

INF8175 - Intelligence artificielle

Méthodes et algorithmes

Module 1: Stratégies de recherche



POLYTECHNIQUE
MONTRÉAL

Quentin Cappart

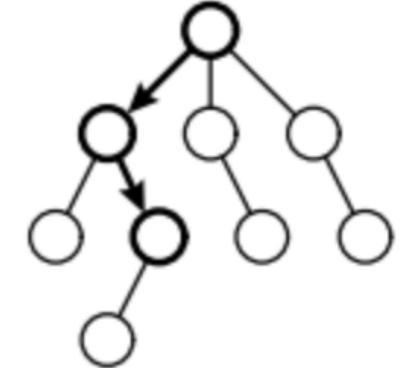
Contenu du cours

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

SS SSSS Stench		Breeze	PIT
	Breeze SS SSSS Stench Gold	PIT	Breeze
SS SSSS Stench		Breeze	
START	Breeze	PIT	Breeze

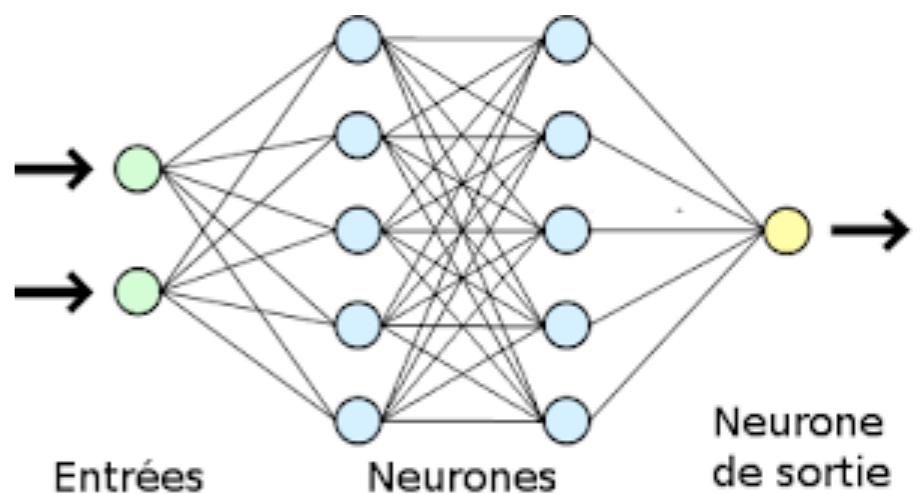
Raisonnement par apprentissage

Module 6: Apprentissage supervisé

Module 7: Réseaux de neurones et apprentissage profond

Module 8: Apprentissage non-supervisé

Module 9: Apprentissage par renforcement



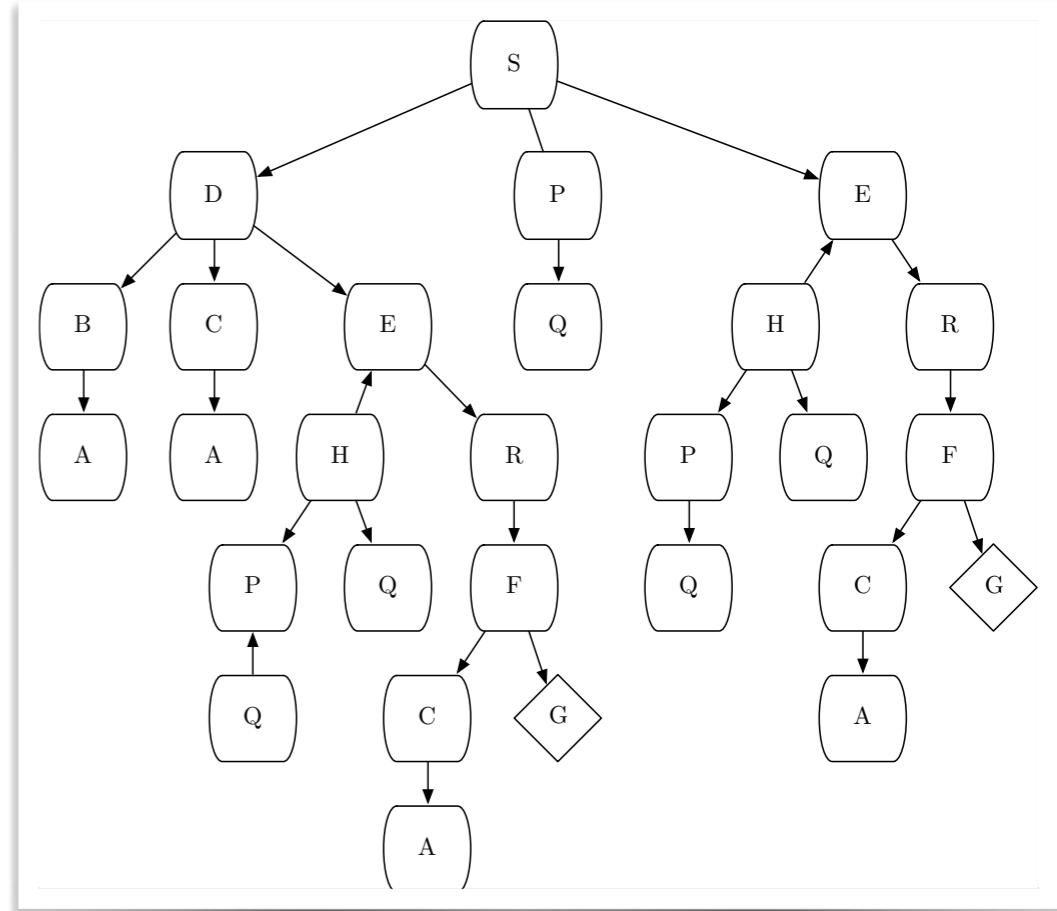
Considérations pratiques et sociétales

Module 10: Utilisation en industrie, éthique, et philosophie

Table des matières

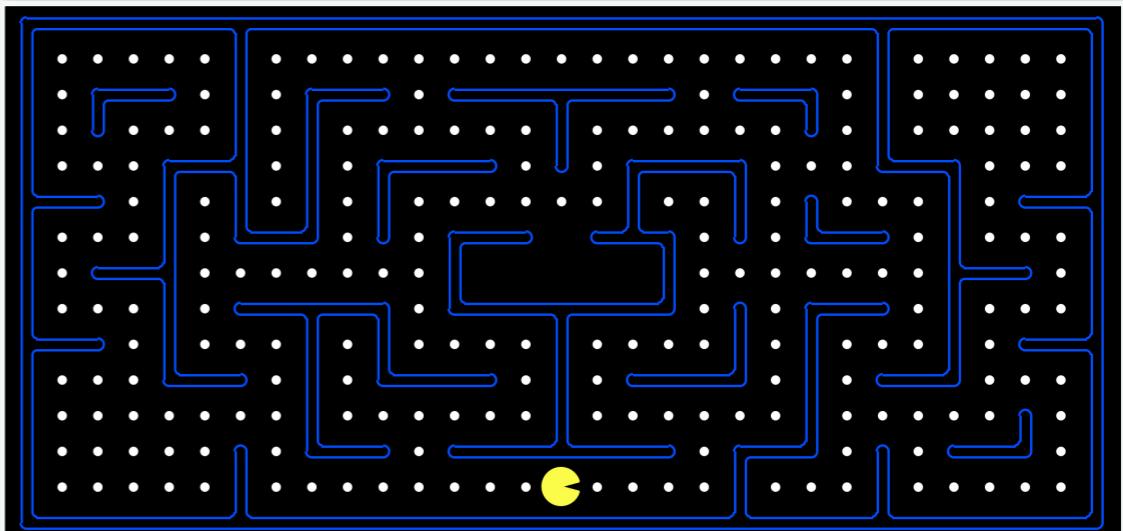
Stratégies de recherche

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*)

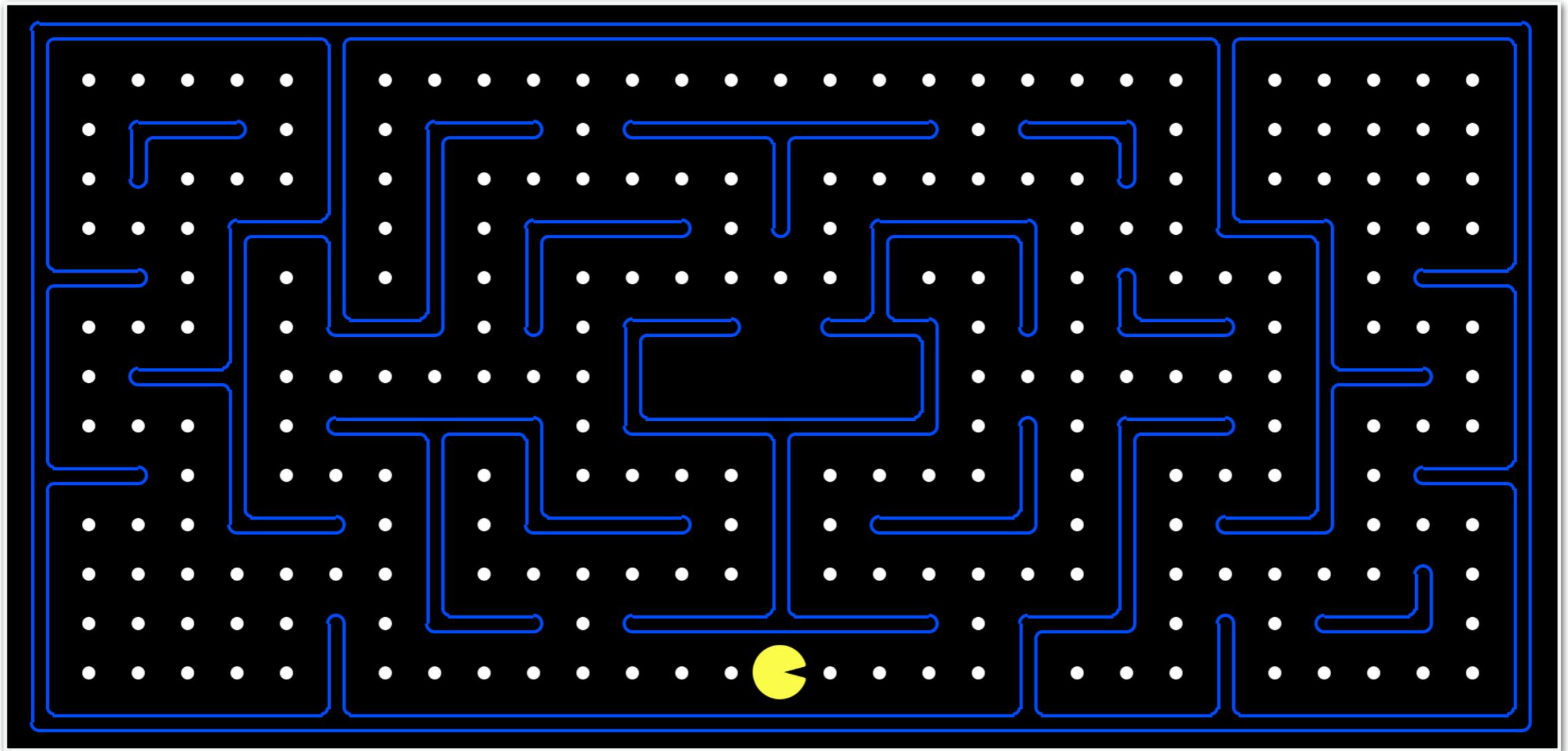


Problèmes abordés

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



Cas d'étude: *Pacman* (1980)



Objectif: concevoir un agent mangeant tous les points en exécutant le moins de déplacements possibles

Hypothèses simplificatrices: pas de fantômes, et pas de bonus

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

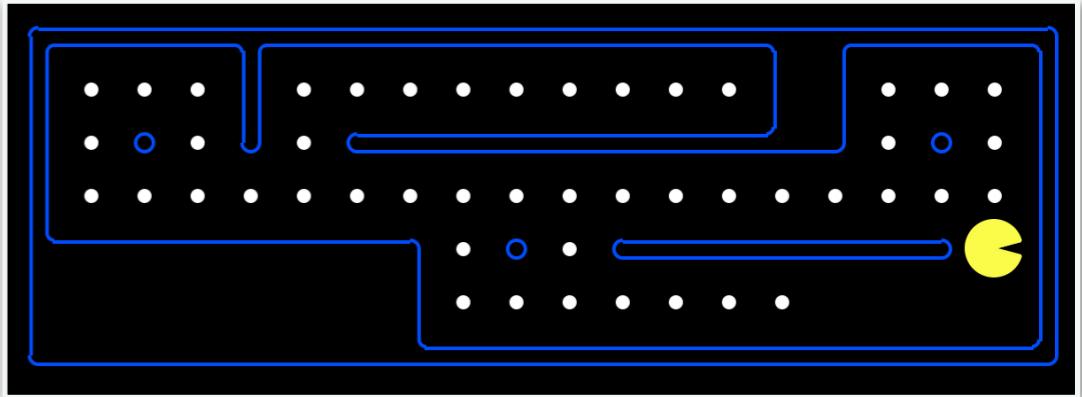


Comment implémenter un agent efficace pour cette tâche ?

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)

Variante: intégration éventuelle d'un mécanisme de mémoire

Résultat: une action sur base de l'observation et de la mémoire

Avantage: très rapide à exécuter

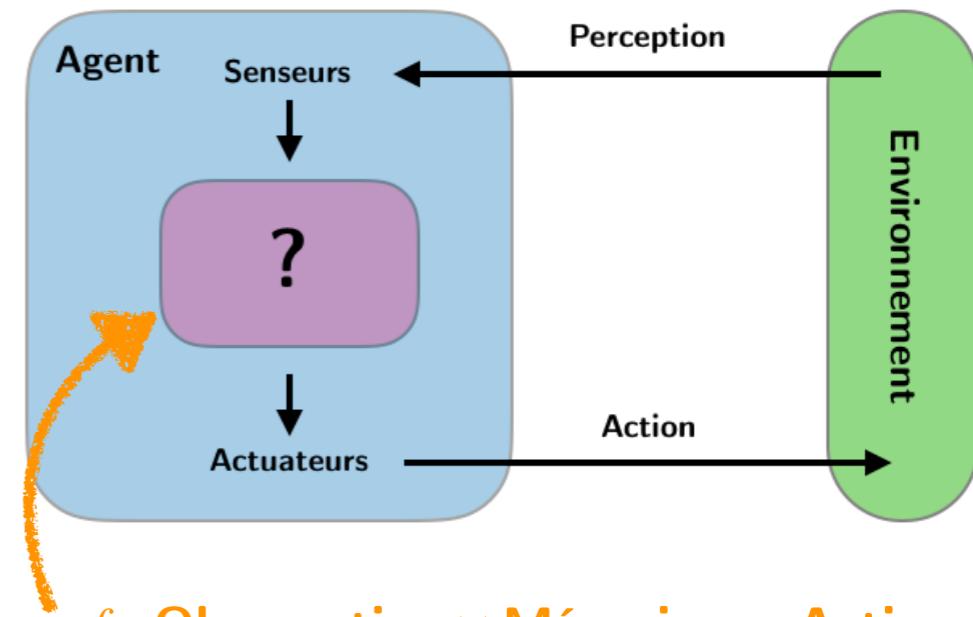
Inconvénient: ne considère pas les conséquences des actions faites

Exemple (Pacman)

Observation (ou perception): la grille du jeu

Mémoire: aucune

Action: prendre la direction du point le plus proche



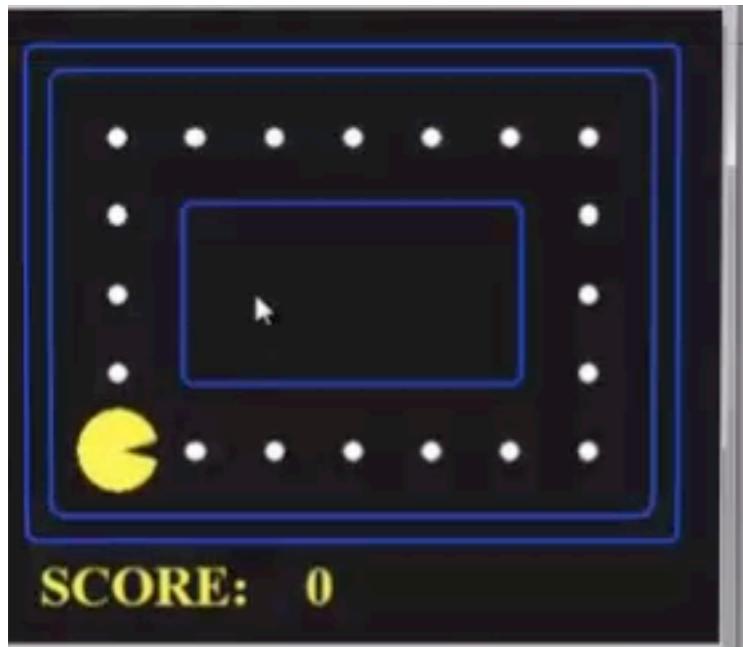
$f : \text{Observation} \times \text{Mémoire} \rightarrow \text{Action}$



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

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

Principe: différents scénarios sont simulés, jusqu'à ce qu'une séquence favorable soit trouvée

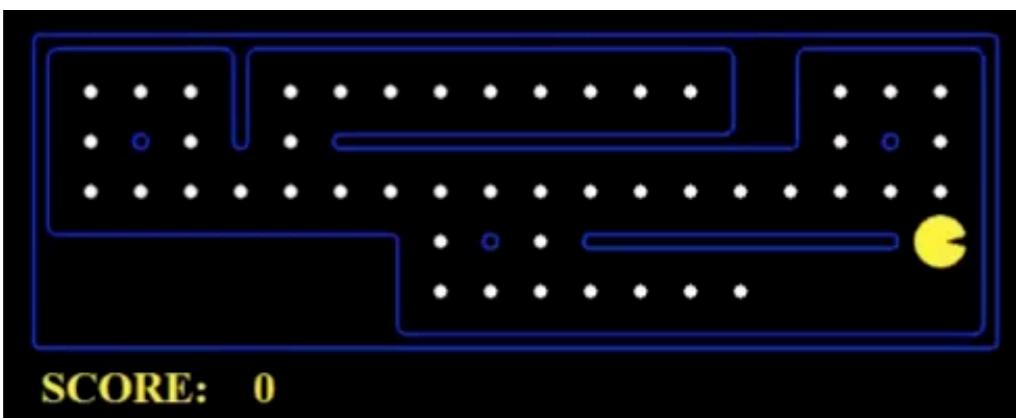
Défi principal: dérouler efficacement les différentes séquences d'actions possibles

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 pour Pacman: manger tous les points



Solution: capture des points en un minimum de mouvements

Nombre d'évaluations: environ 3000 actions ont été évaluées

Temps d'exécution: environ 10 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)

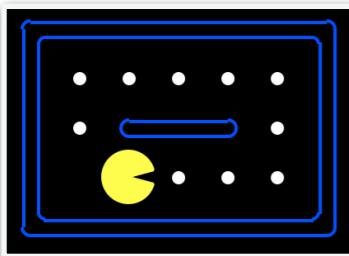
A : un ensemble d'actions

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

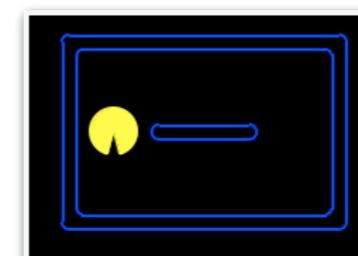
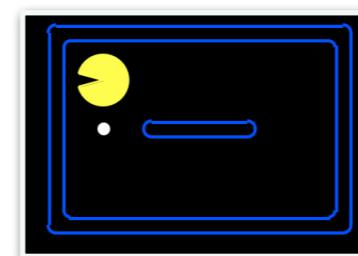
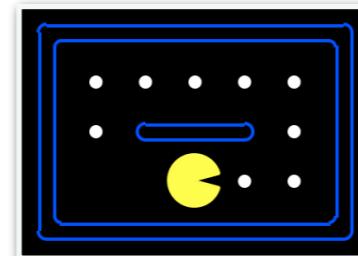
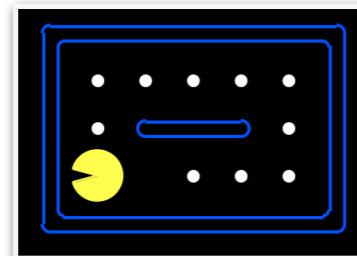
$C : (S \times A \rightarrow \mathbb{R})$: une fonction décrivant le coût d'effectuer une action à partir d'un état

Ensemble d'états

Définition: toutes les configurations possibles que peut prendre l'environnement



Initial



Final

...

Ensemble d'actions

Définition: toutes les actions possibles que peut prendre l'agent

Cas de Pacman: déplacement à gauche, à droite, en haut, ou en bas

Formalisation d'un problème de recherche

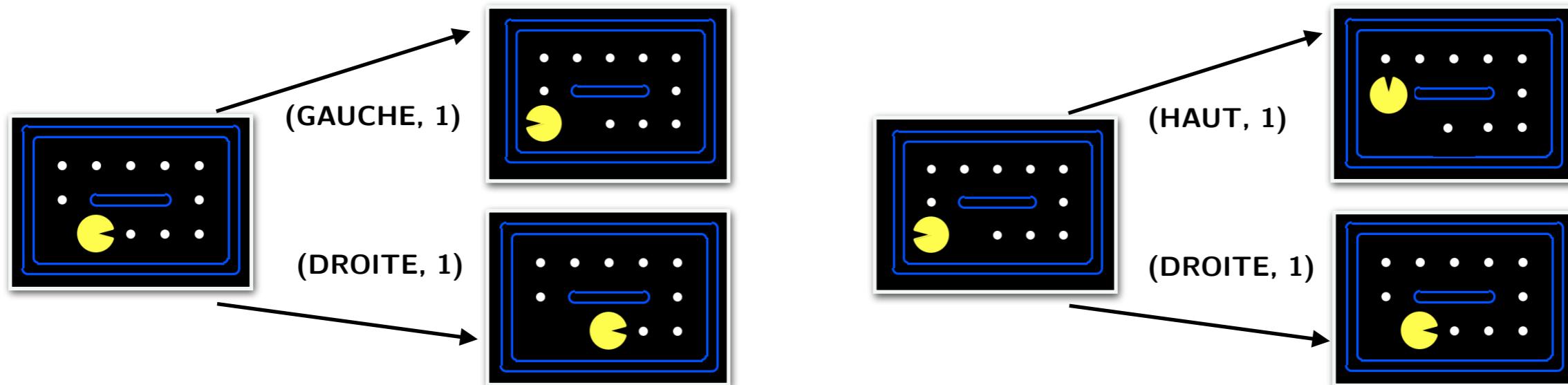
Fonction de transition et de coût

Fonction de transition: indique le nouvel état si une action est faite à partir d'un état spécifique

Fonction de coût: associe une valeur numérique à l'action réalisée sur cet état

Transition de Pacman: le mouvement est effectué et le point est mangé, le cas échéant

Coût de l'action: un coût unitaire est associé à chaque déplacement

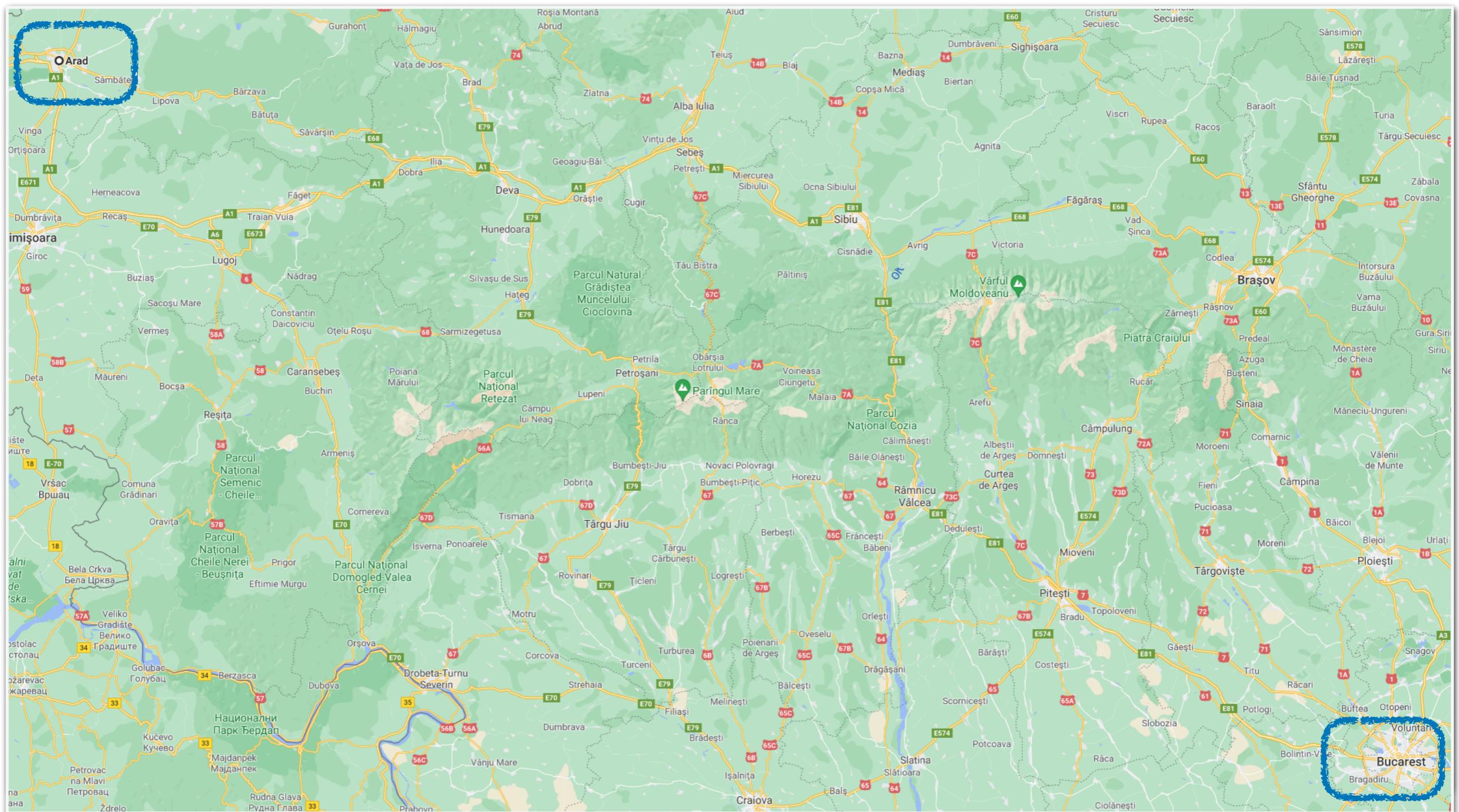


Solution

Définition: 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



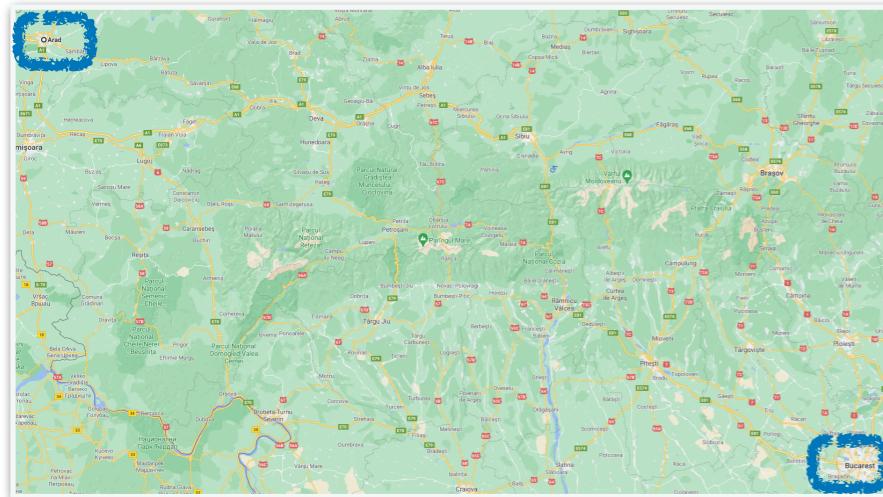
Modélisation d'un problème

Opération consistante à encoder notre problème base d'une formalisation particulière

Exemple: on peut modéliser un problème sur base de la formalisation des slides précédents

Etape 1: le problème modélisé est résolu via un algorithme de résolution

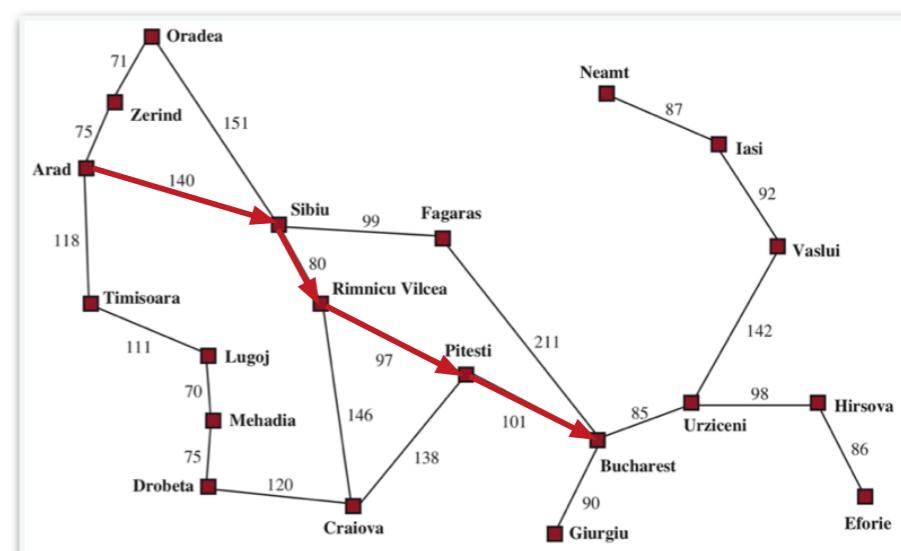
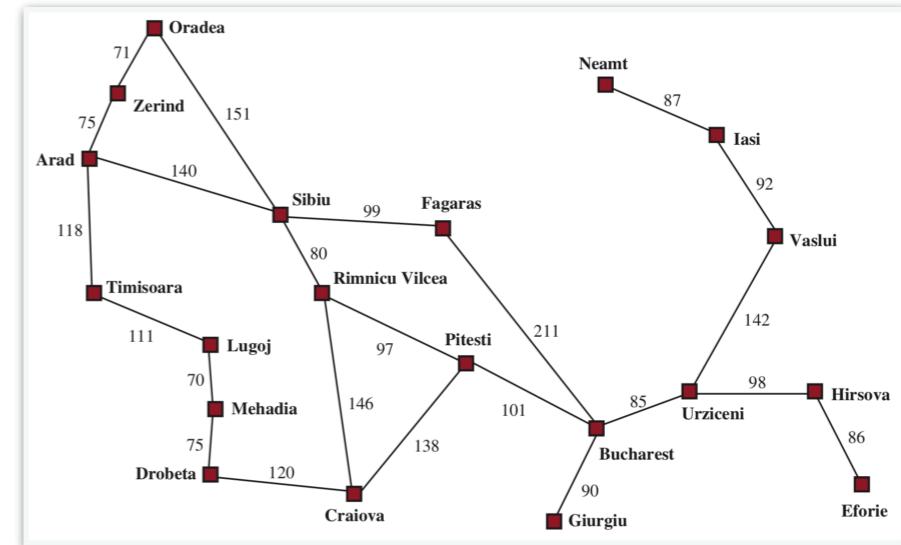
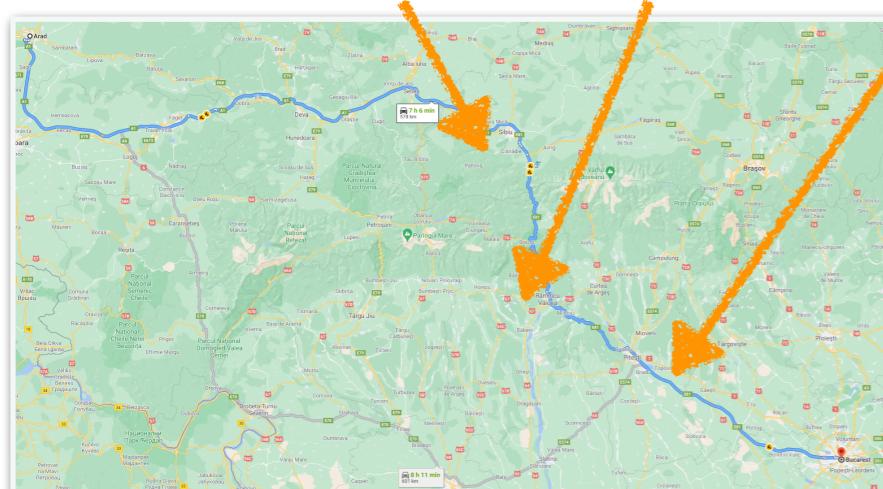
Etape 2: la solution est ré-exprimée comme solution du problème réel



Sibiu

R. Vilcea

Pitesti



Modélisation du monde réel



Niveau d'abstraction

Simplification de la réalité qui est faite en définissant le modèle

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

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

Exemple de la planification de routes

Objectif: atteindre Bucharest à partir d'Arad

Abstraction: considérer un sous-ensemble de villes et de liaisons

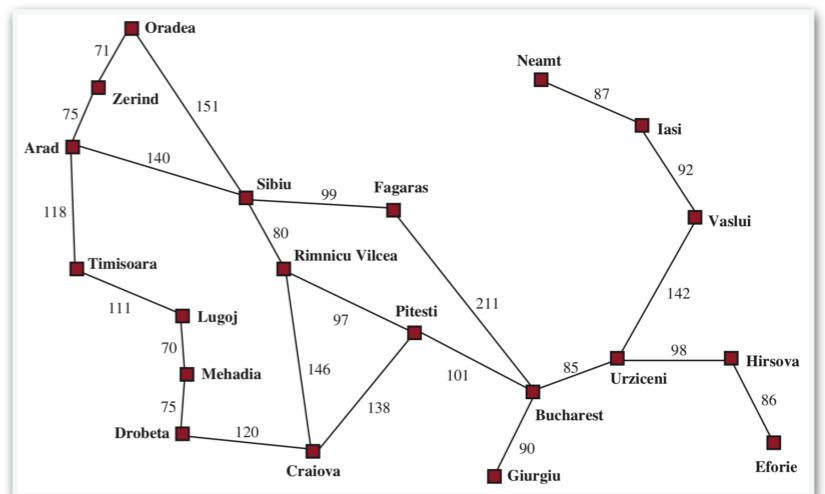
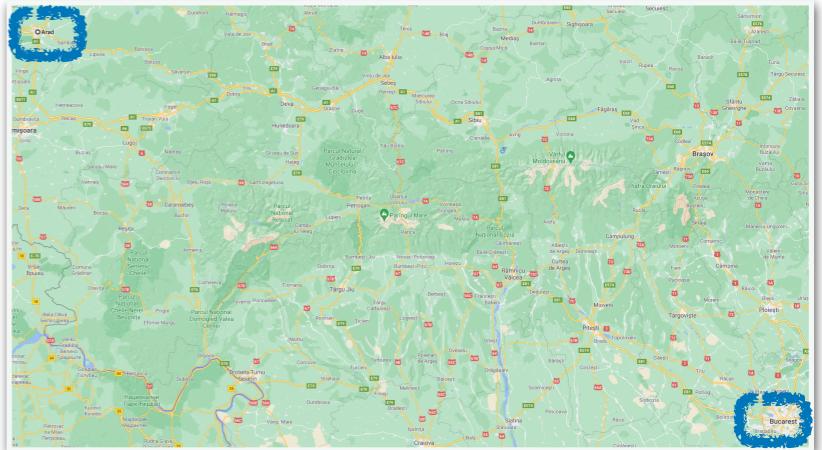
Ensemble d'états: toutes les positions possibles du modèle

Etat initial et final: Arad et Bucharest

Ensemble d'actions: toutes les arêtes adjacentes à notre position

Fonction de transition: effectuer le déplacement sur l'arête

Fonction de coût: pénalisation égale à la distance de l'arête



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

Exemples: plus de points de passages, information temps-réel, présence de travaux, etc.

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

Taille de l'ensemble des états



Comment peut-on quantifier la difficulté à résoudre d'un problème de recherche ?

Réponse: il y a plusieurs facteurs, mais un très important est la taille de l'ensemble des états et d'actions



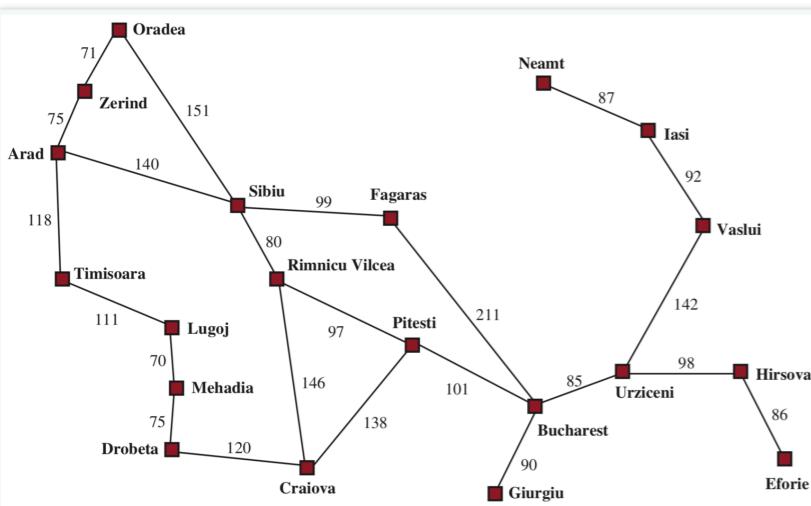
Taille de l'ensemble des états (ou actions)

Nombre d'éléments différents dans l'ensemble des états (ou actions) du problème de recherche

Objectif: trouver une séquence d'actions amenant un état initial à un état final

Intuition: plus le nombre de possibilités est grand, plus il sera difficile de trouver une bonne séquence

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



Combien d'états sont présents dans ce problème de recherche ?

Principe: on a un état par position possible

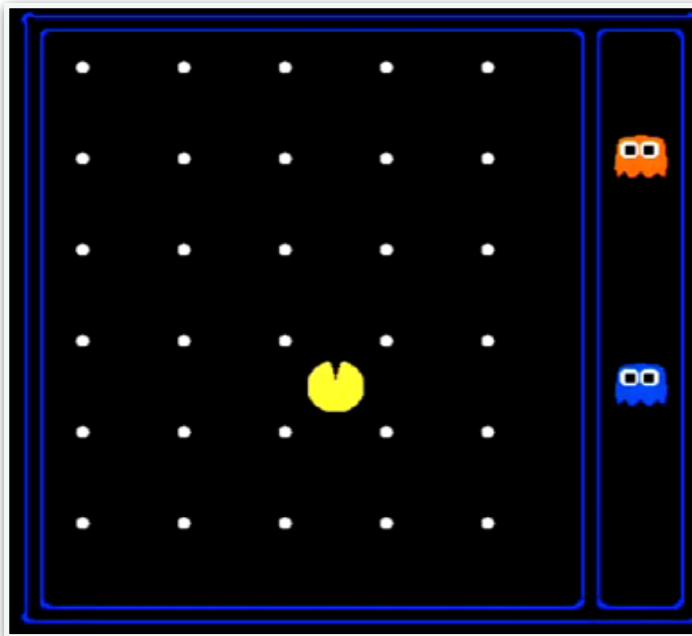
Nombre d'états : 20

Observation: avec plus de villes, j'agrandis mon nombre d'états

Nombre d'actions: au maximum 4 (à Sibiu)

Regardons un cas plus complexe

Taille de l'ensemble des états



Exemple: niveau de Pacman avec 2 fantômes

Positions possibles pour Pacman: 120

Nombre de points à manger: 30

Positions possibles pour un fantôme: 12

? Combien d'états comporte cette représentation globale ?

Besoin: on a besoin d'encoder toutes les configurations de plateau en états

Pacman: défini par sa position (parmi 120 choix)

Pacman : 120 possibilités

Fantôme: défini par sa position (parmi 12 choix)

2 fantomes : 12×12 possibilités

Nourriture: défini par une valeur binaire (mangé ou non)

30 points : $2 \times \dots \times 2 = 2^{30}$ possibilités

$$\text{Nombre total d'états} = 120 \times 2^{30} \times 12^2 = 18,554,258,718,720$$

Mauvaise nouvelle: la taille de l'espace grandit exponentiellement avec le nombre d'informations considérées

En pratique: on est souvent intéressé par avoir une borne supérieure sur le nombre d'états

Intuition: ça nous donne une garantie sur le pire cas qui peut se produire

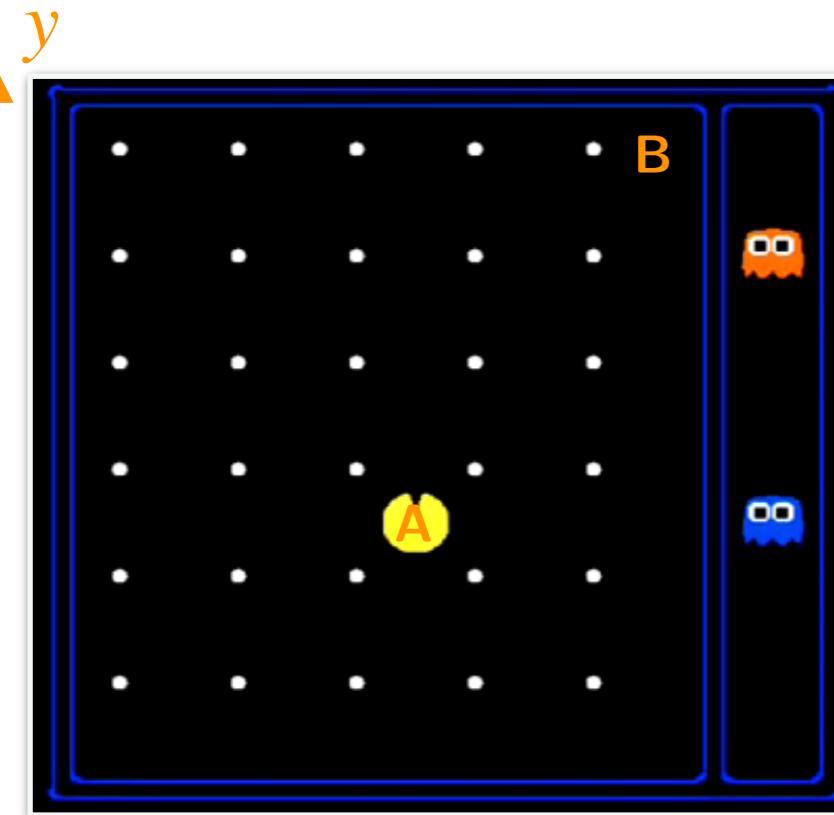
Problème de recherche 1: chemin entre deux points



A t-on besoin de toutes les informations disponibles pour résoudre un problème ?

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

Exemple d'un problème: 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 de Pacman Etat : (x, y)

Etat initial: coordonnées de la position A Etat initial : $x = x_A$ et $y = y_A$

Etat final: coordonnées de la position B Etat final : $x = x_B$ et $y = y_B$

Actions: gauche, droite, haut, bas

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

Haut : $(x, y) \leftarrow (x, y + 1)$

Gauche : $(x, y) \leftarrow (x - 1, y)$

Bas : $(x, y) \leftarrow (x, y - 1)$

Droite : $(x, y) \leftarrow (x + 1, y)$

Détection de l'état final: les coordonnées de Pacman sont sur B



Combien a t-on d'états ?

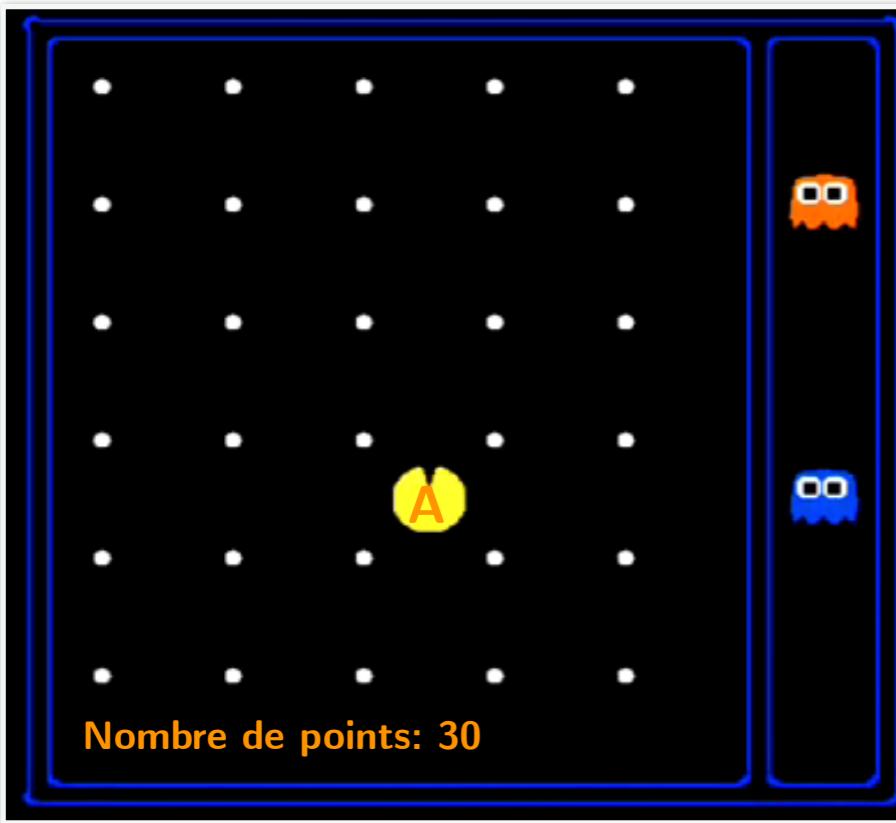
Nombre total d'états = 120

Analyse: seulement les positions de Pacman sont utilisées (et non les fantômes ou les nourritures)

Conclusion: toutes les informations de l'environnement global ne sont pas nécessaires

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

Problème de recherche 2: manger tous les points



Problème: trouver le chemin le plus court pour manger tous les points

?

Comment pourrait-on modéliser ce problème ?

Ensemble d'états:

- (1) Positions possibles de Pacman - celles des points sont fixes
- (2) Indication binaire pour chaque point (p), indiquant s'il est mangé

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

Etat final: valeur de 1 pour tous les points (Pacman non important)

Actions: gauche, droite, haut, bas

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

Etat : $(x, y, p_1, p_2, \dots, p_n)$

?

Combien a t-on d'états ?

Etat initial : $(x_A, y_A, 0, 0, \dots, 0)$

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

Etat final : $(_, _, 1, 1, \dots, 1)$

?

Quelles informations devraient être contenues dans un état ?

(1) Les informations nécessaires pour construire la fonction de transition (et qui changent de valeur)

(2) Les informations nécessaires pour tester si un état est final

Remarque: la position des points n'est pas considérée dans l'état car c'est une valeur fixe

Valeur fixe: valeur utilisée pour les transitions mais non considérée dans la recherche d'une solution

Représentation du problème par un graphe des états



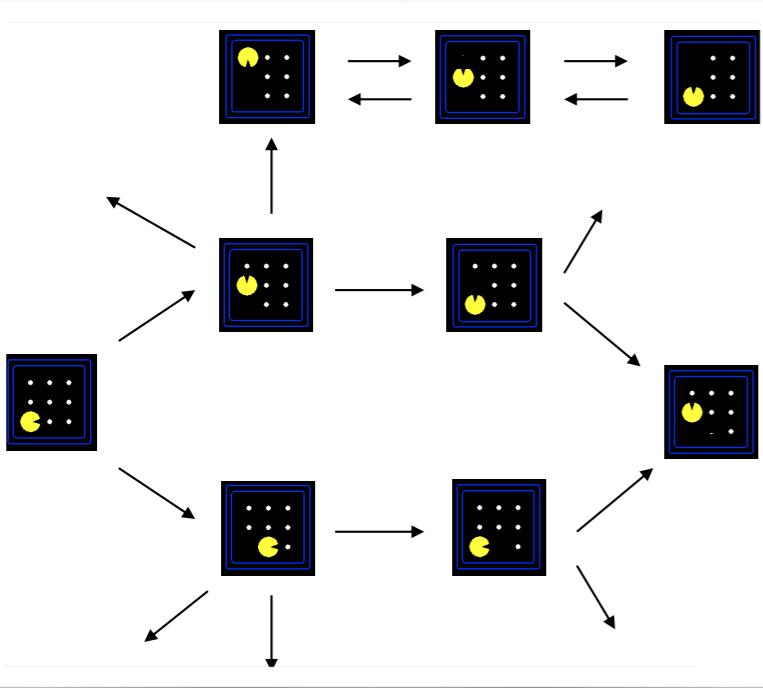
Comment résoudre un problème de recherche ?

Première étape: on va représenter de manière structurée le problème de recherche



Graphe des états

Représentation d'un problème de recherche en un graphe, en liant tous les états par leurs actions



Noeud du graphe: un état possible

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

Poids de l'arête: coût de la transition

Propriété: chaque état n'apparaît qu'une seule fois dans le graphe

Avantage: représente explicitement le problème à résoudre

Avantage: cette représentation est générique

Inconvénient: généralement trop gros pour être mis en mémoire

Inconvénient: plusieurs difficultés pour la conception d'algorithmes

Cette représentation va nous servir de base pour concevoir nos premiers algorithmes de résolution

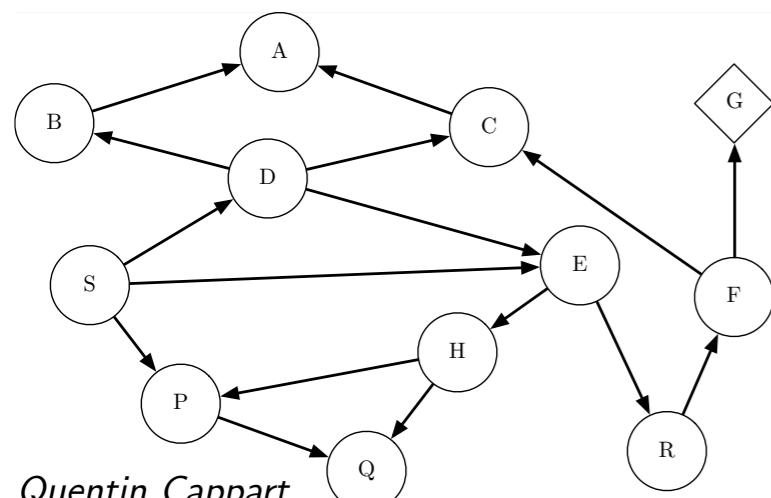


Illustration d'un algorithme de recherche

Ok ! Construisons un premier algorithme pour aller d'un état initial S à un état final G

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

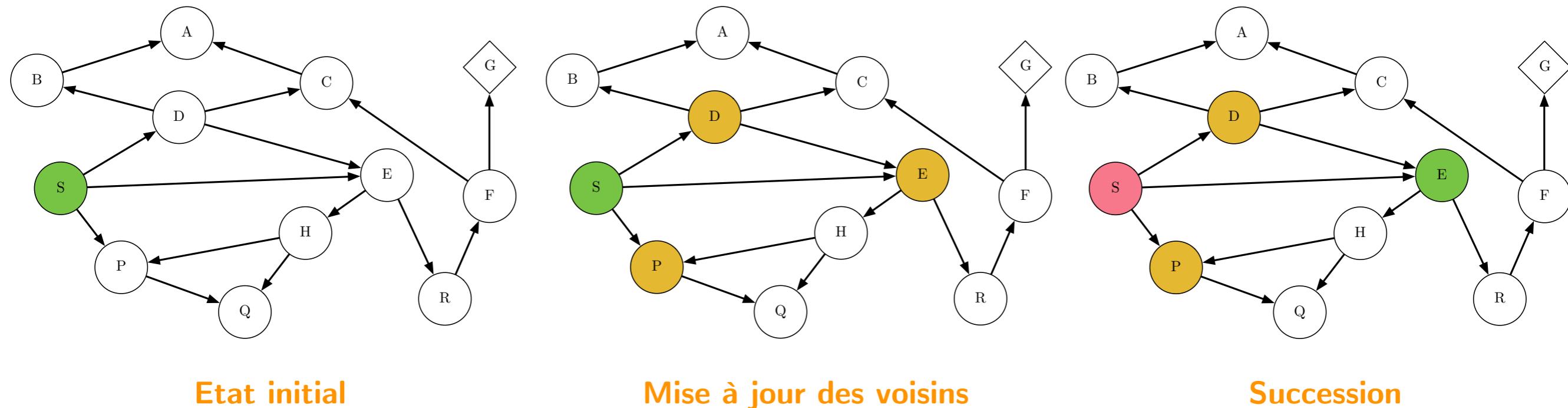
state : notre état actuel (en vert dans l'exemple)

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

fringe : l'ensemble des états atteignables depuis un état visité mais pas encore exploré (en jaune)

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)



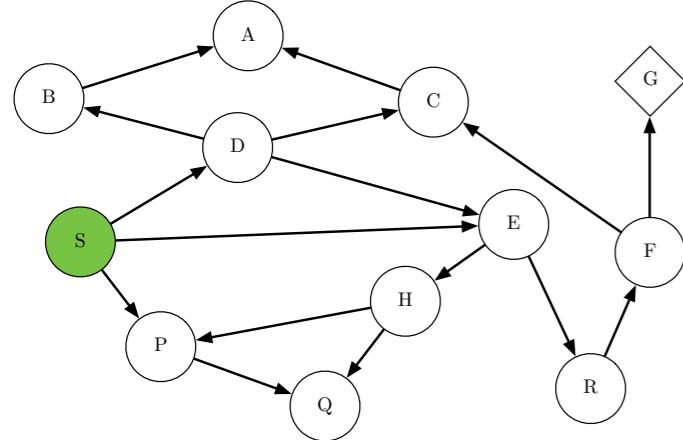
[state : S , fringe : $\langle \rangle$]

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

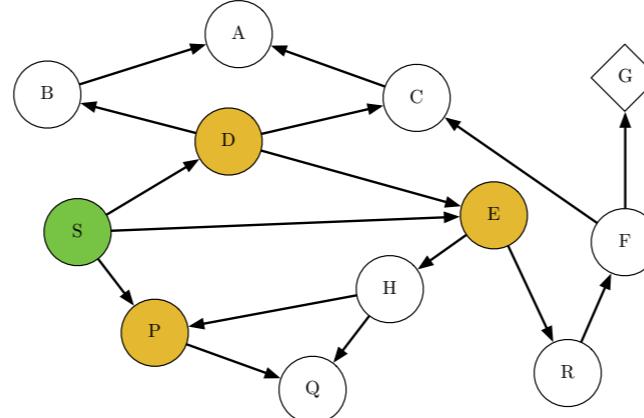
[state : E , fringe : $\langle D, P \rangle$]

Remarque: notez bien que cette représentation est générique, et peut s'appliquer à notre situation Pacman

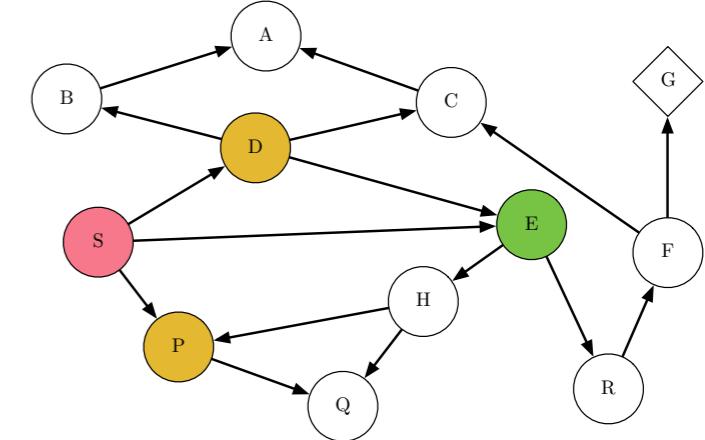
Illustration d'un algorithme de recherche



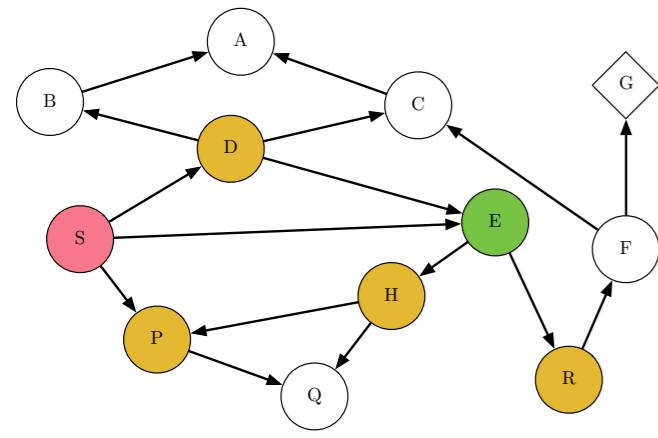
[state : S, fringe : ()]



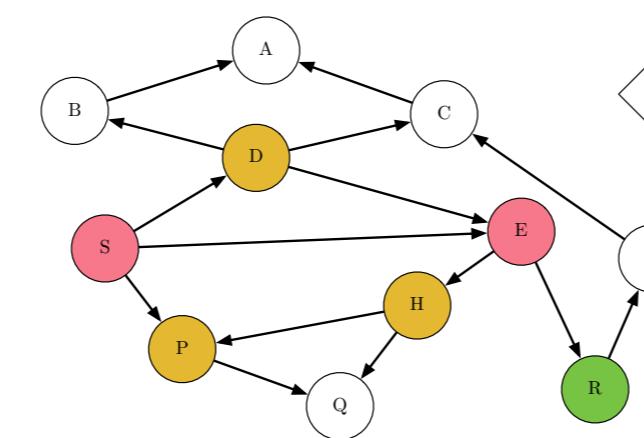
[state : S, fringe : (D, E, P)]



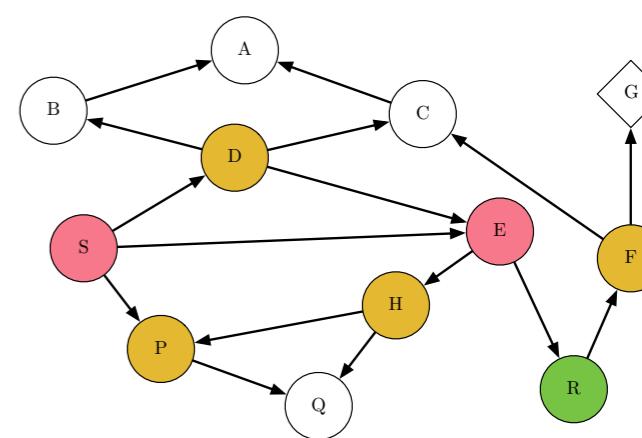
[state : E, fringe : (D, P)]



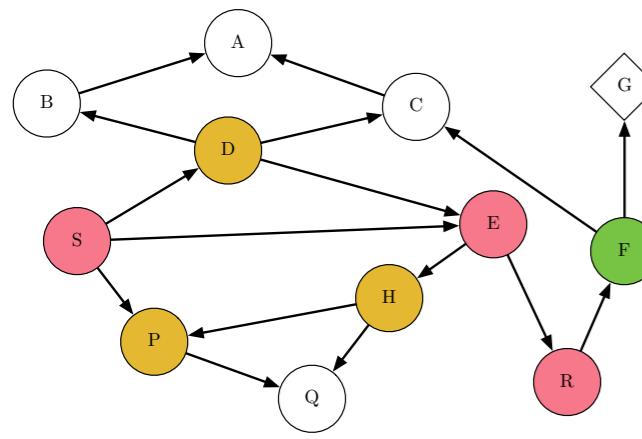
[state : E, fringe : (D, P, H, R)]



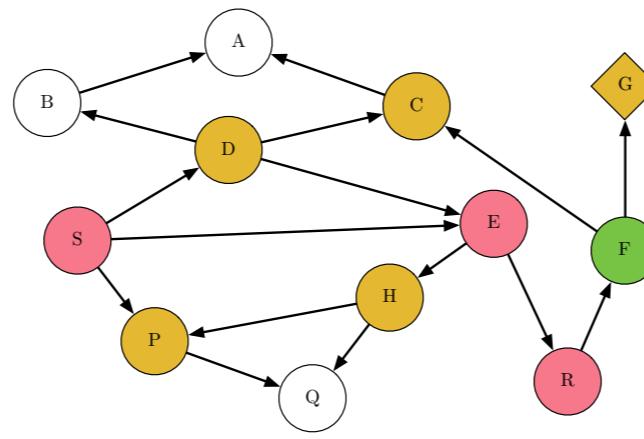
[state : R, fringe : (D, P, H)]



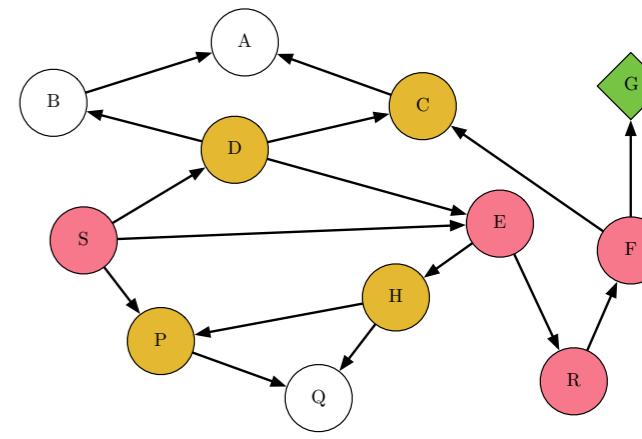
[state : R, fringe : (D, P, H, F)]



[state : F, fringe : (D, P, H)]



[state : F, fringe : (D, P, H, C, G)]



[state : G, path : S → E → R → F → G]

Algorithme de recherche en graphe (*graph search*)

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 :

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

$L = (L \cup C) \setminus s$

$V = V \cup s$

return no solution

Initialisation: commencer à l'état initial du problème à résoudre (P)

Structure de données: garde en mémoire les états déjà visités

Fringe: ensemble des états candidats

Boucle: tant qu'on a des candidats à étendre

Extension d'un état: retirer un noeud candidat de la fringe, et l'étendre

Arrêt: si l'état est final, retourner la solution (chemin du noeud initial au final)

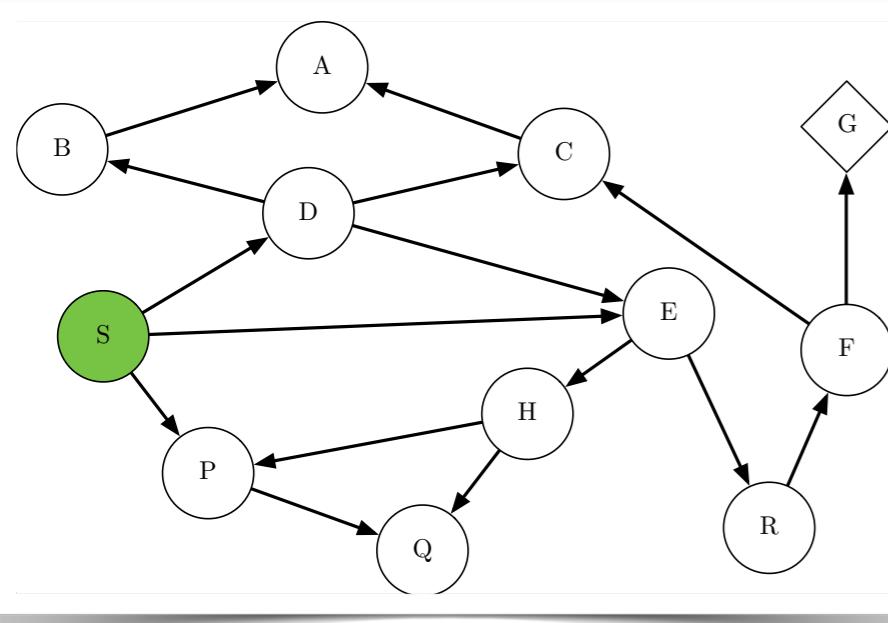
Sinon...

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

... et les ajouter dans la fringe (en empêchant de le régénérer l'état précédent)

Mettre à jour l'ensemble des états visités

Pas de solution: situation où on a pas réussi à atteindre l'état final



On a ainsi notre premier algorithme de résolution !

Bonne nouvelle: offre une excellente base pour plusieurs algorithmes

? Quel est le point de conception majeur de cet algorithme ?

Gestion de la fringe: comment choisir le prochain noeud à étendre ?

Cette question va donner lieu à différents algorithmes de recherches

Algorithme de recherche en graphe (*graph search*)

```
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 :  
             $C = \{c \in \text{succesors}(s, P) \mid c \text{ not in } V\}$   
             $L = (L \cup C) \setminus s$   
             $V = V \cup s$   
    return no solution
```

?

Quelle est la faiblesse majeure de cet algorithme ?

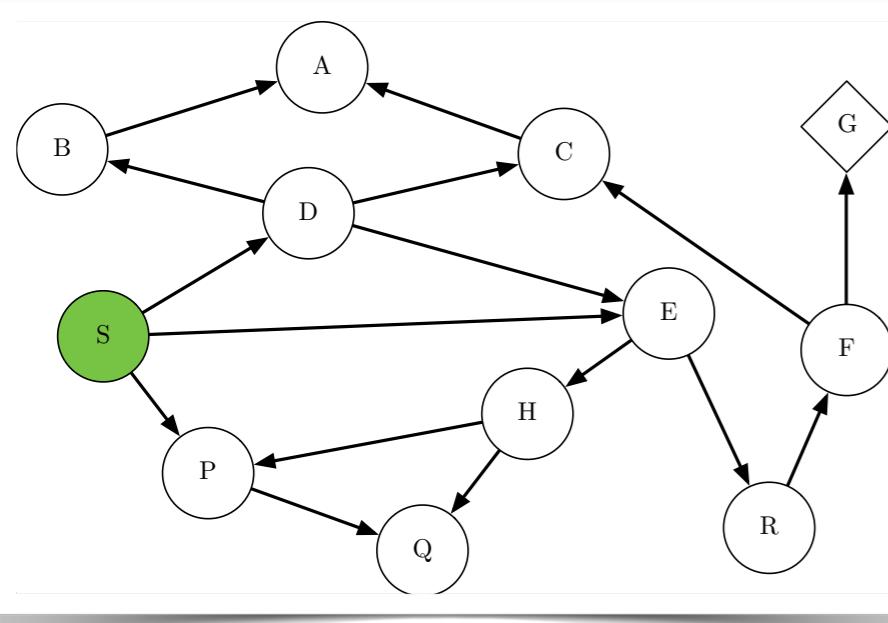
Caractéristique: il demande de retenir tous les états visités (V)

Difficulté: le nombre d'états est souvent extrêmement grand

Nombre d'états pour Pacman : $120 \times 2^{30} = 128,849,018,880$

Problème 1: la mémoire va vite être saturée

Problème 2: le coût d'accès à la mémoire sera plus important



?

Comment peut-on résoudre ce problème ?

Idée: ne pas retenir du tout les états déjà visités

Avantage: réduit drastiquement la consommation mémoire

Inconvénient: engendre un risque de retomber sur un état déjà vu

Recherche en arbre: famille d'algorithmes basés sur ce principe

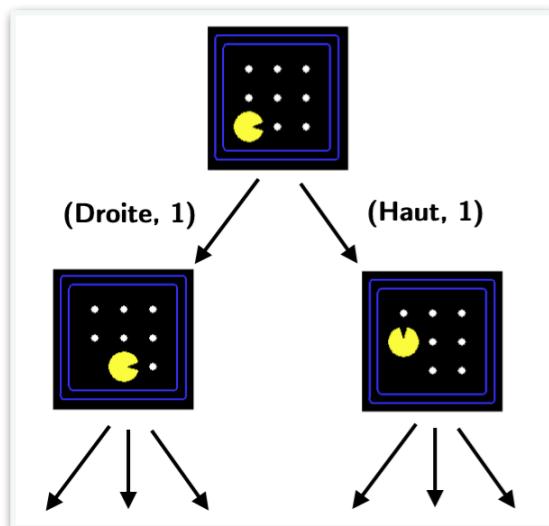
Représentation du problème en arbre



Représentation en arbre

Représentation du problème en arbre, sur base du graphe des états

Différence avec une représentation en graphe: les états peuvent apparaître plusieurs fois



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 (et implicitement les états précédents)

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

Poids de l'arête: coût de la transition

Propriété 1: les états déjà visités ne sont pas mémorisés

Propriété 2: chaque état peut apparaître plusieurs fois dans l'arbre

Propriété 3: l'arbre complet est généralement trop grand pour la mémoire (voire infini)

Bonne nouvelle: un stockage complet n'est pas nécessaire (on *oublie* ce qu'on a vu)

Noeud dans une recherche en graphe: un état

Noeud dans une recherche en arbre: un état et implicitement la séquence d'actions pour y parvenir



Concrètement, qu'est-ce que ça change au niveau de l'algorithme ?

Algorithme de recherche en arbre (*tree search*)

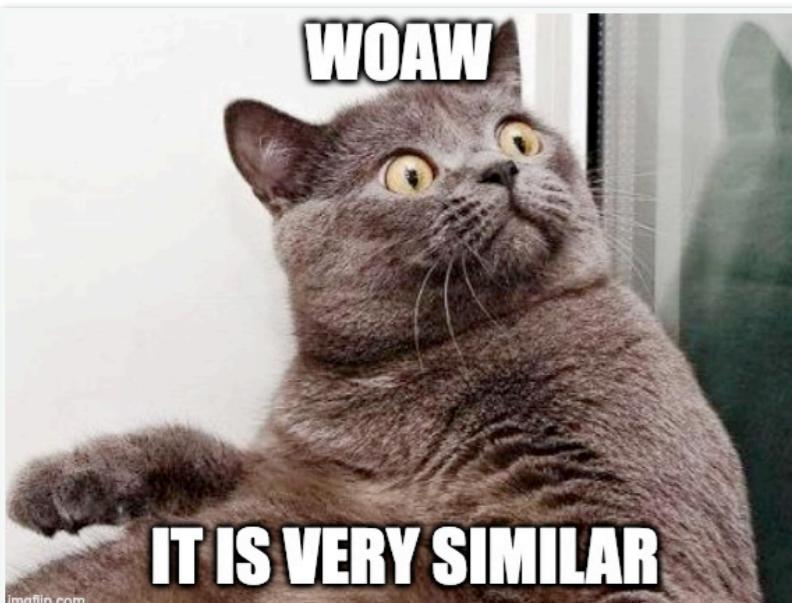
GraphSearch(P) :

```
s = initialState( $P$ )
V =  $\emptyset$ 
L = { $s$ }
while  $L \neq \emptyset$  :
    s = selectAndRemove( $L$ )
    if  $s = goalState(P)$  : return solution
    else :
        C = { $c \in successors(s, P) \mid c \text{ not in } V$ }
        L = ( $L \cup C$ ) \ s
        V =  $V \cup s$ 
return no solution
```



TreeSearch(P) :

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



Grâce à notre formalisation générique, il n'y a que peu de changements

Unique différence: les états déjà visités ne sont pas stockés

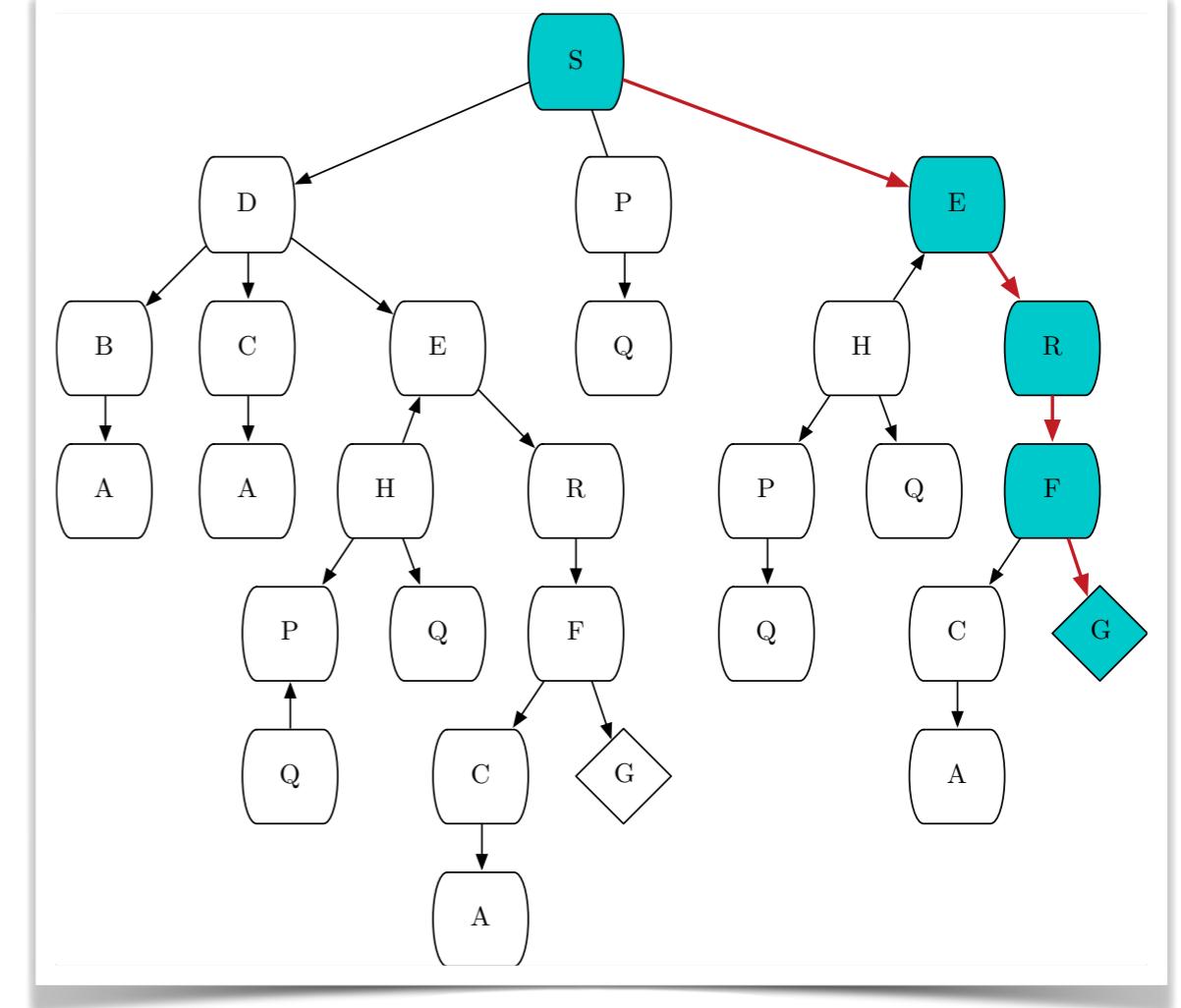
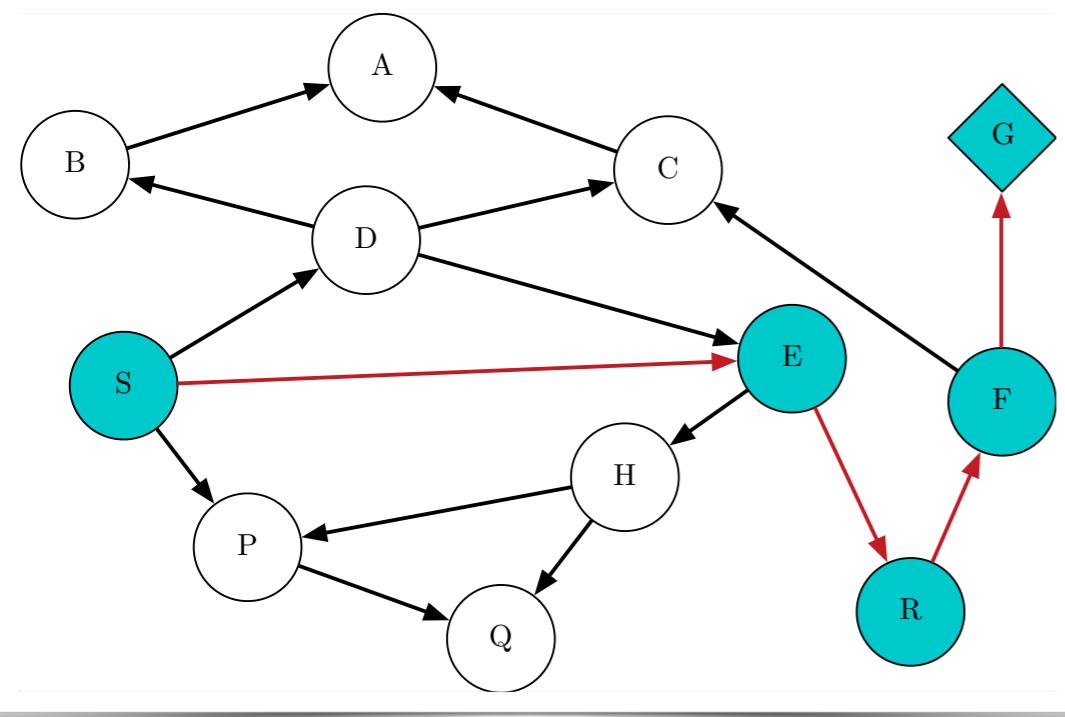
En pratique: on a un risque de retourner sur un état déjà visité

Avantage: permet une meilleure efficacité mémoire

Inconvénient: recherche ralentie par la répétition des états

Choix de conception: toujours la façon de retirer des noeuds de la fringe

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: chaque noeud est relatif au chemin emprunté

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

Remarque 1: ces deux stratégies ont plusieurs variantes permettant d'autres compromis

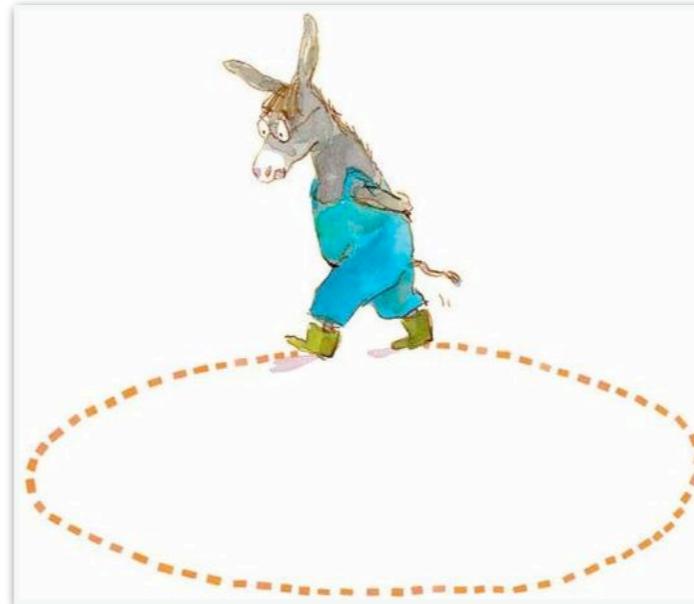
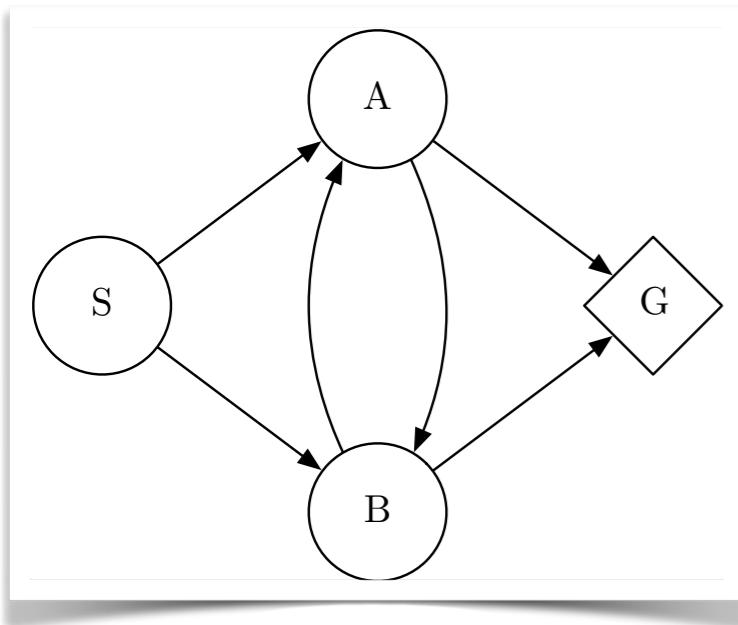
Remarque 2: elles ont également chacune des difficultés spécifiques

Difficulté de la recherche en arbre



Quelle difficulté pratique voyez-vous avec une recherche en arbre ?

Exemple pathologique: situation simple à 4 noeuds



Combien de noeuds comporte l'arbre de recherche relatif ?

Réponse: une infinité... (même si le nombre d'états est fini)

Danger: si on n'y prend pas garde, une recherche dans un arbre peut cycler

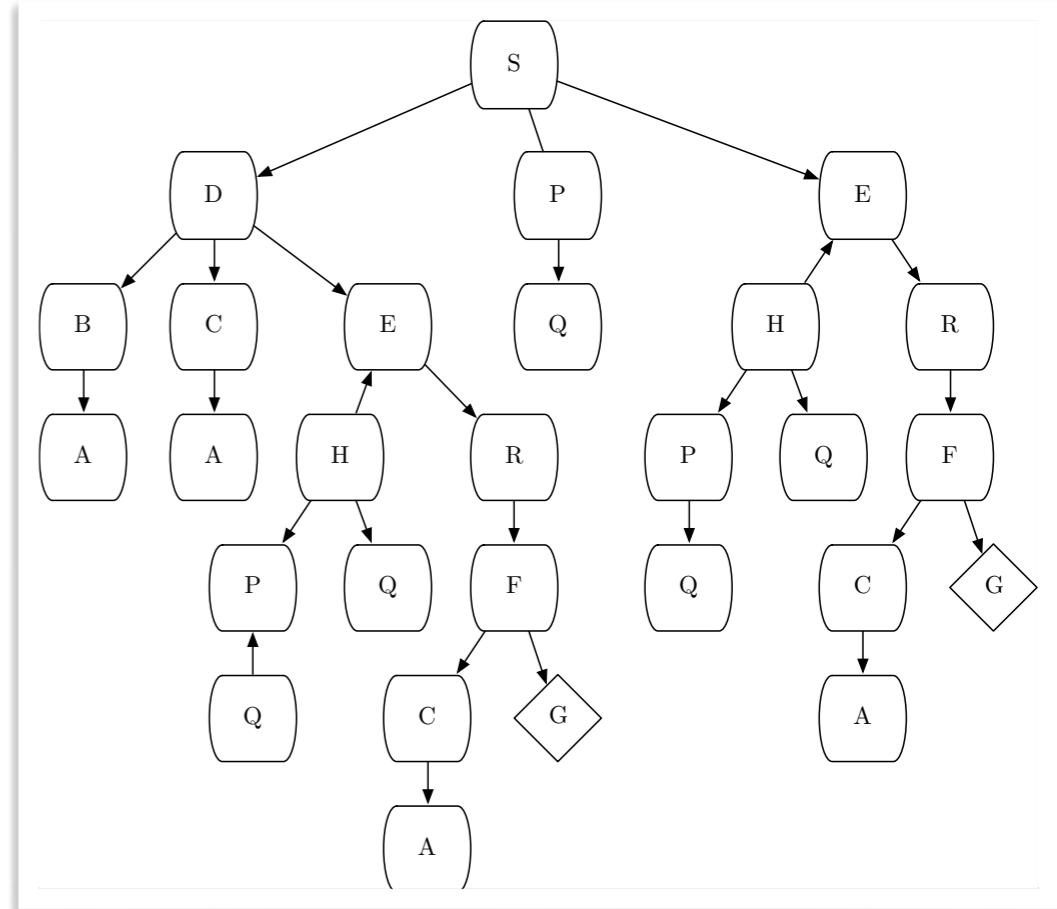
Cycle: $S \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow \dots$

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

Table des matières

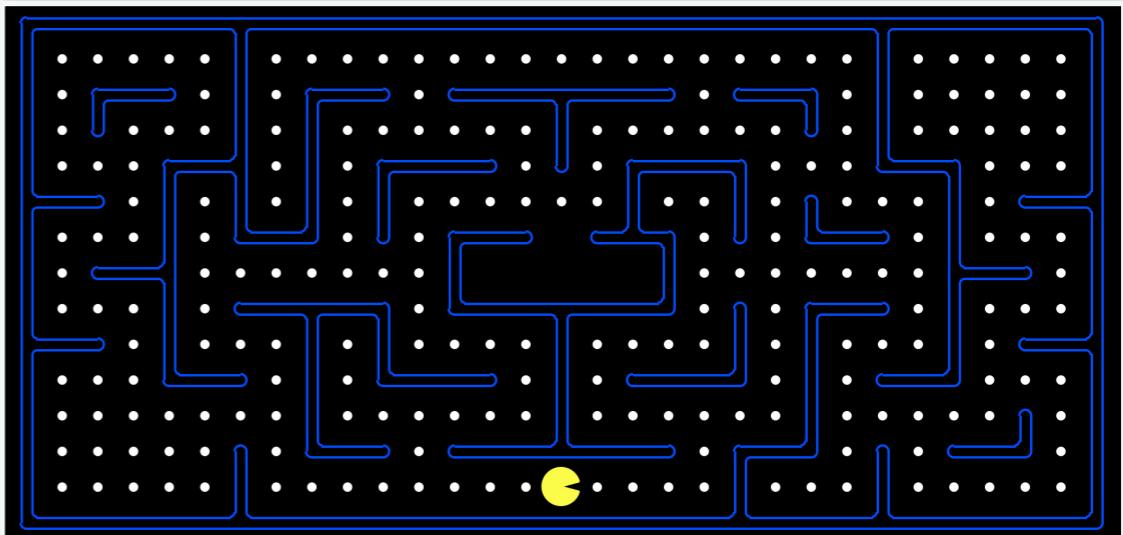
Stratégies de recherche

- ✓ 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

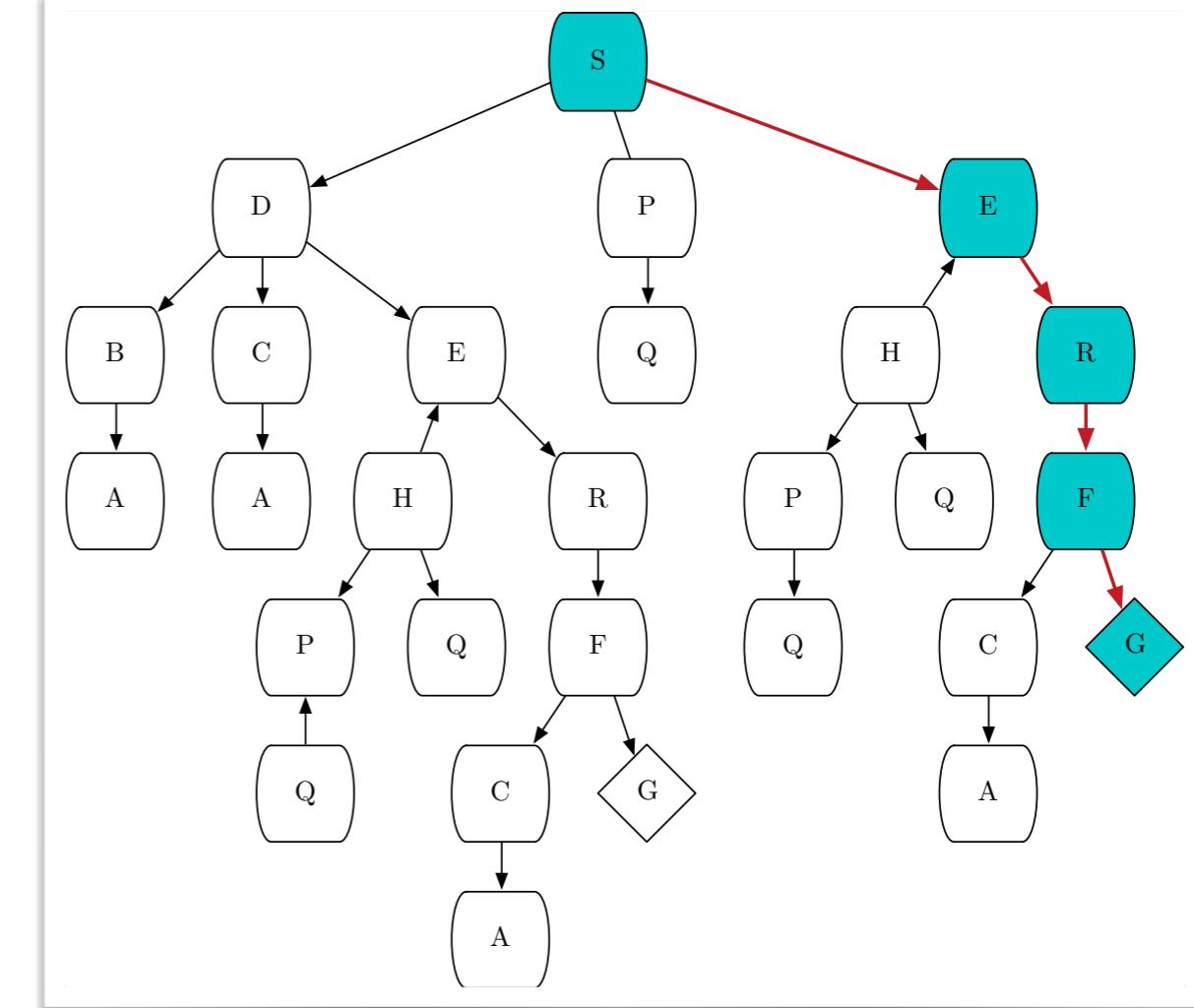
else :

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

$L = L \cup C$

return no solution

$V = \emptyset$



Quel est le point de conception majeur de cet algorithme ?

Gestion de la fringe: comment choisir le prochain noeud à étendre ?

Considération similaire à la recherche en graphe

Répondre à cette question va donner lieu à différents algorithmes de recherche

Critères de performance d'une recherche



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



Critère 1: la complétude (*completeness*)

Principe: garantie que le recherche va trouver une solution (s'il en existe une)

Cas contraire: garantie qu'un échec est reporté



Critère 2: l'optimalité (*optimality*)

Principe: garantie que la recherche retourne la meilleure solution existante

Remarque: cette solution correspond à celle engendrant le moins de coûts



Critère 3: la complexité temporelle (*time complexity*)

Principe: analyse du temps de calcul nécessaire pour exécuter la recherche

Notation: souvent exprimée par la notation asymptotique dans le pire des cas

$\mathcal{O}(n)$

Mesure: souvent calculée en fonction du nombre de noeuds explorés



Critère 4: la complexité spatiale (*space complexity*)

Principe: analyse de la consommation mémoire nécessaire pour exécuter la recherche

Notation: souvent exprimée par la notation asymptotique dans le pire des cas

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

Analyse d'une recherche en arbre

Paramètres

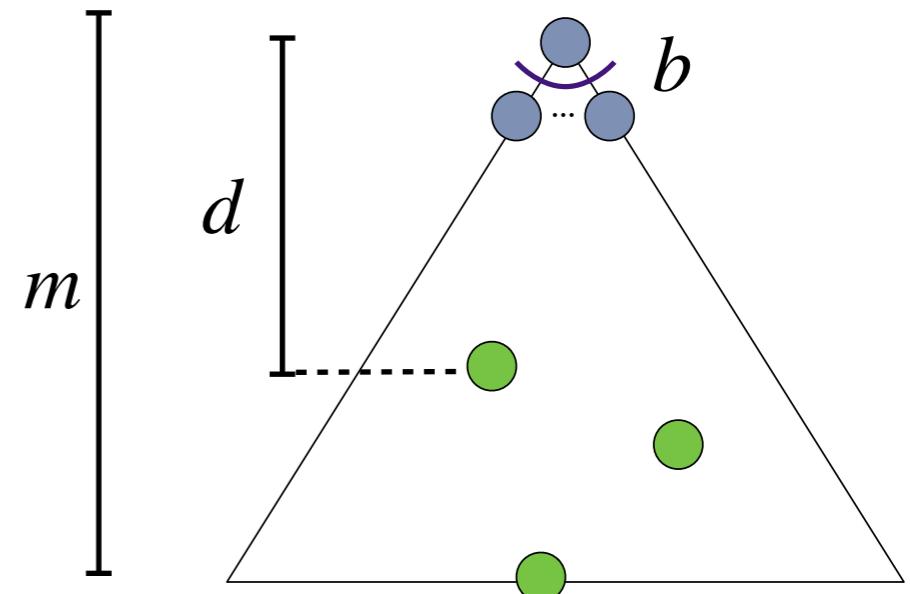
■ : noeud de recherche

● : 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 plus, y a t-il dans un arbre à facteur de branchement b et de profondeur m ?

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

Bonne nouvelle: on peut concevoir des algorithmes qui n'explorent pas tous les noeuds



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

Pour avoir un algorithme concret, on doit indiquer quel noeud retirer en priorité de la *fringe*



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

Algorithme de recherche consistant à retirer systématiquement le dernier noeud ajouté à la *fringe*

Observation: cela revient à retirer en priorité le noeud le plus profond dans l'arbre

Propriété: la *fringe* est une structure LIFO (*last-in first-out* - dernier arrivé, premier sorti)

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

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

$\text{TreeSearch}(P) :$

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

$L = \{s\}$

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



$\text{DepthFirstSearch}(P) :$

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

$L = \text{LIFO}()$

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

while $L \neq \emptyset$:

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

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

else :

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

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

return no solution

Recherche en profondeur - exemple

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 : $\langle A^B, C^D, E^D, P^S, E^S \rangle$]

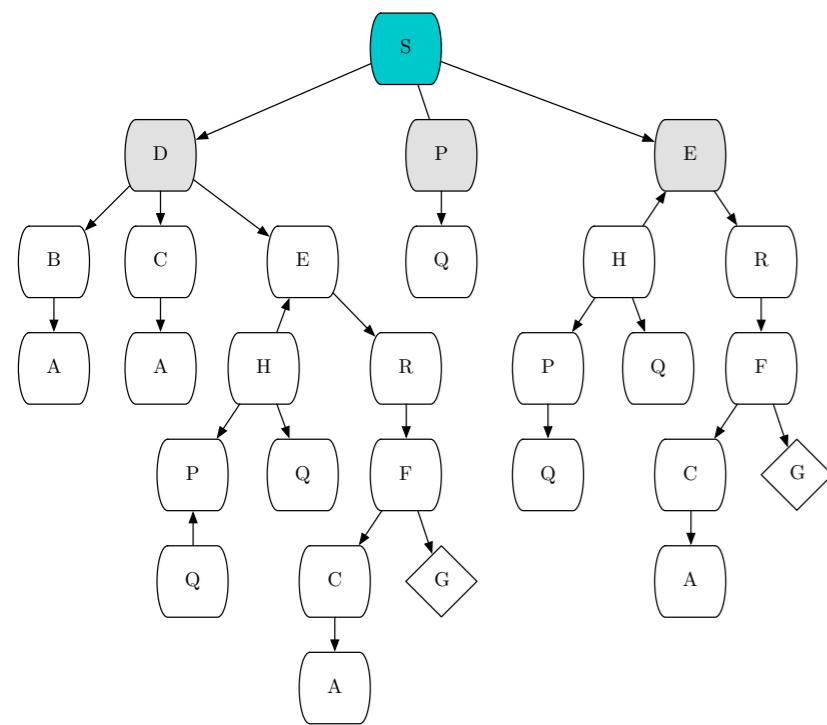
Etape 4 : [state : A^B , fringe : $\langle C^D, E^D, P^S, E^S \rangle$]

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

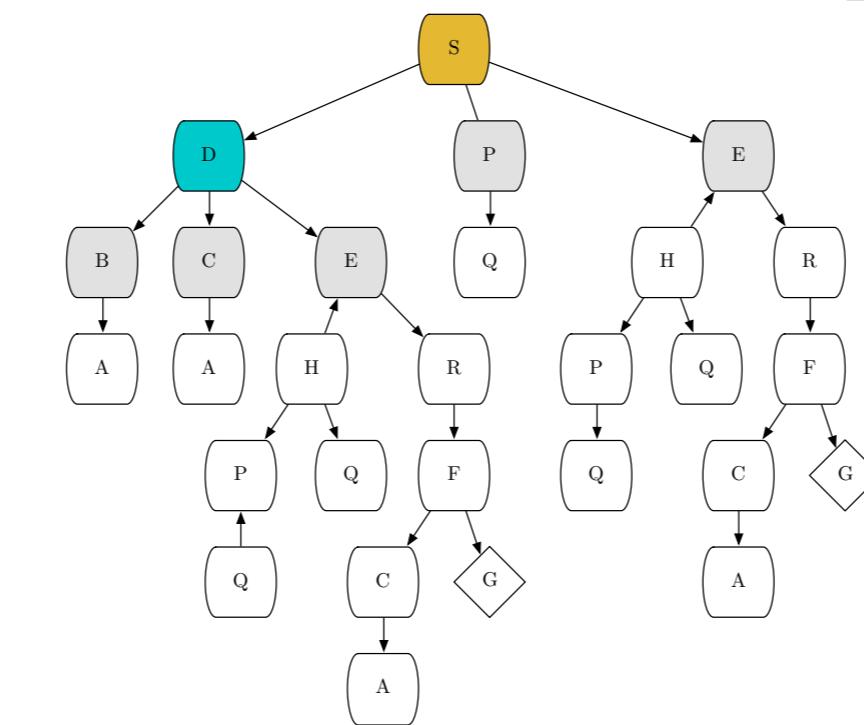
Etape 6 : [state : A^C , fringe : $\langle E^D, P^S, E^S \rangle$]

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

...



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



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

DepthFirstSearch(P) :

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

$L = \text{LIFO}()$

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

while $L \neq \emptyset$:

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

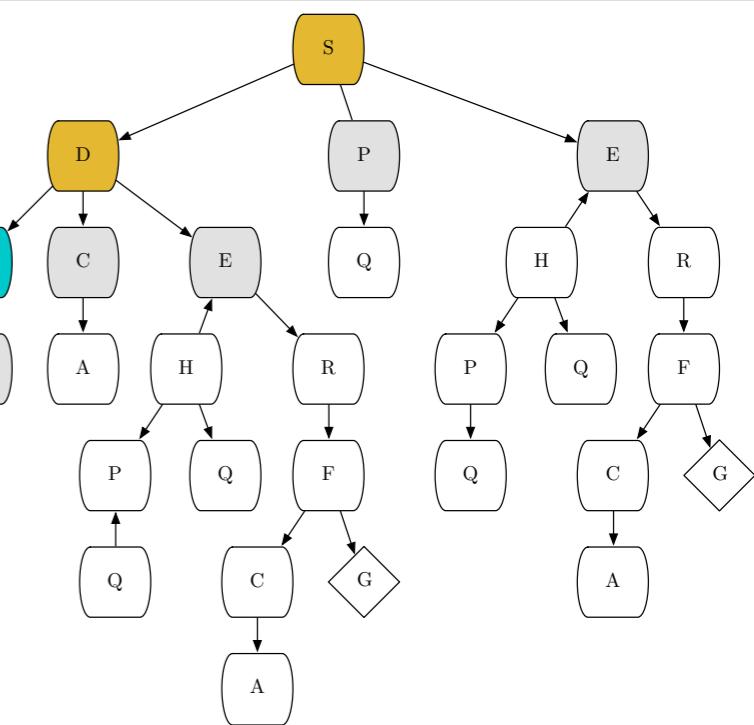
if $s = \text{goalState}(P)$: return solution

else :

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

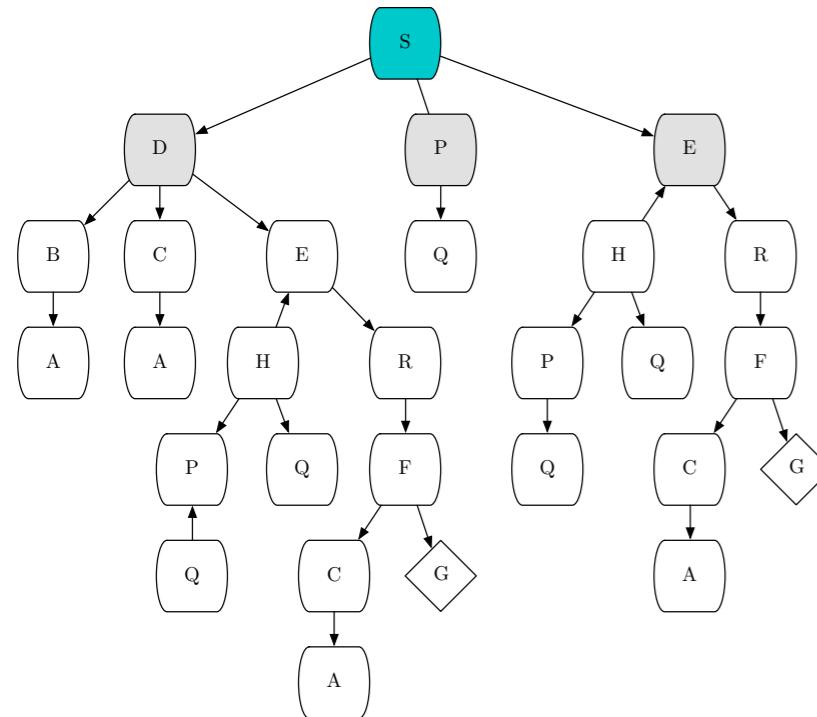
$\text{push}(L, C)$

return no solution

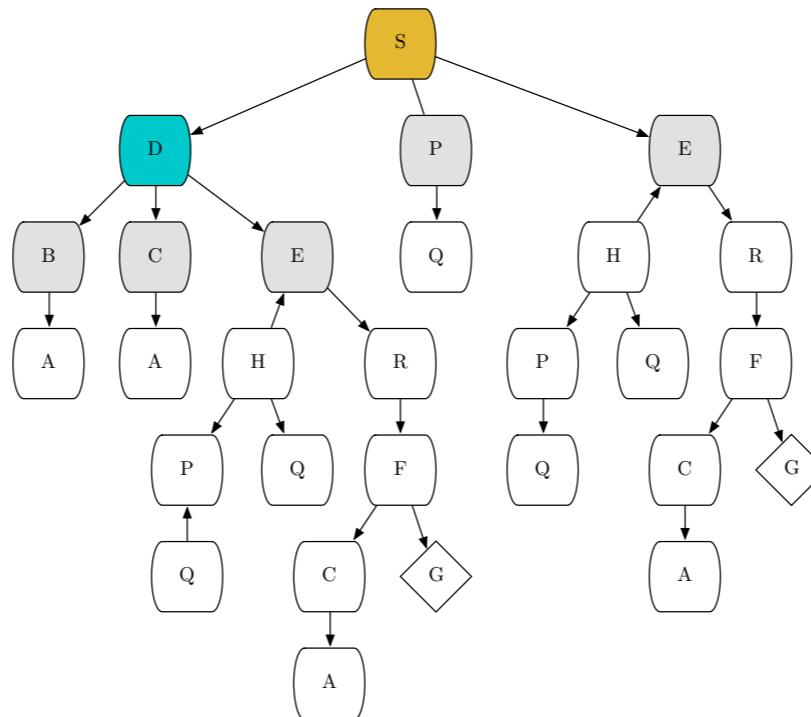


state : B^D , fringe : $\langle A^B, C^D, E^D, P^S, E^S \rangle$

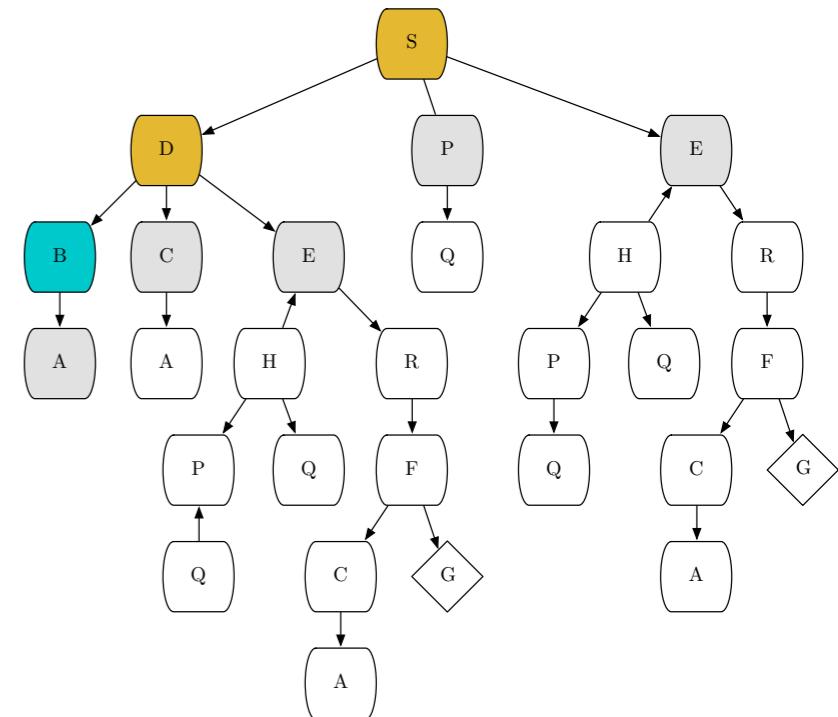
Recherche en profondeur - illustration



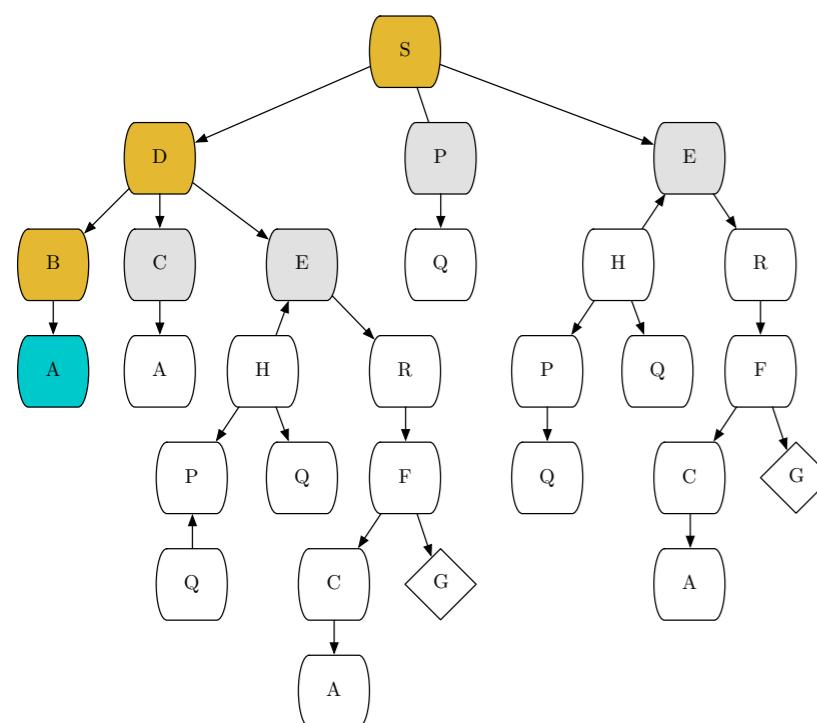
state : S , **fringe** : $\langle D^S, P^S, E^S \rangle$



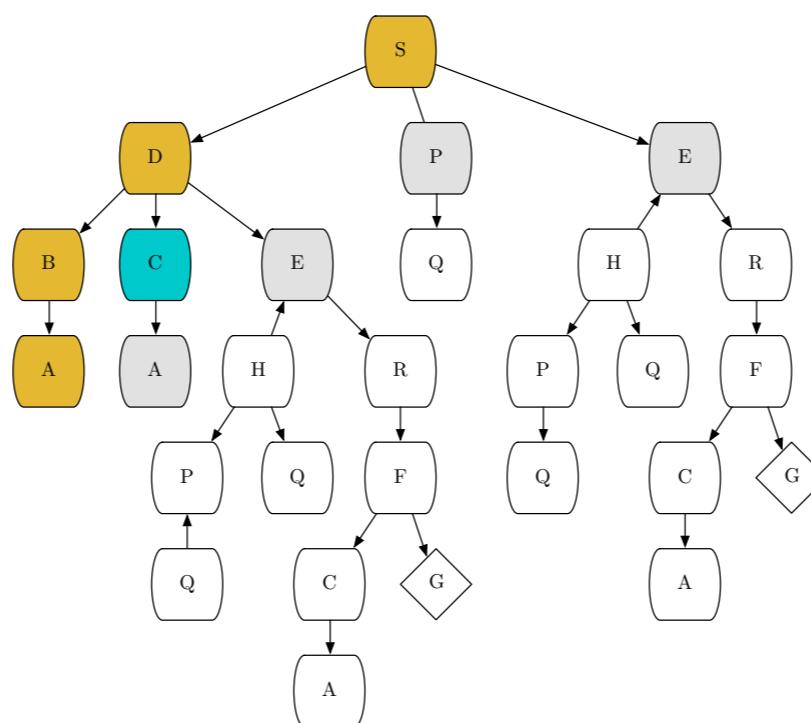
state : D^S , **fringe** : $\langle B^D, C^D, E^D, P^S, E^S \rangle$



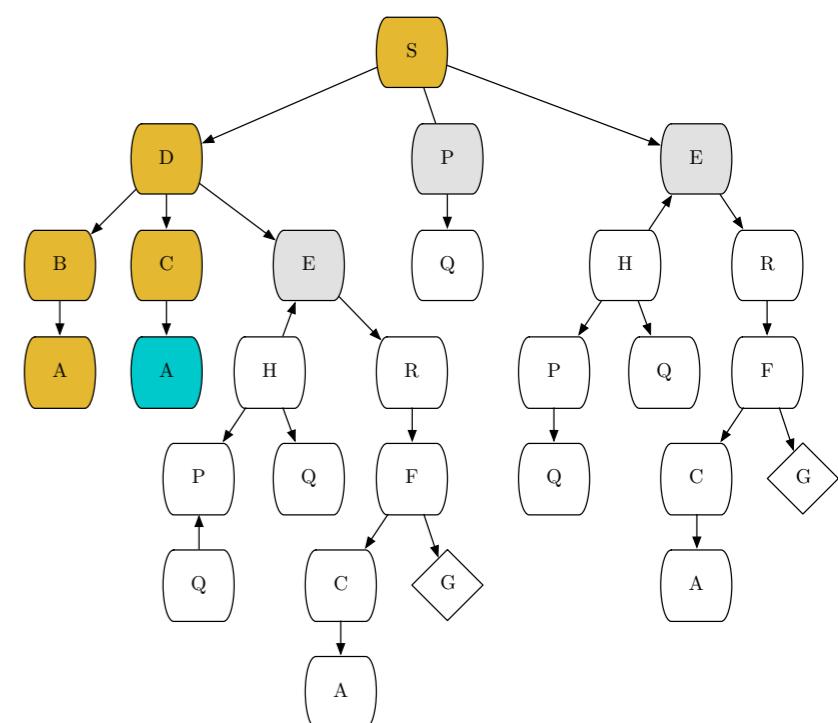
state : B^D , **fringe** : $\langle A^B, C^D, E^D, P^S, E^S \rangle$



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 ?

Critère 1: complexité temporelle (pire des cas)

Pire des cas: tout l'arbre doit être exploré pour trouver une solution

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

Critère 2: complexité spatiale (pire des cas)

Principe: revient à évaluer le nombre maximum de noeuds dans la *fringe*

Fringe: contient seulement les noeuds successeurs des noeuds se trouvant sur le chemin actuel

Observation: les autres noeuds déjà visités (hors du chemin) ne doivent plus être retenus

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

Critère 3: complétude

Observation: la profondeur (m) de l'arbre n'est pas bornée (cycle)

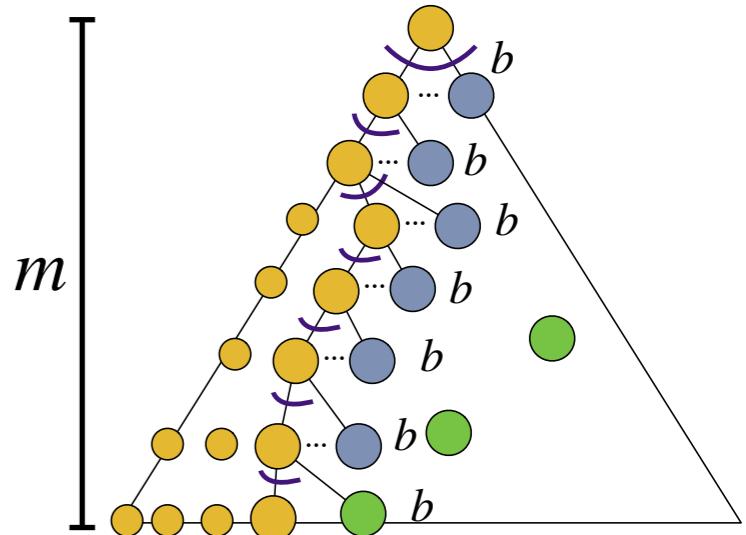
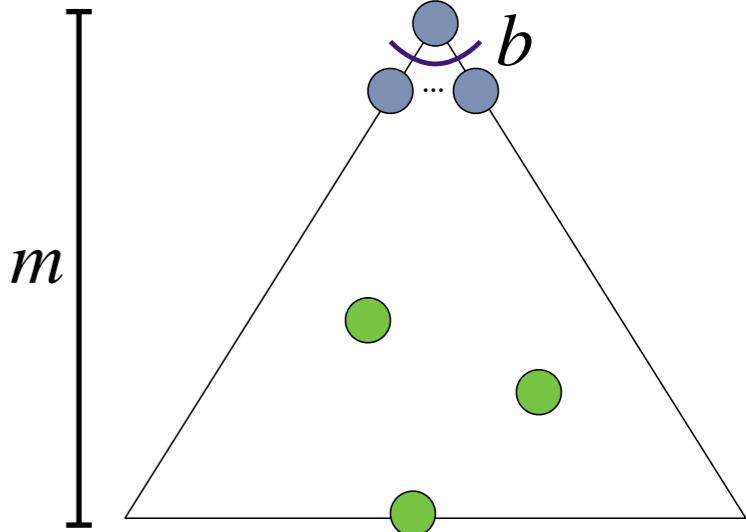
Conséquence: la recherche peut plonger dans une profondeur infinie

Conclusion négative: la recherche est incomplète

Critère 4: optimalité

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

Conclusion négative: la recherche n'est pas optimale



Recherche en largeur (BFS - *breadth first search*)

? Le manque de garanties (complétude et optimalité) du DFS est un problème. A t-on une alternative ?

 **Recherche en largeur (BFS - *breadth first search*)**
Algorithme de recherche consistant à retirer systématiquement le plus ancien noeud de la fringe

Observation: cela revient à retirer en priorité le noeud le moins profond dans l'arbre

Propriété: la *fringe* est une structure FIFO (*first-in first-out* - premier arrivé, premier sorti)

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

$\text{pop}(L)$: retire l'élément le plus ancien de L

Bonne nouvelle: le reste est identique à DFS! (une ligne de code de différence)

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 :

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

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

return no solution



BreadthFirstSearch(P) :

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

$L = \text{FIFO}()$

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

while $L \neq \emptyset$:

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

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

else :

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

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

return no solution

Recherche en largeur (BFS - breadth first search)

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 : $\langle E^S, B^D, C^D, E^D, Q^P \rangle$]

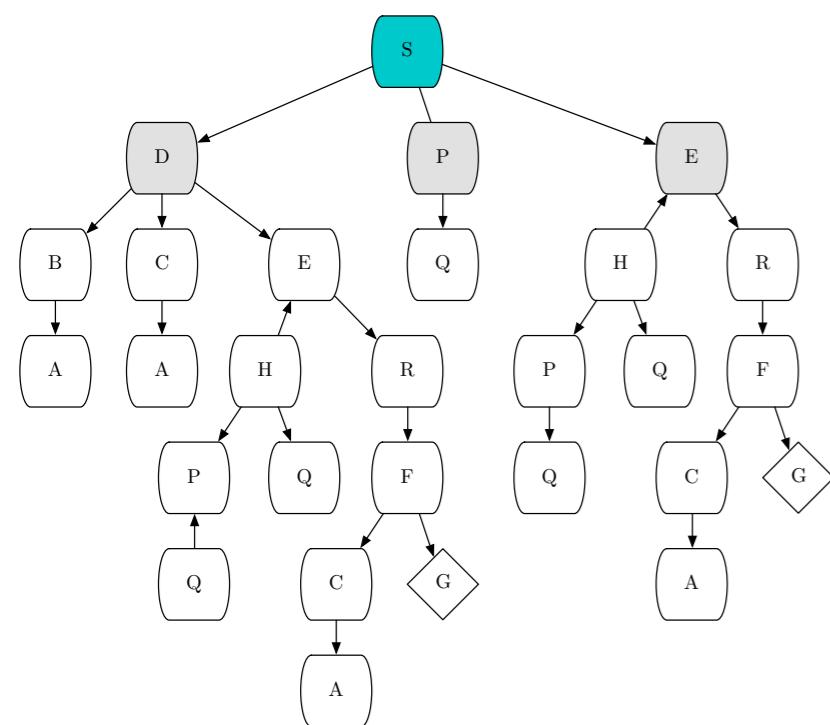
Etape 4 : [state : E^S , fringe : $\langle B^D, C^D, E^D, Q^P, H^E, R^E \rangle$]

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

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

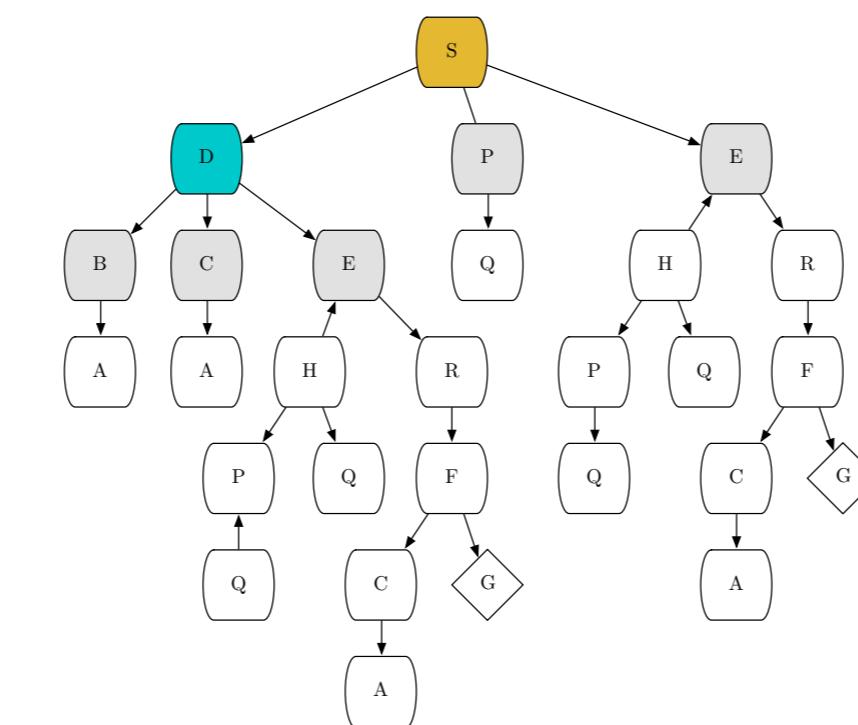
Etape 7 : [state : E^D , fringe : $\langle Q^P, H^E, R^E, A^B, A^C, H^E, R^E \rangle$]

...



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

Quentin Cappart



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

BreadthFirstSearch(P) :

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

$L = \text{FIFO}()$

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

while $L \neq \emptyset$:

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

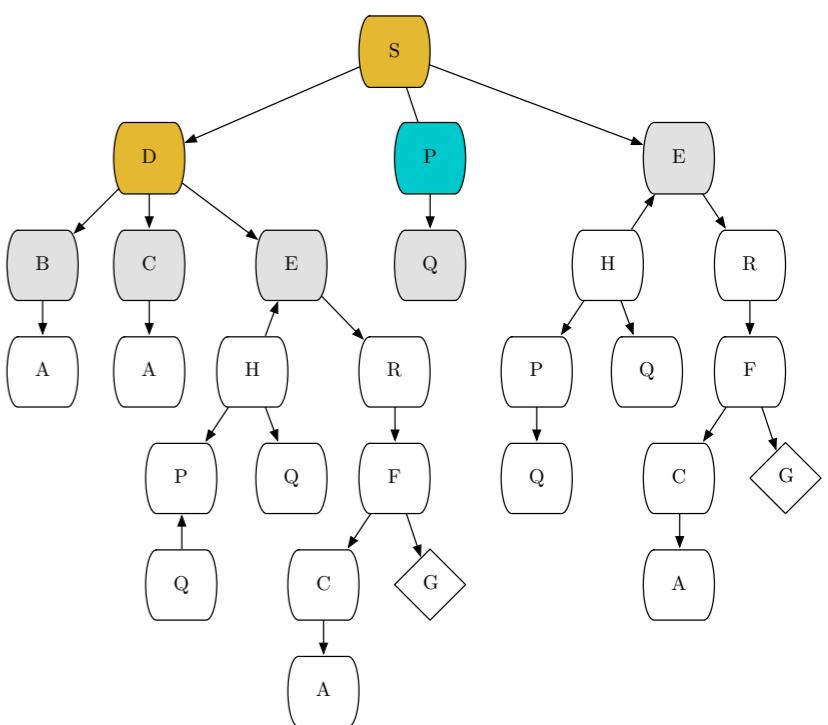
if $s = \text{goalState}(P)$: return solution

else :

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

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

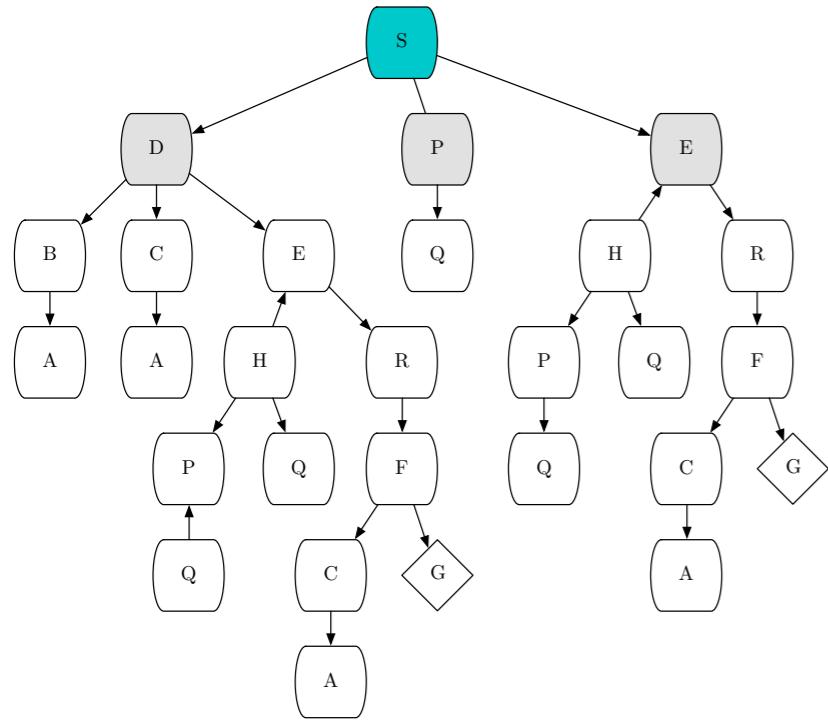
return no solution



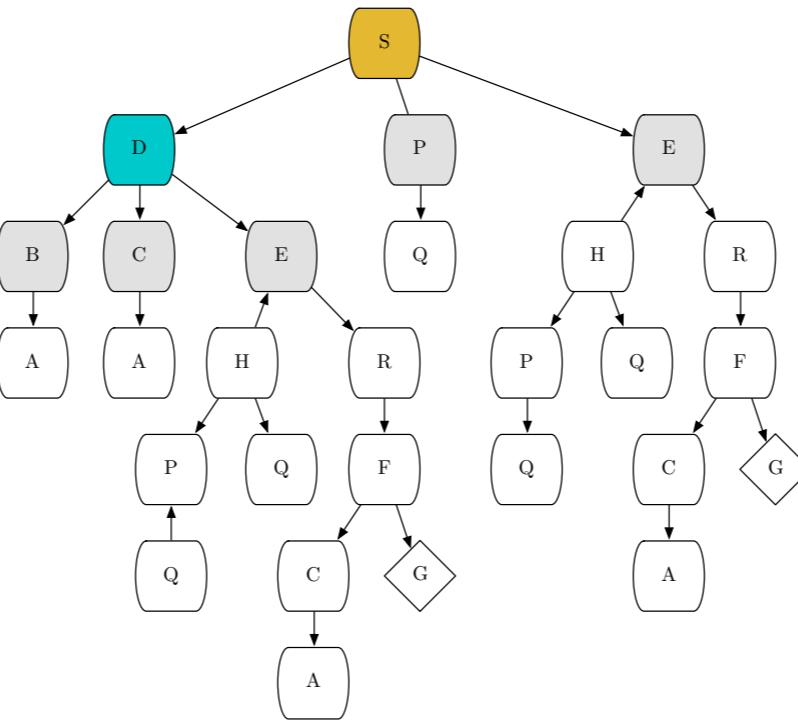
state : P^S , fringe : $\langle E^S, B^D, C^D, E^D, Q^P \rangle$

35

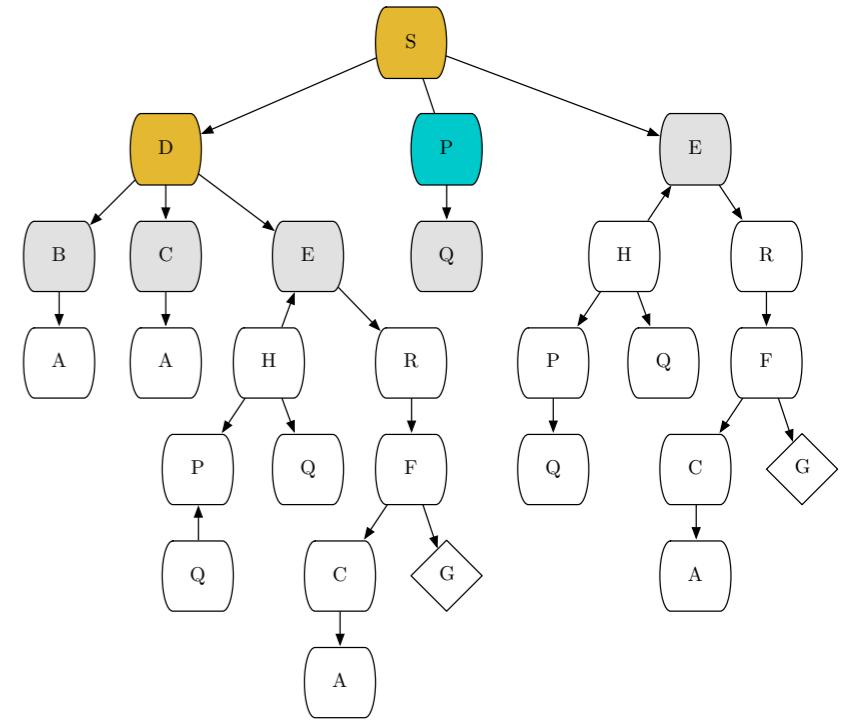
Recherche en largeur - illustration



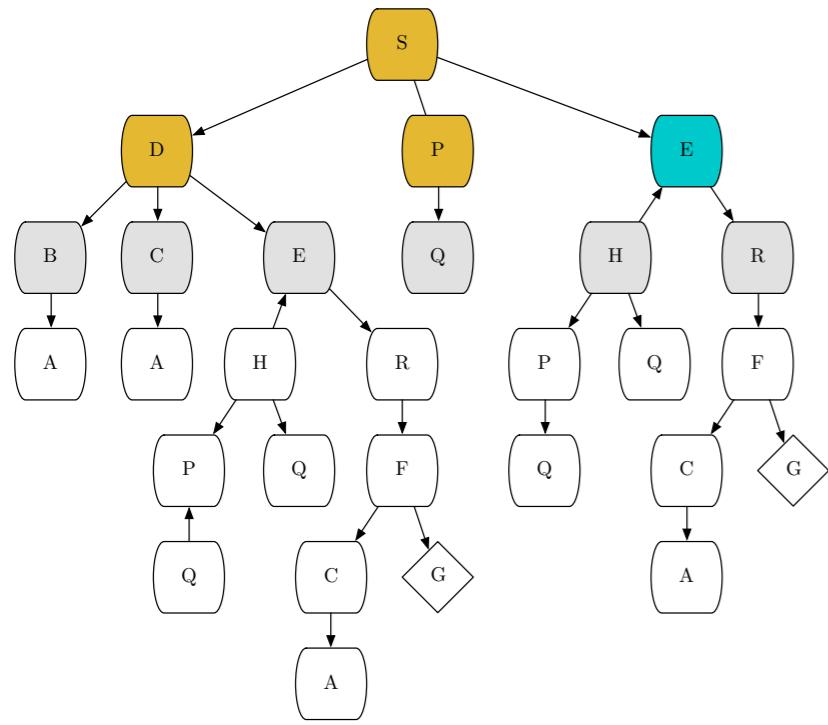
state : S , **fringe** : $\langle D^S, P^S, E^S \rangle$



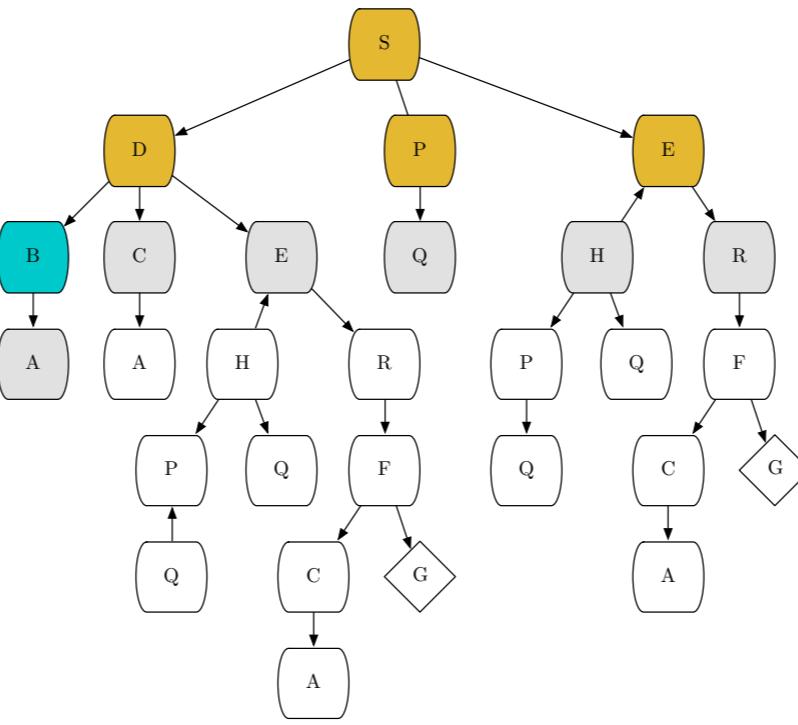
state : D^S , **fringe** : $\langle P^S, E^S, B^D, C^D, E^D \rangle$



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 largeur ?

Critère 1: complexité temporelle

Observation: les niveaux inférieurs au premier état final doivent être explorés

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

Critère 2: complexité spatiale (pire des cas)

Fringe: peut contenir tous les noeuds d'un niveau spécifique de l'arbre

Observation: les autres noeuds visités (niveaux inférieurs) ne doivent plus être retenus

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

Critère 3: complétude

Observation: même en cas de profondeur infinie une solution sera trouvée (s'il en existe une)

Intuition: on explore niveau par niveau

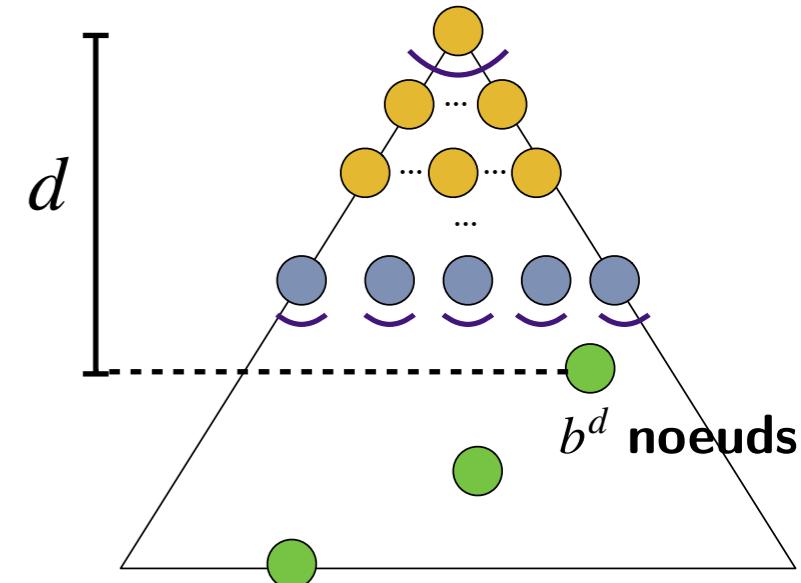
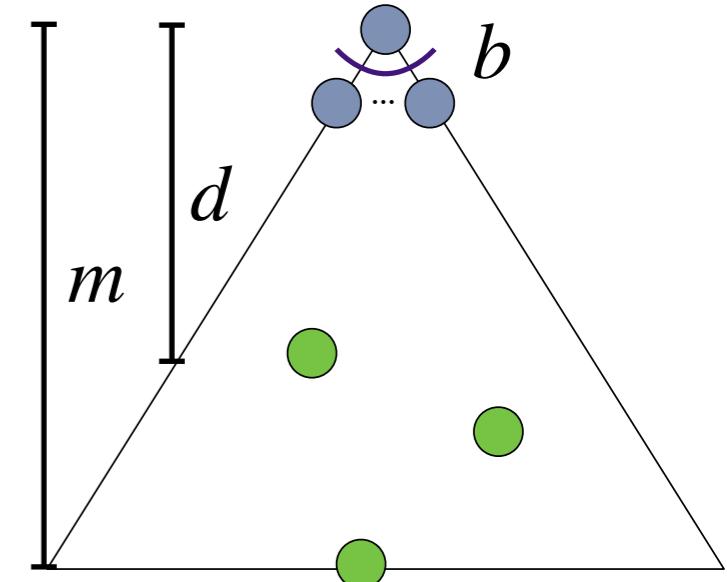
Conclusion positive: la recherche est complète

Critère 4: optimalité

Observation: la solution rentrée est celle ayant le moins d'actions

Conclusion positive: optimal si tous les coûts ont la même valeur

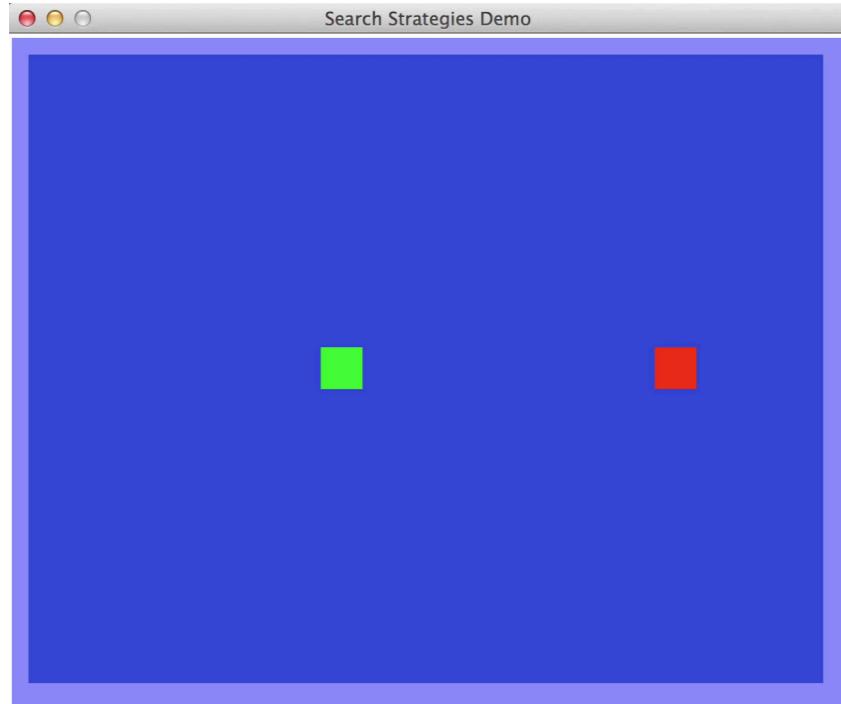
Conclusion négative: non-optimal dans le cas général



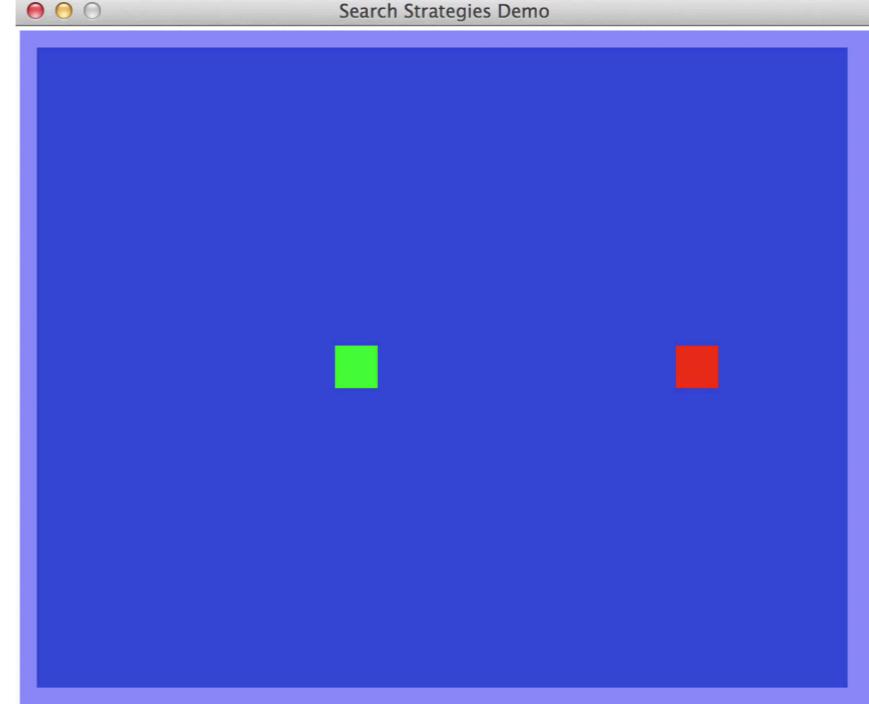
DFS vs BFS: visualisation



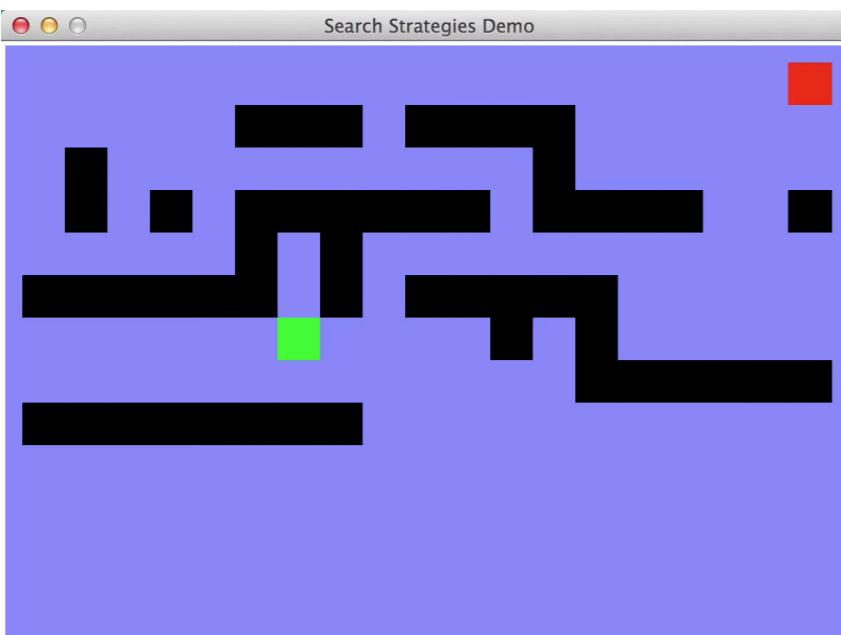
Quelle stratégie de recherche a été appliquée ?



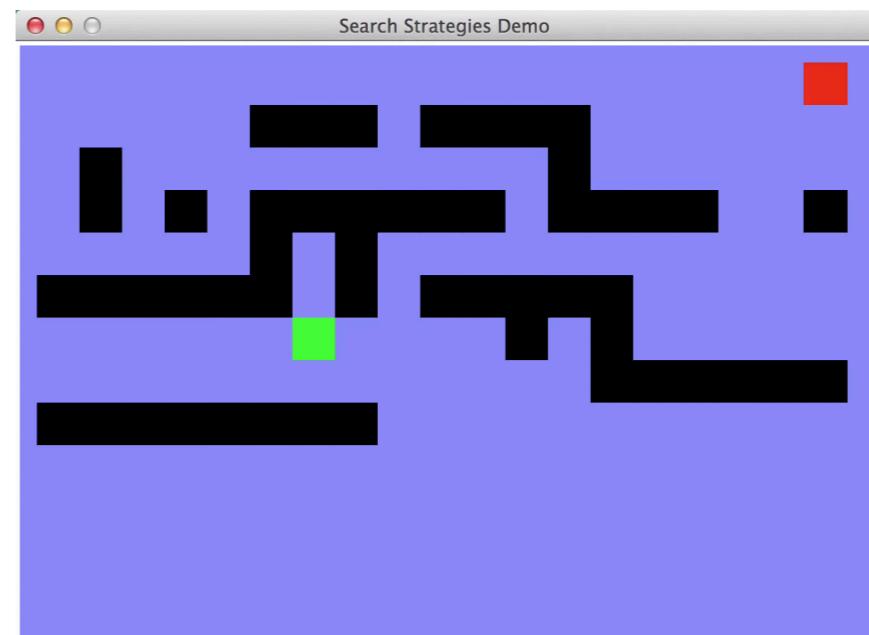
Recherche en largeur



Recherche en profondeur



Recherche en largeur



Recherche en profondeur

DFS vs BFS: comparaison



Quand une alternative est-elle préférable à une autre ?



Arguments pour une recherche en largeur

Argument 1: lorsqu'il est important d'obtenir la solution optimale

Argument 2: lorsqu'obtenir une solution ne requiert que peu d'actions

Argument 3: lorsqu'on a une très bonne capacité mémoire



Arguments pour une recherche en profondeur

Argument 1: lorsque l'arbre est borné (moyen d'enlever les cycles)

Argument 2: lorsqu'avoir la meilleure solution n'est pas important

Argument 3: lorsque la consommation mémoire doit être limitée

En pratique: souvent, en plan B face à un BFS trop coûteux



Peut-on faire mieux ?



On aimeraient pouvoir exploiter les points forts des deux méthodes

Souhait 1: complétude et optimalité de la recherche en largeur

Souhait 2: consommation mémoire similaire à une recherche en profondeur



Comment réaliser cela ?

Recherche à profondeur itérée (*Iterative deepening search* - IDS)



Recherche à profondeur itérée (IDS - iterative deepening search)

Algorithme de recherche qui exécute des DFS, en augmentant itérativement la profondeur maximale

Intuition: un BFS est *simulé* en exécutant des DFS successifs

Motivation: combiner les avantages d'un DFS avec un BFS

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

Solution trouvée: stopper la recherche et retourner la solution

IterativeDeepeningSearch(P) :

```
for  $d \in 1$  to  $\infty$  :  
     $s = \text{DepthLimitedSearch}(P, d)$   
    if  $s \neq \emptyset$  : return  $s$ 
```

DFS borné à la profondeur d

?

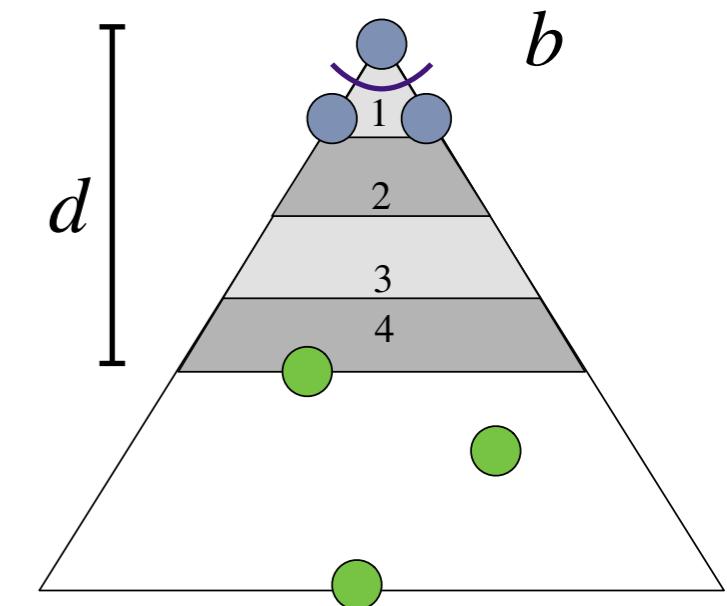
Qu'en est-il des performances ?

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

Complétude: recherche complète (héritage du BFS)

Optimalité: oui, si les coûts sont identiques (héritage du BFS)

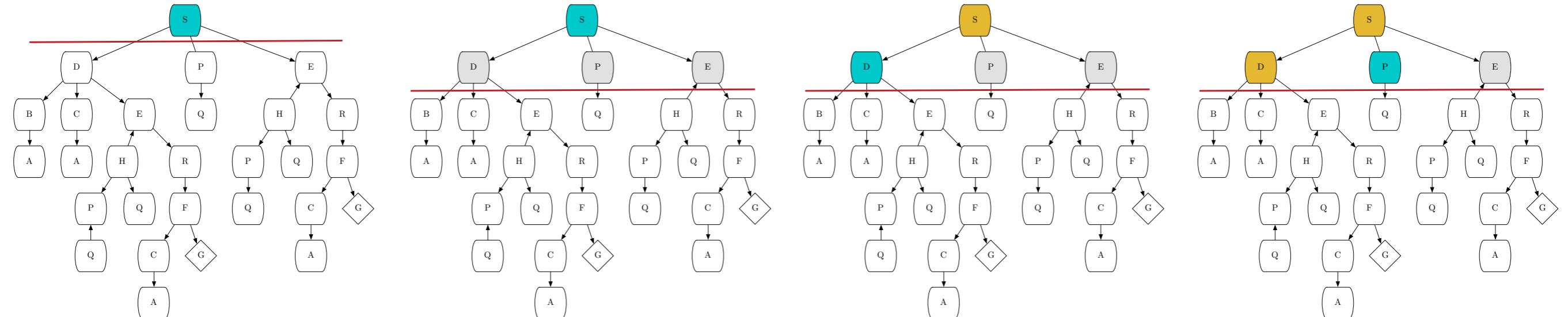
Complexité temporelle : $\mathcal{O}(b^d) + \text{travail redondant}$



DepthLimitedSearch(P, d) :

```
 $s = \text{initialState}(P)$   
 $L = \text{LIFO}()$   
 $\text{push}(L, s)$   
while  $L \neq \emptyset$  :  
     $s = \text{pop}(L)$   
    if  $s = \text{goalState}(P)$  : return solution  
    else :  
         $C = \{c \in \text{succesors}(s, P) \mid \text{depth}(c) \leq d\}$   
         $\text{push}(L, C)$   
return  $\emptyset$ 
```

Iterative deepening search - illustration

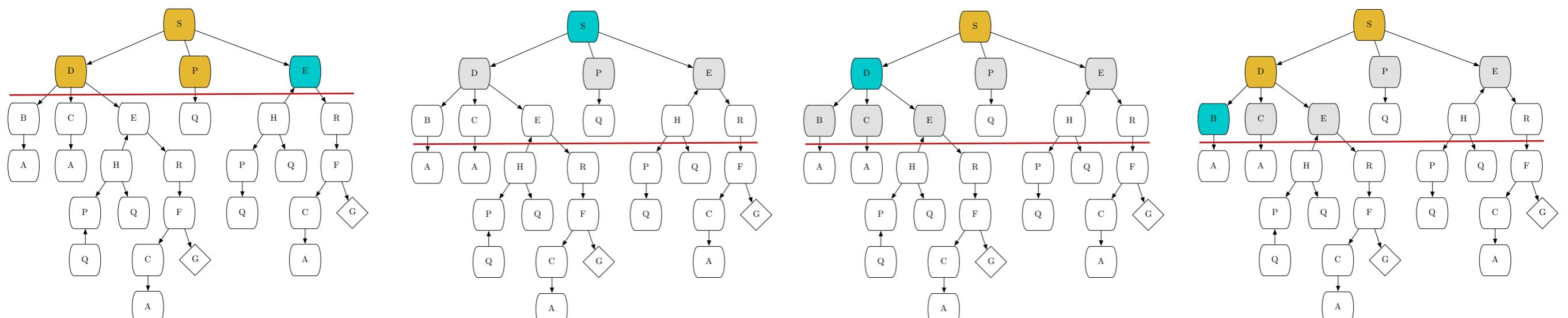


depth = 0

depth = 1

depth = 1

depth = 1



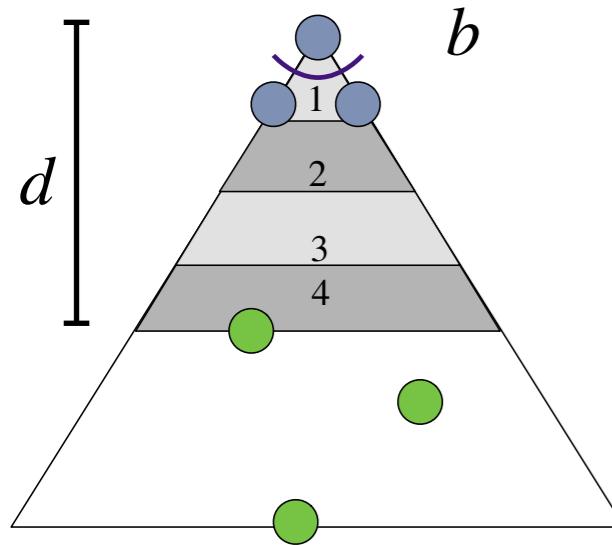
depth = 1

depth = 2

depth = 2

depth = 2

Iterative deepening search - analyse



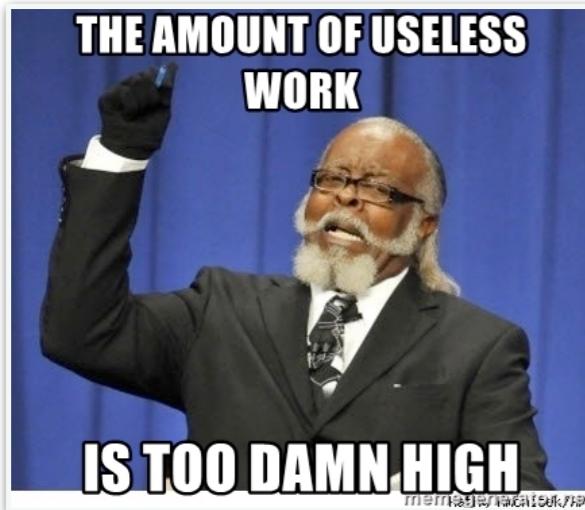
Observation: IDS redemande de ré-exploré des noeuds de l'arbre de recherche

Difficulté: la recherche souffre ainsi de redondance par rapport à DFS ou BFS

Observation: c'est particulièrement le cas pour les noeuds proches de la racine

Observation: le 1er niveau doit être exploré d fois, le 2eme $d-1$ fois, etc.

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



Est-ce un gros problème ?

Fait: c'est exact qu'une certaine quantité de travail redondant est nécessaire

Bonne nouvelle: ce travail redondant est *relativement* négligeable

Exemple: considérons une situation avec $b = 10$ et $d = 5$

Recherche en largeur

$$\text{Nombre de noeuds : } b + b^2 + b^3 + b^4 + b^5$$

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

Iterative deepening search

$$\text{Nombre de noeuds : } db + (d-1)b^2 + (d-2)b^3 + (d-3)b^4 + b^5$$

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

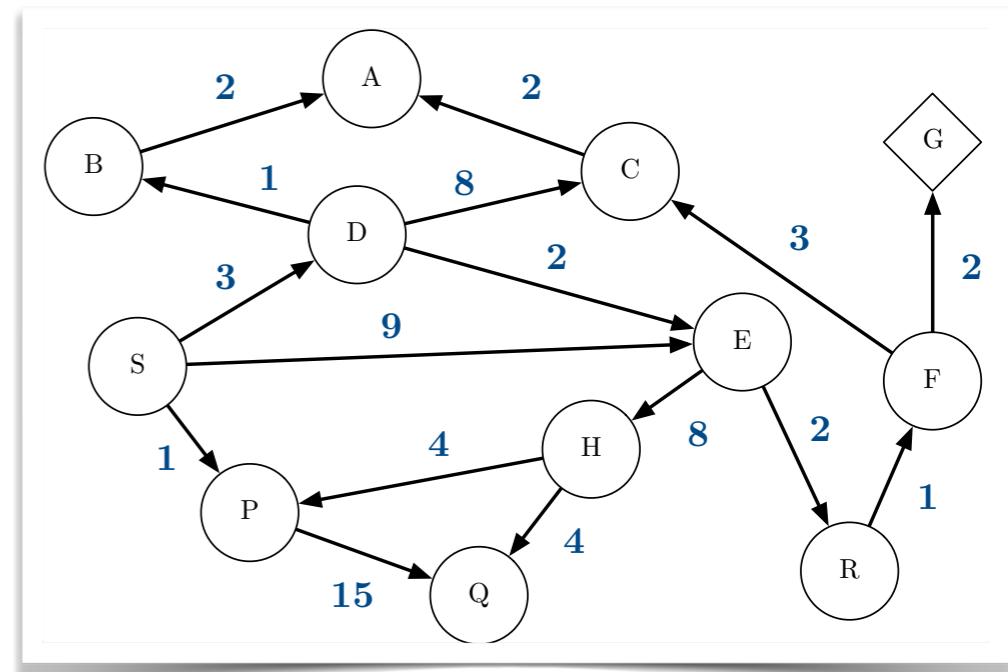
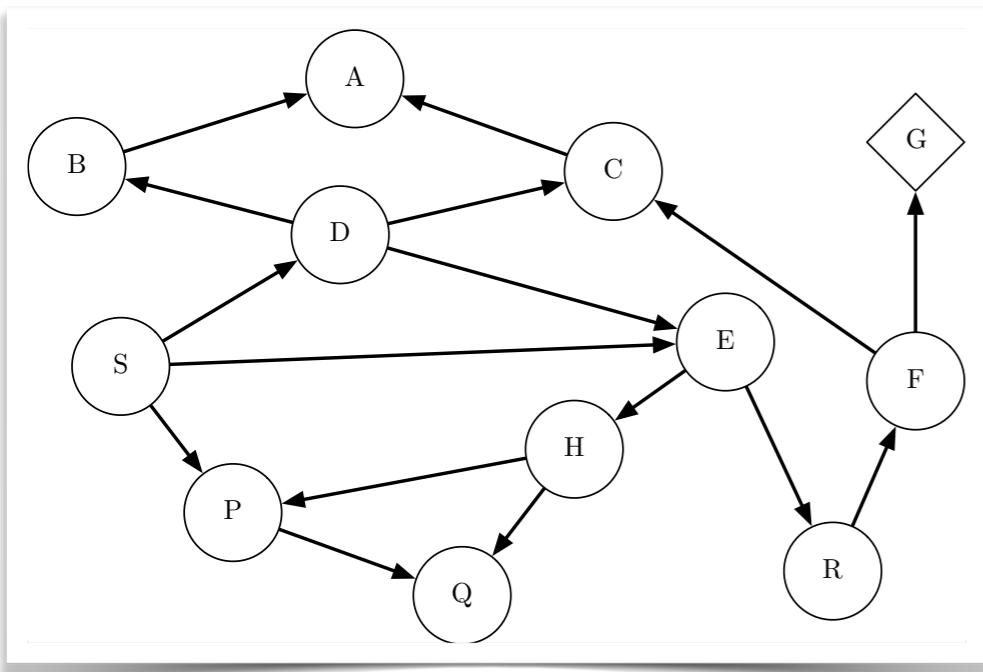
Observation: seulement environ 11% de noeuds supplémentaires (et s'atténue plus le problème est grand)

Explication: le gros du travail se fait sur le dernier niveau qui est n'exploré qu'une fois dans les deux cas

Conclusion: en pratique, IDS est une stratégie majoritairement préférable à un DFS ou un BFS simple

Problèmes de recherche avec coûts non identiques

? Toutes nos situations avaient des coûts identiques ! Que se passe t-il si ce n'est pas le cas ?



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

Solution optimale: celle ayant le plus petit coût (somme des coûts des actions faites)



Que va faire une recherche en largeur ?

Nature du BFS: donne la solution la plus proche en termes de nombre d'actions

Observation: cela ne correspond pas toujours à la solution optimale !

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

Recherche à coût uniforme (UCS - uniform cost search)

 **Recherche à coût uniforme (UCS - uniform cost search)**
Algorithme de recherche consistant à retirer systématiquement le noeud ayant le plus faible coût

Coût d'un noeud: correspond à la somme des coûts depuis la racine

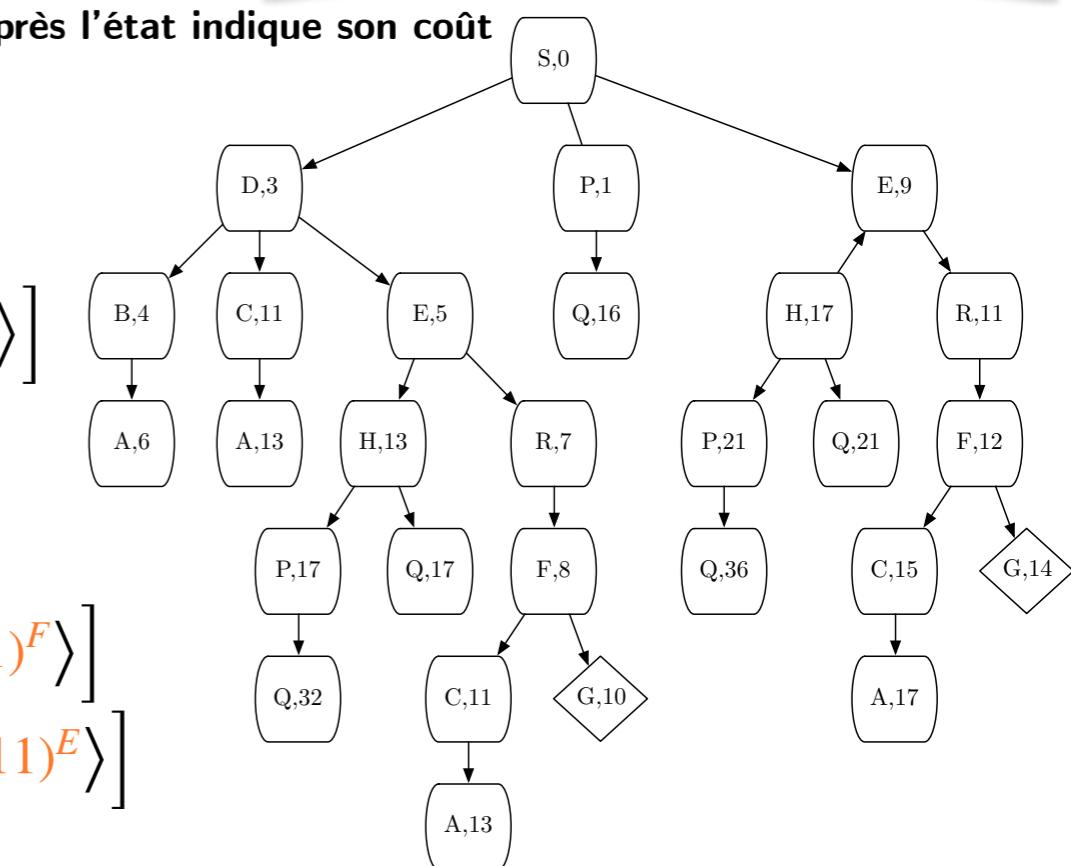
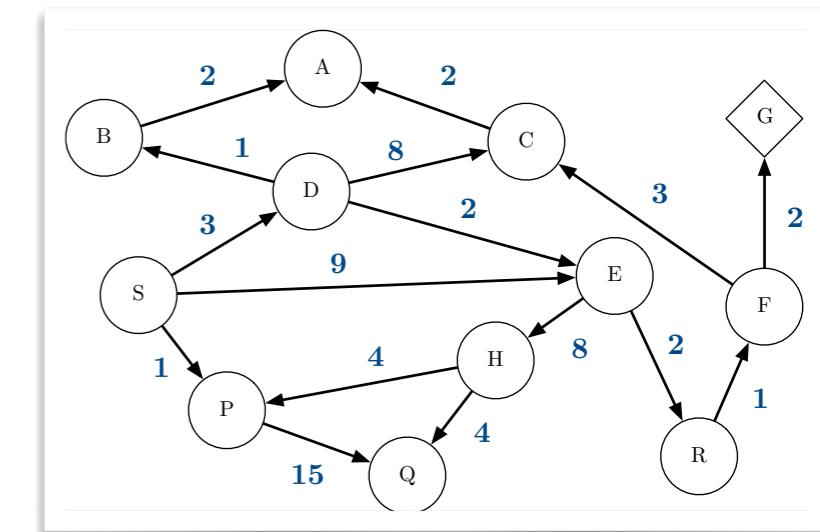
Nouvel élément: on intègre une fonction de coût pour un noeud

$$g(n) : \text{Node} \rightarrow \mathbb{R}$$

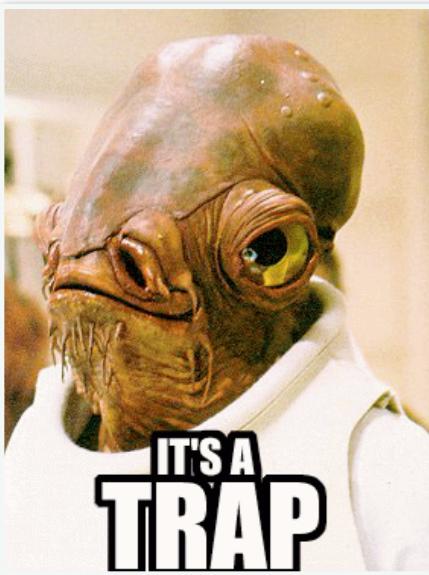
Principe: les coûts sont calculés quand le noeud est dans la frontière

Exemple: exécution de l'algorithme sur cette situation

- 1 : **state** : S , **fringe** : $\langle (D,3)^S, (P,1)^S, (E,9)^S \rangle$
- 2 : **state** : P^S , **fringe** : $\langle (D,3)^S, (E,9)^S, (Q,16)^P \rangle$ **Notation:** le nombre après l'état indique son coût
- 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** : $\langle (E,9)^S, (Q,16)^P, (C,11)^D, (H,13)^E, (G,10)^F, (C,11)^F \rangle$
- 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



UCS - Piège classique et algorithme



?

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

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

Attention: même si l'état final est dans la *fringe*, il ne doit pas être étendu en priorité

Explication: rien n'assure que via *E*, il existe pas un chemin de plus faible coût

Conclusion: la recherche est finie QUE si vous êtes à l'état final

Algorithme: implémentation de la *fringe* par une liste de priorité

Coût: chaque noeud doit contenir en plus l'information de son coût

UniformCostSearch(*P*) :

s = initialState(*P*)

L = PriorityQueue()

push(*L*, *s*, *g(s)*)

while *L* $\neq \emptyset$:

s = pop(*L*)

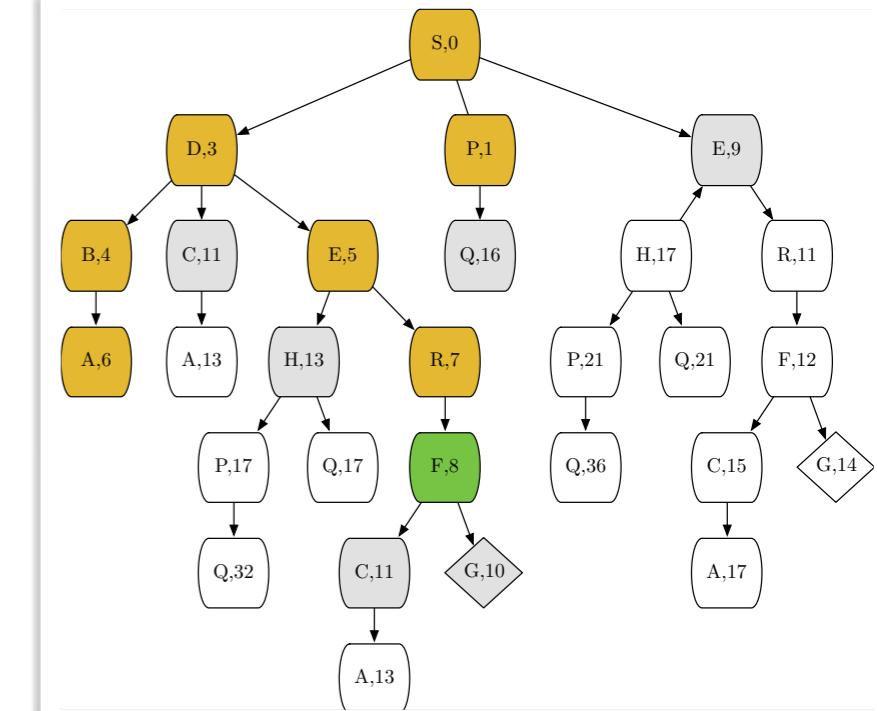
if *s* = goalState(*P*) : return solution

else :

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

push(*L*, *C*)

return no solution



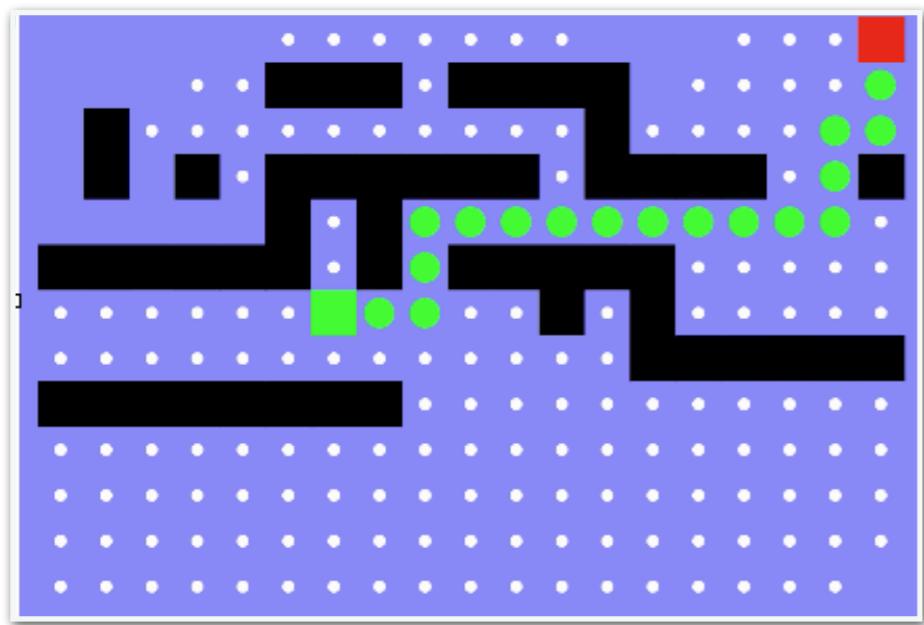
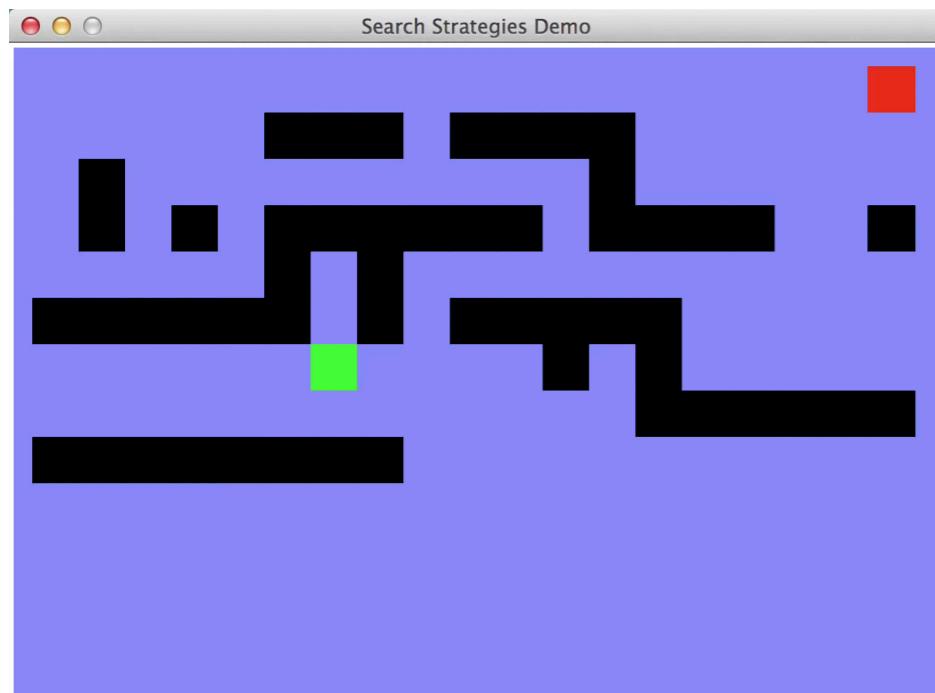
push(*L*, *s*, *g(s)*) : ajoute *s* et son coût à la structure *L*

pop(*L*) : retire l'élément ayant le plus petit coût *g(s)*

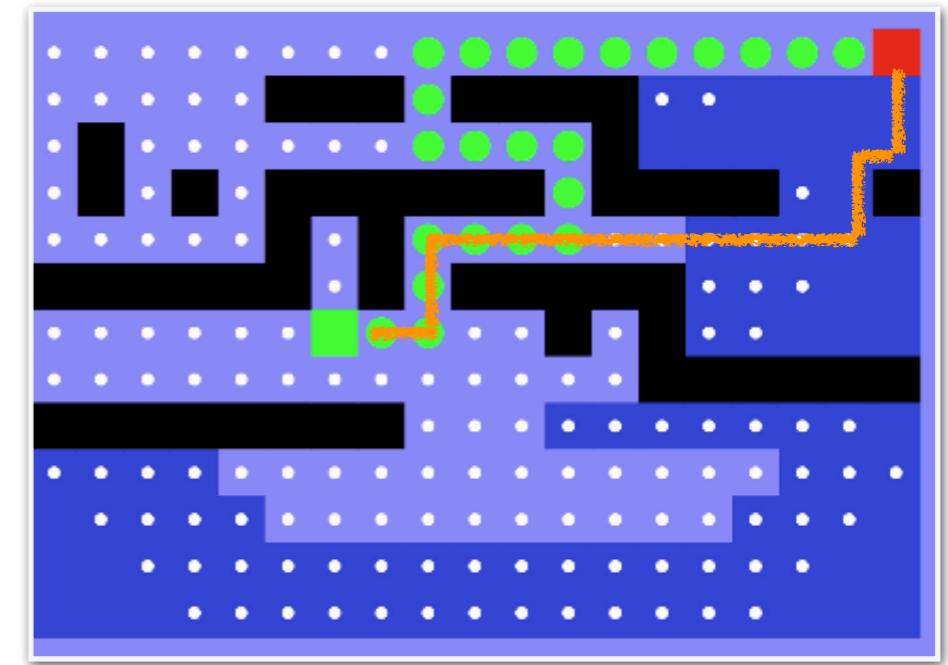
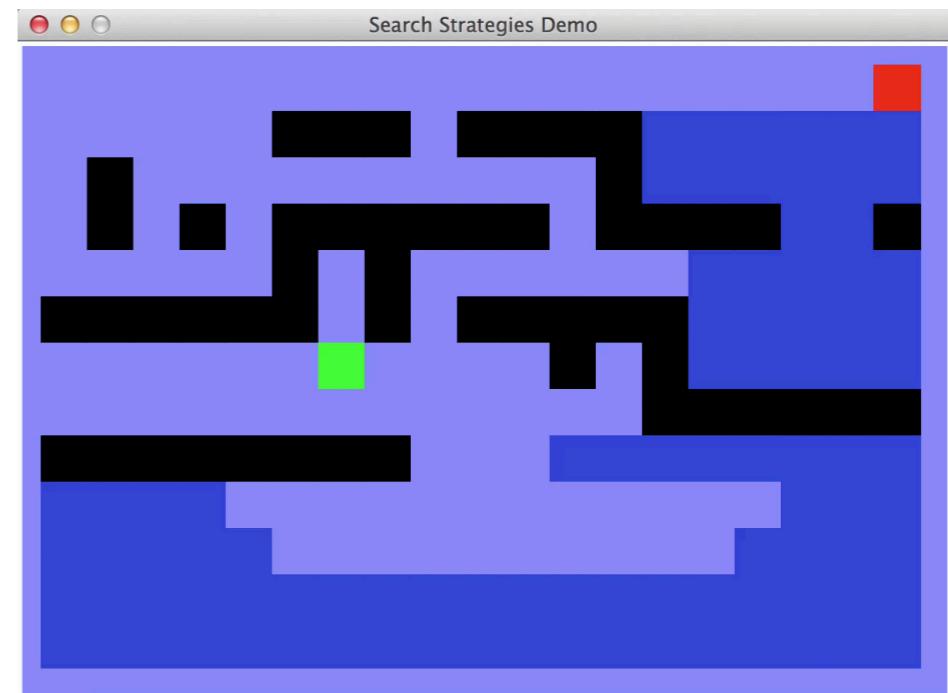
$$g(c) = g(s) + d(s \rightarrow c)$$

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

UCS - illustration



Situation: tous les coûts sont identiques
Equivalent à une recherche en largeur



Situation: la zone bleue a un coût supérieur
Retourne la solution optimale (et non le BFS)

BFS

Recherche à coût uniforme - analyse des performances



Quelles sont les performances d'une recherche à coût uniforme ?

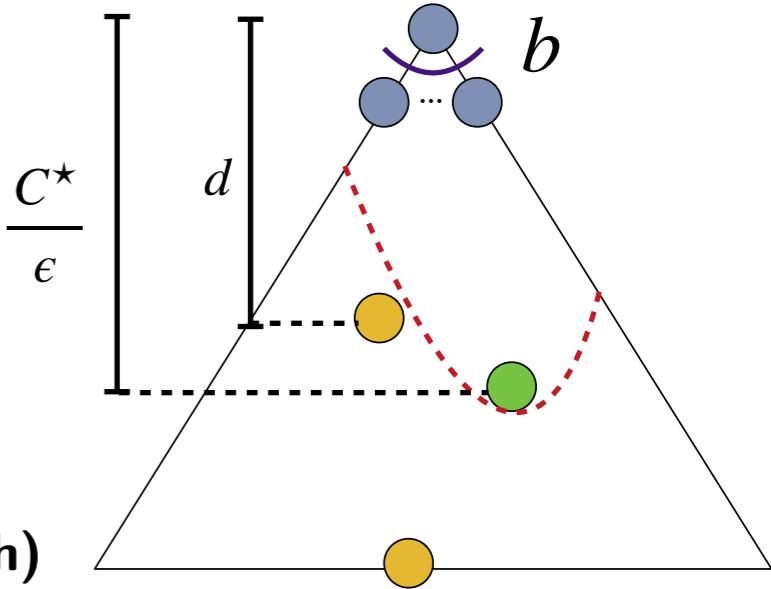
b : nombre maximum de successeurs (branching factor)

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)



Critère 1: complexité temporelle

Intuition: tous les noeuds ayant un coût moindre que celui de la meilleure solution seront étendus

Conclusion: toutes les solutions se situant avant la profondeur effective doivent être explorées

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

Critère 2: complexité spatiale (pire des cas)

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

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

Critère 3: complétude

Bonne nouvelle: recherche complète (hypothèse que les coûts sont positifs et finis)

Critère 4: optimalité

Bonne nouvelle: la recherche est optimale (même argument que pour la recherche en largeur)

Recherche à coût uniforme



Recherche à coût uniforme

Terminologie: utilisée par la communauté de l'intelligence artificielle

Champs d'application: recherche en graphe et recherche en arbre

Difficulté: la taille croît exponentiellement avec le nombre d'états

Difficulté: la taille de l'arbre est éventuellement infinie

En fonction de l'implémentation, quelques différences peuvent exister

Algorithm 1: Dijkstra's algorithm

```
Input: Graph  $G = (V, E)$ 
1  $(\forall x \neq s) dist[x] = +\infty$  //Initialize dist[]
2  $dist[s] = 0$ 
3  $S = \emptyset$ 
4  $Q = V$  // Keyed by  $dist[]$ .
5 while  $Q \neq \emptyset$  do
6    $u = extract\_min(Q)$ 
7    $S = S \cup \{u\}$ 
8   foreach vertex  $v \in Adj(u)$  do
9      $dist[v] = \min(dist[v], dist[u] + w(u, v))$ 
10    //''Relax'' operation.
```

Algorithme de Dijkstra

Propriété: logiquement équivalent à UCS (même ordre d'extension)

Terminologie: utilisée par la communauté des sciences informatiques

Applications: principalement pour référer à des problèmes de réseaux

Exemples: réseaux routiers, télécommunications, etc.

Taille du graphe: souvent raisonnable dans ces applications



Pourquoi doit-on apprendre un nouvel algorithme - et ne pas rester sur Dijkstra ?

Recherche à coût uniforme vs Dijkstra



Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm

Ariel Felner

Information Systems Engineering

Ben-Gurion University

Be'er-Sheva, Israel 85104

felner@bgu.ac.il

I claim that UCS is superior to DA in almost all aspects. It is easier to understand and implement.

Its time and memory needs are also smaller.

The reason that DA is taught in universities and classes around the world is probably only historical.

I encourage people to stop using and teaching DA, and focus on UCS only.

Reste du papier: analyse de UCS et de Dijkstra

Conclusion générale: ne plus utiliser (ni enseigner) Dijkstra, et considérer UCS à la place

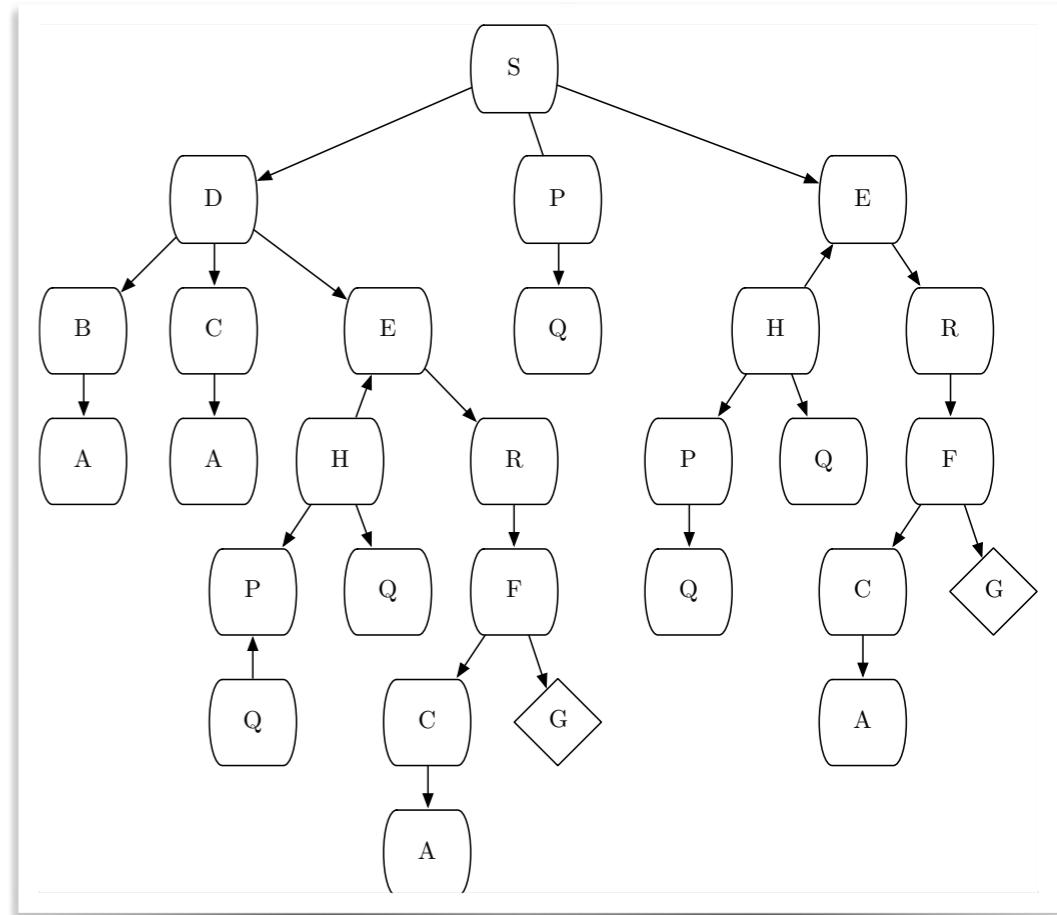
Remarque: il s'agit d'un papier d'opinion, qui garde une certaine subjectivité

Opinion personnelle: je préfère également UCS, par l'aspect générique avec les autres algorithmes

Table des matières

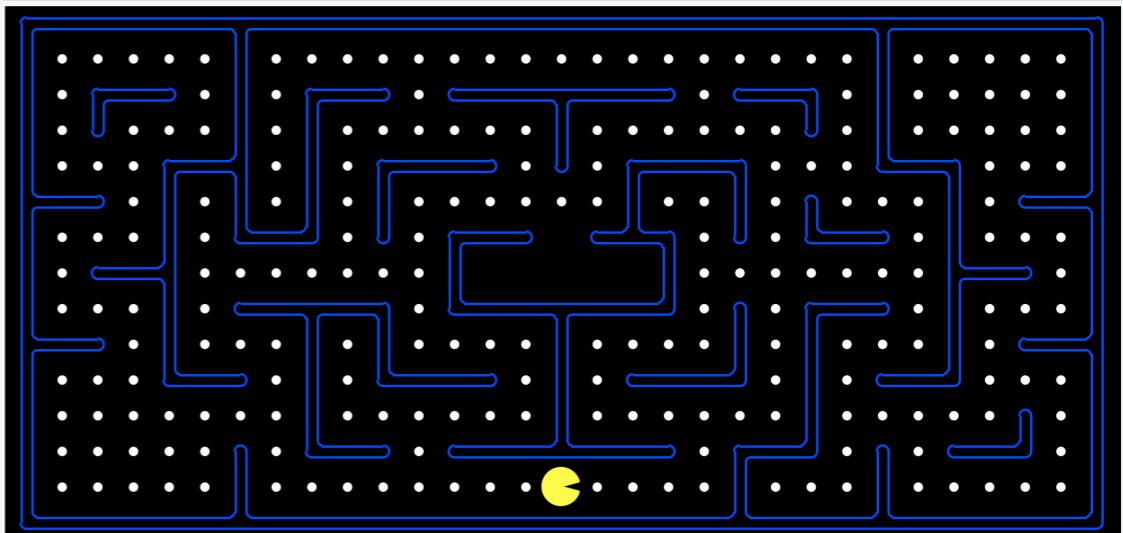
Stratégies de recherche

- ✓ 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



On a déjà vu 4 algorithmes de résolution ! Pourquoi en voir des nouveaux ?

Bonne nouvelle: le dernier algorithme vu (UCS) est complet et optimal

Mauvaise nouvelle: il est très coûteux en termes de mémoire et de temps d'exécution

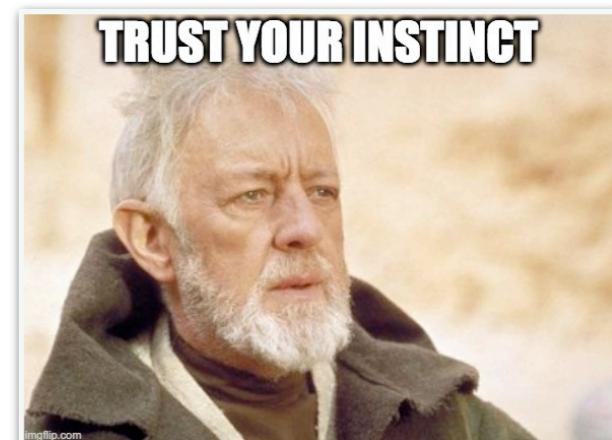
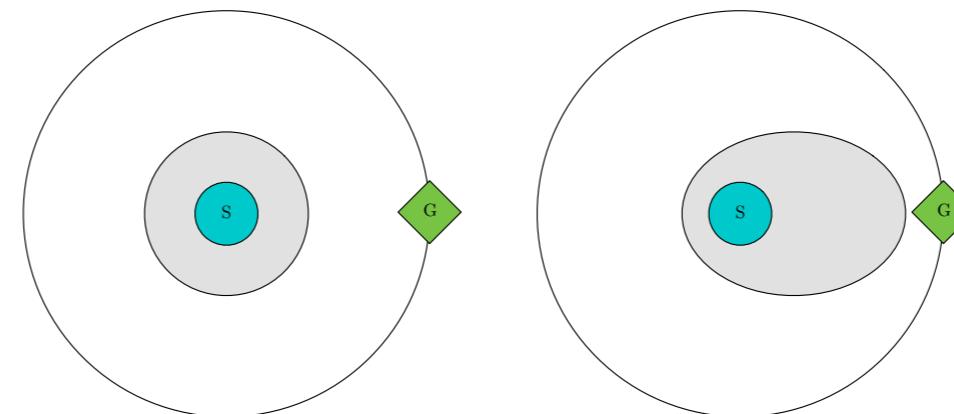
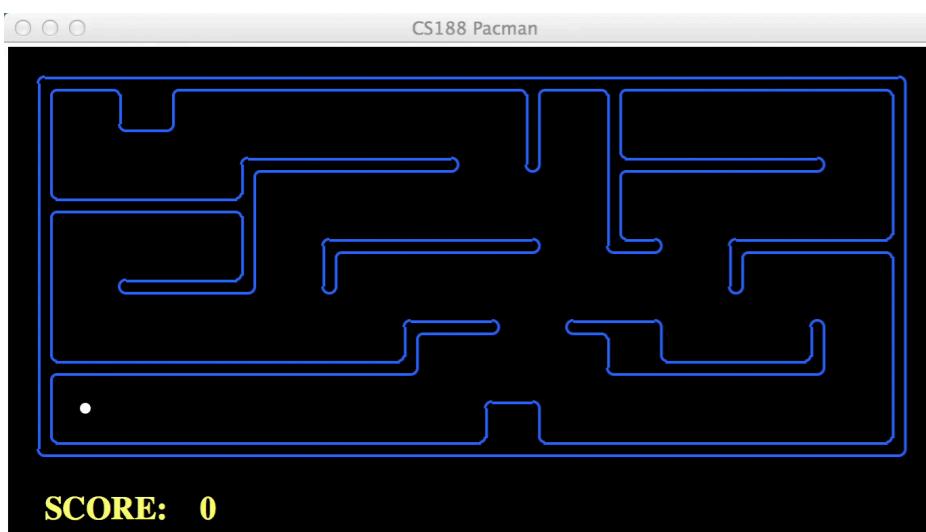


Qu'est-ce qu'on pourrait faire pour améliorer les performances ?

Observation: on n'utilise aucune information spécifique au problème à résoudre

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

Conséquence: on va autant explorer dans toutes les directions avant d'atteindre l'objectif



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

Exemple: 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 (*informed search*)

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



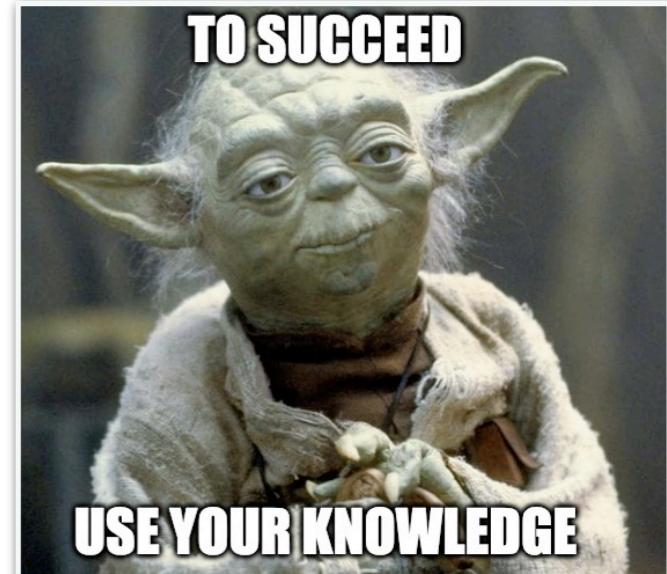
Quels sont les points de conception majeurs de cette famille de méthode ?

Critère 1: le type de connaissance à injecter à la recherche

- (1) La nature de la connaissance à injecter
- (2) La validité et propriétés de la connaissance
- (3) La complexité temporelle pour obtenir cette connaissance

Critère 2: la façon d'injecter cette connaissance dans la stratégie de recherche

- (4) Le degré de confiance que l'on veut avoir dans la connaissance injectée
- (5) Les 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

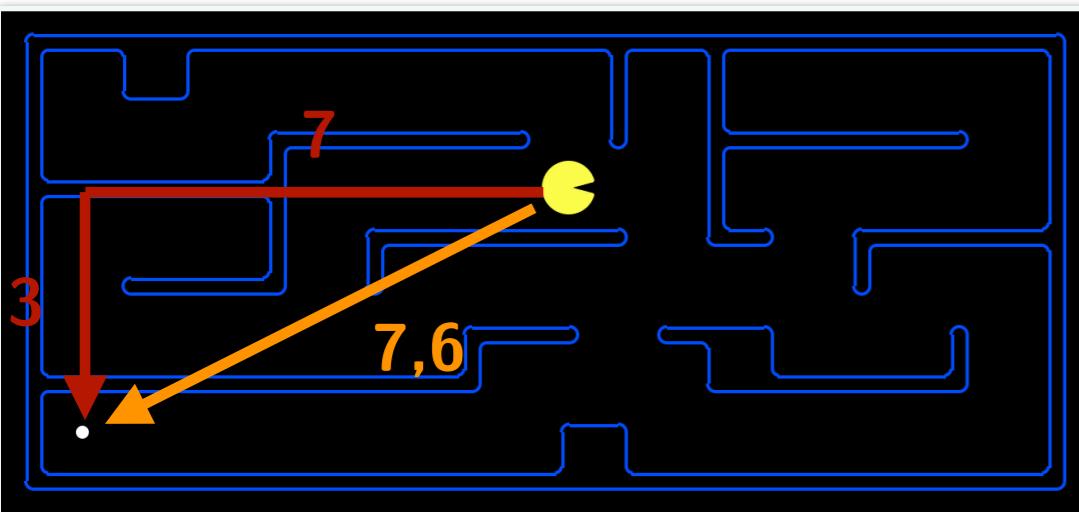
Principe: une heuristique donne une information sur à quel point un état est coûteux

Intuition: on va privilégier l'exploration des états dont l'heuristique retourne une faible valeur

Travail supplémentaire: elles doivent généralement être conçues spécifiquement pour chaque problème



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



(1) Distance euclidienne: chemin à vol d'oiseau

(2) 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

On va voir par la suite des techniques pour concevoir de bonnes heuristiques

Recherche gloutonne (*greedy best-first search*)



Recherche gloutonne (*greedy best-first search*)

Algorithme de recherche consistant à retirer le noeud le moins coûteux selon l'heuristique

Principe: chaque état est évalué selon la fonction heuristique

Besoin: avoir défini une heuristique pour le problème

$$h(n) : \text{Node} \rightarrow \mathbb{R}$$

Implémentation: identique à UCS (seulement la priorité change)

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



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

Observation: on a passé du temps à avoir une bonne formalisation, mais c'est rentable !

Attention: on n'a plus aucune notion de coût déjà généré (comme avec UCS)

Recherche gloutonne (*greedy best-first search*)



Quelles sont les performances d'une recherche gloutonne ?

Critère 1: complexité temporelle

Pire des cas: agit comme un mauvais DFS

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

Critère 2: complexité spatiale

Pire des cas: agit comme un mauvais BFS

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

Critère 3: complétude

Mauvaise nouvelle: la recherche n'est pas complète (même problème que le DFS)

Critère 4: optimalité

Mauvaise nouvelle: très haut risque de tomber sur une solution sous-optimale

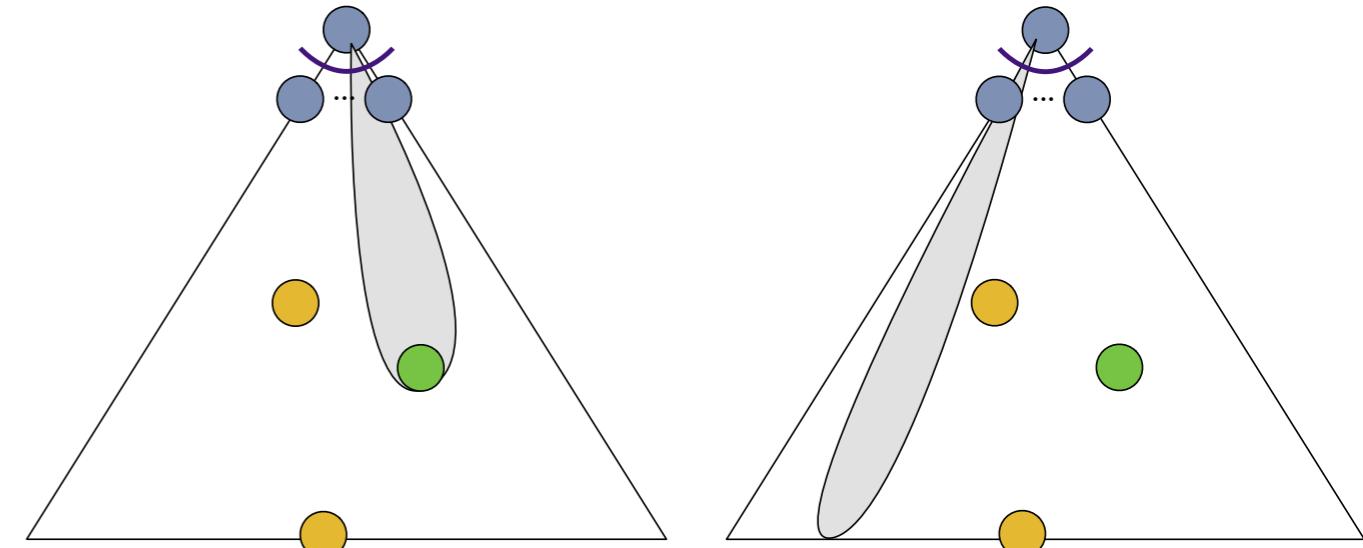


Difficulté majeure: le manque de garanties théoriques

En pratique: les performances peuvent être bonnes si l'heuristique est adaptée

Il y a une distinction entre pire cas théorique et bons résultats en pratique

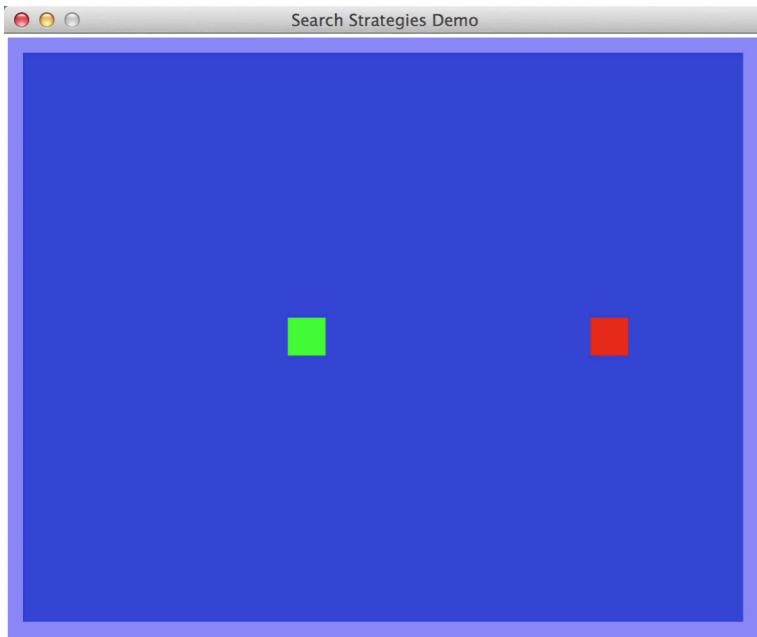
Amélioration possible: apporter plus de garanties à cet algorithme



Recherche gloutonne - exemples



Que va faire une recherche gloutonne dans ces situations ?



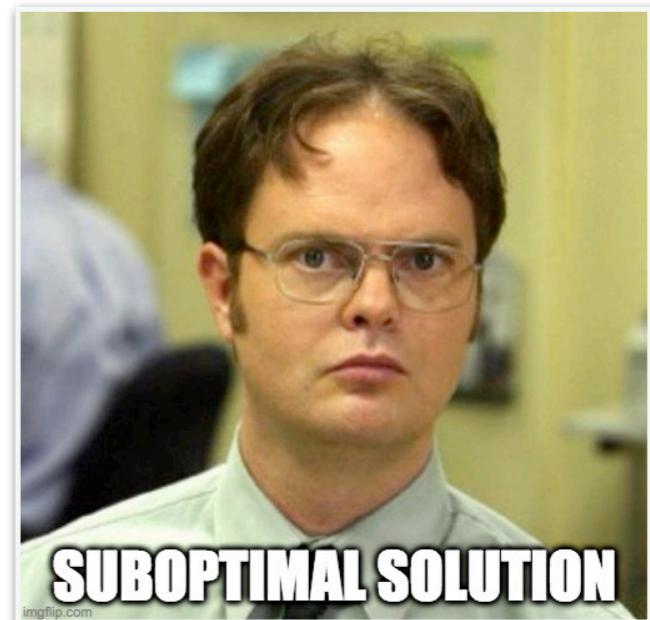
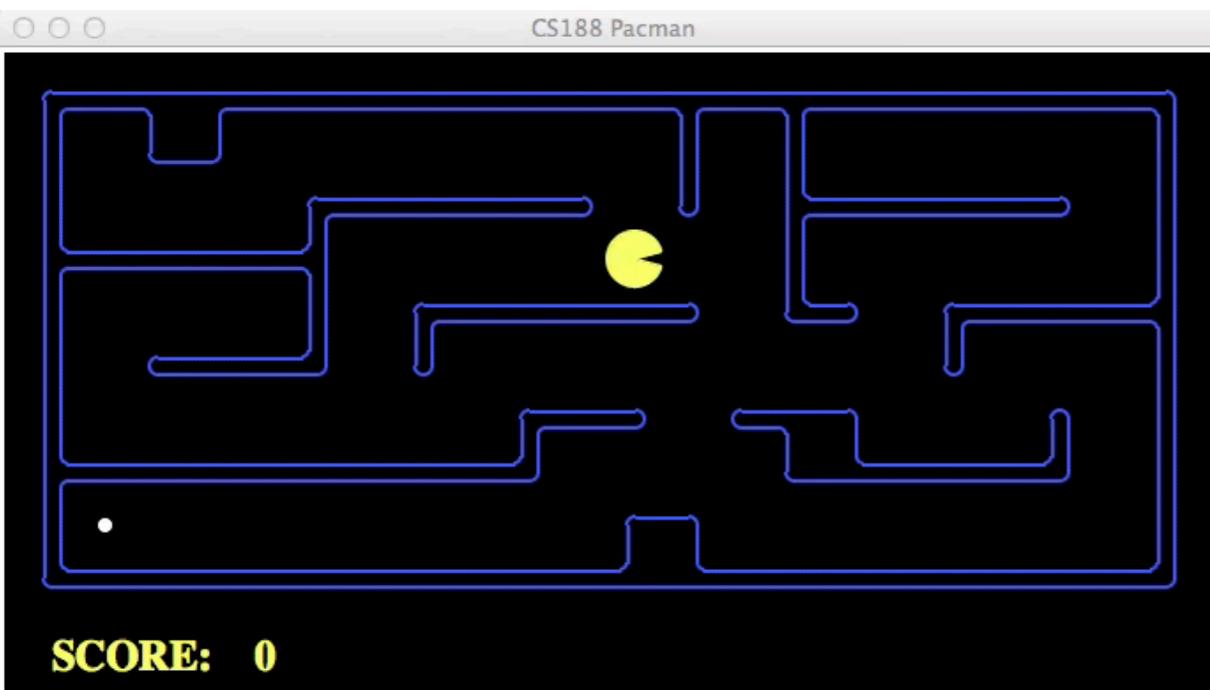
Réponse: ça dépend de l'heuristique qui a été choisie !



*"La fonction heuristique,
vous définirez"*

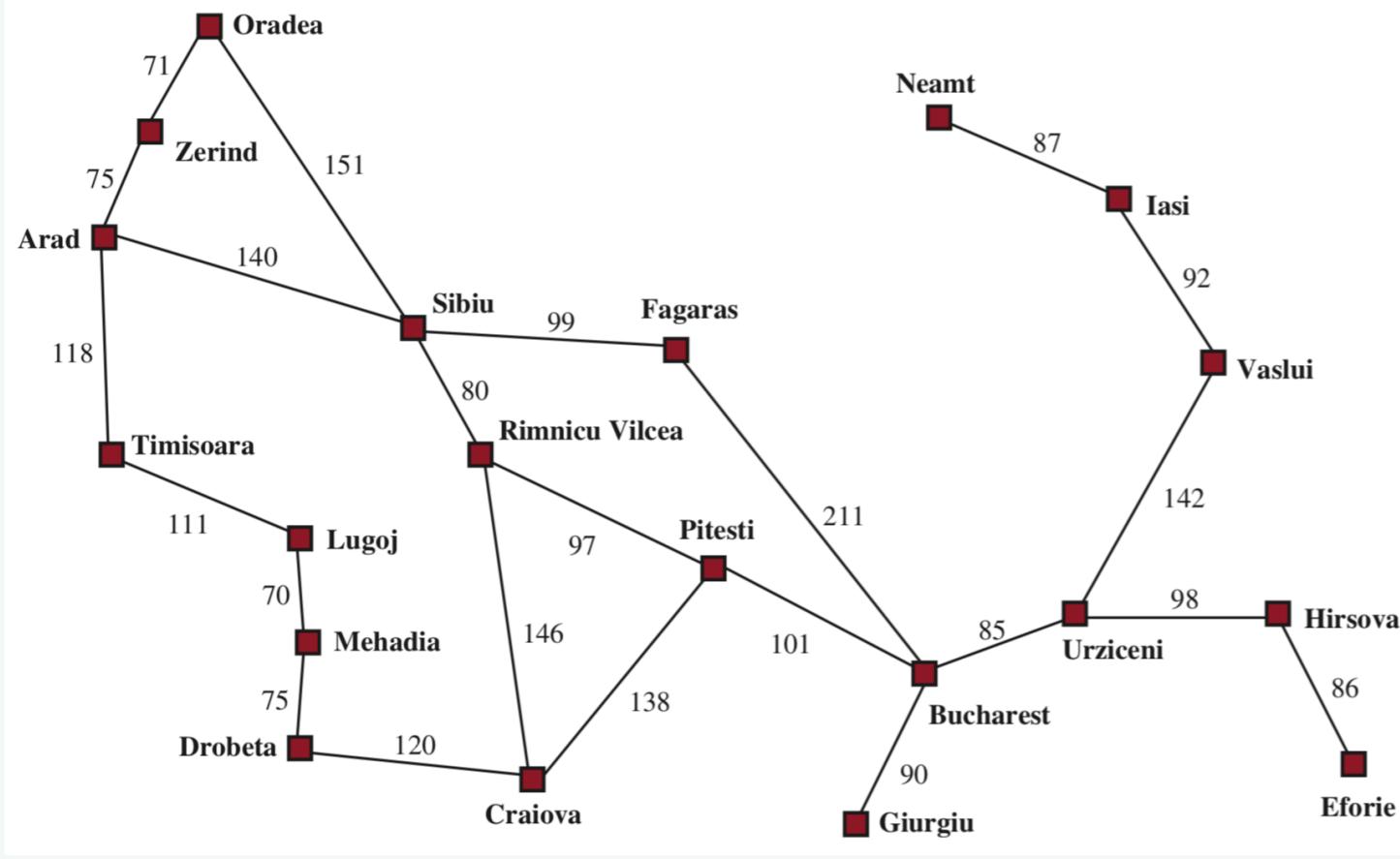
Attention: sans définir l'heuristique, cette question n'a aucun sens

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



Observation: la solution retornnée n'est pas la meilleure

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 ?

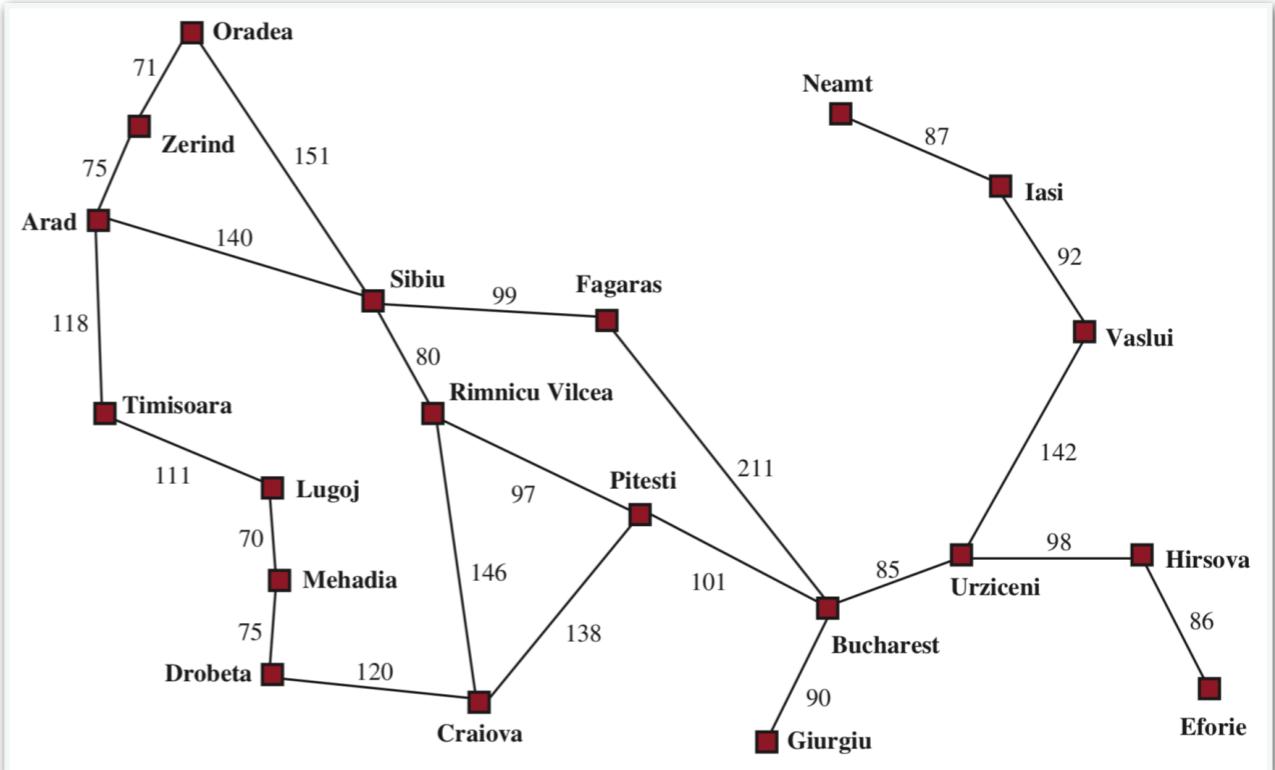
Heuristique: la distance à vol d'oiseau semble être un choix raisonnable

Idéalement: l'heuristique ne doit pas être trop coûteuse à calculer (c'est bien le cas ici)

Recherche gloutonne: sélectionne le successeur ayant la plus faible coût estimé

Recherche gloutonne - exécution de l'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



Exemple: exécution de l'algorithme

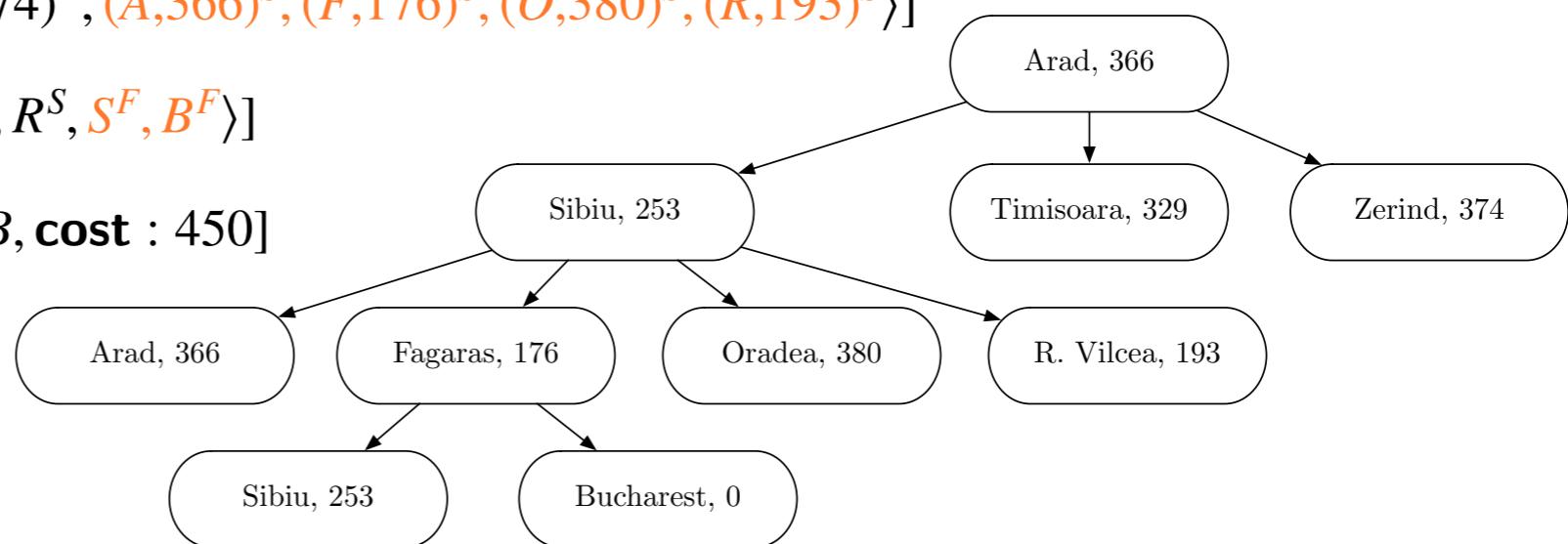
Etape 1 : [state : A, fringe : $\langle (S, 253)^A, (T, 329)^A, (Z, 374)^A \rangle$]

Etape 2 : [state : S^A , fringe : $\langle (T, 329)^A, (Z, 374)^A, (A, 366)^S, (F, 176)^S, (O, 380)^S, (R, 193)^S \rangle$]

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

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

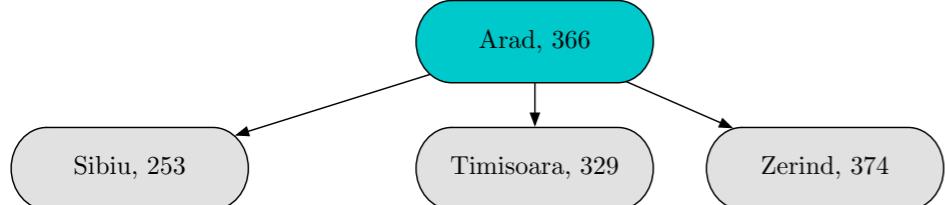
Solution trouvée !



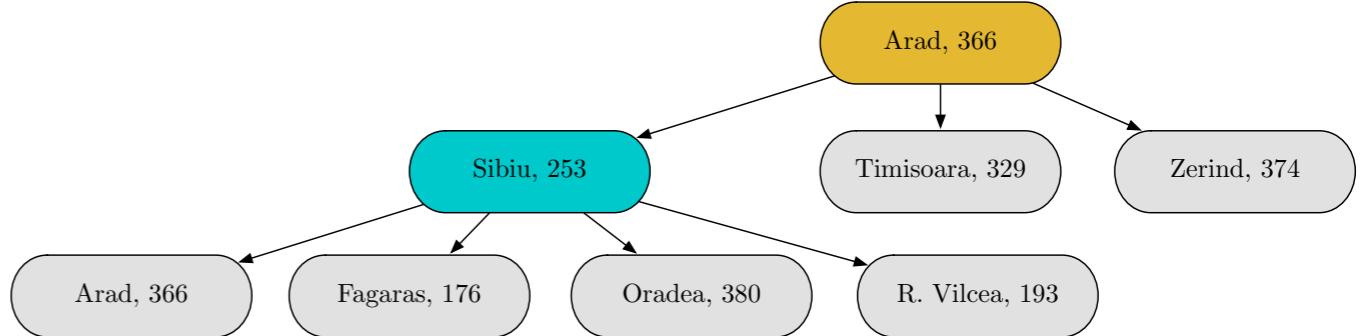
?

Est-ce que notre solution est optimale ?

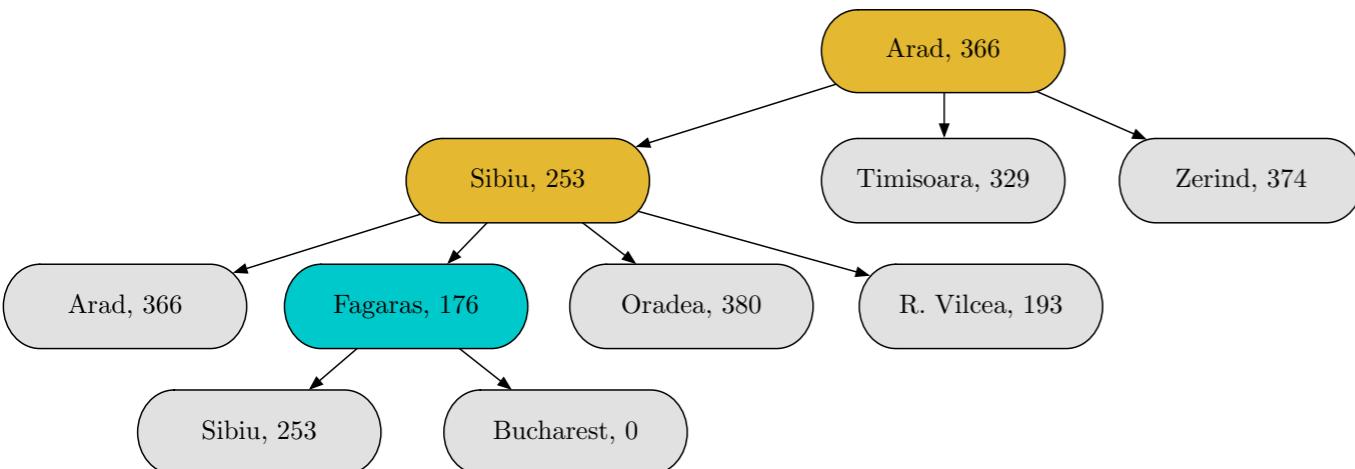
Recherche gloutonne - exemple



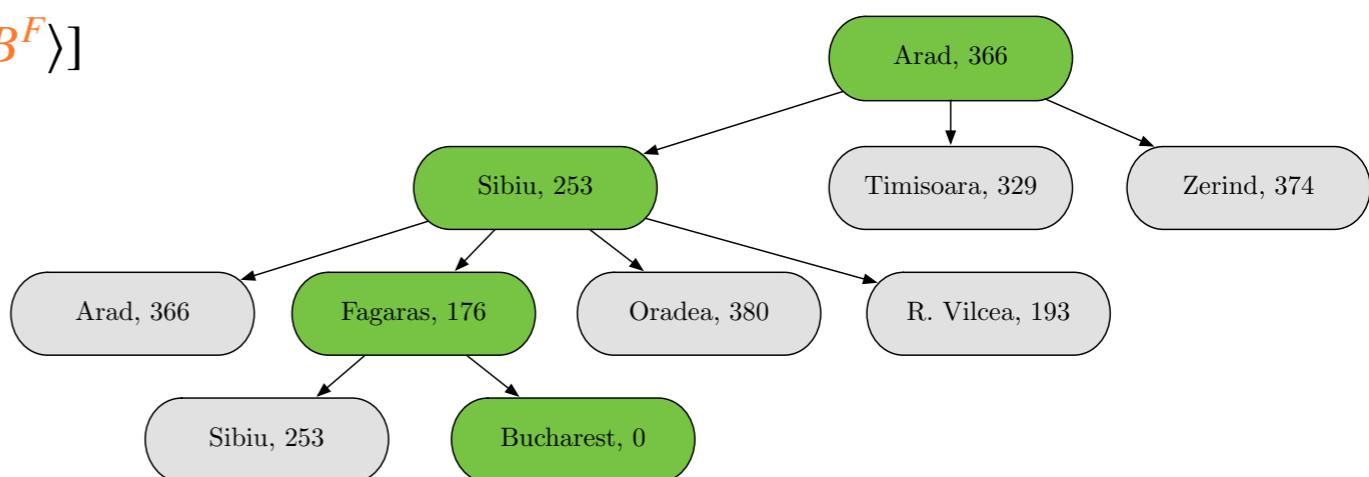
Etape 1 : [state : A , fringe : $\langle S^A, T^A, Z^A \rangle$]



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

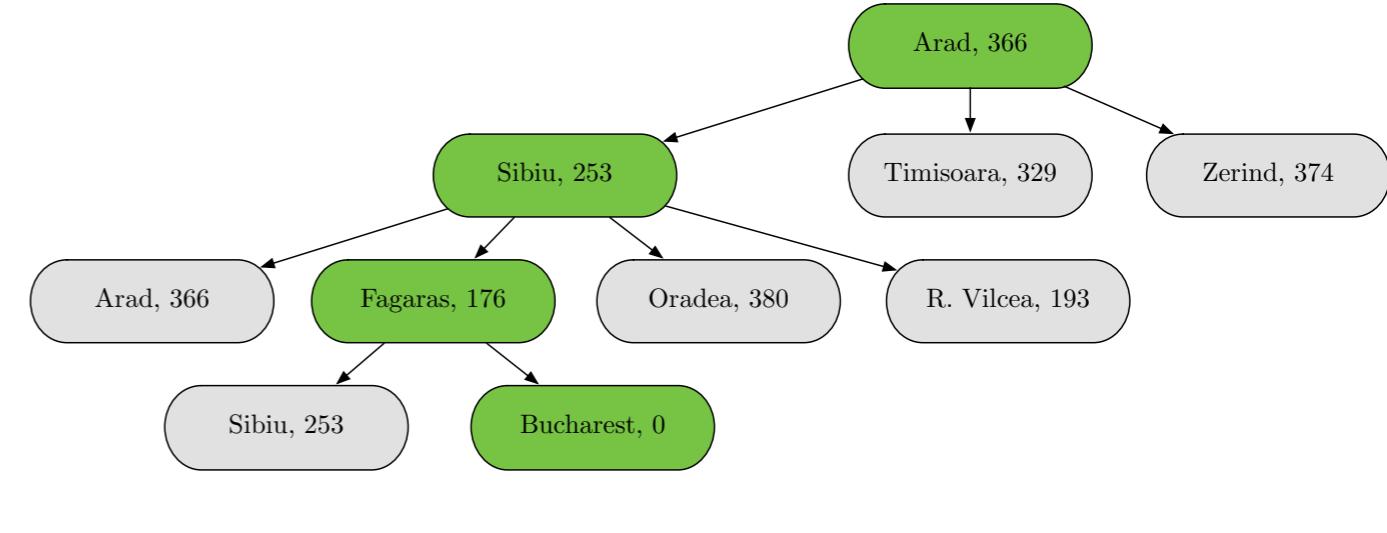
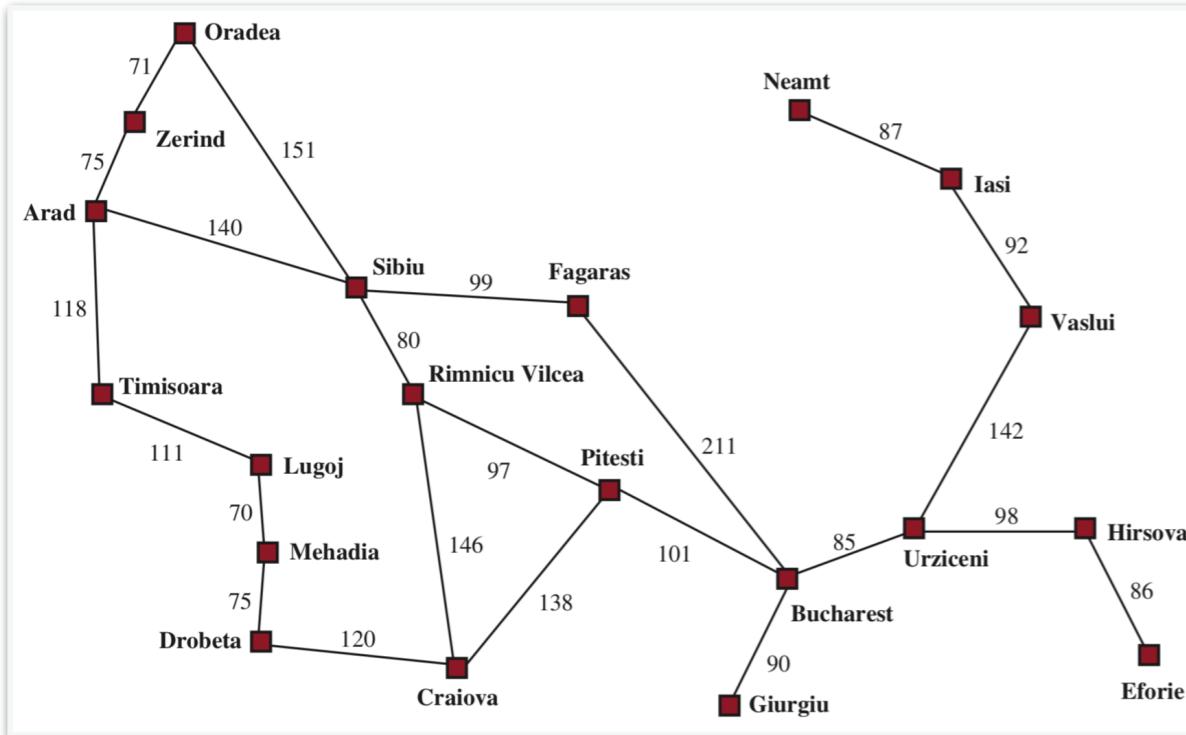


Etape 3 : [state : F^S , fringe : $\langle T^A, Z^A, A^S, O^S, R^S, S^F, B^F \rangle$]



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

Recherche gloutonne - analyse de la solution



?

Est-ce que notre solution est optimale ?

Coût de notre solution: $g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest}) \rightarrow 140 + 99 + 211 = 450$

Coût d'une autre solution: $g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{R. Vilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bucharest}) \rightarrow 140 + 80 + 97 + 101 = 418$

?

Pourquoi a t-on manqué cette meilleure solution ?

Défaut de notre heuristique: elle a complètement négligé R. Vilcea au détriment de Fagaras

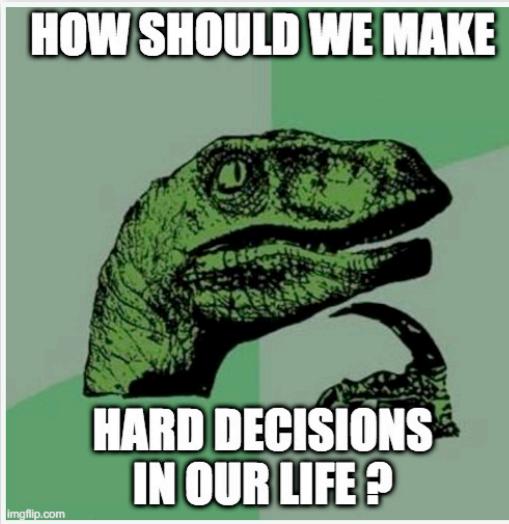
Observation: il s'agit pourtant d'un choix raisonnable, car le coût déjà effectué depuis Arad est moindre

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

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

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

Motivation d'une nouvelle stratégie



Question philosophique: compliqué à répondre, 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

En pratique: souvent nos décisions personnelles tiennent comptes de ces deux aspects



Quel est le lien avec nos stratégies de recherche ?

Stratégie UCS: considère uniquement les coûts déjà générés

Avantage: fiable (optimal)

Inconvénient: lent

Stratégie gloutonne: considère uniquement les coûts futurs estimés

Avantage: peut accélérer la recherche si l'heuristique est bonne

Inconvénient: non-fiable



Idée: 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* (*A-star search*)



Stratégie de recherche A* (*A-star search*)

Stratégie qui consiste à retirer le noeud en fonction des coûts passés et d'une heuristique

Construction de la fonction: somme des coûts passés et du coût heuristique

Priorité A* : $f(s) = g(s) + h(s)$

$g(s)$: coût du noeud s (similaire à UCS)

$h(s)$: heuristique au noeud s (estimation des coûts futurs)

Utilisation de la fonction: sélectionner le prochain noeud à explorer dans la file de priorité

Bonne nouvelle: encore une fois, on ne change quasi rien à notre algorithme

AStarSearch(P , $\textcolor{red}{h}$) :

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

$L = \text{PriorityQueue}()$

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

$\text{push}(L, s, \textcolor{red}{f}(s))$

while $L \neq \emptyset$:

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

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

else :

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

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

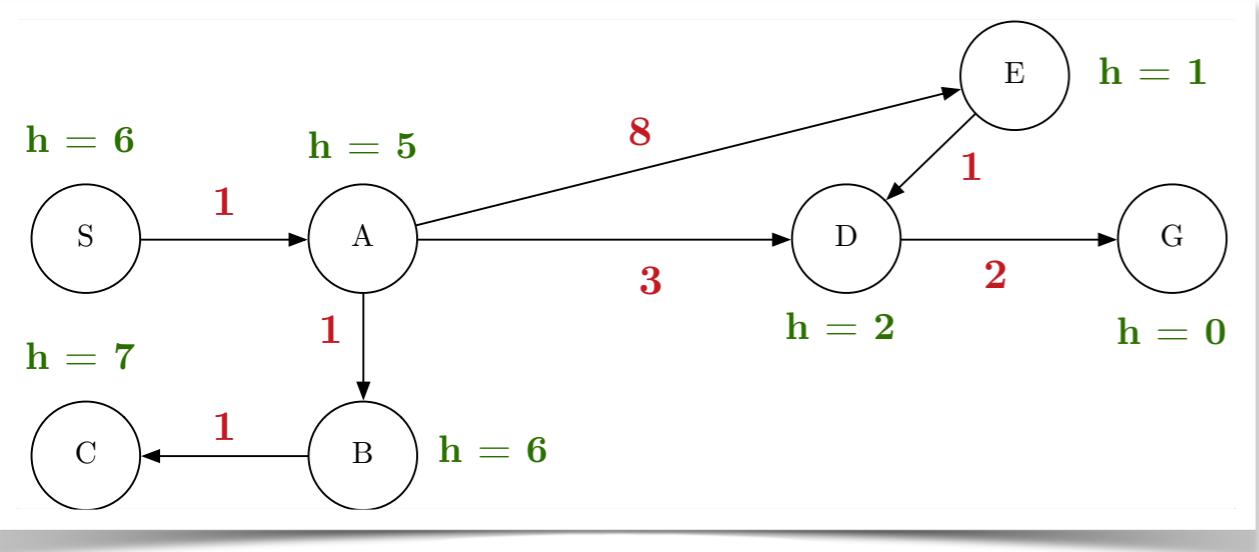
return no solution



Bonne nouvelle: pour le principe de l'algorithme, oui

Mauvaise nouvelle: il nous reste à étudier ses propriétés

Illustration - UCS vs Greedy vs A*



Le coût de chaque noeud est évalué par une fonction heuristique

Recherche à coût uniforme

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

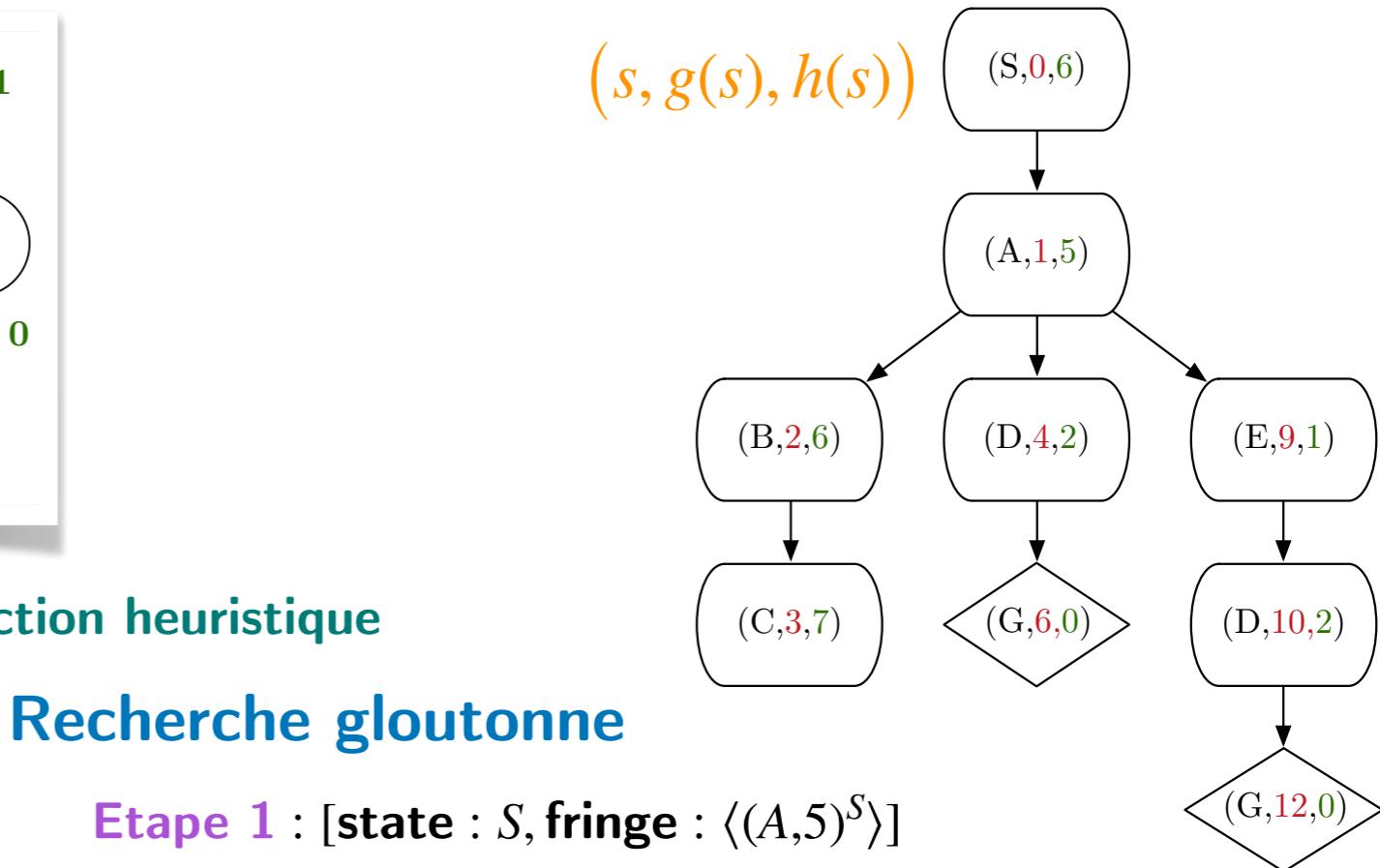
Etape 2 : [state : A^S , fringe : $\langle (B,2)^A, (D,4)^A, (E,9)^A \rangle$]

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*

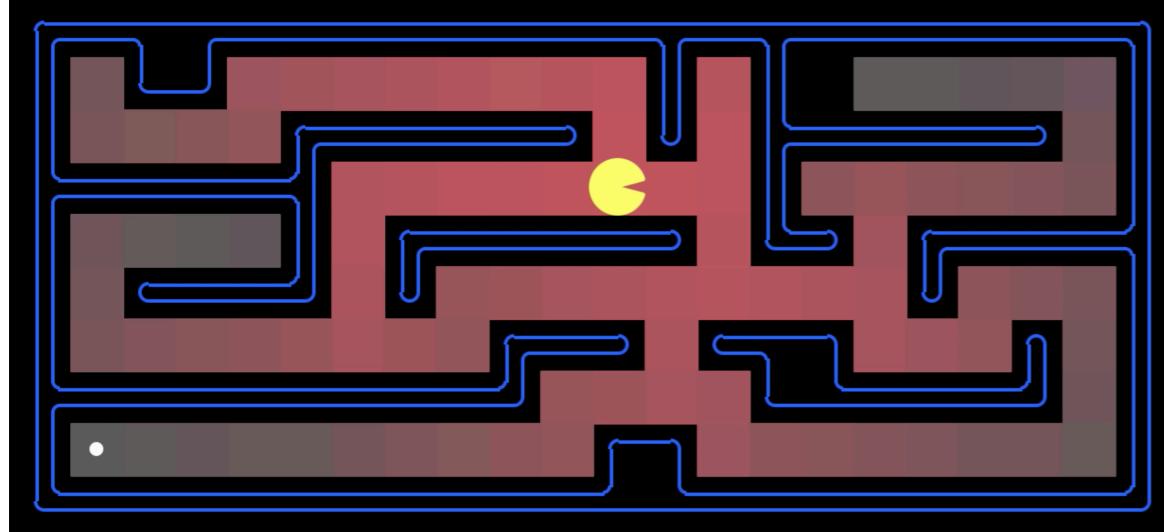
Etape 1 : [state : S , fringe : $\langle (A,6)^S \rangle$]

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

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

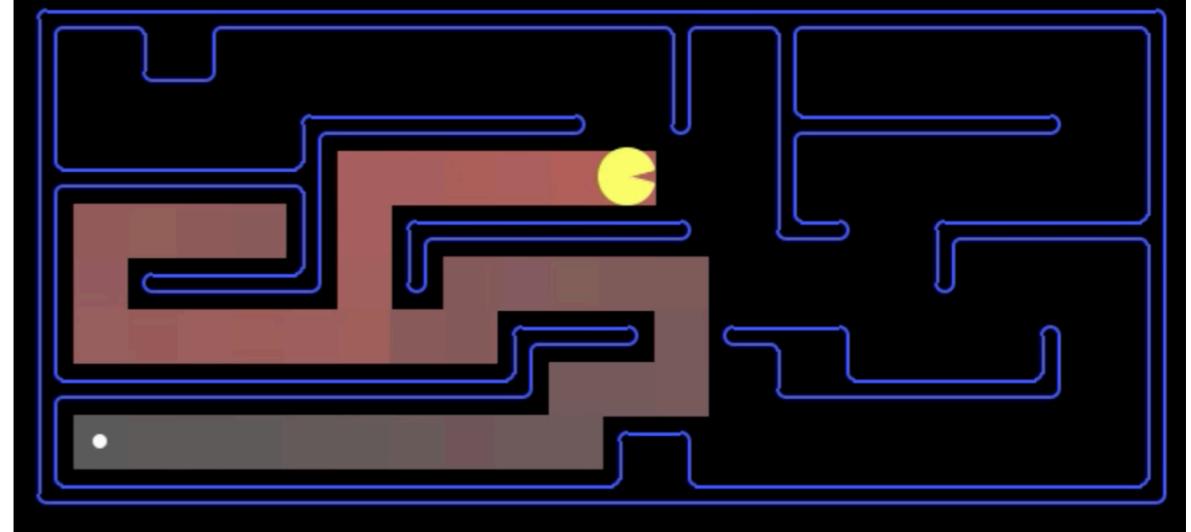
Etape 4 : [state : G^D , path : $S \rightarrow A \rightarrow D \rightarrow G$ (6)]

Recherche A* - exemple



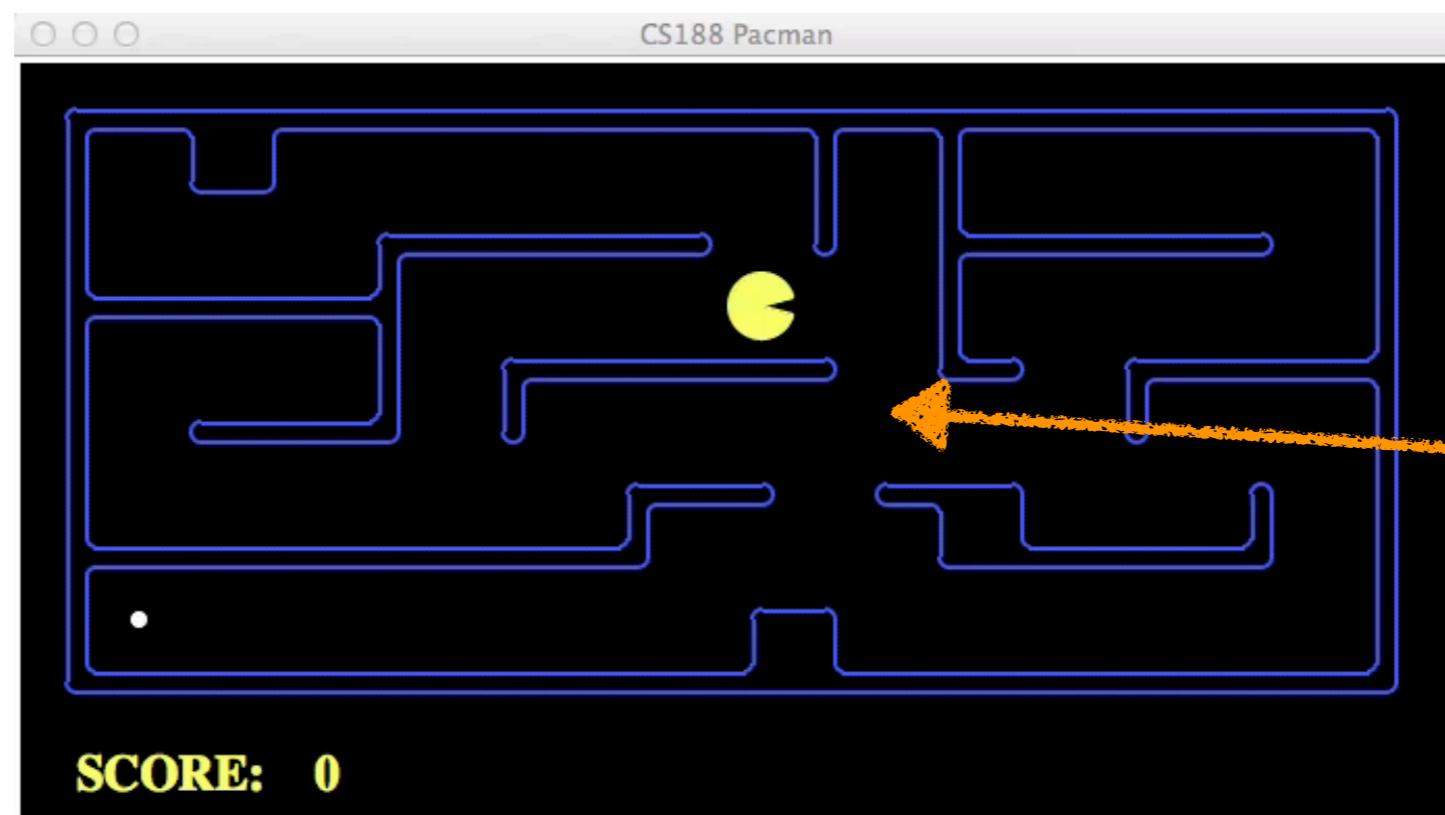
Recherche à coût uniforme

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

?

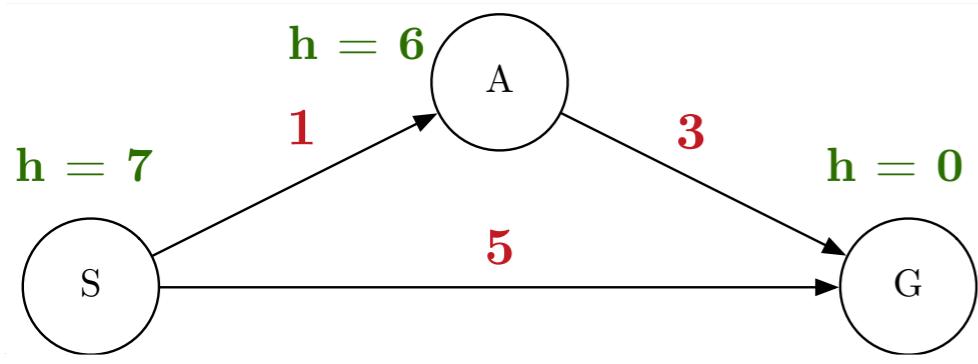
Est-ce que la recherche A* est complète ?

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

Intuition: similaire à la recherche à coût uniforme

?

Est-ce que la recherche A* est optimale ?



Exemple: regardons un cas simple

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

Mauvaise nouvelle: mêmes difficultés qu'une recherche gloutonne en cas de mauvaise heuristique

Observation: une mauvaise solution a eu une meilleure priorité qu'un noeud de la solution optimale

Cause: notre heuristique a donné une estimation trop coûteuse à un état

Une des forces d'A* est qu'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



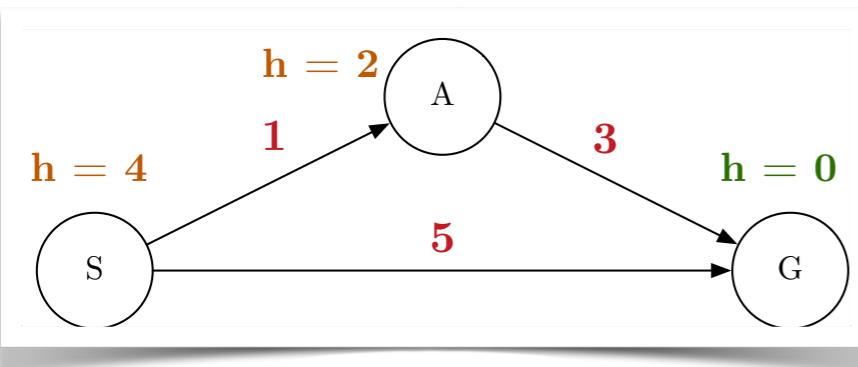
Optimalité d'une recherche A* en arbre

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

Principe: une heuristique admissible garantit que le coût optimal d'un noeud ne sera jamais sur-estimé

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

Intuition: l'heuristique ne va jamais 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 : $\langle (G, 5)^S, (G, 4)^A \rangle$]

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

n : noeud appartenant au chemin optimal

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

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

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

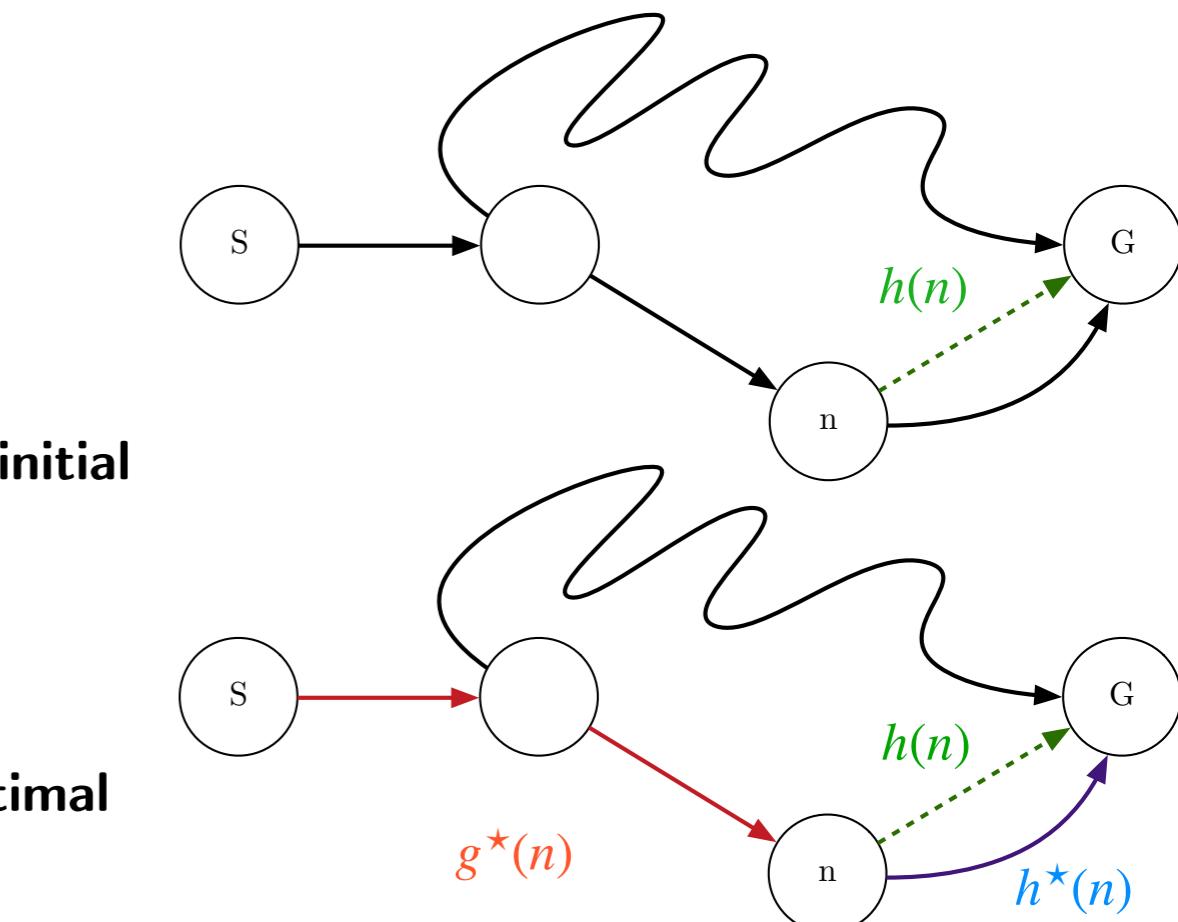
D'un autre côté, on a de (2):

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

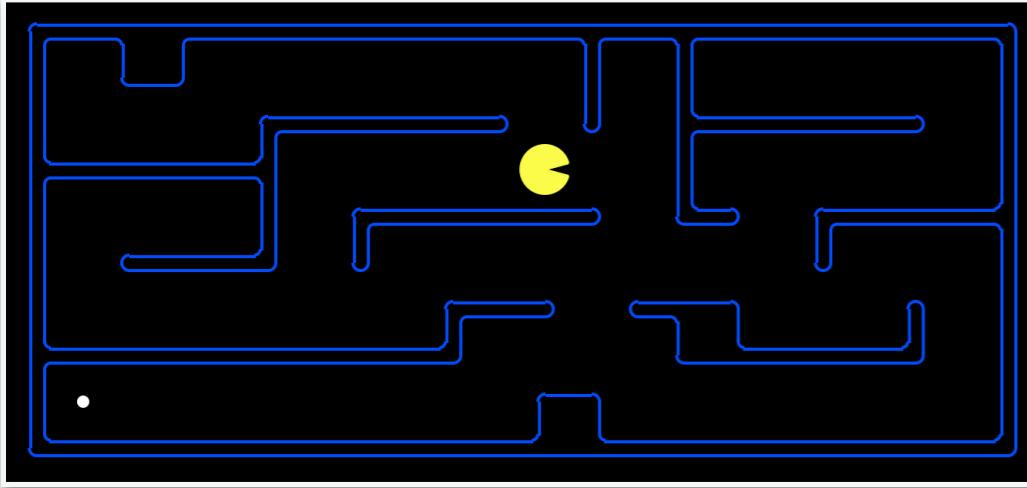
$f(n) \leq C^*$ (définition du coût du chemin optimal)



Contradiction !

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

Exemples d'heuristiques (non)-admissibles



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) $f(n) = g(n) + 0$

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

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)

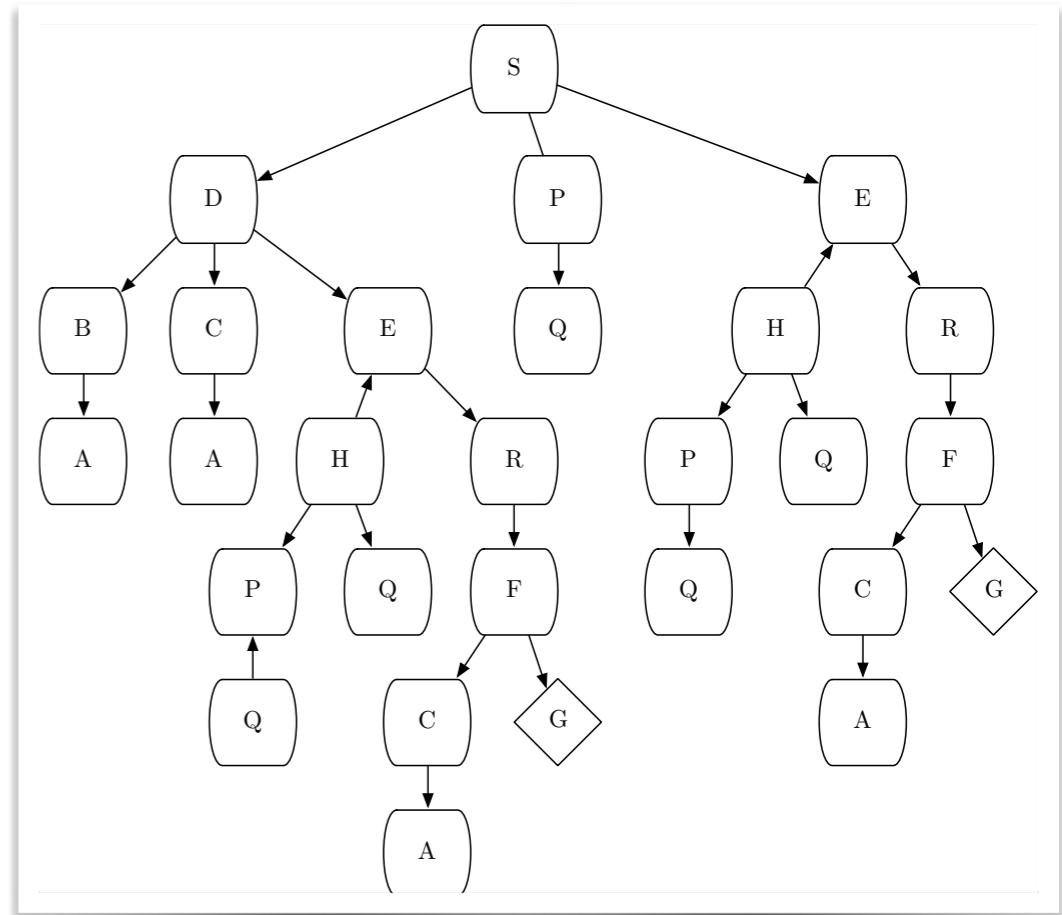
$$h(n) = h^*(n)$$

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

Table des matières

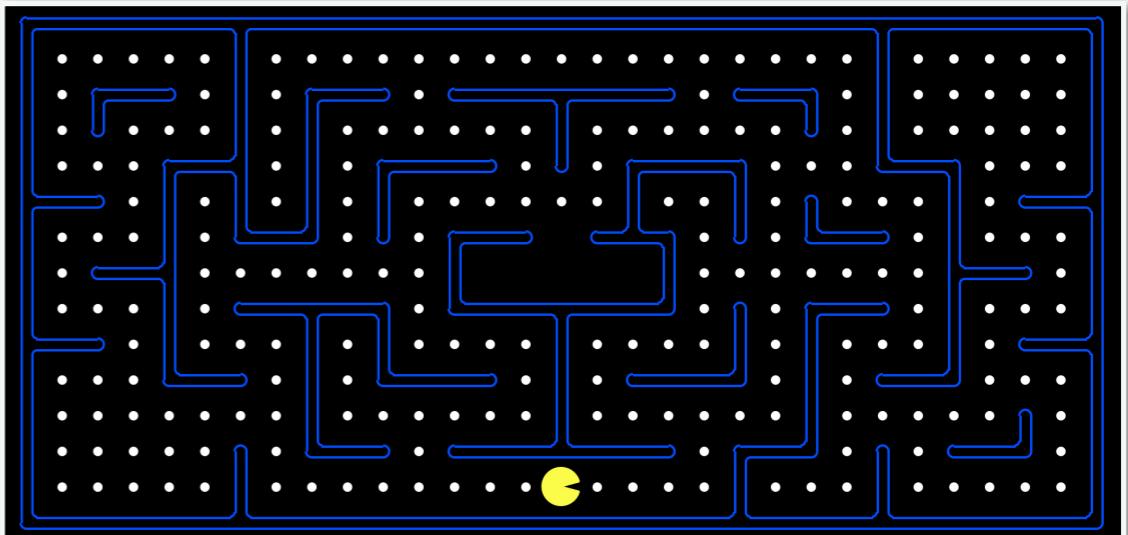
Stratégies de recherche

- ✓ 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) ?

Mauvaise nouvelle: très difficile de répondre à cette question !

Raison: il s'agit plus d'un art qu'une science exacte

Bonne nouvelle: il existe néanmoins des astuces et des bonnes pratiques



Relaxation d'un problème

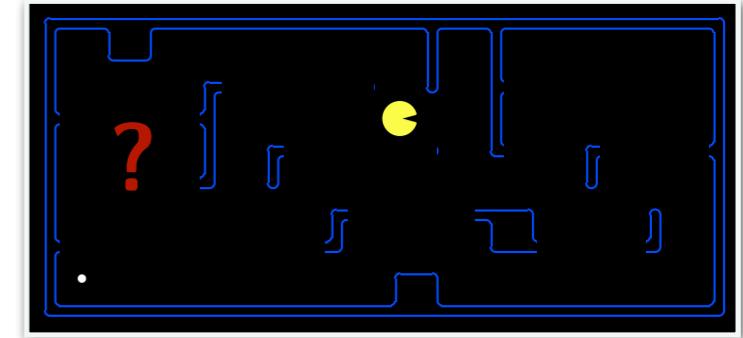
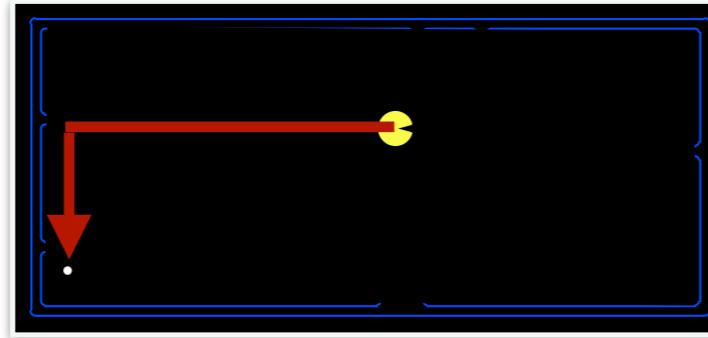
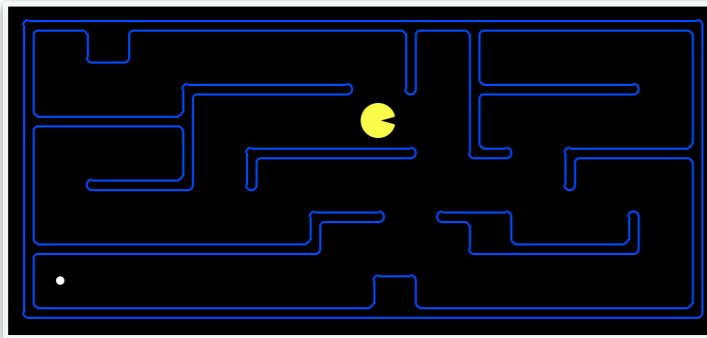
Problème auquel on a retiré certaines contraintes afin de le rendre plus facile à résoudre

Idée: utiliser une solution du problème relaxé pour obtenir une 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)

Exemple: relaxations possible pour Pacman



Une relaxation forte (aucun mur) met en évidence la distance de Manhattan comme heuristique

Construction d'heuristiques - techniques et bonnes pratiques



Collection d'heuristiques

Au lieu de choisir une heuristique, en évaluer plusieurs, et prendre la meilleure pour chaque noeud

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

Supposition: il faut qu'on ait plusieurs heuristiques disponibles

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

Inconvénient: coût de calcul supplémentaire pour obtenir les différentes heuristiques

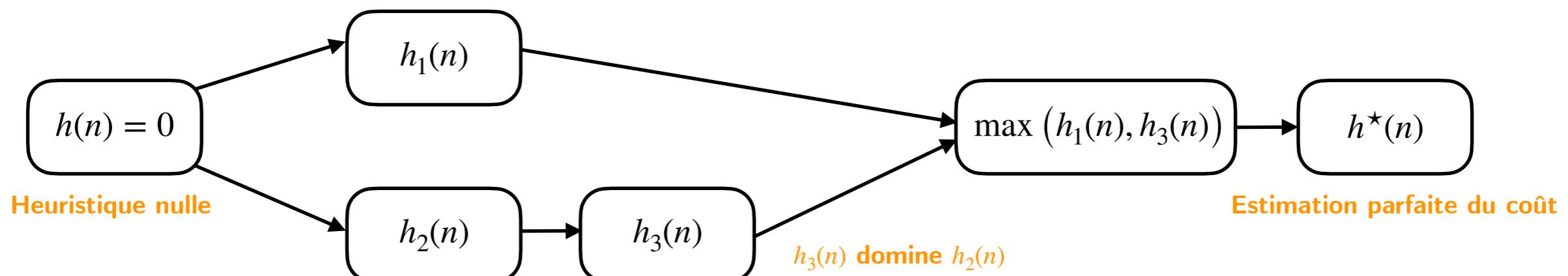


Heuristique dominante

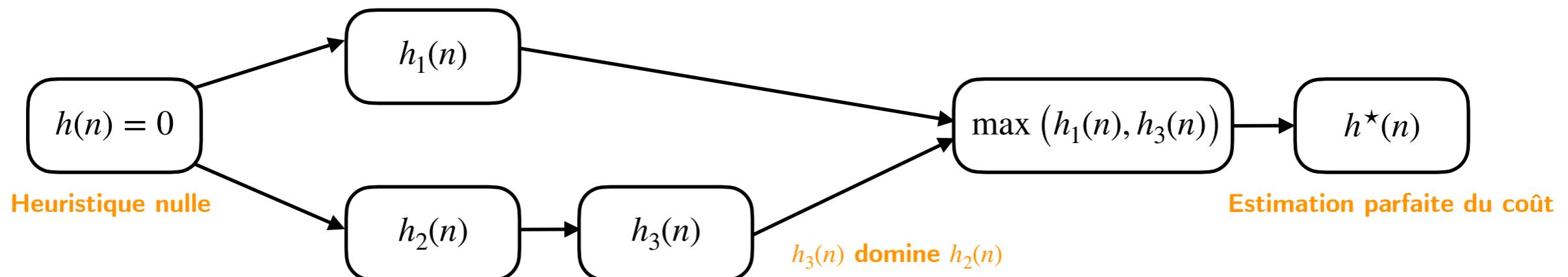
Une heuristique h_2 domine h_1 si elle est strictement plus précise que cette dernière

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

Graphe de dominance: relations explicites des dominances entre vos différentes heuristiques



Construction d'heuristiques - considérations pratiques



Difficile à répondre ! il existe des états où $h1$ est plus précise et inversement

Réponse: aucune heuristique ne domine l'autre

Possibilité de conception: prendre le maximum entre les deux pour l'estimation d'un état

? En pratique, est-il préférable d'utiliser $h2(n)$ ou $h3(n)$?

Certitude: l'heuristique $h3$ domine $h2$, elle est donc plus précise

Incertitude: on n'a aucune information sur le temps de construction de l'heuristique

Obtenir $h3$ est potentiellement beaucoup plus lent que $h2$ (à l'extrême, h^* est inconcevable à obtenir)

En pratique: une heuristique dominée mais plus rapide peut être plus intéressante à utiliser

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

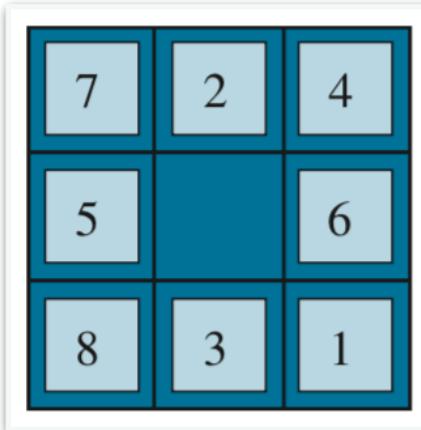
Exemple illustratif - 8-puzzle

Problème du 8-puzzle

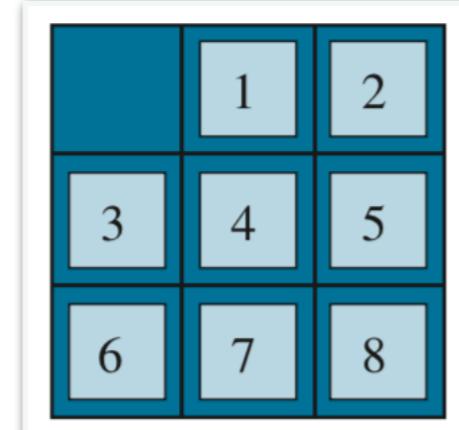
Objectif: déplacer les pièces afin de les ordonner

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

Coût: effectuer le moins de mouvements possibles



Etat initial (s)



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 ?

Observation: 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

Analyse 3: 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



Conseil: 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

Conseil: lorsque vous avez plusieurs bonnes heuristiques, prenez leur maximum

En pratique: il est parfois utile d'avoir des heuristiques non-admissibles

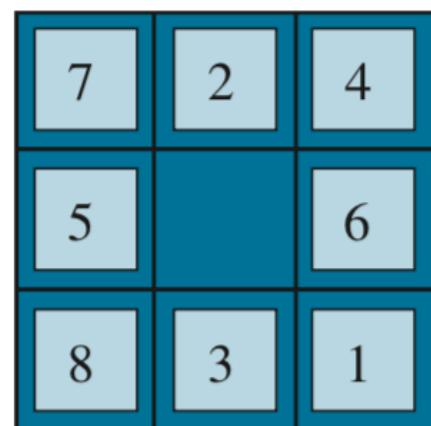
Construction d'heuristiques - techniques supplémentaires

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 précises des pièces ne sont pas prises en compte)

Conception d'heuristiques par apprentissage automatique

Idée: utiliser de l'apprentissage automatique pour construire l'heuristique

Déceler automatiquement les caractéristiques (+ le poids)

Un de mes sujets de recherches principaux :-)

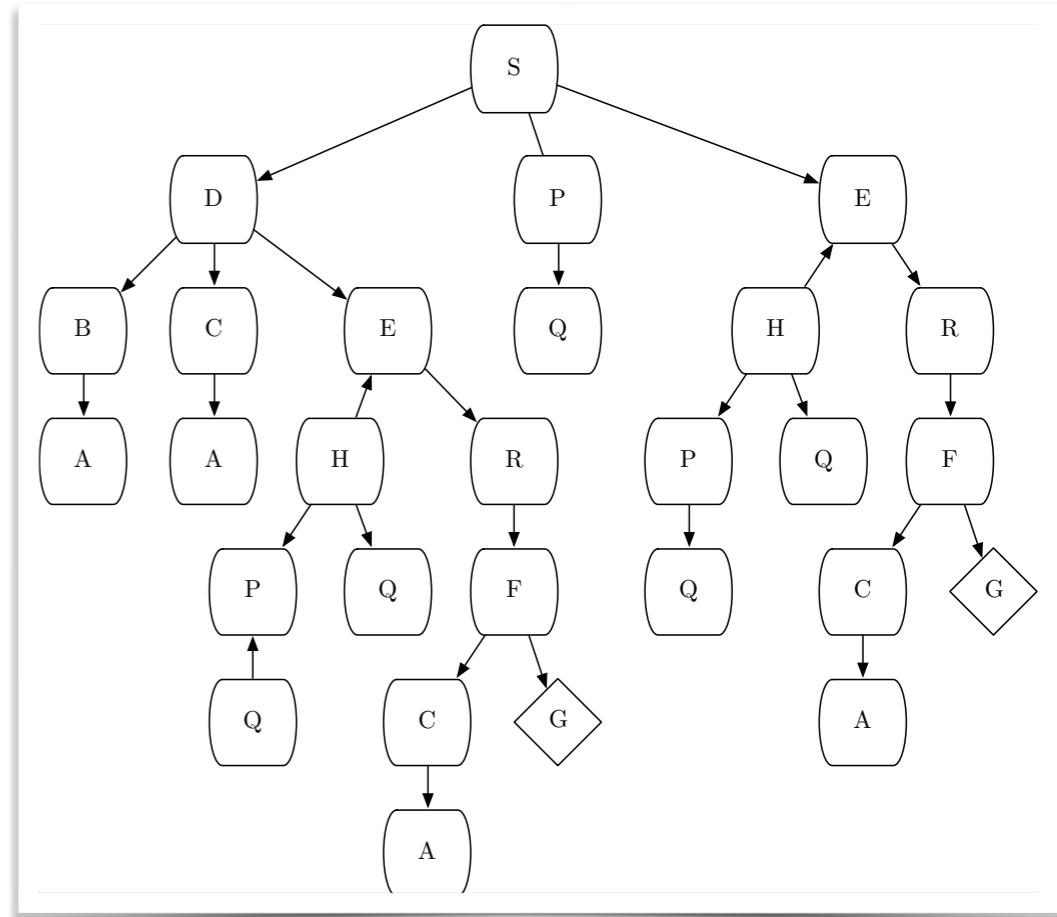


Combinatorial Optimization and
Reasoning in
Artificial Intelligence
Laboratory

Table des matières

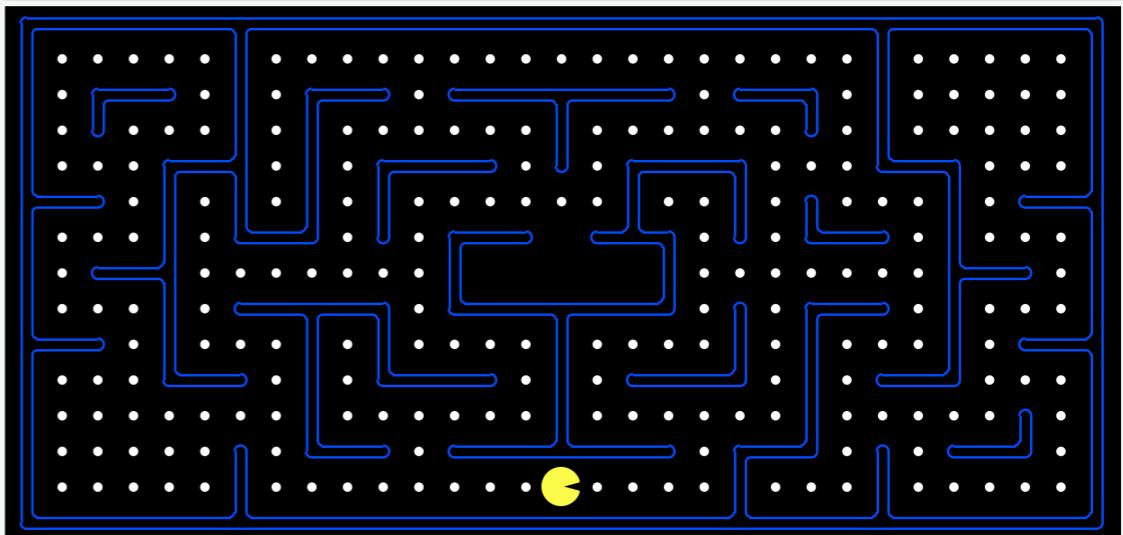
Stratégies de recherche

- ✓ 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

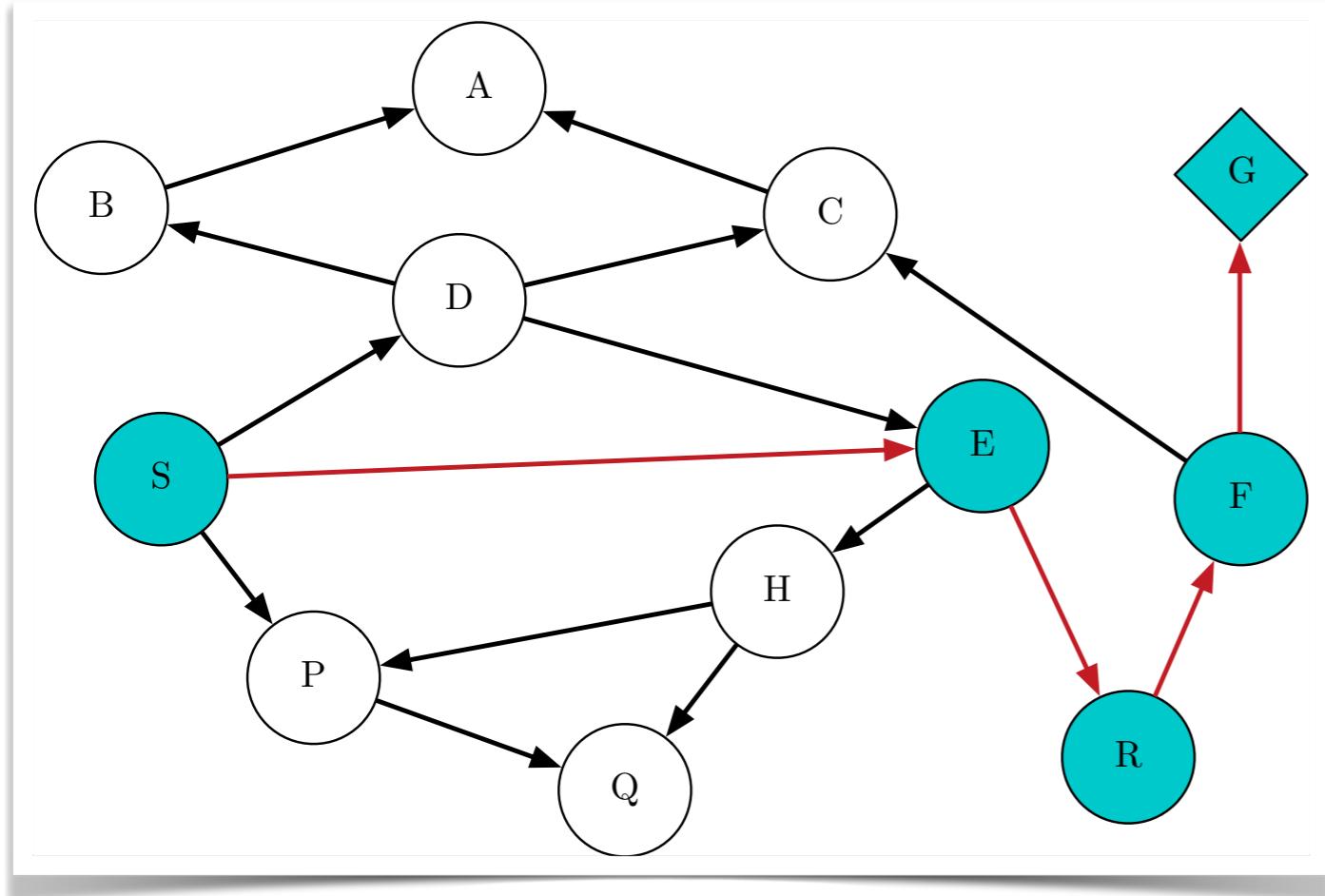


Rappel: recherche en graphe

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

Rappel: on a vu qu'il était possible de construire une recherche en graphe

Principe: 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 :  
             $C = \{c \in \text{succesors}(s, P) \mid c \text{ not in } V\}$   
             $L = (L \cup C) \setminus s$   
             $V = V \cup s$   
    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



Bonne nouvelle: nos algorithmes peuvent être utilisés **presque** sans modification

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

Inconvénient majeur: la consommation mémoire sera plus importante

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*



Mauvaise nouvelle: l'algorithme A* souffre d'une difficulté supplémentaire

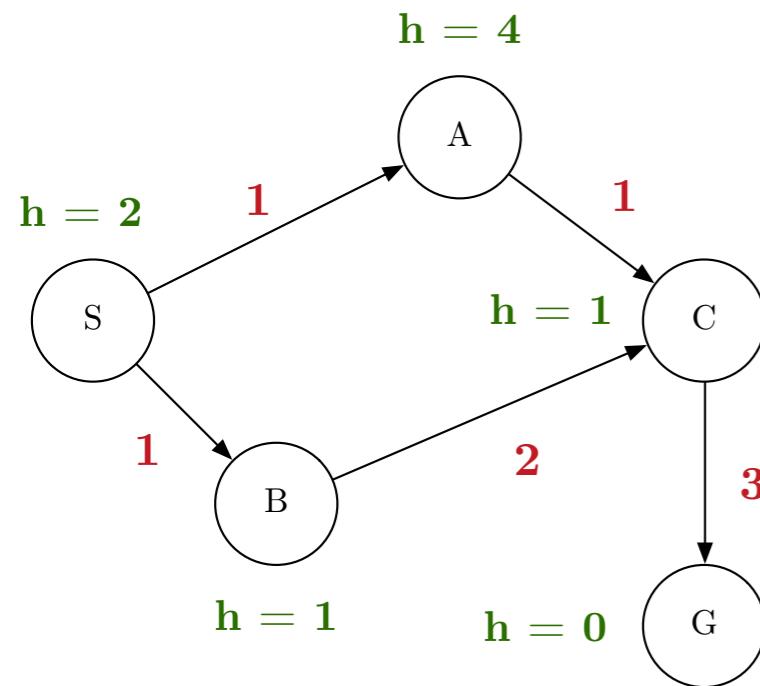
Une heuristique admissible n'est plus suffisante pour garantir l'optimalité

On doit avoir une propriété plus forte pour cela

Introduction du concept d'heuristique consistante

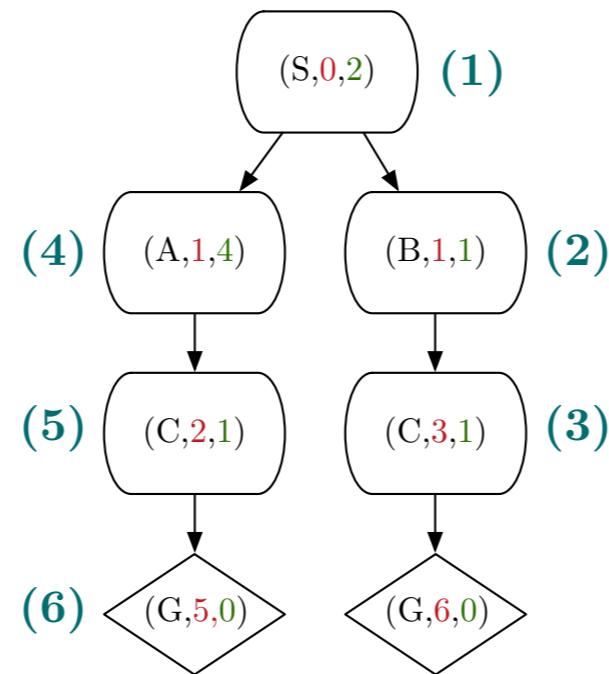
Recherche A* en graphe: cas pathologique

Cas pathologique



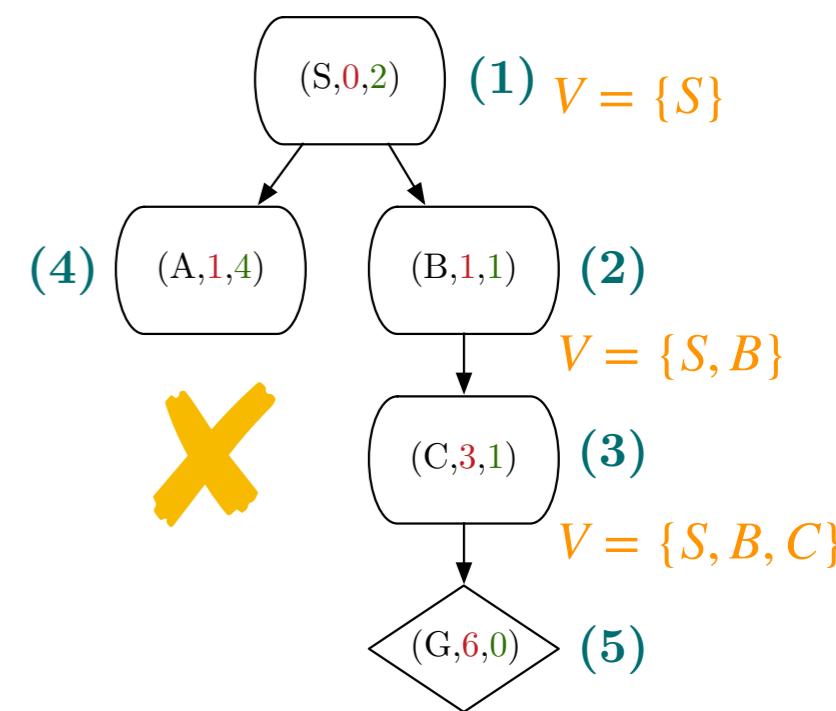
Notez que l'heuristique est admissible

Recherche A* en arbre



Solution optimale obtenue

Recherche A* en graphe



Solution non-optimale obtenue

?

Quel est le soucis selon vous ?

Problème: le noeud $(C,3,1)$ a été étendu avant $(C,2,1)$, or $(C,3,1)$ n'appartient pas au chemin optimal

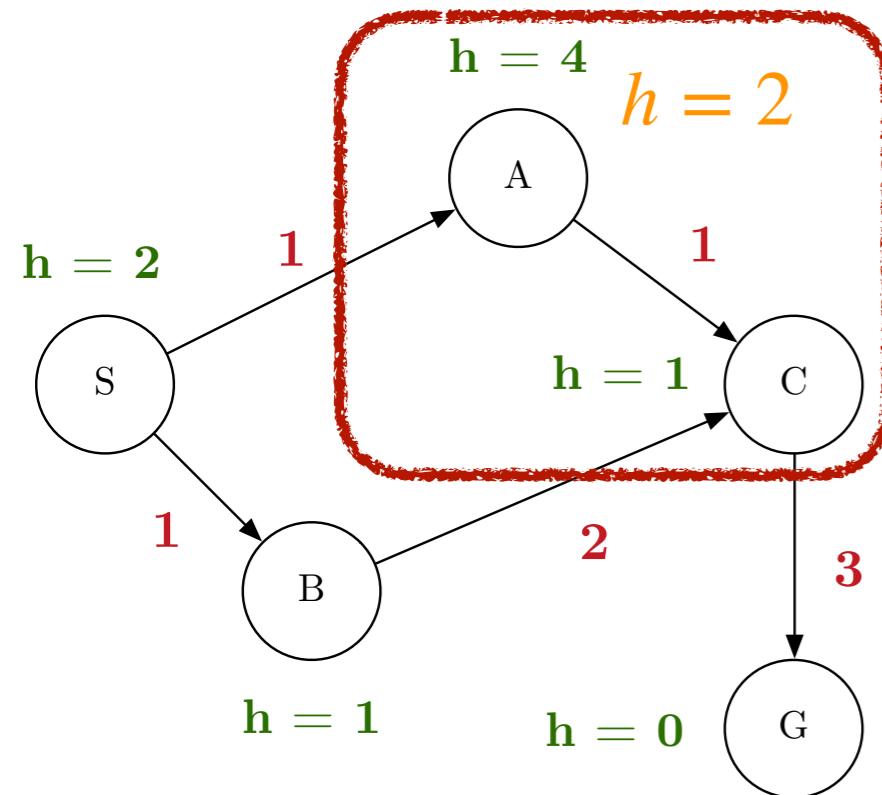
Conséquence: une recherche en graphe ne va jamais étendre $(C,2,1)$, car l'état C a déjà été exploré

Conséquence suivante: 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



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

$4 - 1 \leq 1$ (**Incorrect**)

Coût de l'arc: 1

Estimation heuristique: 3

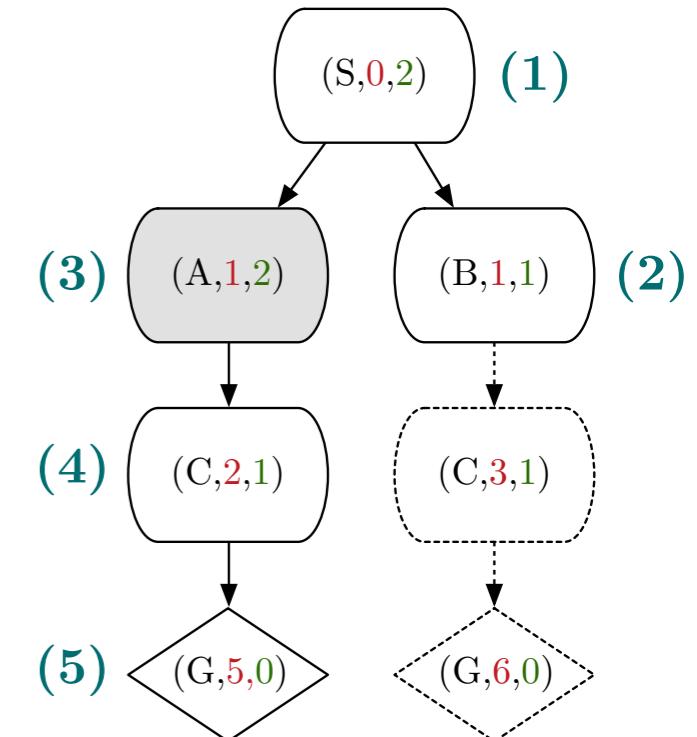
Heuristique non consistante

Avec $h(A) = 2$

$2 - 1 \leq 1$ (**Correct**)

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 !)

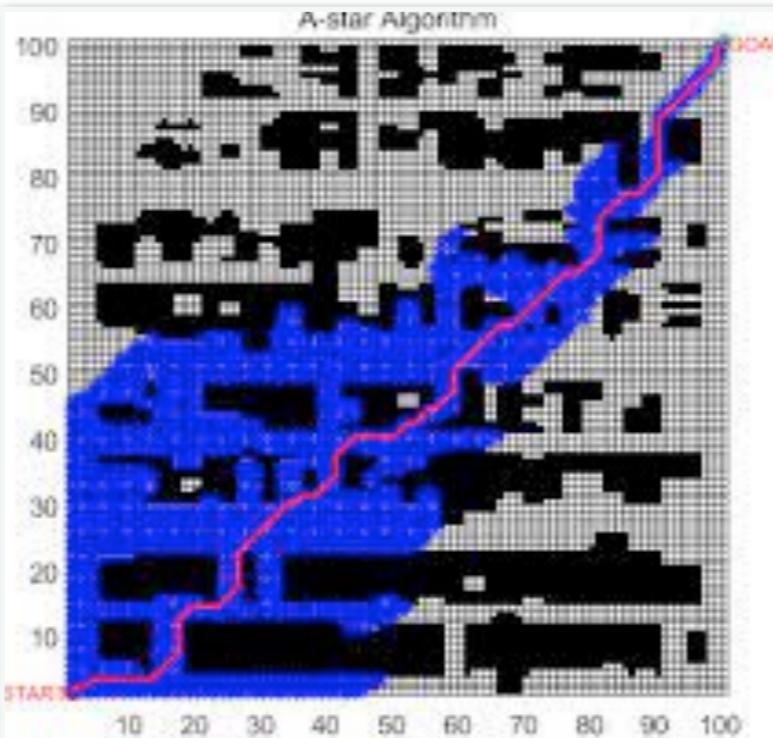
Propriété 2: une heuristique consistante est optimale pour A* avec une recherche en graphe

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'actions 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.)

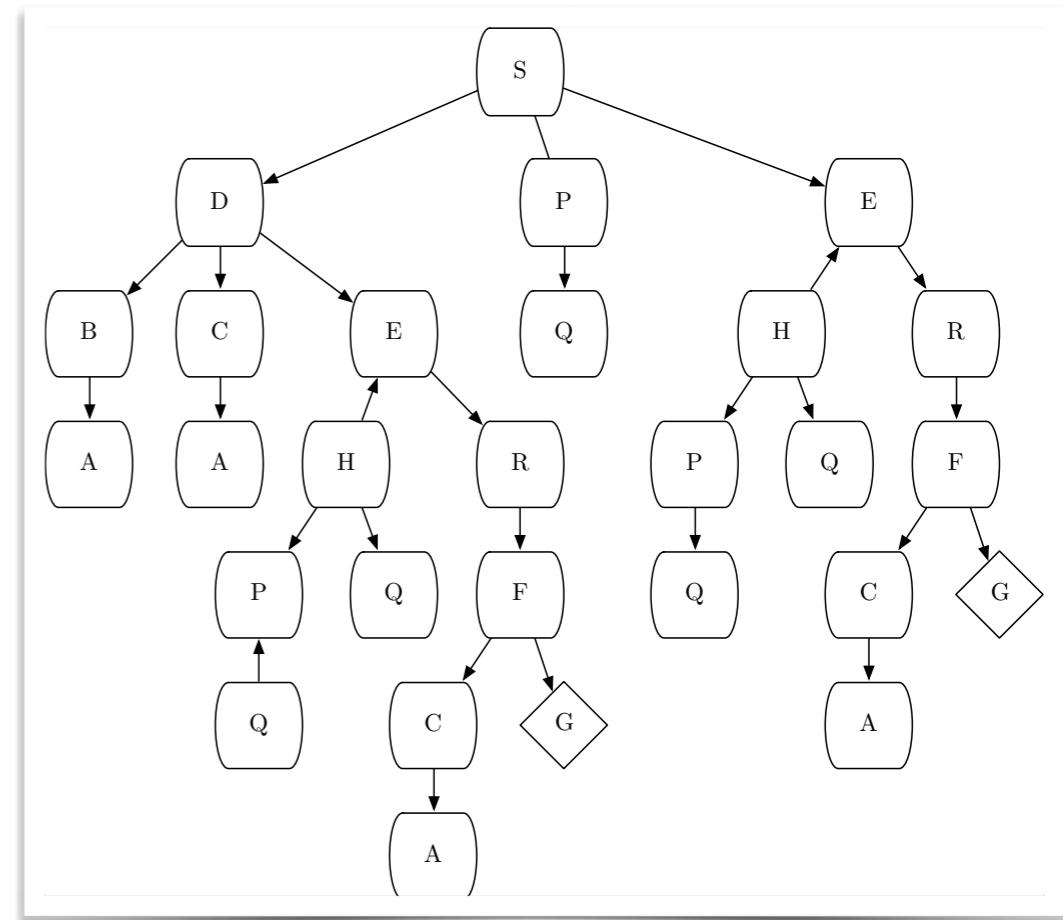
Exemples: réseaux routiers, réseaux sociaux, représentation de connaissances, etc.



Table des matières

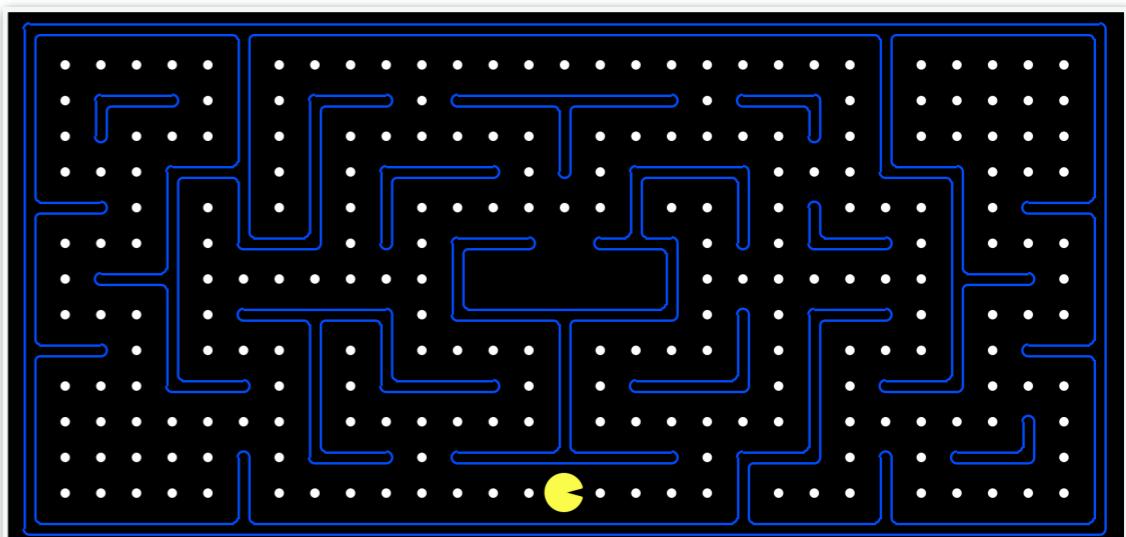
Stratégies de recherche

- ✓ 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



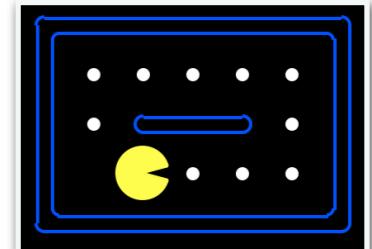
Synthèse des notions vues

Problèmes de recherche

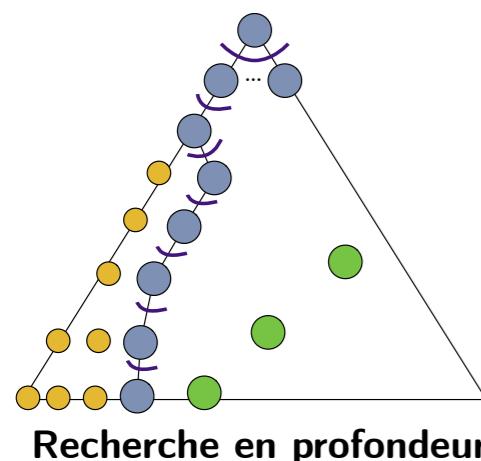
Motivation: planifier d'une séquence d'actions pour aller d'un état initial à un état final

Objectif idéal: trouver la séquence engendrant le moins de coûts

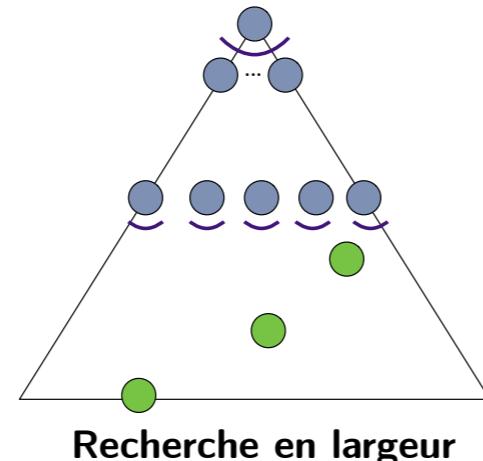
Patron de résolution générique: recherche en graphe ou en arbre



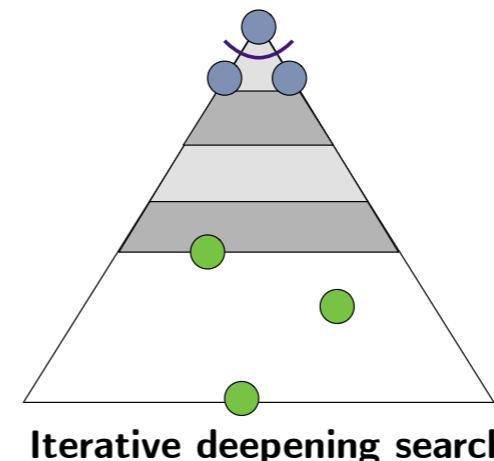
Stratégies sans information



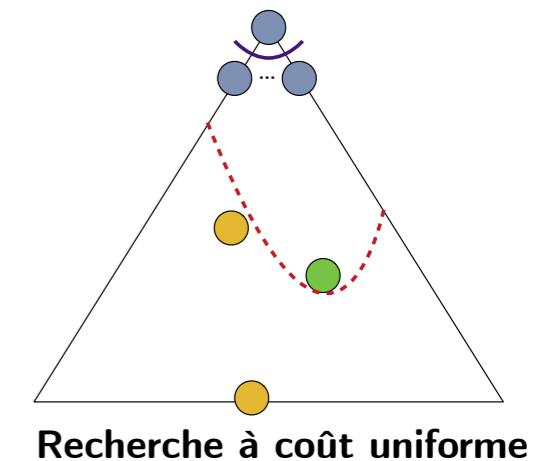
Recherche en profondeur



Recherche en largeur

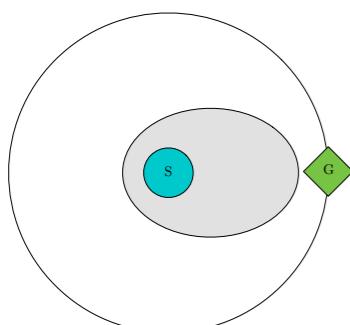


Iterative deepening search



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

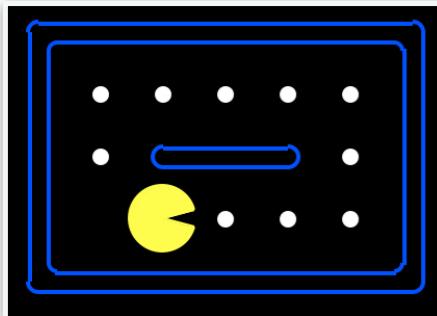
Heuristique admissible: A* avec une recherche en arbre est optimal

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

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

Conseils pratiques

Stratégie de recherche



Pour commencer: considérez d'abord la recherche en graphe

Ensuite: 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



Pour commencer: faites des heuristiques simples et complexifiez les progressivement

Technique de relaxation: 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

Au long de votre travail: testez régulièrement vos algorithmes sur des situations simples

A vérifier: la solution obtenue est bien optimale (sur des petits problèmes)

A vérifier: les noeuds étendus sont bien ceux attendus

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

Attention: 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

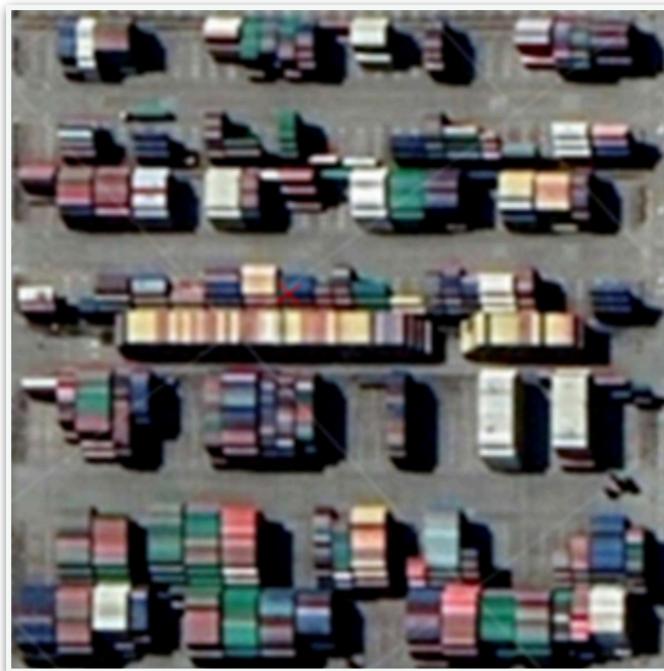
Que faire quand ces propriétés ne sont pas présentes ?

(2) Recherche avec adversaire

Comment agir en présence d'un adversaire ?

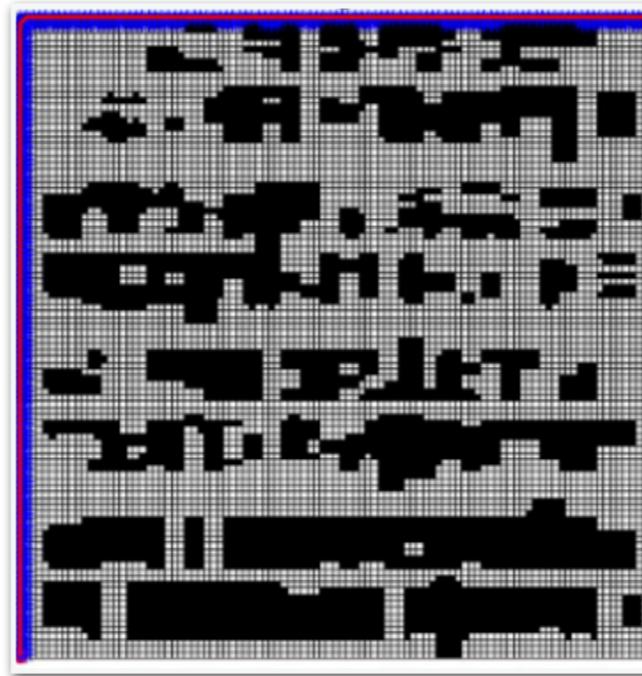
Sujet du prochain module

Dernières illustrations

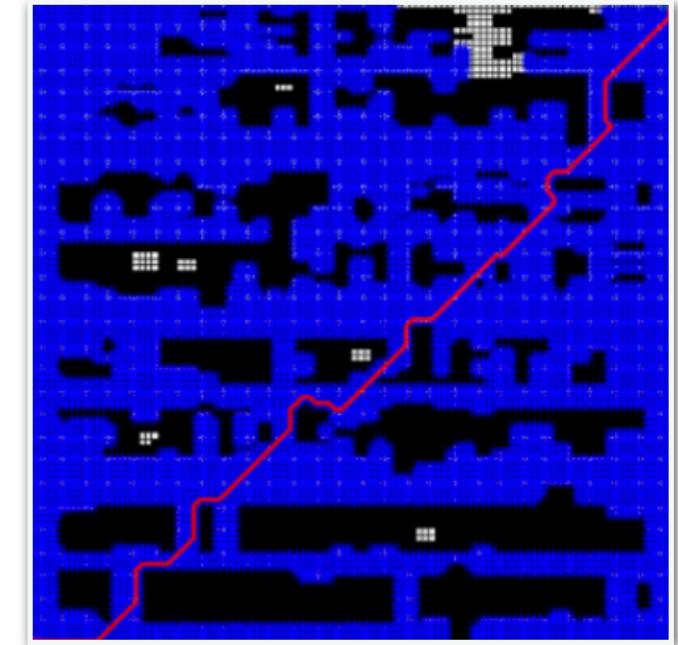


Goal

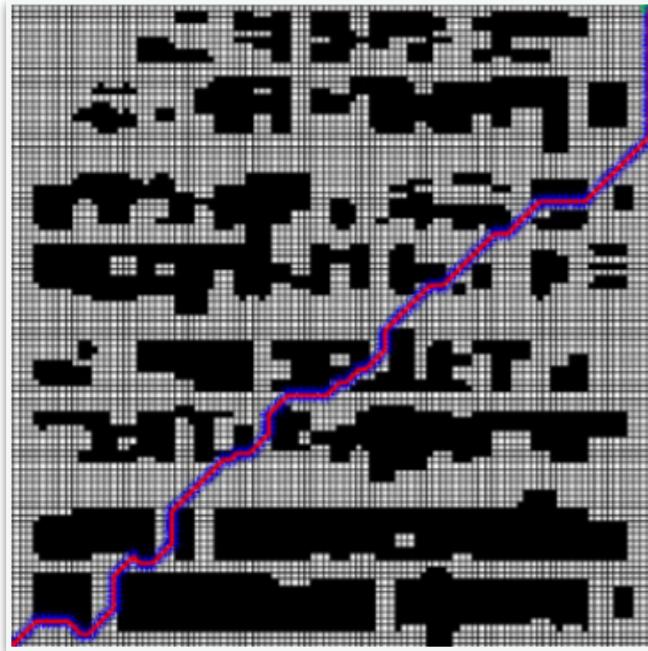
Start



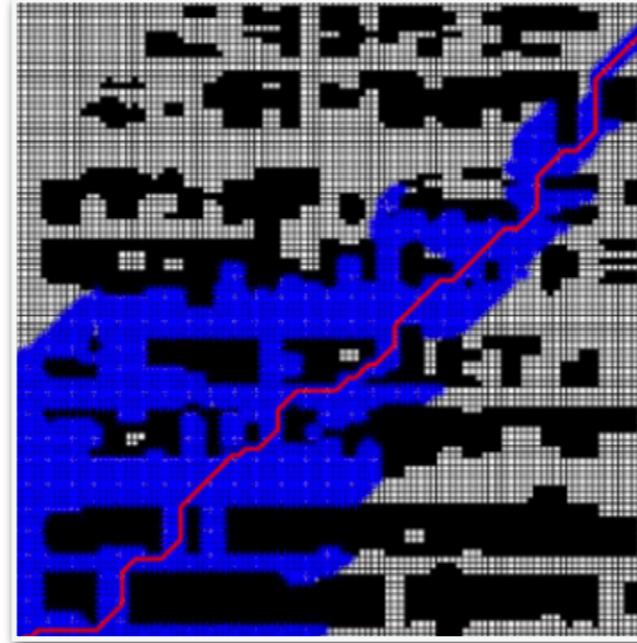
Recherche en profondeur



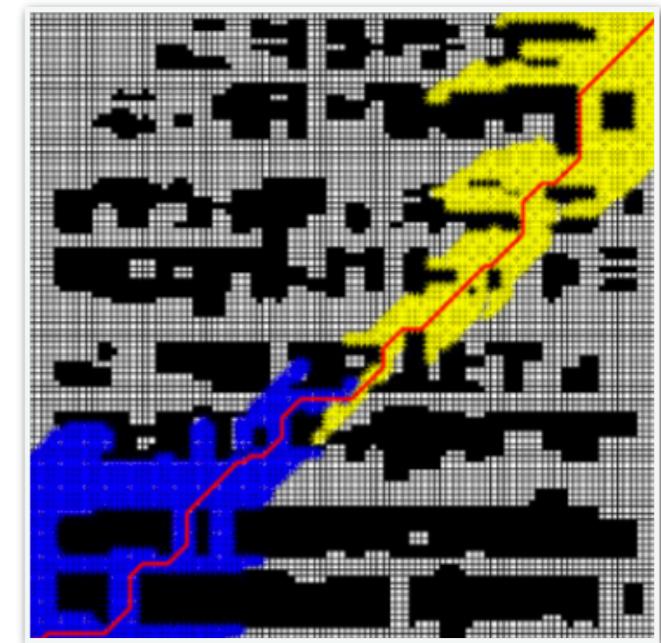
Recherche en largeur



Recherche gloutonne



Recherche A*



A* bidirectionnel

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





DALLE: *Pacman lost in a labyrinth on an oil painting*