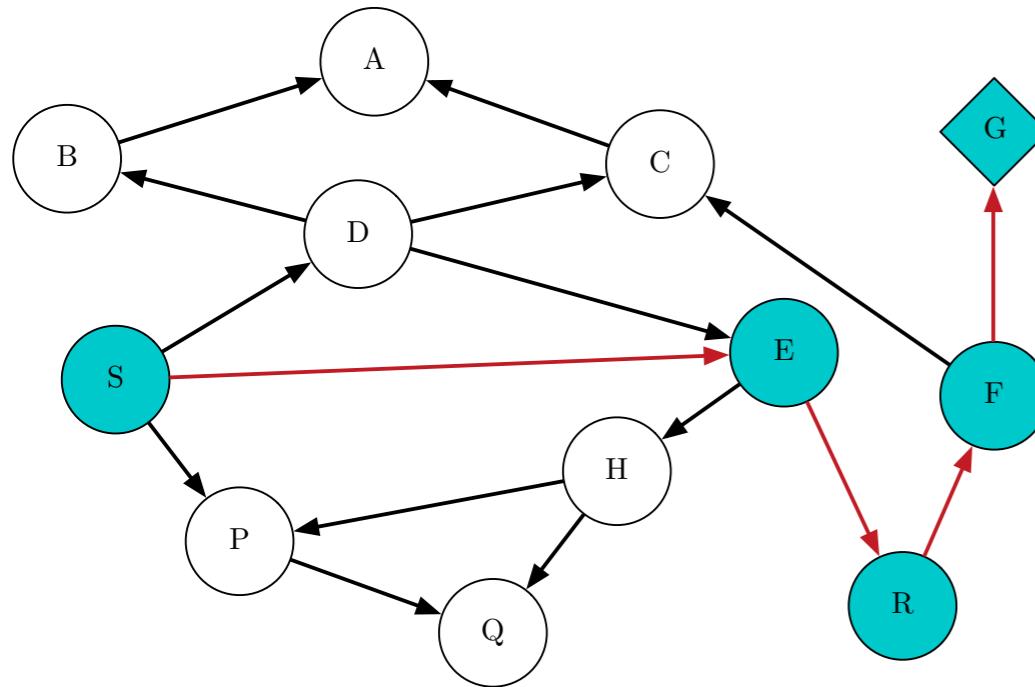
Conceptuellement, un noeud représente un état et la séquence d'actions pour y parvenir



```
TreeSearch( $P$ ) :  
 $s = \text{initialState}(P)$   
 $L = \{s\}$   
while  $L \neq \emptyset$  :  
     $s = \text{selectAndRemove}(L)$   
    if  $s = \text{goalState}(P)$  : return solution  
    else :  
         $C = \{c \in \text{successors}(s, P)\}$   
         $L = L \cup C$   
return no solution
```

Deux grandes familles d'algorithmes de recherches

Recherche en graphe



Chaque noeud correspond à un état

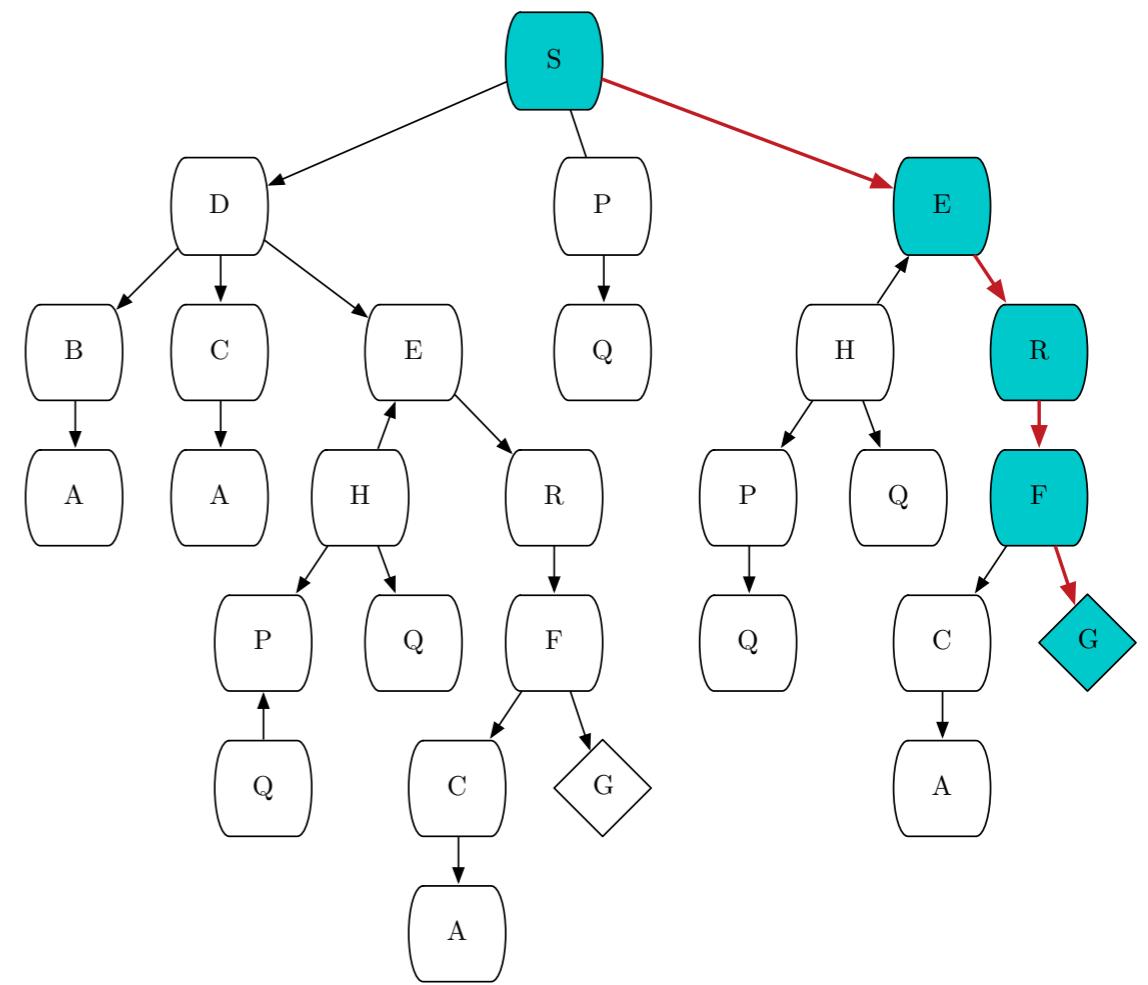
Recherche en graphe: consommation mémoire plus lourde, mais exécution plus rapide

Recherche en arbre: consommation mémoire moins lourde, mais exécution plus lente

Cela dépend également de la stratégie de recherche utilisée

Les deux stratégies ont également des difficultés spécifiques

Recherche en arbre

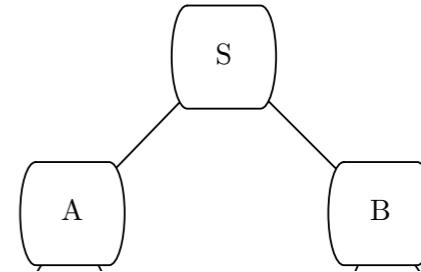
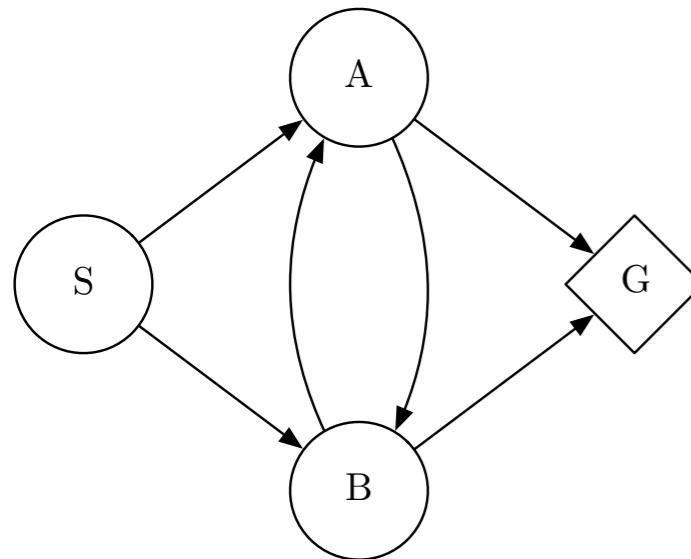


Chaque noeud est relatif au chemin emprunté

Difficulté de la recherche en arbre

Exemple pathologique

Considérons le modèle suivant



Combien de noeuds comporte l'arbre de recherche correspondant ?

Une infinité... ...même si le nombre d'états est fini, et si un état final existe (G)

Si on n'y prend pas garde, une recherche dans un arbre peut cybler indéfiniment

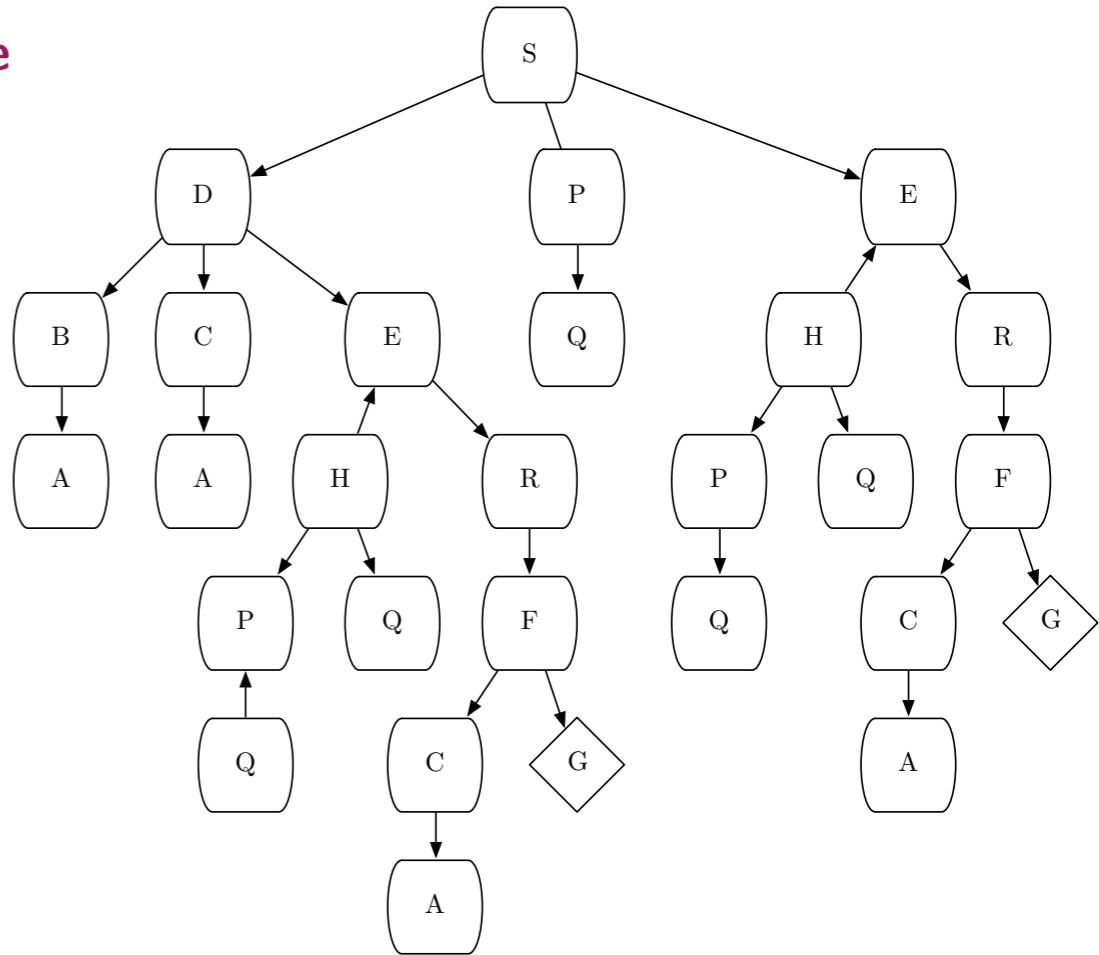
Cycle: $S \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow \dots$

La possibilité d'avoir un arbre infini est une considération cruciale pour la recherche en arbre

Table des matières

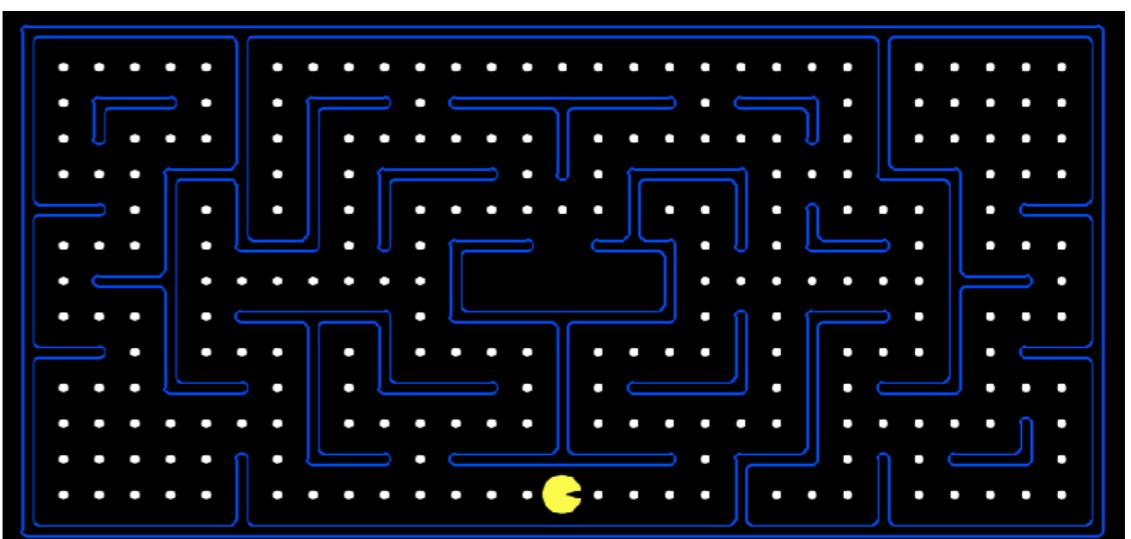
Stratégies de recherche

- ✓ 1. Définition et modélisation d'un problème de recherche
- ✓ 2. Agents réflexes
- ✓ 3. Agents axés sur la recherche
- 4. Recherche en arbre (*tree search*)
- 5. Recherche sans information: DFS, BFS, UCS, IDS
- 6. Recherche avec information: *greedy search, A**
- 7. Conception d'heuristiques
- 8. Recherche en graphe (*graph search*)



Problèmes abordés

- 1. *Pacman*
- 2. *8-puzzle*
- 3. Planification de routes



Algorithme de recherche en arbre

TreeSearch(P) :

$s = \text{initialState}(P)$

$L = \{s\}$

while $L \neq \emptyset$:

$s = \text{selectAndRemove}(L)$

if $s = \text{goalState}(P)$: return solution

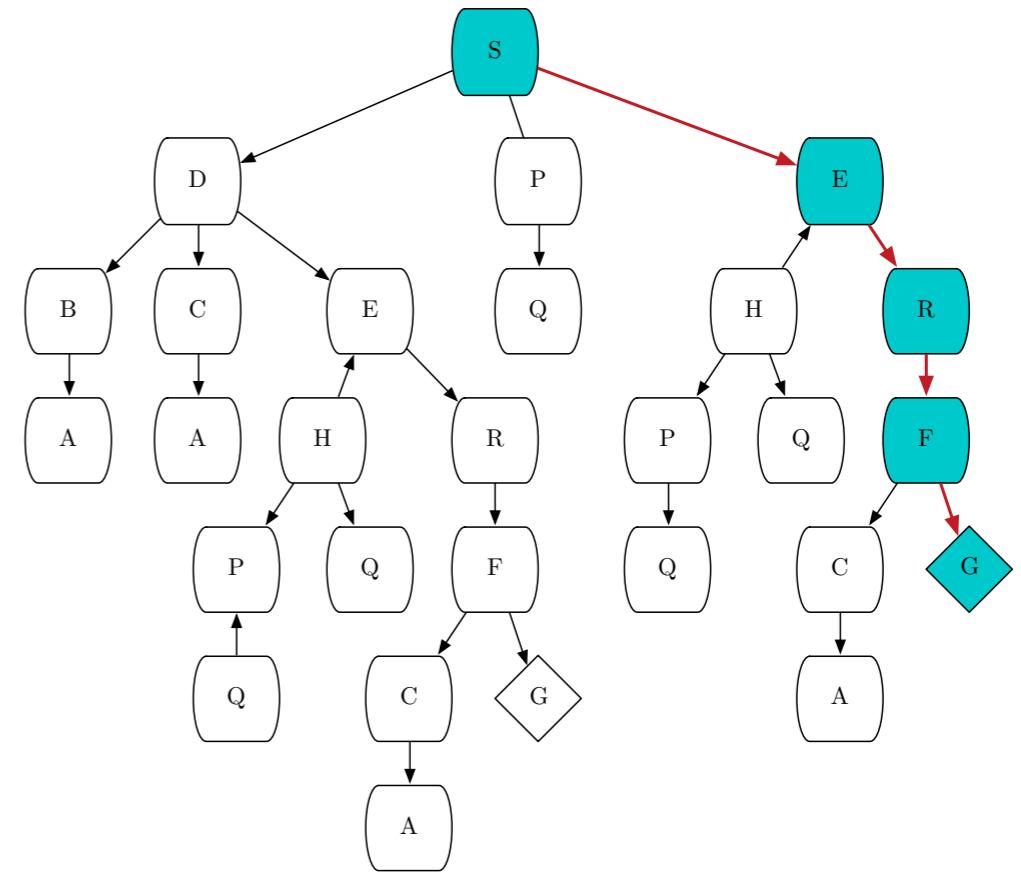
else :

$C = \{c \in \text{succesors}(s, P)\}$

$L = L \cup C$

return no solution

$V = \emptyset$



Quel est le point de conception majeur de cet algorithme ?

Comment choisir le prochain noeud à étendre ? (gestion de la Fringe)

Similaire à la recherche en graphe

Cette question va donner lieu à différents algorithmes de recherche

Ce choix va impacter directement les performances de notre recherche

Critères de performance d'une recherche



Comment évaluer l'efficacité d'une stratégie de recherche ?

Level
Complete



Level
Complete



Complétude (*completeness*)

Garantie que la recherche va trouver une solution (s'il en existe une)

Dans le cas contraire, garantie qu'un échec est reporté

Optimalité (*optimality*)

Garantie que la recherche retourne la meilleure solution existante au problème

Cette solution correspond à celle engendrant le moins de coûts

Complexité temporelle (*time complexity*)

Analyse du temps de calcul nécessaire pour exécuter la recherche

$\mathcal{O}(n)$

Exprimée par la notation asymptotique dans le pire des cas

Complexité spatiale (*space complexity*)

Analyse de la consommation mémoire nécessaire pour exécuter la recherche

Exprimée par la notation asymptotique dans le pire des cas



Une analyse d'un algorithme de recherche devrait tenir compte de ces quatre aspects

Complexité d'une stratégie de recherche en arbre

Paramètres

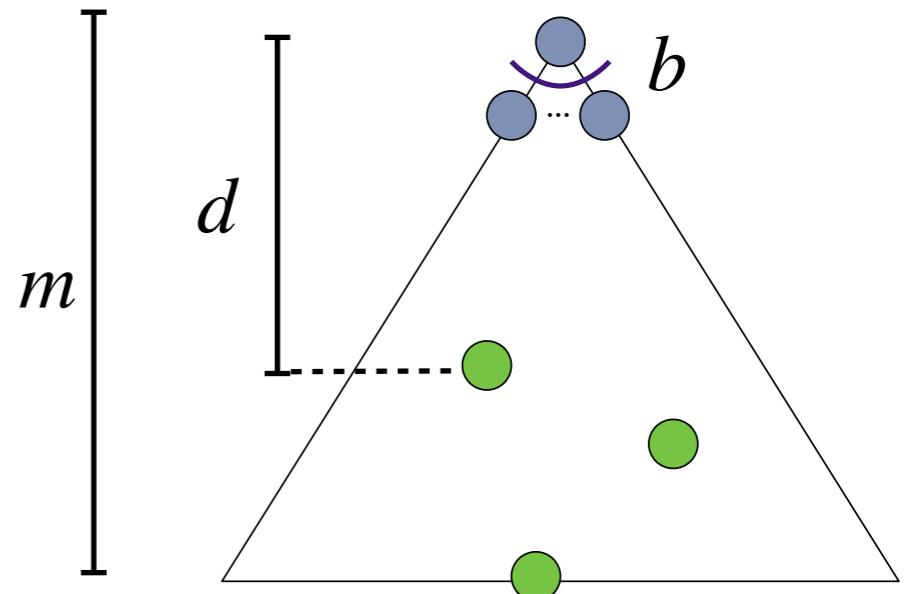
■ : noeud de recherche

● : noeud correspondant à un état final

b : nombre maximum de successeurs (branching factor)

m : profondeur maximale de l'arbre (depth)

d : profondeur de la solution la plus proche



Combien de noeuds, au maximum, y a t-il dans un arbre à facteur de branchement b et de profondeur m ?

Nombre de noeuds dans l'arbre

Niveau 1 : 1 noeud

Niveau 2 : b noeuds

Niveau 3 : b^2 noeuds

Niveau m : b^m noeuds

$$\#\text{noeuds} = 1 + b + b^2 + \dots + b^m = \mathcal{O}(b^m)$$

$(b = 5, m = 10) \rightarrow 12,207,030$ noeuds

$(b = 5, m = 20) \rightarrow 119,209,289,550,780$ noeuds

$(b = 20, m = 10) \rightarrow 10,778,947,368,420$ noeuds



Non ! Et c'est ça qui fait de la recherche en arbre une bon type de stratégie

Recherche en profondeur (DFS - *depth first search*)

Rappel

Point de conception: choisir le noeud à étendre en priorité (*fringe*)

Recherche en profondeur

Retirer systématiquement le dernier noeud ajouté à la *fringe*

Revient à retirer en priorité le noeud le plus profond dans l'arbre

La *fringe* est une structure LIFO (*last-in first-out*)

$\text{push}(L, s)$: ajoute l'élément s à la structure L

$\text{pop}(L)$: retire de L le dernier élément ajouté

Exemple

Notation: l'exposant indique l'état parent d'un état

Etape 1 : [state : S , fringe : $\langle D^S, P^S, E^S \rangle$]

Etape 2 : [state : D^S , fringe : $\langle B^D, C^D, E^D, P^S, E^S \rangle$]

Etape 3 : [state : B^D , fringe : $\langle A^B, C^D, E^D, P^S, E^S \rangle$]

Etape 4 : [state : A^B , fringe : $\langle C^D, E^D, P^S, E^S \rangle$]

Etape 5 : [state : C^D , fringe : $\langle A^C, E^D, P^S, E^S \rangle$]

Etape 6 : [state : A^C , fringe : $\langle E^D, P^S, E^S \rangle$]

Etape 7 : [state : E^D , fringe : $\langle H^E, R^E, P^S, E^S \rangle$]

...

DepthFirstSearch(P) :

$s = \text{initialState}(P)$

$L = \text{LIFO}()$

$\text{push}(L, s)$

while $L \neq \emptyset$:

$s = \text{pop}(L)$

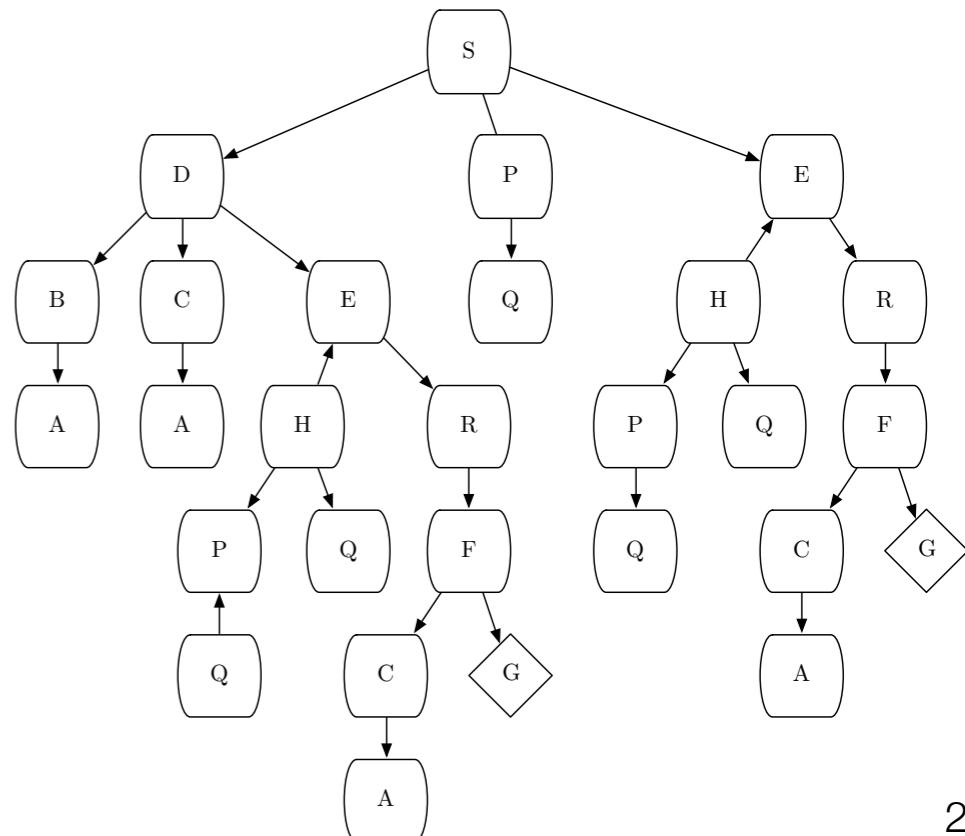
if $s = \text{goalState}(P)$: return solution

else :

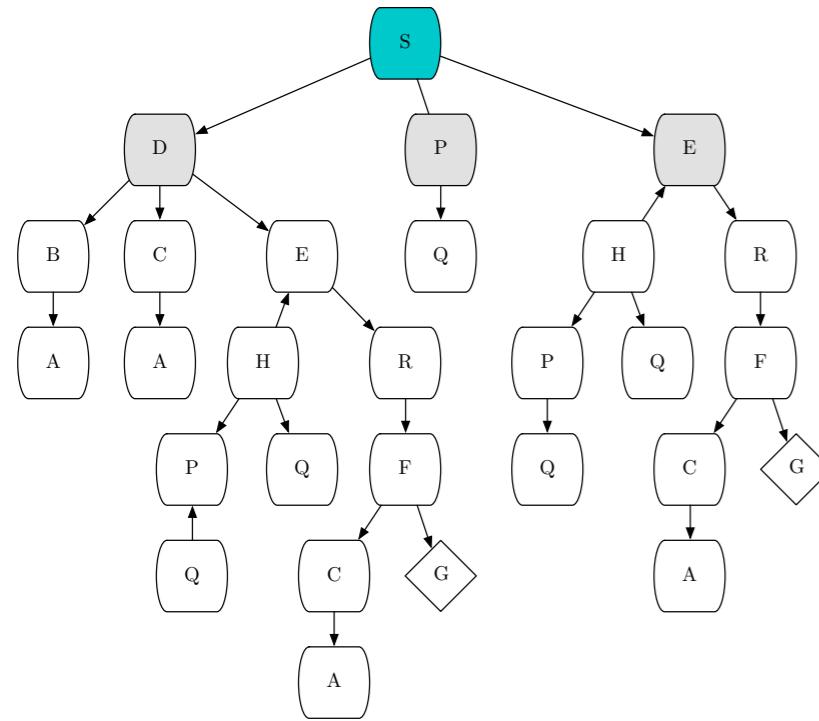
$C = \{c \in \text{successors}(s, P)\}$

$\text{push}(L, C)$

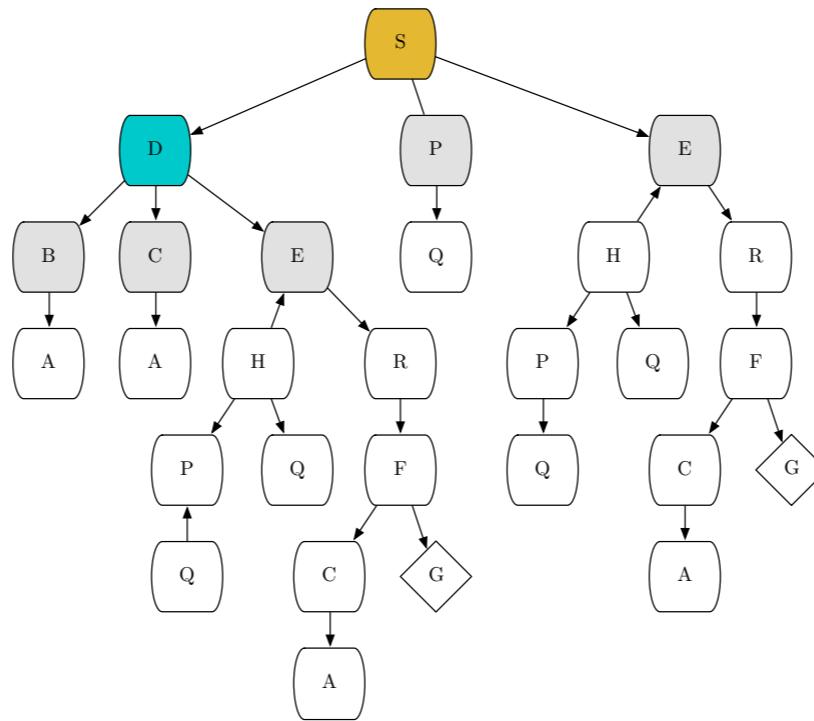
return no solution



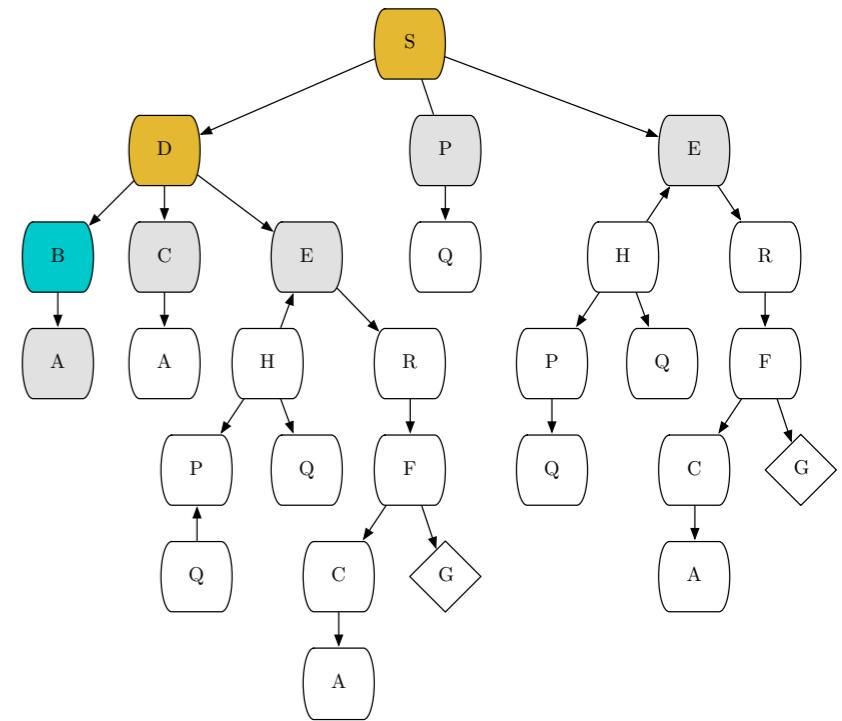
Recherche en profondeur - illustration



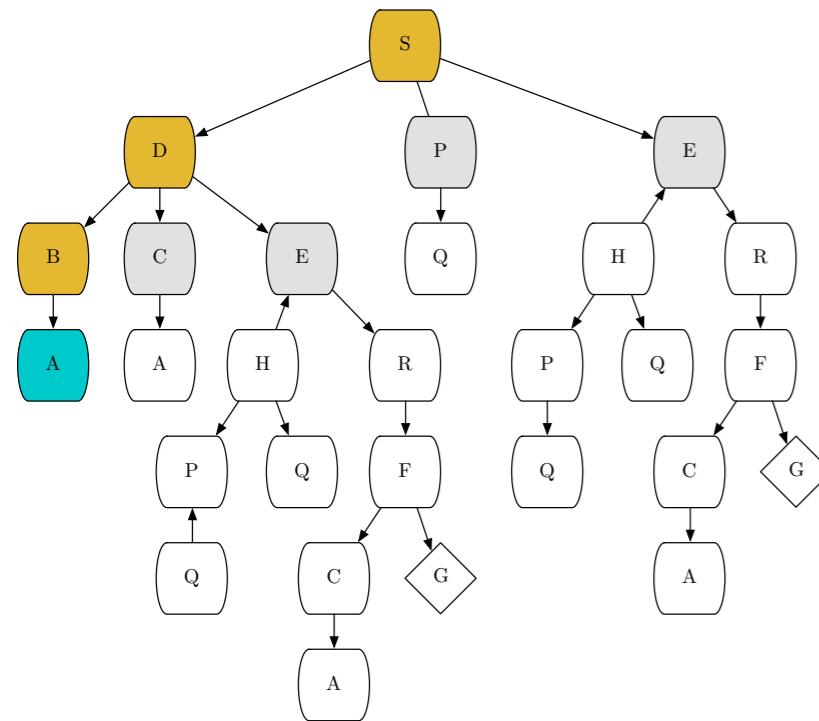
state : S , **fringe** : $\langle D^S, P^S, E^S \rangle$



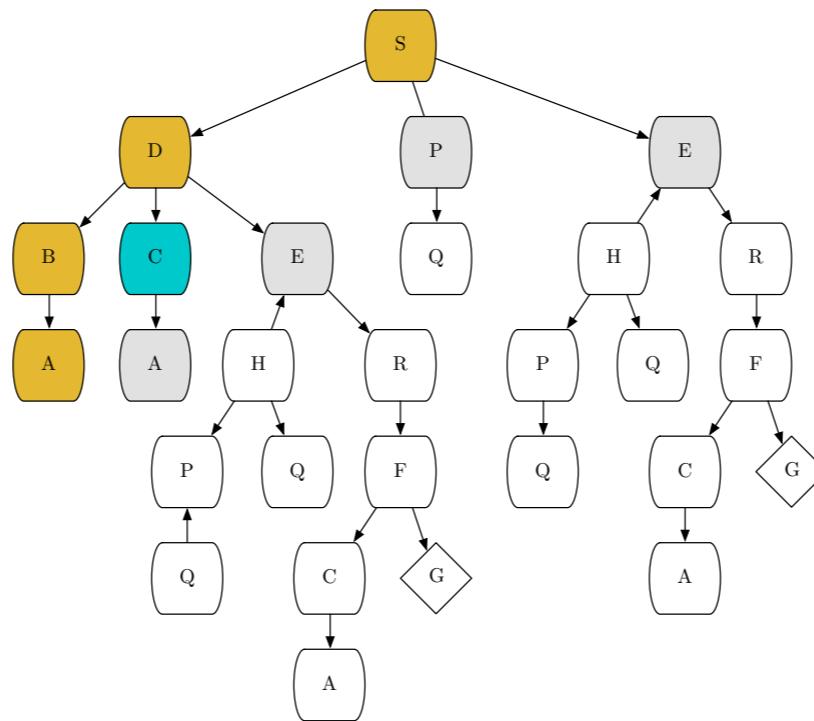
state : D^S , **fringe** : $\langle B^D, C^D, E^D, P^S, E^S \rangle$



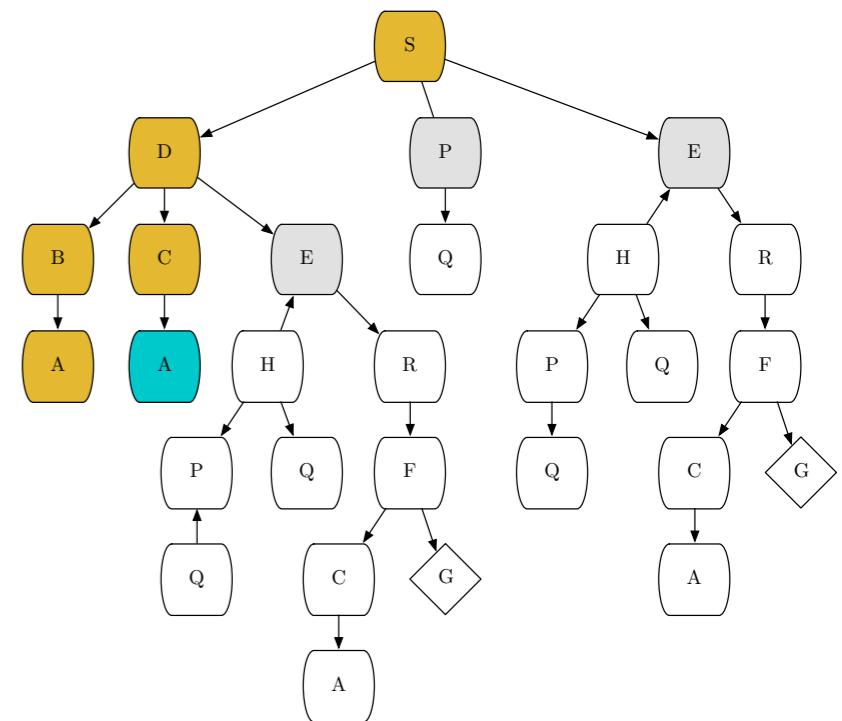
state : B^D , **fringe** : $\langle A^B, C^D, E^D, P^S, E^S \rangle$



state : A^B , **fringe** : $\langle C^D, E^D, P^S, E^S \rangle$



state : C^D , **fringe** : $\langle A^C, E^D, P^S, E^S \rangle$



state : A^C , **fringe** : $\langle E^D, P^S, E^S \rangle$

Recherche en profondeur - analyse des performances

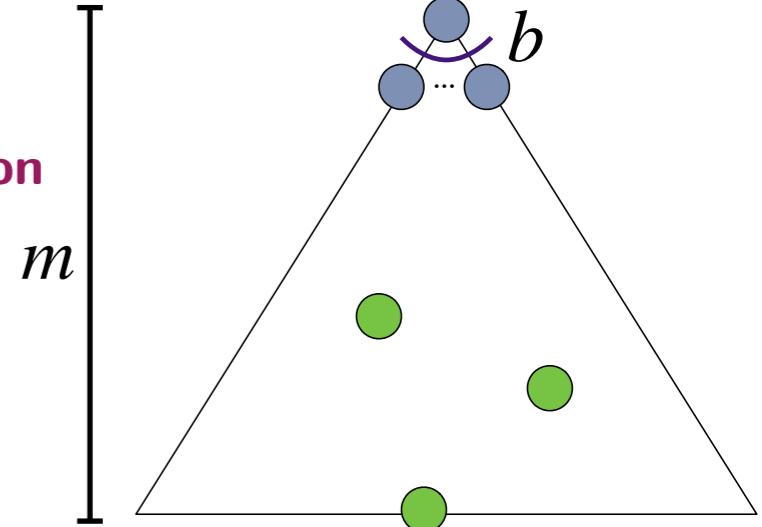


Quelles sont les performances d'une recherche en profondeur ?

Complexité temporelle

Dans le pire des cas, tout l'arbre doit être exploré pour trouver une solution

Complexité temporelle : $\mathcal{O}(b^m)$ (très coûteux)



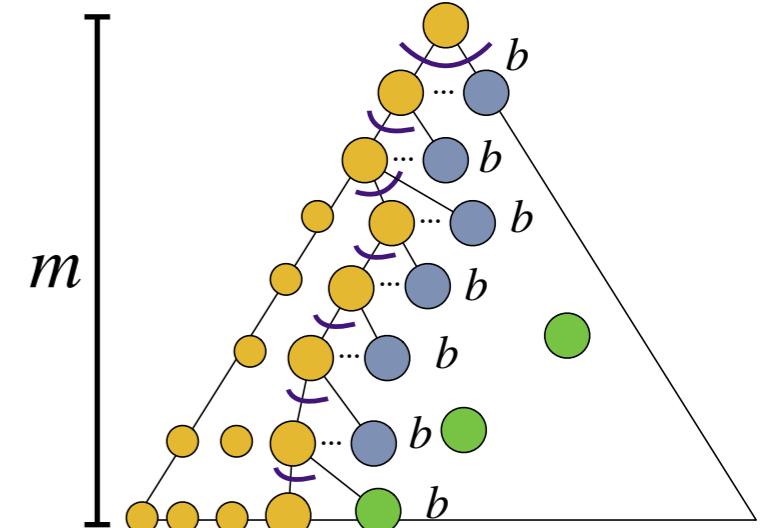
Complexité spatiale

Revient à évaluer le nombre maximum de noeuds stockés dans la Fringe

Uniquement les noeuds successeurs des noeuds se trouvant sur notre chemin actuel depuis la racine

Les autres noeuds déjà visités (hors du chemin) ne doivent plus être retenus

Complexité spatiale : $\mathcal{O}(bm)$ (très efficace)



Complétude

La profondeur (m) de l'arbre n'est pas bornée

La recherche peut plonger dans une profondeur infinie

La recherche est incomplète

Optimalité

La recherche s'arrête dès qu'une solution est trouvée, pas forcément de plus faible coût

La recherche est non-optimale

Recherche en largeur (BFS - breadth first search)

Recherche en largeur

Retirer systématiquement l'élément le plus ancien de la **fringe**

Revient à retirer en priorité le noeud le moins profond

La **fringe** est une structure FIFO (**first-in first-out**)

Le reste est identique à DFS !

push(L, s) : ajoute l'élément s à la structure L

pop(L) : retire l'élément le plus ancien de L

BreadthFirstSearch(P) :

$s = \text{initialState}(P)$

$L = \text{FIFO}()$

push(L, s)

while $L \neq \emptyset$:

$s = \text{pop}(L)$

if $s = \text{goalState}(P)$: return solution

else :

$C = \{c \in \text{successors}(s, P)\}$

push(L, C)

return no solution

Exemple

Etape 1 : [state : S , fringe : $\langle D^S, P^S, E^S \rangle$]

Etape 2 : [state : D^S , fringe : $\langle P^S, E^S, B^D, C^D, E^D \rangle$]

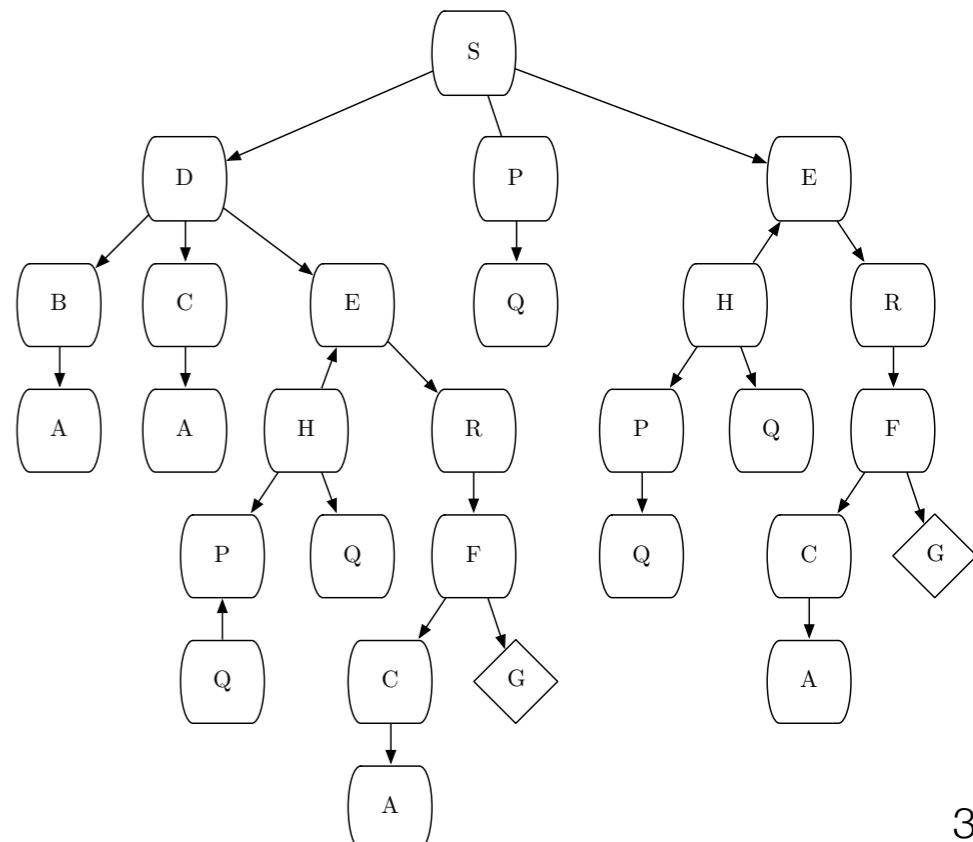
Etape 3 : [state : P^S , fringe : $\langle E^S, B^D, C^D, E^D, Q^P \rangle$]

Etape 4 : [state : E^S , fringe : $\langle B^D, C^D, E^D, Q^P, H^E, R^E \rangle$]

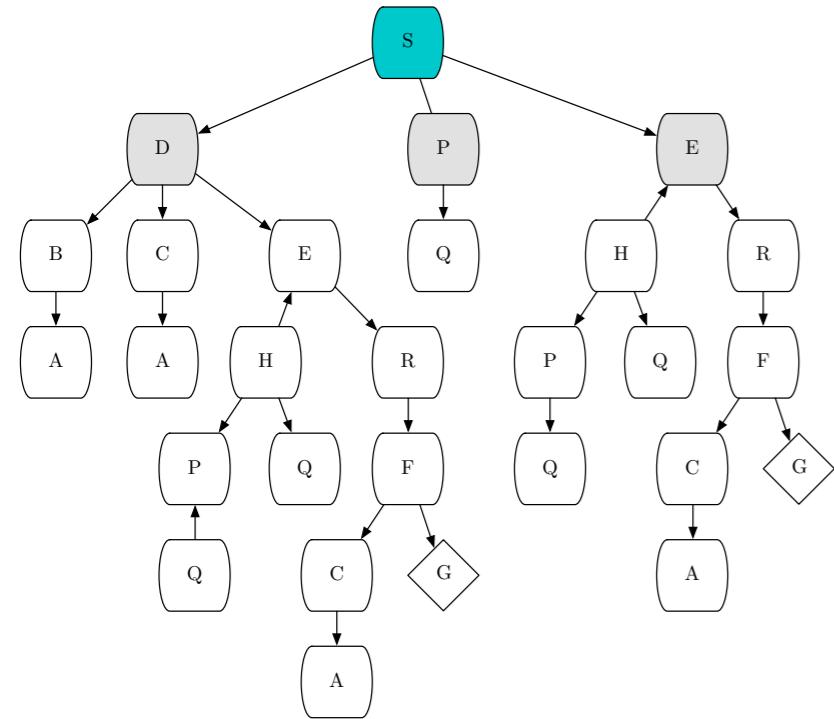
Etape 5 : [state : B^D , fringe : $\langle C^D, E^D, Q^P, H^E, R^E, A^B, A^C \rangle$]

Etape 6 : [state : C^D , fringe : $\langle E^D, Q^P, H^E, R^E, A^B, A^C, H^E, R^E \rangle$]

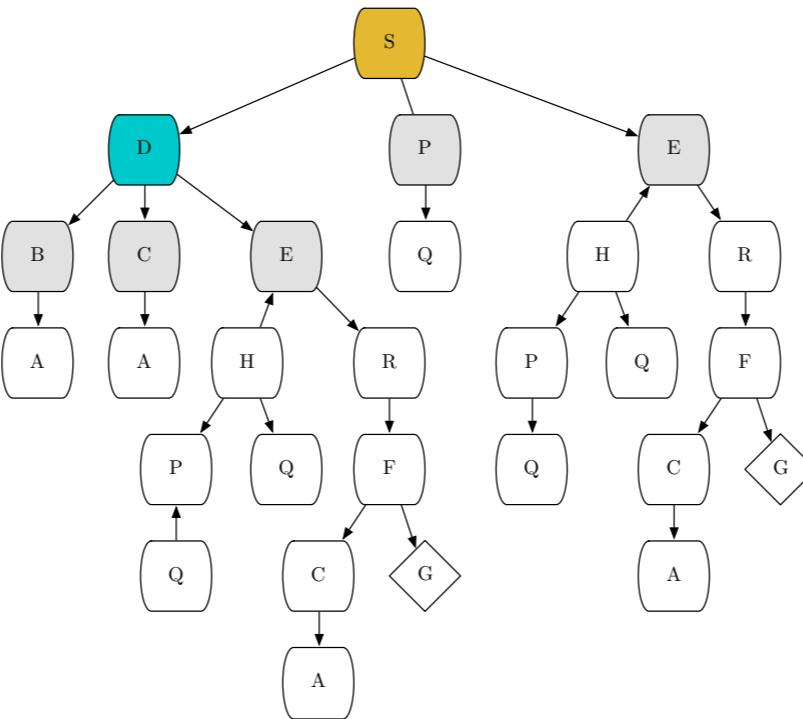
Etape 7 : [state : E^D , fringe : $\langle Q^P, H^E, R^E, A^B, A^C, H^E, R^E \rangle$]



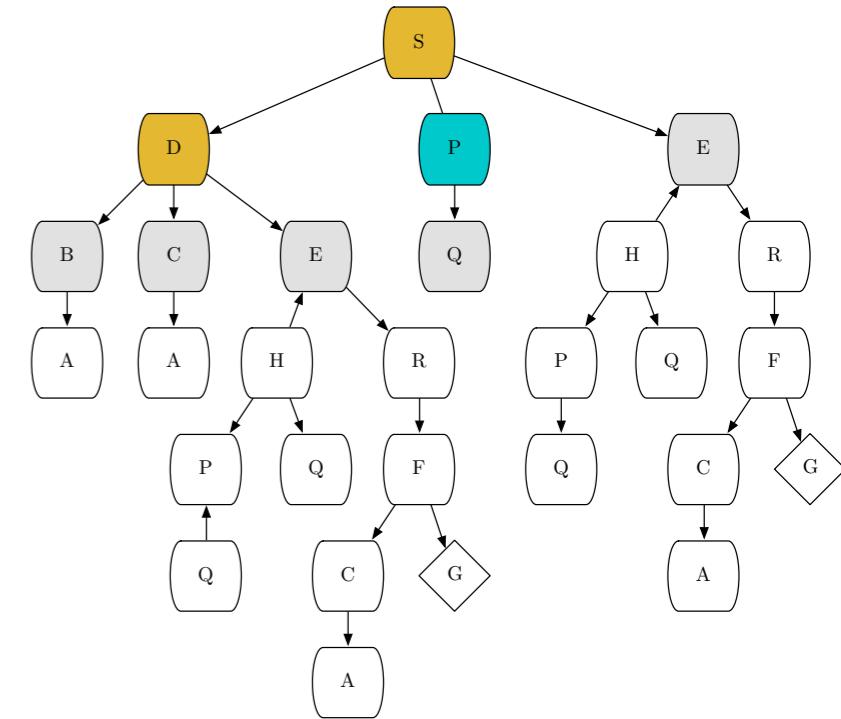
Recherche en largeur - illustration



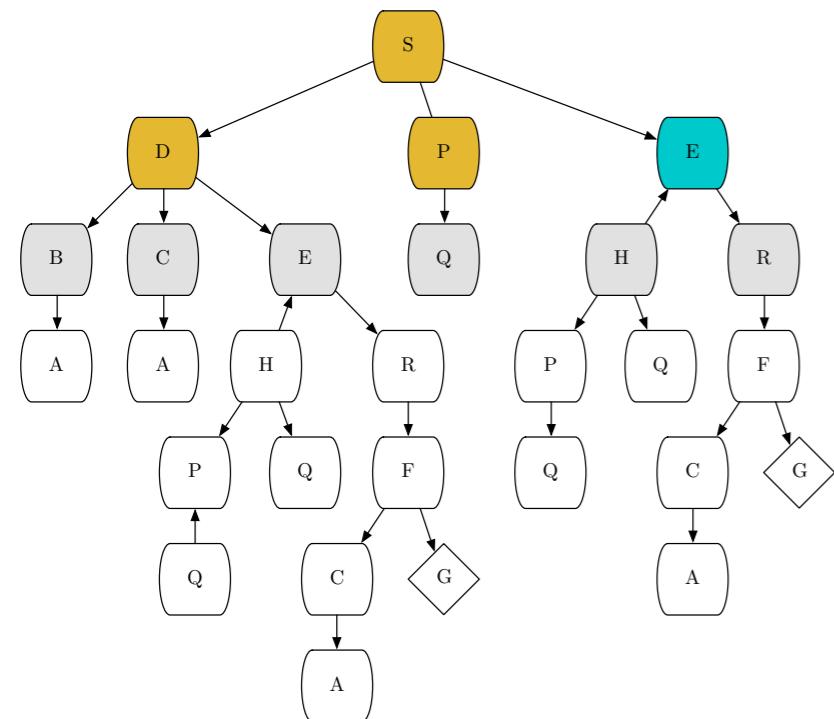
state : S , **fringe** : $\langle D^S, P^S, E^S \rangle$



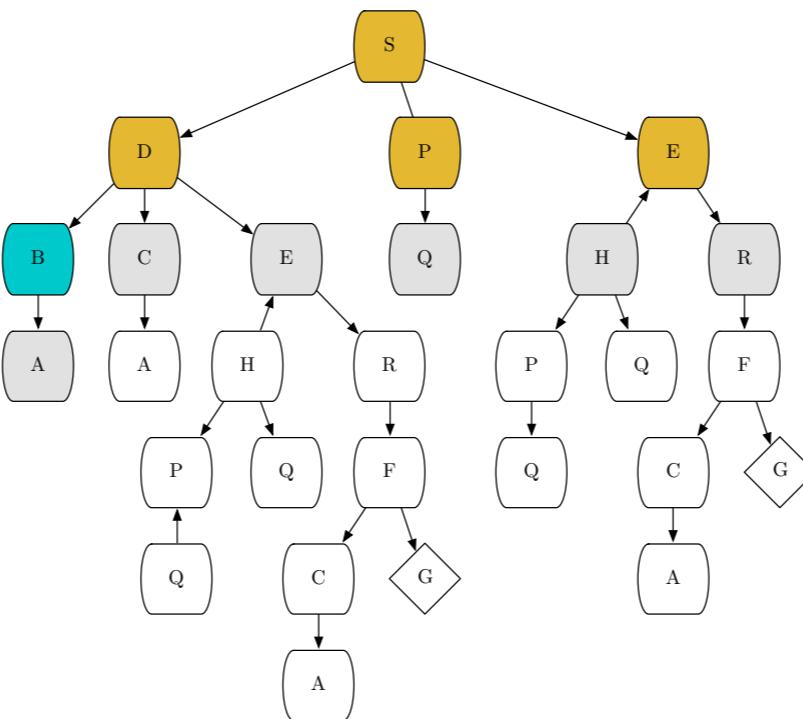
state : D^S , **fringe** : $\langle P^S, E^S, B^D, C^D, E^D \rangle$



state : P^S , **fringe** : $\langle E^S, B^D, C^D, E^D, Q^P \rangle$



state : E^S , **fringe** : $\langle B^D, C^D, E^D, Q^P, H^E, R^E \rangle$



state : C^D , **fringe** : $\langle E^D, Q^P, H^E, R^E, A^B, A^C \rangle$

state : B^D , **fringe** : $\langle C^D, E^D, Q^P, H^E, R^E, A^B \rangle$

Recherche en largeur - analyse



Quelles sont les performances d'une recherche en profondeur ?

Complexité temporelle

Tous les niveaux inférieurs au premier état final doivent être explorés

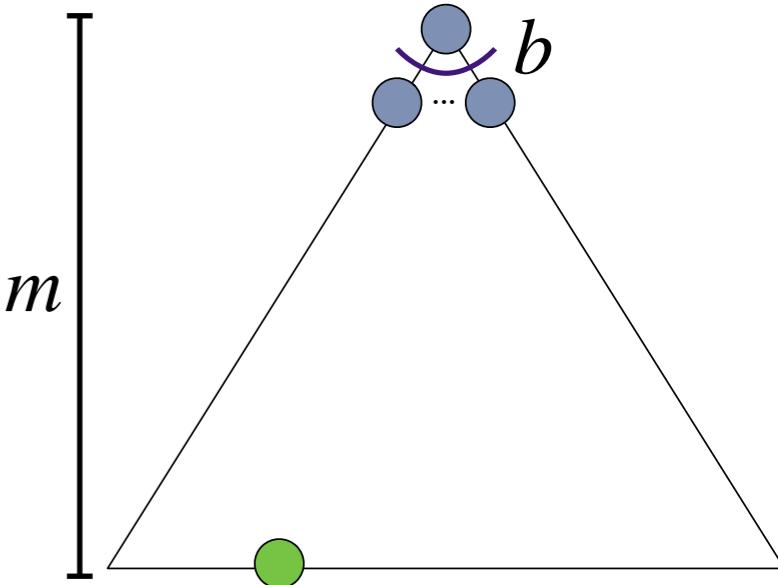
Complexité temporelle : $\mathcal{O}(b^d)$ (très mauvais)

Complexité spatiale

La *fringe* peut contenir tous les noeuds d'un niveau spécifique de l'arbre

Les autres noeuds visités (niveaux inférieurs) ne doivent plus être retenus

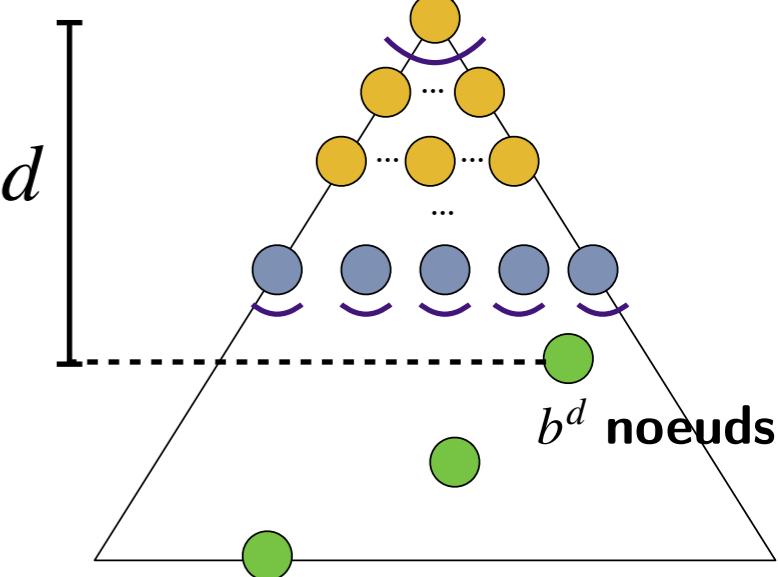
Complexité spatiale : $\mathcal{O}(b^d)$ (très mauvais)



Complétude

Même en cas de profondeur infinie une solution (si existante) sera trouvée

La recherche est complète



Optimalité

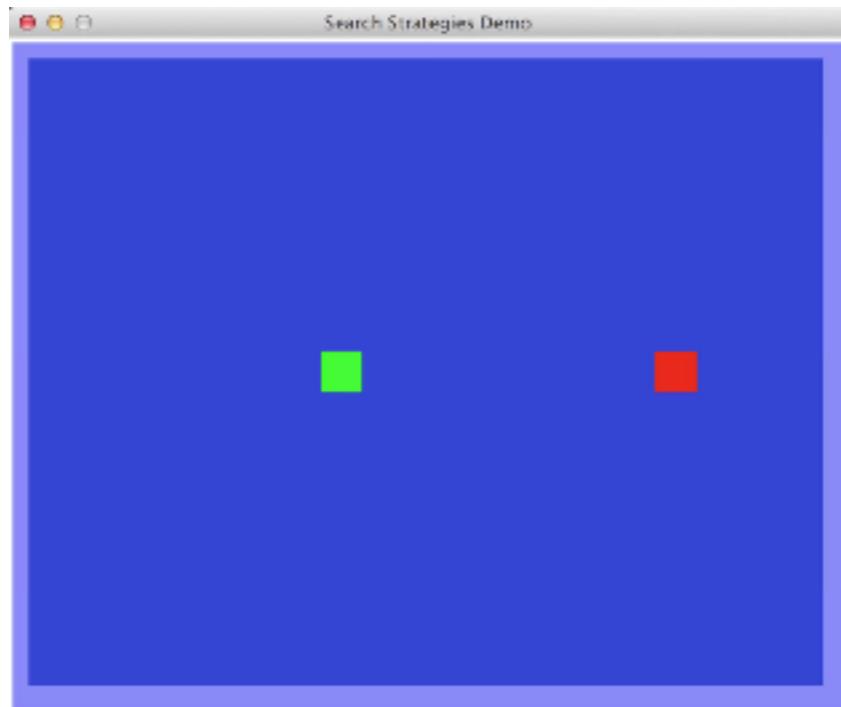
La solution retournée est celle ayant le moins d'actions

Optimal si tous les coûts ont la même valeur

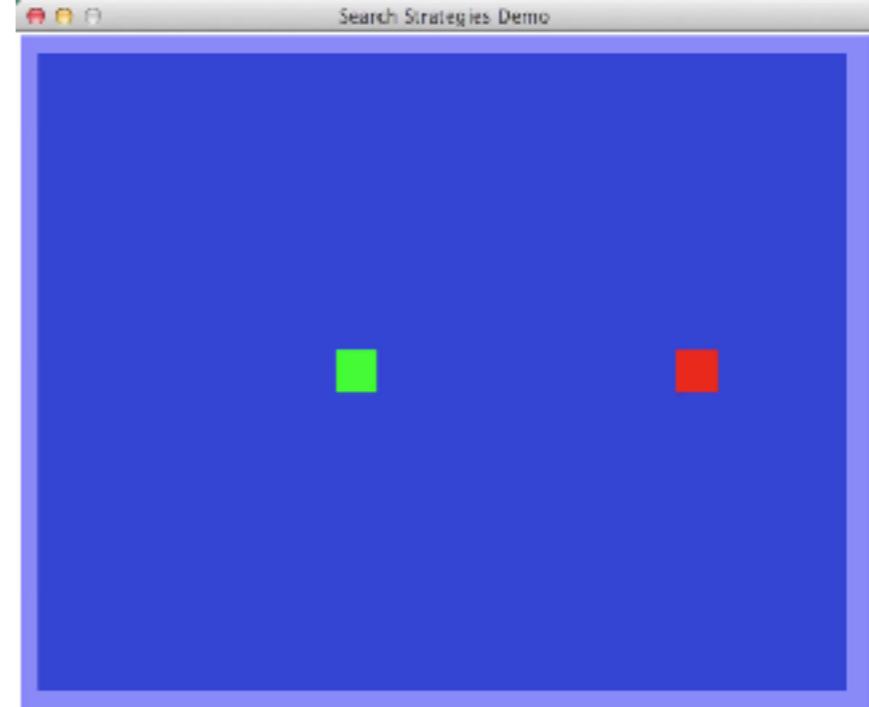
Non-optimal en général

DFS vs BFS: visualisation

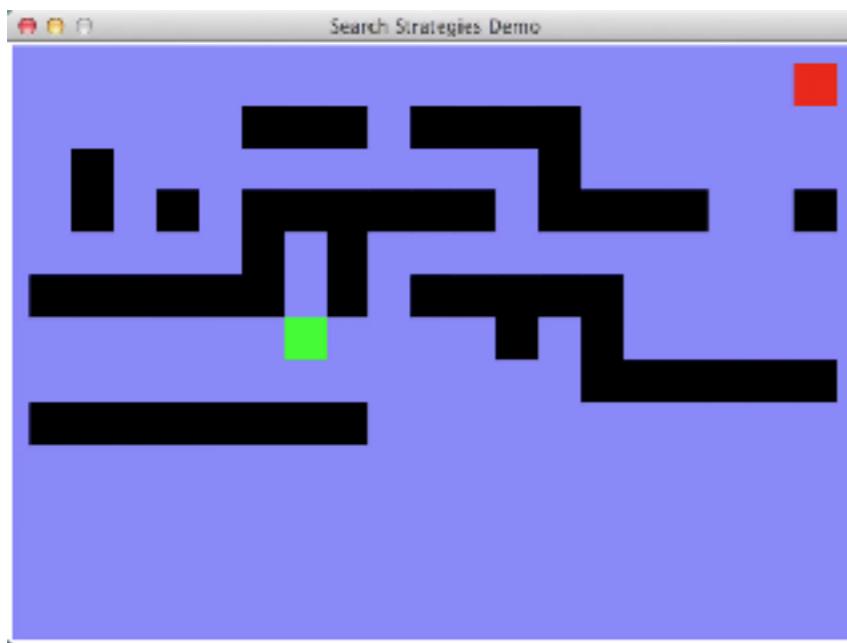
? Quelle stratégie de recherche a été appliquée ?



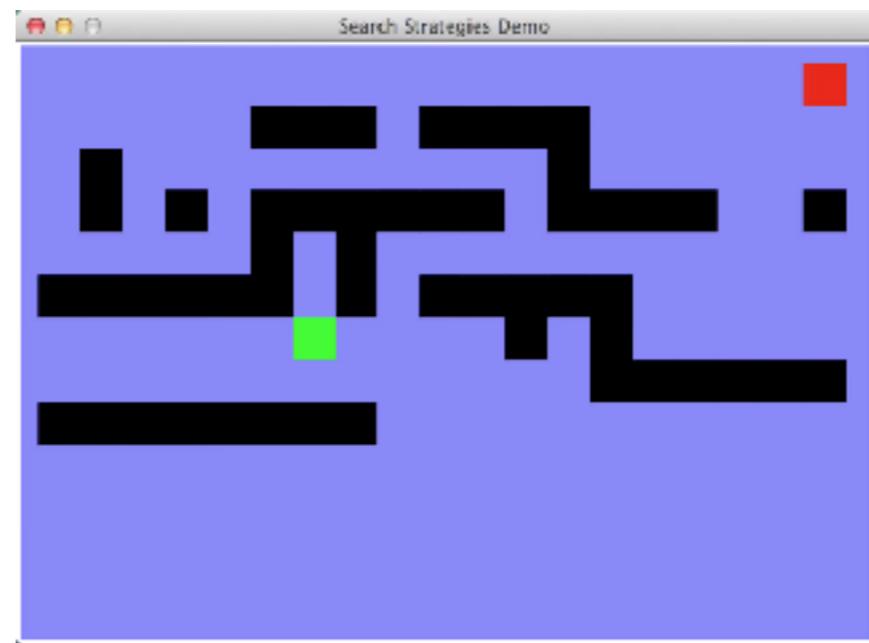
Recherche en largeur



Recherche en profondeur



Recherche en largeur



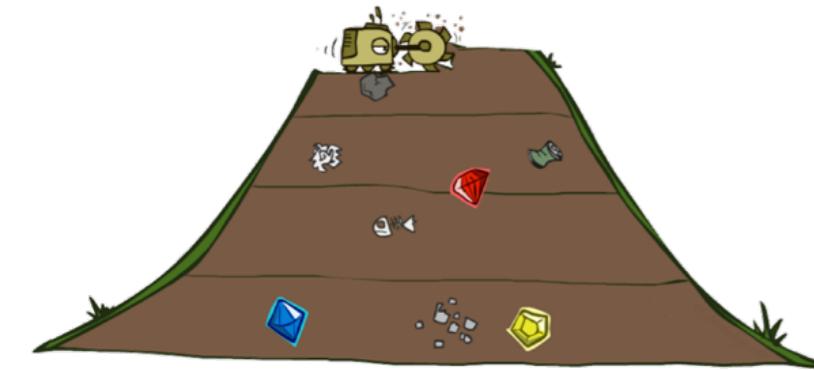
Recherche en profondeur

DFS vs BFS: comparaison



Quand une alternative est-elle préférable à une autre ?

Recherche en largeur



Lorsqu'il est important d'obtenir la solution optimale

Lorsqu'obtenir une solution ne requiert que peu d'actions

Lorsqu'on a une très bonne capacité mémoire

Recherche en profondeur



Lorsque l'arbre est borné (p.e. via un moyen d'enlever les cycles)

Lorsqu'avoir la meilleure solution n'est pas important

Lorsque la consommation mémoire doit être limitée

Souvent, en plan B face à un BFS trop coûteux à faire en pratique



Peut-on combiner ces deux approches ?

Souhait 1: complétude et optimalité de la recherche en largeur

Souhait 2: consommation mémoire similaire à une recherche en profondeur



Iterative deepening search - IDS

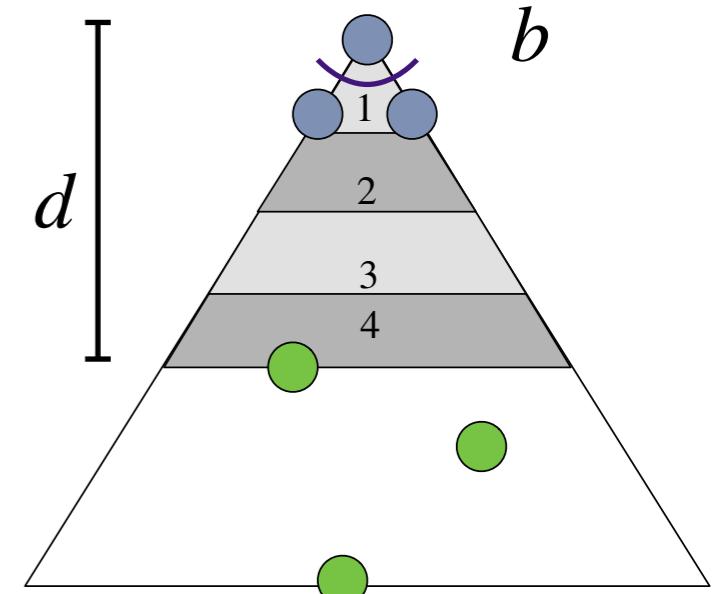
Intuition

Combiner les avantages d'un DFS avec un BFS

Exécuter un DFS, mais en fixant la profondeur maximale

Aucune solution: relancer un DFS jusqu'à un niveau en dessous

Solution trouvée: stopper la recherche et retourner la solution



Algorithme

```
IterativeDeepeningSearch( $P$ ) :
```

```
    for  $d \in 1$  to  $\infty$  :
```

```
         $s = \text{DepthLimitedSearch}(P, d)$ 
```

```
        if  $s \neq \emptyset$  : return  $s$ 
```

```
DepthLimitedSearch( $P, d$ ) :
```

```
     $s = \text{initialState}(P)$ 
```

```
     $L = \text{LIFO}()$ 
```

```
    push( $L, s$ )
```

```
    while  $L \neq \emptyset$  :
```

```
         $s = \text{pop}(L)$ 
```

```
        if  $s = \text{goalState}(P)$  : return solution
```

```
        else :
```

```
             $C = \{c \in \text{successors}(s, P) \mid \text{depth}(c) \leq d\}$ 
```

```
            push( $L, C$ )
```

```
    return  $\emptyset$ 
```

Performances

Complexité spatiale : $\mathcal{O}(bd)$ (héritage du DFS)

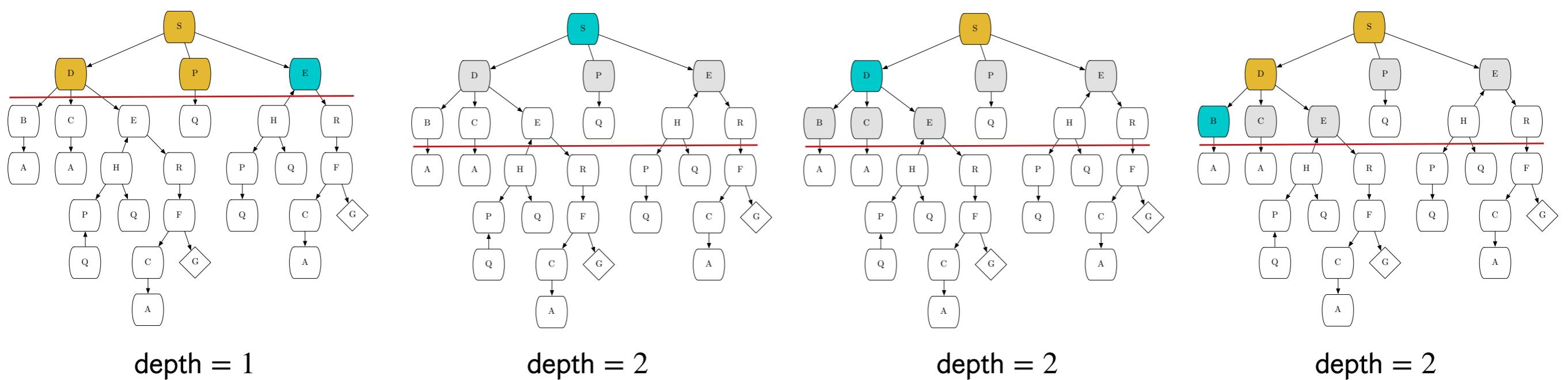
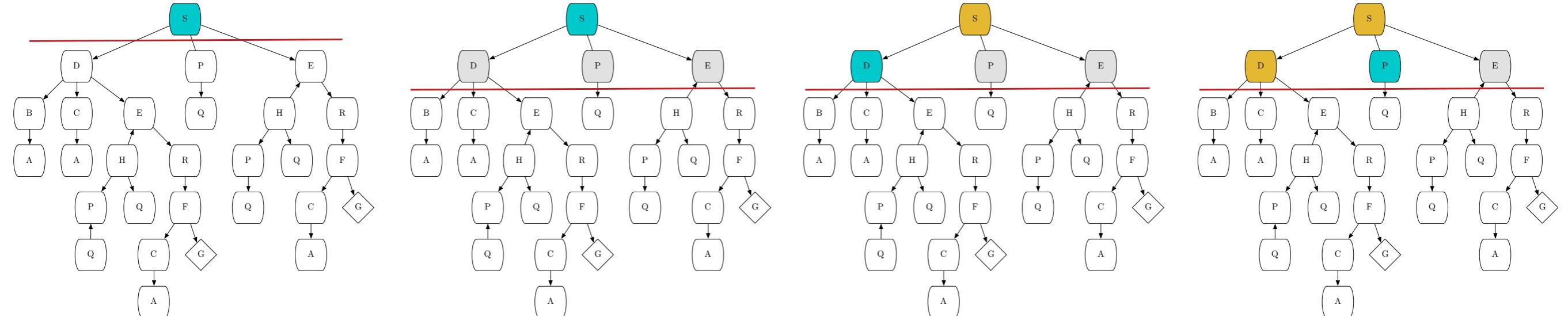
Complexité temporelle : $\mathcal{O}(b^d)$ + travail redondant

Recherche complète

Optimalité si tous les coûts sont identiques

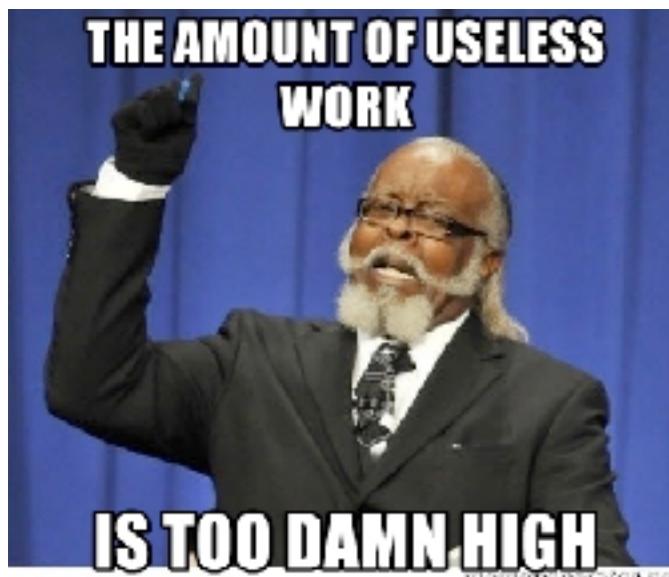
Essentiellement un DFS borné à la profondeur d

Iterative deepening search - illustration



Iterative deepening search - analyse

Redondance de IDS



IDS redemande de ré-exploré des noeuds de l'arbre de recherche
La recherche souffre ainsi de redondance par rapport à DFS ou BFS
Particulièrement pour les noeuds proches de la racine
Le premier niveau doit être exploré d fois, le deuxième $d-1$ fois, etc.

#noeuds explorés : $db + (d - 1)b^2 + (d - 2)b^3 + \dots + 2b^{d-1} + b^d$

?

Est-ce un gros problème ?

C'est exact qu'une certaine quantité de travail redondant est nécessaire

Cependant, ce travail redondant est *relativement* négligeable

Considérons une situation avec $b = 10$ et $d = 5$

Recherche en largeur

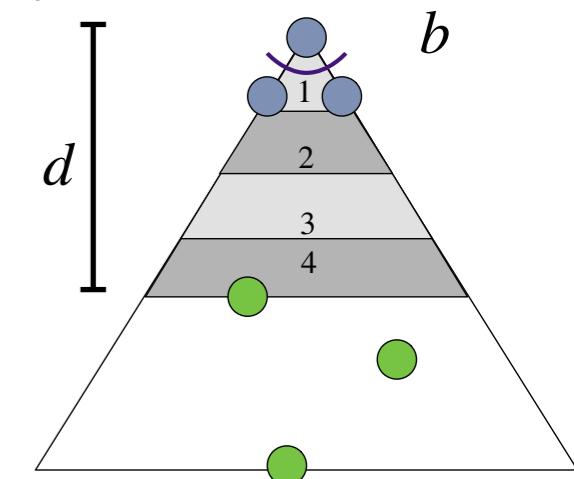
#noeuds : $b + b^2 + b^3 + b^4 + b^5$

$$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

Seulement environ 11% de noeuds supplémentaires

Cette différence s'atténue d'autant plus que la profondeur ou le facteur de branchement est grand

En pratique, IDS est une bonne stratégie, majoritairement préférable à un DFS ou un BFS simple



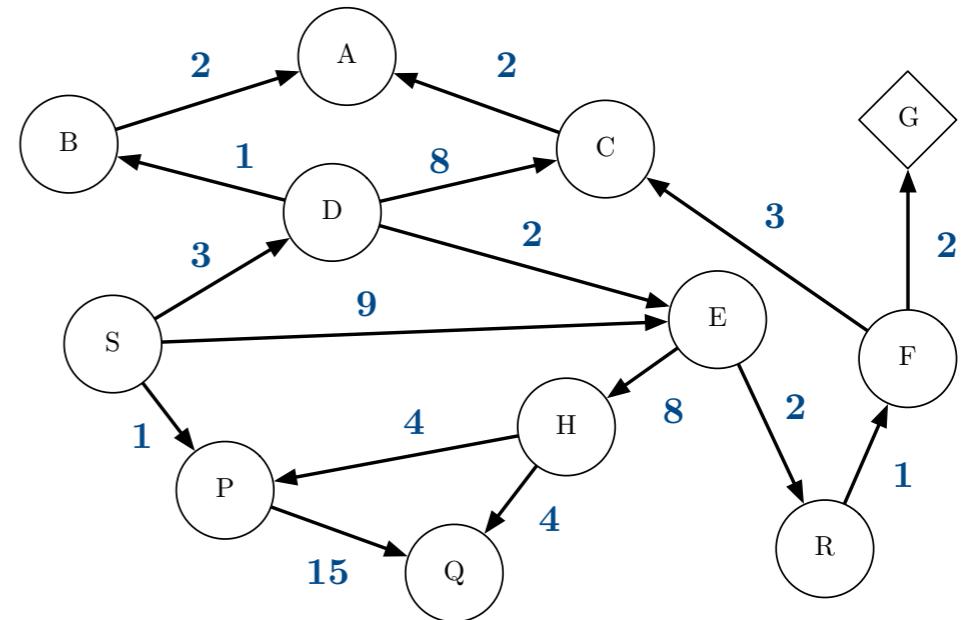
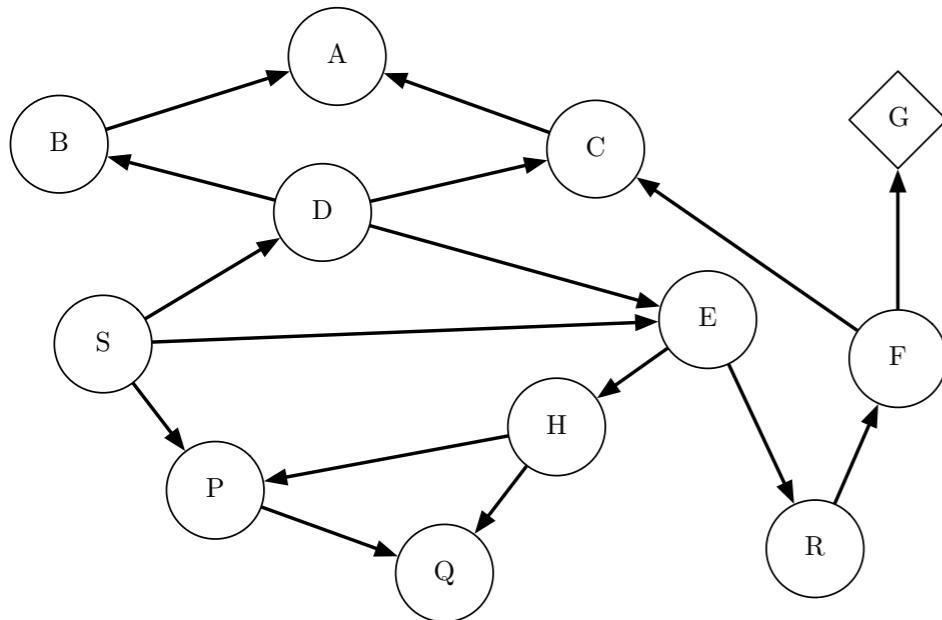
Iterative deepening search

#noeuds : $db + (d - 1)b^2 + (d - 2)b^3 + (d - 3)b^4 + b^5$

$$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

Problèmes de recherche avec coûts non identiques

Problème à coûts non identiques



Chaque action a un coût qui lui est propre (p.e., une distance entre deux noeuds)

La solution optimale est celle ayant le plus petit coût (somme des coûts des actions faites)

? Que va faire une recherche en largeur ?

BFS va retourner la solution la plus proche en termes de nombre d'actions

Cela ne correspond pas toujours à la solution optimale !

Dans cette situation, BFS (et notre version d'IDS) perdent leur comportement optimal

? Que peut-on faire ?

Recherche à coût uniforme (UCS - uniform cost search)

Recherche à coût uniforme

Retirer systématiquement le noeud ayant le plus faible coût

Le coût d'un noeud correspond à la somme des coûts depuis la racine

Intègre une fonction de coût $g(n) : \text{Node} \rightarrow \mathbb{R}$

Exemple

1 : [state : S , fringe : $\langle (D,3)^S, (P,1)^S, (E,9)^S \rangle$]

2 : [state : P^S , fringe : $\langle (D,3)^S, (E,9)^S, (Q,16)^P \rangle$]

3 : [state : D^S , fringe : $\langle (E,9)^S, (Q,16)^P, (B,4)^D, (C,11)^D, (E,5)^D \rangle$]

4 : [state : B^D , fringe : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (E,5)^D, (A,6)^B \rangle$]

5 : [state : E^D , fringe : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (A,6)^B, (H,13)^E, (R,7)^E \rangle$]

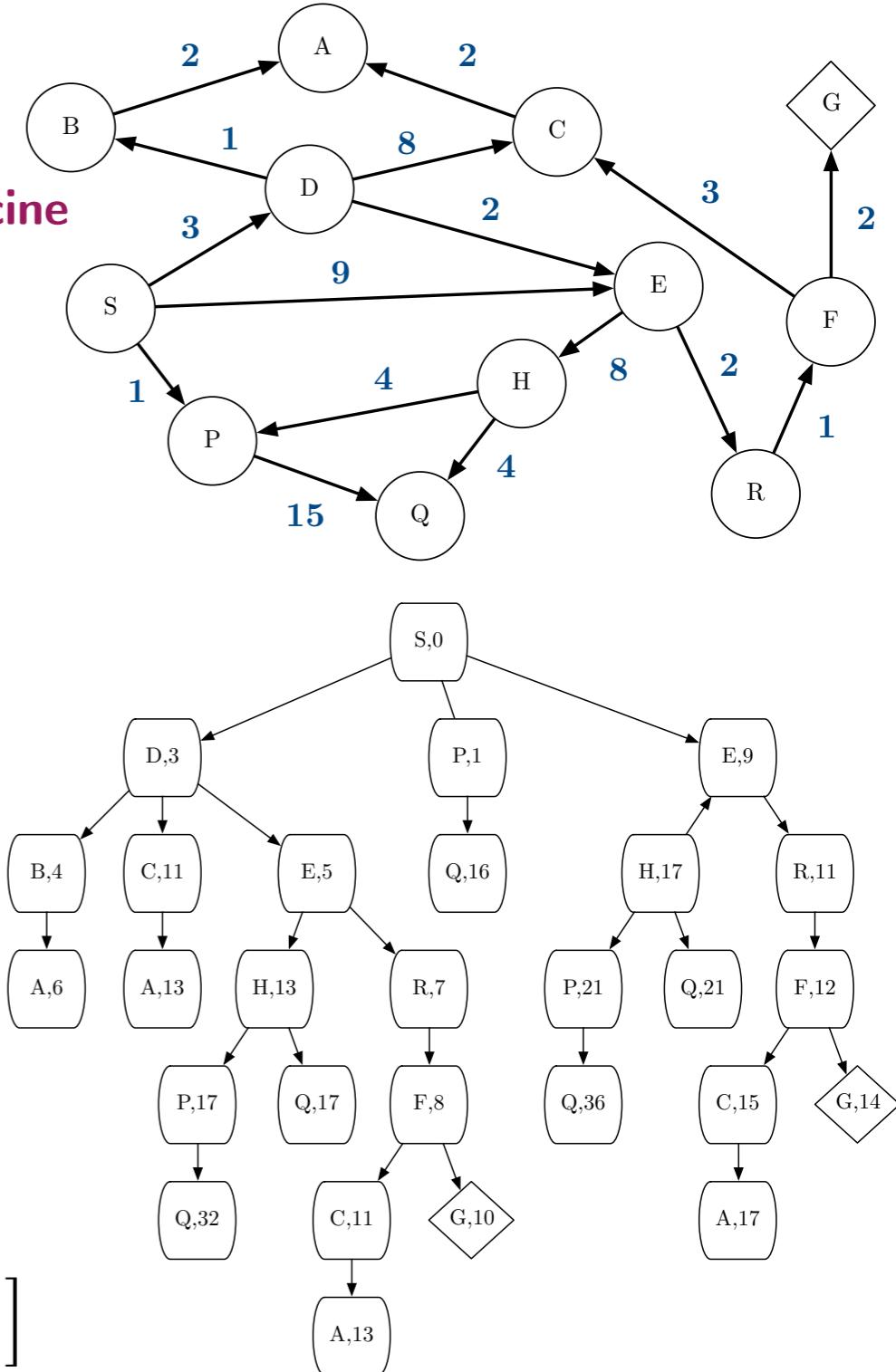
6 : [state : A^B , fringe : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (H,13)^E, (R,7)^E \rangle$]

7 : [state : R^E , fringe : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (H,13)^E, (F,8)^R \rangle$]

8 : [state : F^R , fringe : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (H,13)^E, (G,10)^F \rangle$]

9 : [state : E^S , fringe : $\langle (Q,16)^P, (C,11)^D, (H,13)^E, (G,10)^F, (H,17)^E, (R,11)^E \rangle$]

10 : [state : G^F , path : $S \rightarrow D \rightarrow E \rightarrow R \rightarrow F \rightarrow G$, cost : 10]



Les coûts sont calculés qu'une fois que le noeud est dans la frontière

UCS - Piège classique et algorithme

Considération pratique

Etape 8 : $\left[\text{state} : F^R, \text{fringe} : \langle (E,9), (Q,16), (C,11)^D, (C,11)^F, (H,13), (G,10) \rangle \right]$



Peut-on arrêter la recherche, vu qu'un état final est dans la Fringe ?

Même si l'état final (**G**) est dans la *fringe*, il ne doit pas être étendu en priorité

Rien ne nous assure que via E, il n'existe pas un chemin de plus faible coût

La recherche est finie QUE si vous êtes à l'état final (et non s'il est dans la Fringe)



Algorithme

Implémentation de la *fringe* par une liste de priorité

Chaque noeud doit contenir en plus l'information de son coût

UniformCostSearch(P) :

$s = \text{initialState}(P)$

$L = \text{PriorityQueue}()$

$\text{push}(L, s, g(s))$

while $L \neq \emptyset$:

$s = \text{pop}(L)$

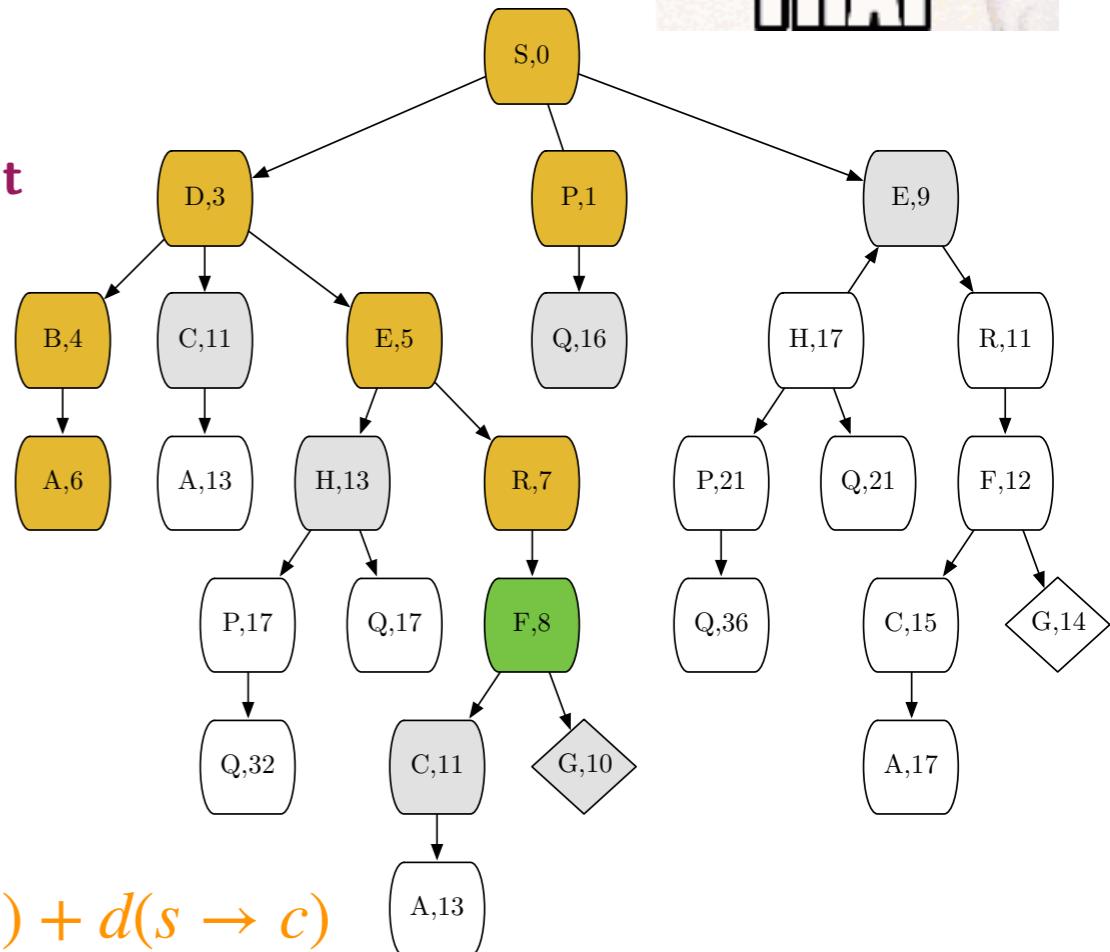
if $s = \text{goalState}(P)$: return solution

else :

$C = \{(c, g(c)) \in \text{succesors}(s, P)\}$

$\text{push}(L, C)$

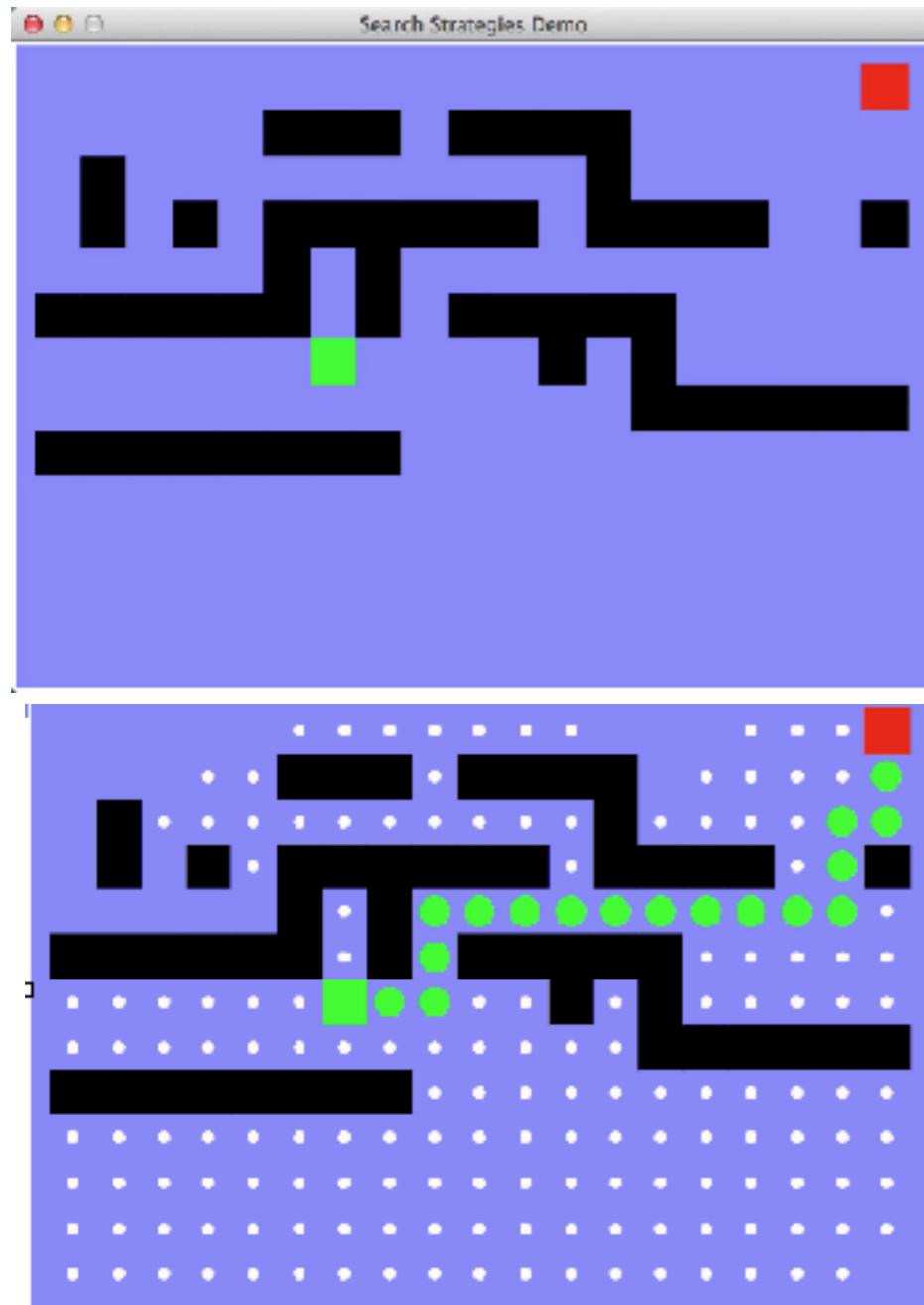
return no solution



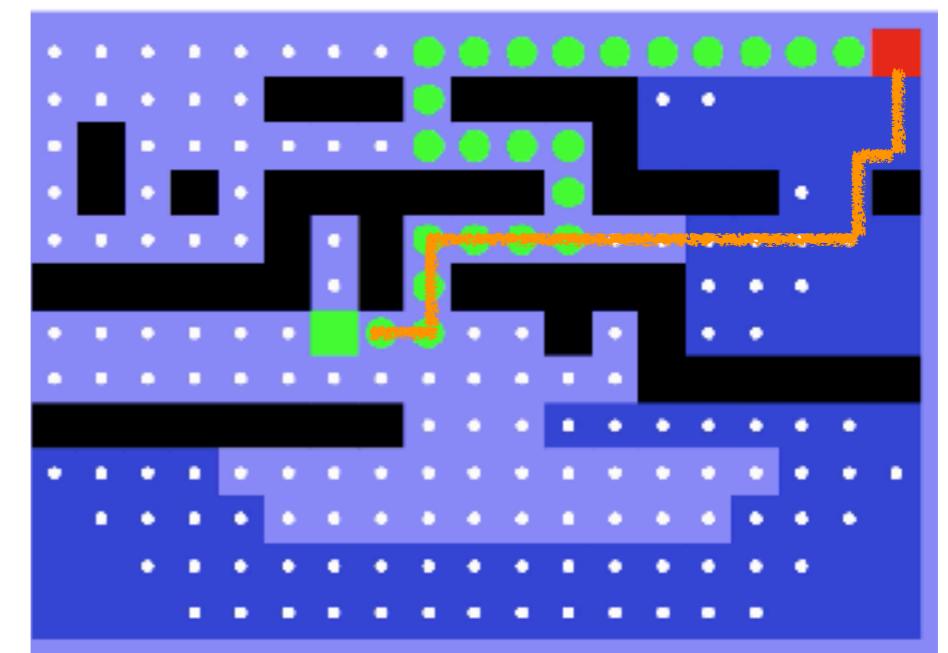
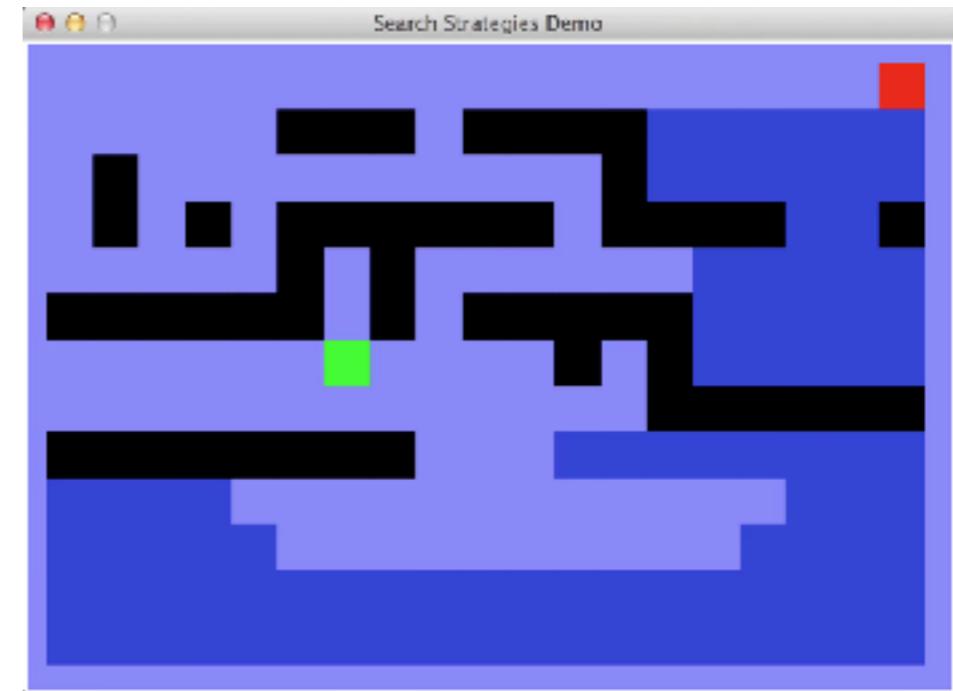
coût de c = coût de s + coût de la transition $s \rightarrow c$

UCS - illustration

Illustration



Tous les coûts sont identiques
Equivalent à une recherche en largeur



La zone bleue indique un coût supérieur
Retourne la solution optimale
Contrairement à une recherche en largeur

Recherche à coût uniforme - analyse des performances

Paramètres

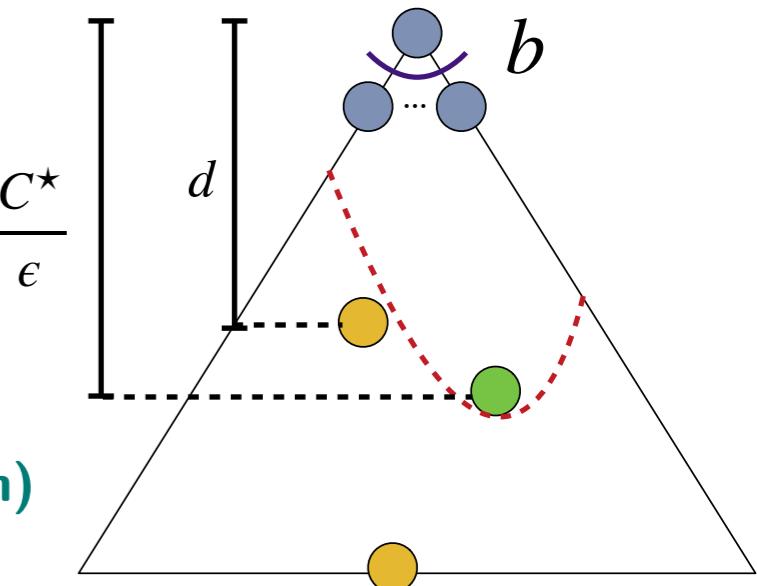
b : nombre maximum de successeurs (branching factor)

d : profondeur de la solution la plus proche

C^* : coût de la solution optimale

ϵ : borne inférieure sur le coût d'une action

$\frac{C^*}{\epsilon}$: profondeur de la solution optimale dans le pire des cas (effective depth)



Complexité temporelle

Toutes les solutions se situant avant la profondeur effective doivent être explorées

Complexité temporelle : $\mathcal{O}(b^{C^*/\epsilon})$ (très coûteux, potentiellement plus que $\mathcal{O}(b^d)$)

Complexité spatiale

Similaire à BFS, mais sur base de la profondeur effective (tous les noeuds du dernier niveau)

Complexité spatiale: $\mathcal{O}(b^{C^*/\epsilon})$

Complétude

La recherche est complète (sous l'hypothèse que les coûts sont positifs et que la solution a un coût fini)

Optimalité

La recherche est optimale (même argument que pour la recherche en largeur)

Recherche à coût uniforme



Algorithme de Dijkstra

Logiquement équivalent à UCS (même ordre d'extension)

Nom utilisé par la communauté des sciences informatiques

Principalement utilisé pour référer à des problèmes de graphe

Réseaux routiers, réseaux de télécommunications, etc.

Le graphe en entrée est le problème à résoudre

La taille du graphe est souvent raisonnable

Recherche à coût uniforme

Nom utilisé par la communauté de l'intelligence artificielle

Principalement utilisé pour référer à des problèmes de recherche

Recherche en graphe et recherche en arbre

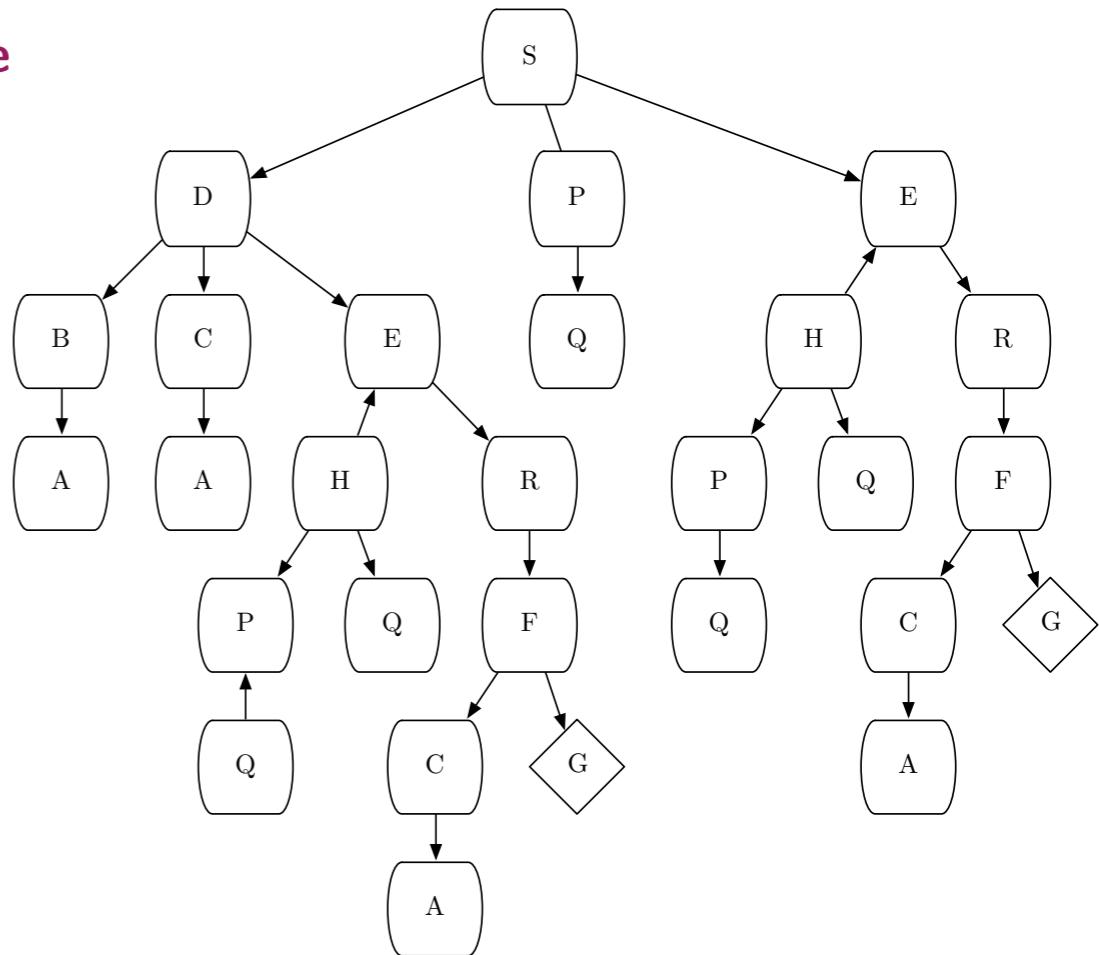
La taille du graphe/arbre croît exponentiellement avec le nombre d'états (taille éventuellement infinie)

En fonction de l'implémentation, quelques différences peuvent exister

Table des matières

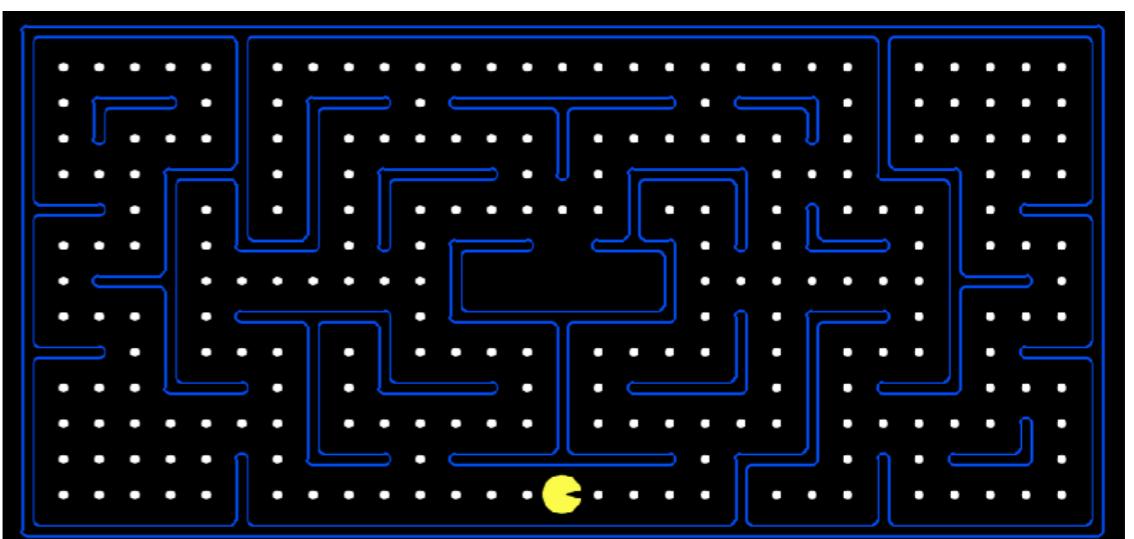
Stratégies de recherche

- ✓ 1. Définition et modélisation d'un problème de recherche
- ✓ 2. Agents réflexes
- ✓ 3. Agents axés sur la recherche
- ✓ 4. Recherche en arbre (*tree search*)
- ✓ 5. Recherche sans information: DFS, BFS, UCS, IDS
- 6. Recherche avec information: *greedy search, A**
- 7. Conception d'heuristiques
- 8. Recherche en graphe (*graph search*)



Problèmes abordés

- 1. *Pacman*
- 2. *8-puzzle*
- 3. Planification de routes



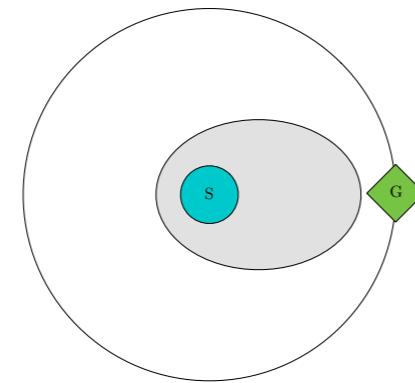
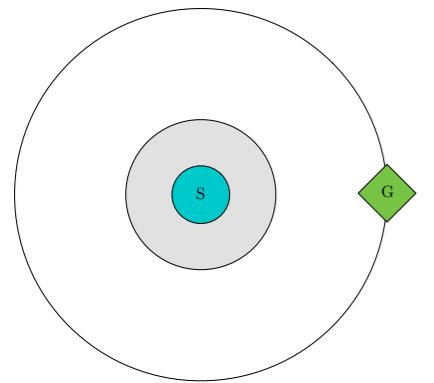
Motivation des stratégies de recherche avec information

Limitation de la stratégie à coût uniforme

Bien qu'elle soit optimale, cette recherche n'utilise aucune information spécifique au problème à résoudre

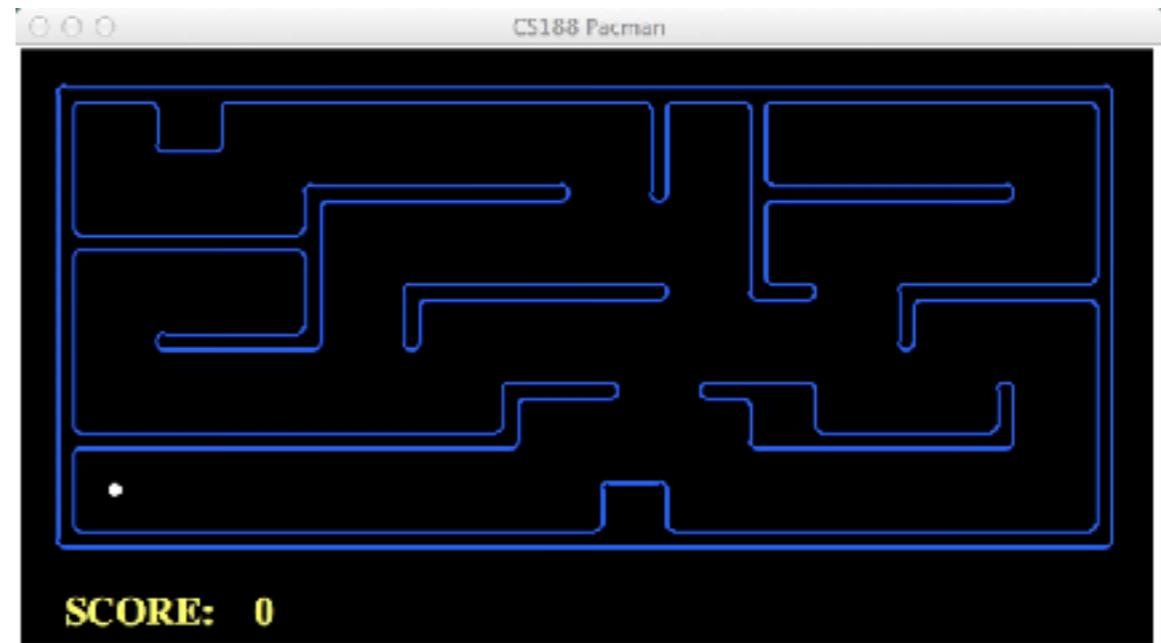
L'exploration se fait de manière symétrique (sur base des coûts) par rapport à l'état initial

On va autant explorer dans toutes les directions avant d'atteindre l'objectif



?

Comment régler ce problème ?



Stratégie de recherche avec information (*informed search*)



Stratégie qui intègre des connaissances que l'on a sur le problème

Permet d'orienter la recherche vers des états que l'on pense être de bonne qualité

De Montréal à Québec, est-ce plus probable de passer par Trois-Rivières ou Toronto ?

Stratégie de recherche avec information (*informed search*)



Stratégie de recherche avec information

Stratégie de recherche qui utilise des connaissances spécifiques au problème afin d'orienter la recherche vers des états qui nous paraissent plus prometteurs



Quels sont les points de conception majeur de cette famille de méthode ?

Points de conception

(1) Le type de connaissance à injecter à la recherche

Nature de la connaissance à injecter

Validité et propriétés de la connaissance

Complexité temporelle pour obtenir cette connaissance

(2) La façon d'injecter cette connaissance dans la stratégie de recherche

Degré de confiance que l'on veut avoir dans la connaissance injectée

Propriétés souhaitées pour la stratégie (complexité spatiale, temporelle, complétude, optimalité)



On rentre vraiment dans de l'intelligence artificielle,
en insérant une forme d'intelligence à nos stratégies de recherche !

Fonction heuristique



Fonction heuristique

Intuition que l'on a sur le coût nécessaire pour atteindre un état final à partir d'un certain état

$h(n)$ = coût estimé pour atteindre un état final à partir d'un état n

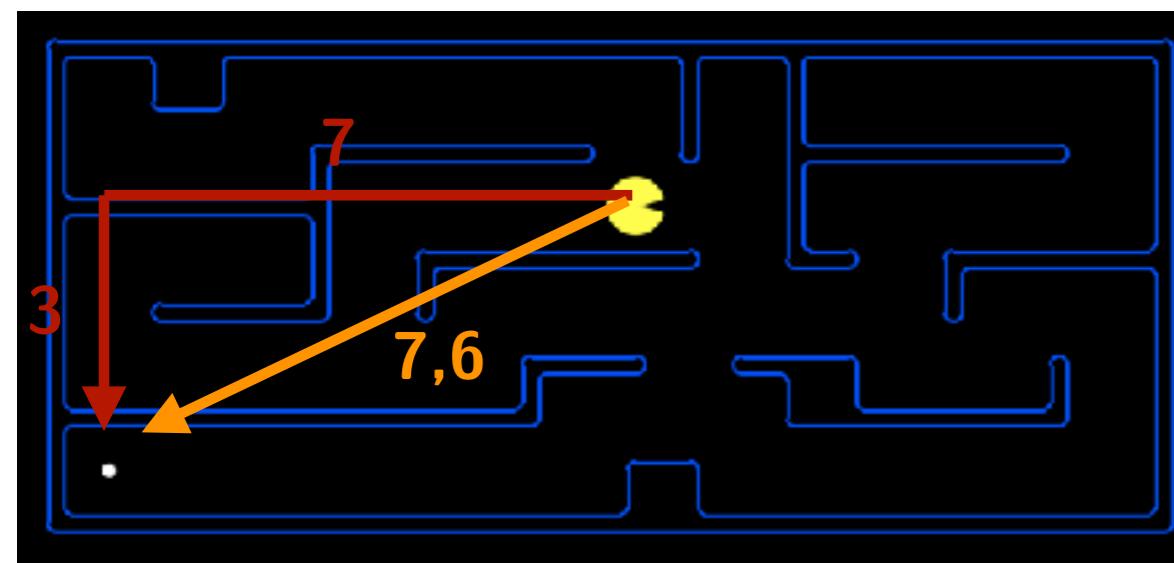
Une heuristique donne ainsi une information sur à quel point un état est coûteux

Intuitivement, on préfèrera privilégier l'exploration des états dont l'heuristique retourne une faible valeur

Elles sont essentiellement conçues pour un problème particulier (ou une classe spécifique de problèmes)



Avez-vous des idées d'heuristiques pour ce problème ?



Exemples d'heuristique

Distance euclidienne: chemin à vol d'oiseau

Distance de Manhattan: chemin le plus court sur une grille



Quelle heuristique vous semble de meilleure qualité ?

Intuitivement, la distance de Manhattan, car elle donne une idée plus précise du coût de chaque état

Recherche gloutonne (*greedy best-first search*)

Recherche gloutonne

Chaque état est évalué selon la fonction heuristique

Revient à retirer le noeud le moins coûteux l'heuristique

Implémentation identique à UCS (seulement la priorité change)

Complexité temporelle

Dans le pire des cas, agit comme un mauvais DFS

Complexité temporelle : $\mathcal{O}(b^m)$

Complexité spatiale

Dans le pire des cas, agit comme un mauvais BFS

Complexité spatiale : $\mathcal{O}(b^m)$

Complétude

Recherche non-complète (même problème que le DFS)

Optimalité

Très haut risque de tomber sur une solution sous-optimale

Commentaires généraux

Le manque de garanties théoriques est le problème principal de cette stratégie de recherche

Malgré de faibles garanties, les performances peuvent être très bonnes si l'heuristique est adaptée

GreedySearch(P, h) :

$s = \text{initialState}(P)$

$L = \text{PriorityQueue}()$

$\text{push}(L, s, h(s))$

while $L \neq \emptyset$:

$s = \text{pop}(L)$

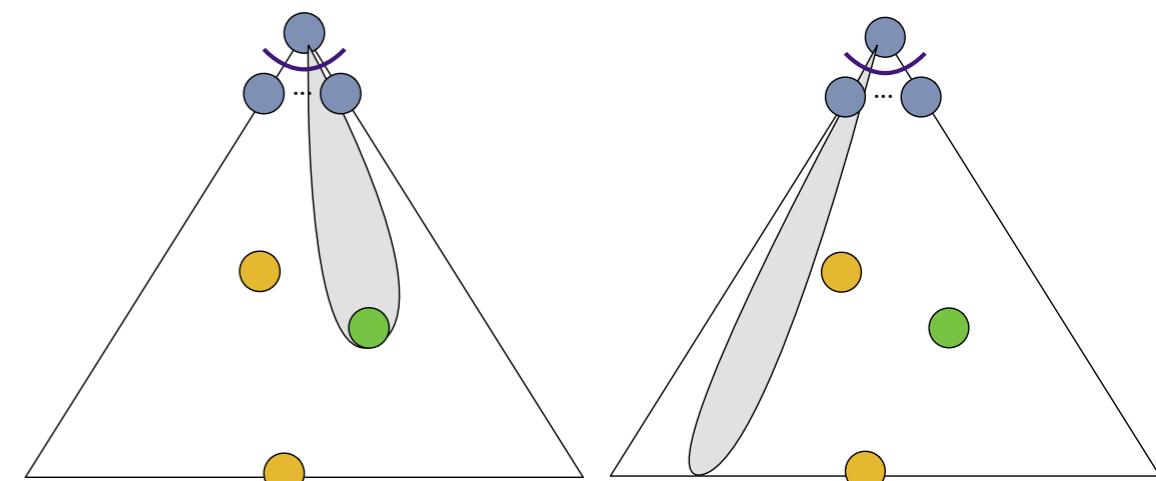
if $s = \text{goalState}(P)$: return solution

else :

$C = \{\langle c, h(c) \rangle \in \text{succesors}(s, P)\}$

$\text{push}(L, C)$

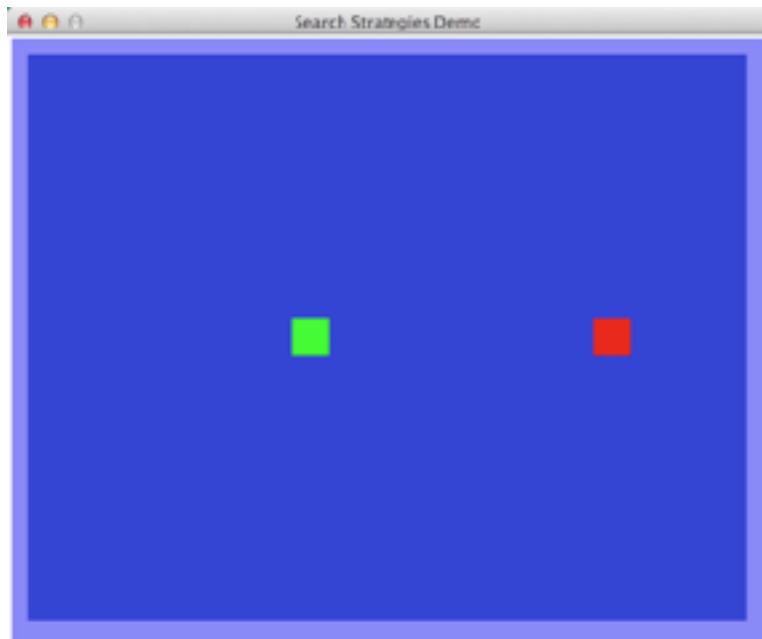
return no solution



Recherche gloutonne - exemples



Que va faire une recherche gloutonne dans ces situations ?



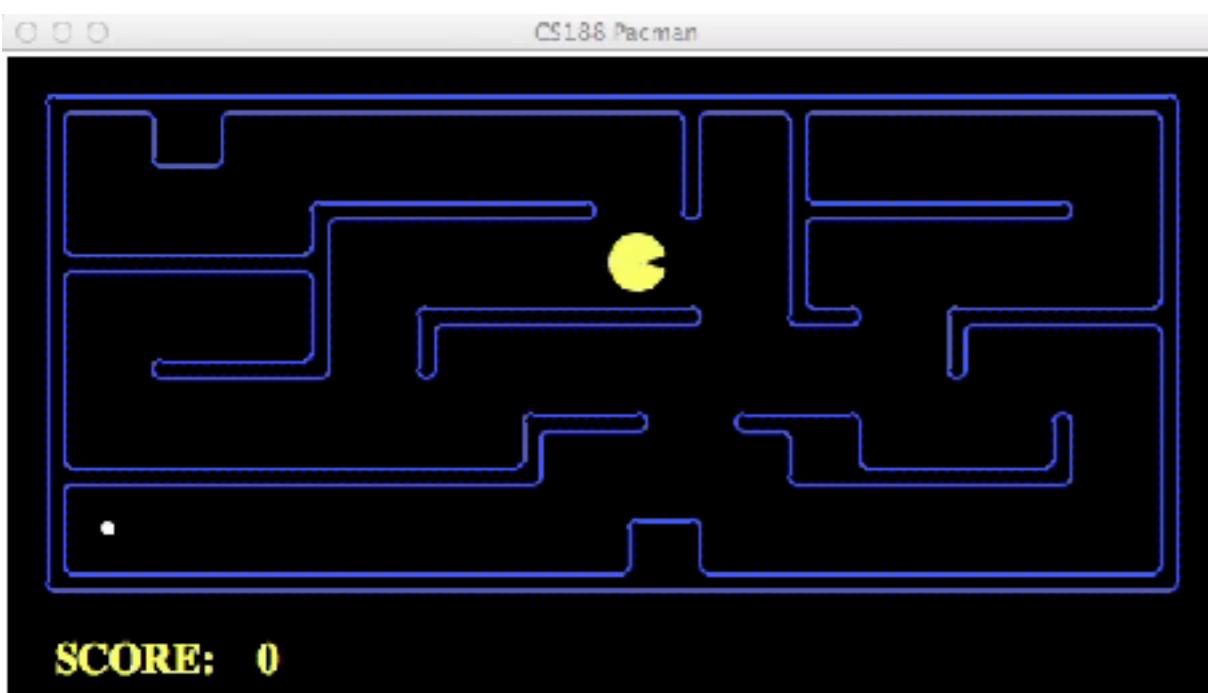
Ca dépend de l'heuristique qui a été choisie !



*"La fonction heuristique,
vous définirez"*

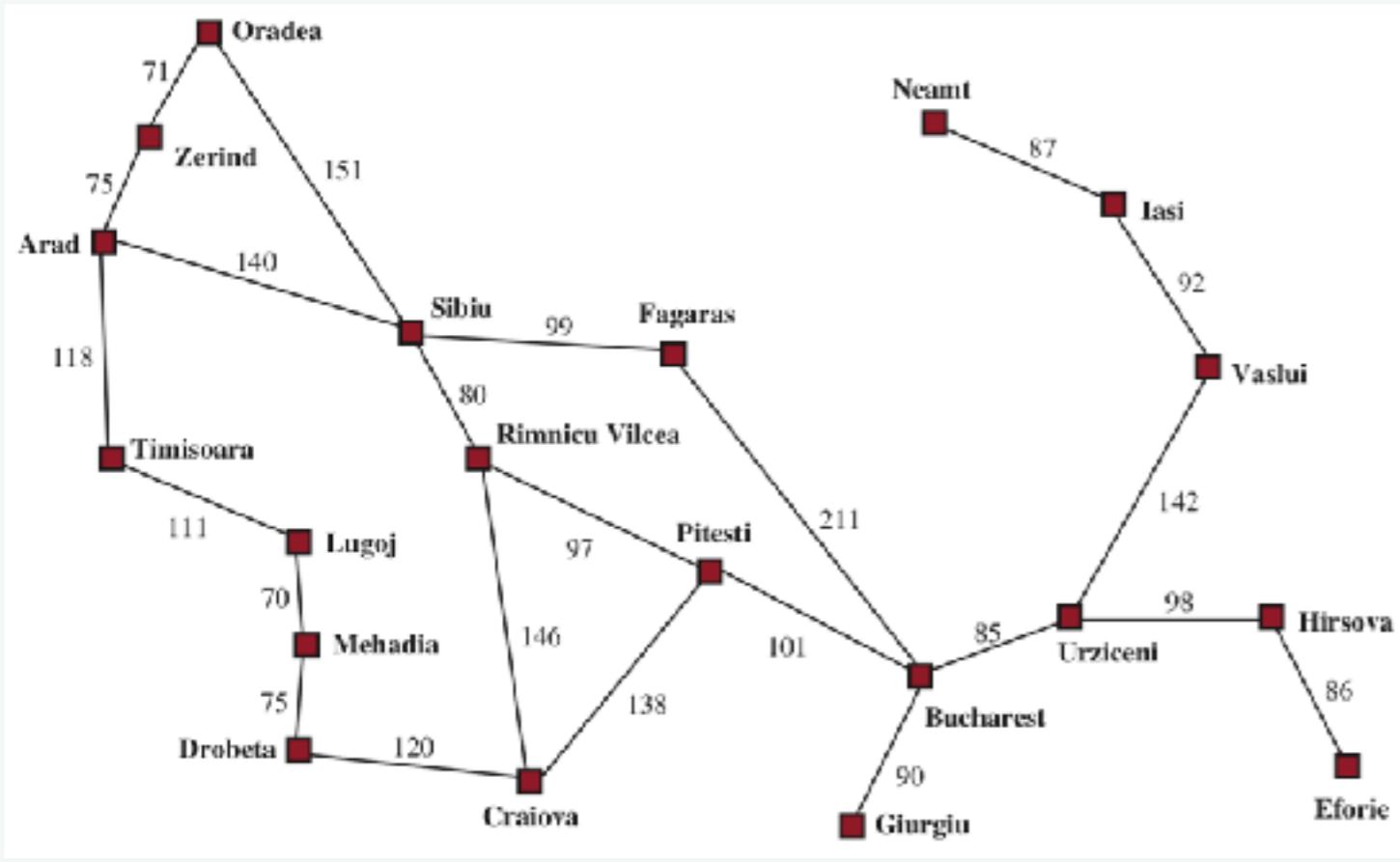
Sans définir l'heuristique, cette question n'a aucun sens

Heuristique: distance de Manhattan entre l'état actuel et l'état final



On remarque le caractère non optimal de cette méthode

Recherche gloutonne - exemple



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Valeur heuristique (vol d'oiseau) de chaque état

Objectif: trouver le chemin le plus court pour aller de Arad à Bucharest



Avez-vous des idées d'heuristiques pour ce problème ?

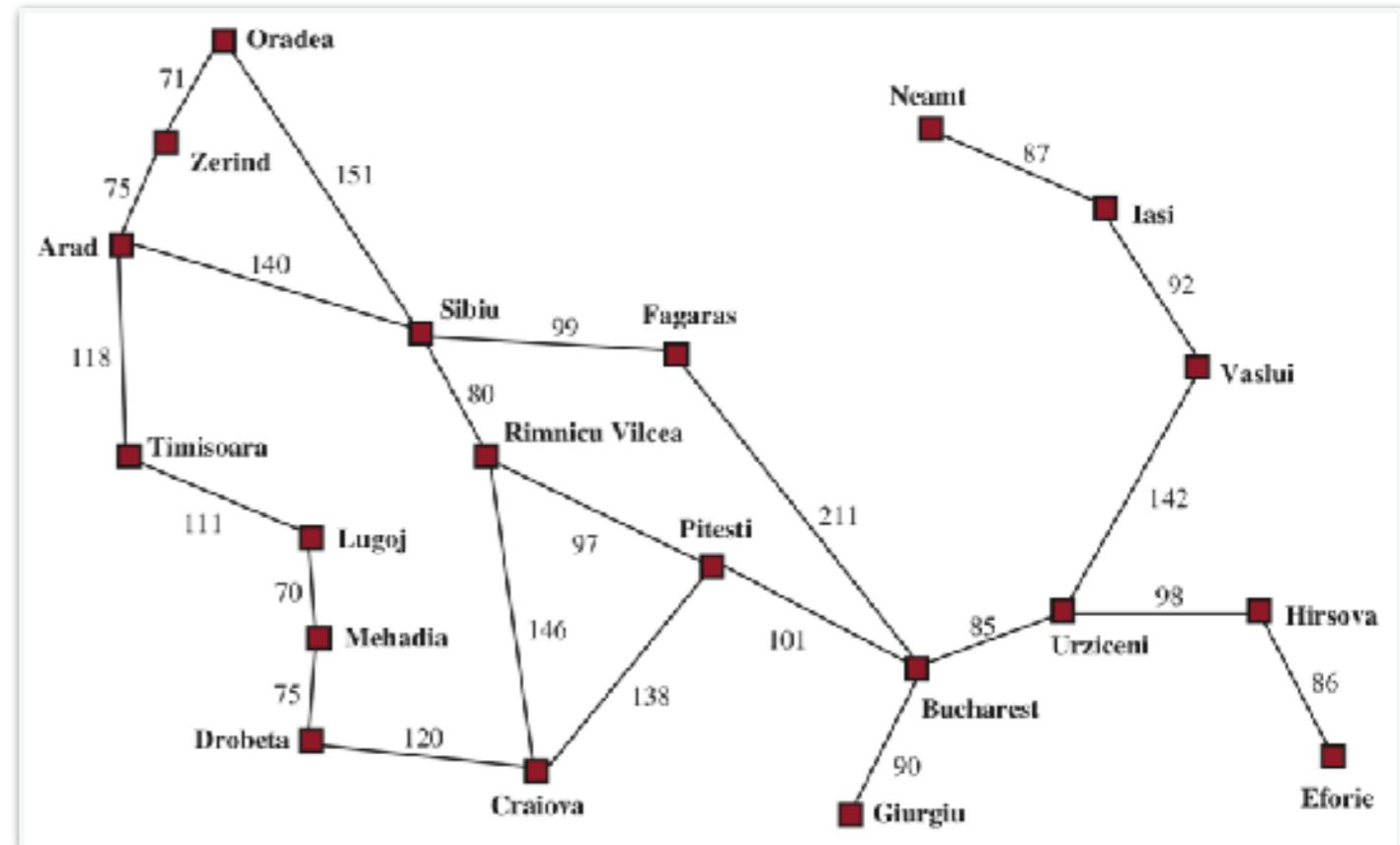
Heuristique: la distance à vol d'oiseau semble être un choix raisonnable

Une recherche gloutonne va ainsi sélectionner le successeur ayant la plus faible coût estimé

Recherche gloutonne - exécution de l'exemple

Fonction heuristique

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



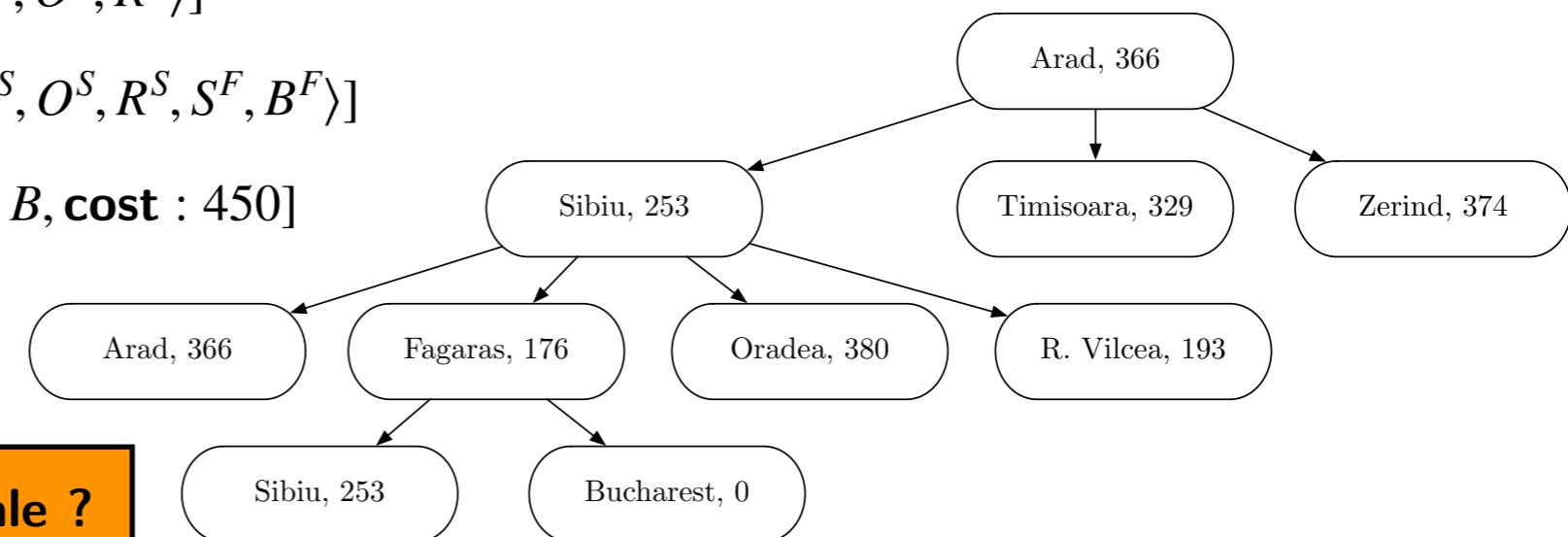
Exécution

Etape 1 : [state : A , fringe : $\langle S^A, T^A, Z^A \rangle$]

Etape 2 : [state : S^A , fringe : $\langle T^A, Z^A, A^S, F^S, O^S, R^S \rangle$]

Etape 3 : [state : F^S , fringe : $\langle T^A, Z^A, A^S, F^S, O^S, R^S, S^F, B^F \rangle$]

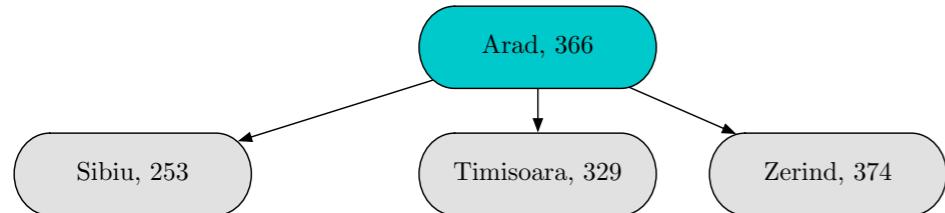
Etape 4 : [state : B^F , path : $A \rightarrow S \rightarrow F \rightarrow B$, cost : 450]



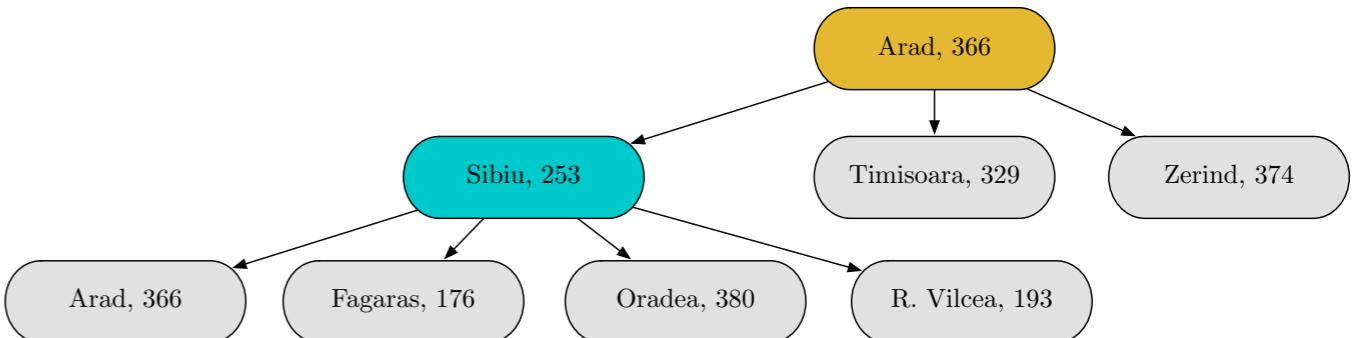
Est-ce que notre solution est optimale ?

Recherche gloutonne - exemple

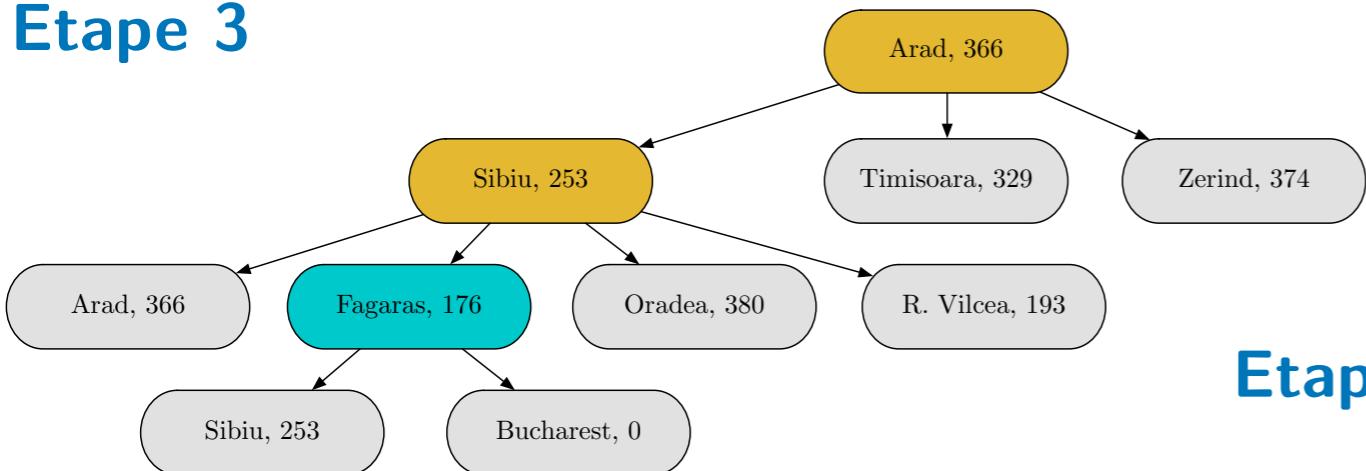
Etape 1



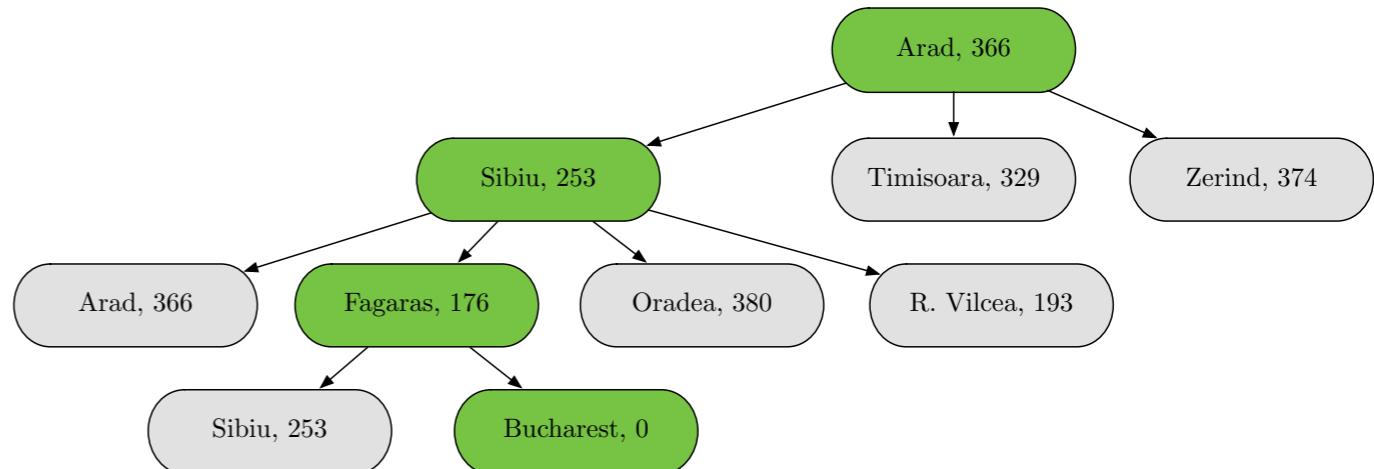
Etape 2



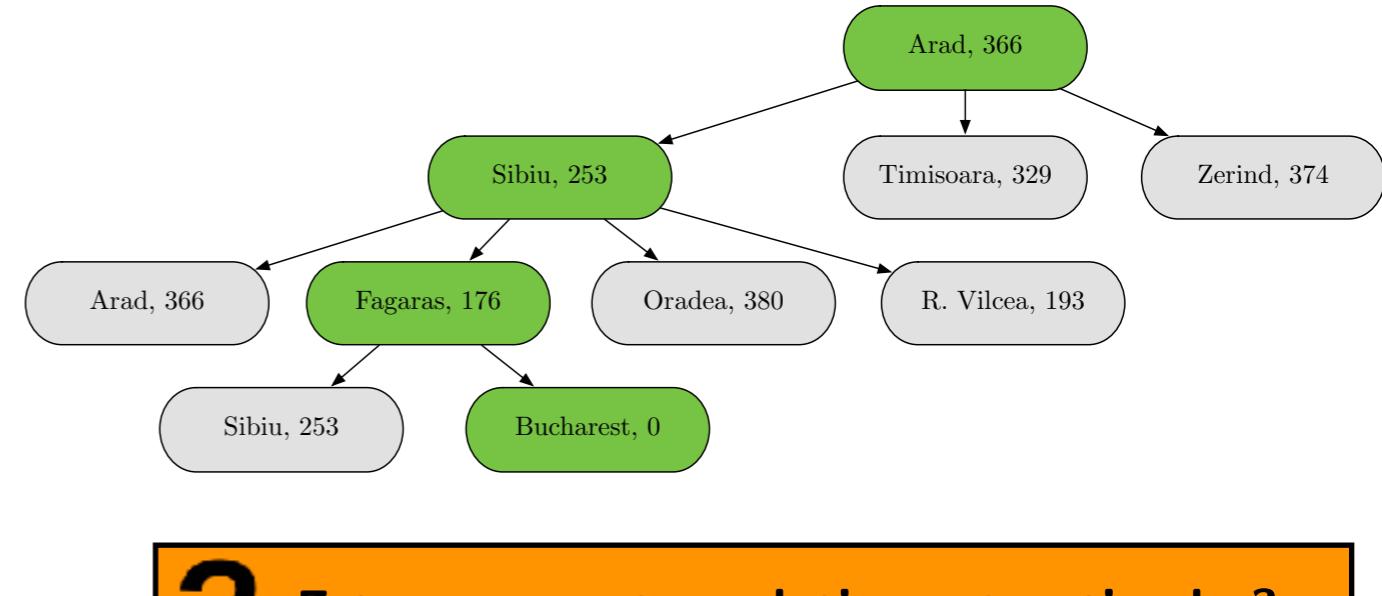
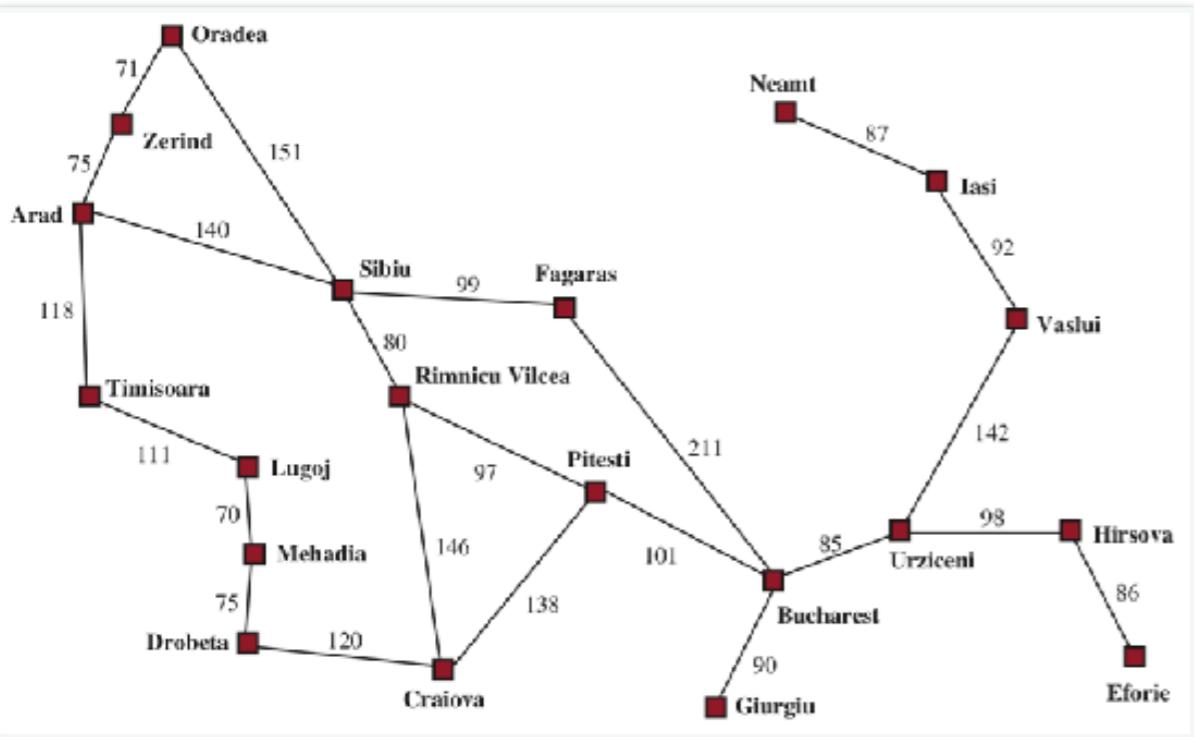
Etape 3



Etape 4



Recherche gloutonne - analyse de la solution



Est-ce que notre solution est optimale ?

Coût de notre solution

$$g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest}) = 140 + 99 + 211 = 450$$

Coût d'une autre solution

$$g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{R. Vilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bucharest}) = 140 + 80 + 97 + 101 = 418$$



Pourquoi a t-on manqué cette meilleure solution ?

Notre heuristique a complètement négligé R. Vilcea au détriment de Fagaras

Or, il s'agit d'un choix raisonnable, car le coût déjà effectué depuis Arad est moindre

$$g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras}) = 140 + 99 = 239$$

$$g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{R. Vilcea}) = 140 + 80 = 220$$

La recherche gloutonne ne tient pas compte des coûts déjà engendrés !

Motivation d'une nouvelle stratégie



Question de philosophie...

Question très difficile à résoudre, mais on peut suivre deux principes

- (1) Tenir compte de nos connaissances antérieures, en tant qu'expérience
- (2) Tenter d'avoir une vision du futur, en tant qu'intuition

Souvent nos décisions personnelles tiennent comptes de ces deux aspects

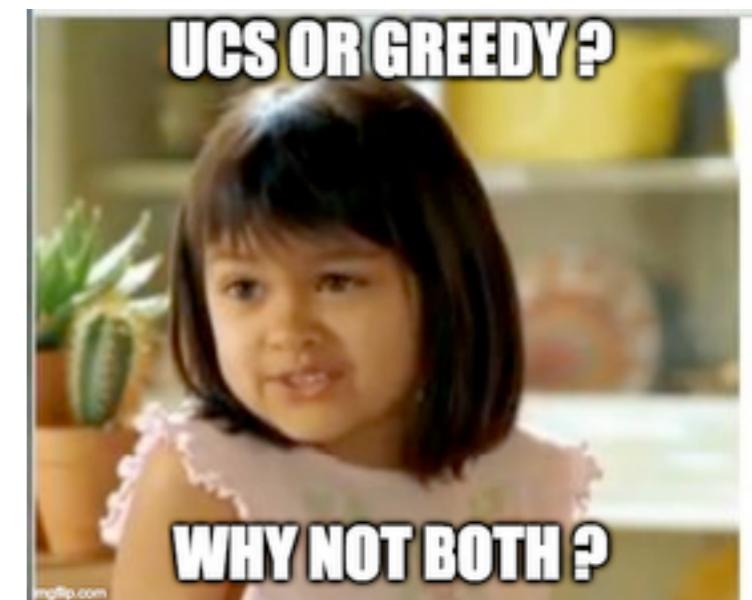
Conception d'une nouvelle stratégie de recherche

- (1) Une stratégie UCS tient uniquement des coûts déjà générés

Fiable (optimal) mais lent

- (2) Une stratégie gloutonne tient uniquement des coûts futurs estimés

Non-fiable mais peut accélérer si notre heuristique est bonne



L'idée est de créer une stratégie tenant à la fois compte du passé et du futur

La combinaison de ces deux aspects forme un des algorithmes de recherche les plus réputé en IA

Stratégie de recherche A*

Stratégie de recherche A*

Stratégie de recherche qui combine une fonction heuristique ainsi que le coût des noeuds

Fonction de sélection d'un état dans la Fringe : $f(s) = g(s) + h(s)$

$g(s)$ est le coût du noeud s (similaire à UCS)

$h(s)$ est la fonction heuristique appliquée au noeud s (estimation des coûts futurs)

Cette fonction est utilisé pour sélectionner le prochain noeud à explorer dans la file de priorité

```
AStarSearch( $P, h$ ) :  
     $s = \text{initialState}(P)$   
     $L = \text{PriorityQueue}()$   
     $f(s) = g(s) + h(s)$   
    push( $L, s, f(s)$ )  
    while  $L \neq \emptyset$  :  
         $s = \text{pop}(L)$   
        if  $s = \text{goalState}(P)$  : return solution  
        else :  
             $C = \{\langle c, g(c) + h(c) \rangle \mid c \in \text{succesors}(s, P)\}$   
            push( $L, C$ )  
    return no solution
```

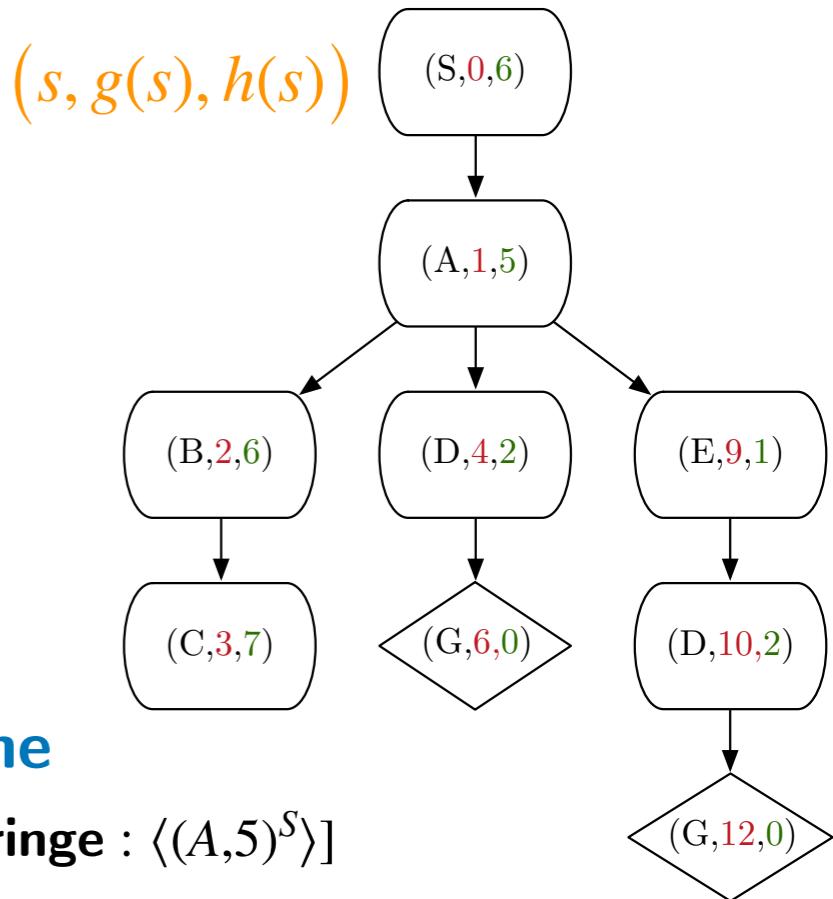
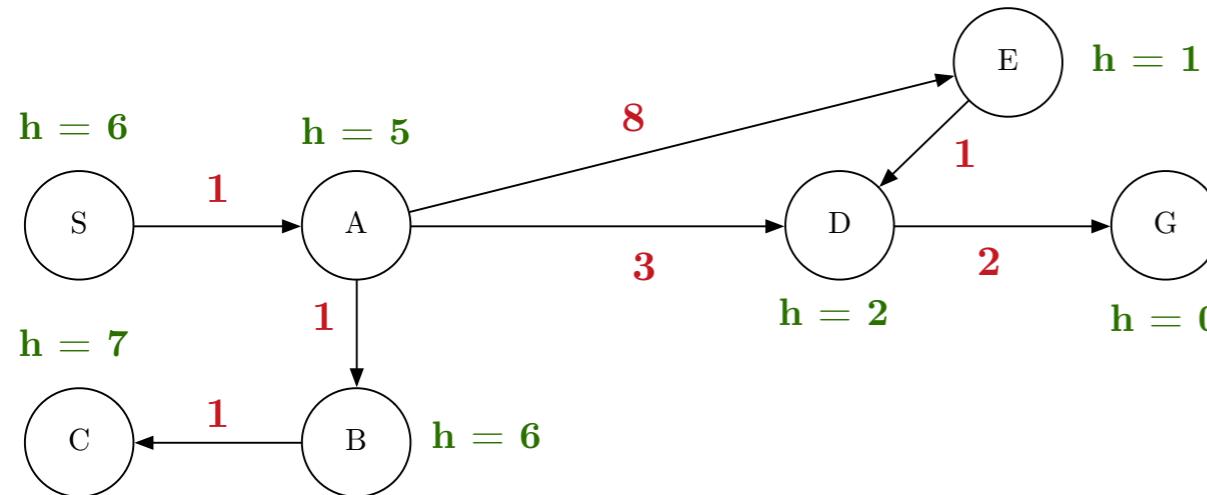


Pour le principe de l'algorithme, oui...
...mais il nous reste à étudier ses propriétés

Illustration - UCS vs Greedy vs A*

Illustration

Le coût de chaque noeud est évalué par une fonction heuristique



Recherche à coût uniforme

Etape 1 : [state : S , fringe : $\langle (A,1)^S \rangle$]

Etape 2 : [state : A^S , fringe : $\langle (B,2)^A, (D,4)^A, (E,9)^A \rangle$]

Etape 3 : [state : B^A , fringe : $\langle (D,4)^A, (E,9)^A, (C,3)^B \rangle$]

Etape 4 : [state : C^B , fringe : $\langle (D,4)^A, (E,9)^A \rangle$]

Etape 5 : [state : D^A , fringe : $\langle (E,9)^A, (G,6)^D \rangle$]

Etape 6 : [state : G^D , path : $S \rightarrow A \rightarrow D \rightarrow G$ (6)]

Recherche gloutonne

Etape 1 : [state : S , fringe : $\langle (A,5)^S \rangle$]

Etape 2 : [state : A^S , fringe : $\langle (B,6)^A, (D,2)^A, (E,1)^A \rangle$]

Etape 3 : [state : E^A , fringe : $\langle (B,6)^A, (D,2)^A, (D,2)^E \rangle$]

Etape 4 : [state : D^E , fringe : $\langle (B,6)^A, (D,2)^A, (G,0)^D \rangle$]

Etape 5 : [state : G^D , path : $S \rightarrow A \rightarrow E \rightarrow D \rightarrow G$ (12)]

Recherche A*

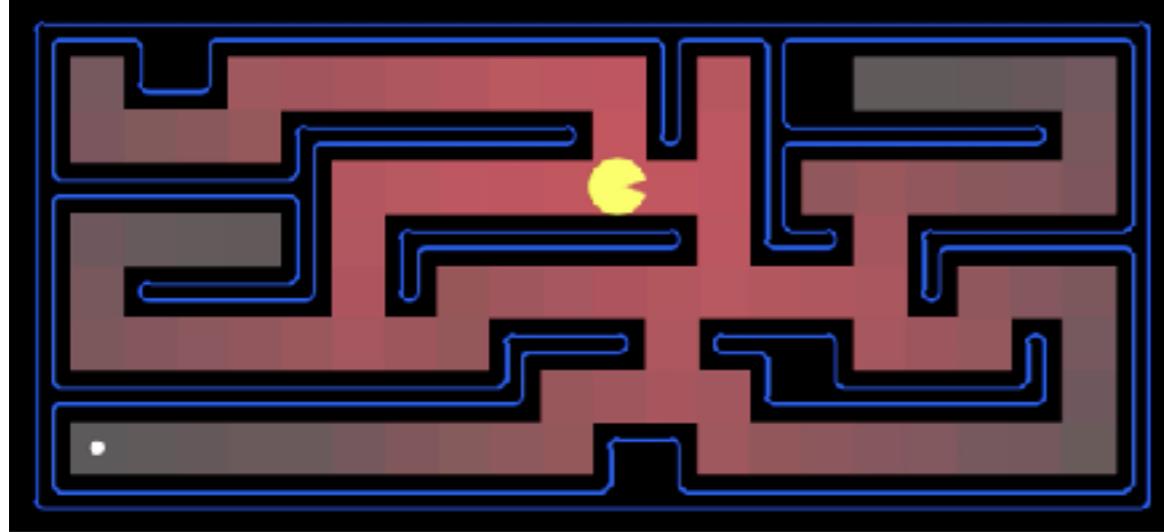
Etape 1 : [state : S , fringe : $\langle (A,6)^S \rangle$]

Etape 2 : [state : A , fringe : $\langle (B,8)^A, (D,6)^A, (E,10)^A \rangle$]

Etape 3 : [state : D^A , fringe : $\langle (B,8)^A, (E,10)^A, (G,6)^D \rangle$]

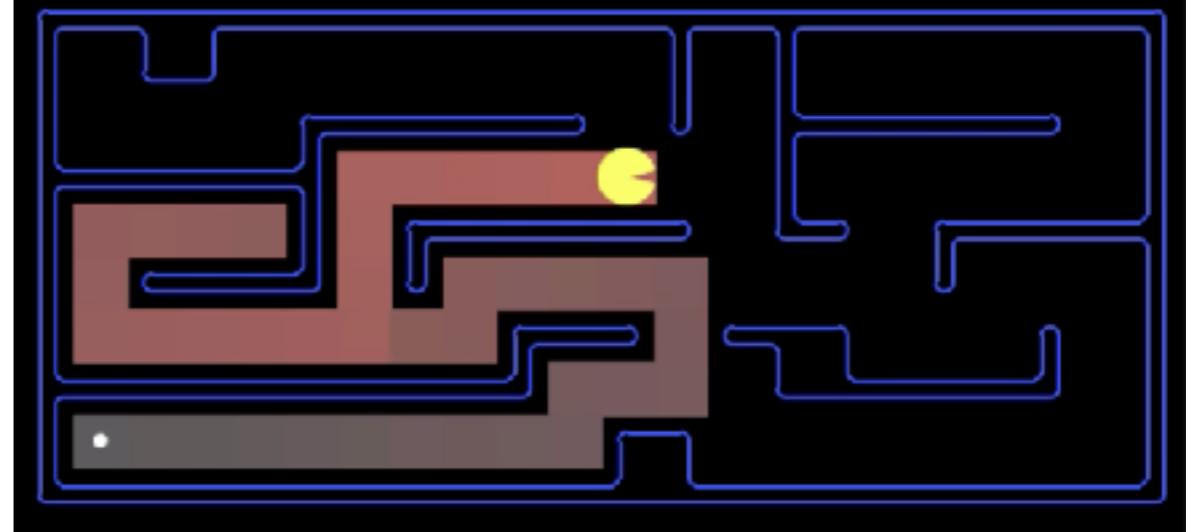
Etape 4 : [state : G^D , path : $S \rightarrow A \rightarrow D \rightarrow G$ (6)]

Recherche A* - exemple



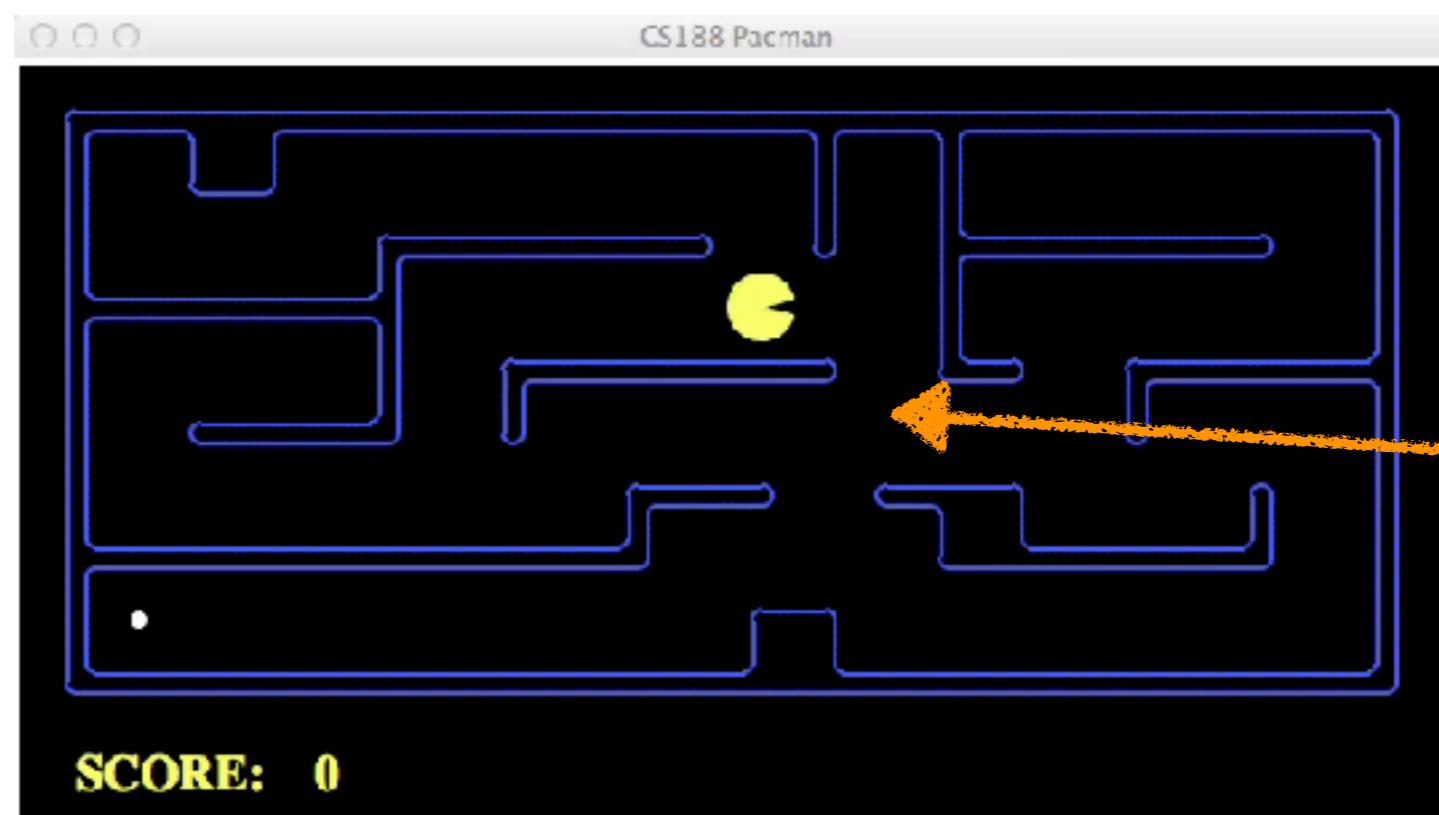
Recherche à coût uniforme

Optimal mais beaucoup de noeuds étendus



Recherche gloutonne

Peu de noeuds étendus mais non optimal



Chemin optimal
découvert avec moins
de noeuds étendus

Recherche A* - propriétés

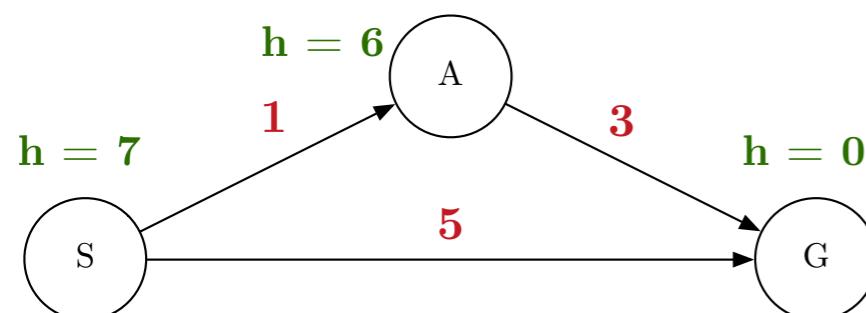
Complétude

La recherche est complète (si tous les coûts sont positifs et que la solution a un coût fini)

Similaire à la recherche à coût uniforme

?

Est-ce que la recherche A* est optimale ?



Etape 1 : [state : S , fringe : $\langle (A, 7)^S, (G, 5)^S \rangle$]

Etape 2 : [state : G^S , path : $S \rightarrow G$ (5)]

Une solution non optimale est obtenue

Souffre des mêmes difficultés d'une recherche gloutonne en cas de mauvaise heuristique

On a permis qu'une mauvaise solution ait une meilleure priorité qu'un noeud de la solution optimale

Cela s'est produit car notre heuristique a donné une estimation trop coûteuse à un état

Bonne nouvelle: il est possible de concevoir des heuristiques empêchant cette situation

(1) Heuristiques admissibles

(2) Heuristiques consistantes

Heuristique admissible



Heuristique admissible

Une heuristique est admissible (ou optimiste) si elle ne surestime jamais le coût pour atteindre le meilleur état final

$$0 \leq h(n) \leq h^*(n), \text{ pour tout état } n$$

$h^*(n)$ indique le coût optimal pour atteindre l'état final à partir de n



Optimalité d'une recherche A* en arbre

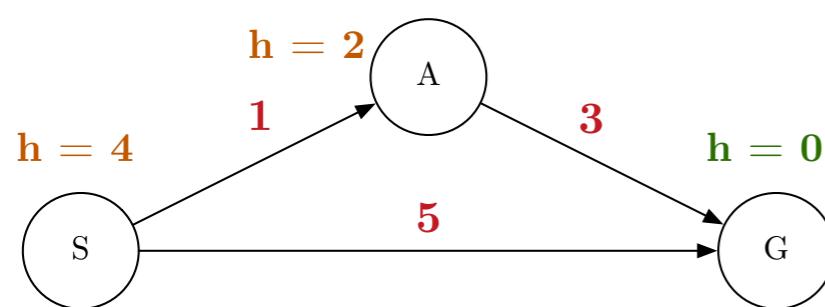
Avec une heuristique admissible, on a la garantie qu'une recherche A* en arbre est optimale

Intuition de ce résultat

Une heuristique admissible garantit que le coût réel optimal d'un noeud ne sera jamais sur-estimé

Autrement dit, l'heuristique va donner une valeur inférieure au coût réel optimal du noeud

Intuition: l'heuristique ne va pas empêcher l'extension du noeud, s'il peut appartenir au chemin optimal



$$h^*(S) = 4$$

$$h^*(A) = 3$$

$$h^*(G) = 0$$

Etape 1 : [state : S , fringe : $\langle (A,3)^S, (G,5)^S \rangle$]

Etape 2 : [state : A^D , fringe : $(G,5)^S, (G,4)^A$]

Etape 3 : [state : G^A , path : $S \rightarrow A \rightarrow G$ (4)]

Définir une heuristique admissible est un point de conception majeur pour une recherche A*

Preuve d'optimalité

Paramètres

n : noeud appartenant au chemin optimal

$g(n)$: coût pour atteindre le noeud n

$h(n)$: fonction heuristique appliquée à l'état n

$f(n) = g(n) + h(n)$: fonction de priorité de l'algorithme A*

$h^*(n)$: coût optimal pour atteindre un état final depuis n

$g^*(n)$: coût optimal pour atteindre un état n depuis l'état initial

C^* : coût de la solution optimale

C : coût d'une solution non optimale (chemin en zigzag)

Preuve par contradiction

Propriété à prouver: si $h(n)$ est admissible, alors A* est optimal

Supposons que:

- (1) A* avec la fonction de priorité $f(n)$ retourne un chemin de coût non-optimal ($C > C^*$)
- (2) $h(n)$ est admissible: $h(n) \leq h^*(n)$

D'un côté, on a de (1):

$f(n) > C^*$ (Il existe forcément un noeud n appartenant au chemin optimal qui n'a pas été étendu)

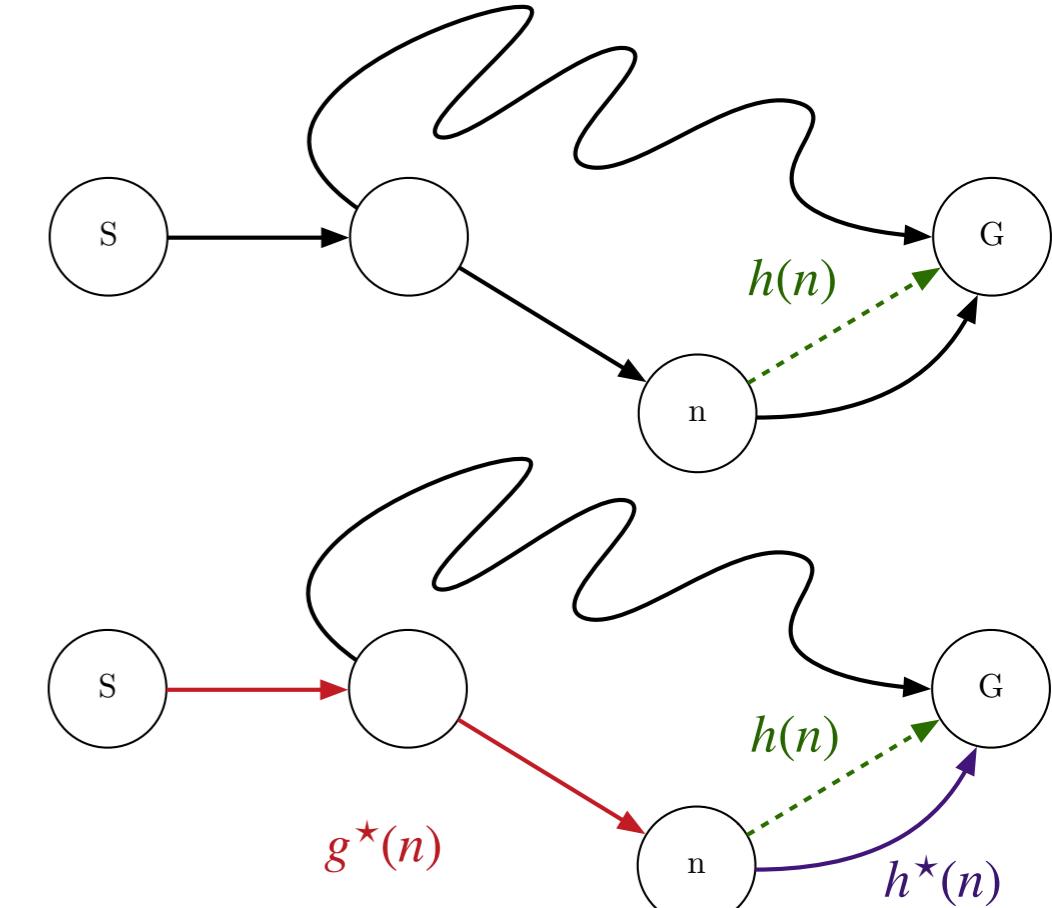
D'un autre côté, on a de (2):

$f(n) = g(n) + h(n)$ (définition de A*)

$f(n) = g^*(n) + h(n)$ (car n appartient au chemin optimal)

$f(n) \leq g^*(n) + h^*(n)$ (définition d'une heuristique admissible)

$f(n) \leq C^*$ (définition du coût du chemin optimal)

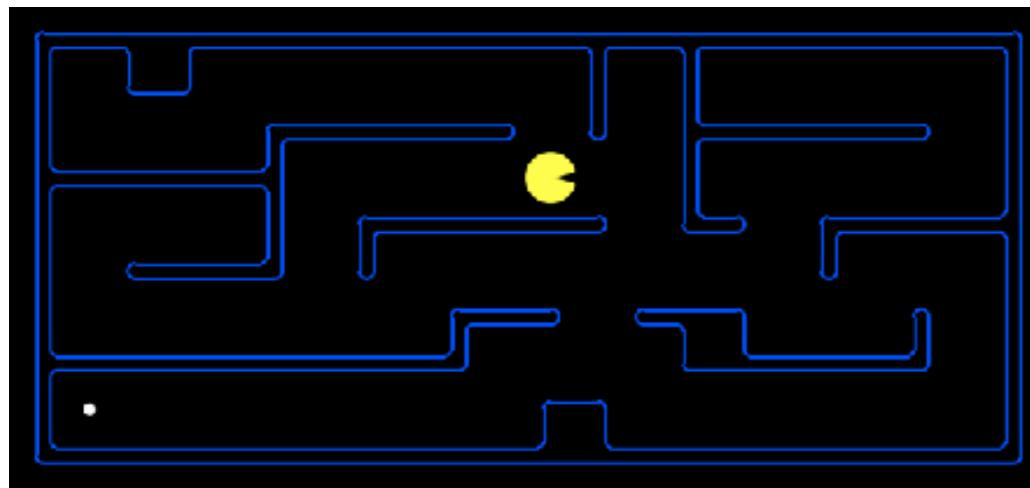


Contradiction !

Si l'heuristique est admissible,
alors A* ne peut pas retourner
une solution non-optimale

Exemples d'heuristiques (non)-admissibles

Exemples d'heuristique



Heuristique admissible: $0 \leq h(n) \leq h^*(n)$, pour tout état n

?

Est-ce que les heuristiques suivantes sont admissibles ?

$$h(n) = 0$$

Oui ! mais inutile... (recherche réduite à UCS)

$$h(n) = \text{EuclidianDistance}(n, G)$$

Oui ! impossible de faire mieux que la distance à vol d'oiseau

$$h(n) = \text{ManhattanDistance}(n, G)$$

Oui ! même raisonnement, car on se déplace sur une grille

$$h(n) = \text{SolutionCostGreedy}(n)$$

Non ! risque de surestimer le coût de la solution

$$h(n) = \text{SolutionCostBFS}(n)$$

Oui ! mais inutile car plus coûteux que A*

Notez que cela est vrai car les coûts sont unitaires !

$$h(n) = \text{SolutionCostDFS}(n)$$

Non ! risque de donner un coût non-optimale (sur-estimation)

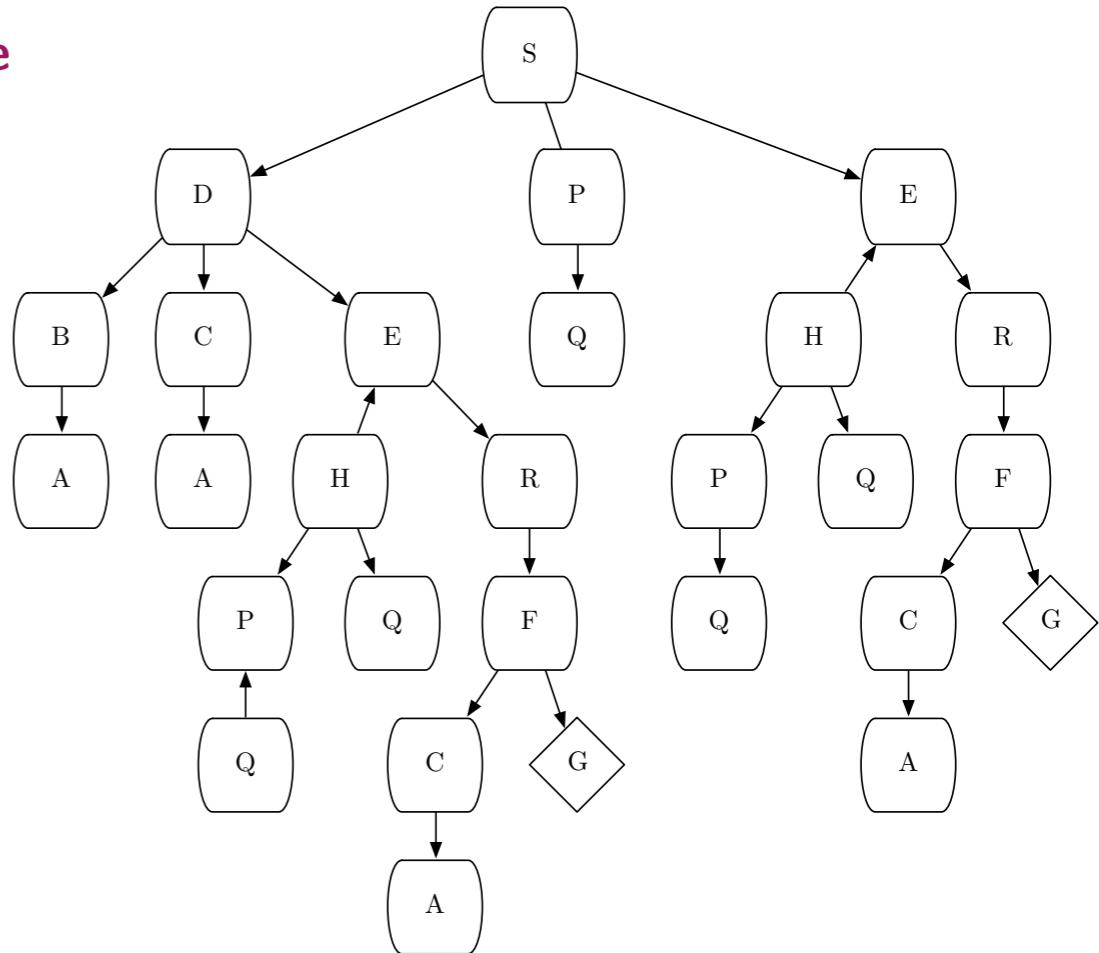
$$h(n) = h^*(n)$$

Oui ! mais inutile... connaître cette valeur revient à avoir la solution

Table des matières

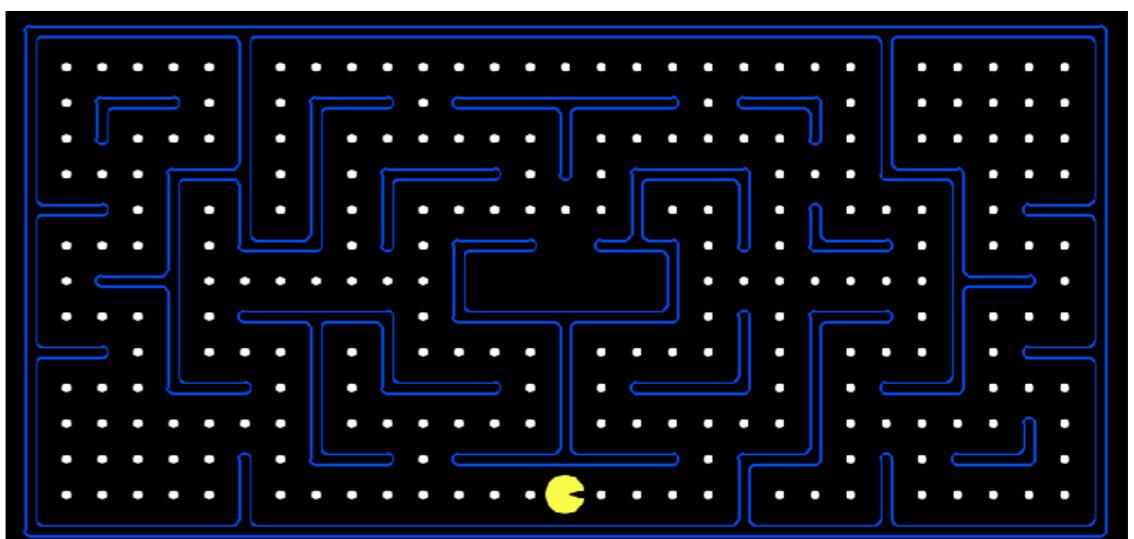
Stratégies de recherche

- ✓ 1. Définition et modélisation d'un problème de recherche
- ✓ 2. Agents réflexes
- ✓ 3. Agents axés sur la recherche
- ✓ 4. Recherche en arbre (*tree search*)
- ✓ 5. Recherche sans information: DFS, BFS, UCS, IDS
- ✓ 6. Recherche avec information: *greedy search, A**
- 7. Conception d'heuristiques
- 8. Recherche en graphe (*graph search*)



Problèmes abordés

- 1. *Pacman*
- 2. *8-puzzle*
- 3. Planification de routes



Construction d'heuristiques - techniques et bonnes pratiques



Comment construire des bonnes heuristiques (admissibles) ?

Très difficile de répondre à cette question !

Il s'agit plus d'un art qu'une science exacte

Il existe néanmoins des astuces et des bonnes pratiques

Relaxation du problème

Un problème relaxé est un problème auquel on a retiré certaines contraintes

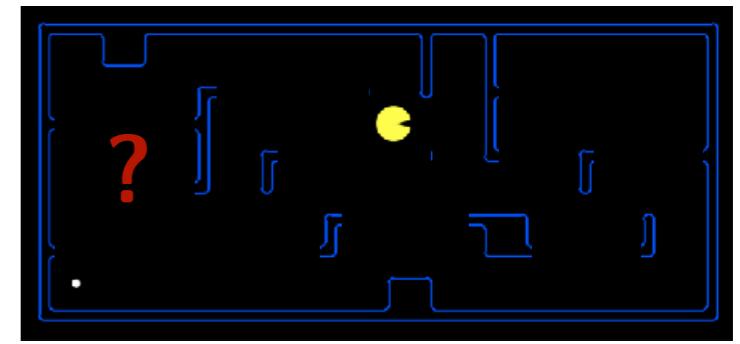
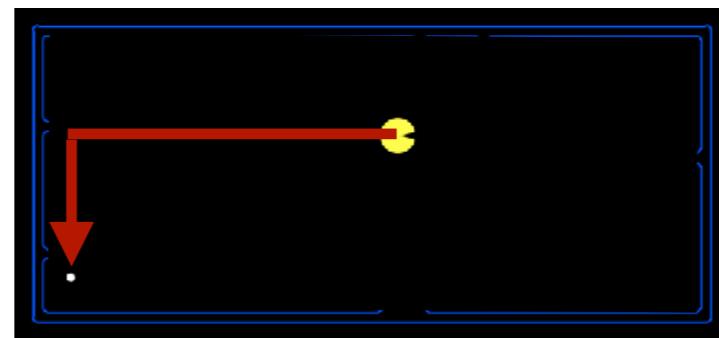
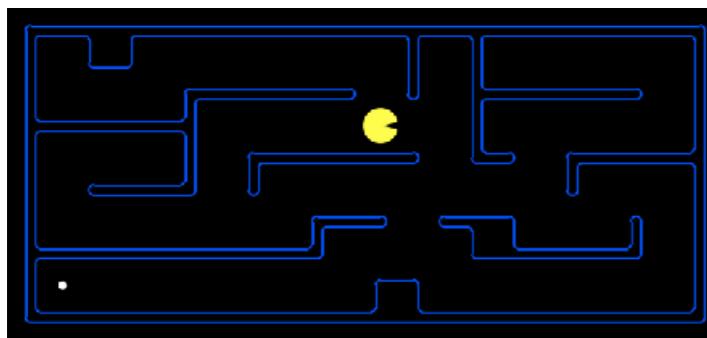
Cela le rend généralement plus facile à résoudre

Une solution au problème relaxé peut donner une bonne heuristique admissible

Relaxation forte: heuristique plus facile à calculer mais peu précise (grosse simplification du problème)

Relaxation faible: heuristique plus précise mais plus dure à obtenir (faible simplification du problème)

Illustration



La relaxation forte met en évidence la distance de Manhattan comme heuristique

Construction d'heuristiques - techniques et bonnes pratiques

Collection d'heuristiques

Il est possible que vous ayez plusieurs heuristiques pour un même problème

Au lieu d'en choisir une, vous pouvez les évaluer toutes, et prendre la meilleure pour chaque noeud

$$h(n) = \max(h_1(n), h_2(n), \dots, h_k(n))$$

Avantage: si vos différentes heuristiques sont admissibles, alors leur maximum l'est aussi

Inconvénient: coût de calcul supplémentaire pour obtenir les différentes heuristiques

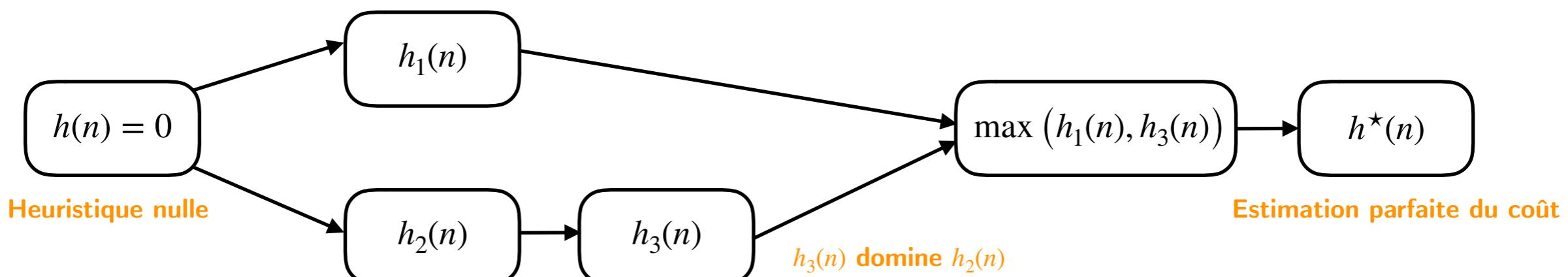
Heuristiques dominantes

Il est également possible qu'une heuristique soit strictement moins précise qu'une autre

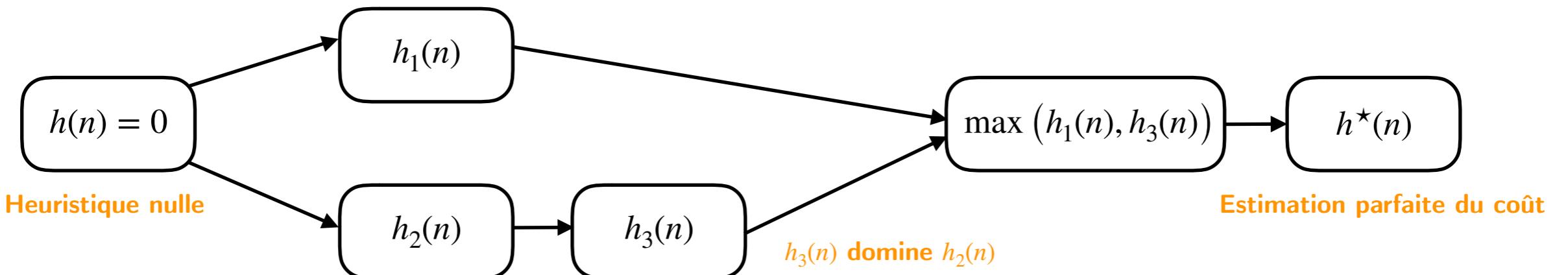
$$h_1(n) < h_2(n), \text{ pour chaque état } n$$

On dit alors que l'heuristique est dominée par une autre

Exemple de relations de dominance



Construction d'heuristiques - considérations pratiques



?

Quelle heuristique est plus précise entre $h_1(n)$ et $h_2(n)$?

Difficile à répondre ! il existe des états où h_1 est plus précise et inversement

Aucune heuristique ne domine l'autre

Une solution est de prendre le maximum entre les deux pour l'estimation d'un état

?

En pratique, est-il préférable d'utiliser $h_2(n)$ ou $h_3(n)$?

Certitude: l'heuristique h_3 domine h_2 , elle est donc plus précise

Par contre, on n'a aucune information sur le temps de construction de l'heuristique

Obtenir h_3 est potentiellement beaucoup plus lent que h_2 (à l'extrême, h^* est inconcevable à obtenir)

Une heuristique dominée mais plus rapide peut être plus intéressante à utiliser

La vitesse d'exécution d'une heuristique est également une considération importante !

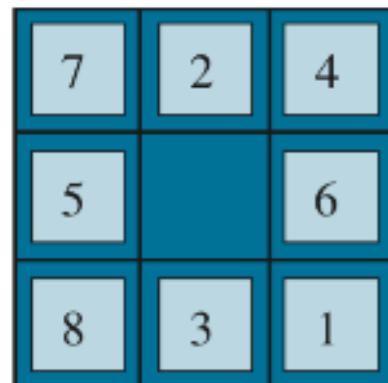
Exemple illustratif - 8-puzzle

Problème du 8-puzzle

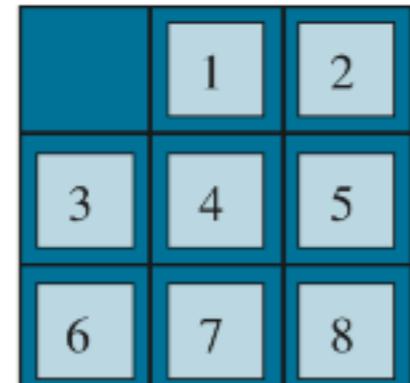
Objectif: déplacer les pièces afin de les ordonner

Contrainte: une pièce ne peut se déplacer que dans le trou

Coût: effectuer le moins de mouvements possibles



Etat initial



Etat final

Modélisation

Etats: chaque position possible du plateau

Actions: déplacer la case vide (G,D,H,B)

Transition: déplacement de la case

Coût: unitaire pour chaque déplacement

?

Quelle est la taille de l'espace d'états ?

?

Avez-vous une idée d'heuristique ?

Taille de l'espace

Chaque pièce est unique et n'est placée qu'une seule fois

Nombre d'états: $9! = 362880$

Note: il s'agit d'une borne supérieure car toutes les configurations ne sont pas atteignables

<https://mathworld.wolfram.com/15Puzzle.html>

Heuristiques

Choix 1: nombre de pièces placées incorrectement

$$h_1(s) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$$

Relaxation: les pièces peuvent se téléporter directement à leur position finale

Choix 2: distance de Manhattan cumulée de chaque pièce

$$h_2(s) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

Relaxation: les pièces peuvent se déplacer sans conflit à leur position finale

Exemple illustratif - 8-puzzle



Quelle heuristique vous paraît la plus précise ?

Search Cost (nodes generated)	
<i>d</i>	BFS
6	128
8	368
10	1033
12	2672
14	6783
16	17270
18	41558
20	91493
22	175921
24	290082
26	395355
28	463234

d : coût de la solution optimale

Analyse des performances

Nombre de noeuds explorés avant de trouver la solution

Résultats moyens sur 100 puzzles aléatoires pour chaque *d*

Analyse 1: A* bat la stratégie BFS, même avec l'heuristique simple

Analyse 2: L'heuristique 2 est largement plus précise que l'heuristique 1

On peut facilement démontrer qu'elle domine l'autre

Attention: le temps d'exécution n'est pas pris en compte (négligeable ici)

Conseils de conception



Ayez toujours en tête la complexité temporelle de vos heuristiques

Exemple: Une heuristique précise mais lente ne vaut peut-être pas le coup...

A temps d'exécution égal, préférez une heuristique ayant de plus grandes valeurs

Lorsque vous avez plusieurs bonnes heuristiques, prenez leur maximum

Il est parfois utile d'avoir des heuristiques non-admissibles

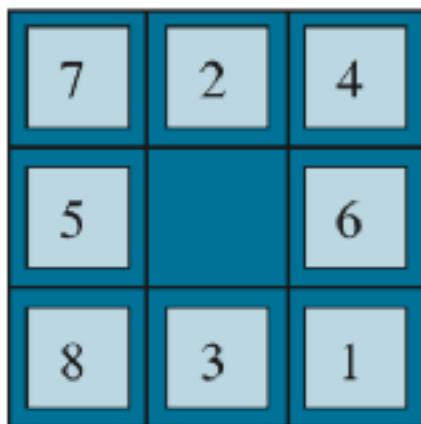
Construction d'heuristiques - techniques et bonnes pratiques

Conception d'heuristiques par des connaissances expertes

Idée: utiliser nos propres connaissances du problème (ou celles d'un expert) pour concevoir l'heuristique

Cela revient à être capable de quantifier adéquatement le coût de chaque état

Principe: identifier les caractéristiques d'un état, et de les pondérer en fonction de leur importance



Exemple: évaluer la qualité d'un état du 8-puzzle

$x_1(n)$: nombre de tuiles mal placées

$x_2(n)$: nombre de paires de tuiles adjacentes qui ne sont pas adjacentes dans l'état final

w_1, w_2 : poids associé aux deux caractéristiques (à déterminer par l'expert)

$$h(n) = w_1 x_1(n) + w_2 x_2(n)$$

Avantage: permet d'intégrer des connaissances très variées et non triviales

Inconvénient: difficile de s'assurer de l'admissibilité de l'heuristique

Inconvénient: compliquer à concevoir et à calibrer

Inconvénient: souvent incomplet (p.e., les positions des pièces ne sont pas prises en compte)

Conception d'heuristiques par apprentissage automatique

Même idée, mais utiliser de l'apprentissage automatique pour construire l'heuristique

Déceler automatiquement les caractéristiques, et leur poids

Un de mes sujets de recherches principaux :-)

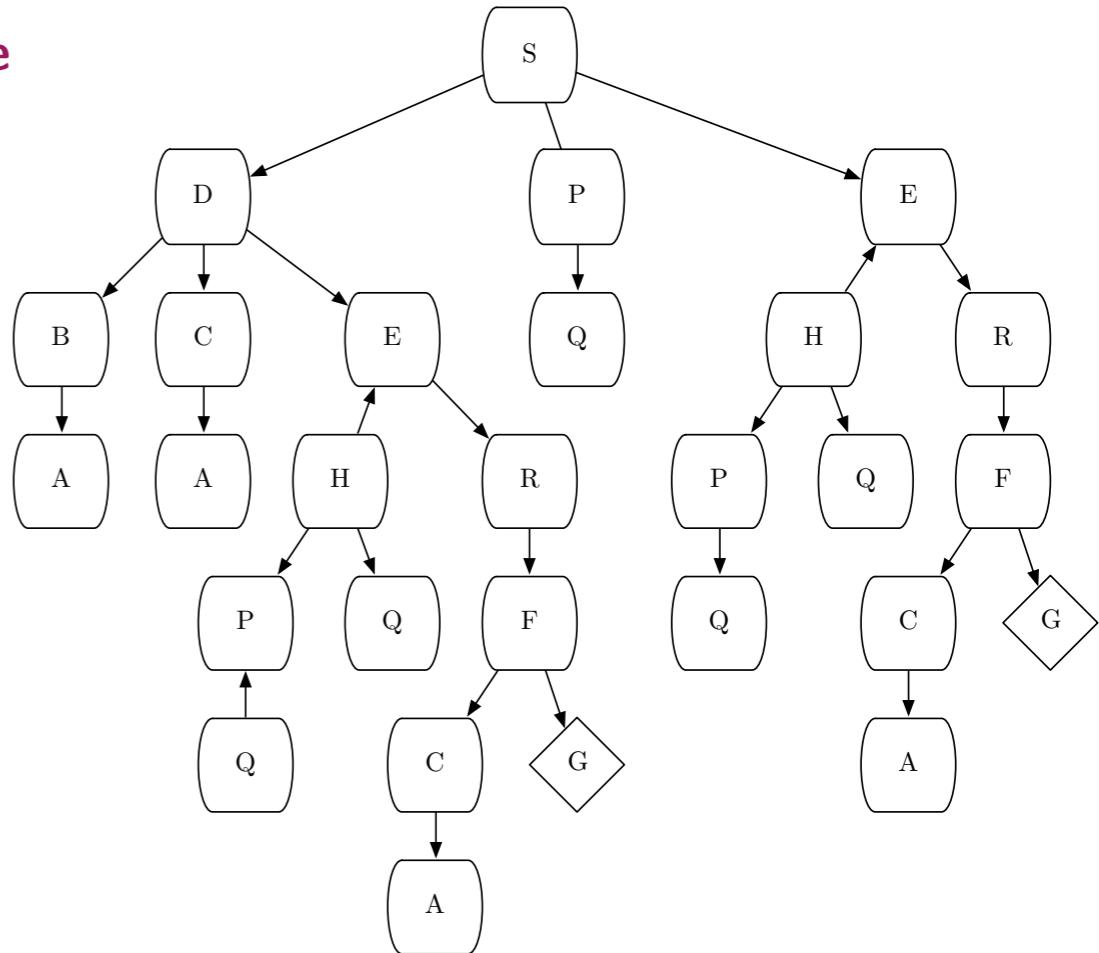


Combinatorial Optimization and
Reasoning in
Artificial Intelligence
Laboratory

Table des matières

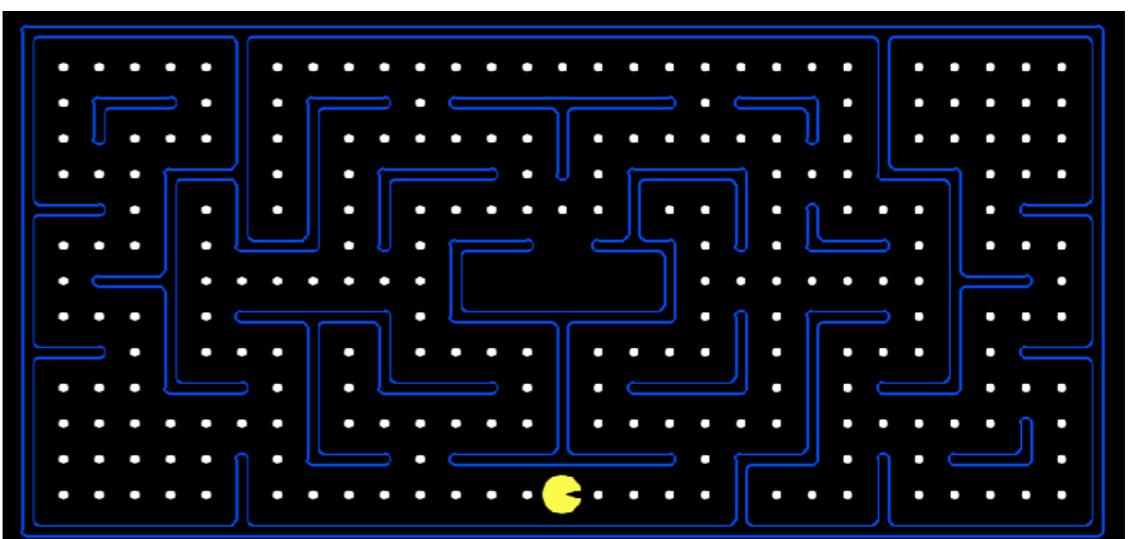
Stratégies de recherche

- ✓ 1. Définition et modélisation d'un problème de recherche
- ✓ 2. Agents réflexes
- ✓ 3. Agents axés sur la recherche
- ✓ 4. Recherche en arbre (*tree search*)
- ✓ 5. Recherche sans information: DFS, BFS, UCS, IDS
- ✓ 6. Recherche avec information: *greedy search, A**
- ✓ 7. Conception d'heuristiques
- 8. Recherche en graphe (*graph search*)



Problèmes abordés

- 1. *Pacman*
- 2. *8-puzzle*
- 3. Planification de routes



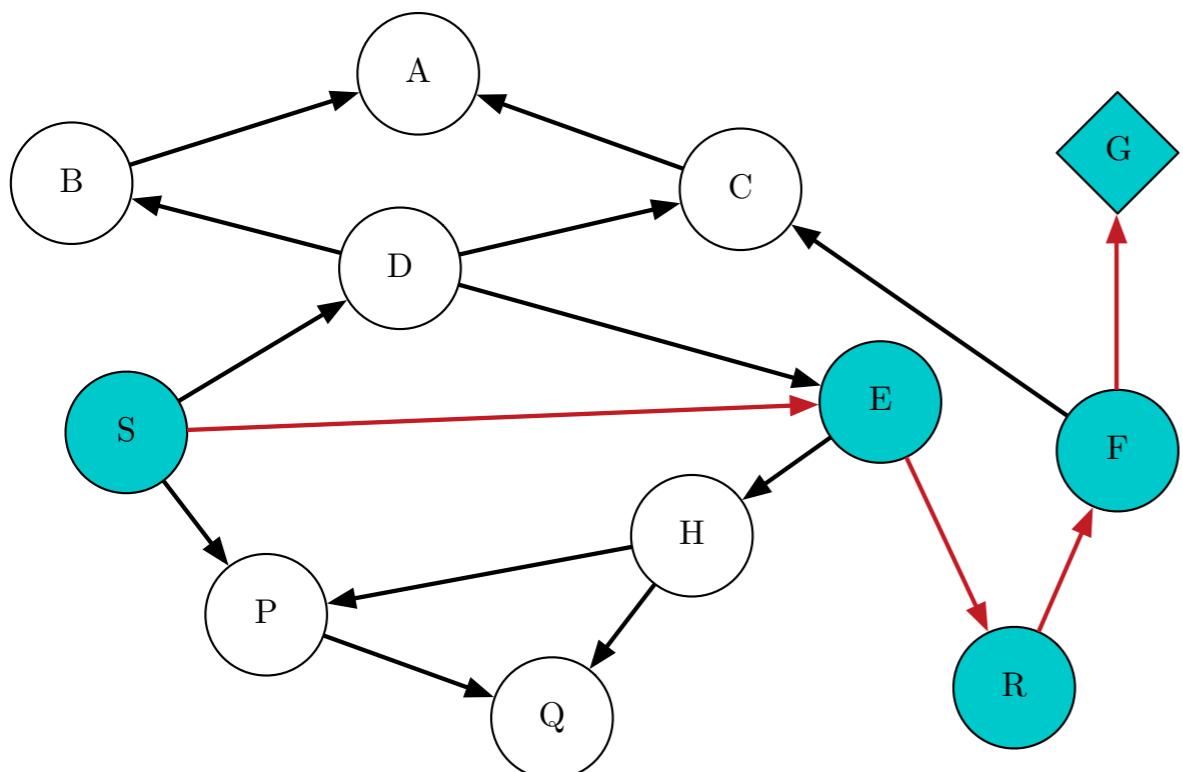
Rappel: recherche en graphe

Recherche en graphe

Jusqu'à présent nos analyses se sont concentrées sur les recherches en arbre

Or, on a vu qu'il était possible de construire une recherche en graphe

Le principe est de maintenir un ensemble des états déjà visités



GraphSearch(P) :

$s = \text{initialState}(P)$

$V = \emptyset$

$L = \{s\}$

while $L \neq \emptyset$:

$s = \text{selectAndRemove}(L)$

if $s = \text{goalState}(P)$: return solution

else :

$C = \{c \in \text{succesors}(s, P) \mid c \text{ not in } V\}$

$L = L \cup C$

$V = V \cup s$

return no solution

Avantage: on ne revisite pas le même état lors de la recherche

Inconvénient: on doit garder tous les états vus en mémoire, ce qui peut être très coûteux

Performances de la recherche en graphe



A t-on les mêmes algorithmes (DFS, BFS, A*, etc.) avec la recherche en graphe ?

Généralité de nos stratégies



Tous les algorithmes vus peuvent être utilisé presque sans modification

Simple ajout d'un mécanisme de mémoire pour retenir les noeuds visités

Inconvénient: la consommation mémoire sera plus importante

Dans le pire des cas, tous les états doivent être retenus

Possibilités de mitiger cette difficulté (retenir qu'un certain nombre d'états)

Cas particulier de la recherche A*



L'algorithme A* souffre d'une difficulté supplémentaire

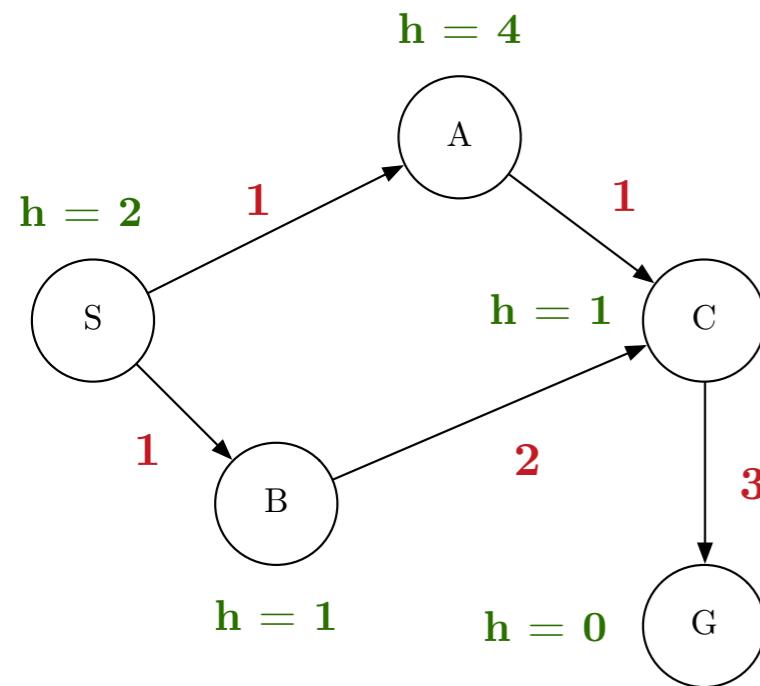
Une heuristique admissible n'est plus suffisante pour garantir l'optimalité

Une propriété plus forte doit être obtenue...

Introduction du concept d'heuristique consistante

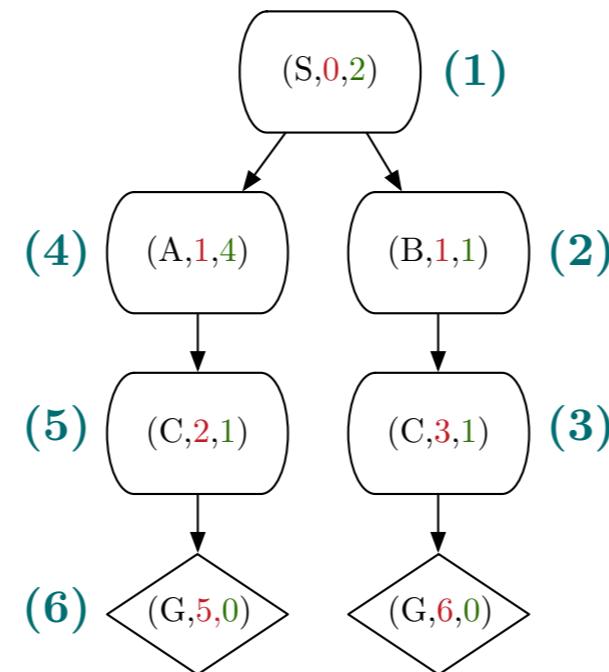
Recherche A* en graphe: cas pathologique

Cas pathologique



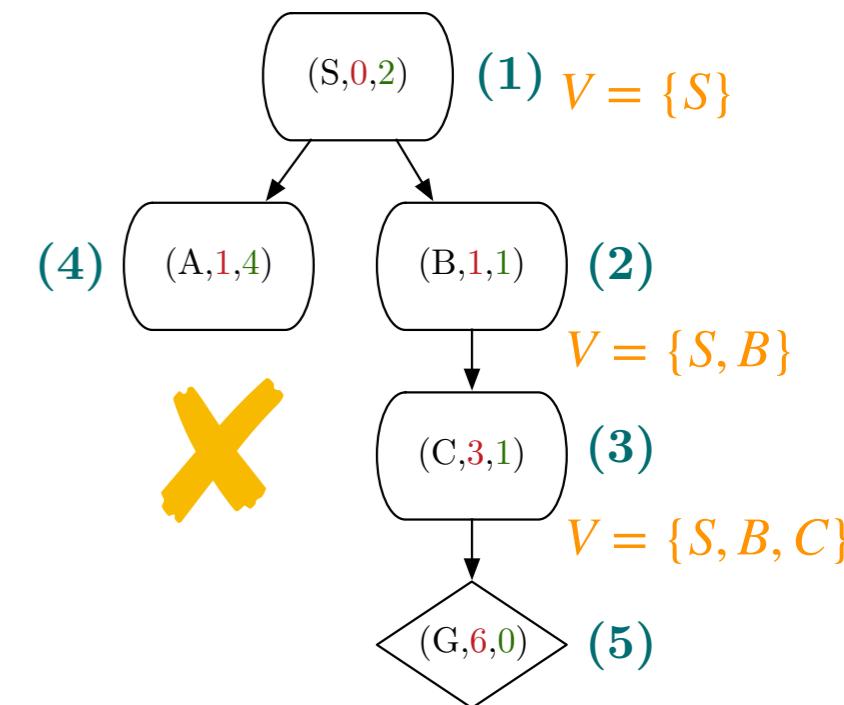
Notez que l'heuristique est admissible

Recherche A* en arbre



Solution optimale obtenue

Recherche A* en graphe



Solution non-optimale obtenue

?

Quel est le soucis selon vous ?

Le noeud $(C,2,1)$ a été étendu après $(C,3,1)$, or ce dernier n'appartient pas à la solution optimale

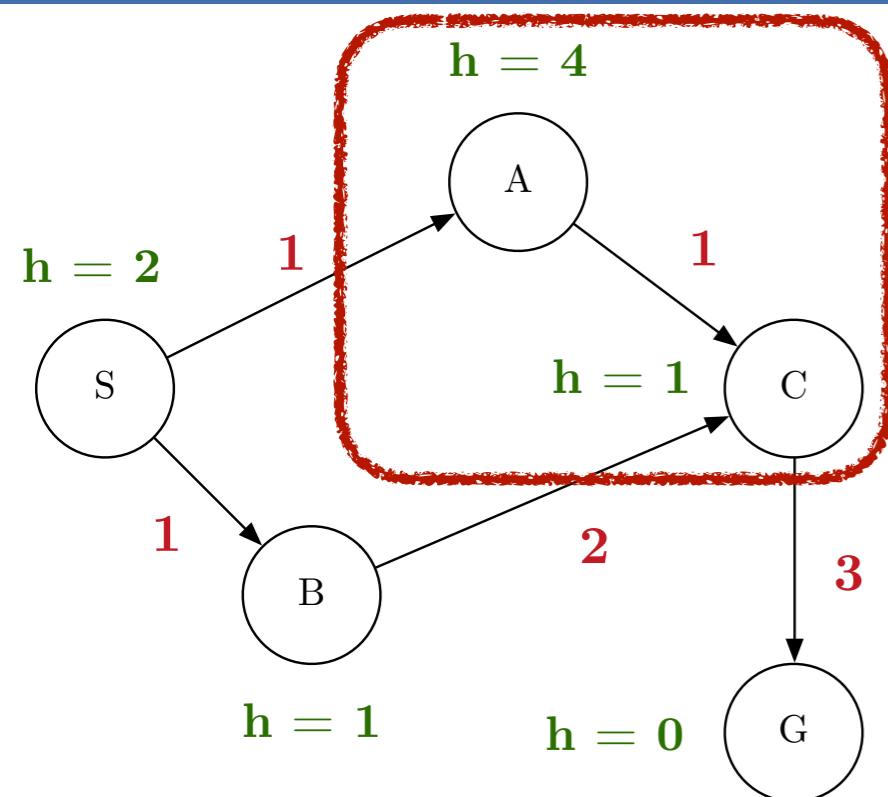
Par conséquent, une recherche en graphe ne va jamais étendre $(C,2,1)$, car l'état C a déjà été exploré

La solution optimale ne sera ainsi jamais découverte

Problème: notre heuristique a autorisé l'extension de $(C,3,1)$ avant $(C,2,1)$

Le soucis vient de l'heuristique appliquée à l'état A

Heuristique consistante



$$h(A) - h(C) \leq c(A \rightarrow C)$$

$$4 - 1 \leq 1$$

Coût de l'arc: 1

Estimation heuristique: 3

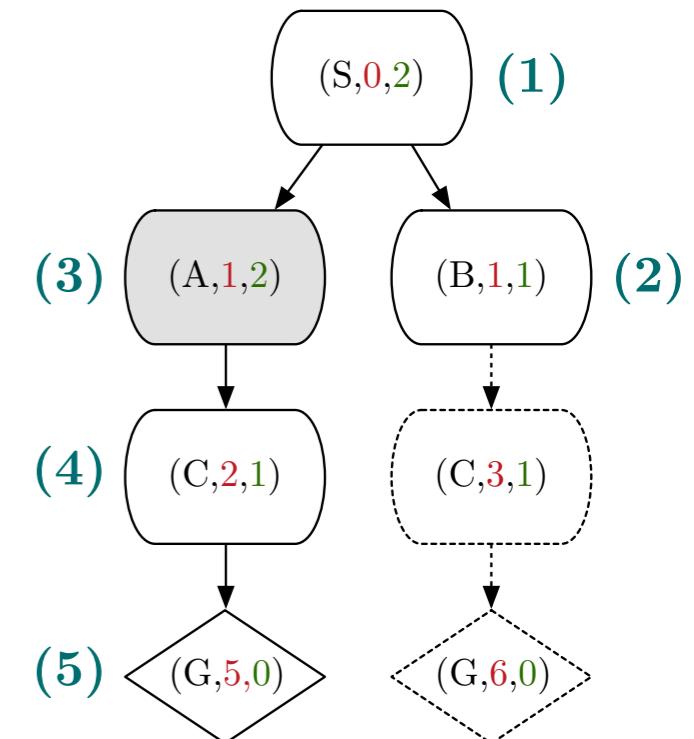
Heuristique non consistante

Avec $h(A) = 2$

$$2 - 1 \leq 1$$

Heuristique consistante

Solution optimale trouvée



Ce blocage vient du fait que l'heuristique appliquée en **A** et en **C** a sur-estimé le coût d'aller à **C** (3 vs 1)

Conséquence: aller à **C** via **A** est considéré (à tort) comme une mauvaise transition

Heuristique consistante: assure qu'un état ne peut être exploré QUE via le chemin optimal

 **Heuristique consistante**

Une heuristique est consistante si sa valeur appliquée à un arc de transition ne surestime jamais le coût de l'arc

$$h(n) - h(n') \leq c(n \rightarrow n'), \text{ pour toutes paires d'états-sucesseurs } (n \rightarrow n')$$

Propriété 1: une heuristique consistante est également admissible (l'inverse n'est pas vrai !)

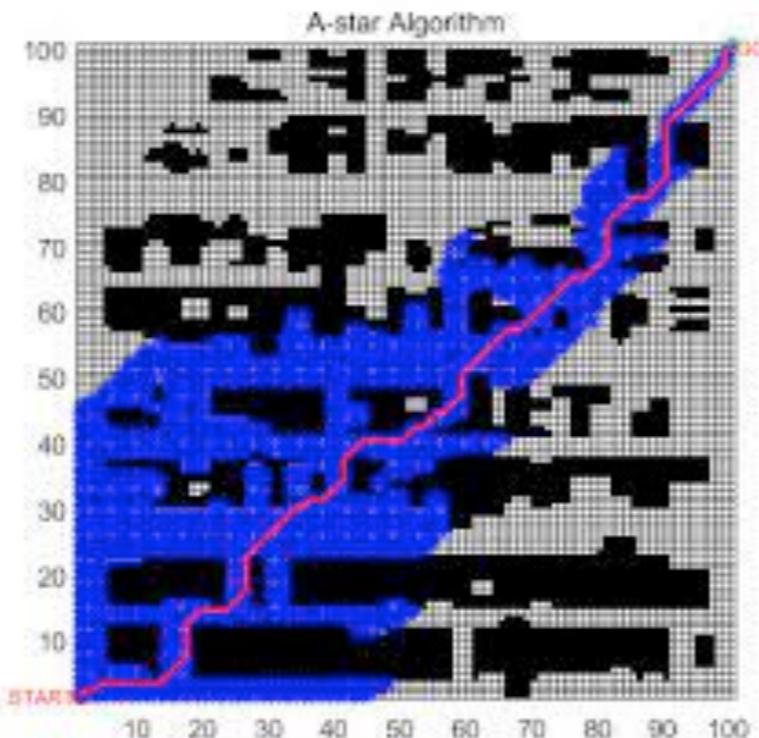
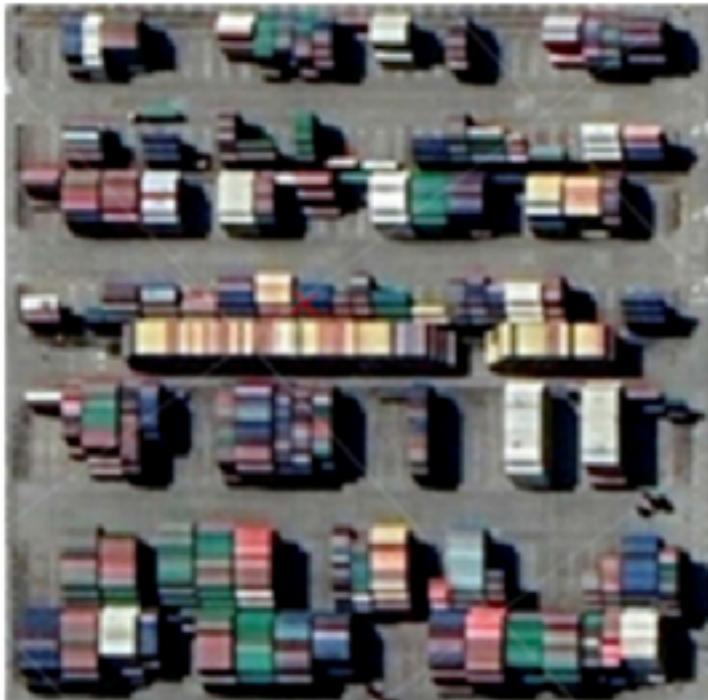
Propriété 2: une heuristique consistante est optimale pour A* avec une recherche en graphe

Applications

Pathfinding

Objectif: trouver un chemin adéquat dans un environnement parsemé d'obstacles

Applications: gestion d'entrepôts, déplacement dans les jeux vidéos, etc.



Planification de tâches

Objectif: trouver une séquence d'action pour réaliser une tâche

Applications: chaîne de production, séquençage général de tâches, etc.

Exploration de graphes

Représentation standard de systèmes complexes (réseaux routiers, sociaux, etc.)

Exemples: réseaux routiers, réseaux sociaux, représentation de connaissances, etc.



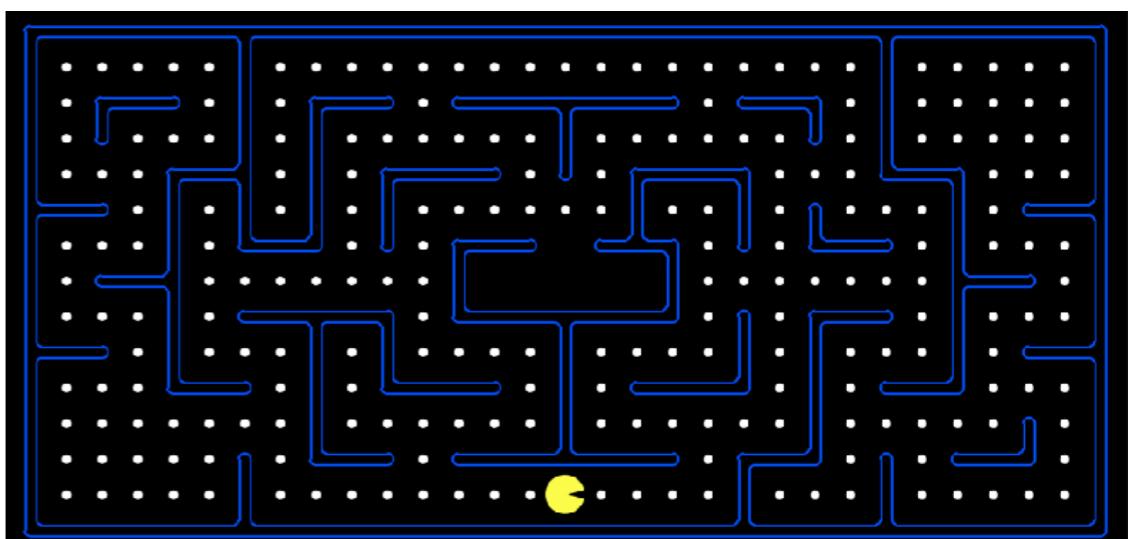
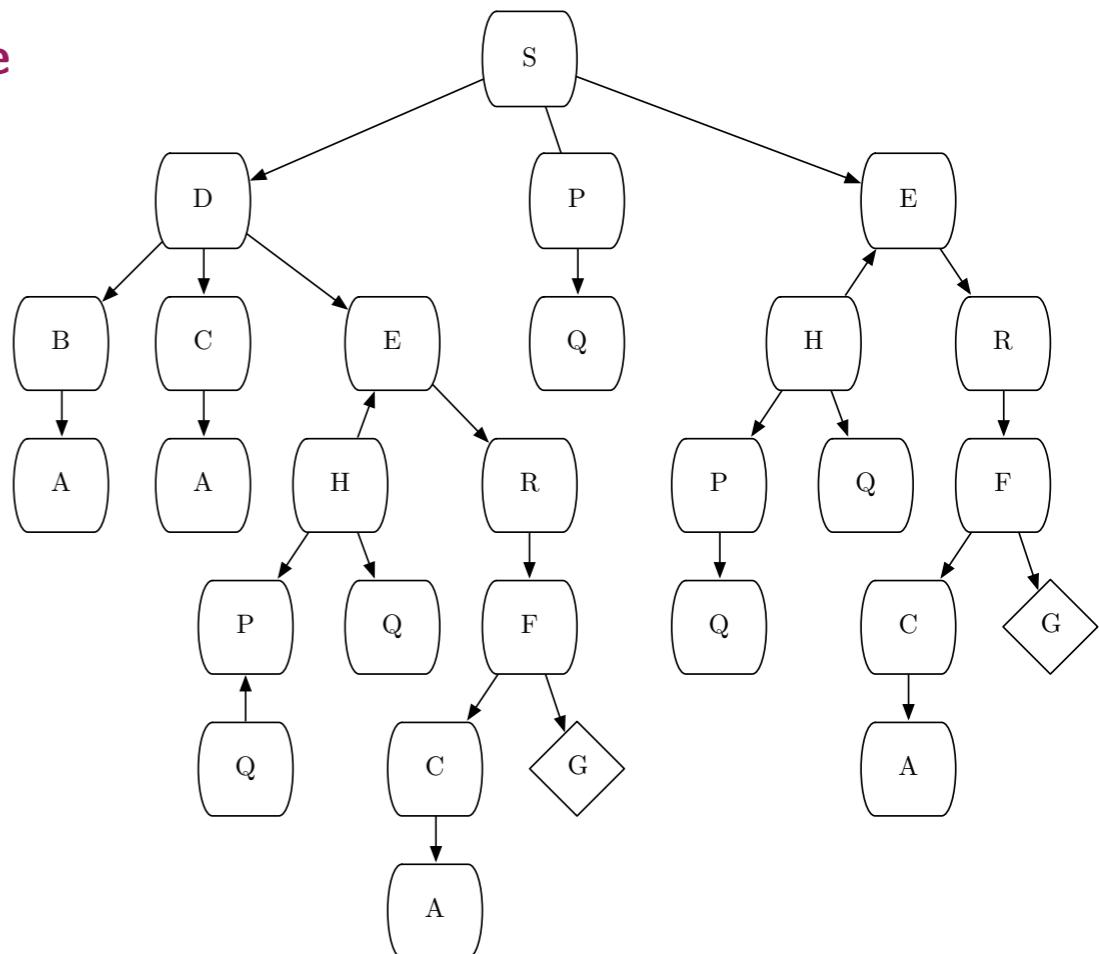
Table des matières

Stratégies de recherche

- ✓ 1. Définition et modélisation d'un problème de recherche
- ✓ 2. Agents réflexes
- ✓ 3. Agents axés sur la recherche
- ✓ 4. Recherche en arbre (*tree search*)
- ✓ 5. Recherche sans information: DFS, BFS, UCS, IDS
- ✓ 6. Recherche avec information: *greedy search, A**
- ✓ 7. Conception d'heuristiques
- ✓ 8. Recherche en graphe (*graph search*)

Problèmes abordés

- 1. *Pacman*
- 2. *8-puzzle*
- 3. Planification de routes



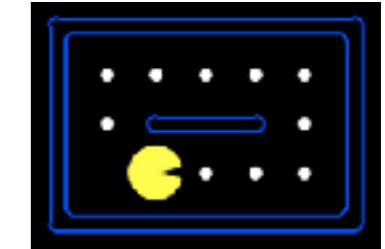
Synthèse des notions vues

Problèmes de recherche

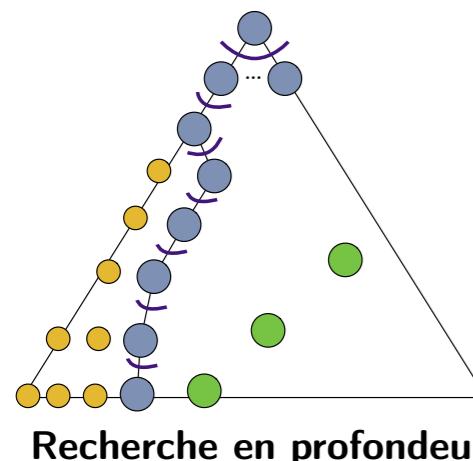
Motivation: planifier d'une séquence d'actions pour aller d'un état initial à un état final

Objectif idéal: trouver la séquence engendrant le moins de coûts

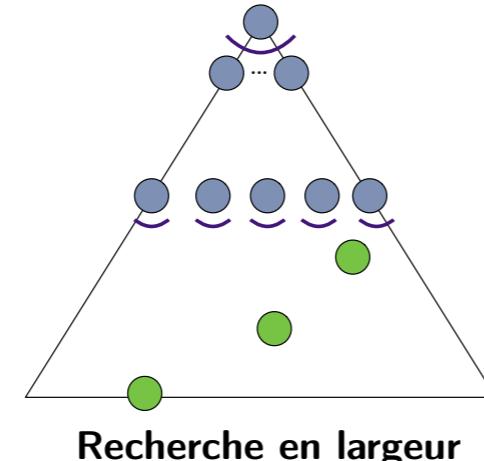
Patron de résolution générique: recherche en graphe ou en arbre



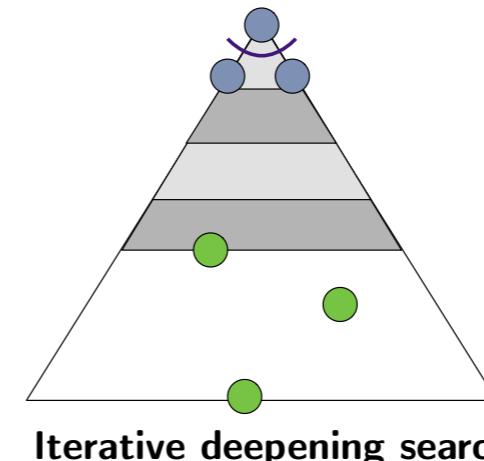
Stratégies sans information



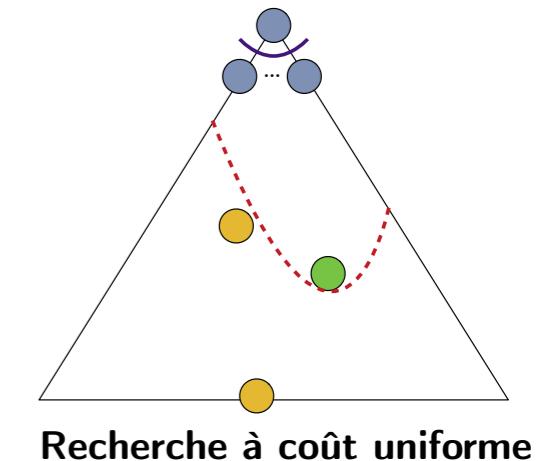
Recherche en profondeur



Recherche en largeur

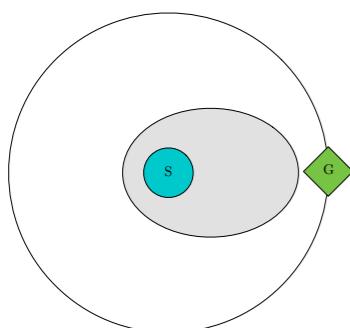


Iterative deepening search



Recherche à coût uniforme

Stratégies avec information



Inclusion d'une heuristique (information du problème) pour accélérer la recherche

Recherche gloutonne: rapide mais sans garantie

Recherche A*: combinaison de la recherche gloutonne et de coût uniforme

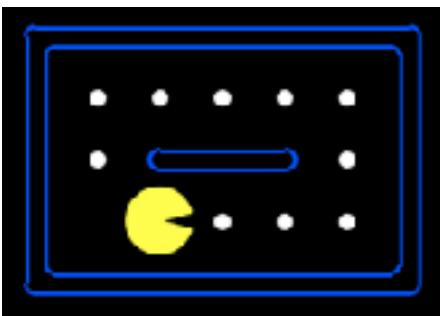
Heuristique admissible: A* avec une recherche en arbre est optimal

Heuristique consistante: A* avec une recherche en arbre/graphe est optimal

Conception d'heuristiques: relaxation, connaissances expertes, etc.

Conseils pratiques (pour les TPs)

Stratégie de recherche

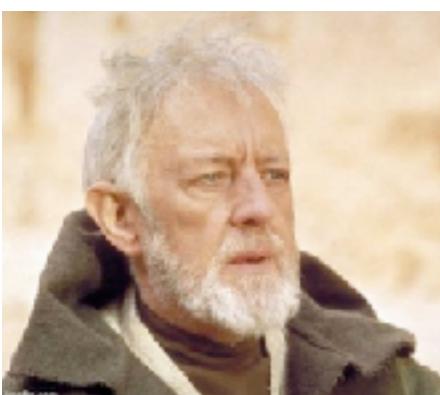


Considérez d'abord la recherche en graphe

Passez en recherche en arbre si vous avez des problèmes de mémoire

Recherche en graphe: implémentez la mémoire par un ensemble (*set*), et non une liste

Conception d'heuristiques



Commencez avec des heuristiques simples et complexifiez les progressivement

La technique de relaxation est souvent un bon point de départ

Une heuristique précise n'est pas toujours la meilleure en pratique (temps d'exécution)

Si l'optimalité n'est pas requise n'ayez pas peur d'avoir des heuristiques non admissibles

Implémentation



Testez régulièrement vos algorithmes sur des situations simples

Vérifiez que la solution obtenue est bien optimale (sur des petits problèmes)

Vérifiez que les noeuds étendus soient bien ceux attendus

Une grande partie de votre code peut-être réutilisé pour les différentes stratégies !

Il est très facile d'avoir des bugs cachés !

Méthodes avancées et limitation de nos stratégies

Autres stratégies de recherche

Recherche bidirectionnelle: recherche à partir de l'état initial et de l'état final et les faire se rencontrer

Iterative-deepening A* search (IDA*): principe de IDS mais pour A*

Simplified memory-bounded A* (SMA*): A* en indiquant une borne sur la mémoire à utiliser

Weighted A*: perdre l'optimalité pour accélérer la recherche via une heuristique faiblement non admissible

Méthodes de pruning: enlever les cycles dans une recherche en arbre

Conceptions d'heuristiques

Décomposer le problème en sous-problèmes et résoudre les sous-problèmes pour obtenir une heuristique

Pré-calculer certaines solutions et les insérer comme estimations heuristiques

Limitations

(1) L'environnement considéré à présent est entièrement observable, statique, et déterministe

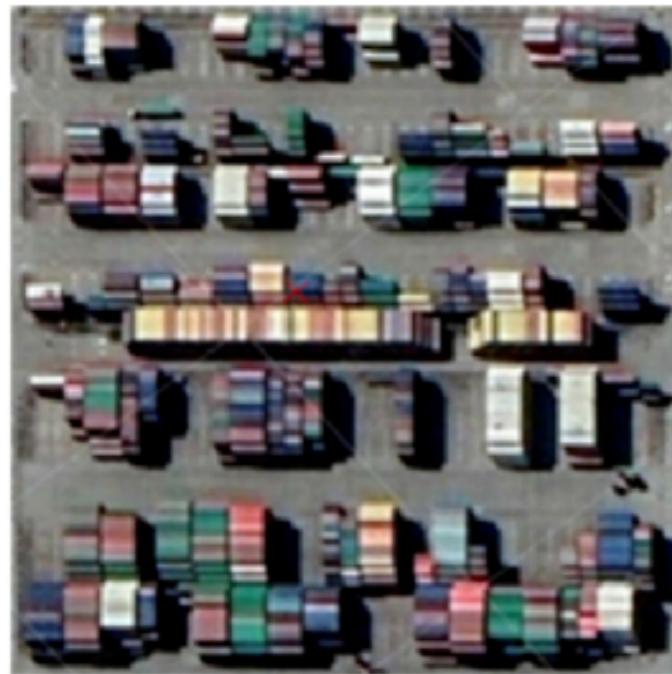
Que faire quand ces propriétés ne sont pas présentes ?

(2) Recherche avec adversaire

Comment agir en présence d'un adversaire ?

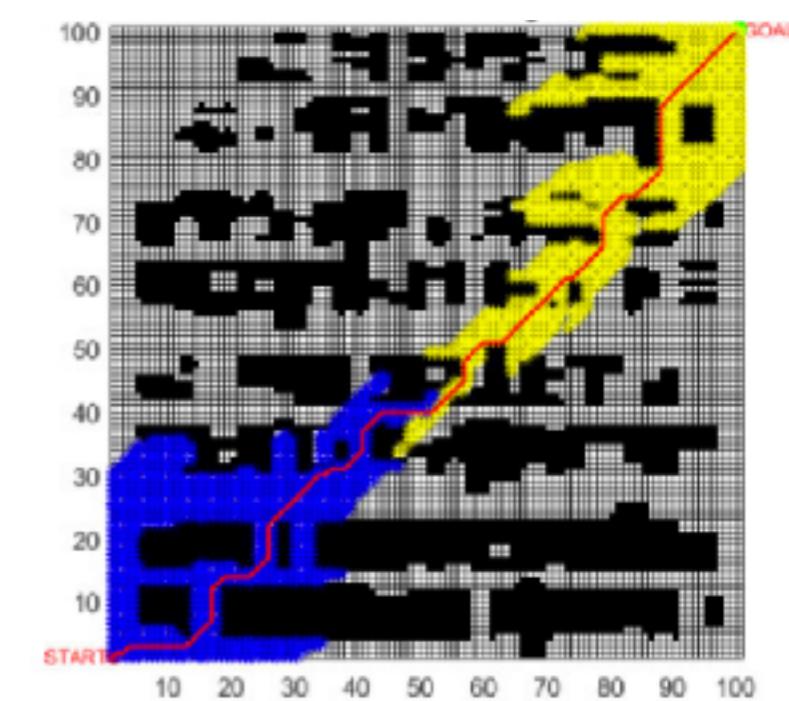
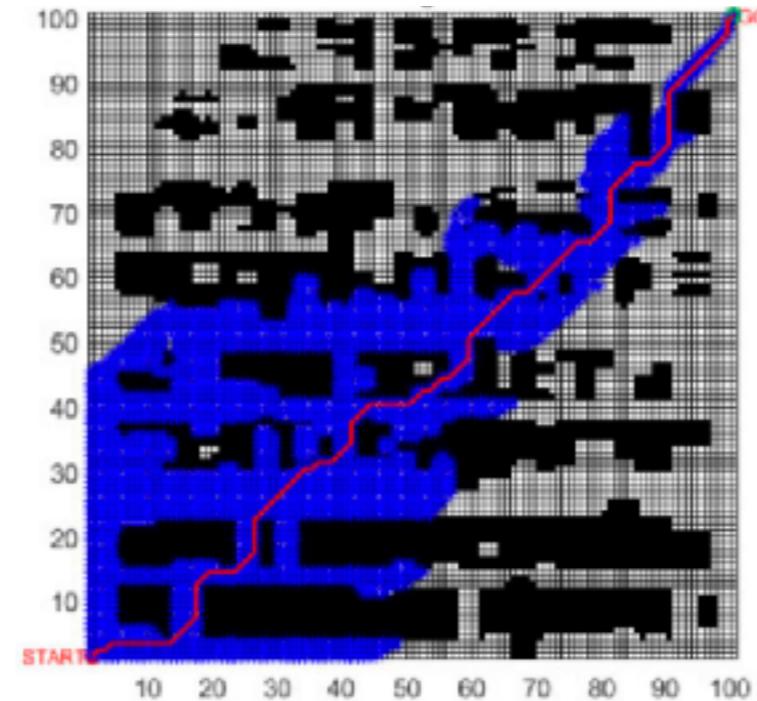
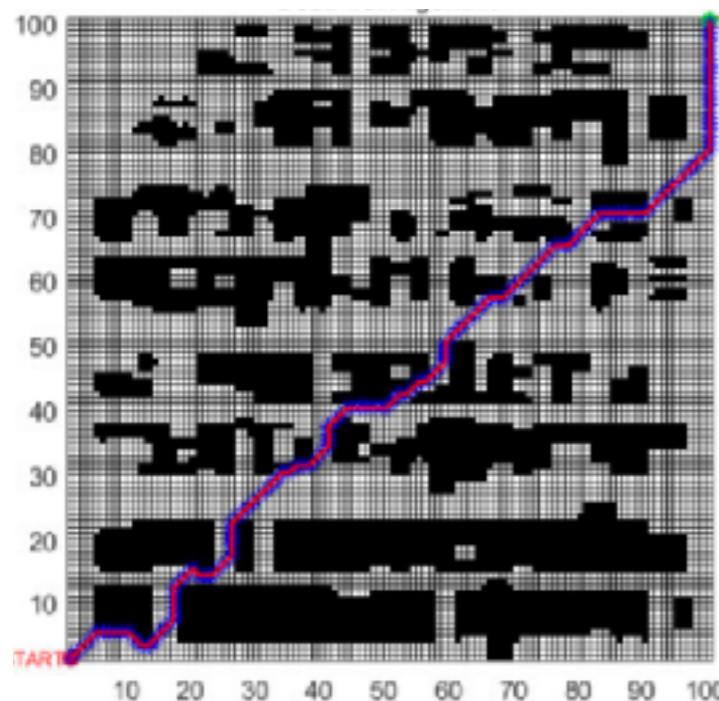
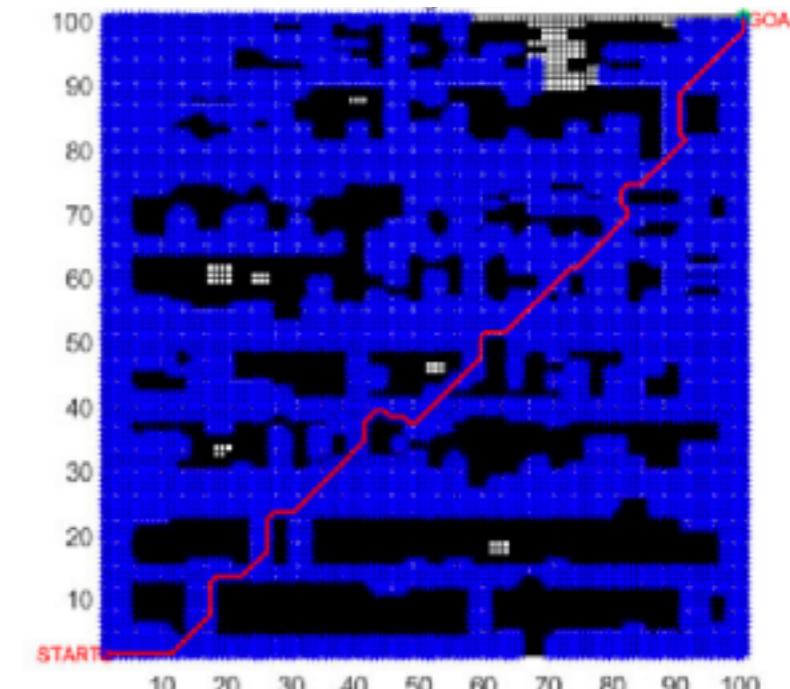
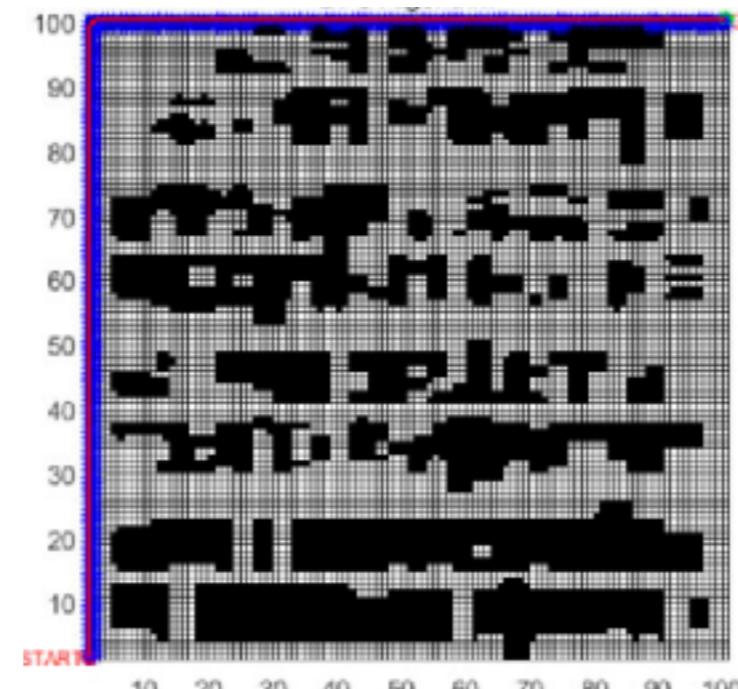
Sujet du prochain module

Dernières illustrations



Start

Goal



Note: la définition du problème fait que les coûts ne sont pas totalement unitaires (expliquant pourquoi BFS n'est pas optimal)

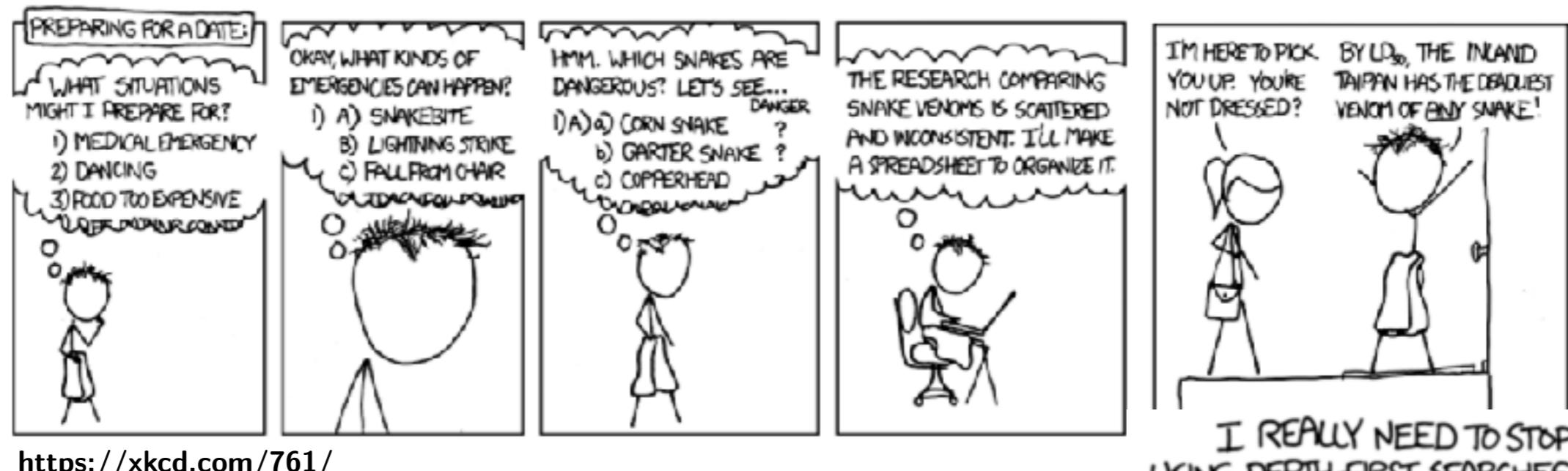
Exemples de questions d'examen

Théorie

1. Expliquer le fonctionnement et les propriétés d'une stratégie de recherche vue
2. Donner les avantages/inconvénients d'une stratégie de recherche par rapport à une autre
3. Expliquer ce qu'est une heuristique admissible ou consistante, et l'implication de ces propriétés

Pratique

1. Modéliser une situation comme un problème de recherche (états, actions, coût, succession)
2. Savoir évaluer la taille d'un espace de recherche
3. Savoir appliquer correctement un algorithme de recherche sur une situation donnée
4. Savoir construire des heuristiques non-triviales pour résoudre différents problèmes



INF8215 - Intelligence artificielle

Méthodes et algorithmes

Stratégies de recherche: FIN



**POLYTECHNIQUE
MONTRÉAL**

Quentin Cappart