



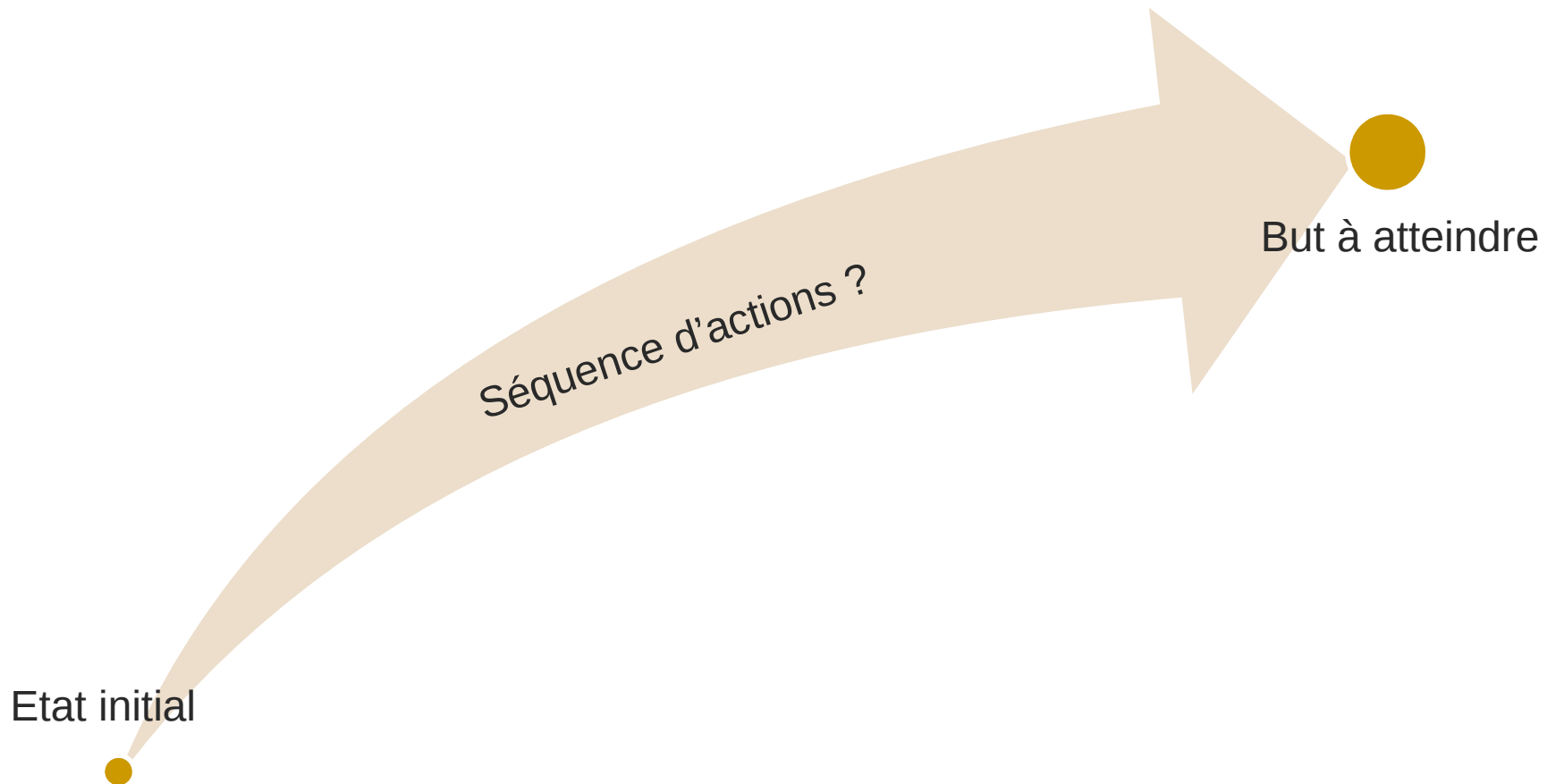
# Résolution de problème

2I013 : Jeu à 2 joueurs  
Cours 2

# [ Jeu à deux joueurs ]

- Dans le cadre d'un jeu à deux joueurs, décider du meilleur coup à effectuer revient à résoudre un problème où :
  - L'objectif est de gagner la partie
    - Prendre des décisions successives menant à une situation de victoire
  - Mais où
    - Les situations successives dépendent d'un joueur adverse qu'on ne connaît pas a priori
    - Chaque décision doit être prise en un temps raisonnable
- Résolution de problèmes basée sur des estimations de gains / risques liées aux décisions de jeu

# [ Résolution de problème ]



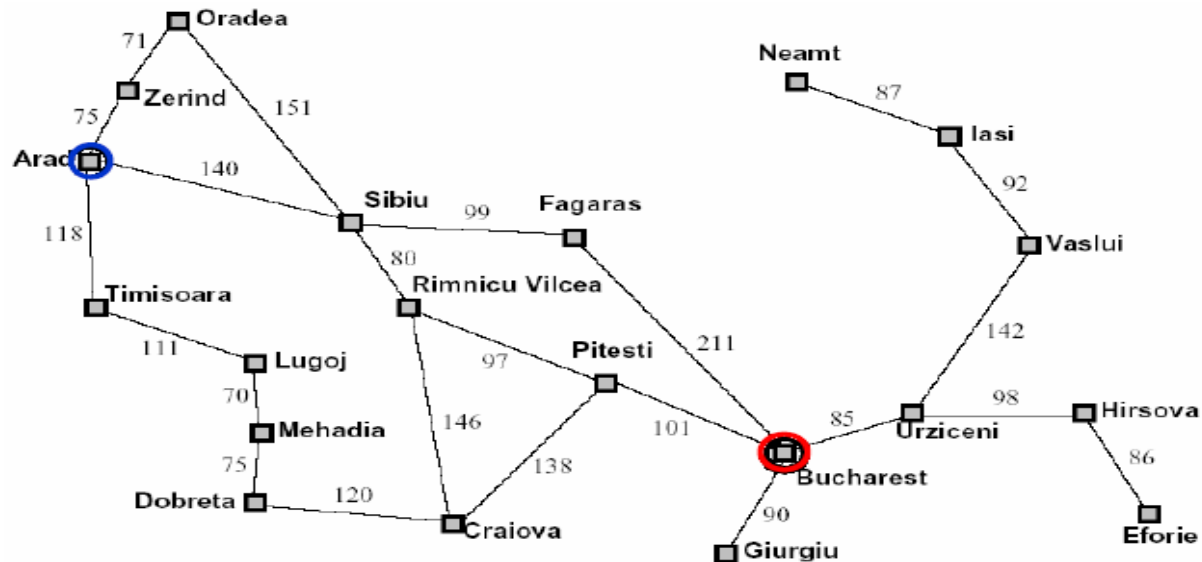
# Résolution de problème

- La solution d'un problème s'obtient sous la forme d'une séquence d'actions menant à une solution souhaitée.
- Pour cela il faut :
  - Exprimer l'objectif
  - Spécifier l'espace d'états du problème : dans quelles situations peut-on se trouver ?
  - Définir les transitions entre états du problème
  - Adopter une politique de recherche de la solution

# Définition d'un problème de recherche

- Espace d'états
  - chaque état est une représentation abstraite de l'environnement
  - l'espace d'état est généralement discret
- Fonction de transition
  - représentation abstraite des actions possibles
  - fonction : [ état  $\rightarrow$  sous-ensemble d'états successeurs]
  - Transitions déterministes ou probabilistes
- Test-solution (ou test de but)
  - habituellement une condition à satisfaire
  - parfois la description explicite d'un état
- Coût du chemin (ou fonction-coût)
  - fonction : [Chemin  $\rightarrow$  Nat ]
  - Habituellement : coût du chemin = somme des coûts de ses étapes

# Exemple de problème: Recherche d'itinéraire



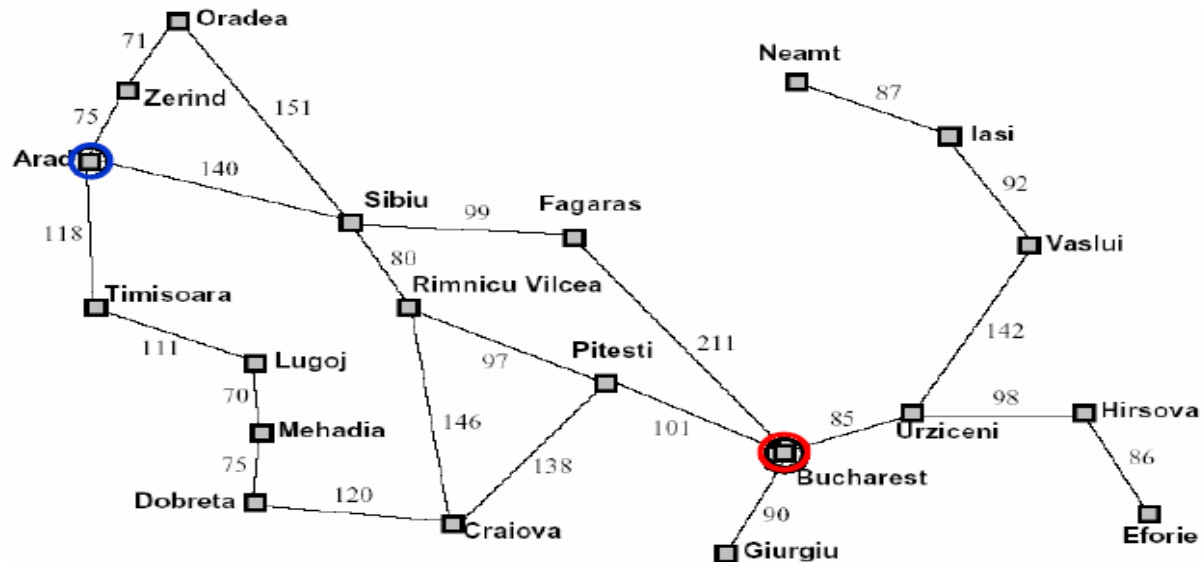
états?

actions?

Test de but?

Cout de chemin?

# Exemple de problème: Recherche d'itinéraire



états? Villes (e.g., Arad, Zerind, etc...)

actions? Déplacements selon transitions (e.g., Arad → Zerind, Sibiu → Fagaras, etc ...)

Test de but? Test explicite : "*position == Bucarest*" ?

Cout de chemin? Somme des poids des arcs empruntés (*min = 418*)

# Exemple de problème: Le puzzle à 8 pièces

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

états?

actions?

Test de but?

Cout de chemin?



# Exemple de problème: Le puzzle à 8 pièces

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

états? Configurations possibles du puzzle

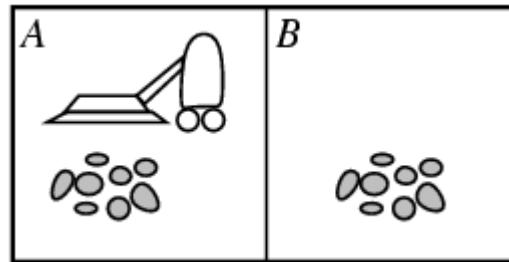
actions? Déplacer une pièce vers G, D, H, B

Test de but? Configuration dans un état particulier (*Goal State*)

Cout de chemin? Nombre d'actions du chemin (nb déplacements)



# Exemple de problème: L'aspirateur



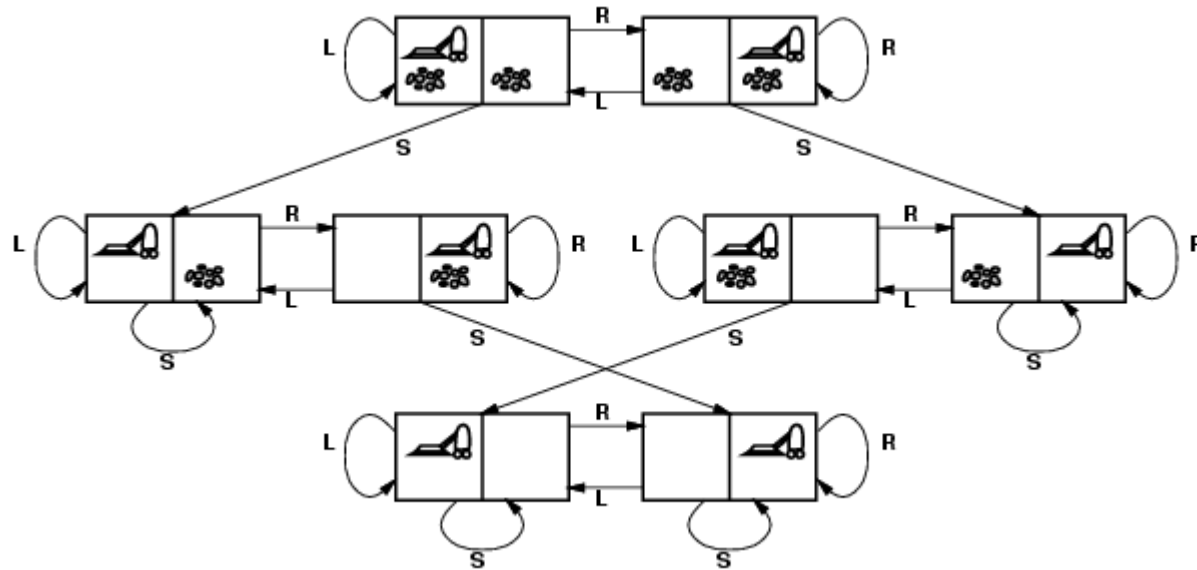
états?

actions?

Test de but?

Coût de chemin?

# Exemple de problème: L'aspirateur



états? Présence de poussière et emplacement du robot

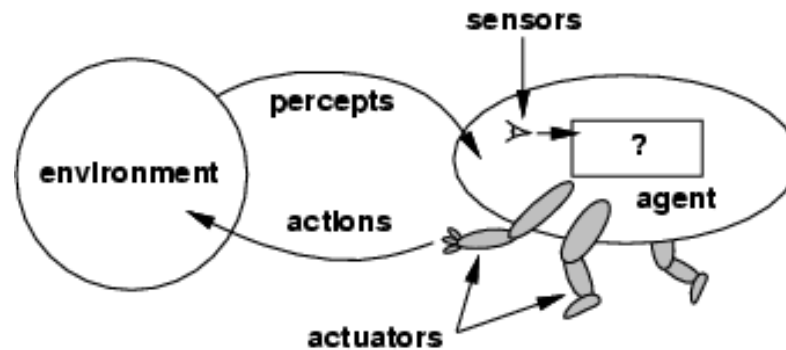
actions? Gauche (L), Droite (R), Aspirer (S)

Test de but? Plus de poussière nulle part

Coût de chemin? Nombre d'actions

# [Agent pour la résolution de problème]

- IA => construire des agents « intelligents » permettant de résoudre un ou plusieurs problèmes
- Un **agent** est une entité qui perçoit et qui agit



- Le programme **agent** s'exécute sur une architecture physique
- agent = architecture + programme

# [Agents pour la résolution de problème]

- De façon abstraite un agent est une fonction de l'espace des historiques de perceptions dans l'espace des actions:

$$[f: P^* \rightarrow A]$$

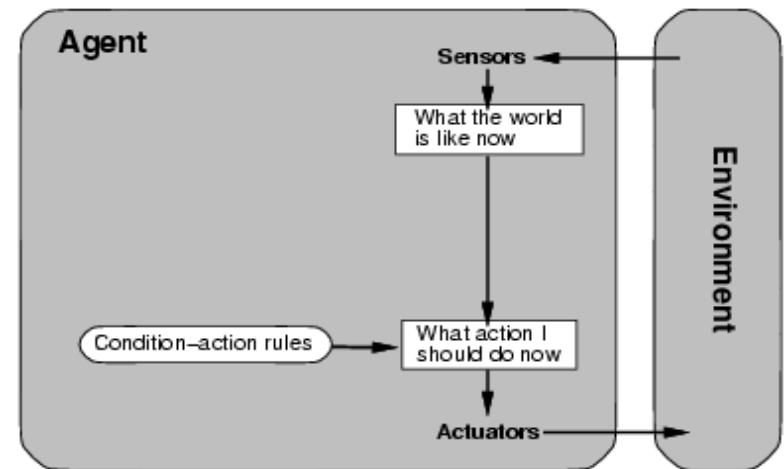
- Pour une classe d'environnement et de tâche donnée on cherche l'agent (ou la classe d'agents) qui a la meilleure *"performance"*
- Problème: Limitations calculatoires (e.g. explosion combinatoire etc) rend cette recherche impossible généralement.
  - Objectif : construire le meilleur programme étant donné les ressources machines à disposition.

# [Agents pour la résolution de problème]

- 5 types d'agents
  - Agents à réflexes simples
  - Agents à réflexes basés sur des modèles
  - Agents basés sur des buts
  - Agents basés sur l'utilité
  - Agents apprenants

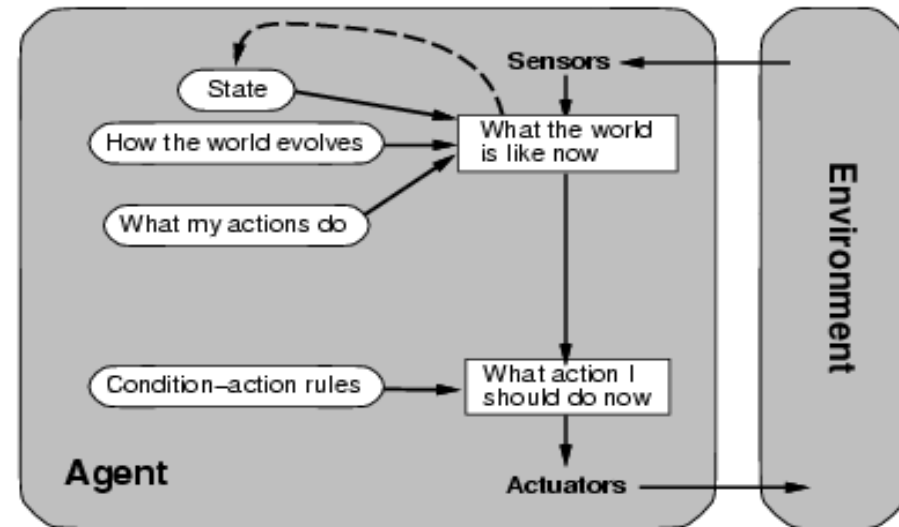
# [Agents pour la résolution de problème]

- Agents à réflexes simples
  - Application de règles simples *if... then...* basées sur des perceptions de l'état de l'environnement
  - Pas d'historique des états précédents
  - Peu efficace dans des environnements partiellement observables (cycles infinis possibles)



# [Agents pour la résolution de problème]

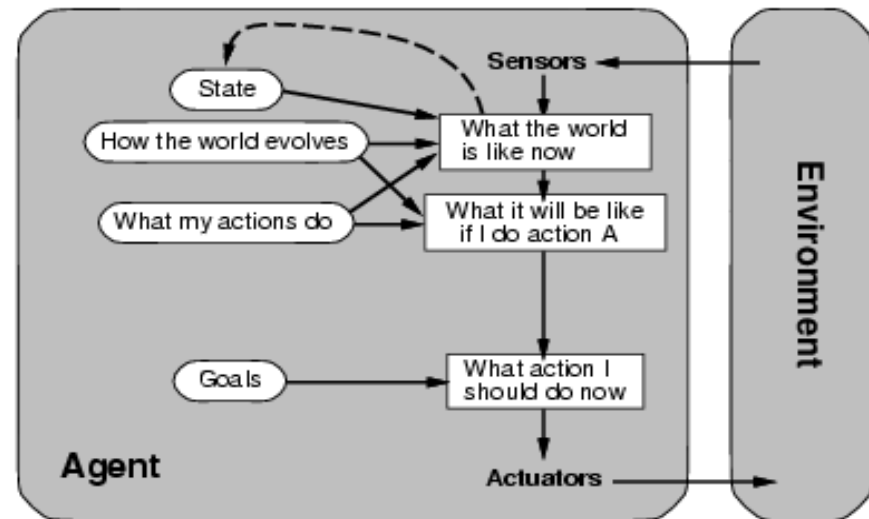
- Agents à réflexes basés sur des modèles
  - Application de règles simples *if... then...* basées sur un modèle de l'environnement
  - La partie non observable de l'environnement est estimée à partir des états rencontrés et d'un modèle des conséquences des actions sur cet environnement.





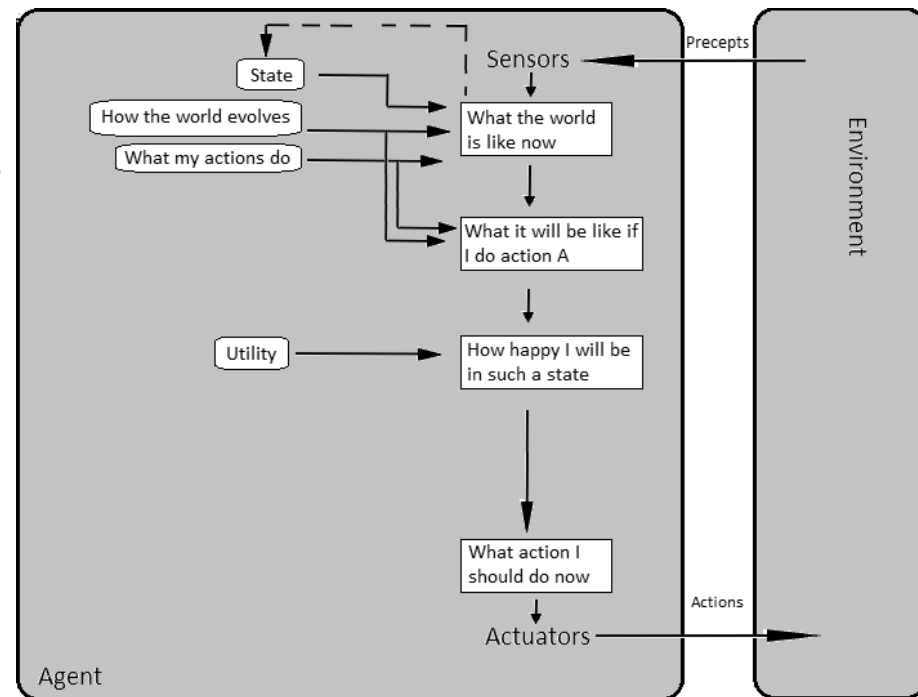
# [Agents pour la résolution de problème]

- Agents basés sur des buts
  - Le modèle de l'environnement estime l'état présent du monde ainsi que son état futur selon les différentes actions possibles.
  - Sélection des actions selon qu'elles permettent de remplir différents buts pre-déterminés.



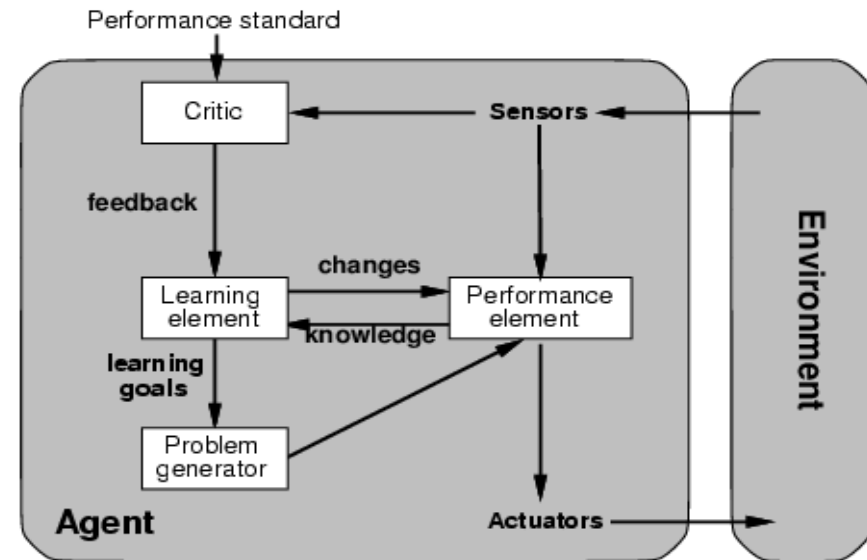
# [Agents pour la résolution de problème]

- Agents basés sur l'utilité
  - Le modèle de l'environnement estime l'état présent du monde ainsi que son état futur selon les différentes actions possibles.
  - Sélection des actions selon une estimation de leur utilité (traduisant à quel point on s'approche d'une situation recherchée)



# [Agents pour la résolution de problème]

- Agents apprenants
  - Les décisions sont prises selon une utilité estimée selon un modèle appris sur des décisions précédentes
  - Des informations de performance sont fournies pour juger de la pertinence des décisions prises et si besoin mettre à jour le modèle



# [Types d'environnement]

- Complètement ou partiellement observable
- Déterministe ou stochastique
- Statique ou dynamique

# [Types de problèmes]

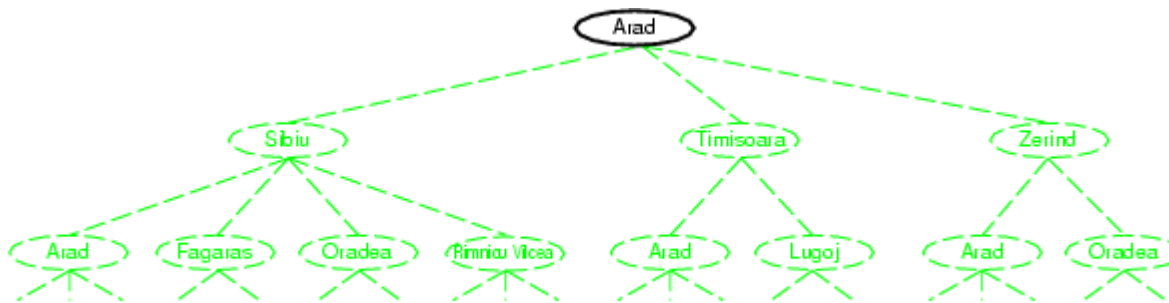
- Déterministe, complètement observable
  - L'Agent sait exactement dans quel état il sera selon une action donnée.  
La solution est une séquence d'actions.
- Non-observable
  - L'Agent peut n'avoir aucune idée d'où il est.
- Non déterministe et/ou partiellement observable
  - Espace d'états inconnu → problème d'exploration

# [ Recherche dans des arbres ]

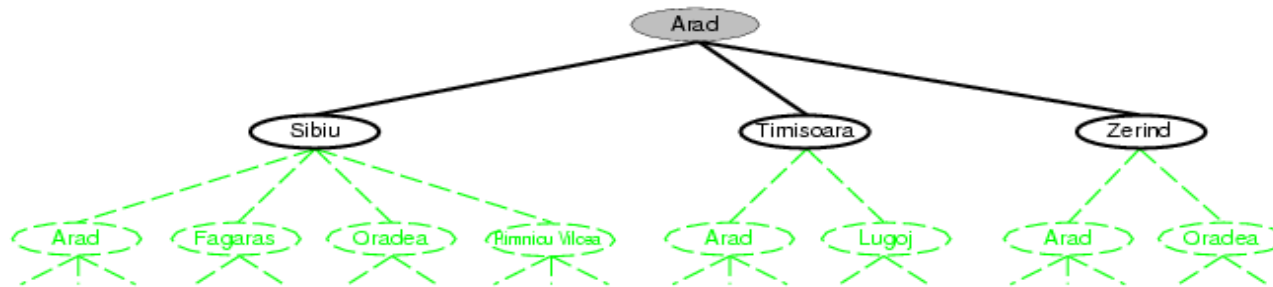
- Idée fondamentale:
  - Offline (hors-ligne), simulation de l'exploration de l'espace d'états par génération des états suivants des états déjà explorés (**expansion** des états)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

# [Tree search example]

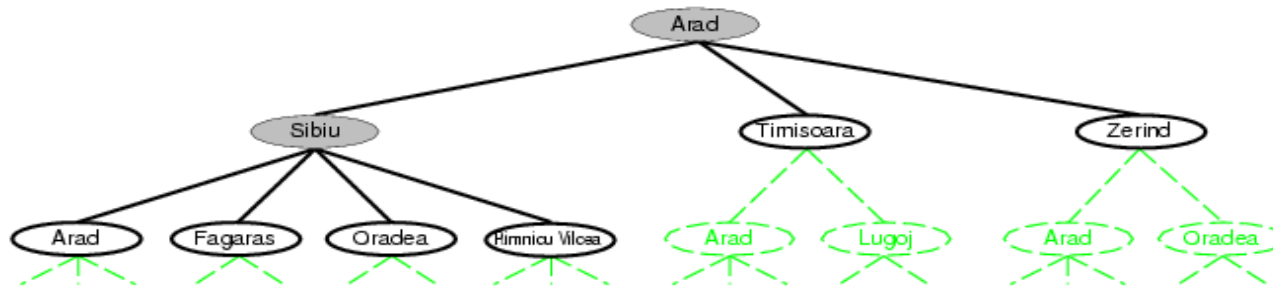


# [Tree search example]



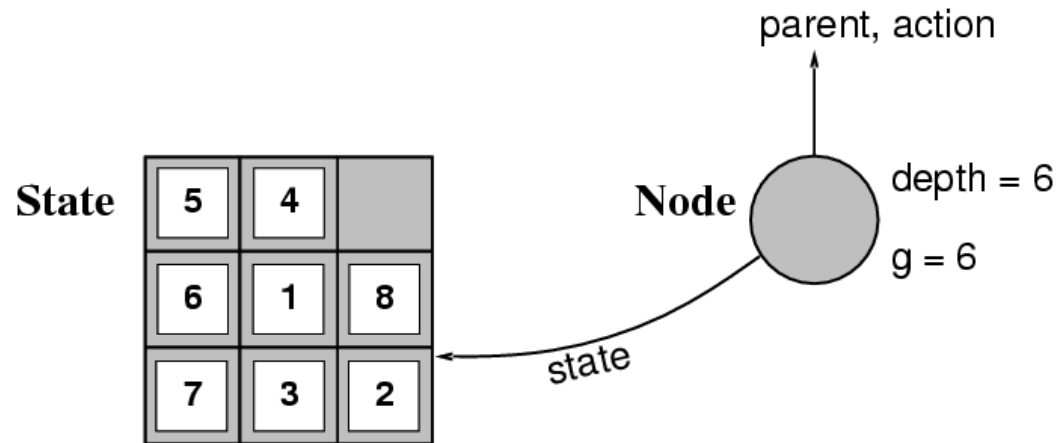


# [Tree search example]



# Implémentation: états vs. noeuds

- Un **état** est une (représentation de) une configuration physique (réelle)
- Un **noeud** est une structure de données constitutive d'un arbre de recherche et inclut **un état**, **un noeud parent**, une **action**, **un cout de chemin  $g(x)$** , une **profondeur**



- A chaque niveau de la recherche, la fonction d'Expansion crée de nouveaux noeuds (noeuds « enfants ») en fonction des états successeurs d'un noeud courant.

# [ Implémentation: recherche générique dans un arbre ]

**Entrées** : un *problème* et une *stratégie*

**Sortie** : une *solution*, ou *échec*

initialiser l'arbre de recherche avec l'état initial du *problème*

**itérer**

**si** il n'y a pas de candidat à développer **alors**

**retourner** *échec*

choisir un nœud à développer en appliquant la *stratégie*

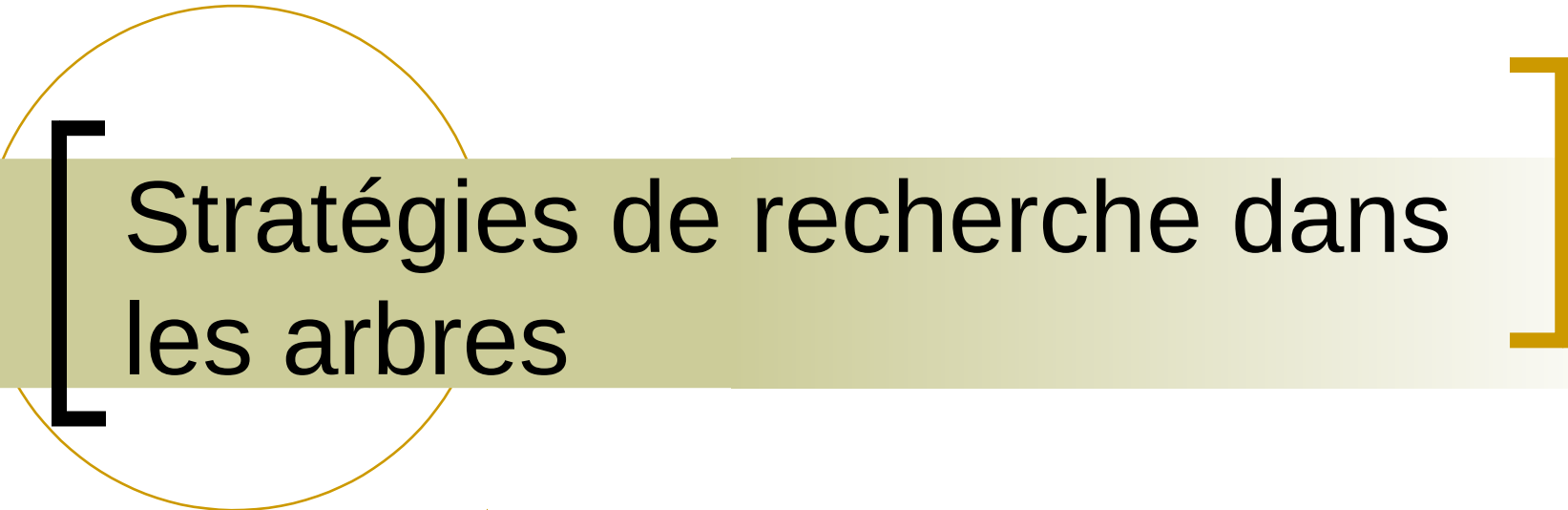
**si** le nœud contient un état final **alors**

**retourner** la *solution* qui correspond

**sinon**

développer le nœud

ajouter les nœuds du résultat dans l'arbre de recherche



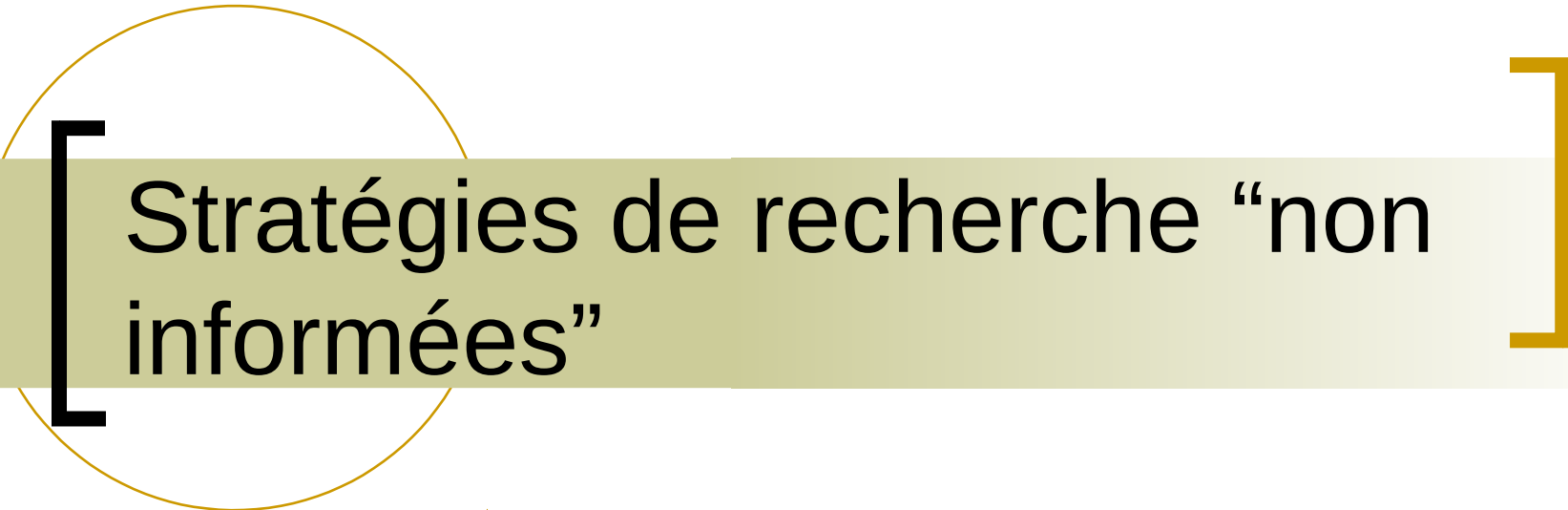
# Stratégies de recherche dans les arbres

# [ Stratégies de recherche ]

- Une stratégie de recherche est définie par le choix de **l'ordre d'expansion des noeuds**
- Deux grands types de stratégies :
  - Les stratégies « non-informées »
    - Pas d'utilisation d'une distance avec les états recherchés pour choisir l'ordre des noeuds
    - Uniquement prise en compte du but : l'état remplit-il un but recherché ?
  - Les stratégies « informées »
    - Nœuds les plus prometteurs sont explorés en premier
    - Nécessite de déterminer une fonction de distance avec l'état recherché / d'utilité en fonction du but à atteindre

# [ Stratégies de recherche ]

- Les stratégies sont évaluées suivant:
  - **Complétude**: Trouve-t-on toujours une solution s'il en existe une?
  - **Complexité en temps**: Nombre de nœuds générés
  - **Complexité en mémoire**: Nombre de nœuds en mémoire
  - **Optimalité**: Trouve-t-on toujours la solution de coût minimal?
- Les complexités en temps et mémoire sont mesurées par :
  - $b$ : facteur de branchement maximum dans l'arbre de recherche
  - $d$ : Profondeur de la solution optimale (racine est en 0)
  - $m$ : Profondeur maximale de l'espace d'états (éventuellement  $\infty$ )



# Stratégies de recherche “non informées”

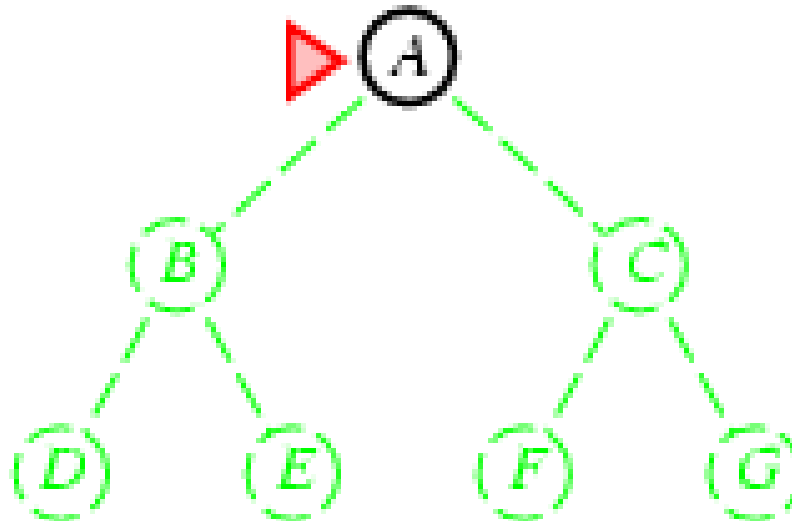
# [Stratégies de recherche “non informées”]

- Largeur d’abord
- Cout uniforme
- Profondeur d’abord
- Profondeur limitée
- Profondeur limitée itératif



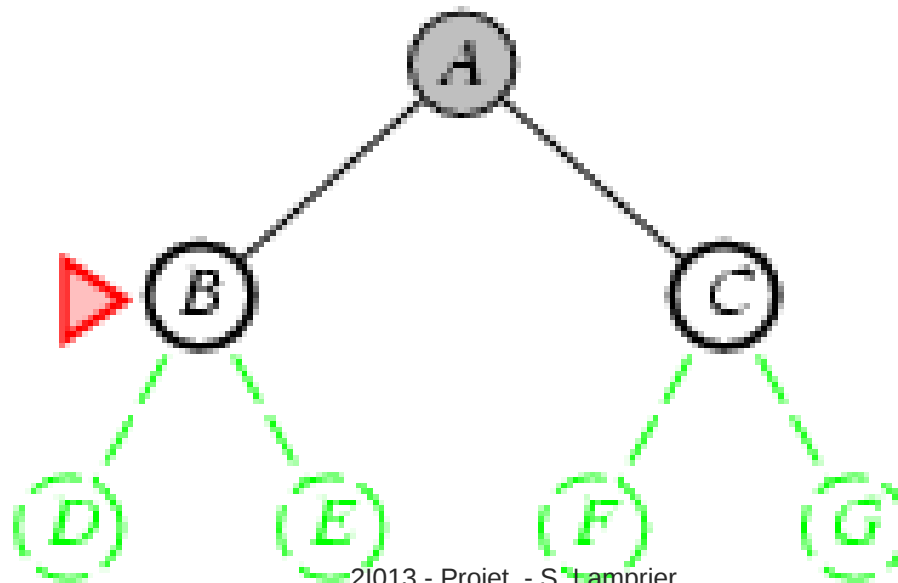
# [ Largeur d'abord ]

- Expansion du noeud le moins profond
- **Implémentation:**
  - *fringe* est une liste FIFO, i.e., les nouveaux suivants sont ajoutés en queue



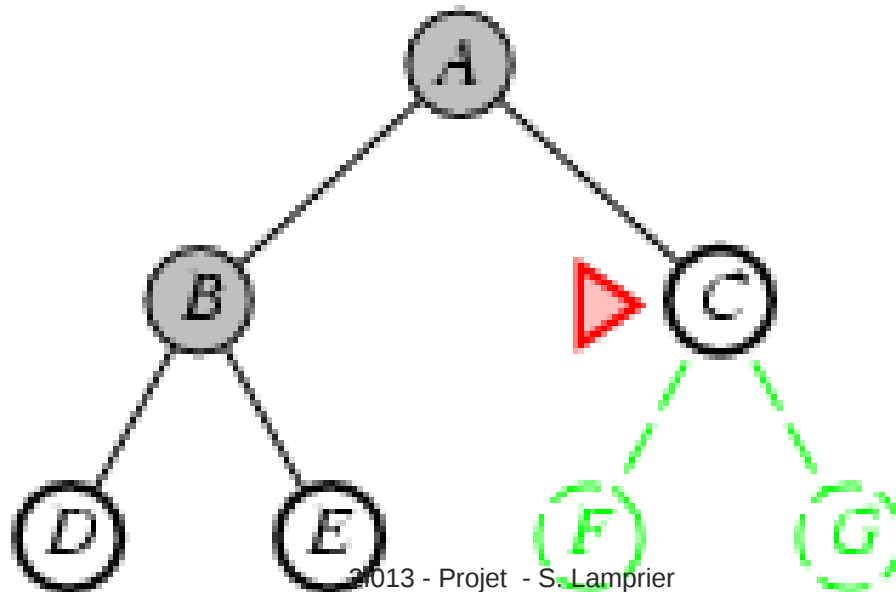
# [ Largeur d'abord ]

- Expansion du noeud le moins profond
- **Implémentation:**
  - *fringe* est une liste FIFO, i.e., les nouveaux suivants sont ajoutés en queue



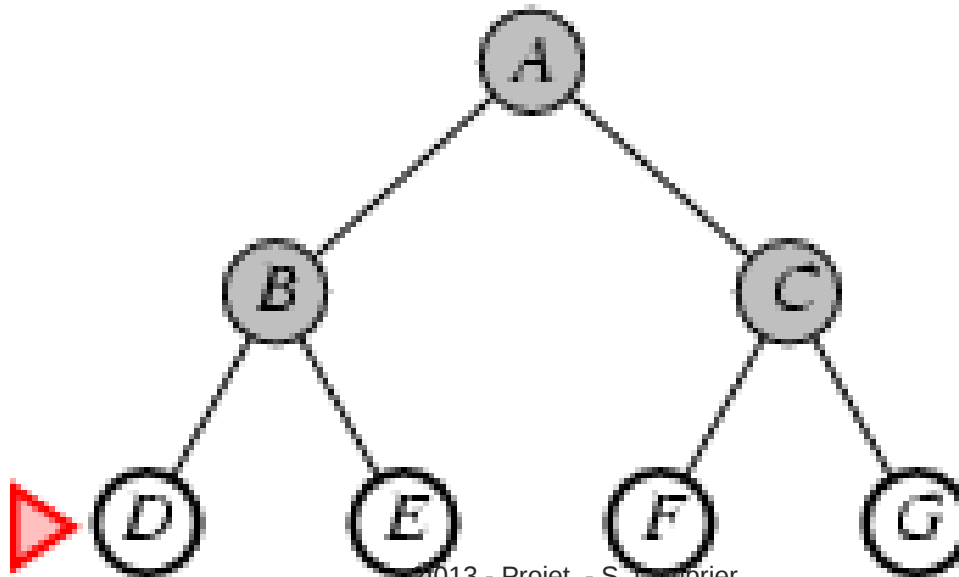
# [ Largeur d'abord ]

- Expansion du noeud le moins profond
- **Implémentation:**
  - *fringe* est une liste FIFO, i.e., les nouveaux suivants sont ajoutés en queue



# [ Largeur d'abord ]

- Expansion du noeud le moins profond
- **Implémentation:**
  - *fringe* est une liste FIFO, i.e., les nouveaux suivants sont ajoutés en queue



# [ Largeur d'abord ]

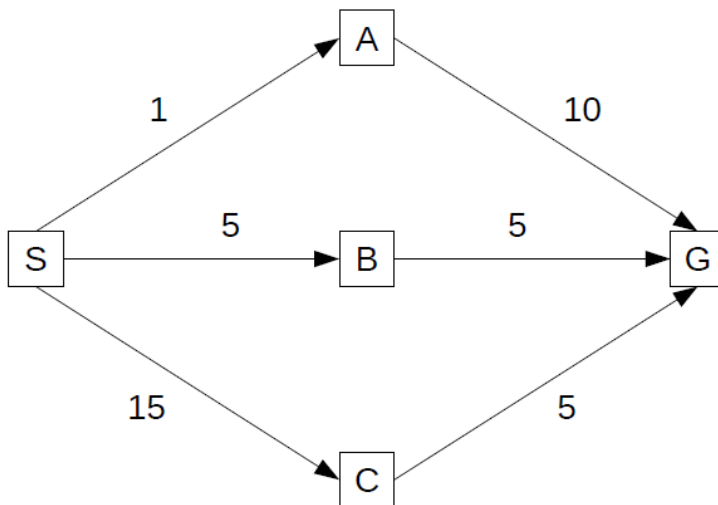
- Complet?
- Temps?
- Mémoire?
- Optimal?

# [ Largeur d'abord ]

- Complet? Oui (si  $b$  est fini)
- Temps?  $1+b+b^2+b^3+\dots +b^d \approx O(b^d)$
- Mémoire?  $O(b^d)$  (on garde tous les noeuds en mémoire)
- Optimal? Oui (si cout = 1 par étape)
- L'espace est le problème principal de cet algorithme

# [ Coût uniforme ]

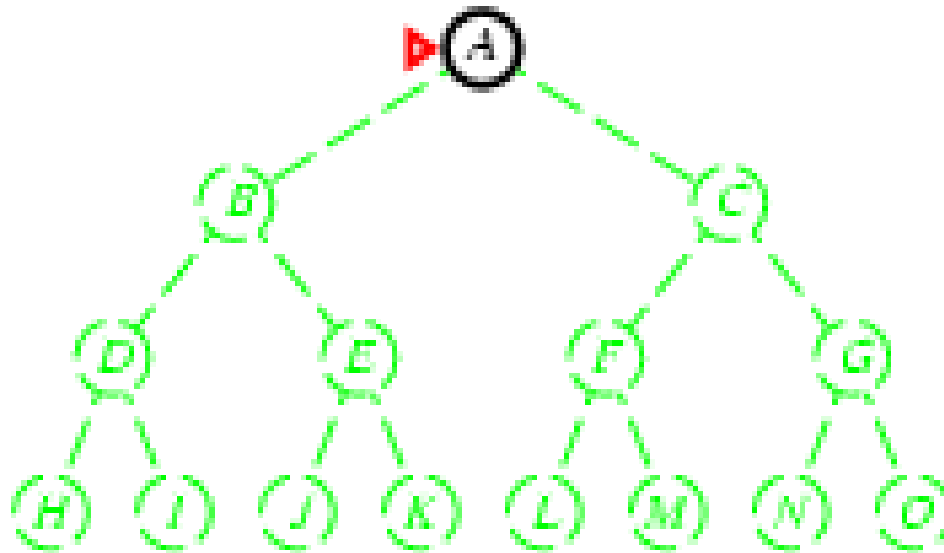
- Développement en priorité du nœud qui a le coût le plus faible
- File triée par coût croissant
- Equivalent à exploration en largeur d'abord si toutes les actions ont le même coût



Contenu de la file	Étape	# nœuds développés
[(S,0)]	Développe (S,0)	
[(A,1);(B,5);(C,15)]	Développe (A,1)	1
[(B,5);(G,11);(C,15)]	Développe (B,5)	2
[(G,10);(G,11);(C,15)]	Développe (G,10)	3
∅		4

# [ Profondeur d'abord ]

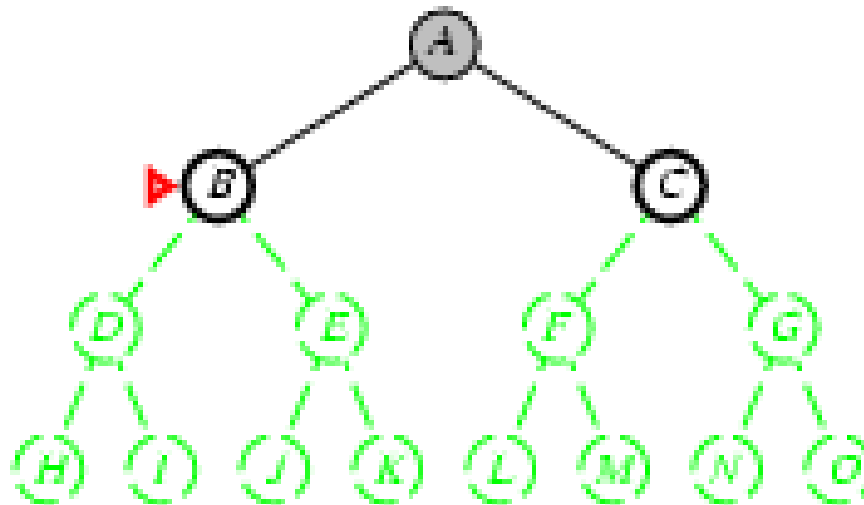
- Expansion du noeud le plus profond
- **Implémentation:**
  - *fringe* = liste LIFO, i.e., suivants insérés en tête





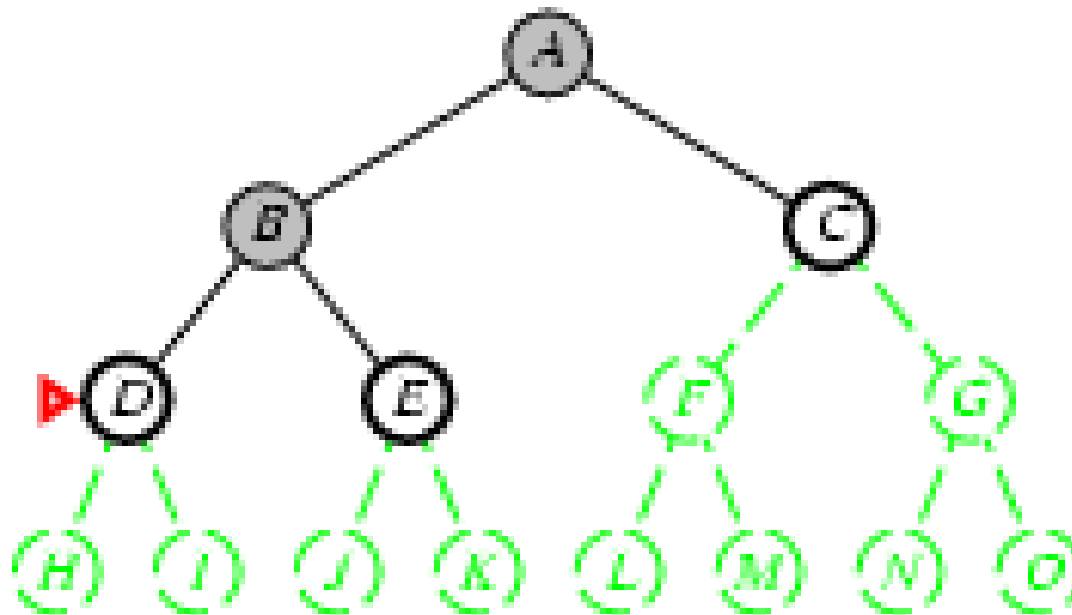
# [ Profondeur d'abord ]

- Expansion du noeud le plus profond
- **Implémentation:**
  - *fringe* = liste LIFO, i.e., suivants insérés en tête



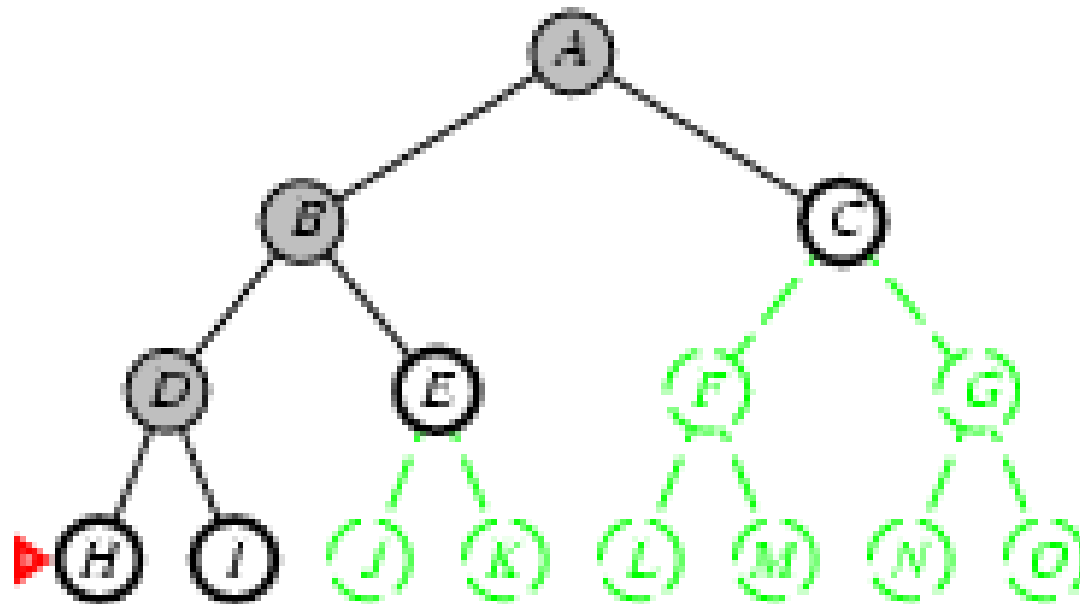
[

]



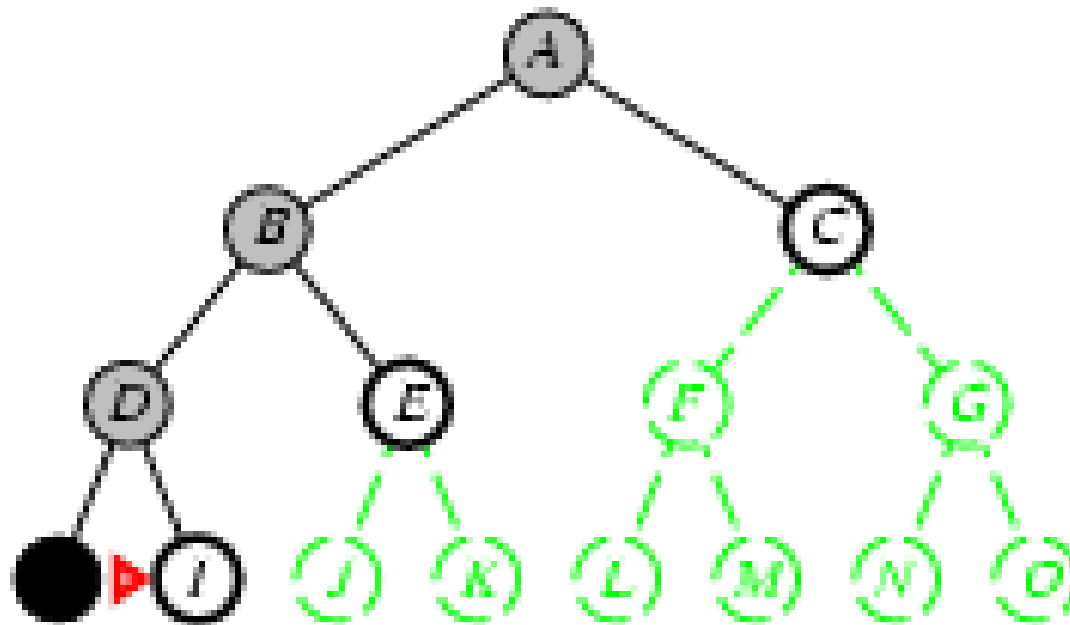
[

]



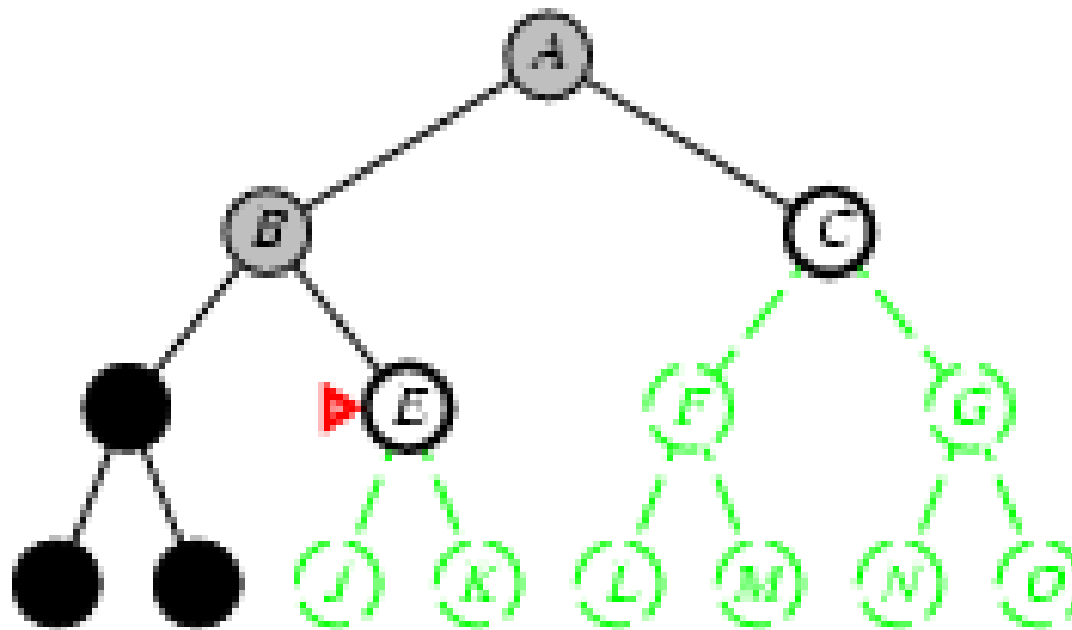
[

]



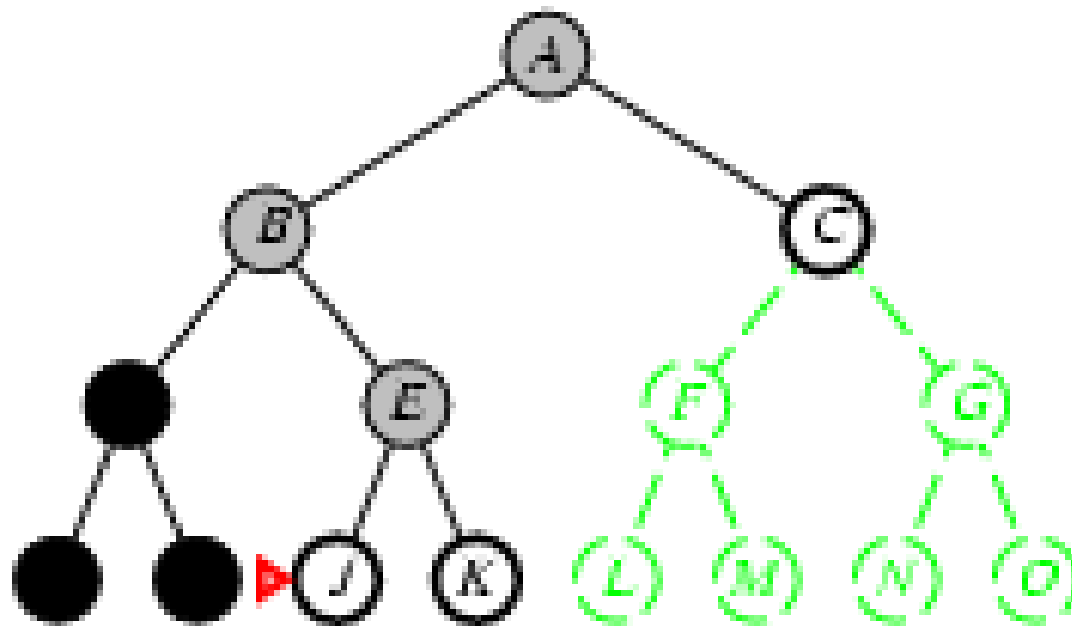
[

]



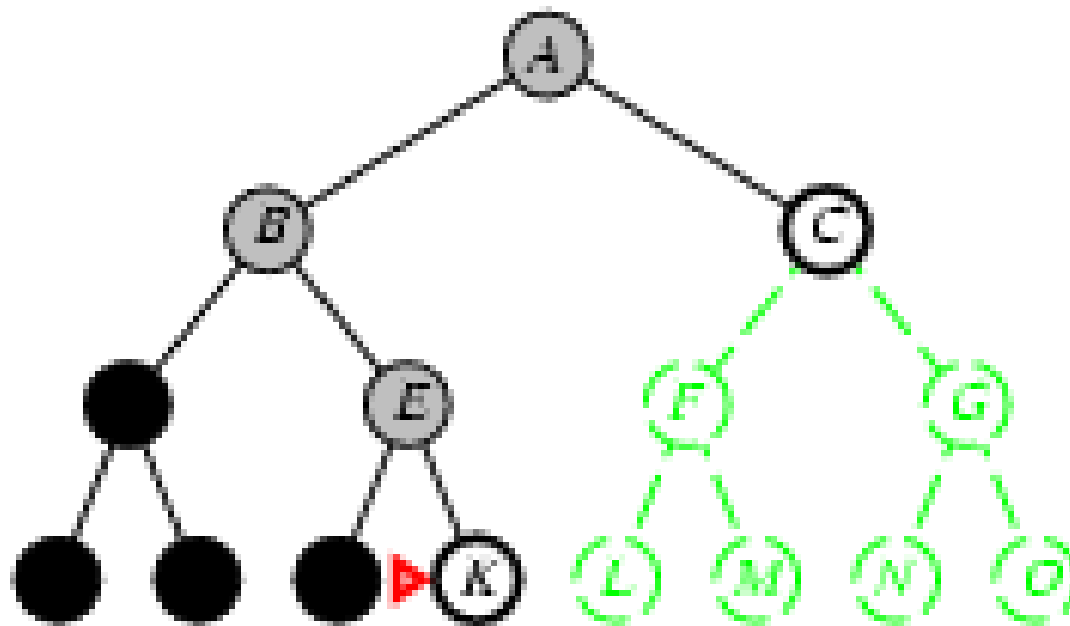
[

]



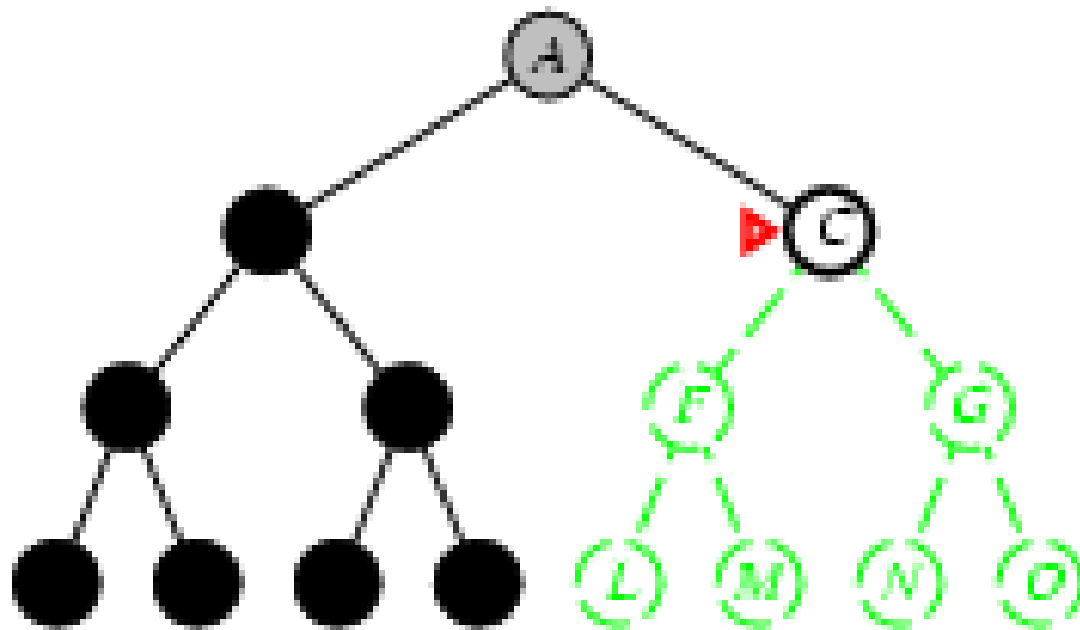
[

]



[

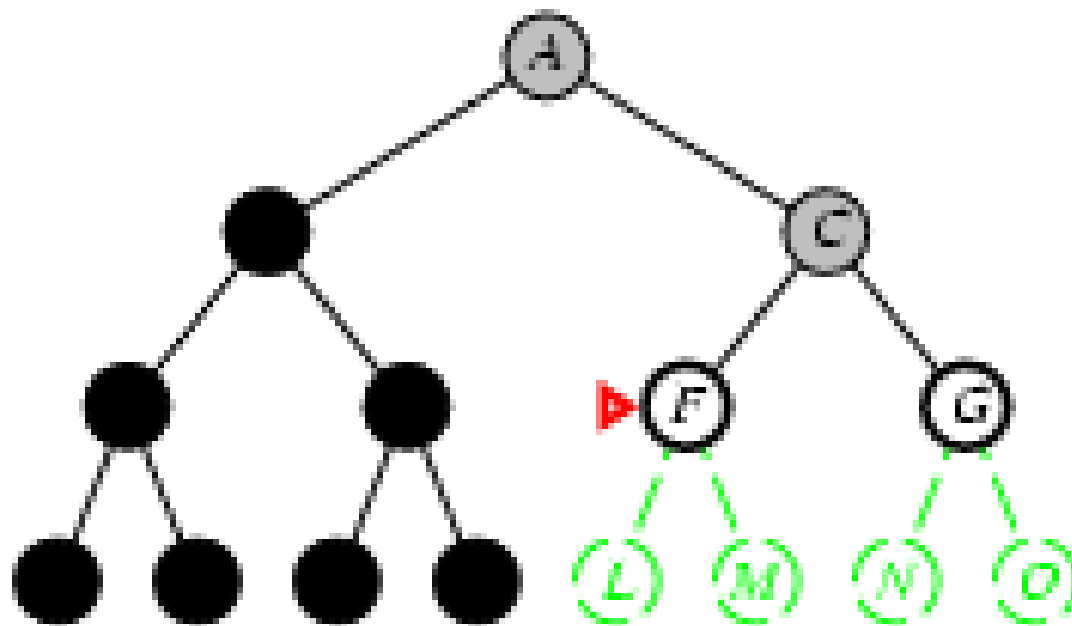
]





[

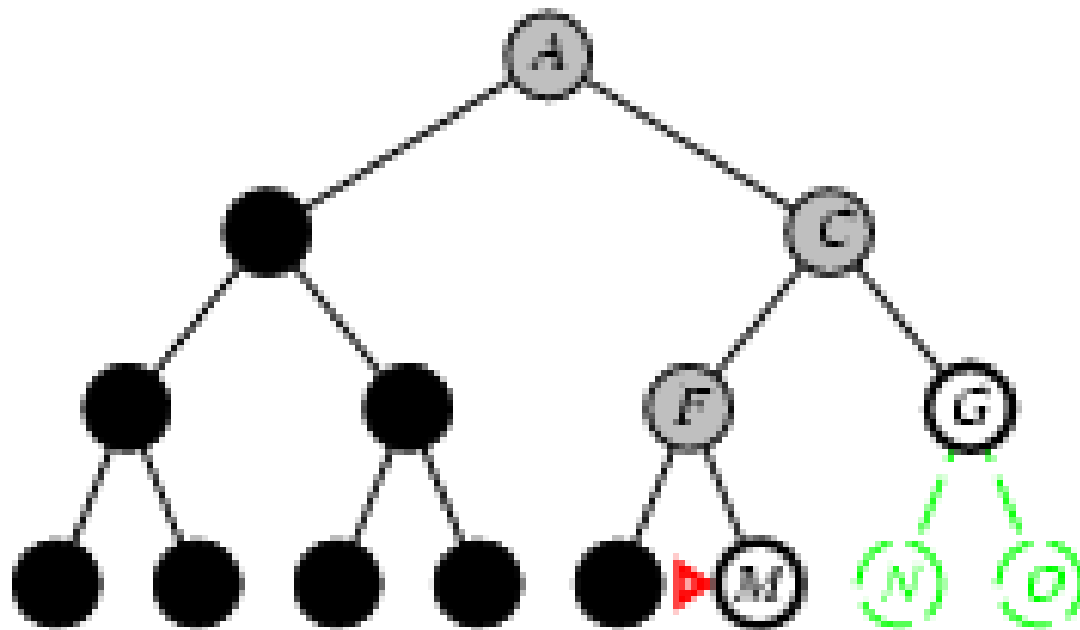
]





[

]



# [ Profondeur d'abord ]

- Complet?
- Temps?
- Mémoire?
- Optimal?

# [ Profondeur d'abord ]

- Complet? Non (sauf si  $m$  est fini)
- Temps?  $O(b^m)$
- Mémoire?  $O(bm)$  (uniquement les noeuds correspondant aux expansions des noeuds parents de la solution)
- Optimal? Non

# [ Profondeur limitée ]

= Profondeur d'abord mais en limitant la profondeur à  $l$ , i.e., les noeuds à profondeur  $l$  n'ont pas de suivants

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

# [ Profondeur limitée ]

- Complet?
- Temps?
- Mémoire?
- Optimal?

# [ Profondeur limitée ]

- Complet? Non
- Temps?  $O(b^l)$
- Mémoire?  $O(b^l)$
- Optimal? Non



# Profondeur limitée itérative

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH( problem, depth)  
    if result  $\neq$  cutoff then return result
```

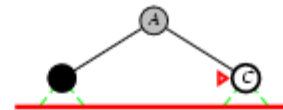
# [ Profondeur limitée itérative / =0 ]

Limit = 0



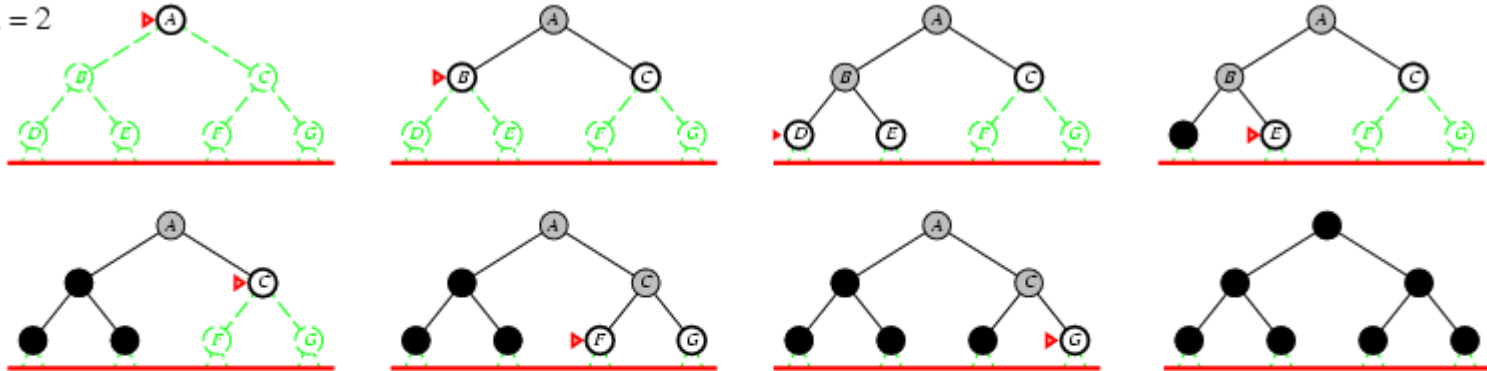
[ / = 1 ]

Limit = 1



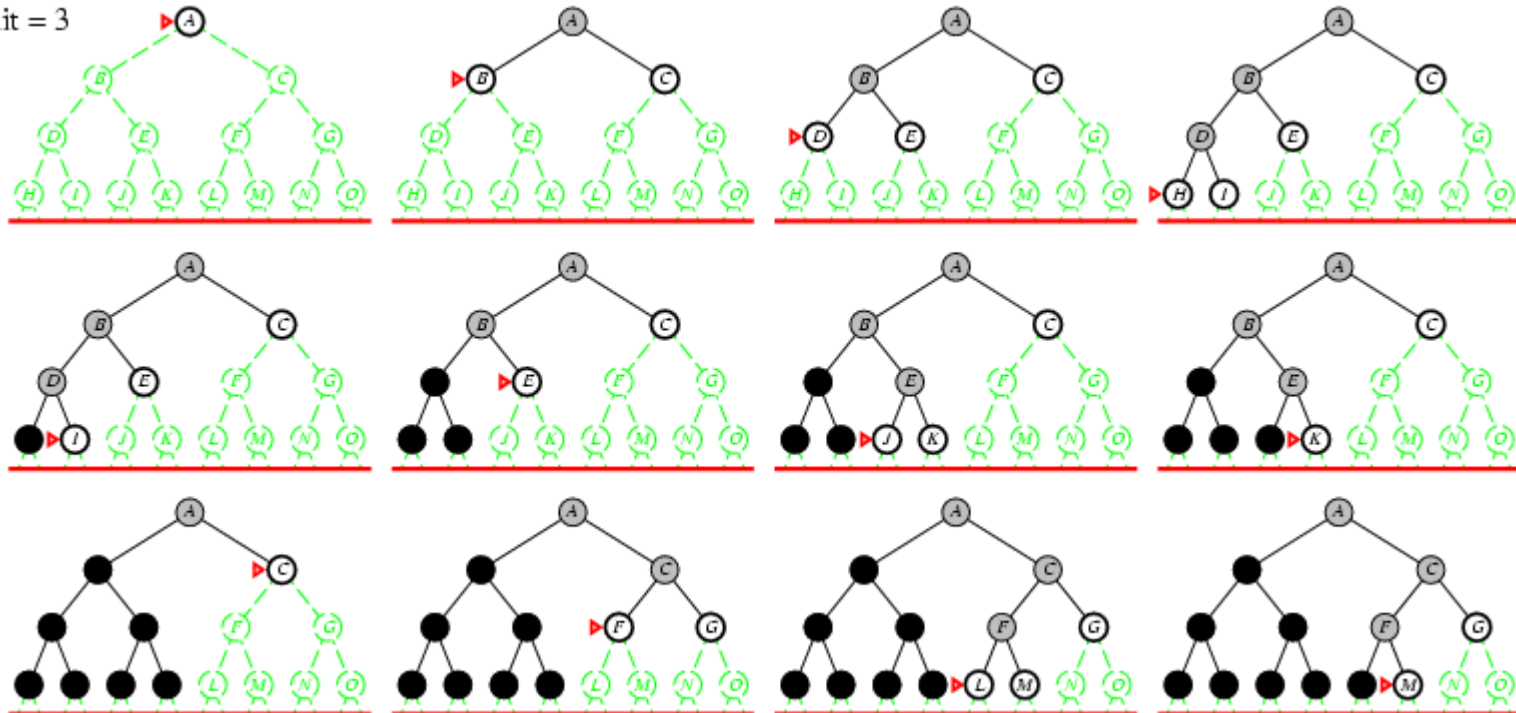
# Profondeur limitée itérative / =2

Limit = 2



# Profondeur limitée itérative / =3

Limit = 3



# [ Profondeur limitée itérative ]

- Nombre de nœuds générés par une recherche en profondeur fixée à  $d$  avec un facteur de branchement  $b$ :

$$N_{PL} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

- Nombre de nœuds générés par une recherche en profondeur limitée itérative à  $d$  avec un facteur de branchement  $b$ :

$$N_{PLI} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For  $b = 10, d = 5$ ,
  - $N_{PL} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{PLI} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- Surplus =  $(123,456 - 111,111)/111,111 = 11\%$

# [ Profondeur limitée itérative ]

- Complete?
- Temps?
- Mémoire?
- Optimal?

# [ Profondeur limitée itérative ]

- Complete? Oui
- Temps?  $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Mémoire?  $O(bd)$
- Optimal? Oui si coût d'étape = 1



# [ Résumé des algorithmes ]

<i>Critère</i>	<i>Largeur d'abord</i>	<i>Coût uniforme</i>	<i>Profondeur d'abord</i>	<i>Profondeur limitée</i>	<i>Itérative en profondeur</i>	<i>Bidirectionnelle (si applicable)</i>
<i>Complète</i>	oui <sup>1</sup>	oui <sup>1,2</sup>	non	non	oui <sup>1</sup>	oui <sup>1,4</sup>
<i>Temps</i>	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
<i>Espace</i>	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
<i>Optimale</i>	oui <sup>3</sup>	oui	non	non	oui <sup>3</sup>	oui <sup>3,4</sup>

b : facteur de branchement

d : profondeur de la solution la moins profonde

m : profondeur maximale de l'arbre de recherche

l : profondeur limite

<sup>1</sup> si b est fini

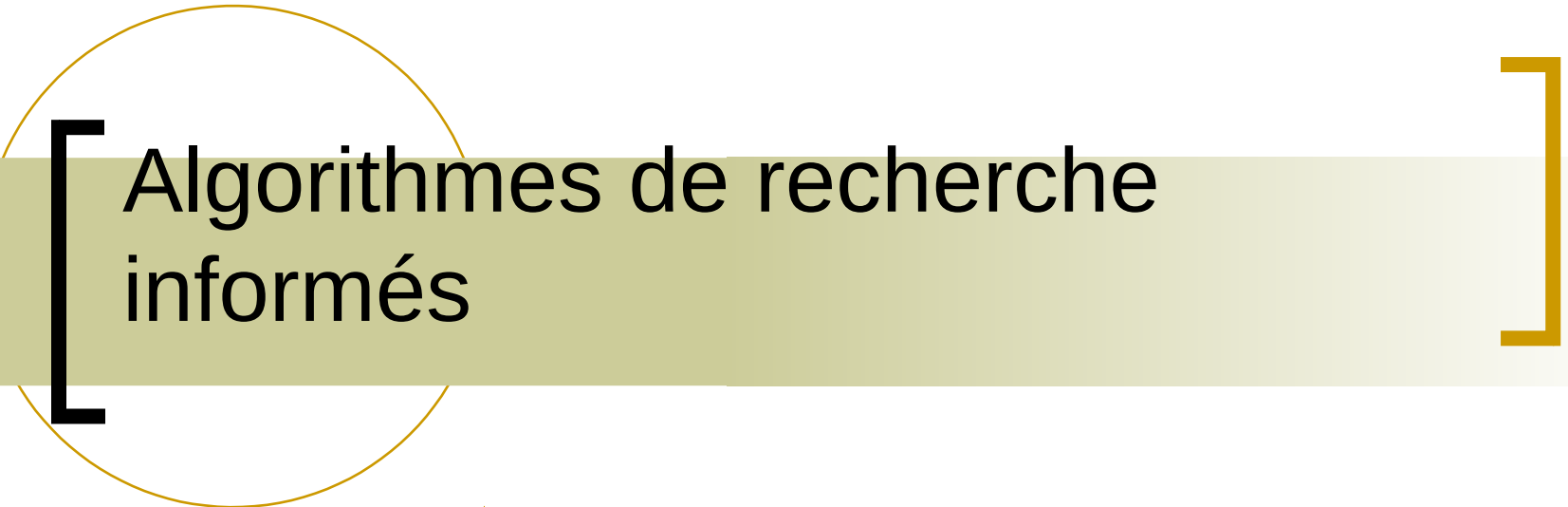
<sup>2</sup> si les coûts des étapes sont  $\geq \epsilon$  avec  $\epsilon$  positif

<sup>3</sup> si les coûts d'étapes sont tous identiques

<sup>4</sup> si les deux directions utilisent une exploration en largeur d'abord

# [ Résumé ]

- La formulation du problème requiert l'abstraction du monde réel (on oublie les détails) pour définir un espace d'état qui peut être exploré efficacement
- Grand nombre de stratégies « non informées »
- Profondeur Limitée Itérative utilise une mémoire linéaire et pas bcp plus de temps que les autres algos.



# Algorithmes de recherche informés

# [ Algorithmes de recherche informés ]

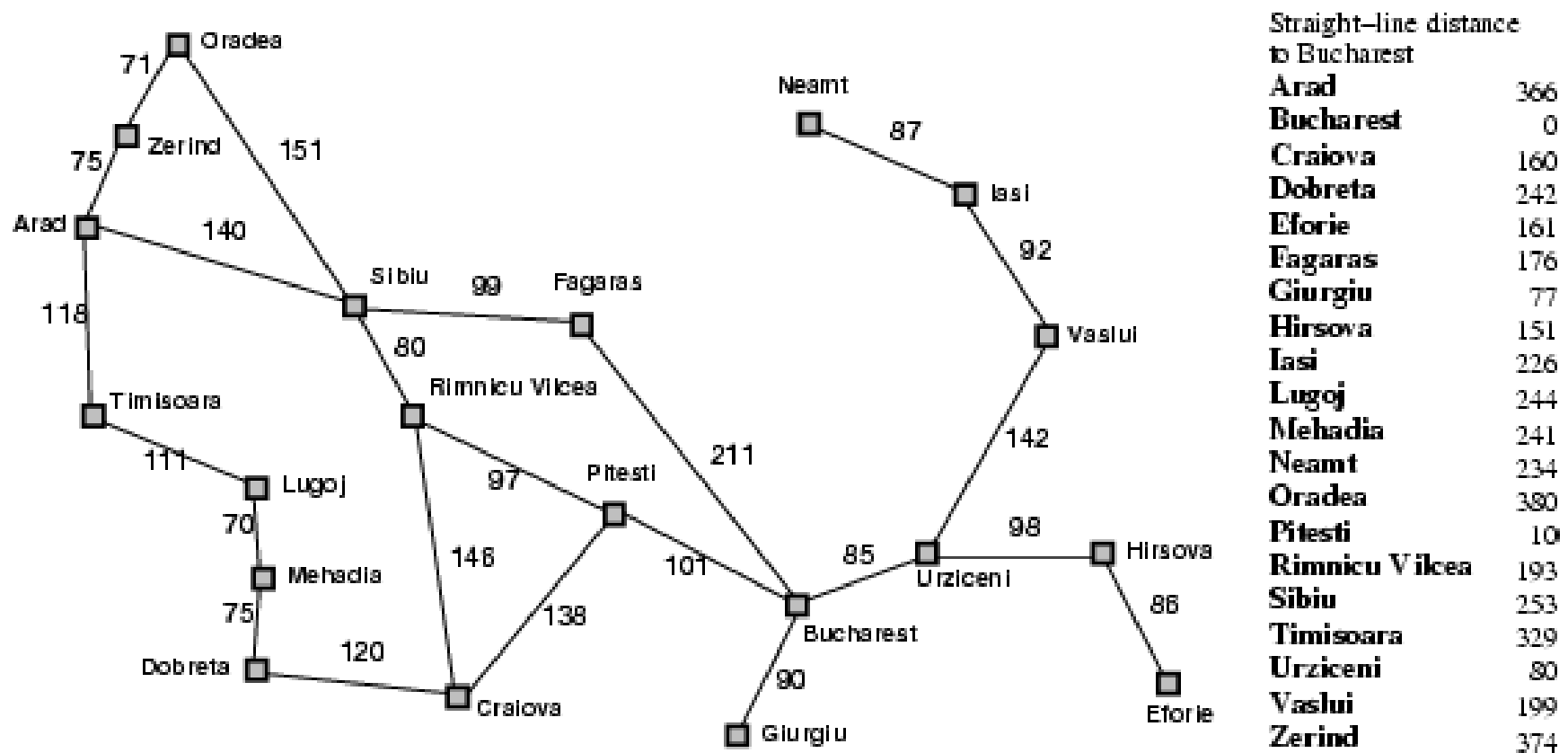
- Meilleur d'abord
- Meilleur d'abord vorace
- $A^*$
- Heuristiques
- ...

# [Meilleur d'abord]

- Idée : utiliser une **fonction d'évaluation**  $f(n)$  pour chaque noeud
  - Estimée de l'utilité
  - Expansion des noeuds les plus utiles
- Implémentation:
  - Ordonner les noeuds dans la liste de noeuds courante par ordre d'utilité décroissante
- Cas particuliers:
  - Meilleur d'abord vorace
  - $A^*$

# Cas du parcours en Roumanie

## Couts en km



# [Meilleur d'abord]

- Fonction d'évaluation  $f(n) = h(n)$  (**h**euristique)
- = estimée du cout du noeud  $n$  au but
- e.g.,  $h_{SLD}(n)$  = distance en ligne droite de  $n$  à Bucarest
- Meilleur d'abord vorace expand le noeud qui **semble** être le plus près du but

# [Meilleur d'abord exemple]

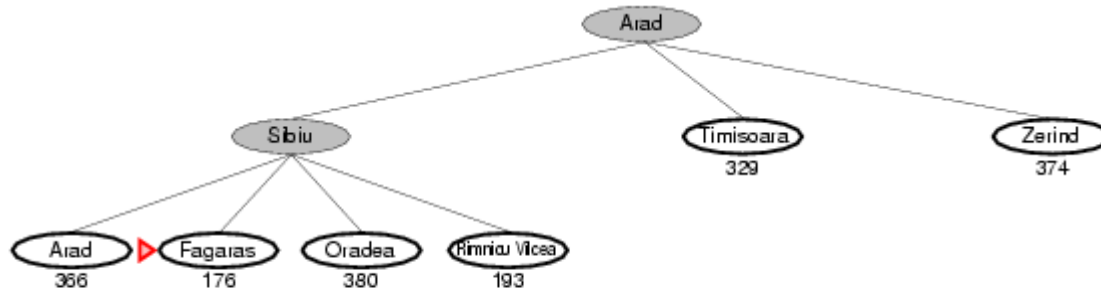




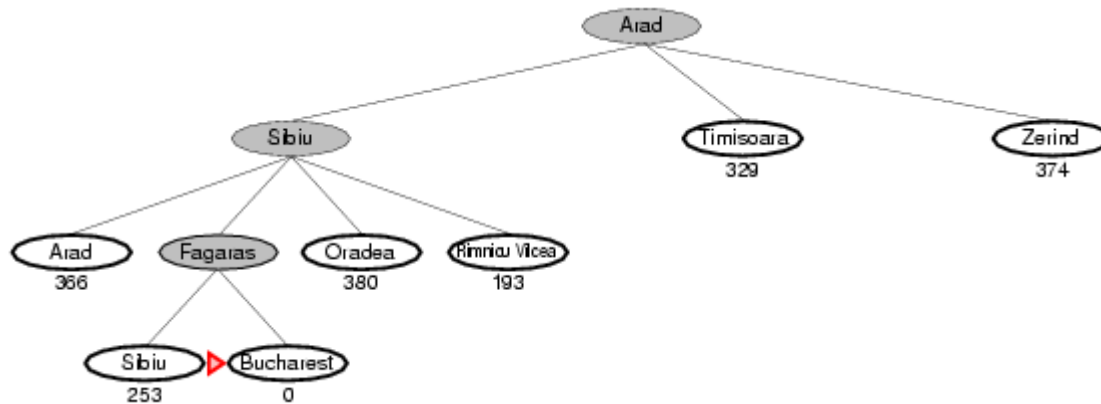
# [Meilleur d'abord exemple]



# [ Meilleur d'abord exemple ]



# [ Meilleur d'abord exemple ]



# [Meilleur d'abord]

- Complet ?
- Temps ?
- Mémoire ?
- Optimal ?

# [Meilleur d'abord]

- Complet ? Non – Cf. boucles
  - e.g., lasi  $\rightarrow$  Neamt  $\rightarrow$  lasi  $\rightarrow$  Neamt  $\rightarrow$
- Temps ?  $O(b^m)$ , mais un bon heuristique peut baisser radicalement cette complexité
- Mémoire ?  $O(b^m)$  – tous les noeuds en mémoire
- Optimal ? Non

# [Meilleur d'abord vorace]

- Variante du meilleur d'abord où l'on ne revient jamais en arrière
- Complet ?
- Temps ?
- Mémoire ?
- Optimal ?

# [ Meilleur d'abord vorace ]

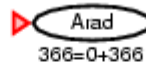
- Variante du meilleur d'abord où l'on ne revient jamais en arrière
- Complet ? Non – Cf. boucles
  - e.g., lasi → Neamt → lasi → Neamt →
- Temps ?  $O(b^m)$
- 
- Mémoire ?  $O(b)$
- 
- Optimal ? Non

# [ Algorithme $A^*$ ]

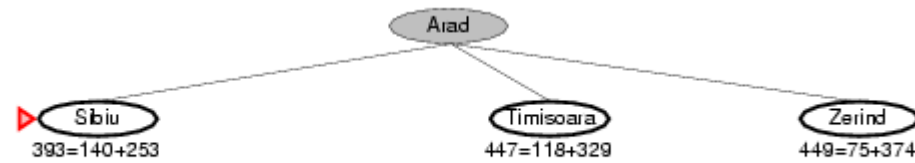
- Idée : éviter d'expandre des chemins dont on sait qu'ils sont déjà trop coûteux
- Fonction d'évaluation  $f(n) = g(n) + h(n)$
- $g(n)$  = cout pour atteindre  $n$
- $h(n)$  = estimée du cout de  $n$  au but
- $f(n)$  = estimée du cout total du chemin atteignant le but via  $n$



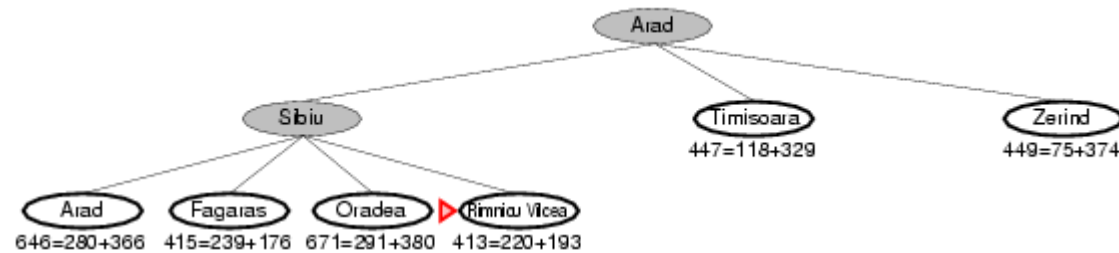
# [ $A^*$ : exemple ]



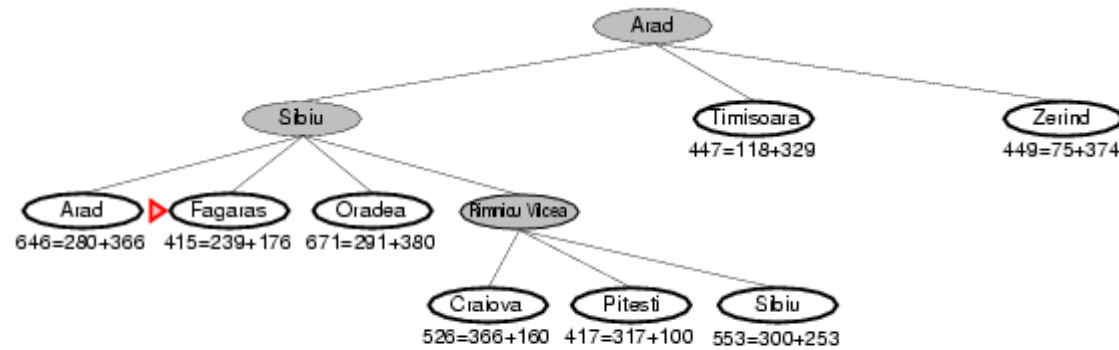
# [ $A^*$ : exemple ]



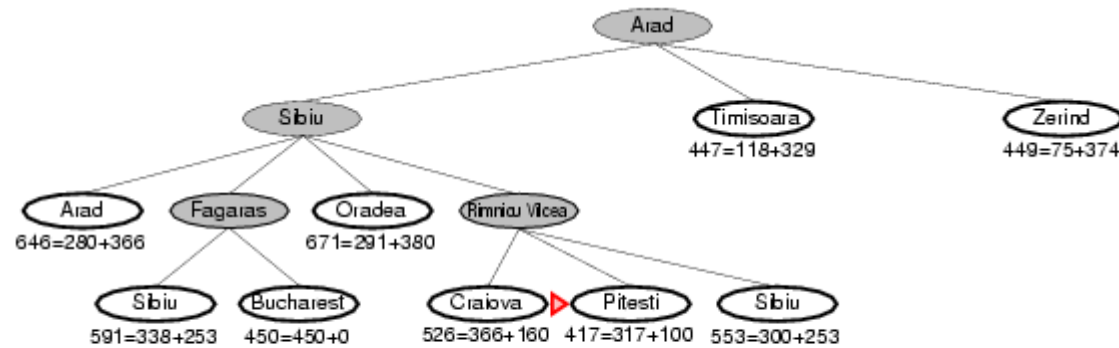
# [ $A^*$ : exemple ]



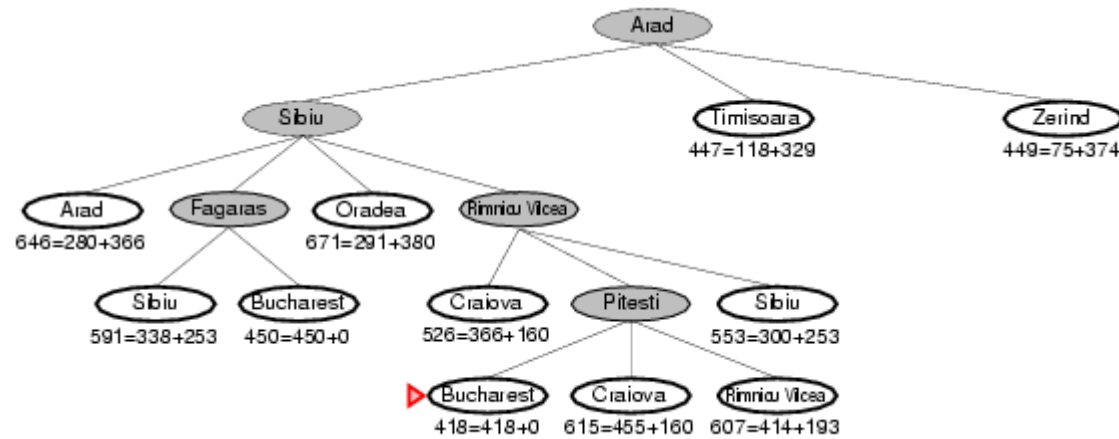
# [ $A^*$ : exemple ]



# [ $A^*$ : exemple ]



# [ $A^*$ : exemple ]



# [ Heuristiques admissibles ]

- Un heuristique  $h(n)$  est **admissible** si pour tout nœud  $n$ ,  
 $h(n) \leq h^*(n)$ , où  $h^*(n)$  est le **vrai** cout pour atteindre l'état final (le but) à partir de  $n$ .
- Un heuristique admissible **ne surestime jamais** le cout pour atteindre le but, i.e., il est **optimiste**
- Exemple:  $h_{SLD}(n)$  (ne surestime jamais la vraie distance par la route)
- **Théorème**: Si  $h(n)$  est admissible, l'algorithme  $A^*$  utilisant une stratégie de TREE-SEARCH est optimal

# [ Propriétés de A\* ]

- Complet? Oui (sauf si infinité de noeuds tels que  $f \leq f(G)$  )
- Temps?  $O(b^d)$ , *mais tout depend de la qualité de l'heuristique*
- Mémoire?  $O(b^d)$ , *tous les noeuds en memoire*
- Optimal? Oui



# [Heuristiques admissibles]

Exemple du 8-puzzle:

- $h_1(n)$  = nombre de carreaux mal placés
- $h_2(n)$  = distance total Manhattan  
(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- $h_1(S) = ?$
- $h_2(S) = ?$

# [Heuristiques admissibles]

Exemple du 8-puzzle:

- $h_1(n)$  = nombre de carreaux mal placés
- $h_2(n)$  = distance total Manhattan  
(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- $h_1(S) = ?$  8
- $h_2(S) = ?$   $3+1+2+2+2+3+3+2 = 18$

# [ Dominance ]

- Si  $h_2(n) \geq h_1(n)$  pour tout  $n$  (et tous les deux admissibles)
- Alors  $h_2$  domine  $h_1$
- $\rightarrow h_2$  est meilleur pour la recherche

# [ Problèmes relachés ]

- Un problème moins contraint (sur les actions) est appelé un **relaxed problem**
- Le cout d'une solution optimale pour un relaxed problem est un heuristique admissible pour le problème original
- Si les règles du 8-puzzle sont relachées et qu'un carreau peut bouger n'importe où, alors  $h_1(n)$  donne la solution la plus courte
- Si les règles sont relachées et qu'un carreau peut se déplacer vers n'importe quelle place **adjacente**, alors  $h_2(n)$  donne la solution la plus courte