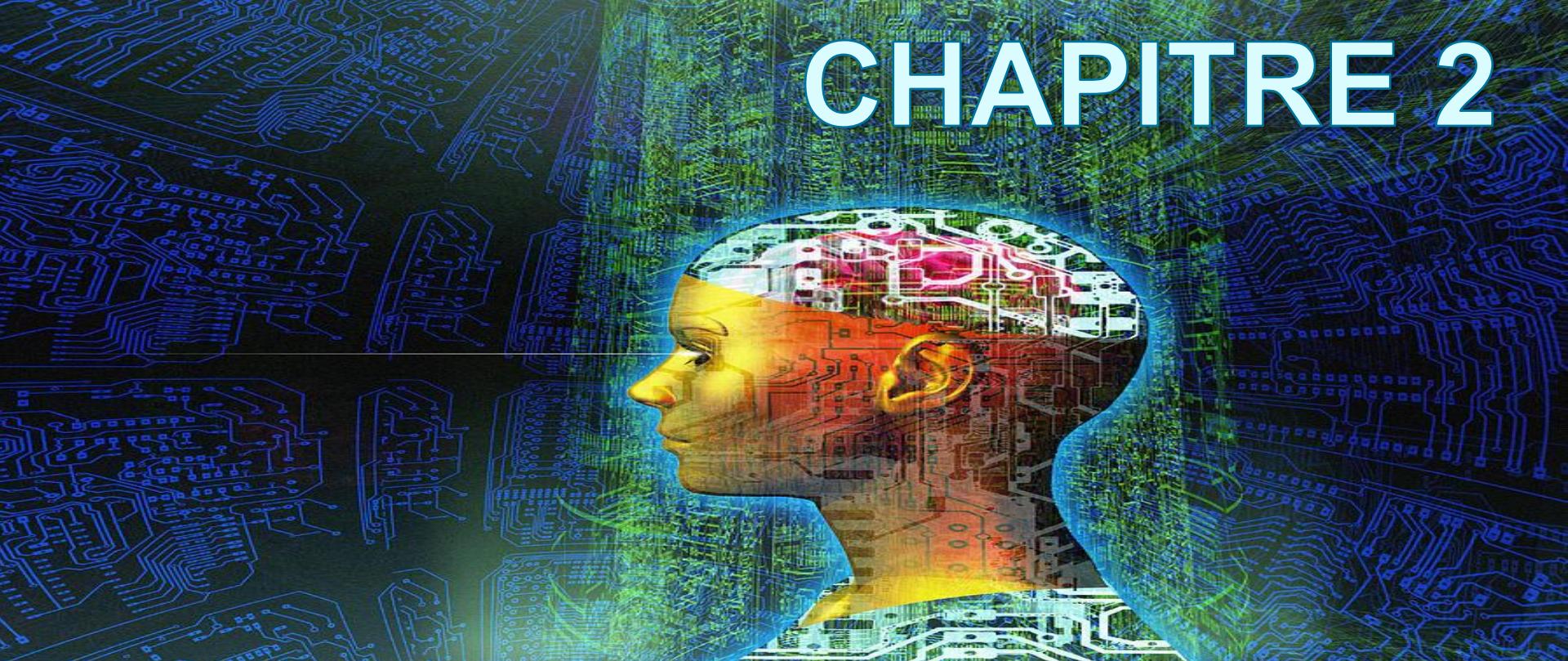


CHAPITRE 2



RÉSOLUTION DE PROBLÈMES PAR EXPLORATION



Plan

- Agents de résolution de problèmes
- Problèmes bien définis
- Exemples de problèmes
- Recherche de solutions
- Stratégies d'explorations systématiques :
 - Non informées,
 - Informées (heuristiques)
- Fonctions heuristiques
- Stratégies d'explorations locales

Plan

Agents de résolution de problèmes

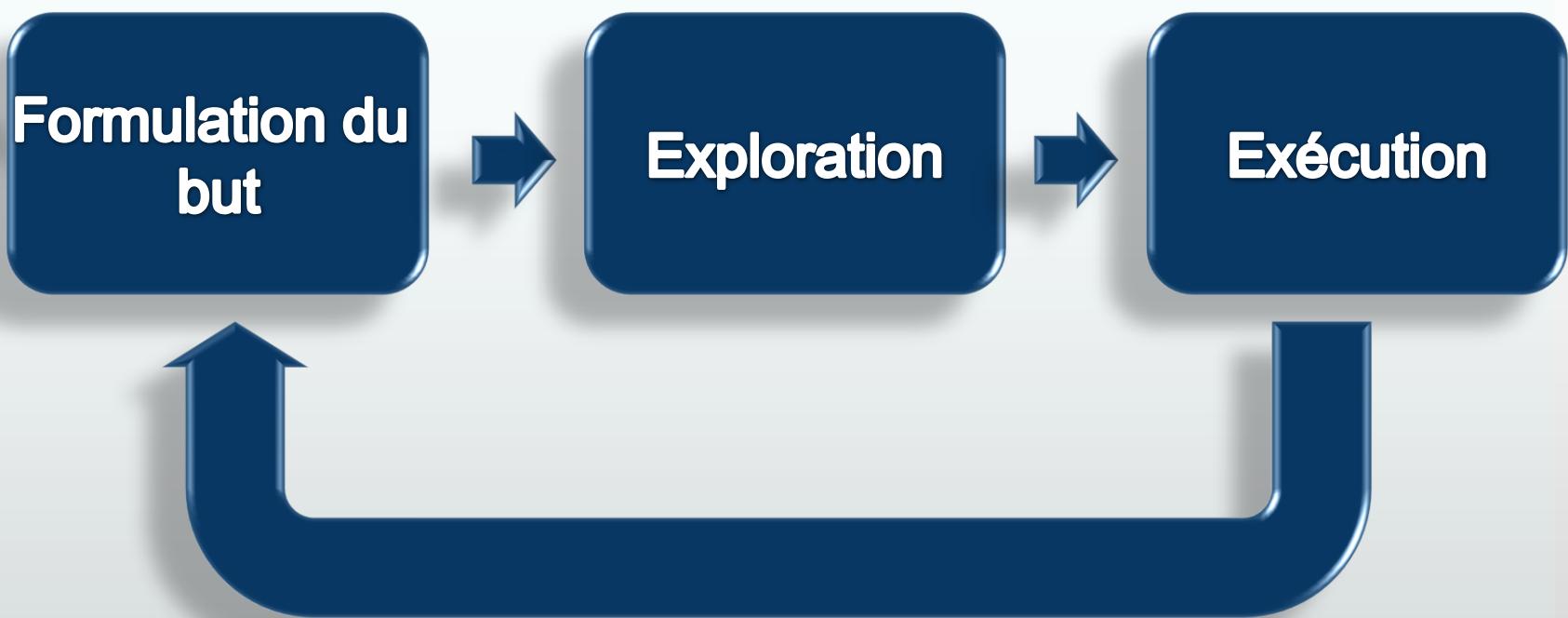
- Problèmes bien définis
- Exemples de problèmes
- Recherche de solutions
- Stratégies d'explorations :
 - Non informées,
 - Informées (heuristiques)
- Fonctions heuristiques

Agents de résolution de problèmes

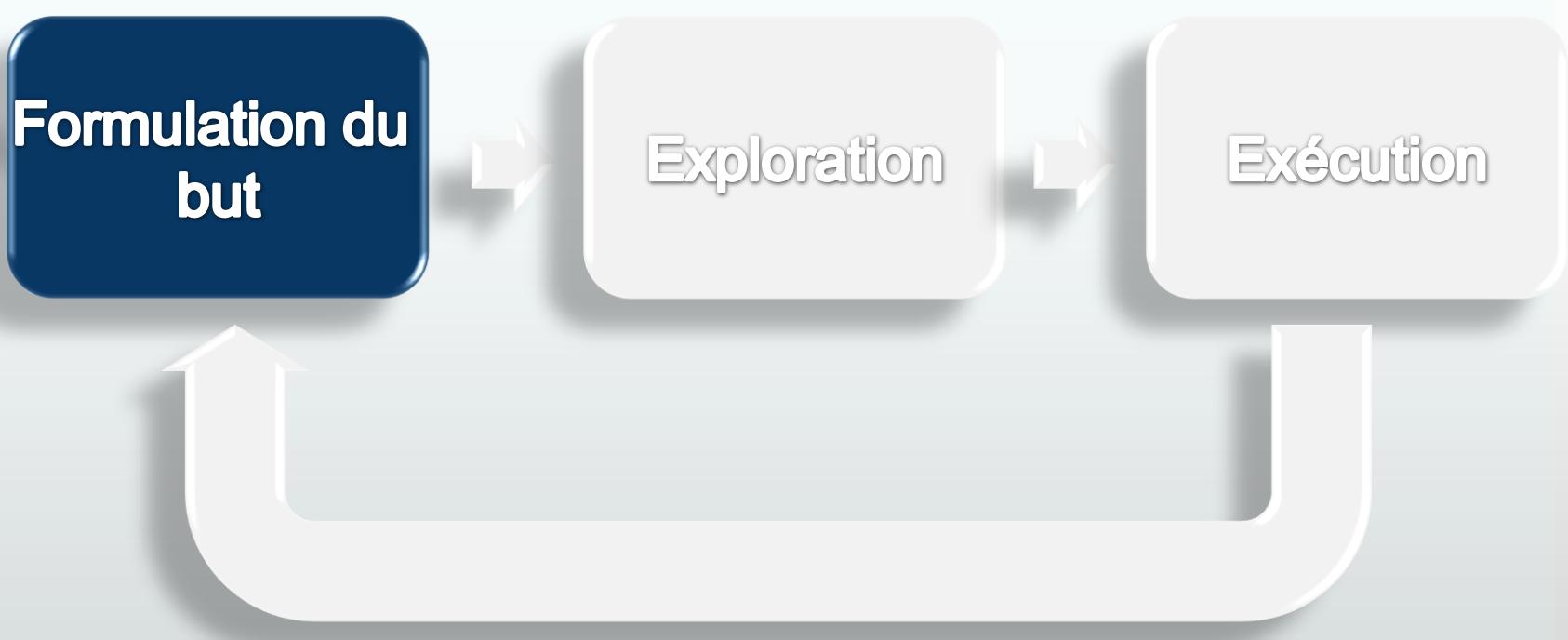
Agents de résolution de problèmes

- ❑ Se sont des agents fondés sur les buts.
- ❑ Les états du monde sont des atomes sans structure interne visible.
- ❑ Un agent intelligent doit maximiser sa mesure de performance en formulant un but et en essayant de le satisfaire.
- ❑ Exemple : agent en vacances dans Arad en Roumanie doit formuler son but selon la situation actuelle et la mesure de performance.

Approche de l'agent pour la résolution de problème



Approche de l'agent pour la résolution de problème



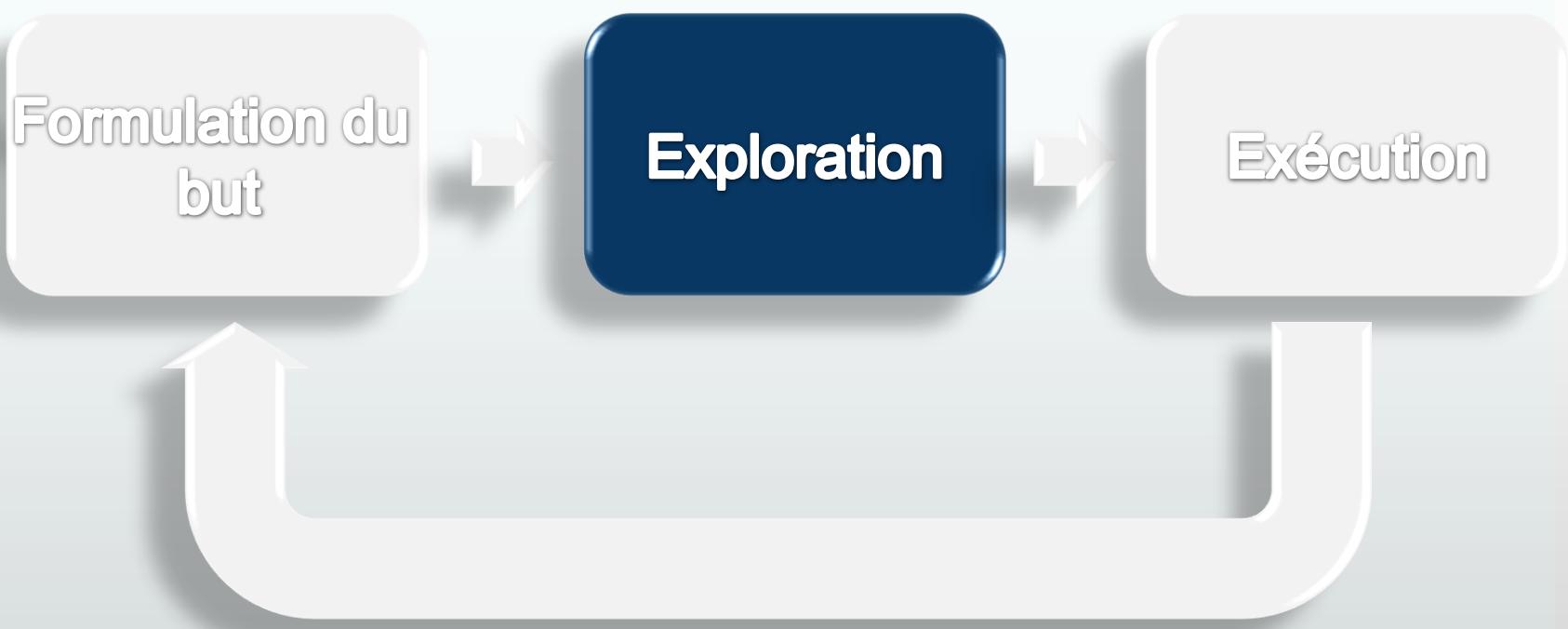
Formulation du but

- ❑ Ensemble d'états où le but est satisfait.
- ❑ L'agent doit découvrir comment agir pour arriver à un état but : **aller à Bucarest**.
- ❑ Il doit donc choisir un ensemble d'actions lui permettant d'atteindre son but.
- ❑ Mais il ne peut pas considérer directement l'état but, il faut le décomposer en sous-buts : c'est la **formulation de problème**.

Formulation de problème

- ❑ Processus consistant à décider quelles actions et quels états considérer en vue d'un but donné :
 - ❑ États : les différentes villes,
 - ❑ Actions : déplacement entre les villes.

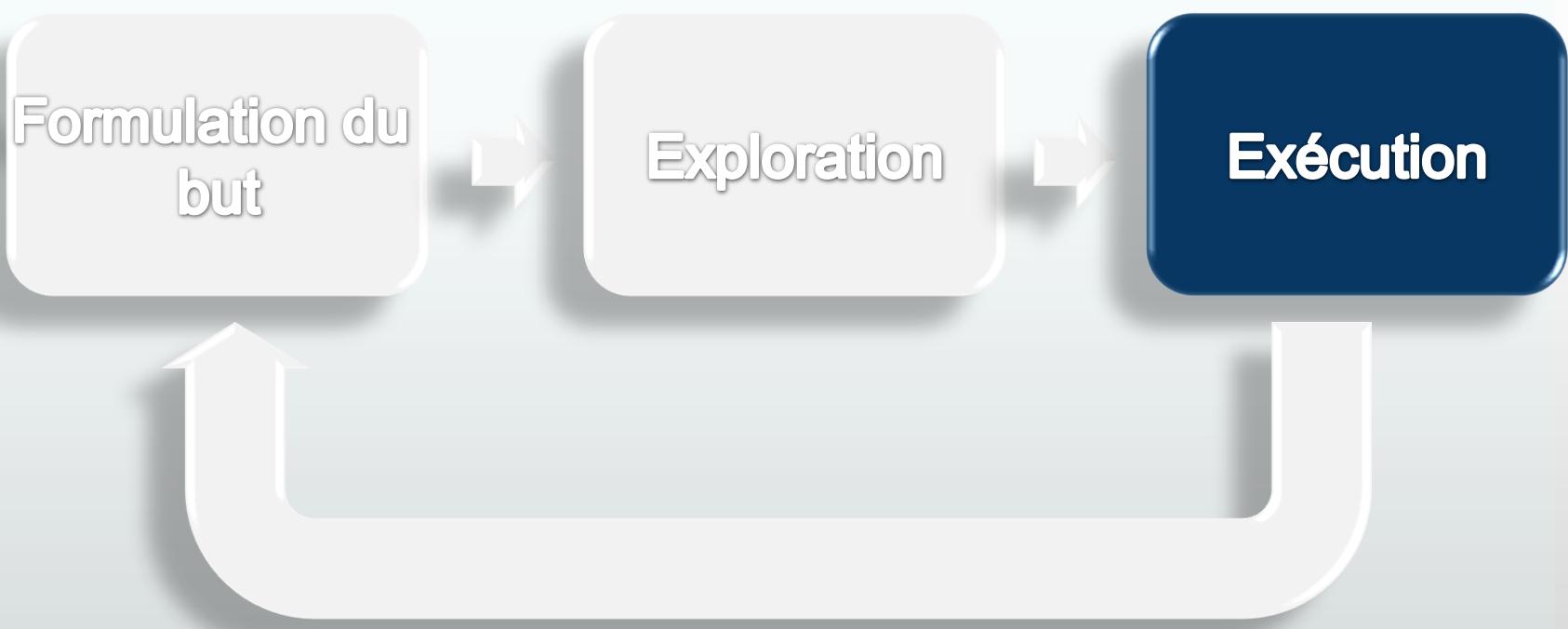
Approche de l'agent pour la résolution de problème



Exploration

- ❑ Processus de recherche d'une séquence d'actions qui atteint le but.
- ❑ Un algorithme d'exploration prend en entrée un problème et renvoie une **solution optimale** sous forme d'une séquence d'actions : **Arad, Subiu, Fagaras, Bucarest.**

Approche de l'agent pour la résolution de problème



Exécution

□ Les actions recommandées sont réellement effectuées.

Agent élémentaire de résolution de problème

fonction AGENT-SIMPLE-RÉSOLUTION-PROBLÈME(*percept*) **retourne** une action persistante: *seq*, une séquence d'actions, initialement vide
état, une description de l'état courant du monde
but, un but, initialement vide
problème, une formulation du problème

```
état ← ACTUALISER-ÉTAT(état, percept)
si seq est vide alors
    but ← FORMULER-BUT(état)
    problème ← FORMULER-PROBLÈME(état, but)
    seq ← EXPLORER(problème)
    si seq = échec alors retourner une action vide
action ← PREMIER(seq)
seq ← RESTE(seq)
retourner action
```

Plan

- ❑ Agents de résolution de problèmes
- ❑ **Problèmes bien définis**
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ Stratégies d'explorations :
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques

Problèmes bien Définis

Définition d'un problème

- ❑ Information que l'agent utilise pour décider quoi faire.
- ❑ Essentiellement :
 - ❑ information = états + actions.

Problème : cinq composants

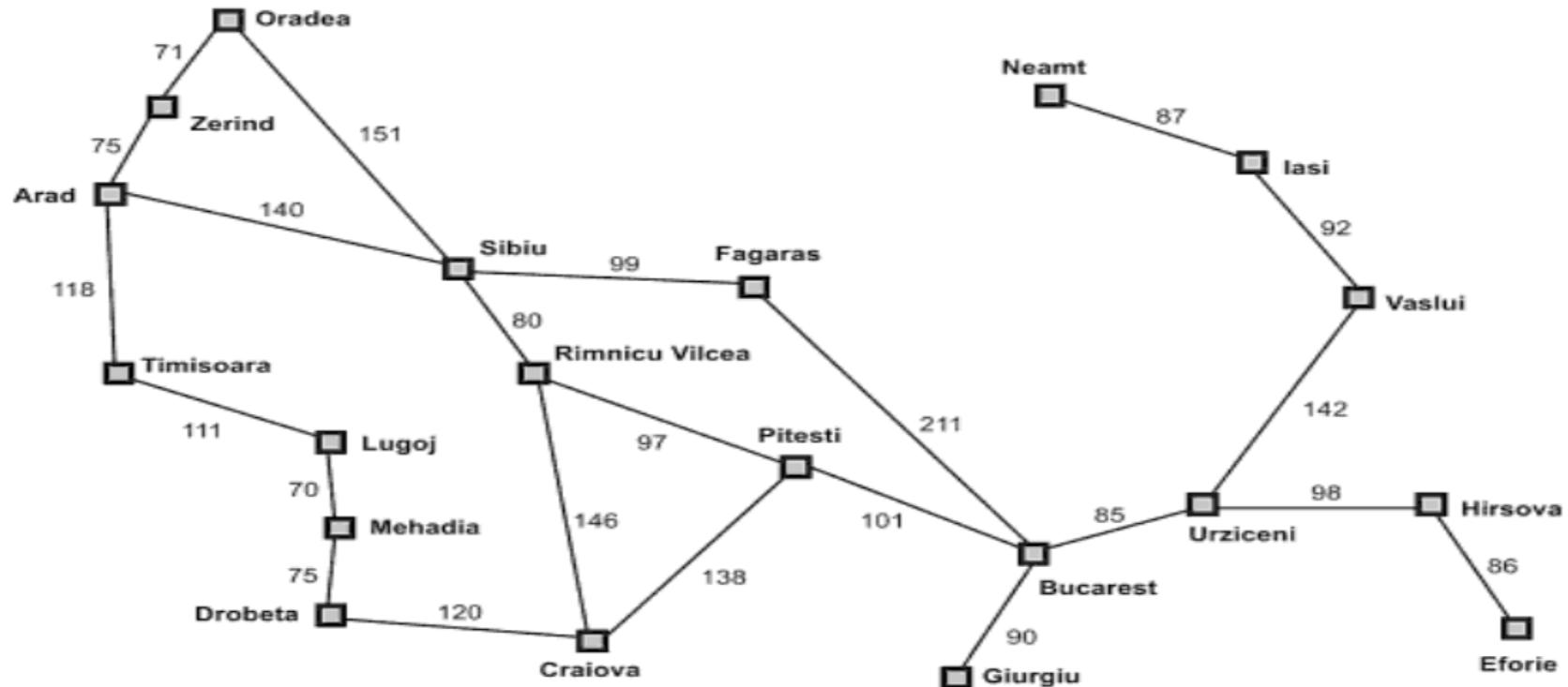
- L'**état initial** du problème : une ville,
- Les **actions** : passer d'un état à un autre,
- **Modèle de transition** : définit l'état résultant de l'exécution d'une action à partir d'un état,
- **Test de buts** : détermine si un état donné est un état but (concret : Dans(Bucarest), abstrait : état d'échec et mat).
- Le **coût** du chemin : coût numérique de performance de l'agent (longueur en km).

Espace d'états

- **Espace d'états** = état initial + actions + modèle de transitions.
- L'ensemble de tous les états accessibles par une séquence d'actions à partir de l'état initial.
- L'espace d'états est un **graphe** dont les nœuds sont les états et les liens entre les nœuds forment les actions.
- Un **chemin** est une séquence d'états reliées par une séquence d'actions.

Graphe de l'espace d'états de l'agent en Roumanie

- Abstraction : suppression des détails : paysage, météo, radio, ...



Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ **Exemples de problèmes**
- ❑ Recherche de solutions
- ❑ Stratégies d'explorations :
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques

Exemples de problèmes

Types de problèmes

- **Problèmes de jouets** (*Toys problems*) : illustrer ou expérimenter diverses méthodes de résolution de problèmes (recherche).
- **Problèmes du monde réel** : un problème réel dont la description est compliquée, mais on peut trouver une idée de leur formalisation :
 - Certains sont des extensions du problème de Roumanie (systèmes embarqués dans les voitures, voyageur de commerce),
 - D'autres sont plus complexes (routage de flux vidéo, planification d'opérations militaires).

Exemples de problèmes de jouets

- ❑ Jeu de taquin,
- ❑ Problème des tours de Hanoi,
- ❑ Problème du chien, de la chèvre et du chou,
- ❑ L'aspirateur,
- ❑ Le problème des huit reines,
- ❑ Les mots croises,
- ❑ Les jeux d'échecs, de dames,...

Exemple 1(Aspirateur)

États :

Les emplacements de l'agent et de la poussière

Actions :

Gauche, droite,
aspirer

État initial :

Un état arbitraire

Coût

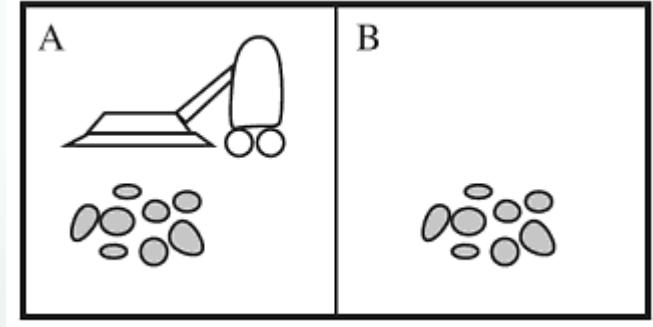
1 par déplacement

Test du but :

Vérifie que le sol est propre

Modèle de transition :

Effets prévus à part quelques uns



Exemple 2 (Voyageur en Roumanie)

États :
Graphe des points
de la carte

Actions :
Arrêtes sortantes
d'un point

État initial :
Point de départ
arbitraire

Coût
Distance, durée,
difficulté, danger,
etc.

Test du but :
Destination finale

Modèle de transition :
Ville suivante



Exemple 3 (Jeu de taquin)

États :

Un état est une configuration du jeu de taquin

Actions :

Déplacer une case à droite, à gauche, en haut, en bas.

État initial :

Un état arbitraire

Coût

Un point par déplacement, nombre de cases mal placées, etc.

Test du but :

Un état arbitraire

Modèle de transition :
Etats résultants des actions

5	4	
6	1	8
7	3	2

Etat initial

1	2	3
8		4
7	6	5

Etat final

Exemple 4 (Huit reines)

États :

Configuration de 0 à 8 reines sur la grille sans conflit

Actions :

Ajouter une reine sur n'importe quelle case vide la plus à gauche de la grille sans créer de conflit

État initial :
La grille vide

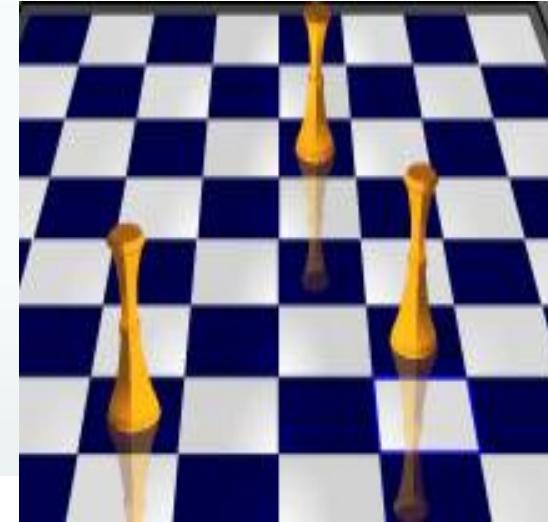
Coût

Un coût constant pour chaque action ou 0

Test du butt :

Configuration de huit reines avec aucune reine sous attaque

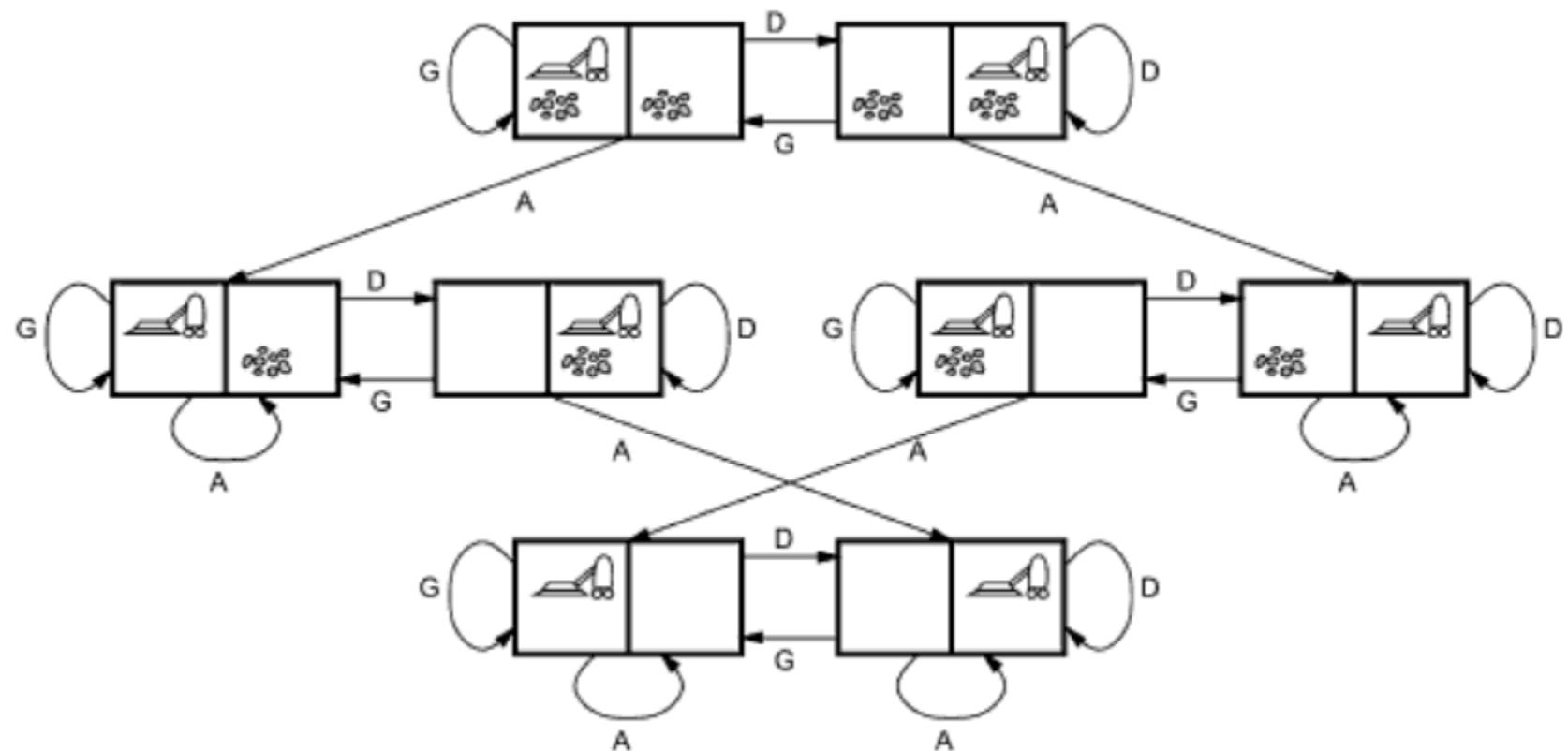
Modèle de transition :
Configuration résultante de l'ajout d'une reine



Exercice (représentation du problème des bidons)

- Vous disposez de 2 bidons, un de 4 gallons et l'autre de 3 gallons, vides.
Il n'y a aucune marque de graduation.
Vous disposez également d'une pompe capable de remplir les bidons.
- Comment remplir le bidon de 4 gallons avec exactement 2 gallons d'eau ?

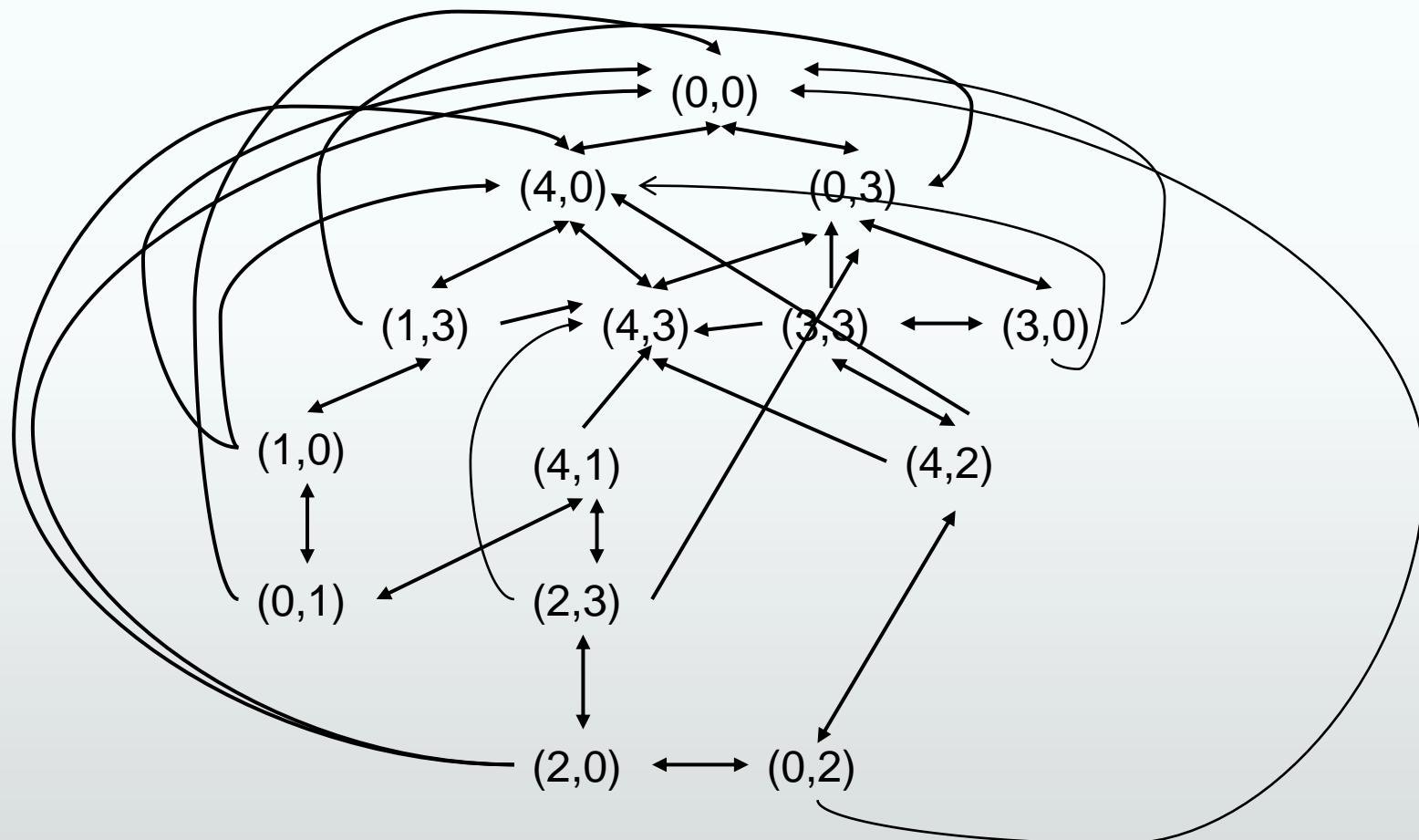
Espace d'états de l'aspirateur



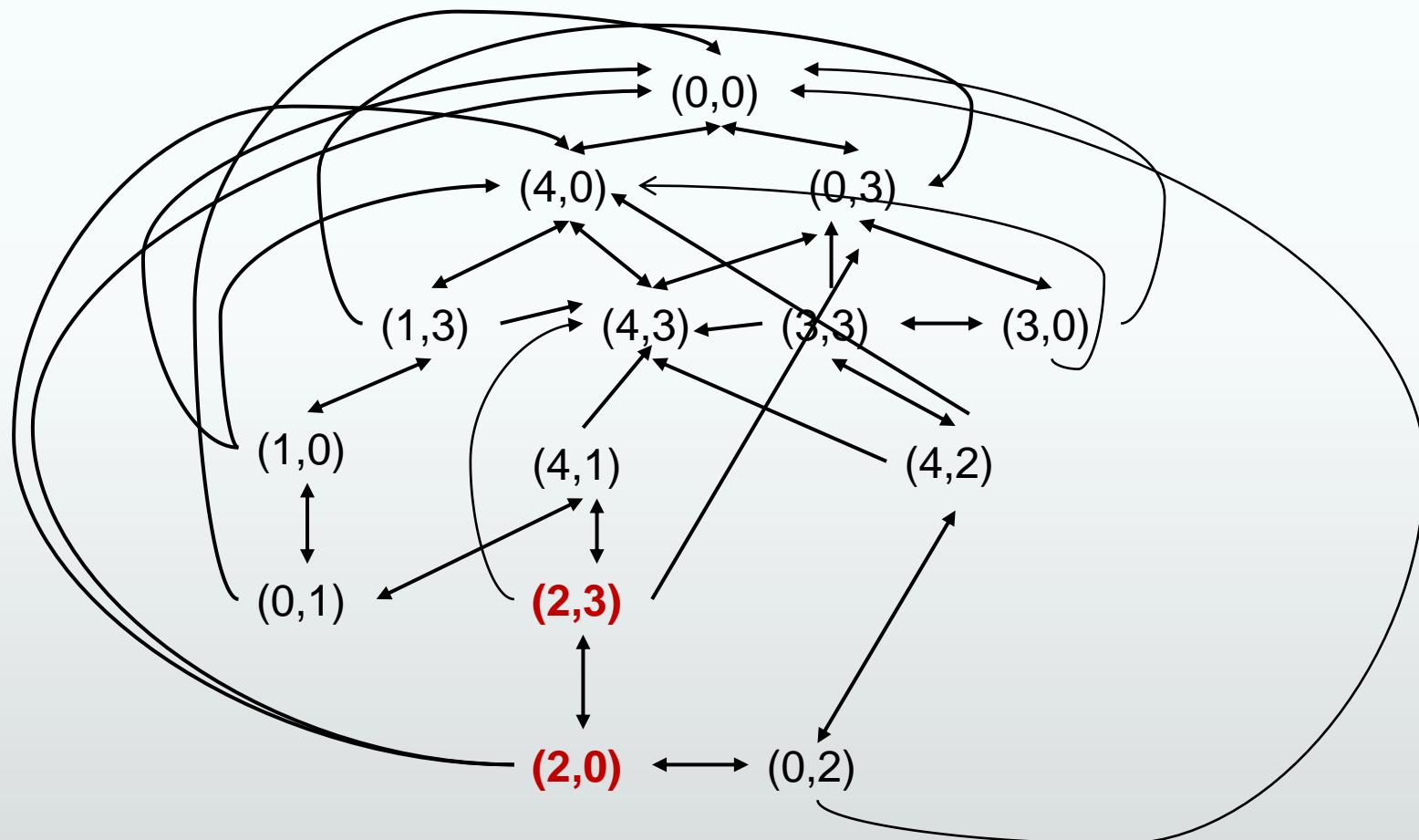
Exercice

□ Représenter l'espace d'états du problème de bidons.

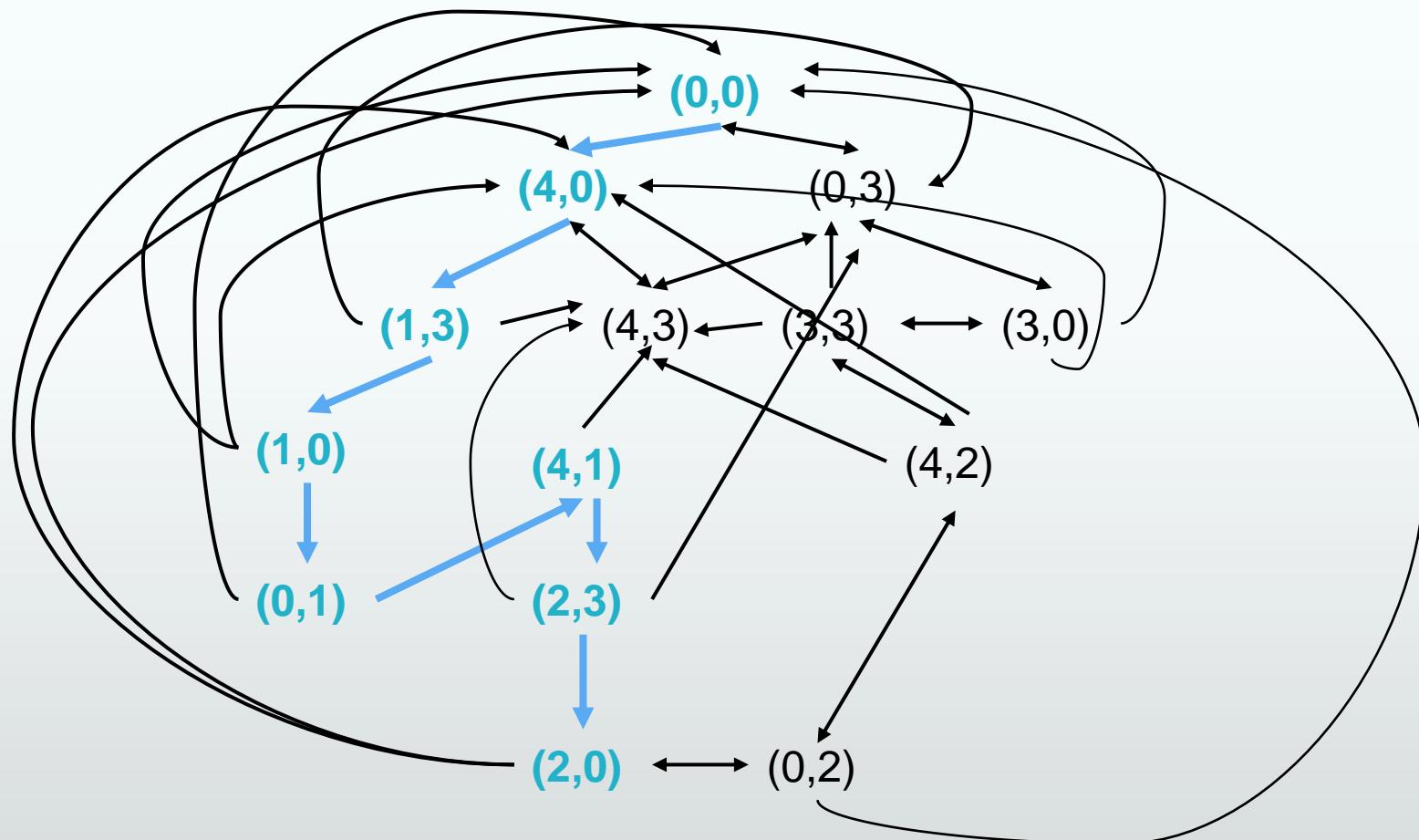
Espace d'états du problème des bidons



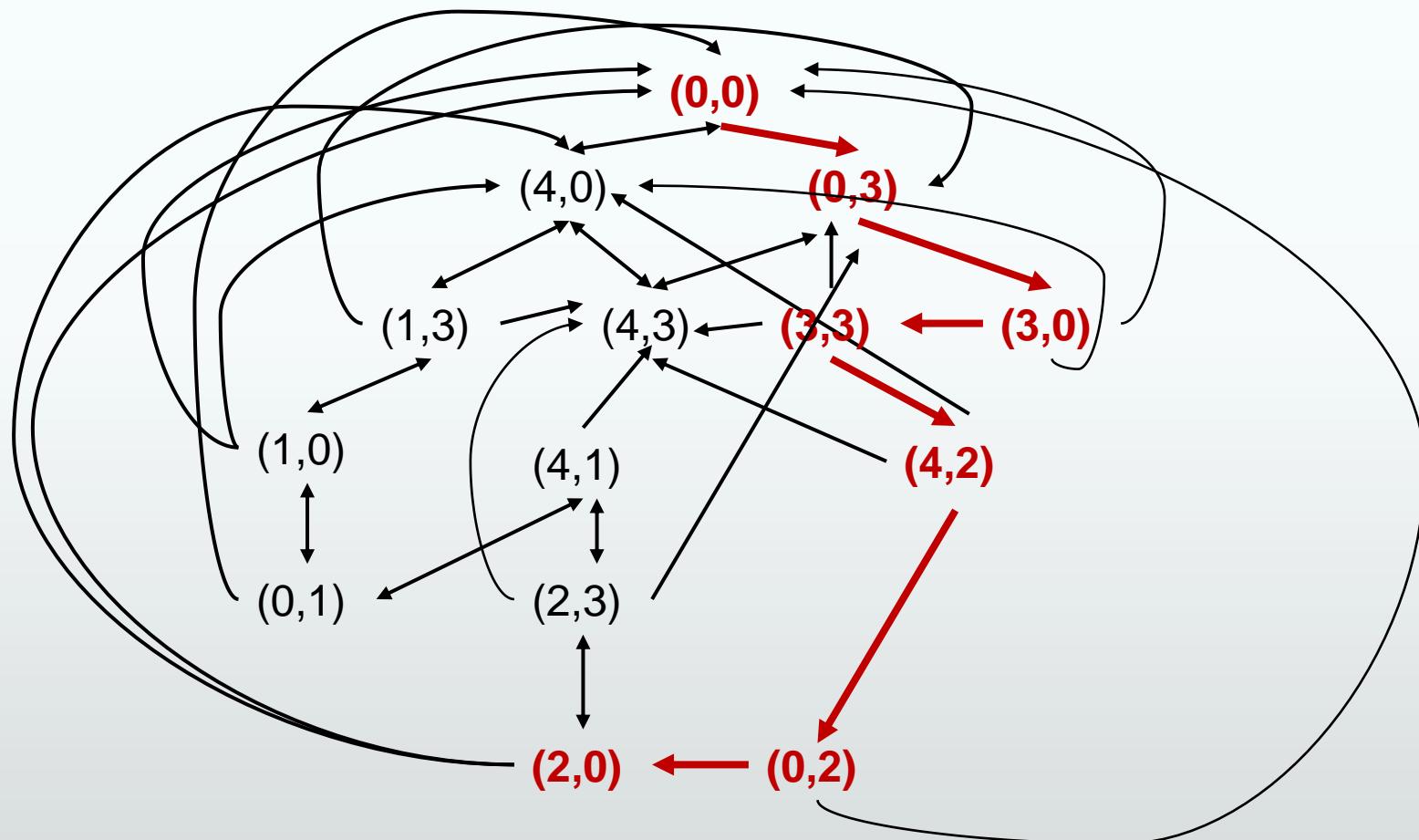
Etats buts dans l'espace d'états du problème des bidons



Chemin (non une solution) dans l'espace d'états



Une solution (optimale) du problème des bidons

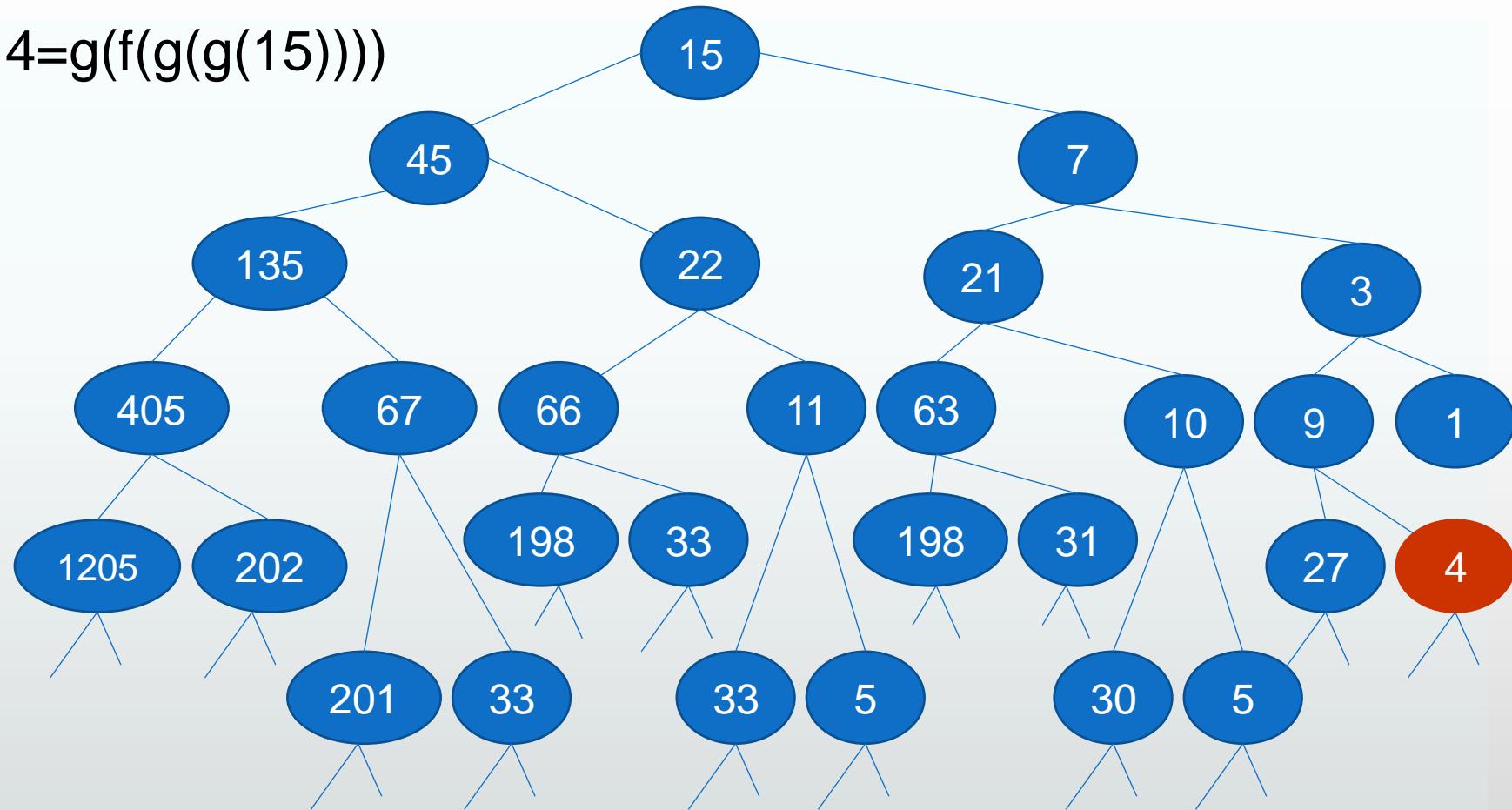


Exemple d'un espace d'états infini

- Les états : entiers naturels.
- Nœud initial : 15.
- Nœud but : 4.
- Les actions sont deux fonctions f et g :
 - $f(x)=3x$,
 - $g(x)=x/2$ (partie entière).
- Modèle de transition : états résultants des actions.
- Coût : constant.

Exemple d'un espace d'états infini

$$4=g(f(g(g(15))))$$



Plan

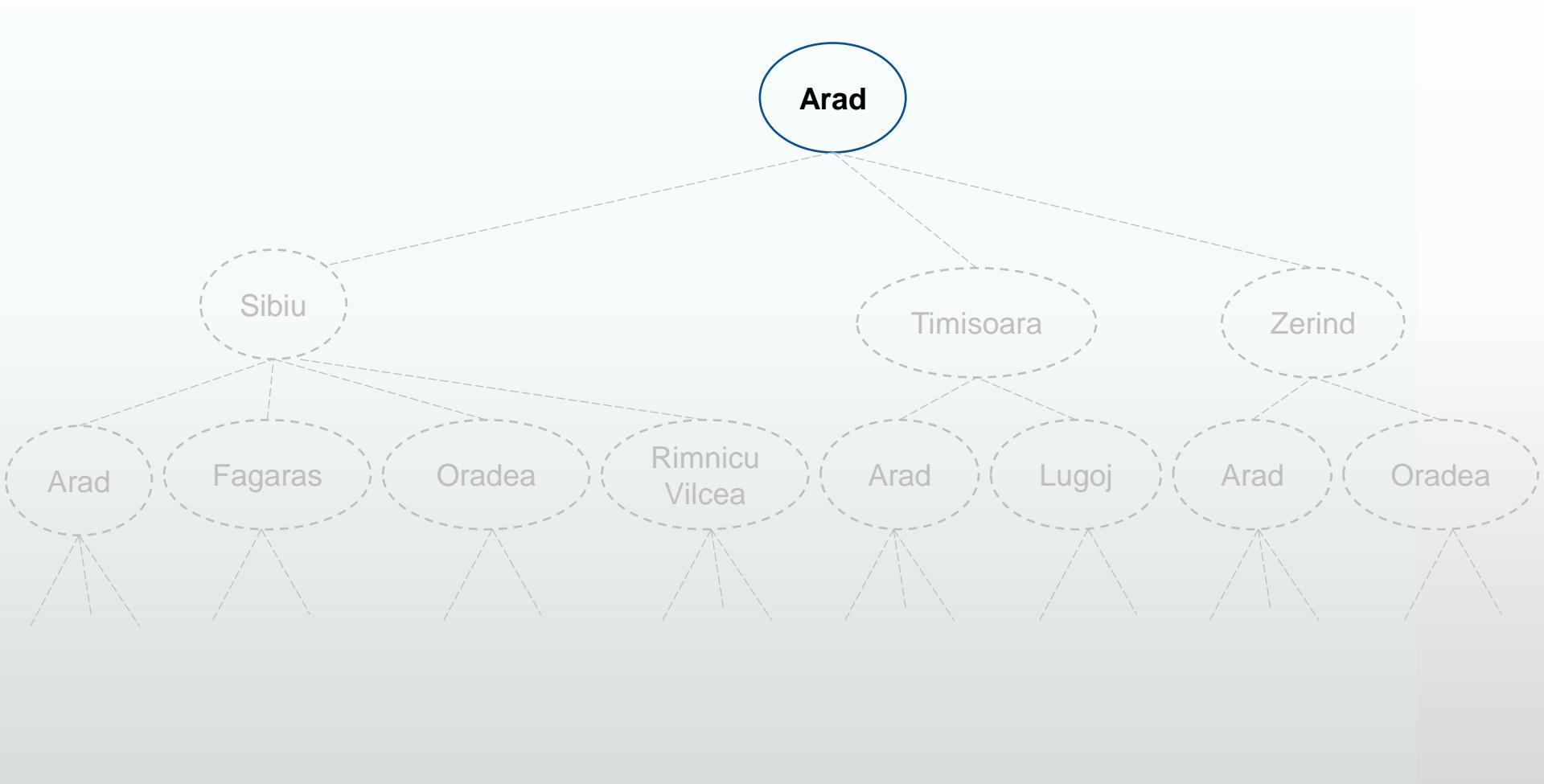
- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ **Recherche de solutions**
- ❑ Stratégies d'explorations :
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques

Recherche de solutions

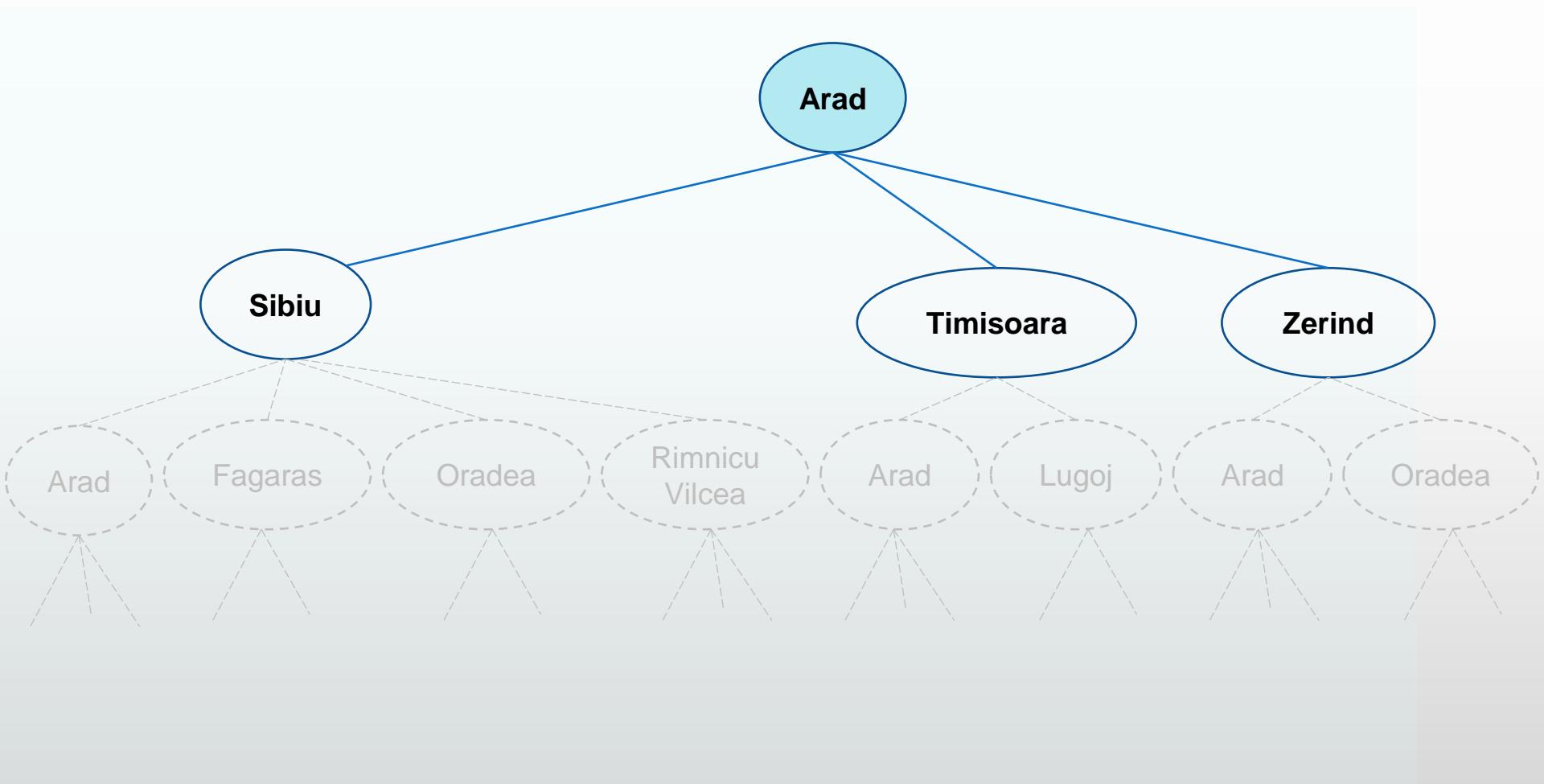
Recherche de solutions

- ❑ Une solution est une séquence d'actions.
- ❑ Les algorithmes d'exploration examinent diverses séquences d'actions possibles à partir de l'état initial, c'est **l'arbre d'exploration** (\neq espace d'états) :
 - ❑ État initial : racine,
 - ❑ Branches : actions,
 - ❑ Nœuds : états de l'espace d'états du problème.

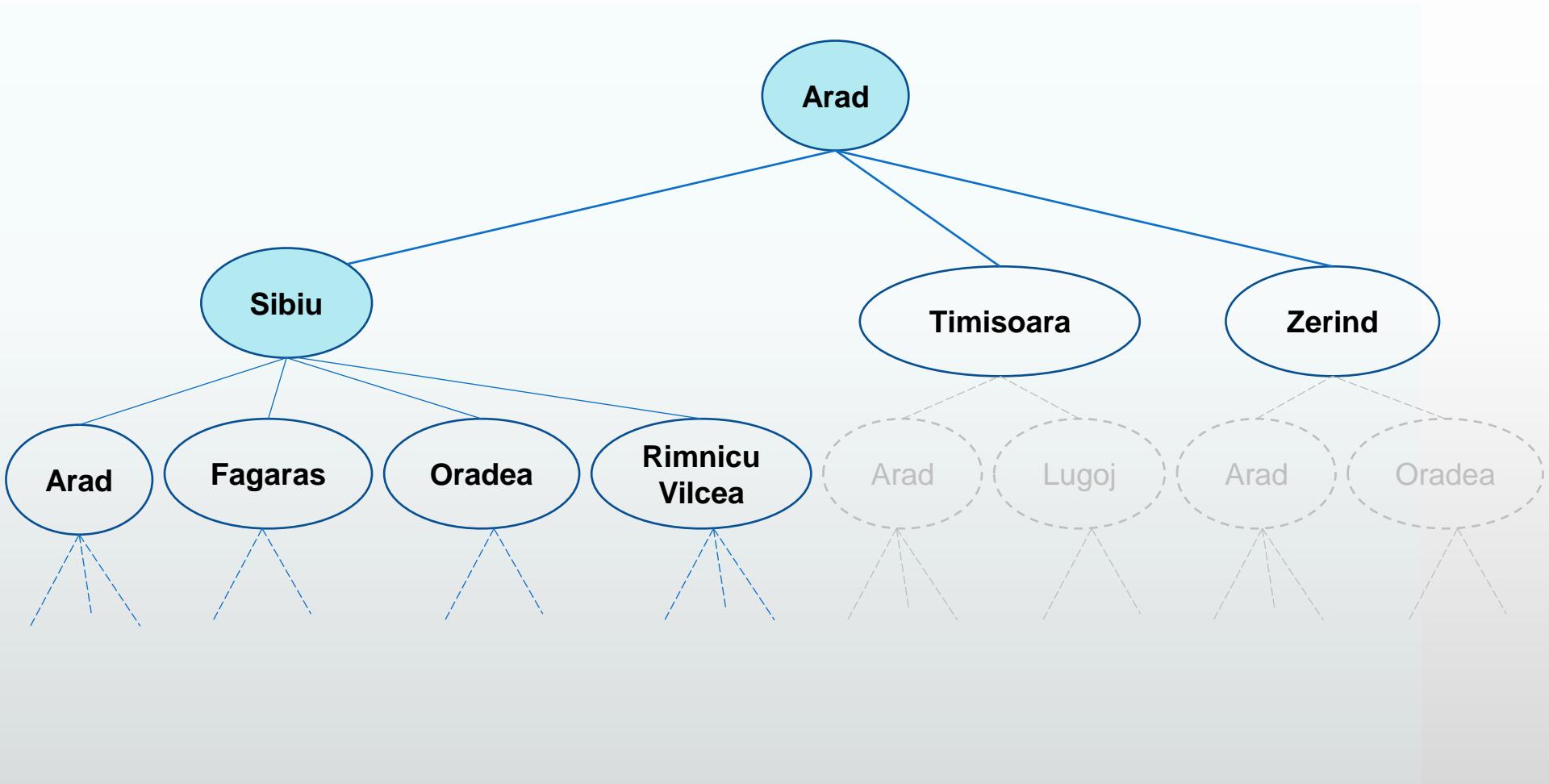
Arbre d'exploration partiel de l'itinéraire Arad-Bucarest



Arbre d'exploration partiel de l'itinéraire Arad-Bucarest



Arbre d'exploration partiel de l'itinéraire Arad-Bucarest

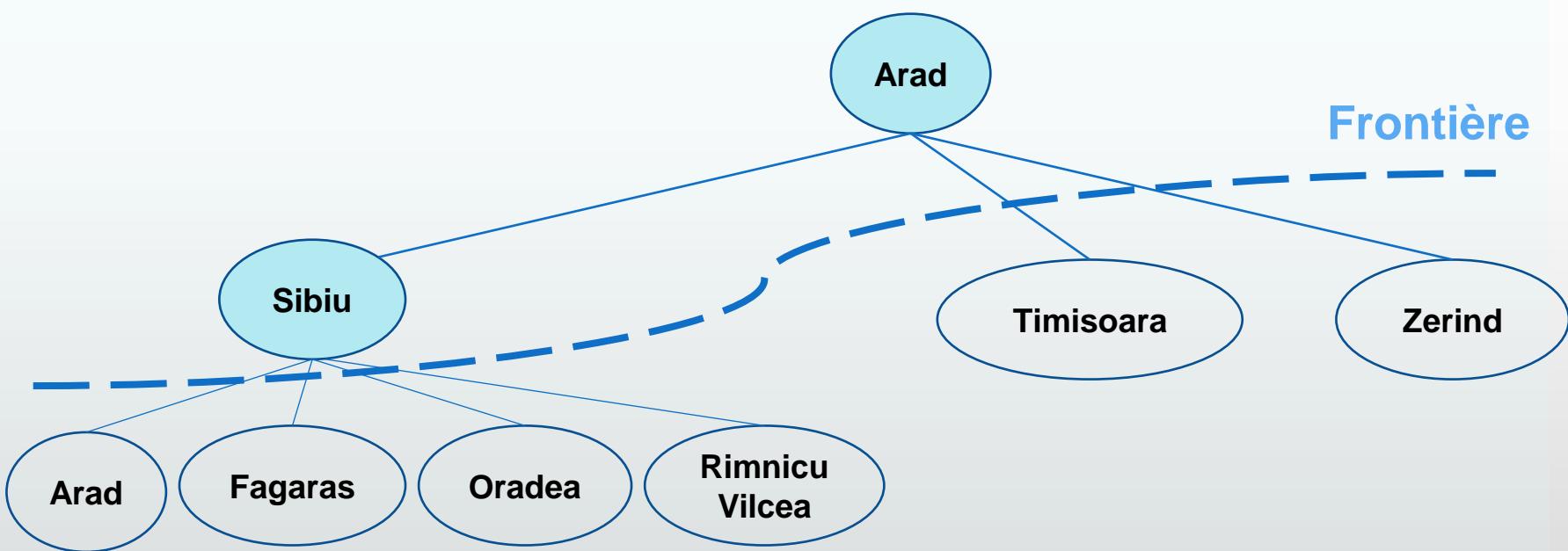


Exploration de l'exemple

1. Le nœud racine est le nœud initial (Arad).
2. On teste si l'état courant (Arad) est un état but.
3. Si oui, on renvoie la solution et arrêt.
4. Si non, et si l'état peut être développé, on développe l'état courant (fils : Sibiu, Timisoara, Zerind).
5. On sélectionne l'état à examiner, les autres restent en attente.
6. On revient à 2 avec l'état sélectionné est l'état courant.

Frontière

- ❑ Nœuds feuilles générés mais non encore développés.



Algorithme général d'exploration en arbre

fonction EXPLORER-ARBRE(*problème*) **retourne** une solution, ou échec

initialiser la frontière avec l'état initial de *problème*

faire en boucle

si la frontière est vide **alors retourner** échec

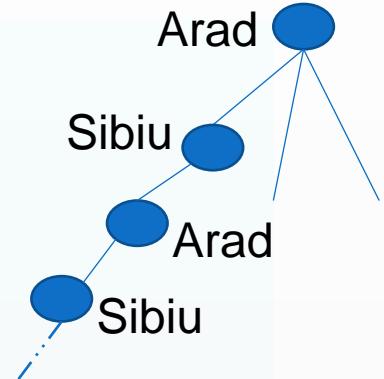
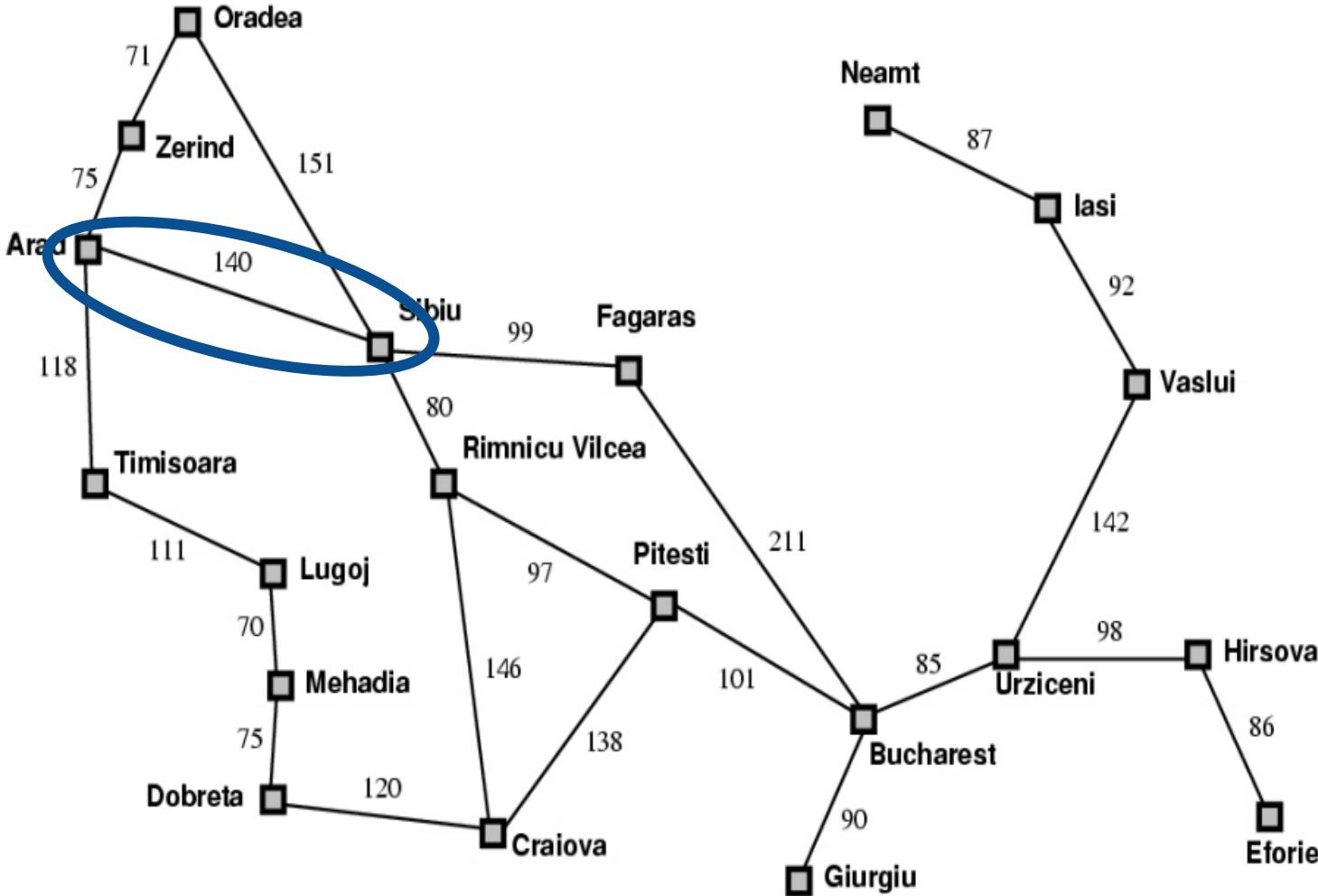
 choisir un nœud feuille et l'enlever de la frontière

si le nœud contient un état but **alors retourner** la solution correspondante
 développer le nœud choisi, en ajoutant les nœuds obtenus à la frontière

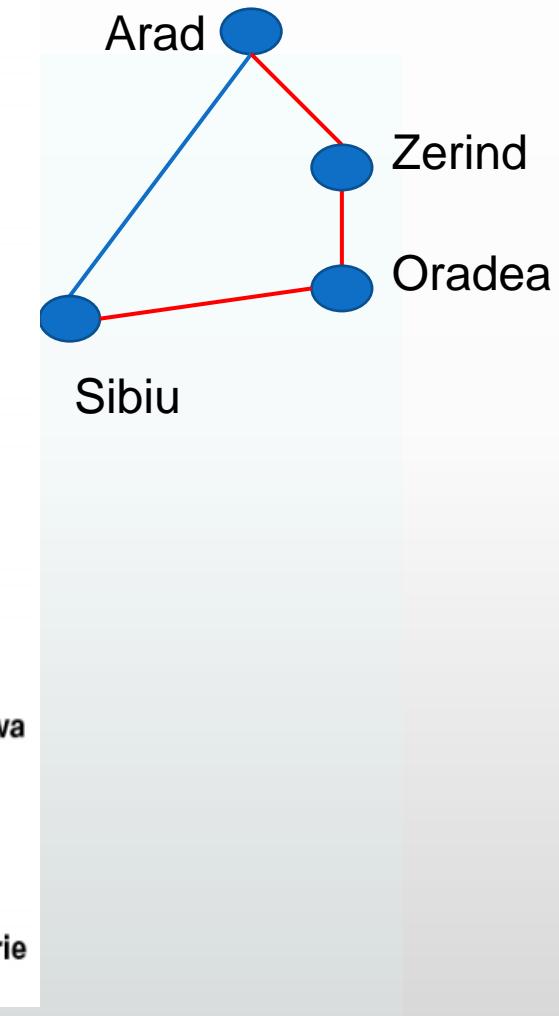
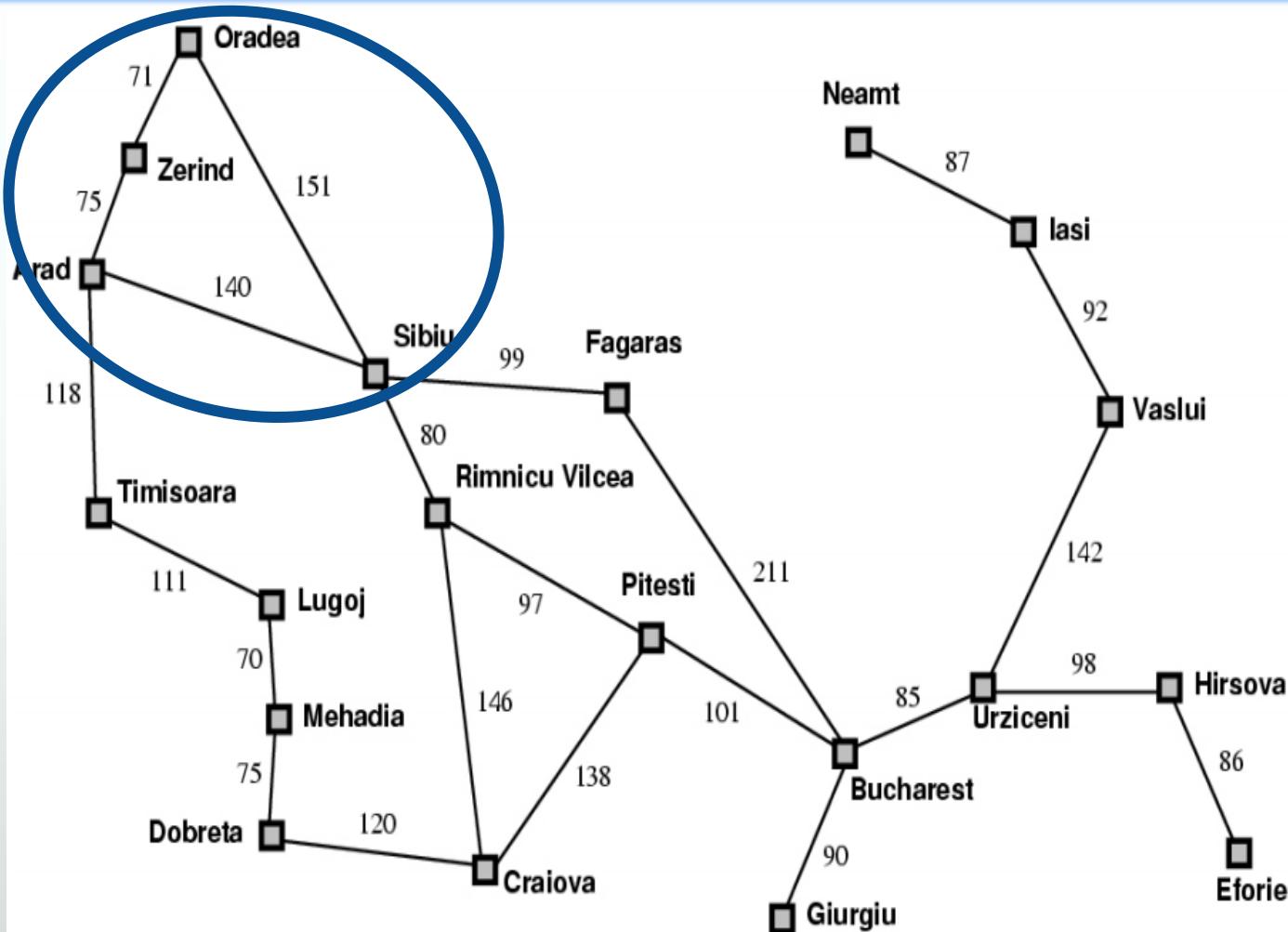
Chemins avec boucles et chemins redondants

- Chemins avec boucles (cas particulier) :
 - Présence d'états répétés,
 - Arbre d'exploration infini.
- Chemins redondants :
 - Plusieurs façons d'aller d'un état à un autre.
 - Problèmes impraticables même si on évite les boucles.

Boucle



Redondance



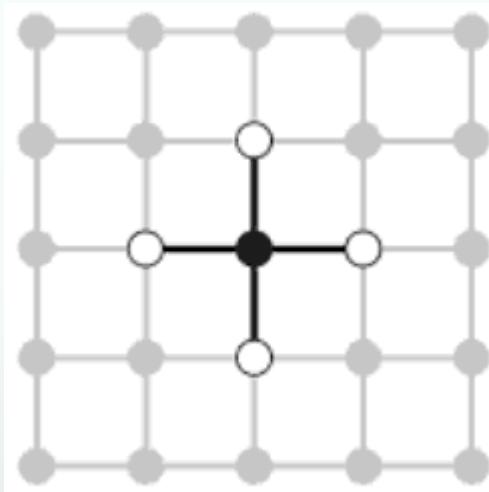
Algorithme général d'exploration en graphe

- *Les algorithmes qui oublient leur histoire sont condamnés à la répéter*

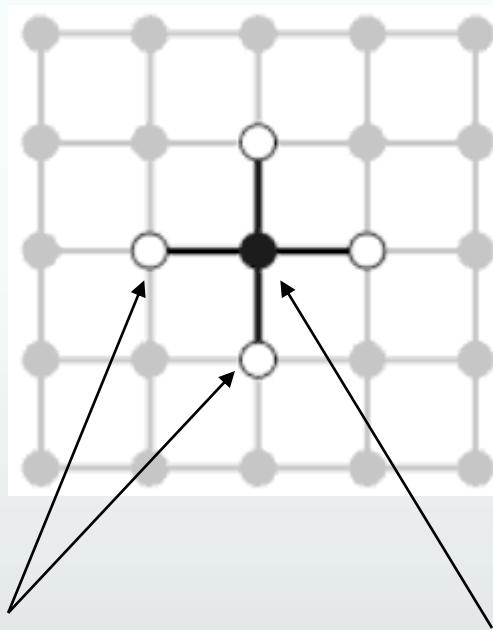
Artificial Intelligence : modern approach

```
fonction EXPLORER-GRAPHE(problème) retourne une solution, ou échec
    initialiser la frontière avec l'état initial de problème
    initialiser l'ensemble des nœuds explorés à vide
    faire en boucle
        si la frontière est vide alors retourner échec
        choisir un nœud feuille et l'enlever de la frontière
        si le nœud contient un état but alors retourner la solution correspondante
        ajouter le nœud à l'ensemble des nœuds explorés
        développer le nœud choisi, en ajoutant les nœuds obtenus à la frontière
        seulement si ils ne sont ni dans la frontière,
        ni dans l'ensemble des nœuds explorés.
```

Séparation noeuds Frontière-explorés



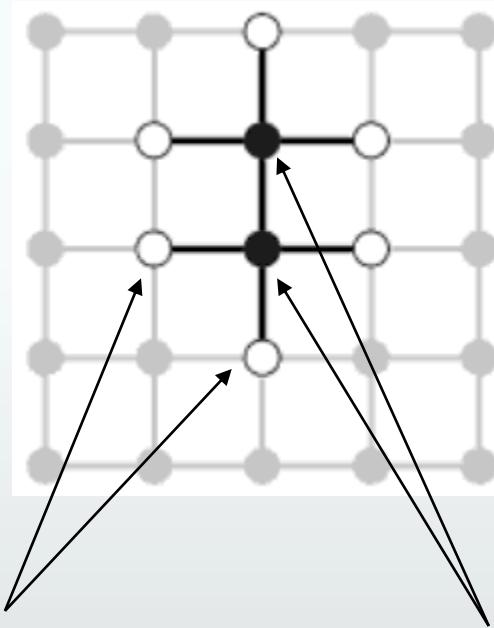
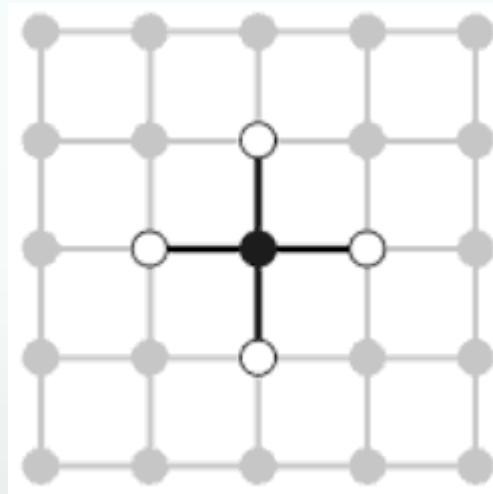
Séparation nœuds Frontière-explorés



Nœuds de
la frontière

Nœud
exploré

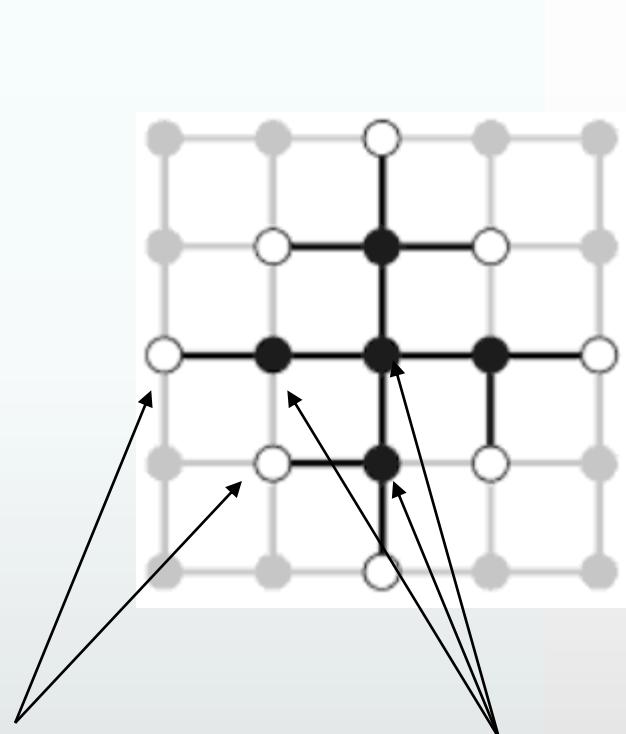
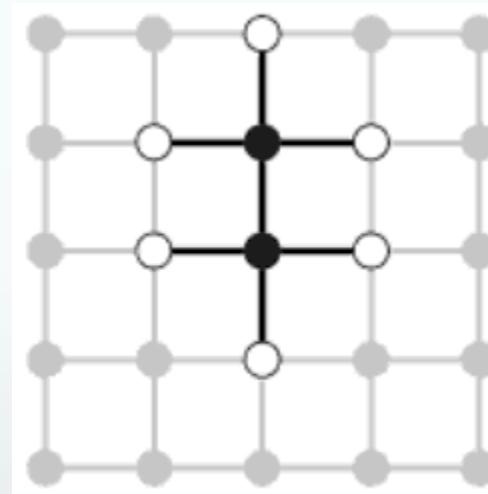
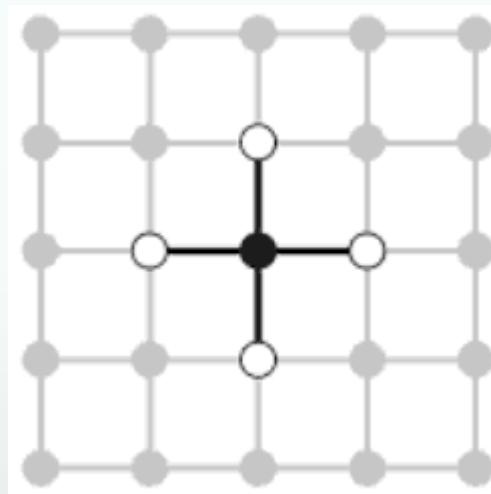
Séparation nœuds Frontière-explorés



Nœuds de la frontière

Nœuds exploré

Séparation noeuds Frontière-explorés



Nœuds de la frontière

Nœuds exploré

Exploration

- ❑ Les algorithmes d'exploration ont la même structure de base, ils diffèrent par la **stratégie d'exploration**.
- ❑ **Nœud** de recherche :
 - ❑ État : l'état de l'espace des états.
 - ❑ Nœud parent : le nœud dans l'arbre d'exploration qui a produit ce nœud.
 - ❑ Action : L'action qui a été appliquée au parent pour générer ce nœud.
 - ❑ Coût du chemin : coût $g(n)$ du chemin à partir de l'état initial jusqu'à ce nœud.
- ❑ **Nœud** \neq état.

Evaluation de la résolution de problèmes

- Complétude : si une solution existe, l'algorithme garanti son obtention.
- Optimalité : la solution trouvée est la meilleure.
- Complexité en temps : temps nécessaire pour trouver la solution.
- Complexité en espace : mémoire nécessaire pour effectuer l'exploration.

Complexité

- La complexité en temps et en espace dépend de :
 - b : facteur de branchement, c'est le nombre maximal de successeurs d'un nœud donné,
 - d : profondeur à laquelle se trouve le meilleur nœud solution (moins d'étapes depuis la racine),
 - m : longueur maximale d'un chemin dans l'espace d'états.

Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ **Stratégies d'explorations :**
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques

Stratégies d'exploration

Types de stratégies d'exploration

□ Exploration aveugle :

- Pas d'autres informations sur les états que celles fournies dans la définition du problème.
- Elles génèrent des successeurs et distinguent un état final d'un état non final.

□ Exploration informée (heuristiques) :

peuvent déterminer si un état non final est meilleur qu'un autre.

Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ Stratégies d'explorations :
 - ❑ **Non informées,**
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques

Exploration non informée

- ❑ N'exploitent **aucune information** sur la structure de l'arbre ou la présence potentielle de nœuds-solution pour **optimiser la recherche** :
 - ❑ Exploration en largeur d'abord
 - ❑ Exploration à coût uniforme
 - ❑ Exploration en profondeur d'abord
 - ❑ Exploration en profondeur limitée
 - ❑ Exploration itérative en profondeur
 - ❑ Exploration bidirectionnelle
- ❑ La plupart des problèmes réels sont susceptibles de provoquer une **explosion combinatoire** du nombre d'états possibles.

Exploration en largeur d'abord

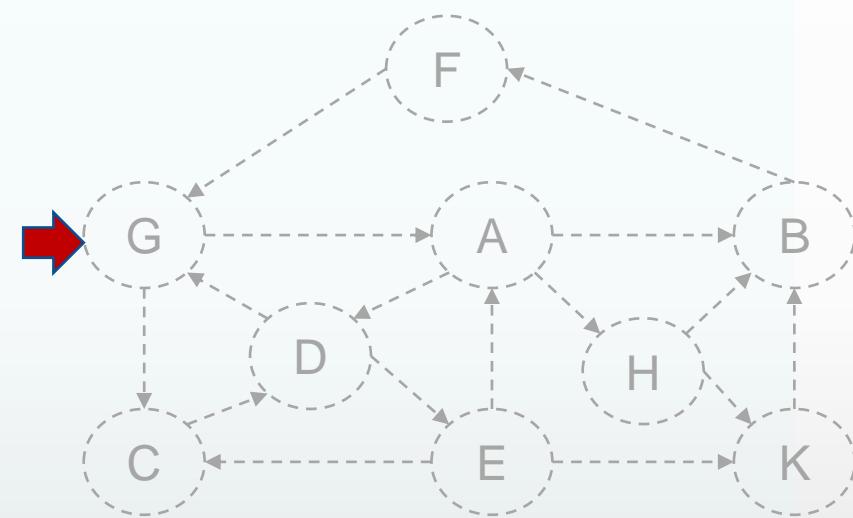
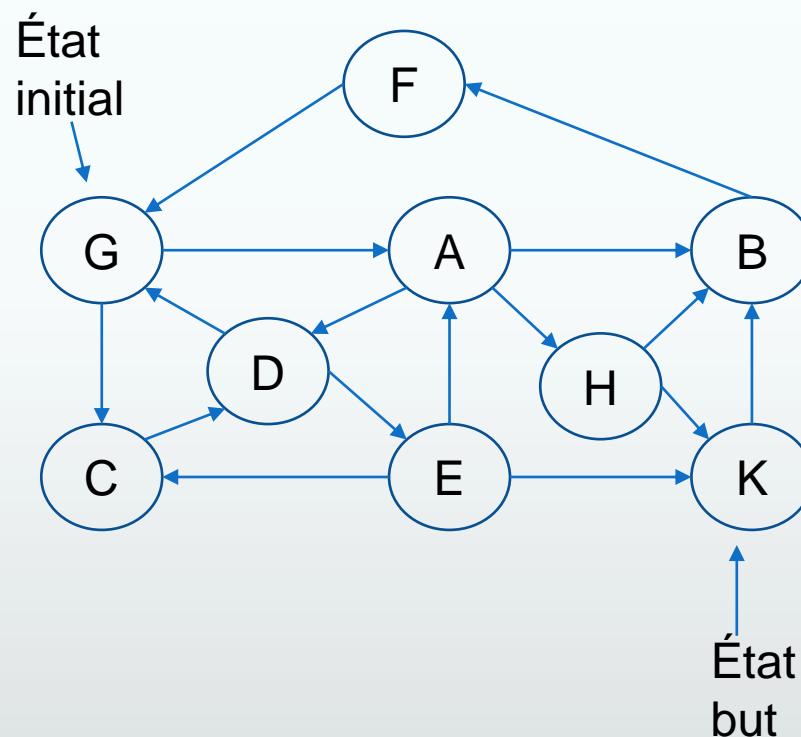
Principe de l'exploration en largeur d'abord

- On commence par développer le nœud racine puis tous les nœuds successeurs, puis les successeurs des successeurs, ...
- Tous les nœuds à une profondeur i sont développés avant ceux de niveau $i+1$.
- Graphe : on rechercherait tous les points dans un rayon circulaire fixe, augmentant ce cercle pour rechercher des intersections de plus en plus loin du nœud initial.
- Il suffit de mémoriser la frontière dans une file FIFO : les nœuds les plus profonds sont stockés à la fin et les moins profonds au début.

Algorithme d'exploration en largeur d'abord

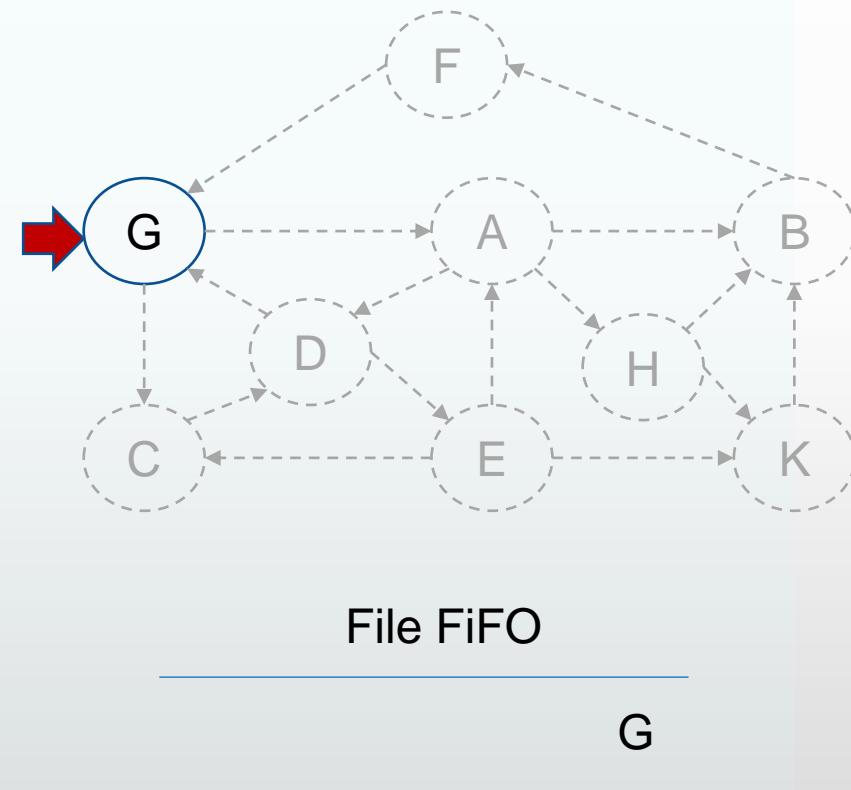
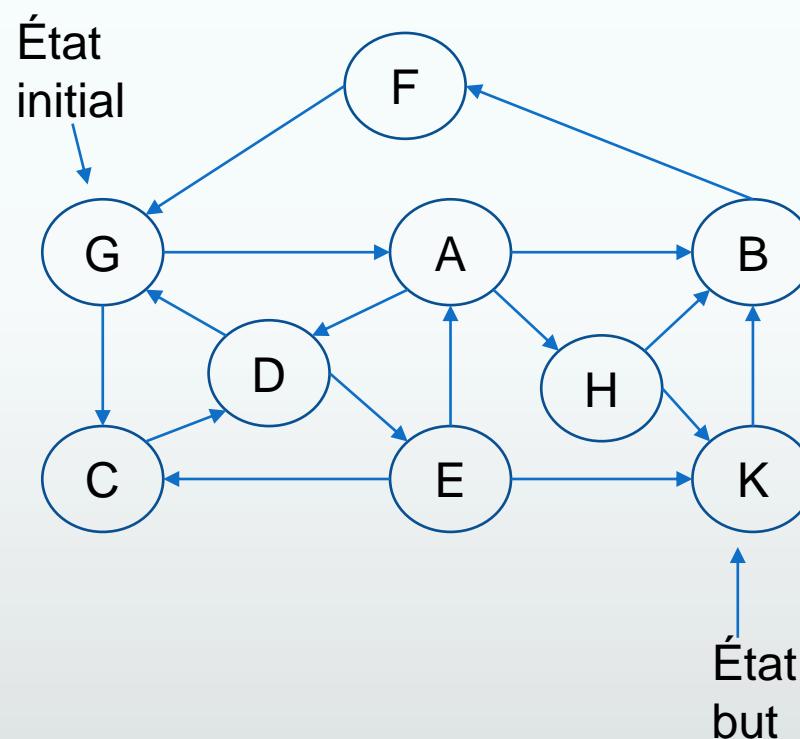
```
fonction EXPLORATION-LARGEUR(problème) retourne une solution, ou échec
    nœud ← un nœud avec ÉTAT = problème.ÉTAT-INITIAL, COÛT-CHEMIN = 0
    si problème.TEST-BUT(nœud.ÉTAT) alors retourner SOLUTION(nœud)
    frontière ← une file FIFO avec nœud comme seul élément
    exploré ← un ensemble vide
    faire en boucle
        si VIDE?(frontière) alors retourner échec
        nœud ← POP(frontière) /* choisit le nœud le moins profond dans la frontière */
        ajoute nœud.ÉTAT à exploré
        pour chaque action dans problème.ACTIONS(nœud.ÉTAT) faire
            fils ← NŒUD-FILS(problème, nœud, action)
            si fils.ÉTAT n'est pas dans exploré ou dans frontière alors
                si problème.TEST-BUT(fils.ÉTAT) alors retourner SOLUTION(fils)
                frontière ← INSÉRER(fils, frontière)
```

Exemple d'exploration en largeur d'abord en graphe

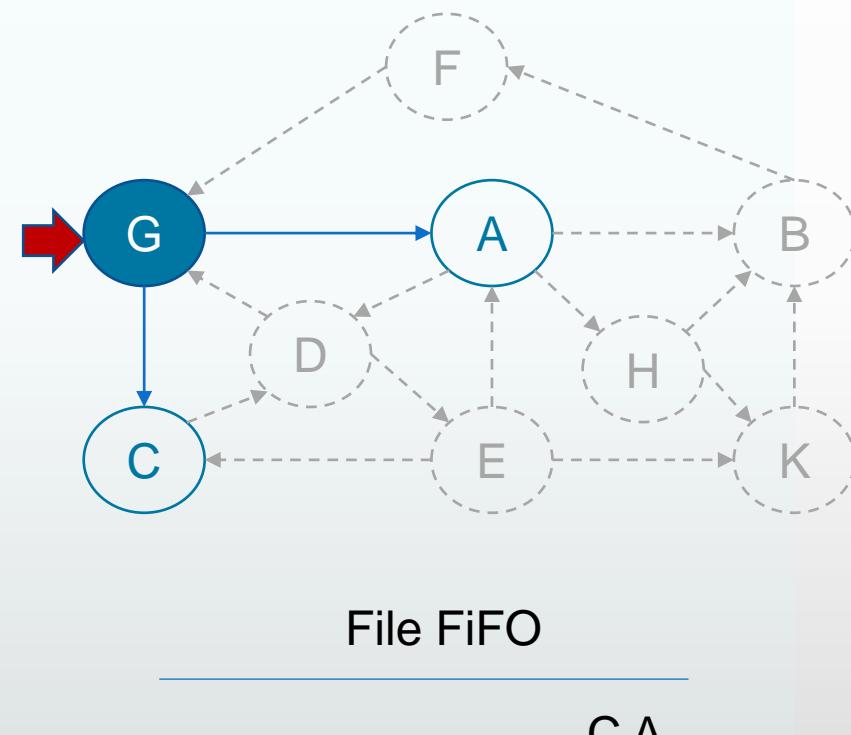
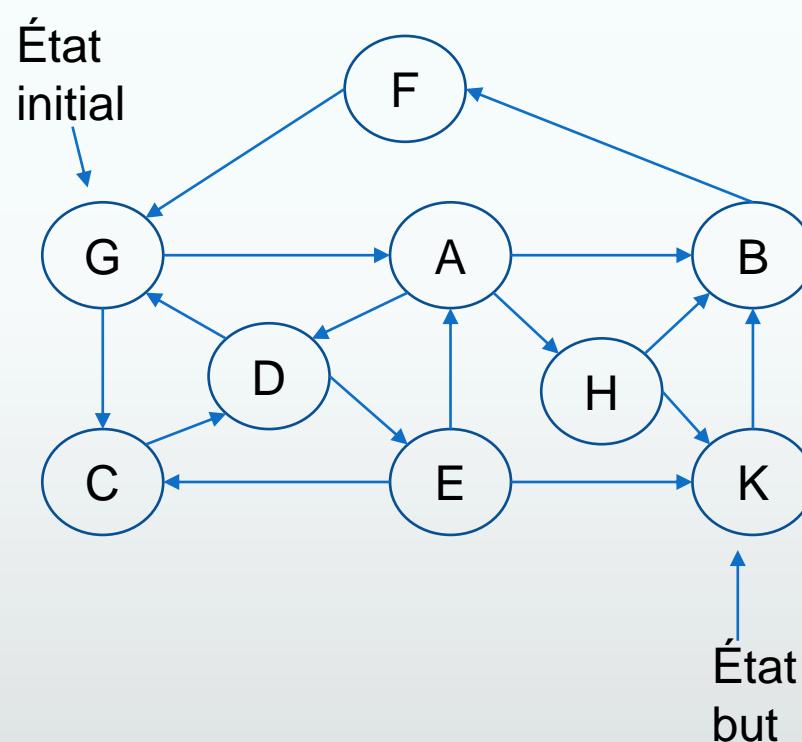


File FiFO

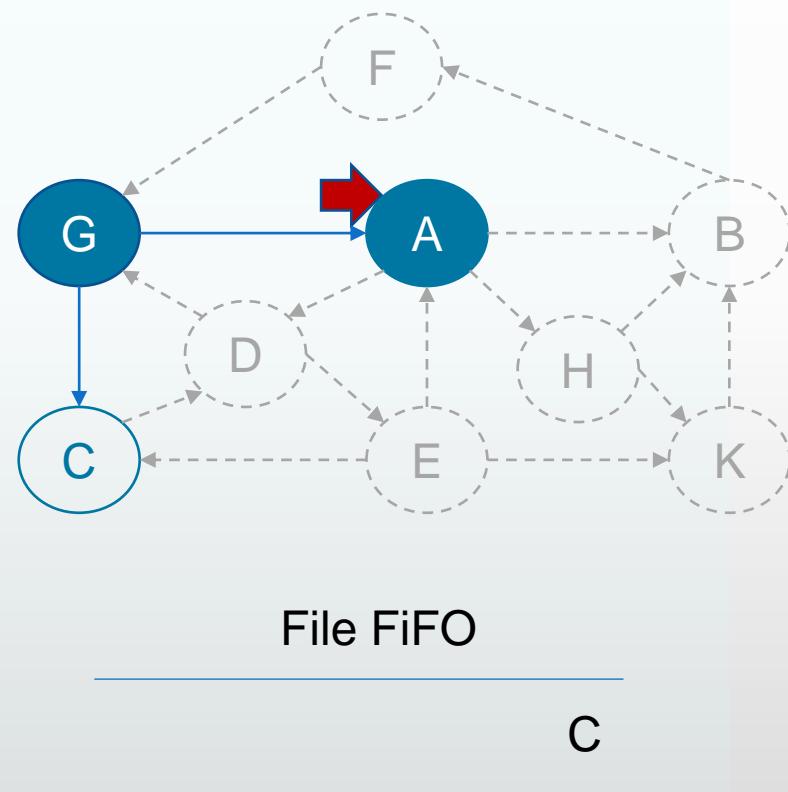
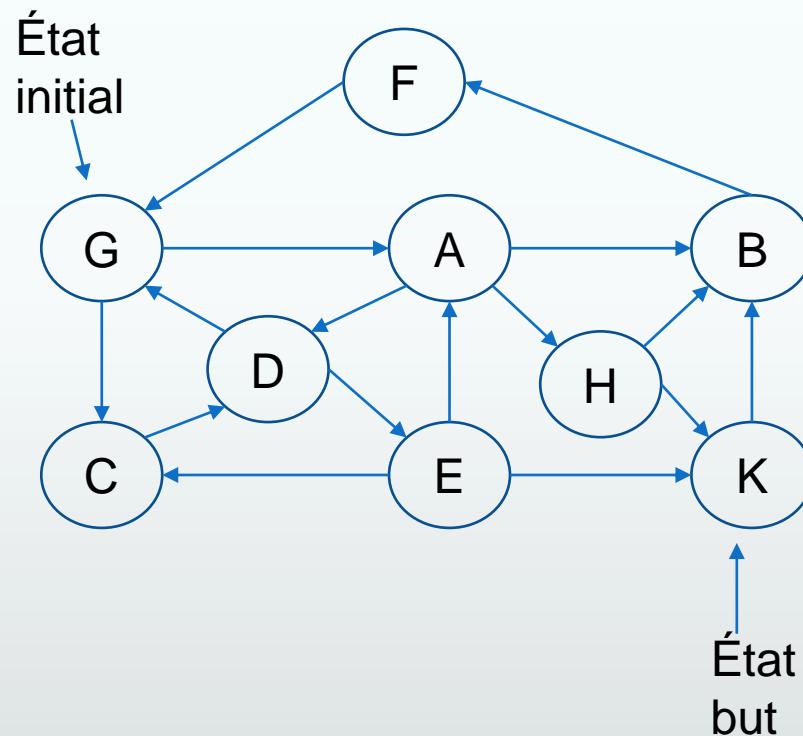
Exemple d'exploration en largeur d'abord en graphe



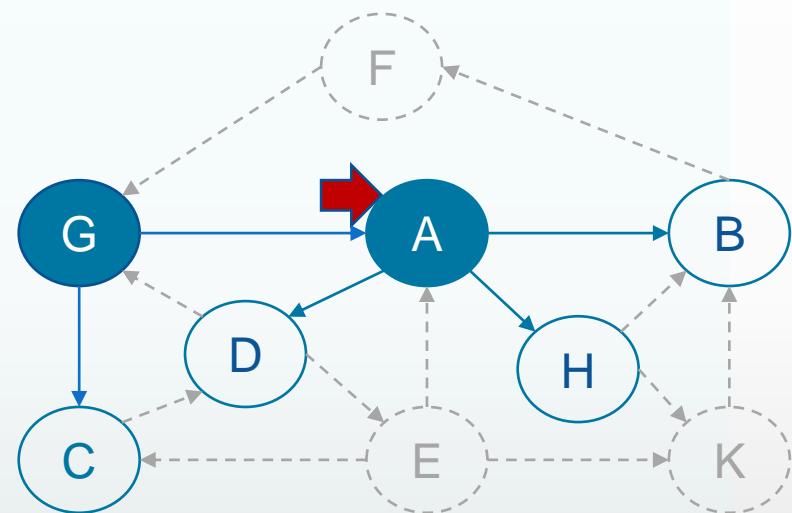
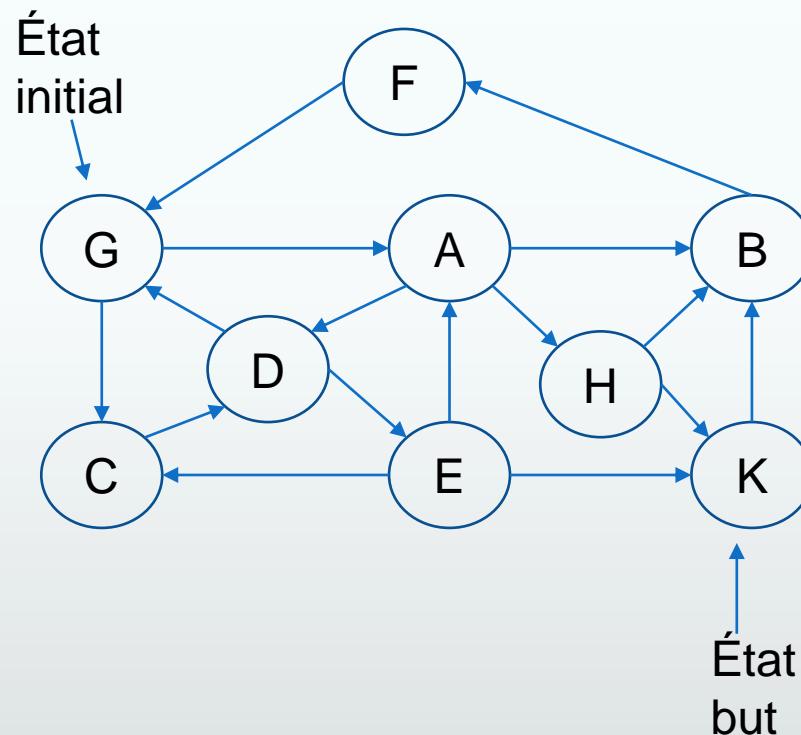
Exemple d'exploration en largeur d'abord en graphe



Exemple d'exploration en largeur d'abord en graphe



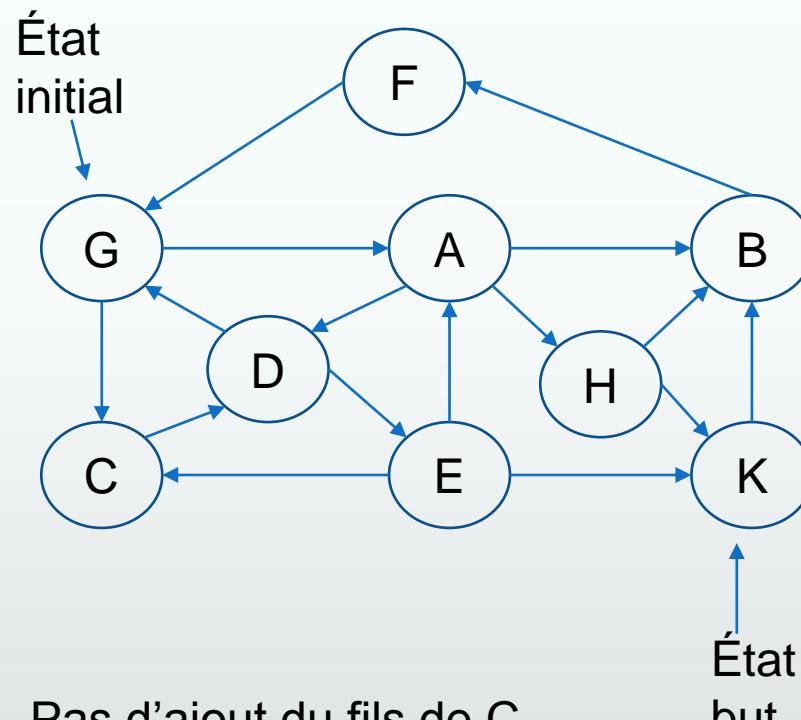
Exemple d'exploration en largeur d'abord en graphe



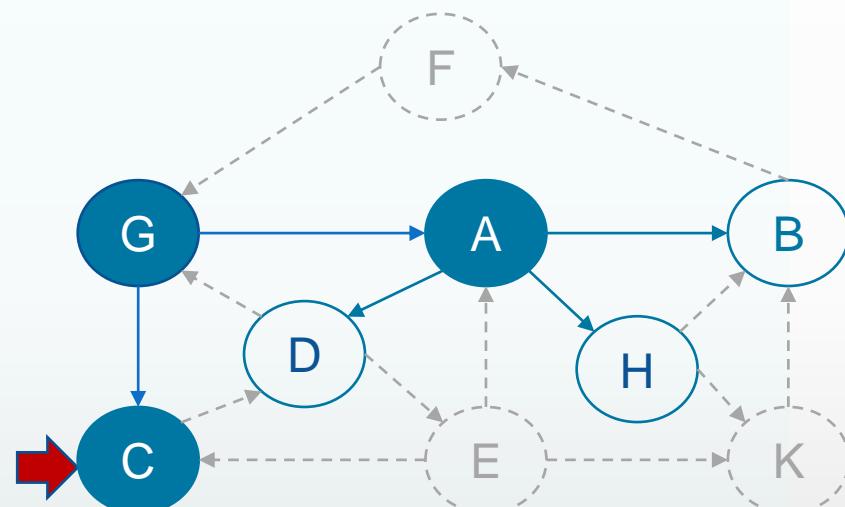
File FiFO

H D B C

Exemple d'exploration en largeur d'abord en graphe



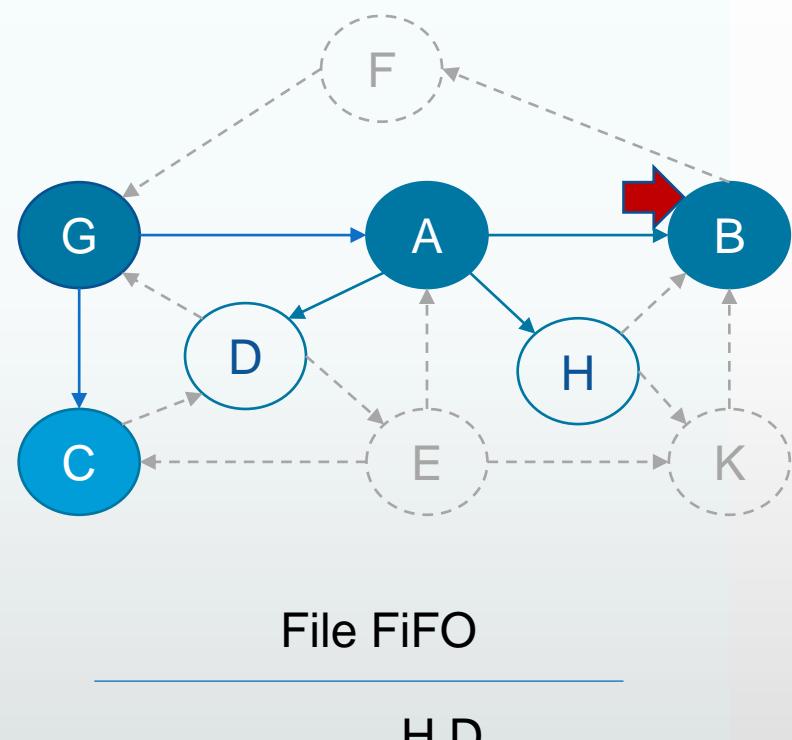
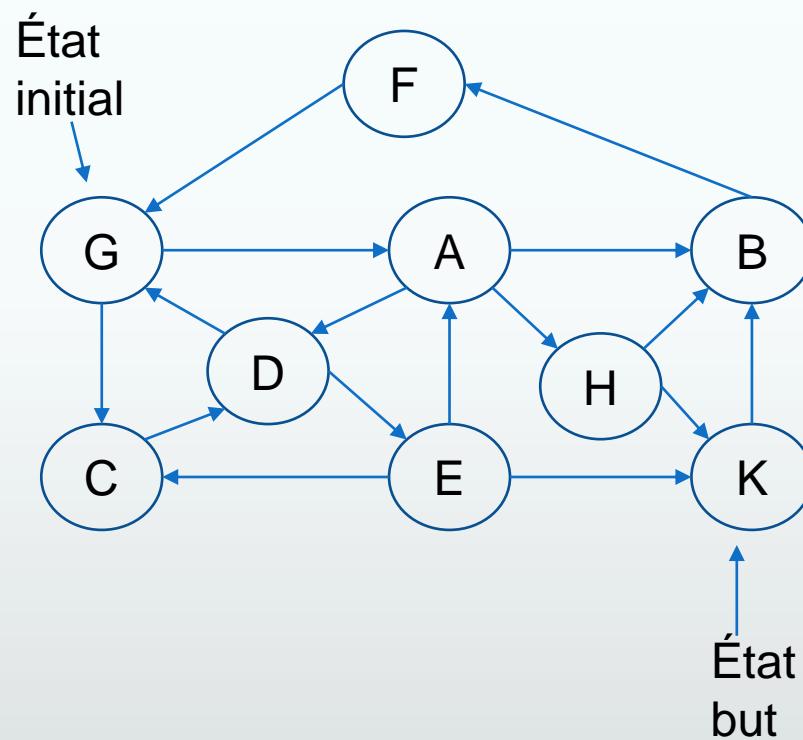
Pas d'ajout du fils de C (D) à la frontière car il y est déjà



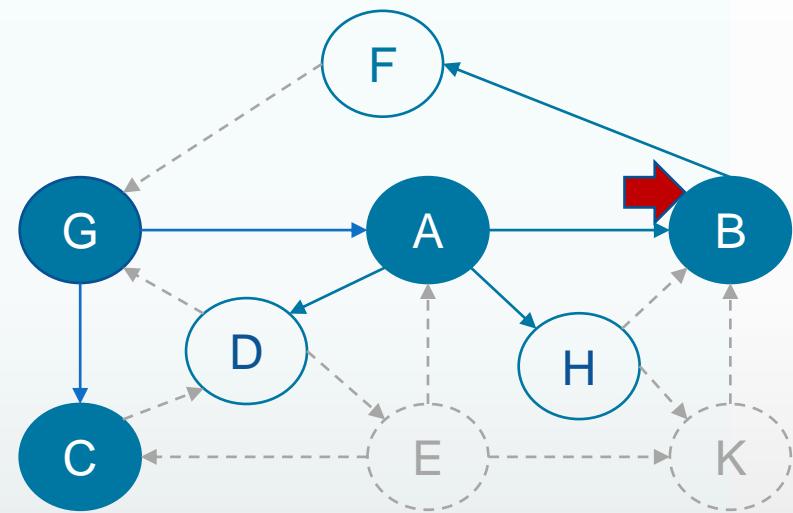
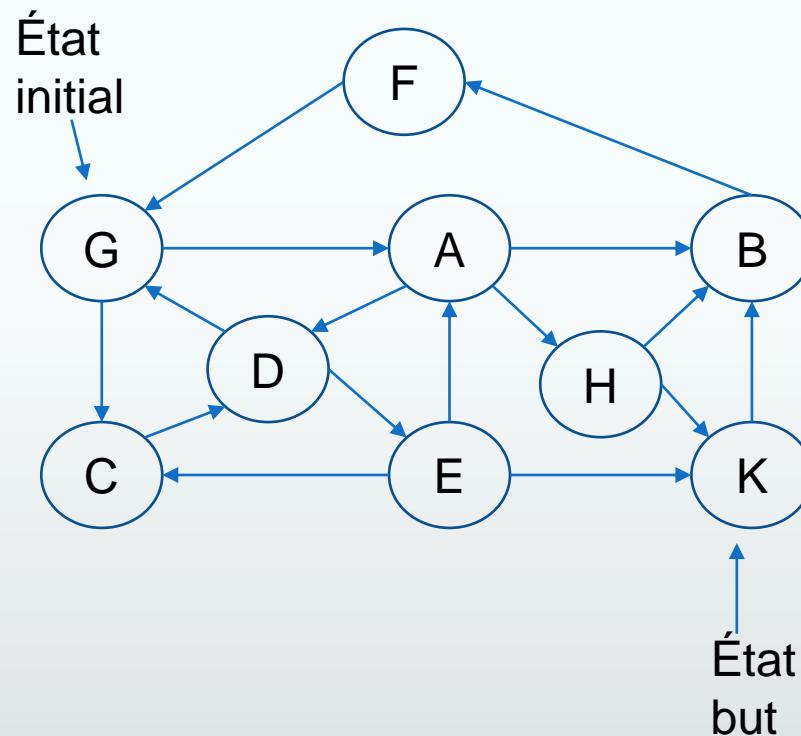
File FiFO

H D B

Exemple d'exploration en largeur d'abord en graphe



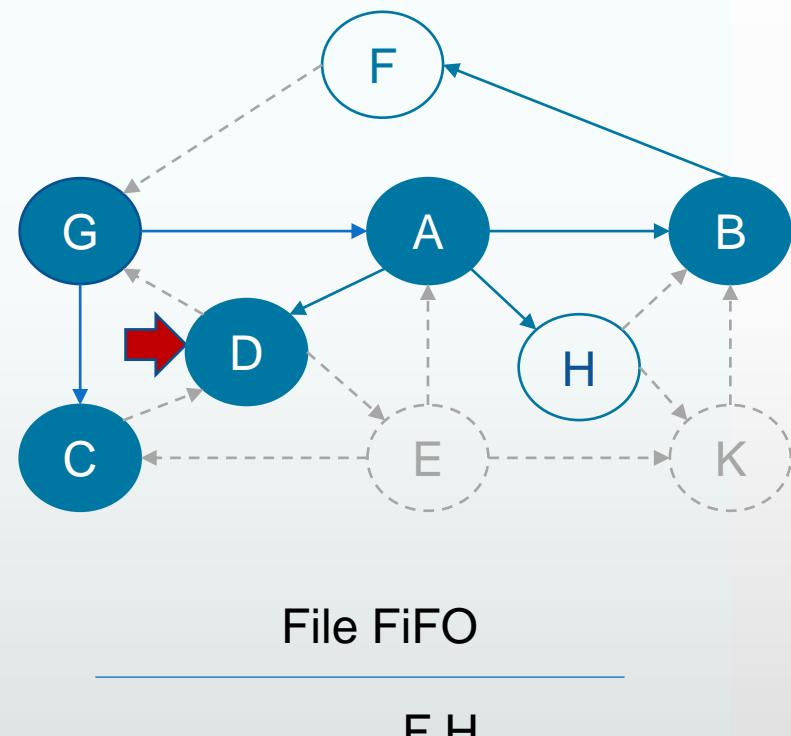
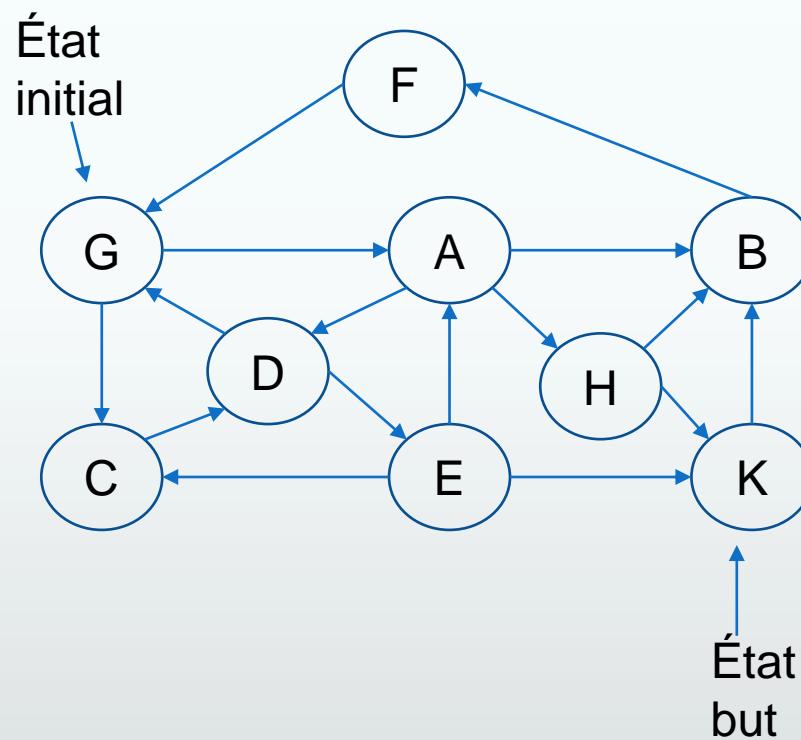
Exemple d'exploration en largeur d'abord en graphe



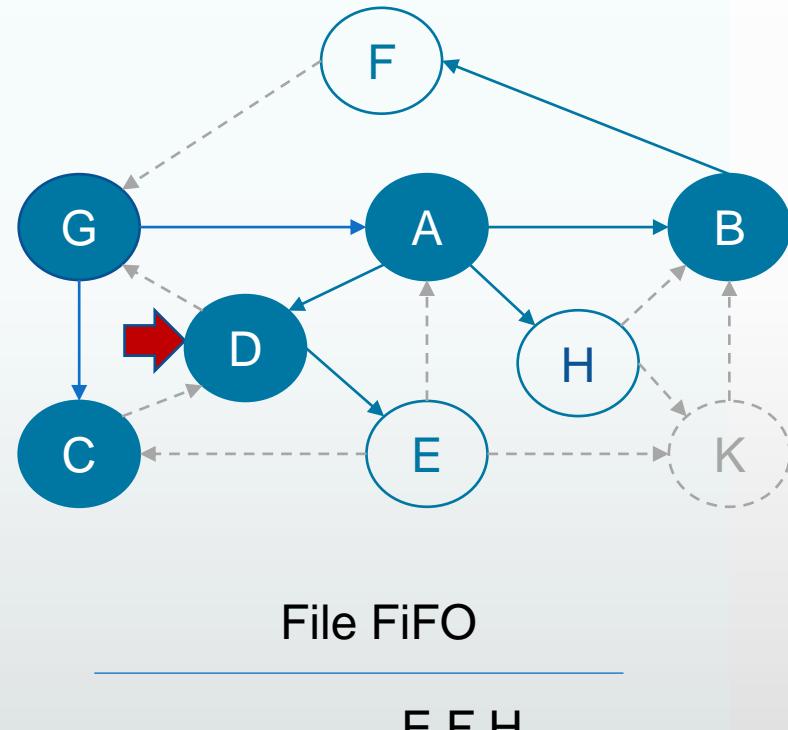
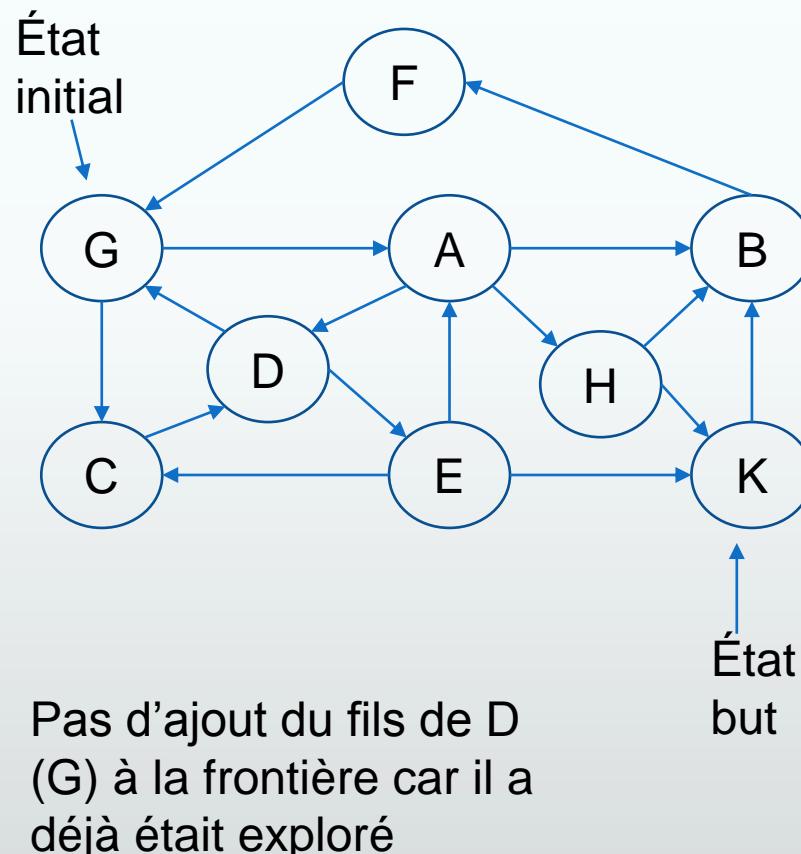
File FiFO

F H D

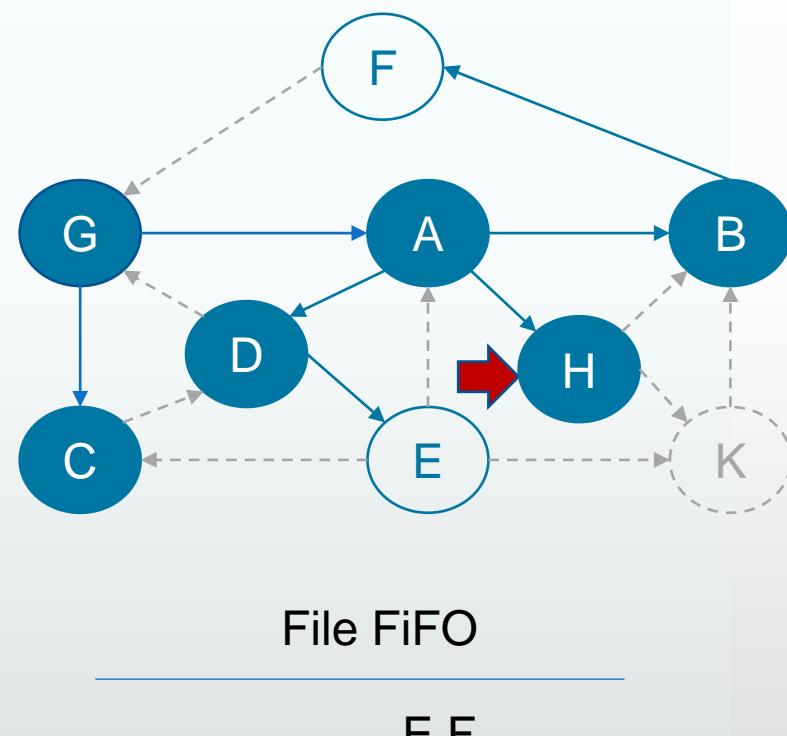
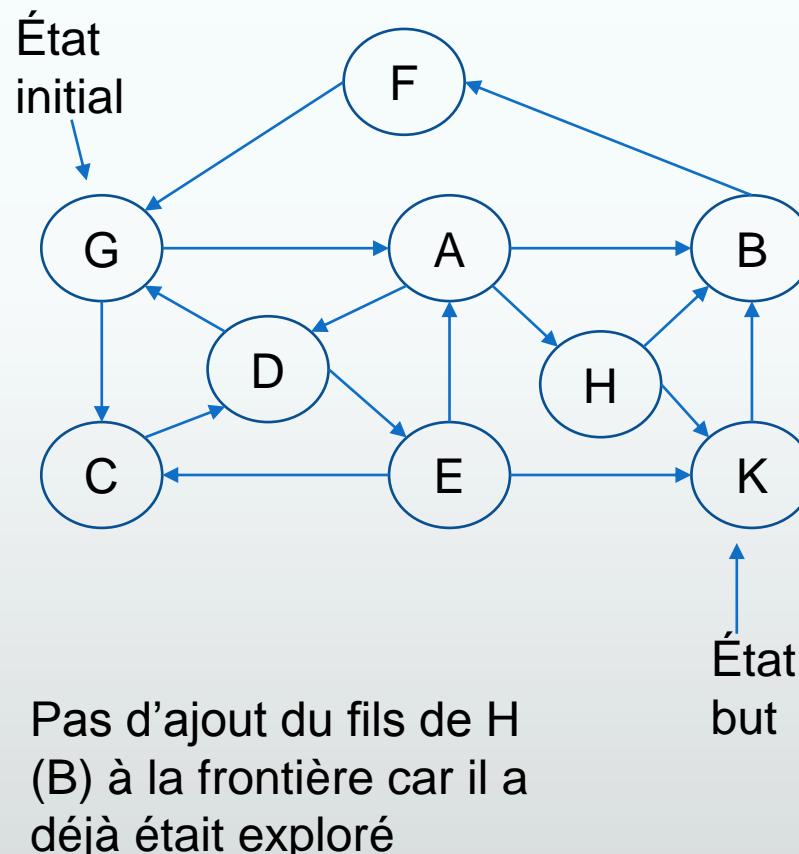
Exemple d'exploration en largeur d'abord en graphe



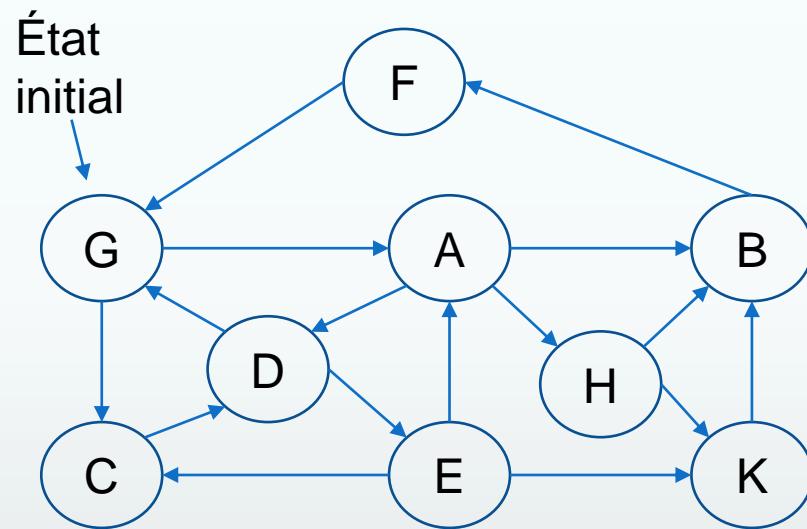
Exemple d'exploration en largeur d'abord en graphe



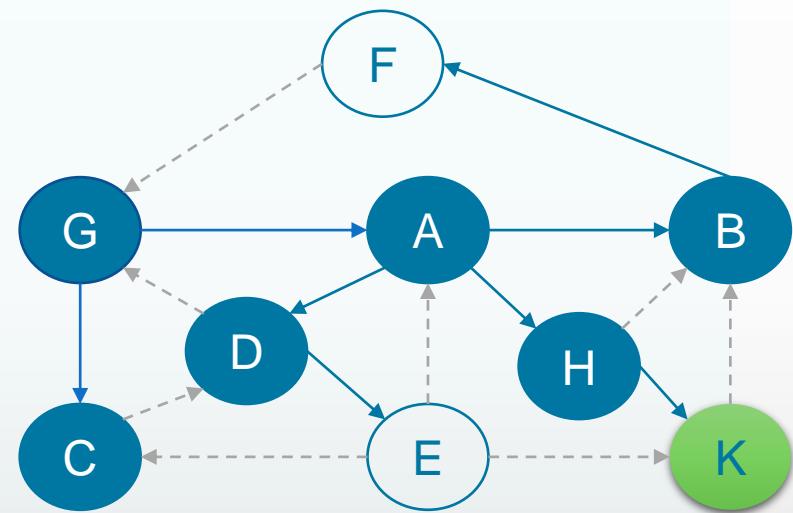
Exemple d'exploration en largeur d'abord en graphe



Exemple d'exploration en largeur d'abord en graphe



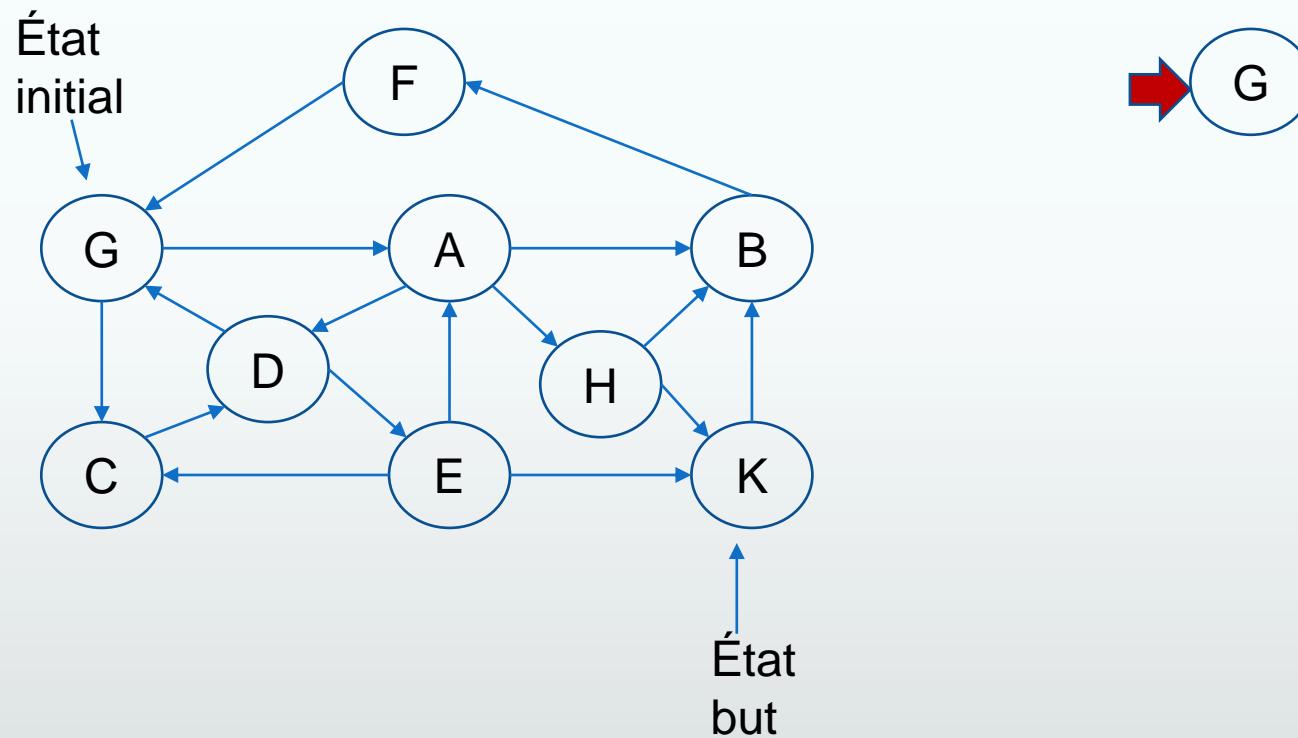
L'algorithme n'attend pas l'exploration du fils de H il effectue le test du but avant de le placer dans la frontière



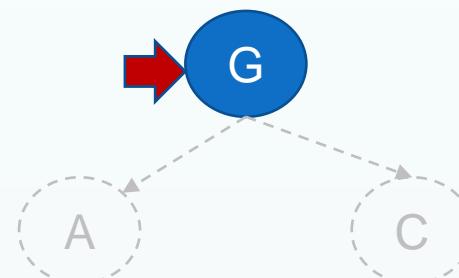
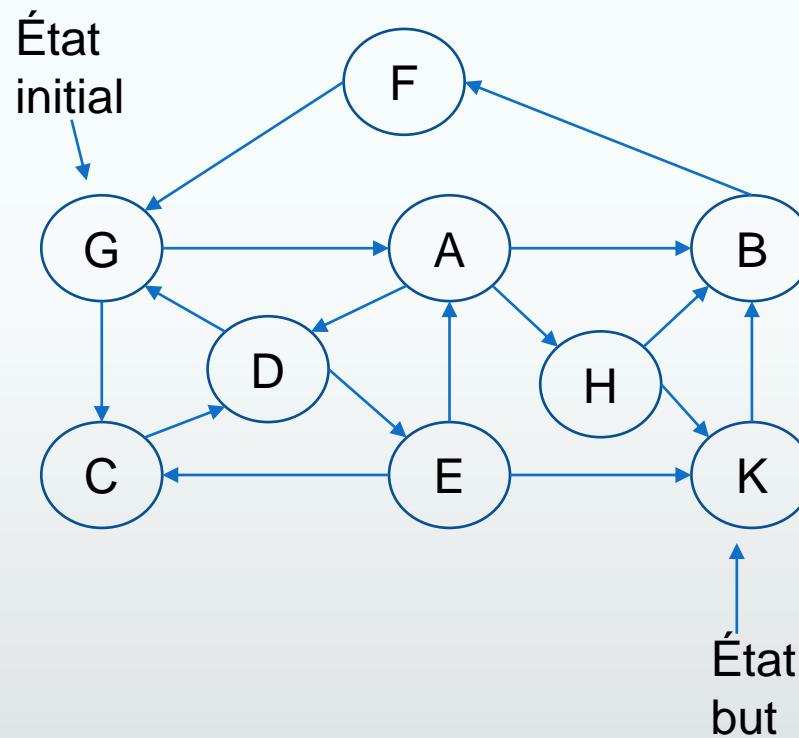
File FiFO

E F

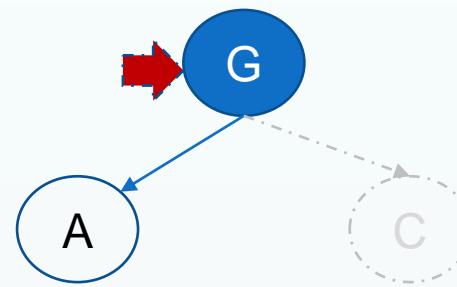
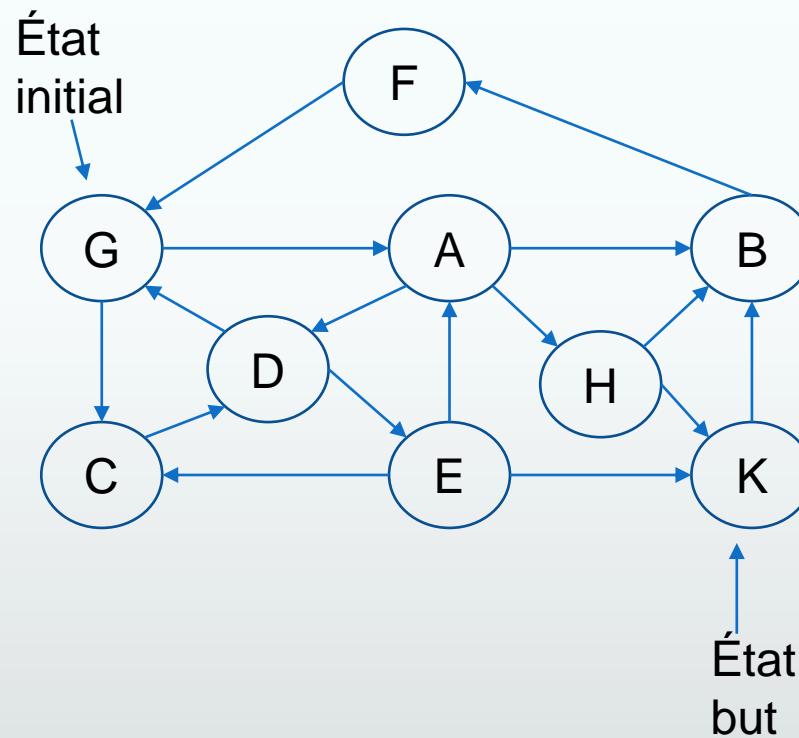
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



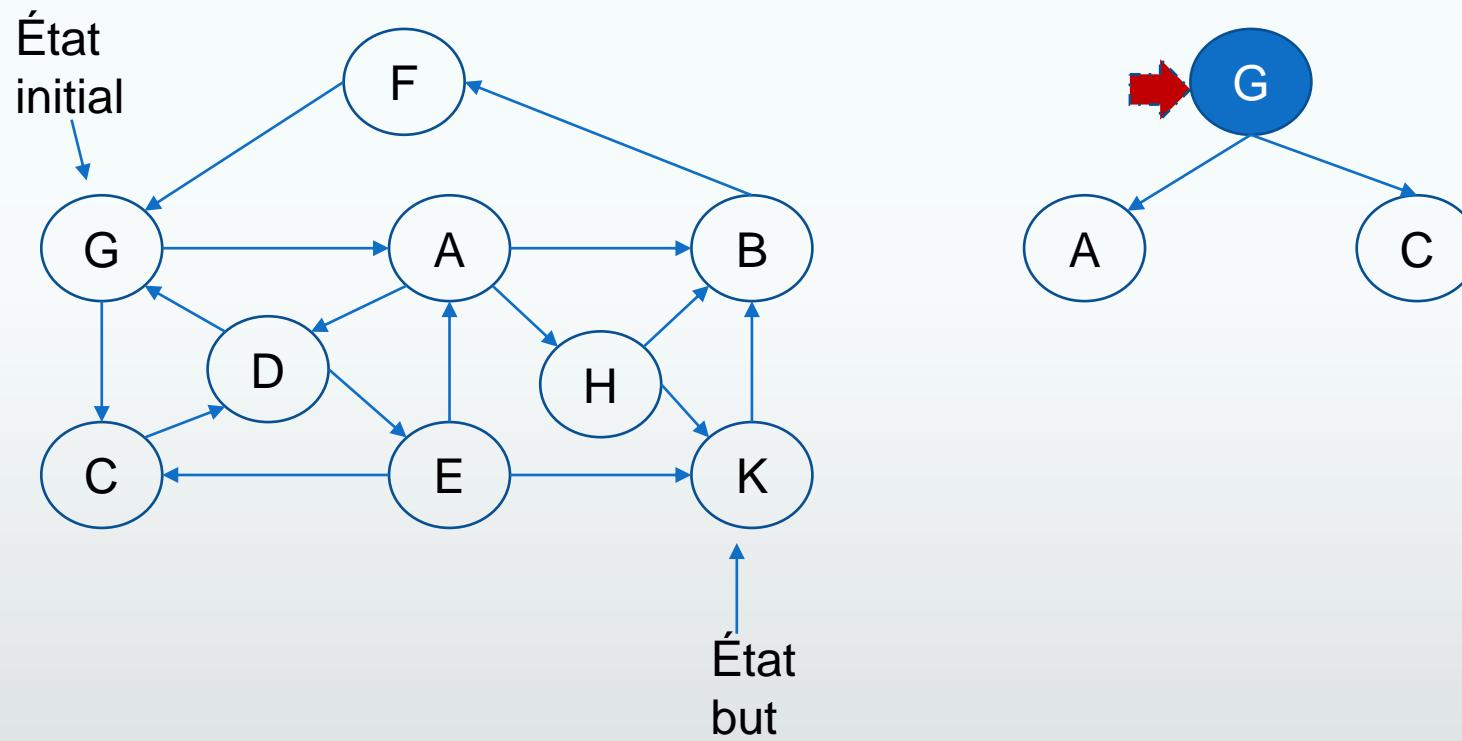
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



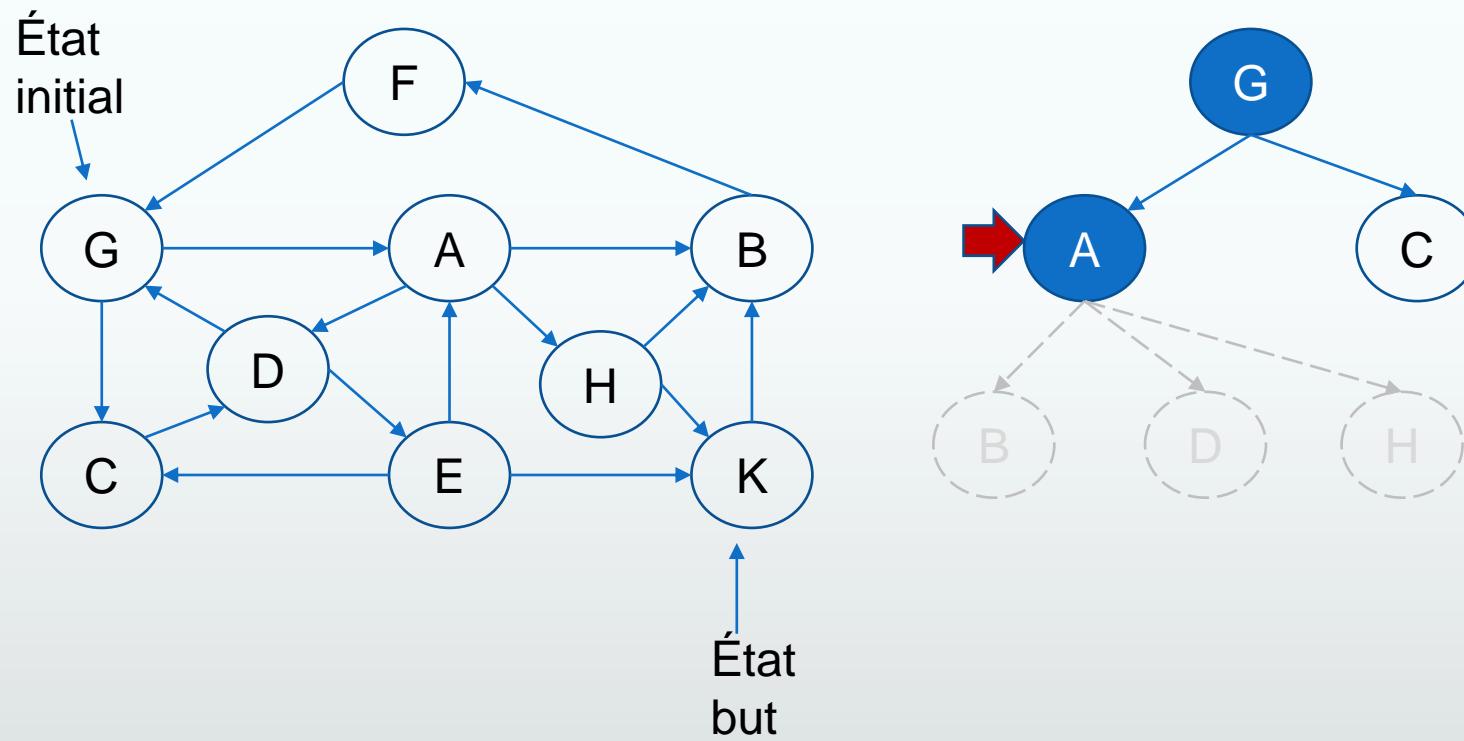
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



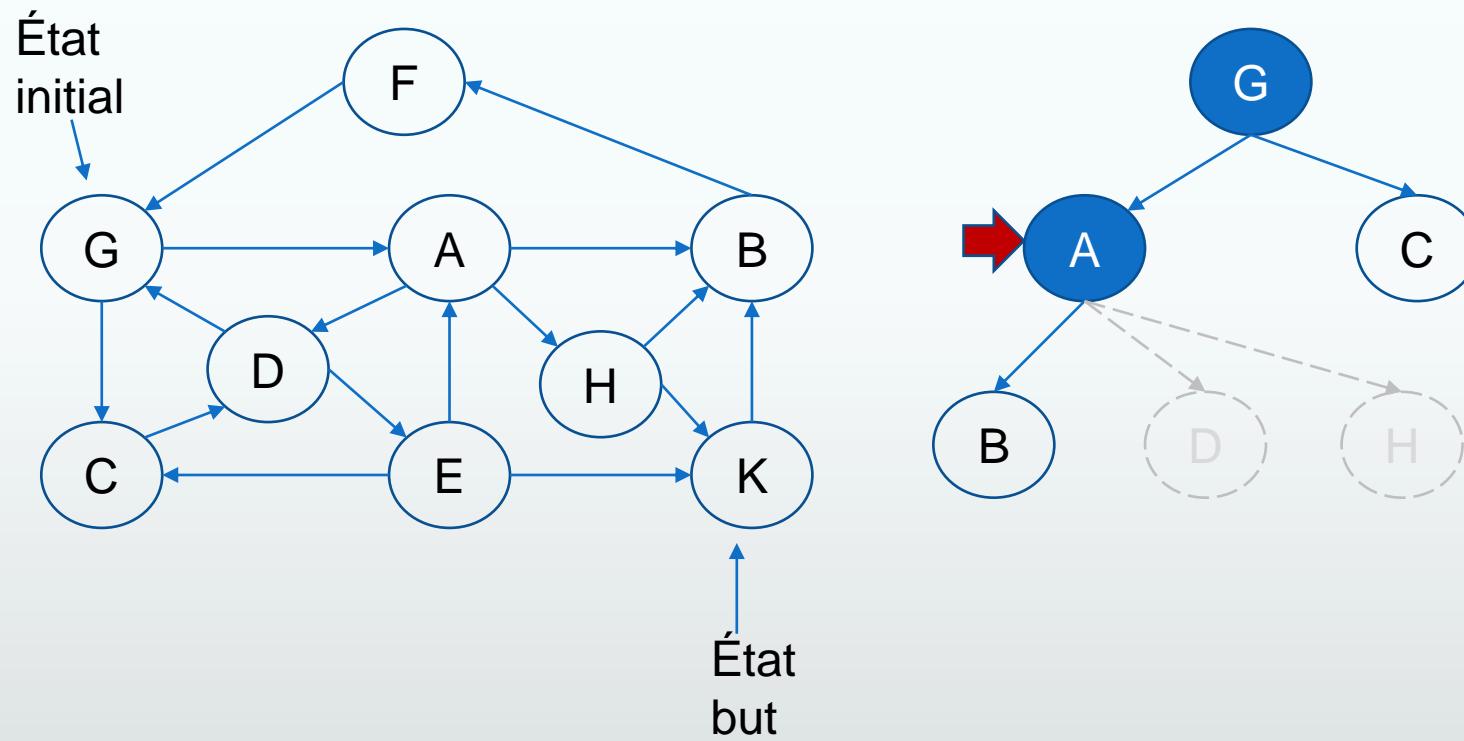
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



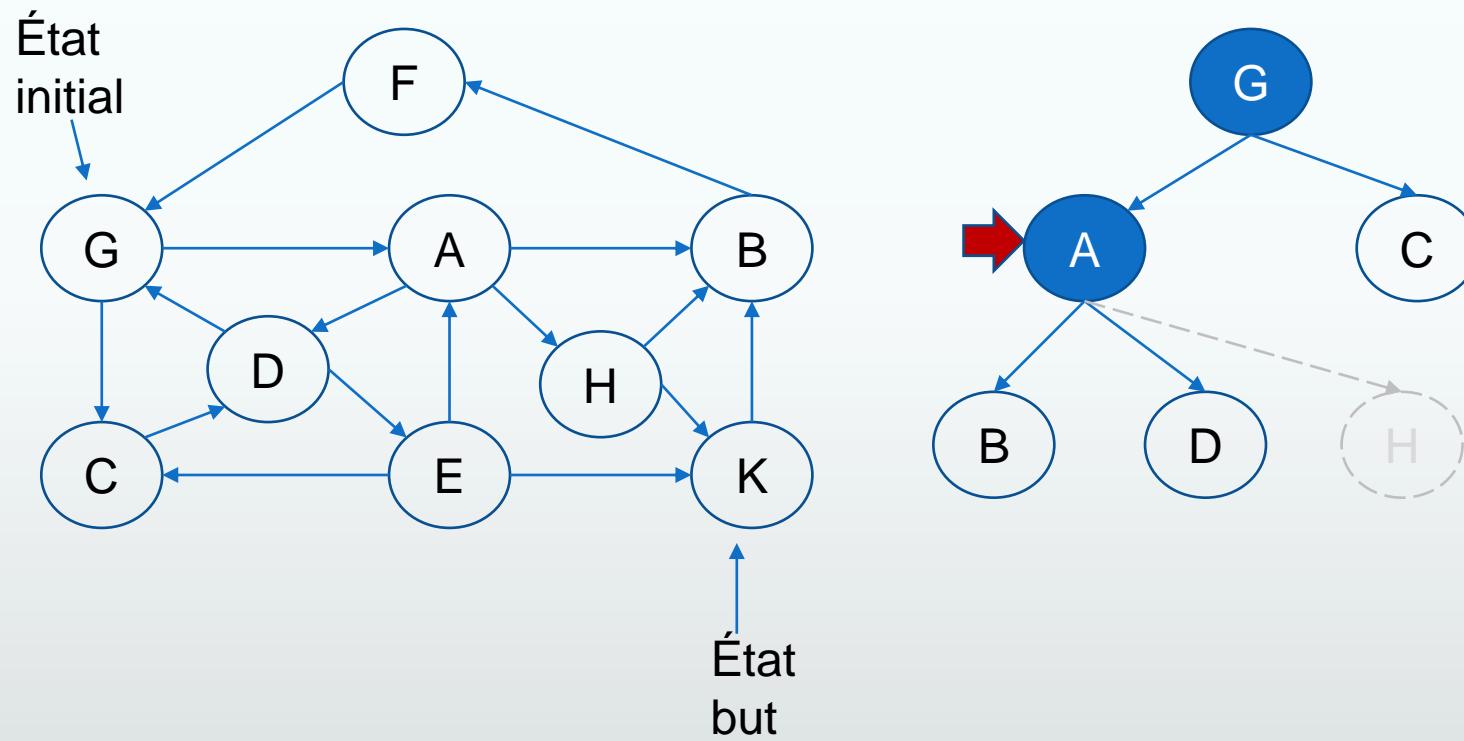
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



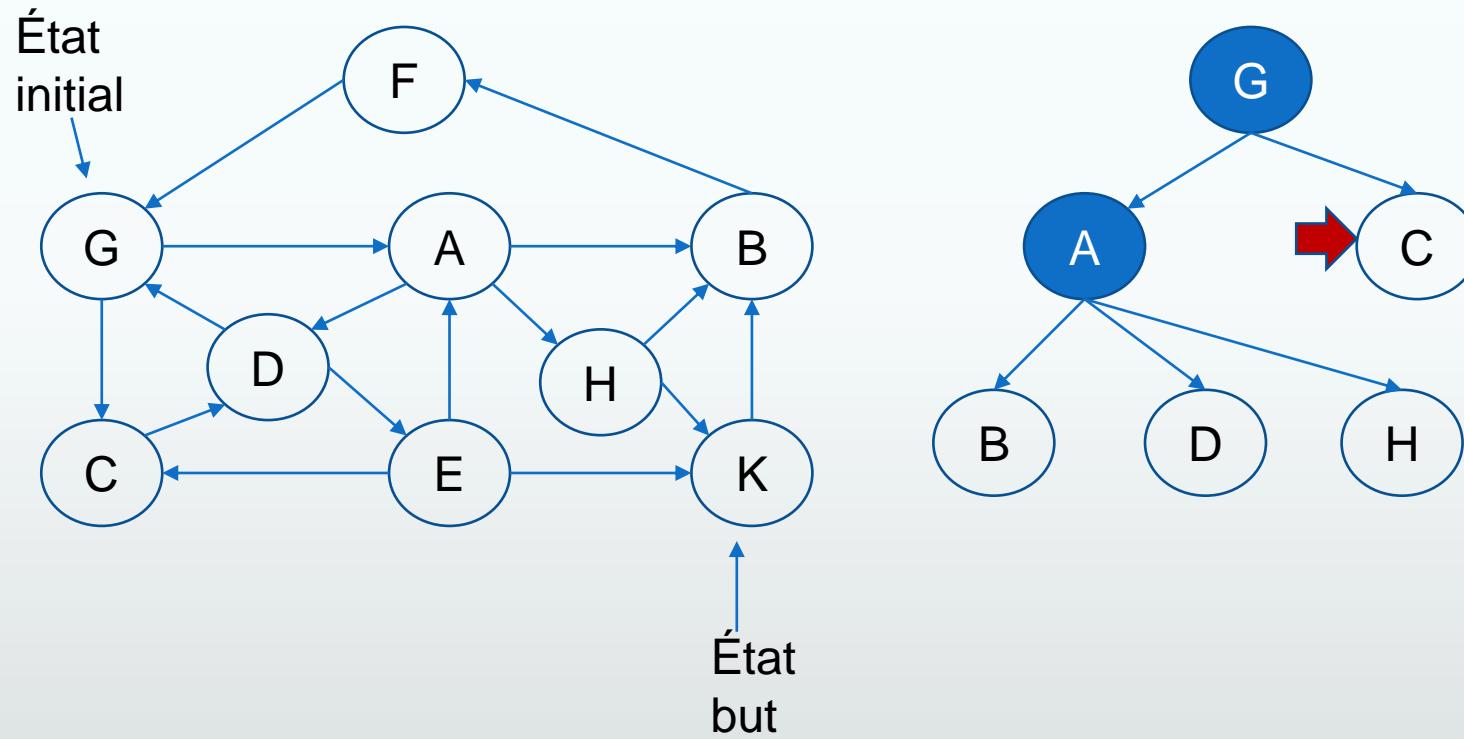
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



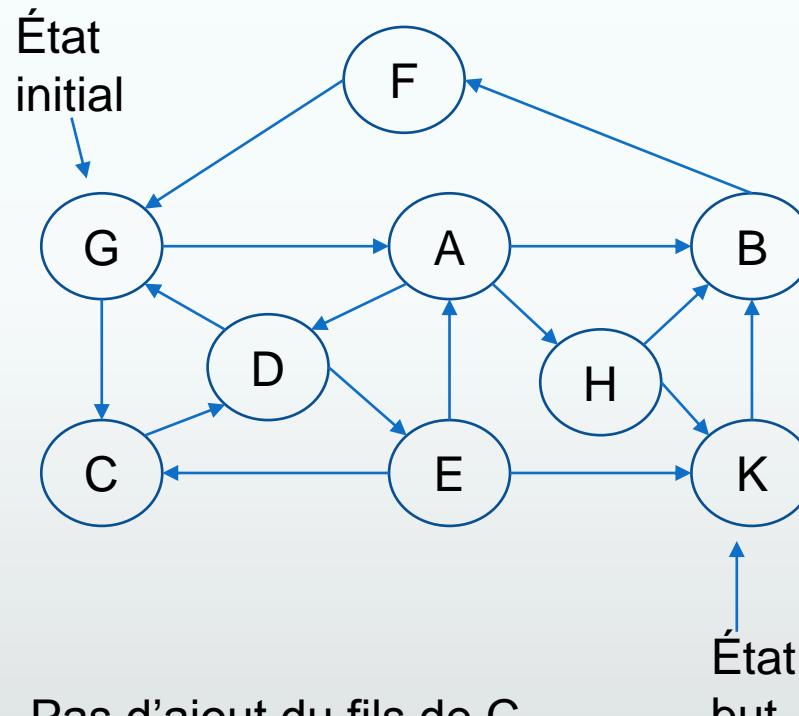
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



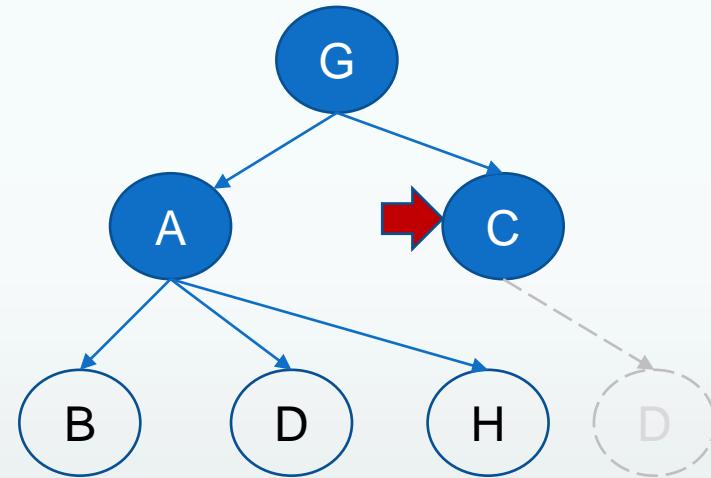
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



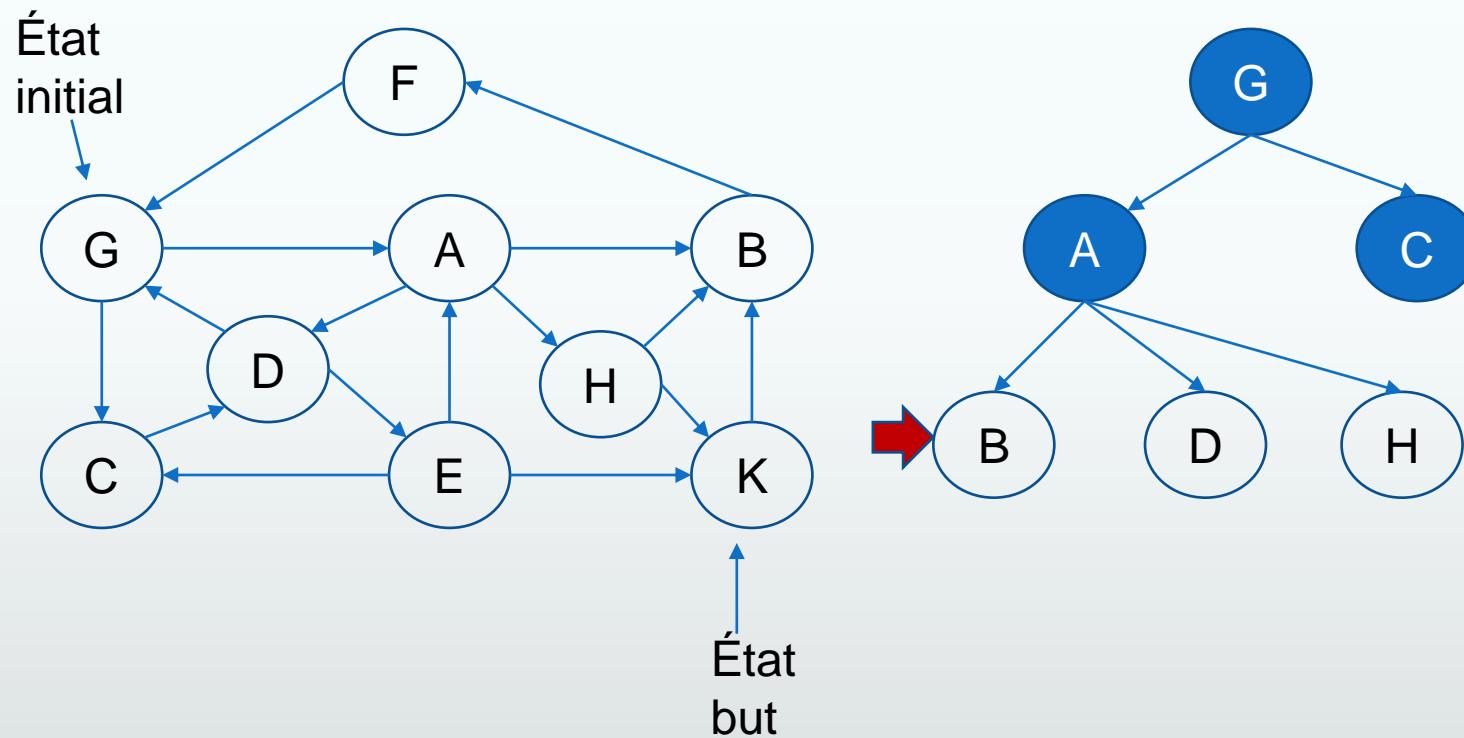
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



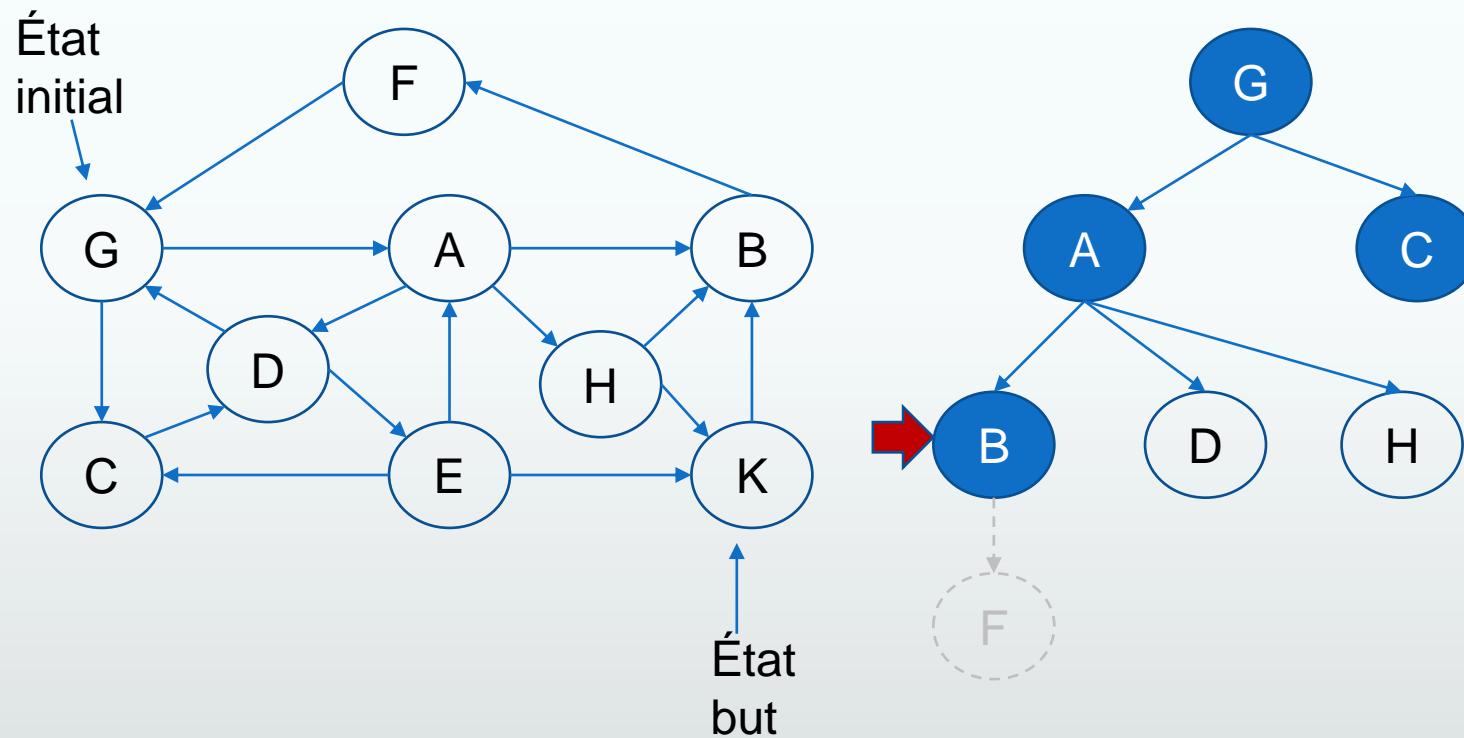
Pas d'ajout du fils de C
(D) à la frontière car il y
est déjà



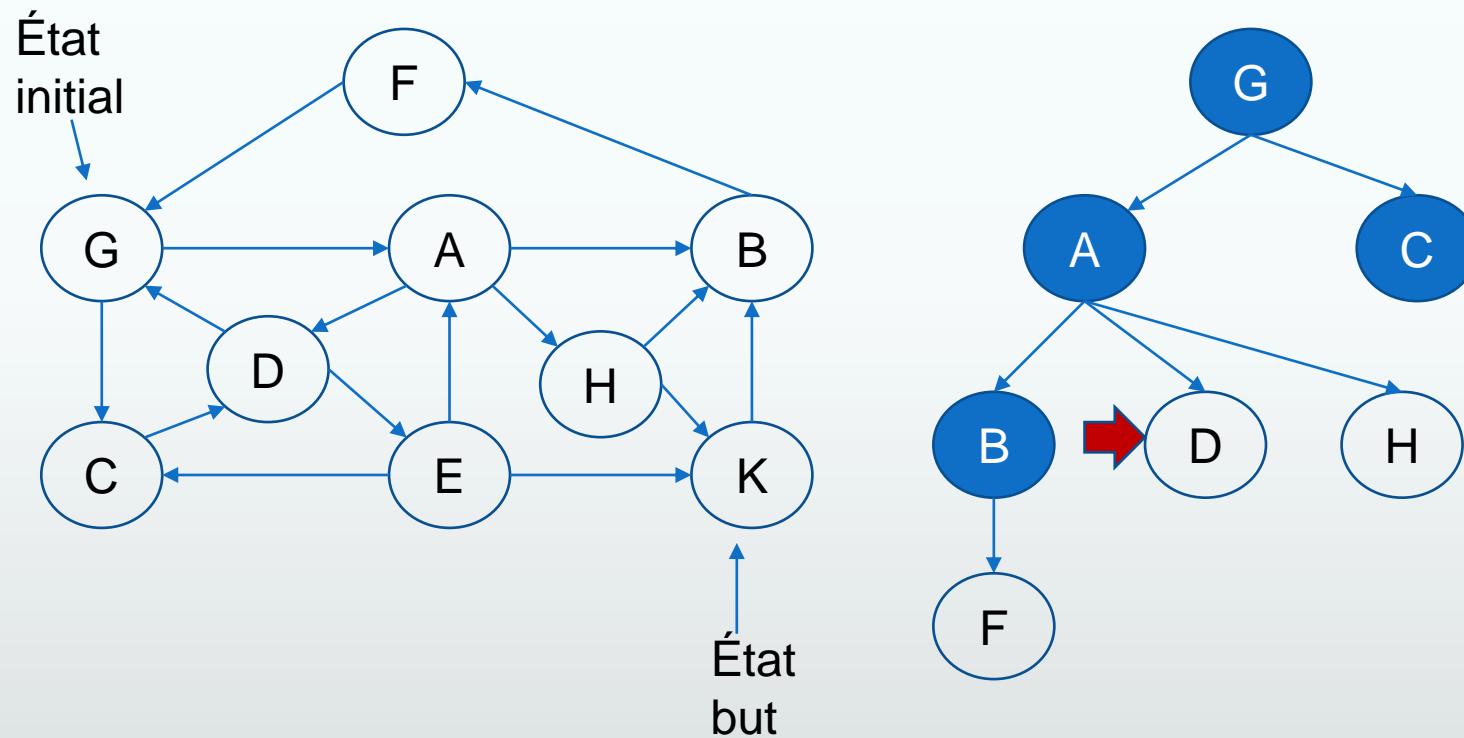
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



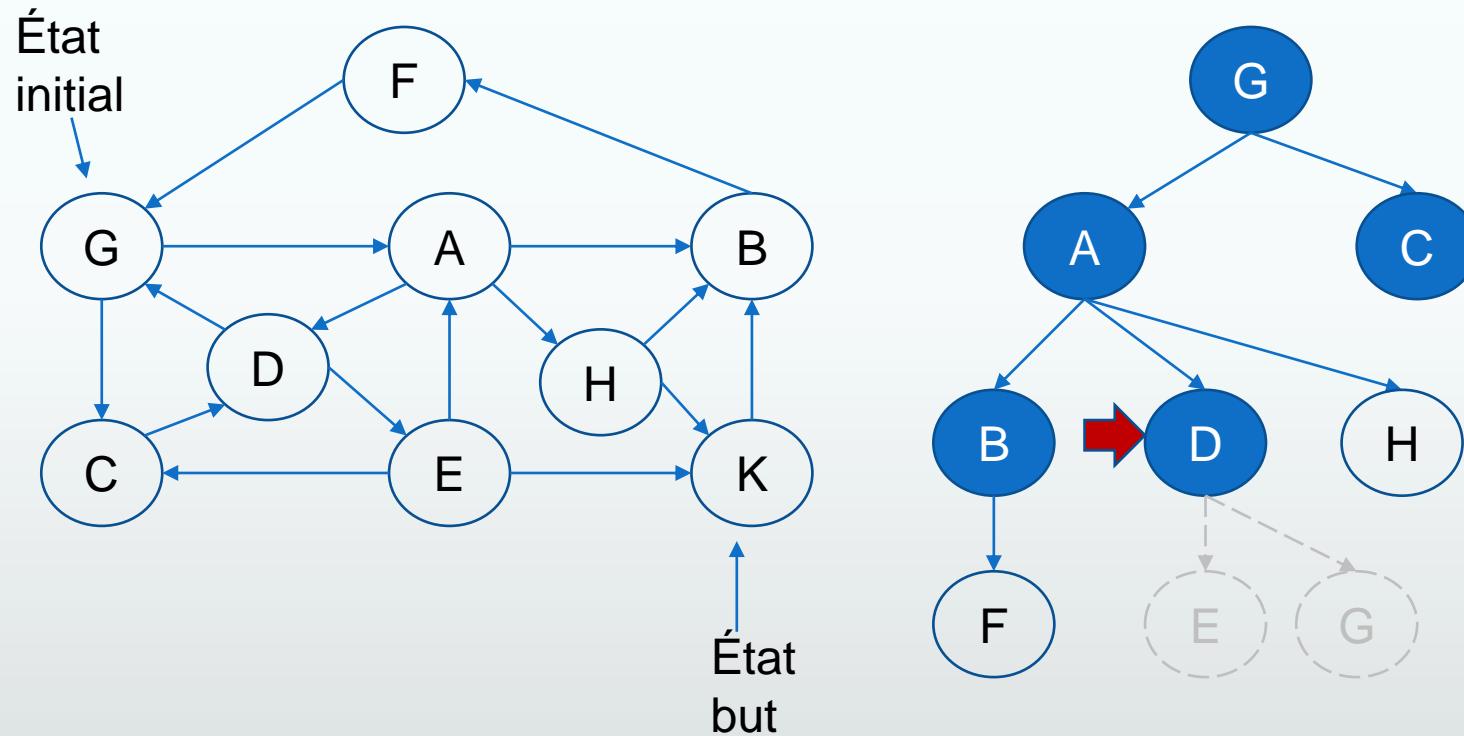
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



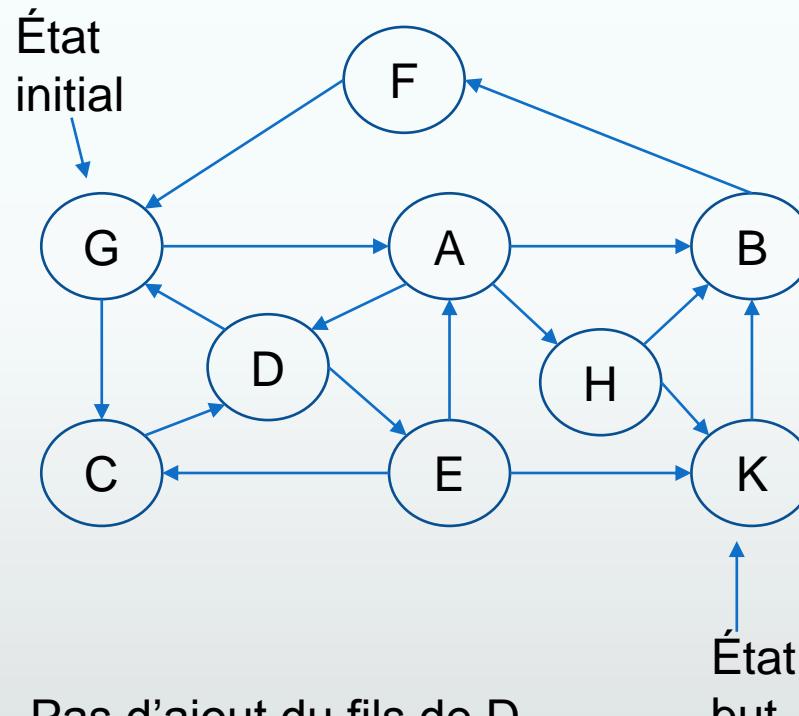
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



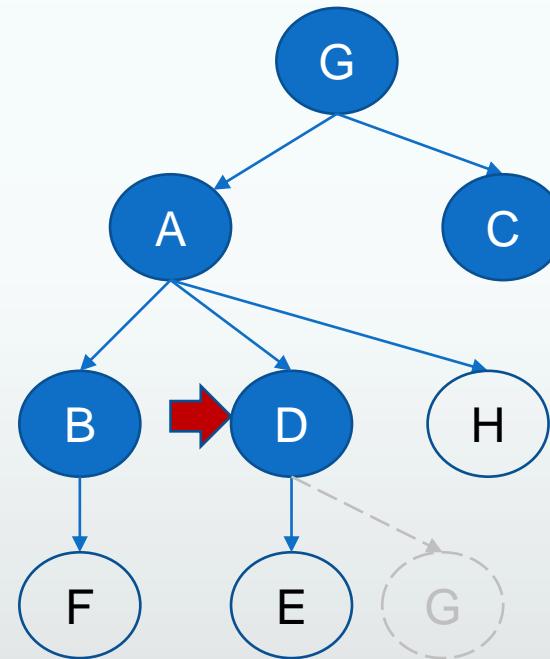
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



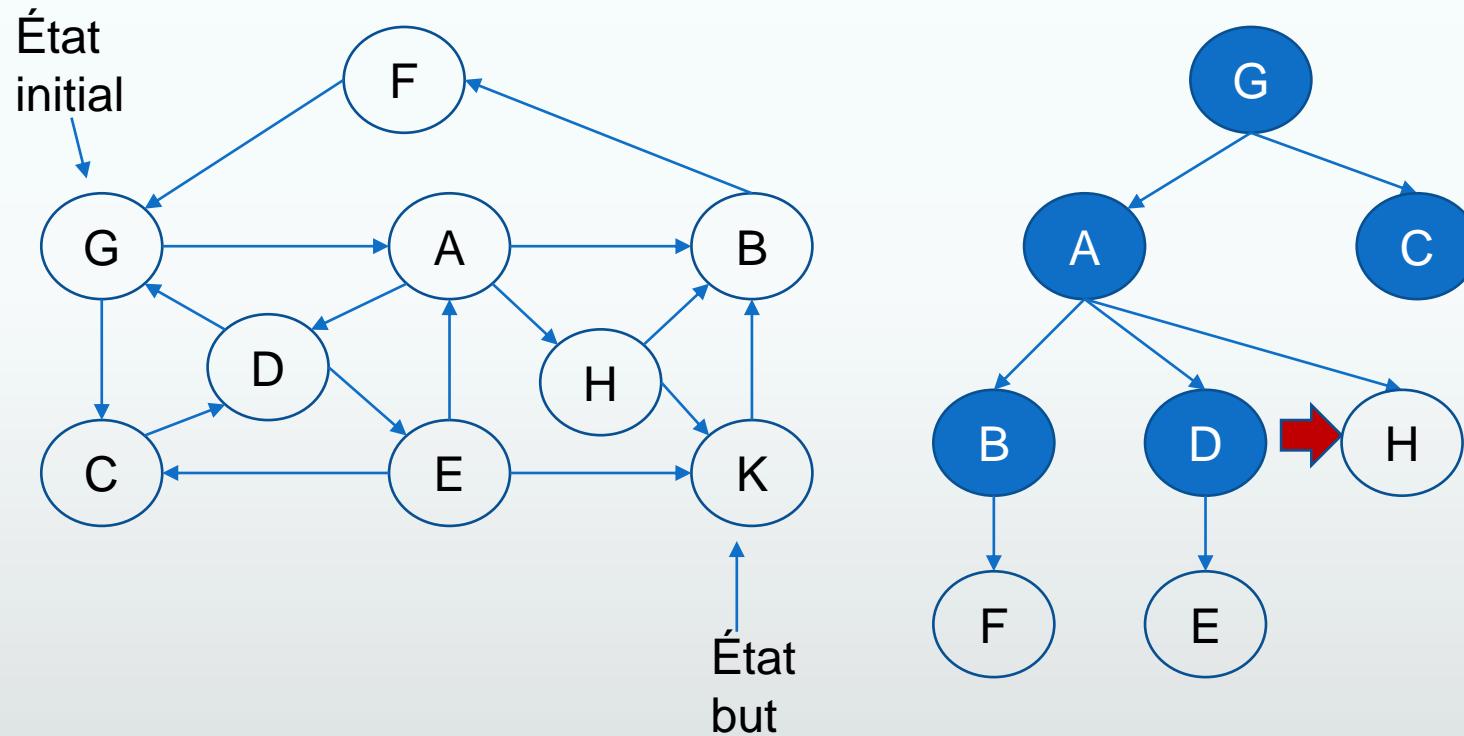
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



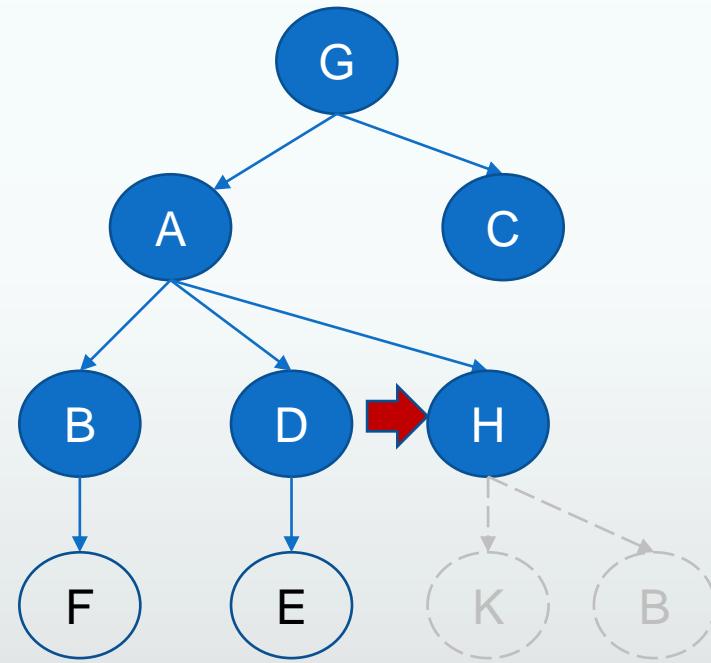
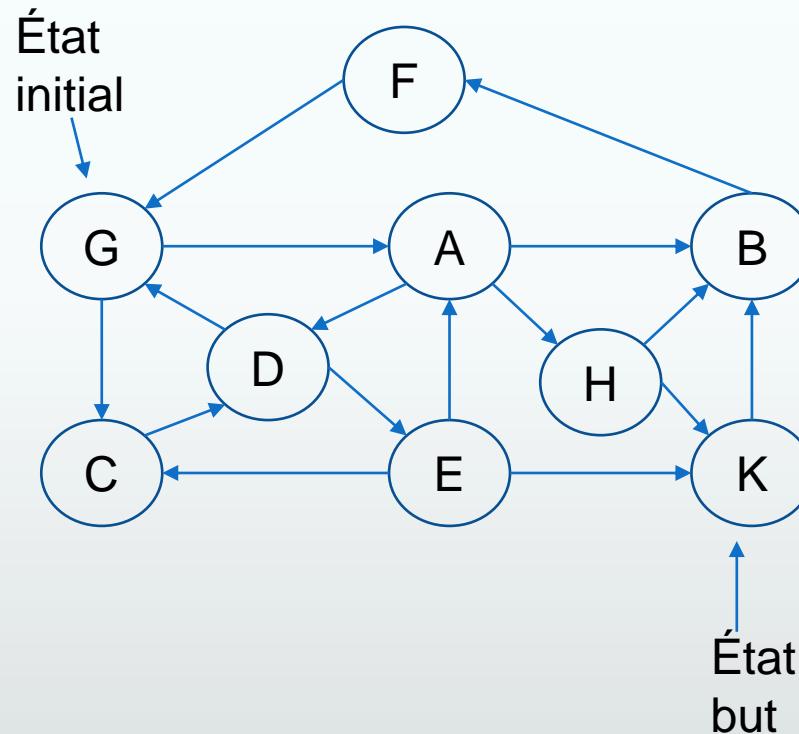
Pas d'ajout du fils de D (G) à la frontière car il a déjà été exploré



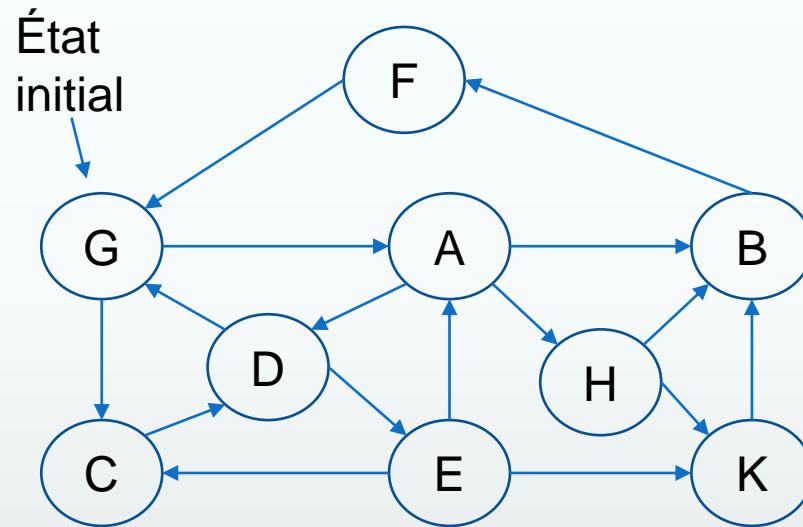
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



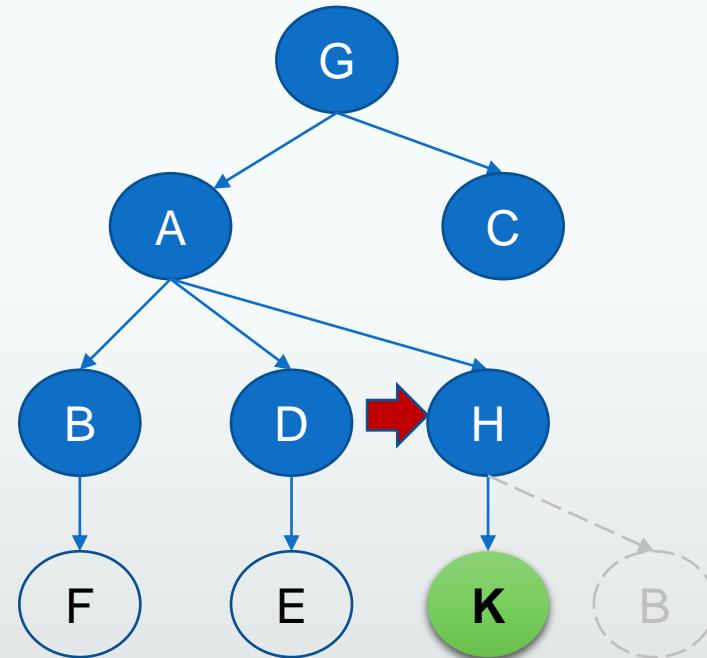
Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



Exemple d'exploration en largeur d'abord en graphe (schéma arborescent)



L'algorithme n'attend pas l'exploration du fils de H il effectue le test du but avant de le placer dans la frontière



Performance

- Complétude : **oui**, si le nœud but se trouve à une profondeur d , l'algorithme le trouvera une fois qu'il génère les nœuds des niveaux précédents, mais b doit être fini.
- Optimalité : pas nécessairement, pour la garantir il faut que le coût du chemin soit une fonction non décroissante de la profondeur du nœud (ex. coût fixe)

Performance

- Complexité temporelle :
 $1+b+b^2+b^3+\dots+b^d=O(b^d)$.
 - *Si le test de but se fait une fois le nœud choisi (comme l'algorithme général) et non une fois produit la complexité serait $O(b^{d+1})$.*
- Complexité spatiale : $O(b^d)$ puisqu'il faut garder les nœuds en mémoire ($O(b^{d-1})$ explorés et $O(b^d)$ sur la frontière).

Conclusion

- Efficace si on ne sait pas par où se trouve le but (toutes les étapes ont le même coût).
- Mais une perte de temps et d'espace si on connaît plus d'informations sur notre destination.
- N'est pas adaptée aux problèmes exponentiels :

Profondeur	Nœuds $b=10$	Temps 1 000 000 nœuds/seconde	Mémoire 1000 octets/nœud
12	10^{12}	13 jours	1 pétaoctets
16	10^{16}	350 années	10 exaoctets

Exploration à coût uniforme *(Uniform Cost Search* *UCS)*

Principe de l'exploration à coût uniforme

- ❑ Développement du nœud qui a le coût de chemin le plus faible $g(n)$:
 - ❑ $g(n)$ =coût à partir du nœud initial jusqu'au nœud développé.
- ❑ La frontière est stockée dans une file triée par coût (g) croissant (file de priorité).
- ❑ Le test du but est appliqué à un nœud une fois choisi.
- ❑ Un test vérifie si un meilleur chemin existe vers un nœud sur la frontière (table de hachage pour l'implémentation).
- ❑ Équivalent a l'exploration en largeur d'abord si tous les coûts des actions sont égaux.

Algorithme d'exploration à coût uniforme

fonction EXPLORATION-COÛT-UNIFORME(*problème*) **retourne** une solution, ou échec

nœud \leftarrow un nœud avec ÉTAT = *problème*.ÉTAT-INITIAL, COÛT-CHEMIN = 0

frontière \leftarrow une file de priorité ordonnée par COÛT-CHEMIN, avec *nœud* comme seul élément

exploré \leftarrow un ensemble vide

faire en boucle

si VIDE?(*frontière*) **alors retourner** échec

nœud \leftarrow POP(*frontière*) /* choisit le nœud de la *frontière* avec le coût le plus faible */

si *problème*.TEST-BUT(*nœud*.ÉTAT) **alors retourner** SOLUTION(*nœud*)

 ajouter *nœud*.ÉTAT à *exploré*

pour chaque *action* dans *problème*.ACTIONS(*nœud*.ÉTAT) **faire**

fils \leftarrow NŒUD-FILS(*problème*, *nœud*, *action*)

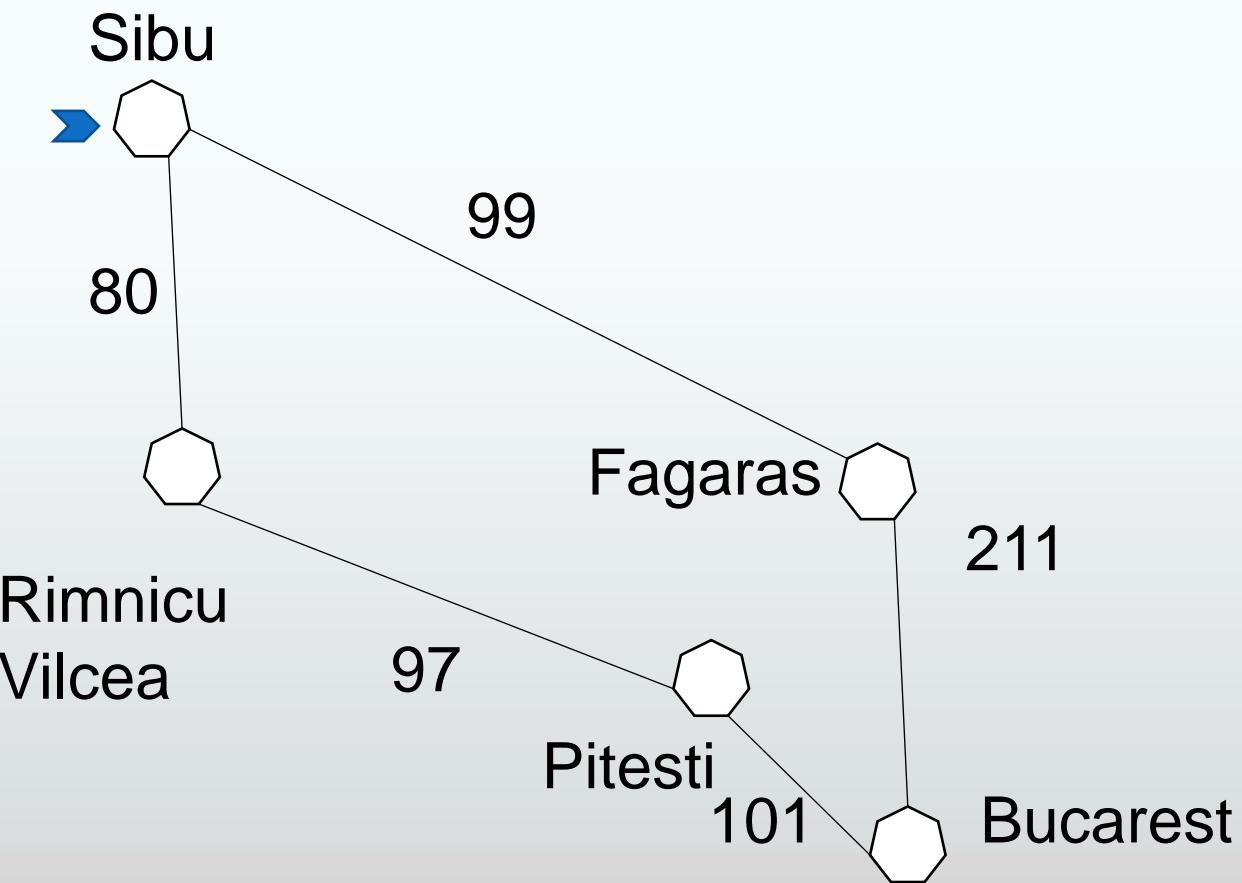
si *fils*.ÉTAT n'est pas dans *exploré* ou dans *frontière* **alors**

frontière \leftarrow INSÉRER(*fils*, *frontière*)

sinon si *fils*.ÉTAT est dans *frontière* avec un COÛT-CHEMIN plus élevé **alors**

 remplacer ce nœud de *frontière* avec *fils*

Exemple



Exemple

Sibiu
➡ 

File (frontière)	Nœud développé
[(Sibiu,0)]	

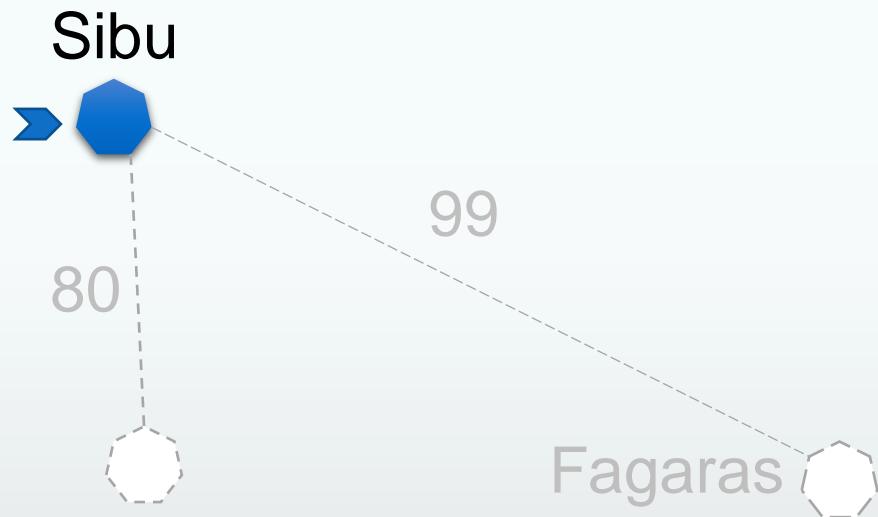
Exemple

Sibiu
» 

File (frontière)	Nœud développé
[(Sibiu,0)]	

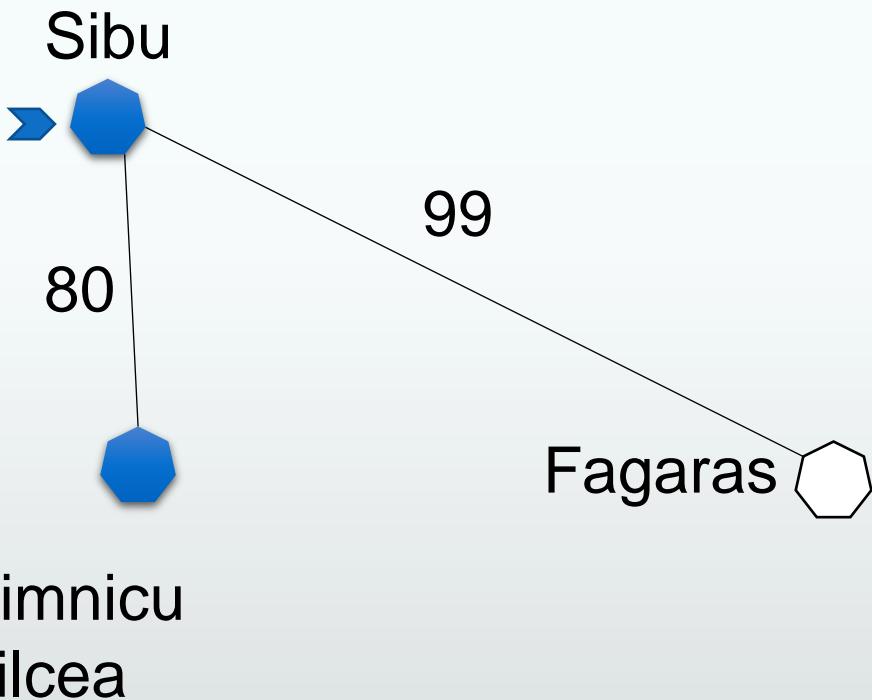
Exemple

File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)



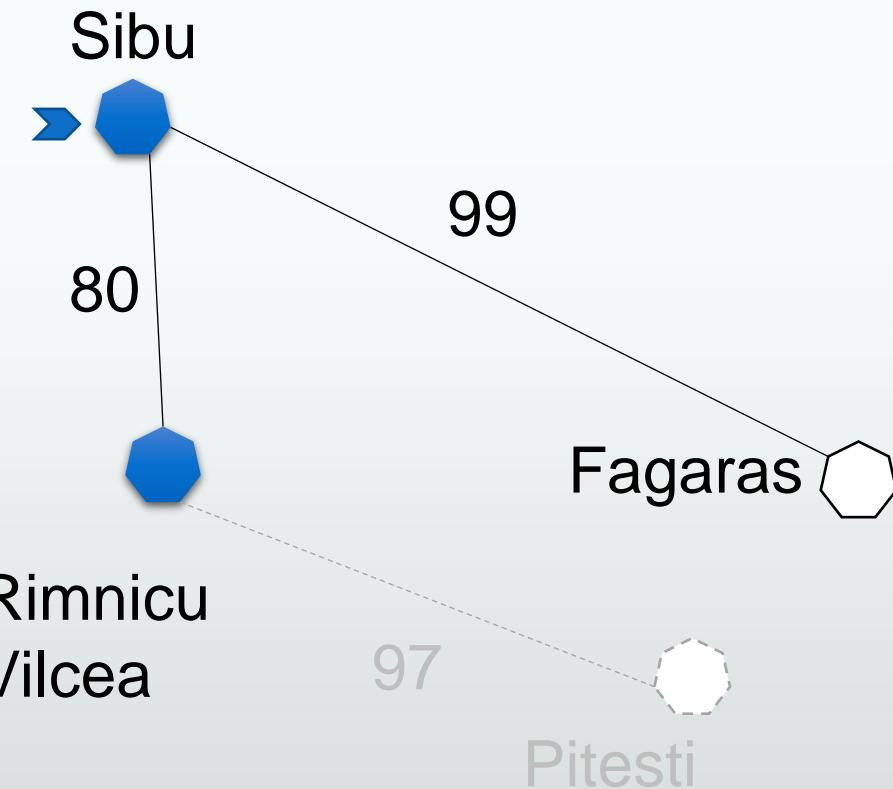
Rimnicu
Vilcea

Exemple



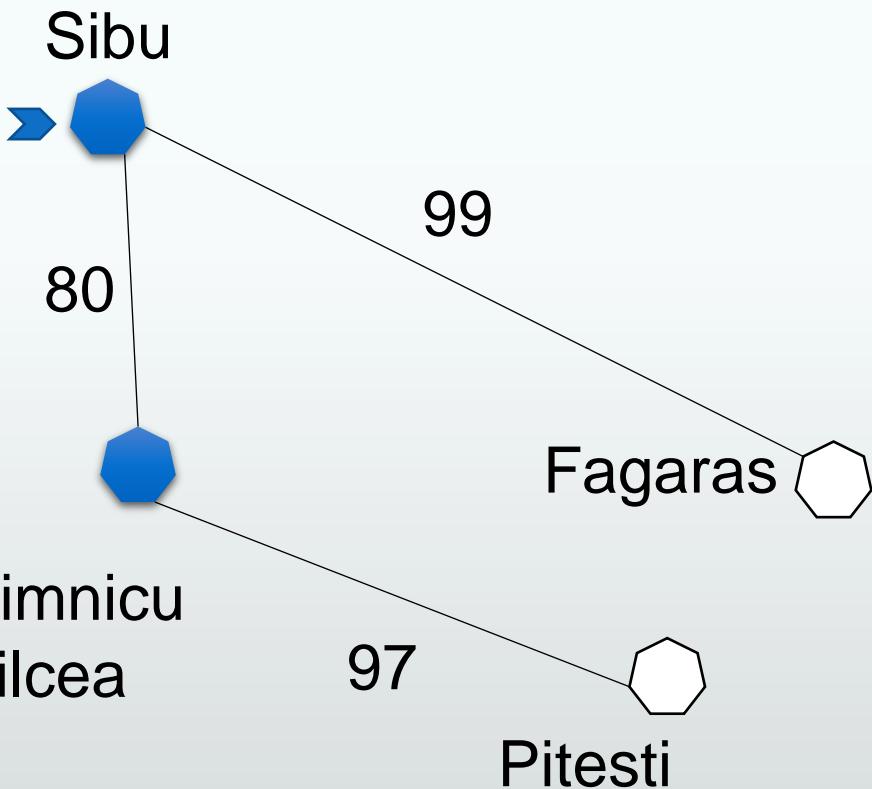
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	

Exemple



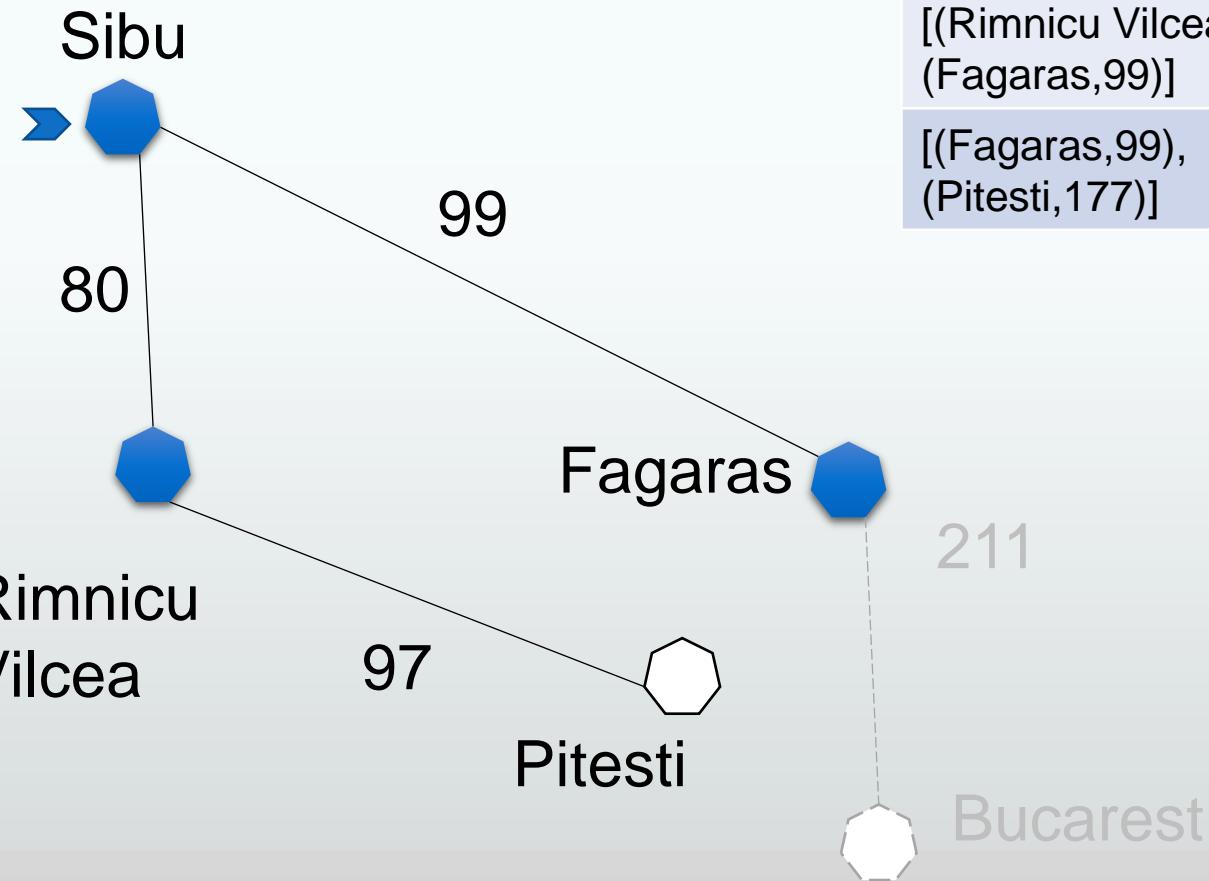
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)

Exemple



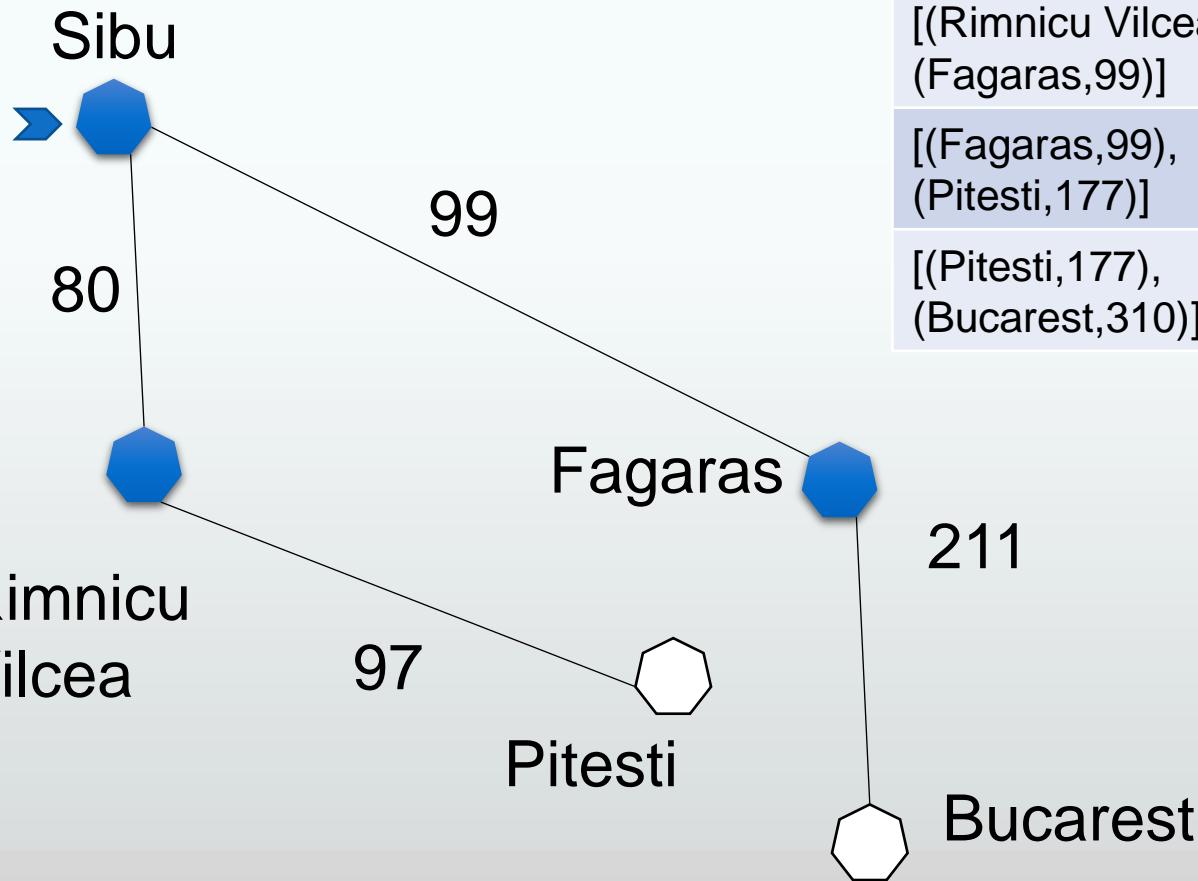
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	

Exemple



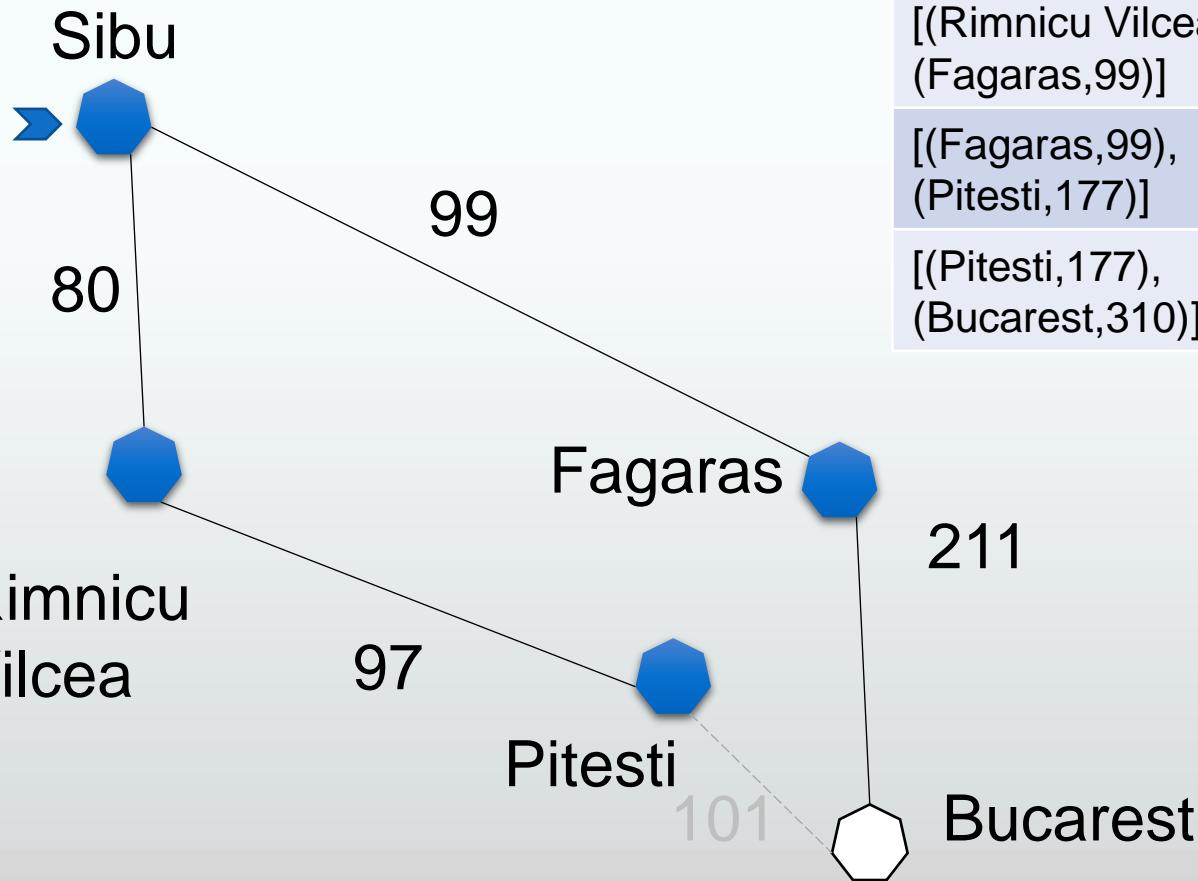
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	(Fagaras,99)

Exemple



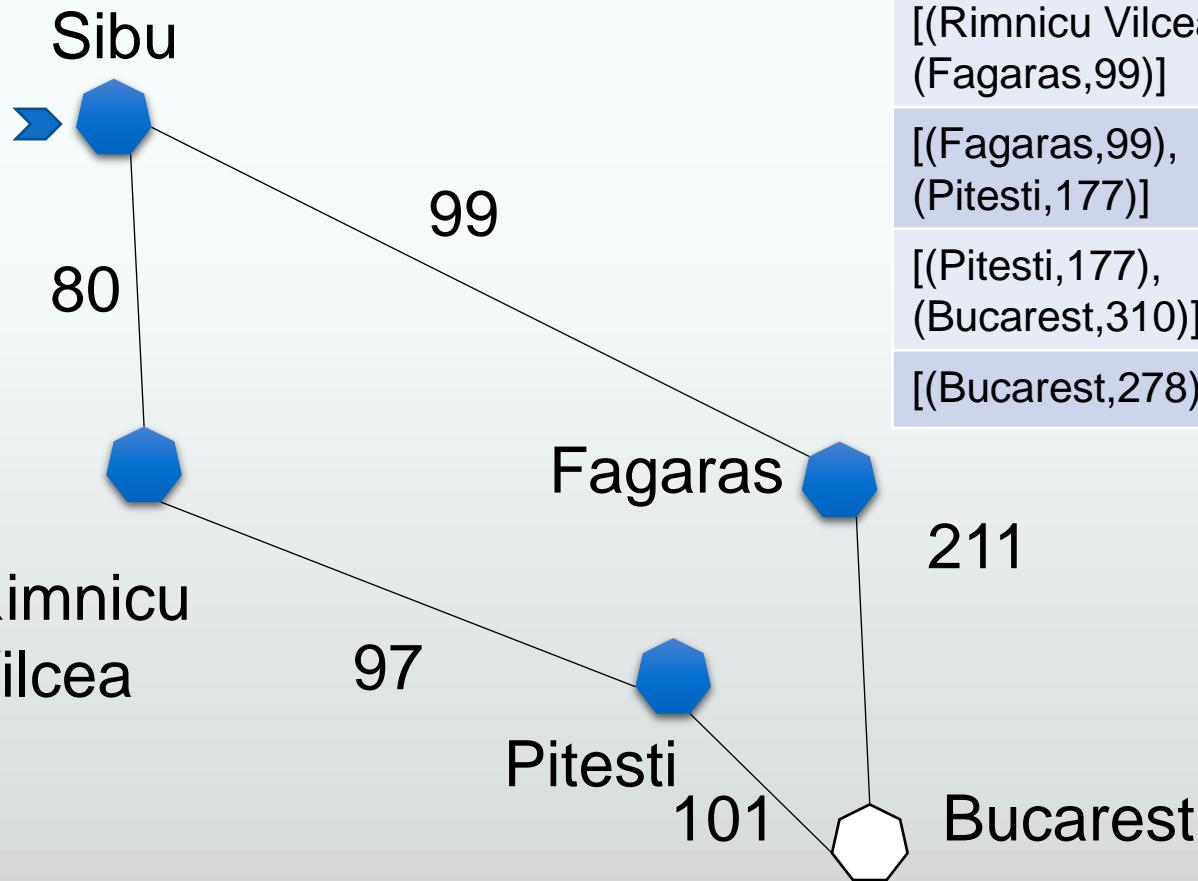
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	(Fagaras,99)
[(Pitesti,177), (Bucarest,310)]	

Exemple



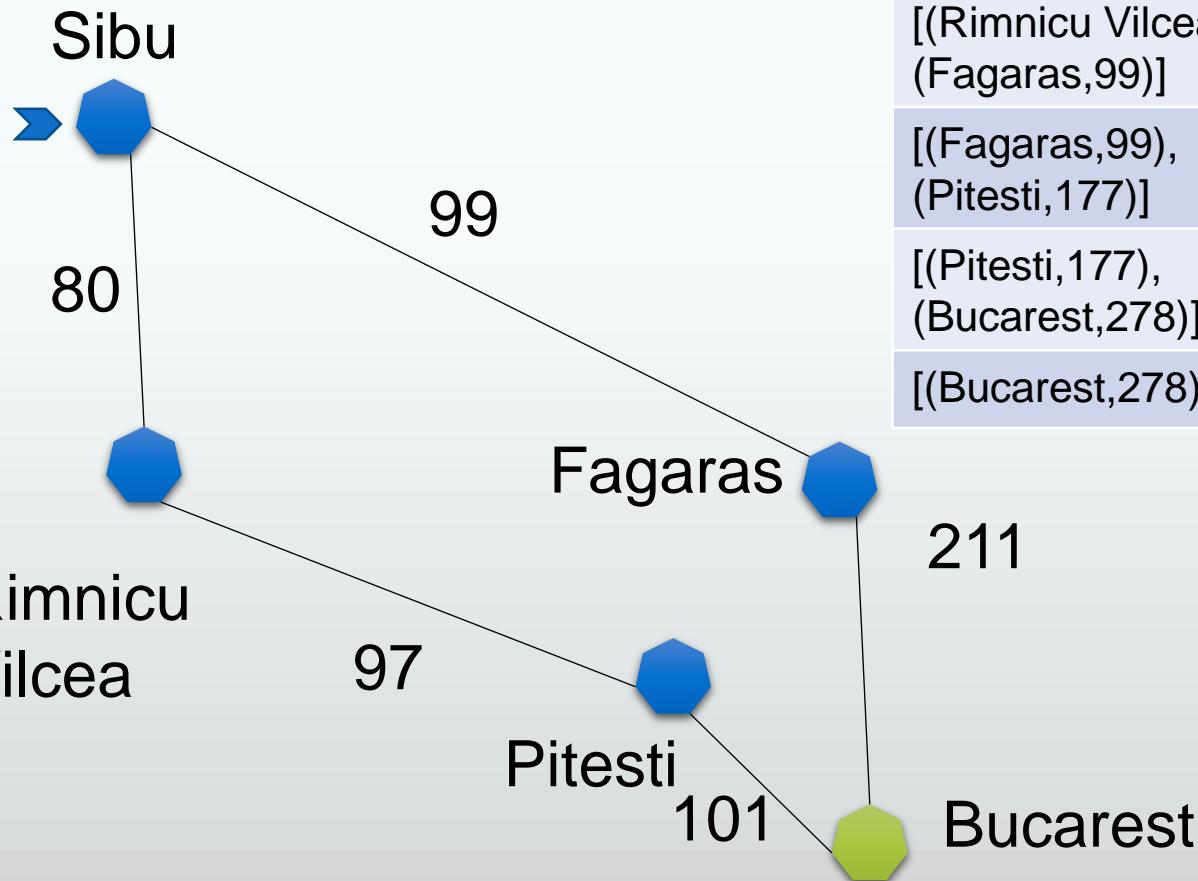
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	(Fagaras,99)
[(Pitesti,177), (Bucarest,310)]	(Pitesti,177)

Exemple



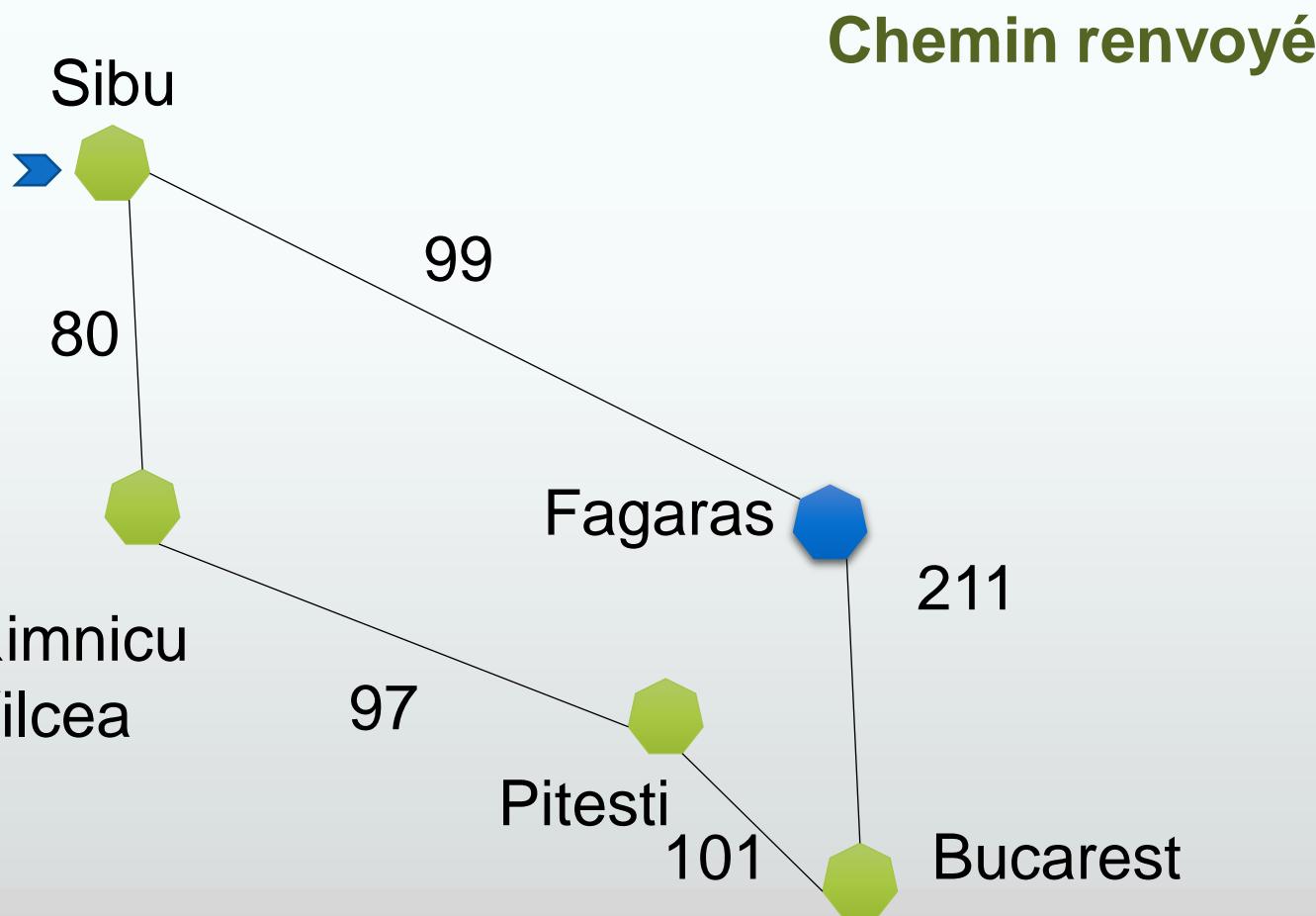
File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	(Fagaras,99)
[(Pitesti,177), (Bucarest,310)]	(Pitesti,177)
[(Bucarest,278)]	

Exemple



File (frontière)	Nœud développé
[(Sibiu,0)]	(Sibiu,0)
[(Rimnicu Vilcea,80), (Fagaras,99)]	(Rimnicu Vilcea,80)
[(Fagaras,99), (Pitesti,177)]	(Fagaras,99)
[(Pitesti,177), (Bucarest,278)]	(Pitesti,177)
[(Bucarest,278)]	(Bucarest,278)

Exemple



Performance de l'exploration à coût uniforme

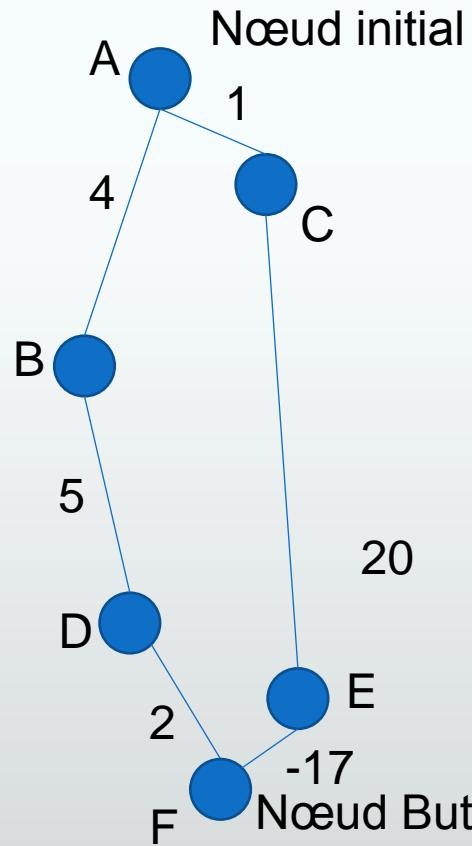
- Complétude : oui, si le coût de chaque étape \geq à une constante positive ϵ (éviter les chemins infinis d'actions à coût nul).
- Optimalité : oui, les nœuds sont développés dans l'ordre de coût de chemin optimal (mais les coûts ne doivent pas être négatifs).
- Complexité en temps et en espace :
 - Soit C^* le coût de la solution optimale.
 - Soit ϵ le coût minimal de chaque action.
 - Dans le pire des cas: $O(b^{1+C^*/\epsilon})$ peut être $>b^d$ (peut explorer des petits chemins avant d'explorer des chemins plus coûteux mais plus intéressants) .
 - Egal à $O(b^{d+1})$ si les coûts des actions sont égaux.

Remarque sur l'optimalité

- ❑ L'optimalité est garantie car l'algorithme développe les nœuds par ordre de coût de chemin optimal :
 - ❑ Tout nœud choisi pour le développement est atteint par un chemin optimal, car sinon un autre nœud sur la frontière aurait été choisi.
 - ❑ Le coût des actions n'est jamais < 0 , donc les chemins ne deviennent jamais plus court lorsque des nœuds sont ajoutés.

Et si les coûts étaient négatifs ?

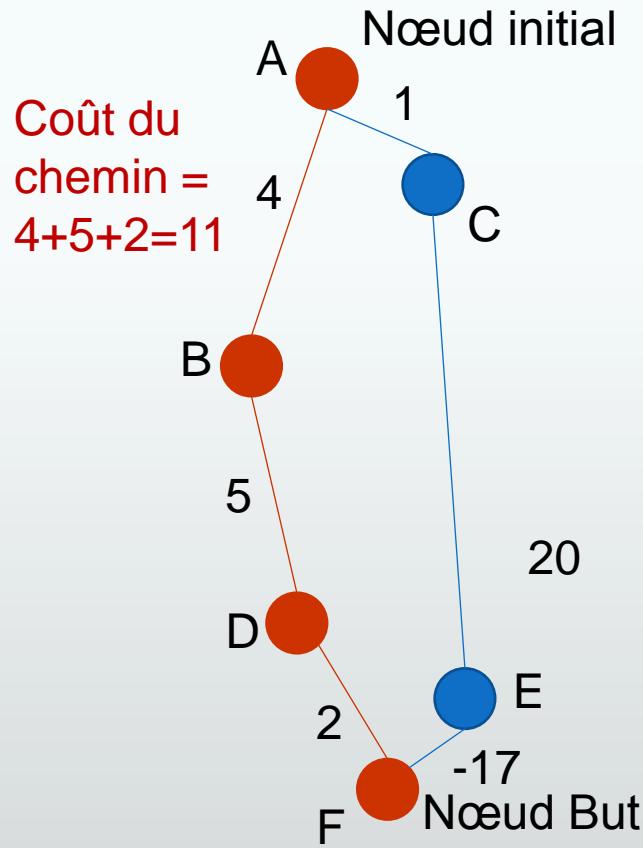
❑ Développement :



File (frontière)	Nœud développé
[(A,0)]	(A,0)
[(C,1), (B,4)]	(C,1)
[(B,4),(E,21)]	(B,4)
[(D,9),(E,21)]	(D,9)
[(F,11),(E,21)]	(F,11) test du but réussi

Et si les coûts étaient négatifs ?

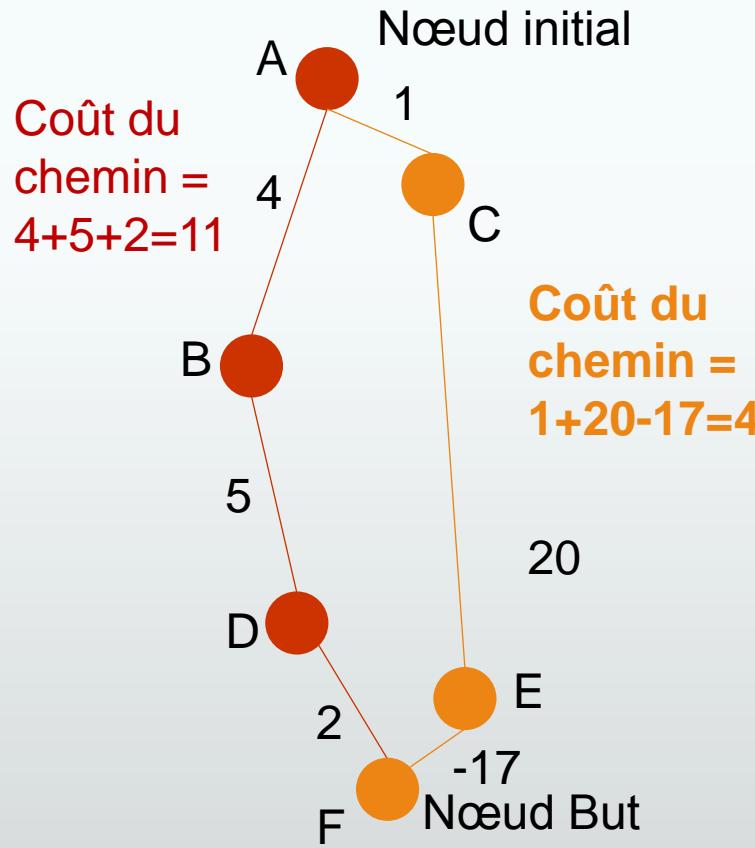
❑ Solution retournée



File (frontière)	Nœud développé
[(A,0)]	(A,0)
[(C,1), (B,4)]	(C,1)
[(B,4),(E,21)]	(B,4)
[(D,9),(E,21)]	(D,9)
[(F,11),(E,21)]	(F,11) test du but réussi

Et si les coûts étaient négatifs ?

❑ Mais ...



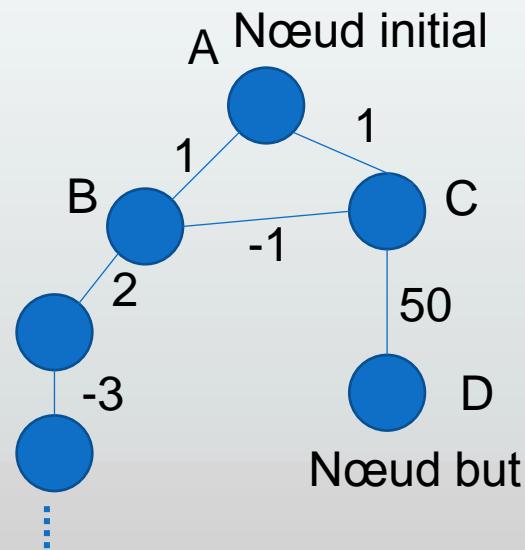
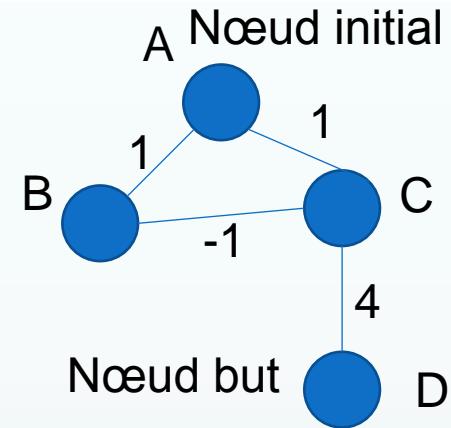
File (frontière)	Nœud développé
[(A,0)]	(A,0)
[(C,1), (B,4)]	(C,1)
[(B,4),(E,21)]	(B,4)
[(D,9),(E,21)]	(D,9)
[(F,11),(E,21)]	(F,11) test du but réussi

Remarque sur la complétude

- L'algorithme garanti la complétude si b est fini car :
 - Il existe un nombre fini d'actions avant que le coût du chemin soit égal au coût du but, donc il sera forcément atteint.
 - Mais le coût des actions doit être supérieur à 0, sinon boucle infinie.

Et si les coûts sont ≤ 0 ?

- Dans le cas de l'exploration d'arbres, les boucles infinies sont faciles à comprendre : l'exploration n'atteint jamais D, elle bouclera entre les états A, B et C.
- Dans le cas de l'exploration de graphes, les espaces d'états infinis sont problématiques : imaginez que le coût des nœuds de la branche infinie partant de B ne dépassera jamais 50.

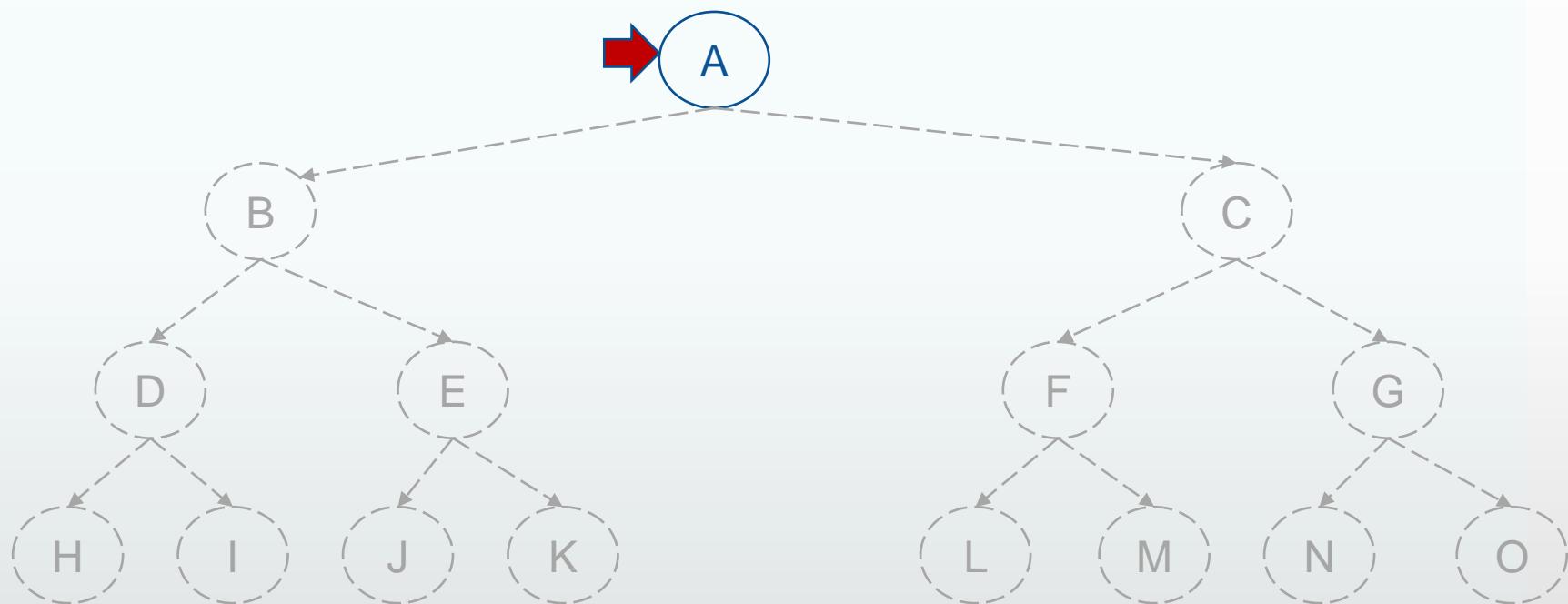


Exploration en profondeur d'abord (*Depth First Search DFS*)

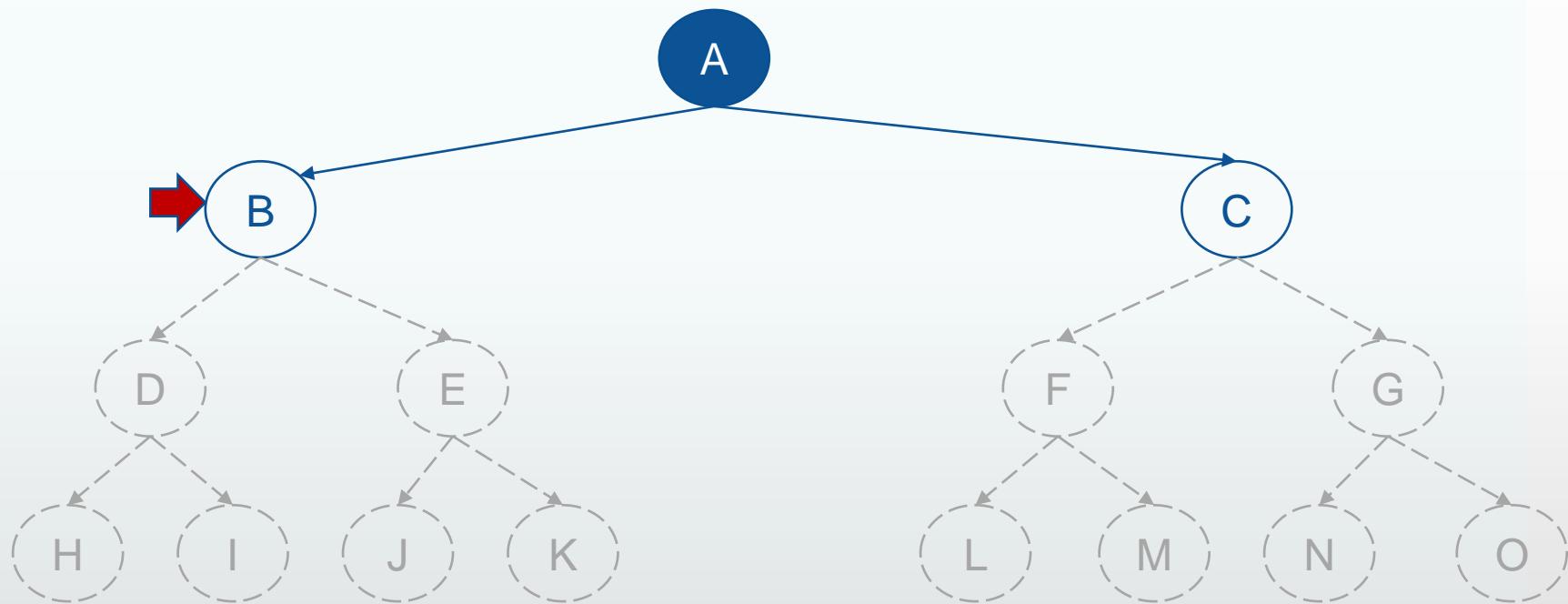
Principe de l'exploration en profondeur d'abord

- Développer toujours le nœud le plus profond de la frontière courante.
- Frontière implémentée par une file LIFO.
- Variante :fonction récursive.

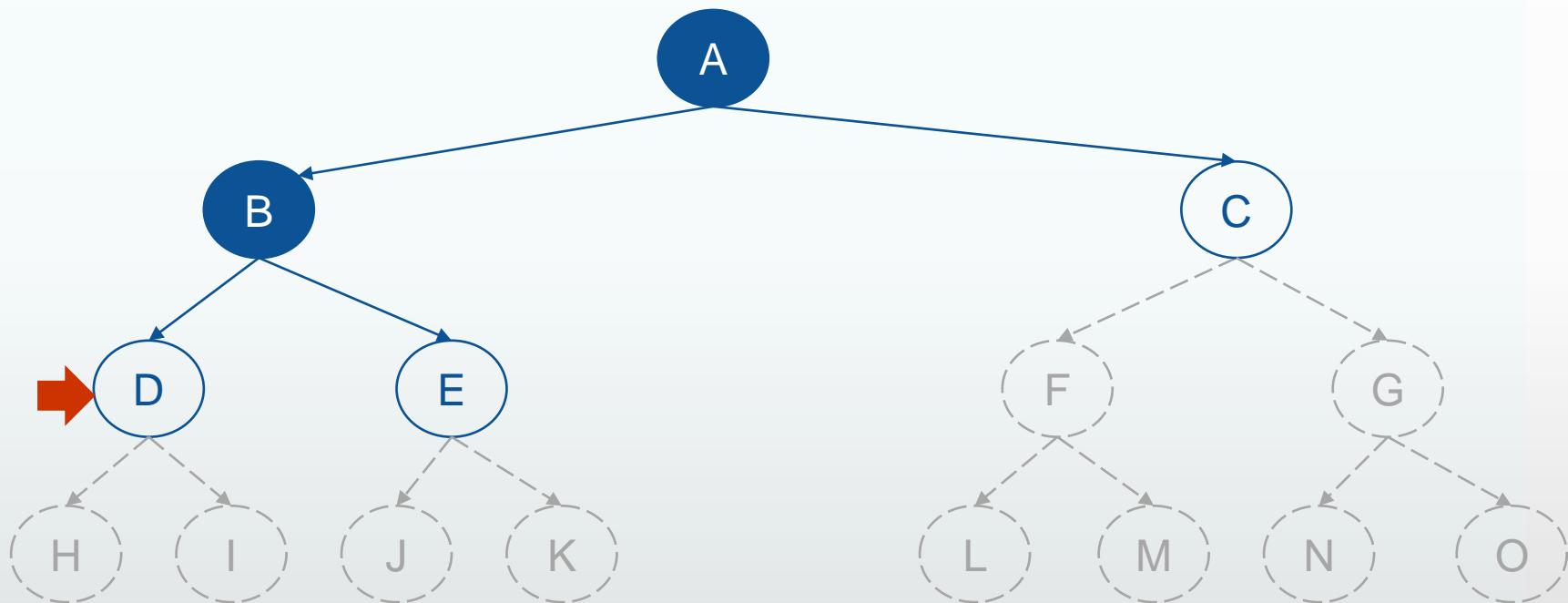
Exploration en profondeur d'abord



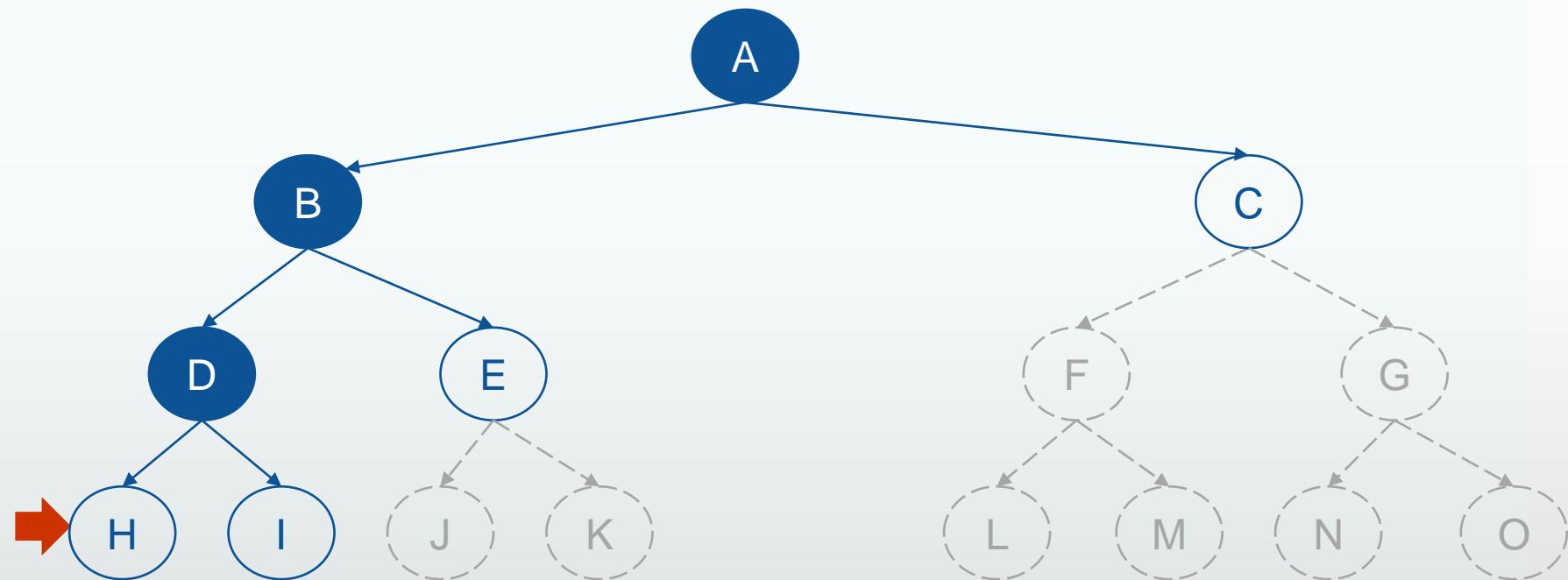
Exploration en profondeur d'abord



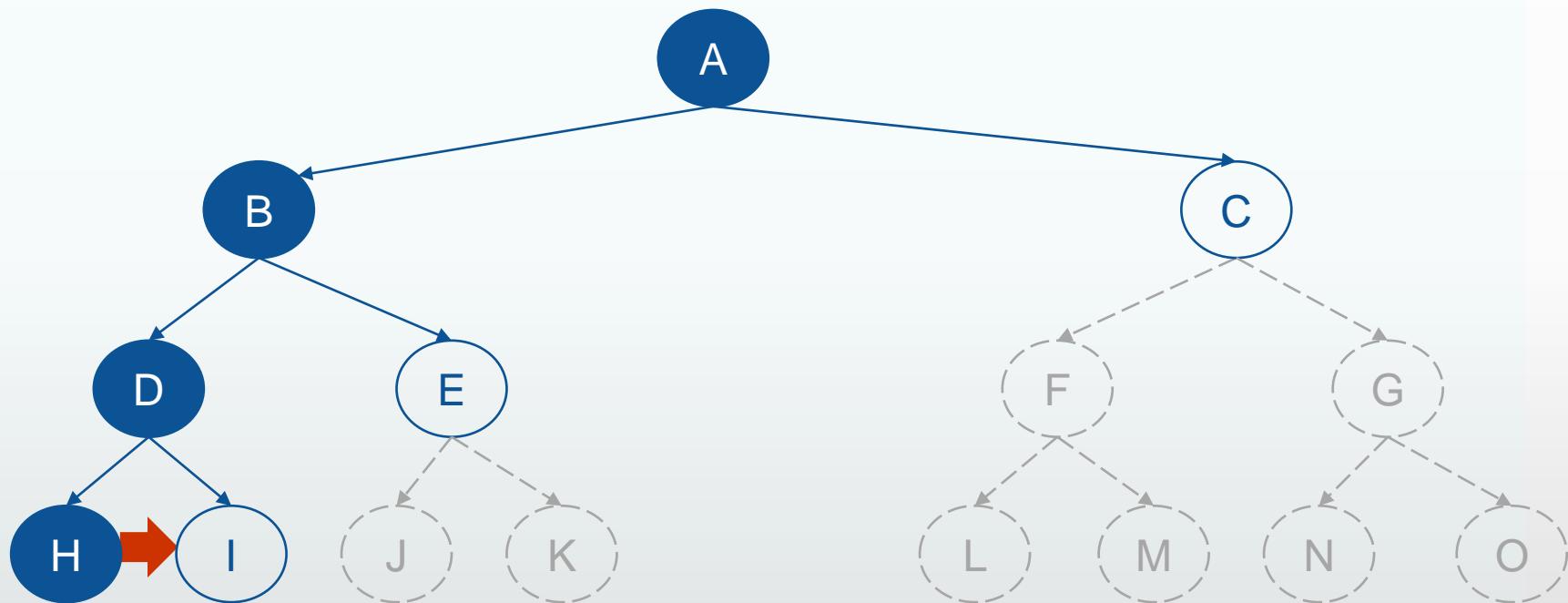
Exploration en profondeur d'abord



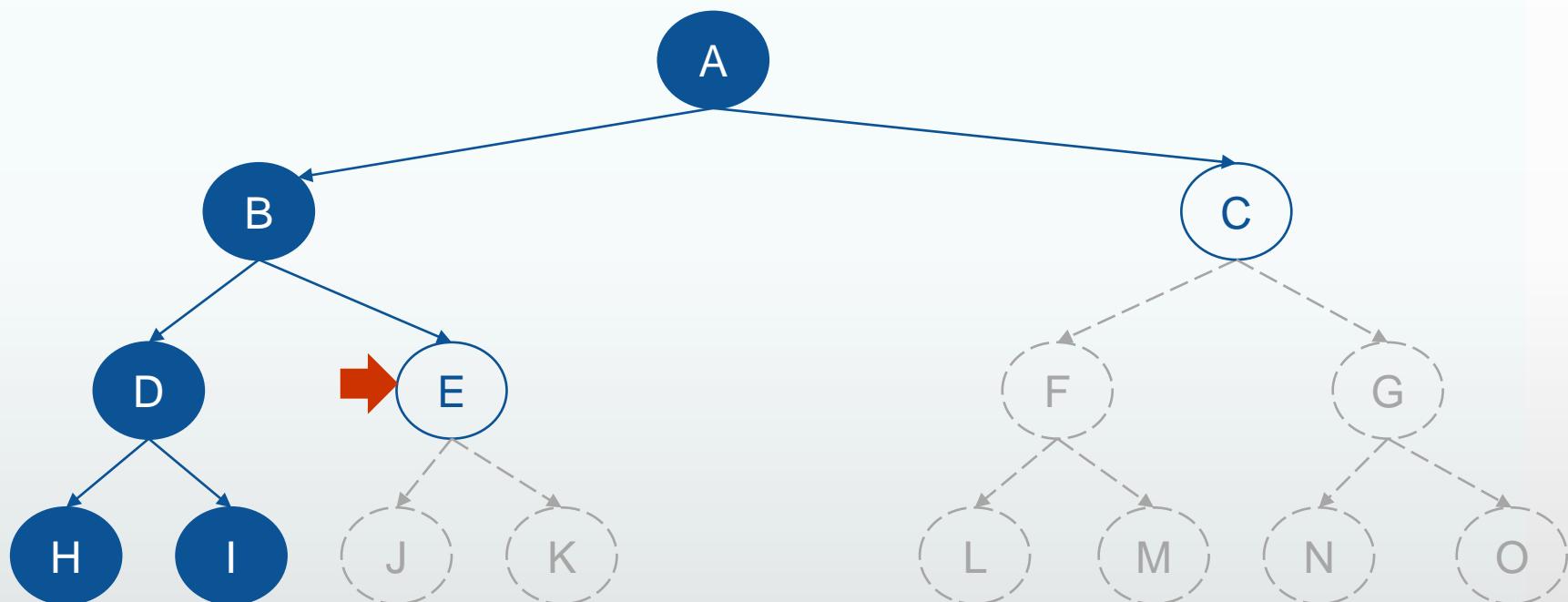
Exploration en profondeur d'abord



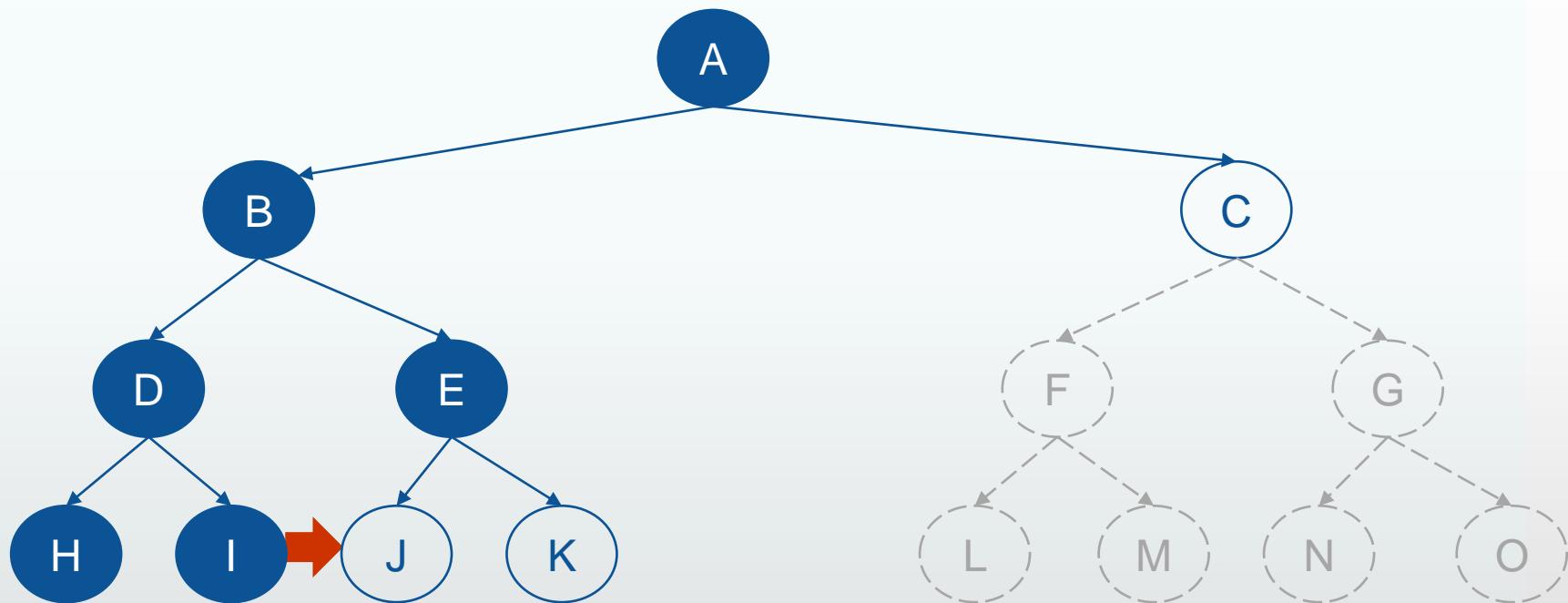
Exploration en profondeur d'abord



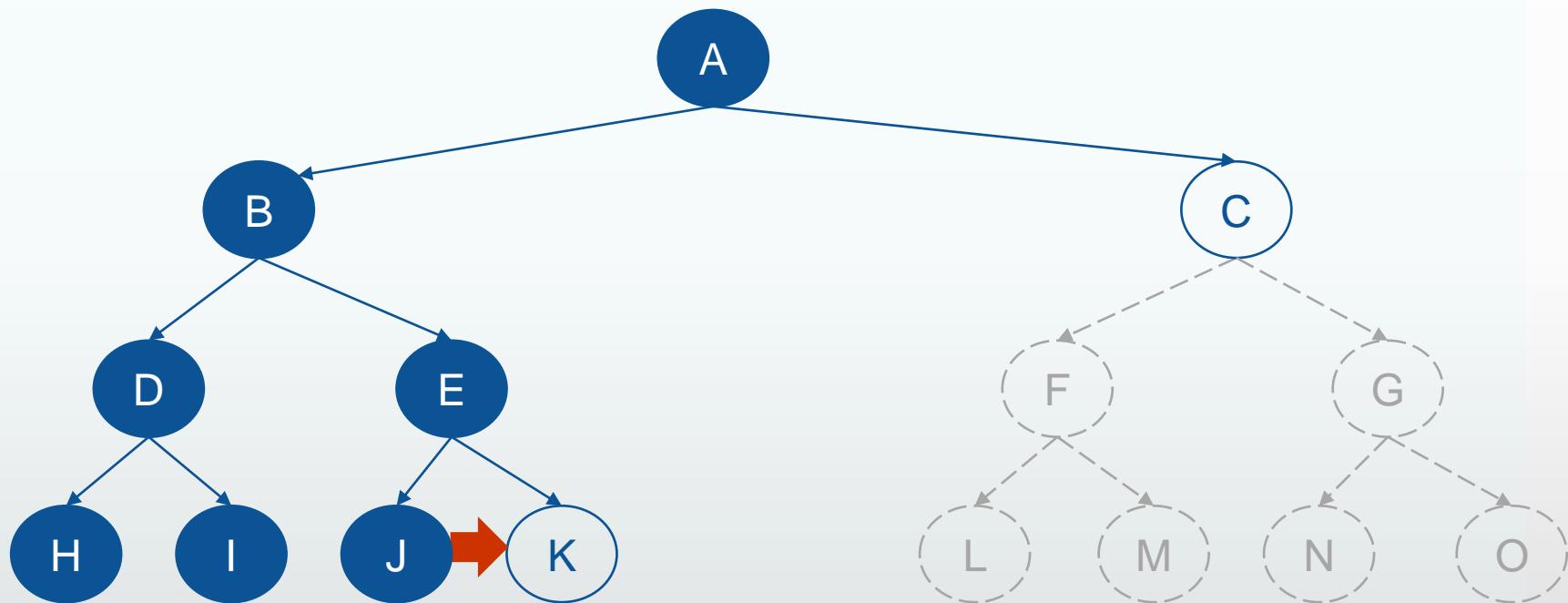
Exploration en profondeur d'abord



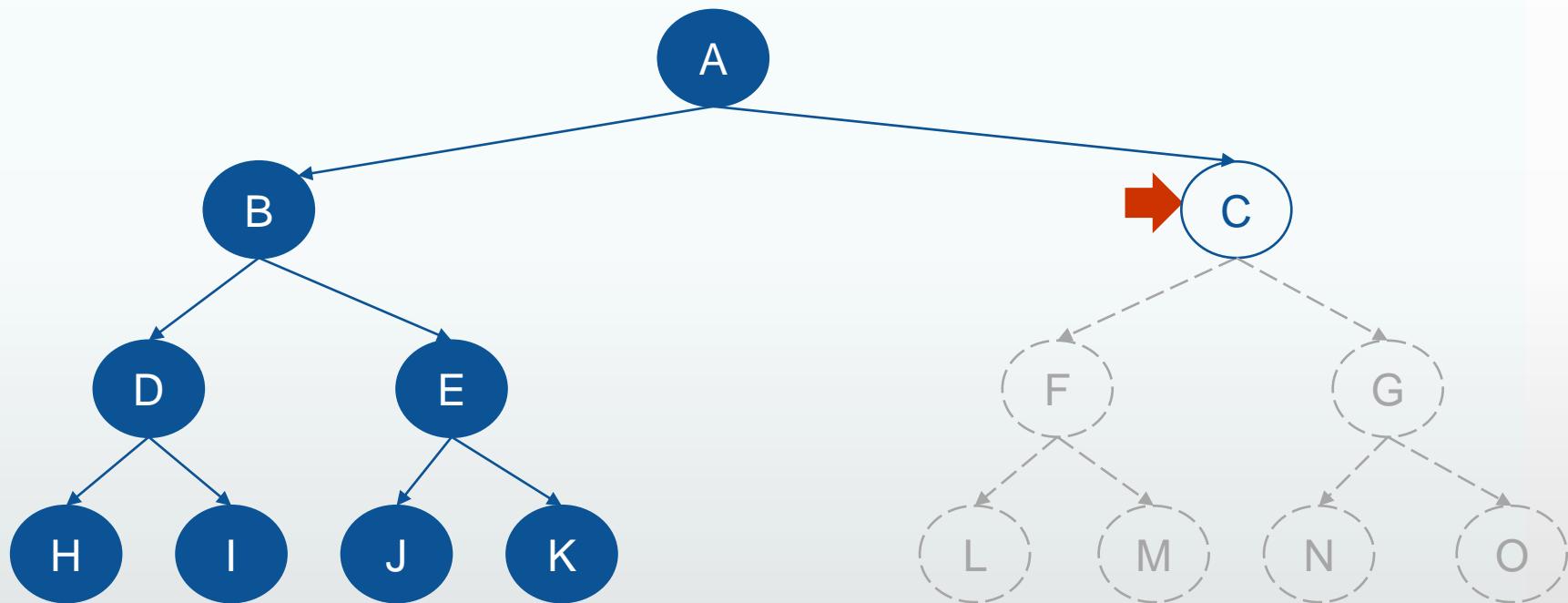
Exploration en profondeur d'abord



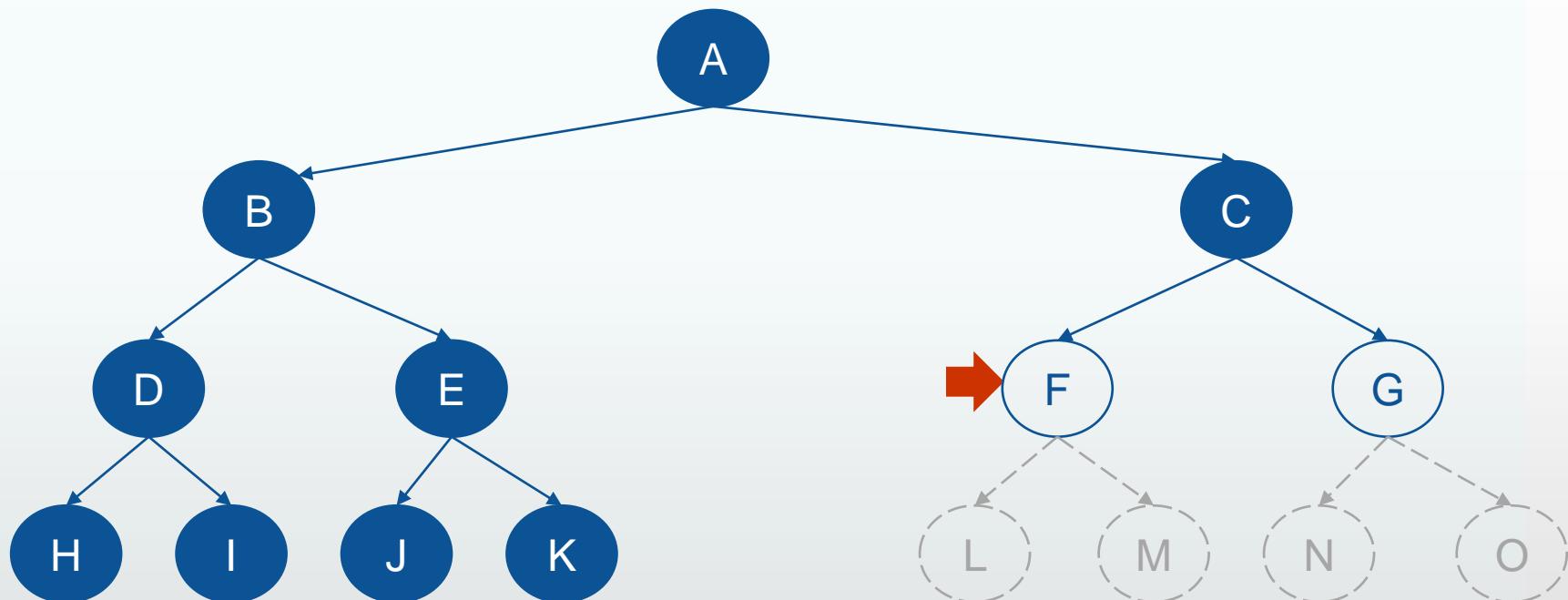
Exploration en profondeur d'abord



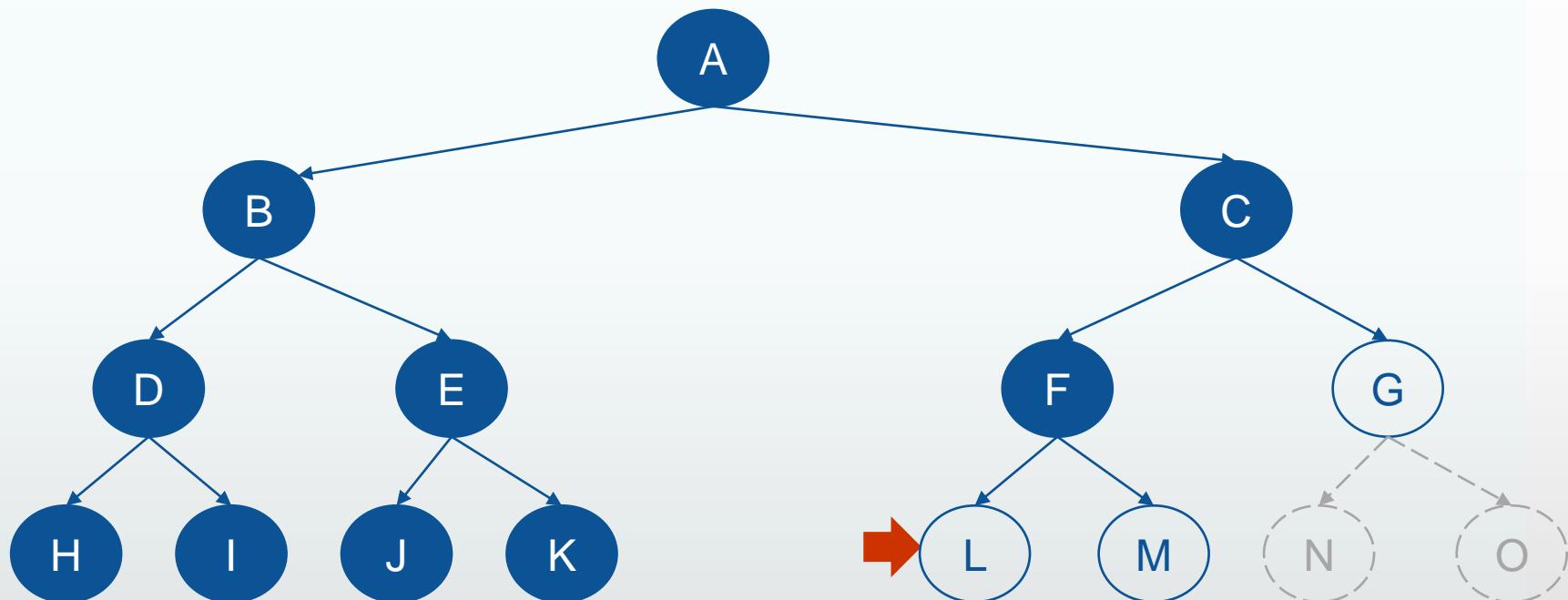
Exploration en profondeur d'abord



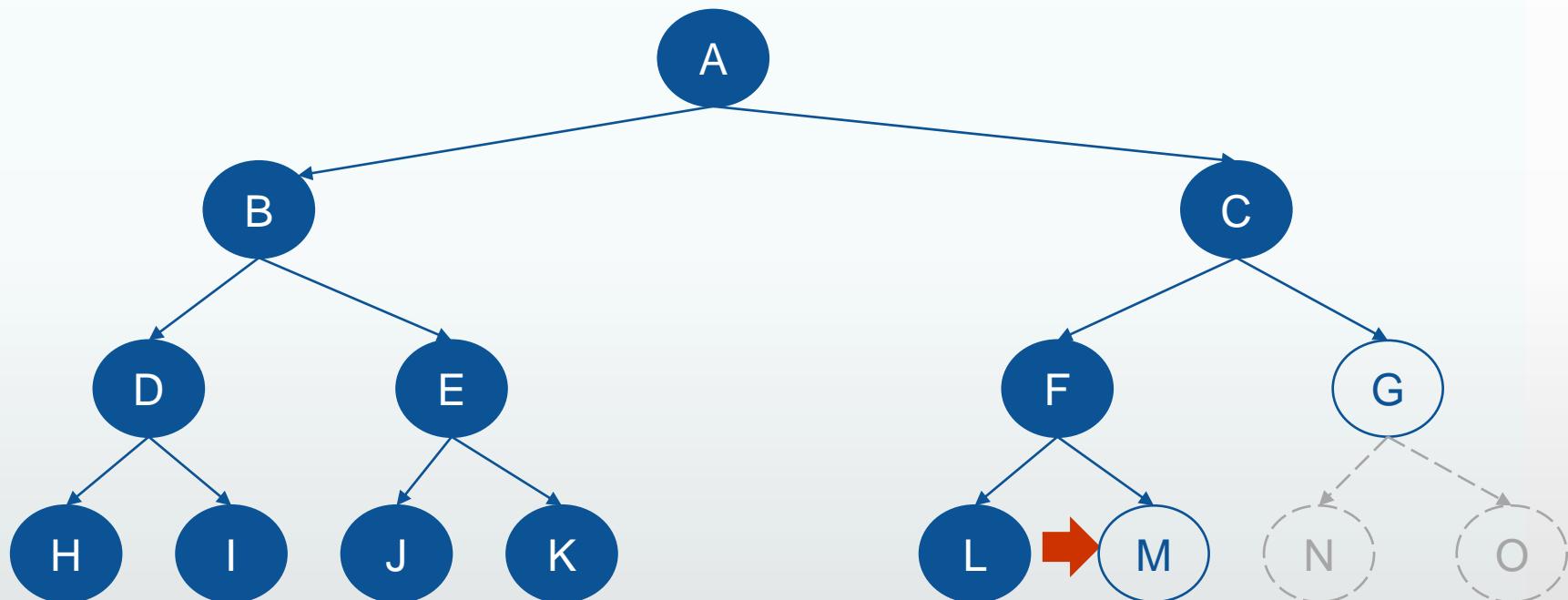
Exploration en profondeur d'abord



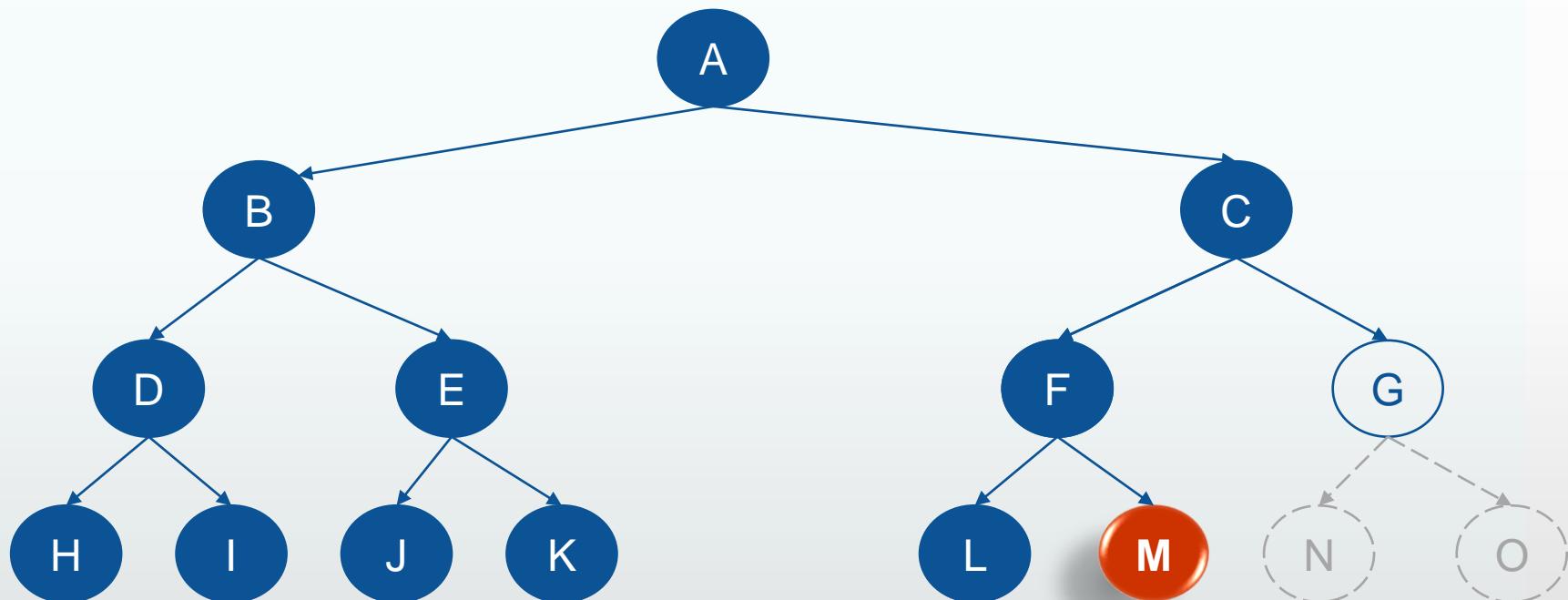
Exploration en profondeur d'abord



Exploration en profondeur d'abord



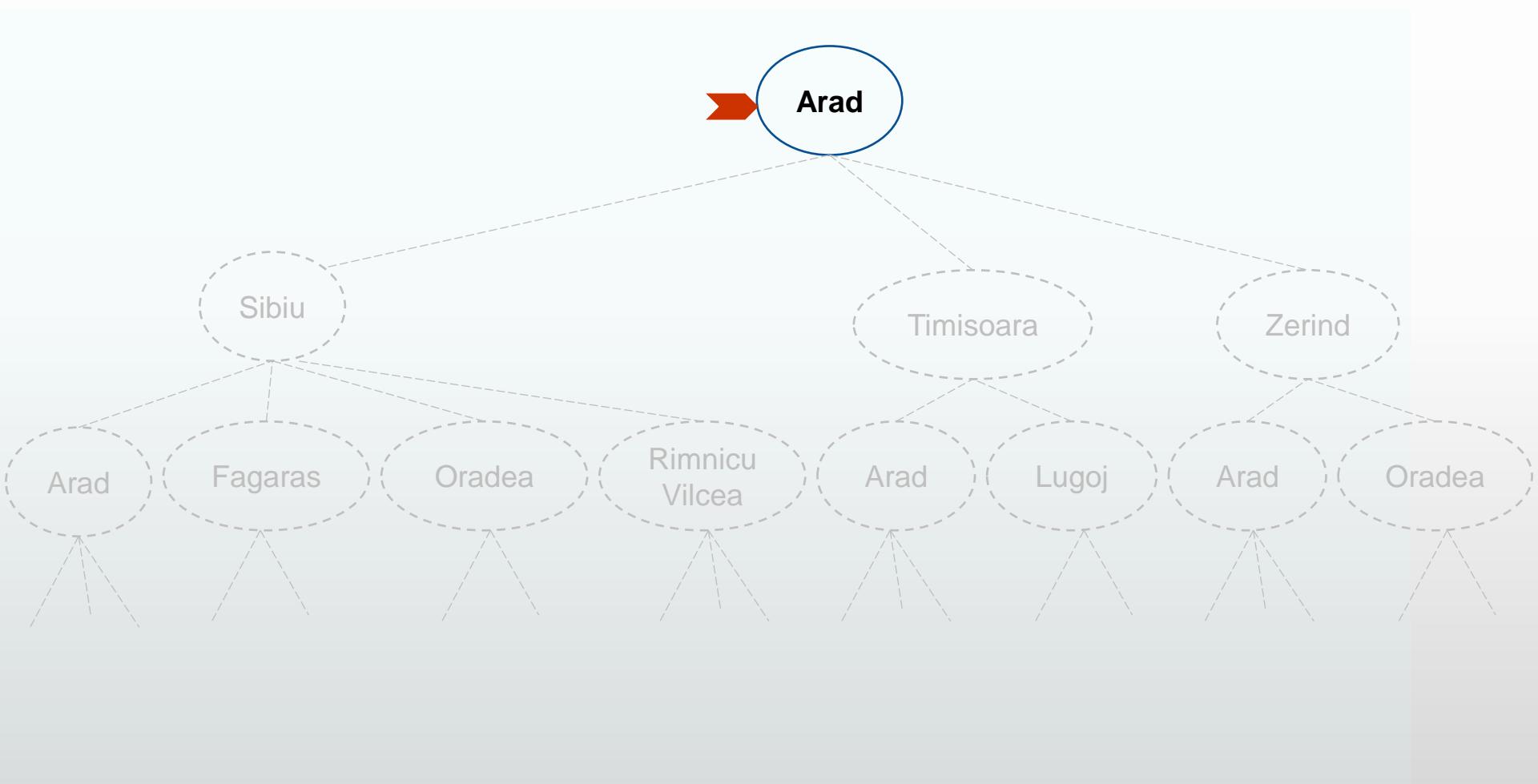
Exploration en profondeur d'abord



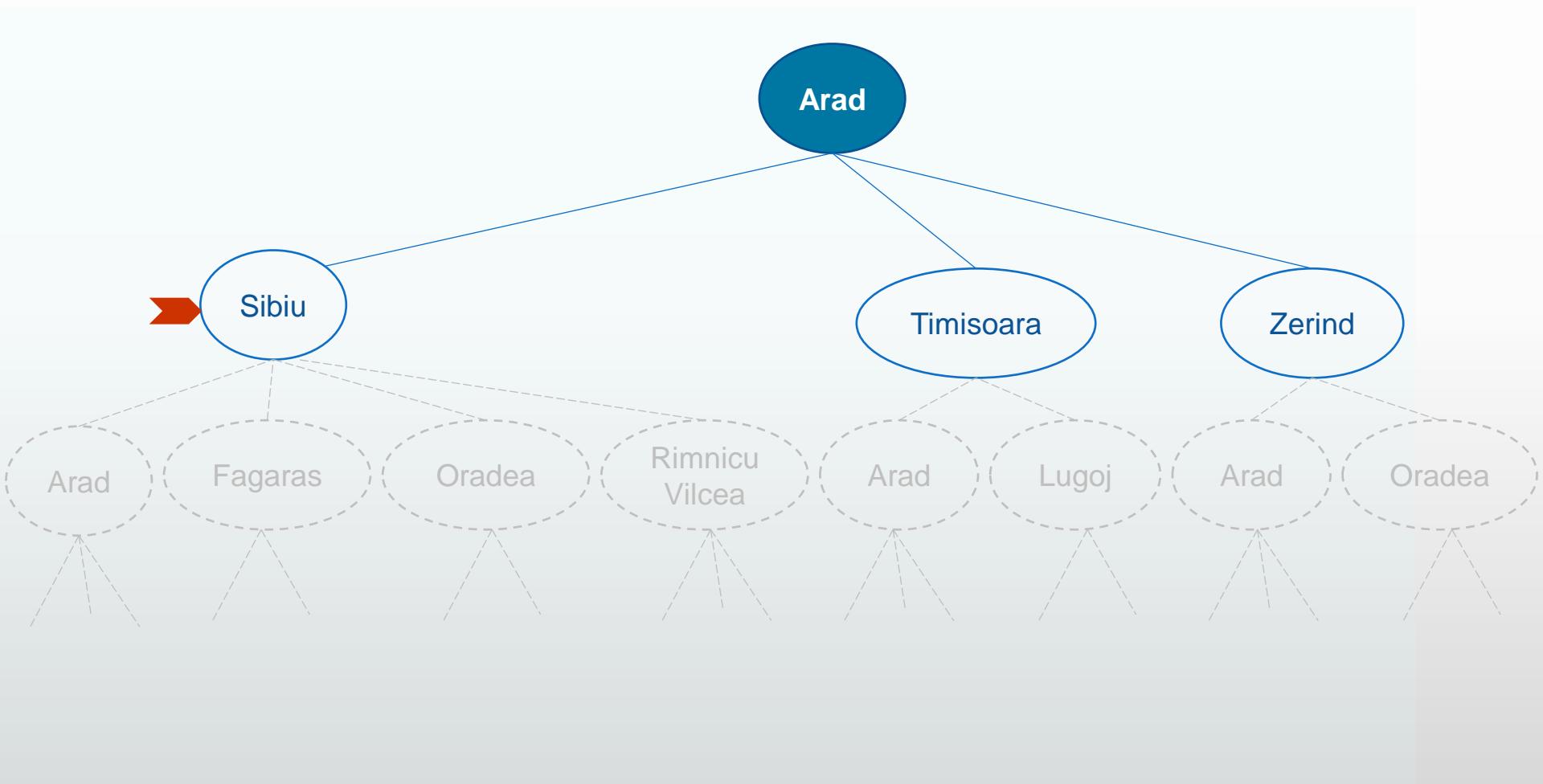
Performances

- Complétude : oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre).

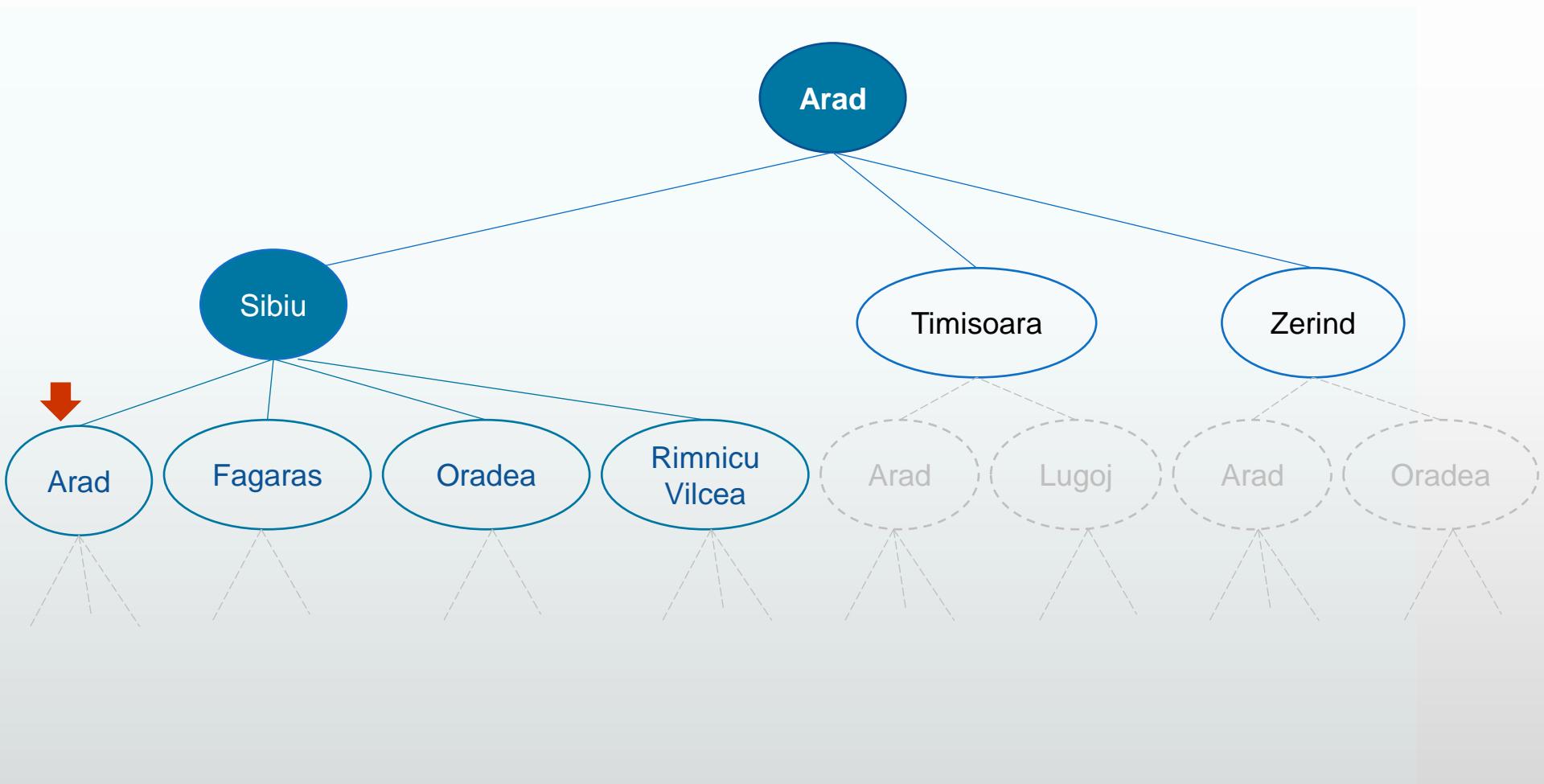
Boucle dans l'exploration en arbre



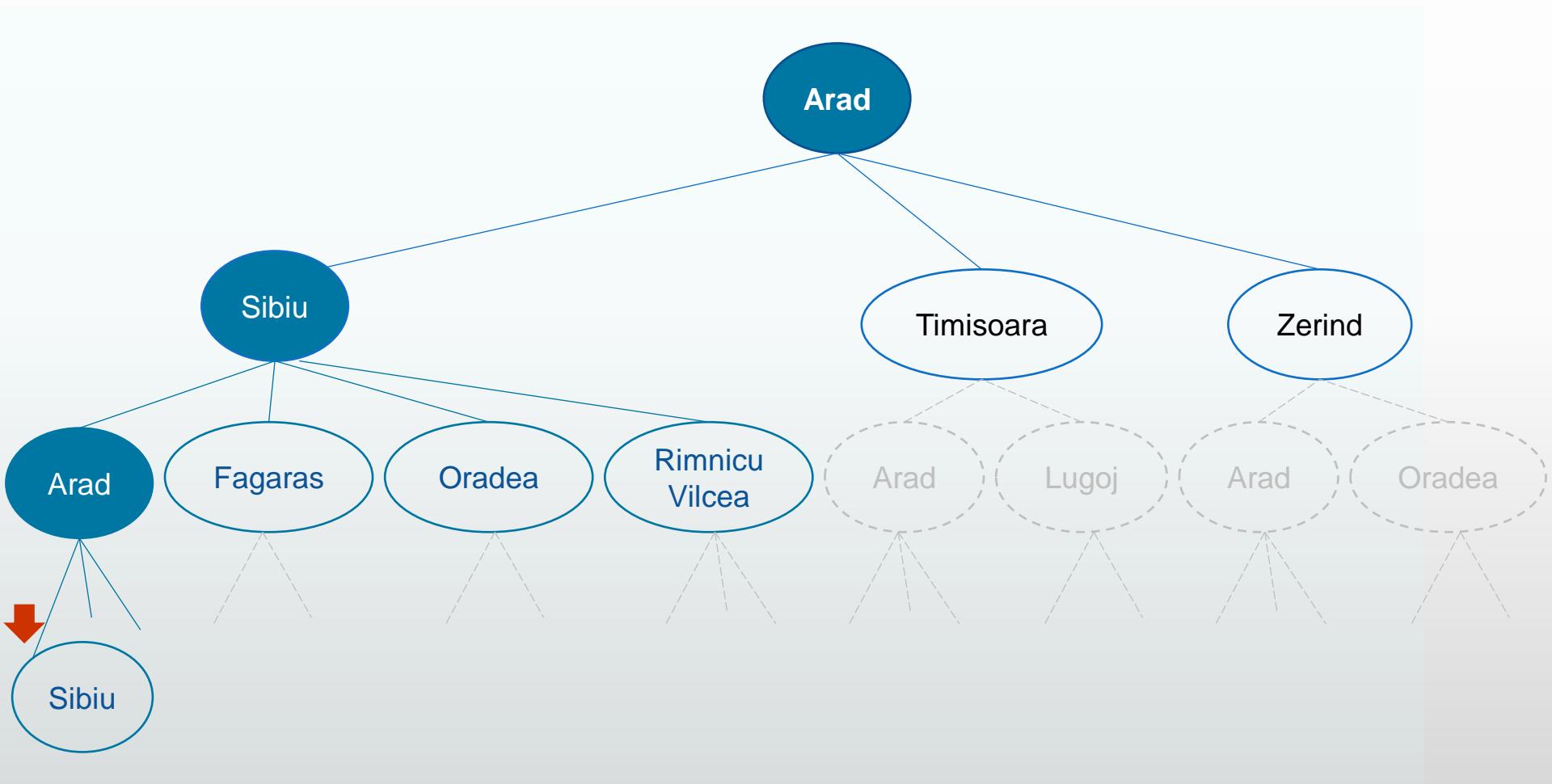
Boucle dans l'exploration en arbre



Boucle dans l'exploration en arbre



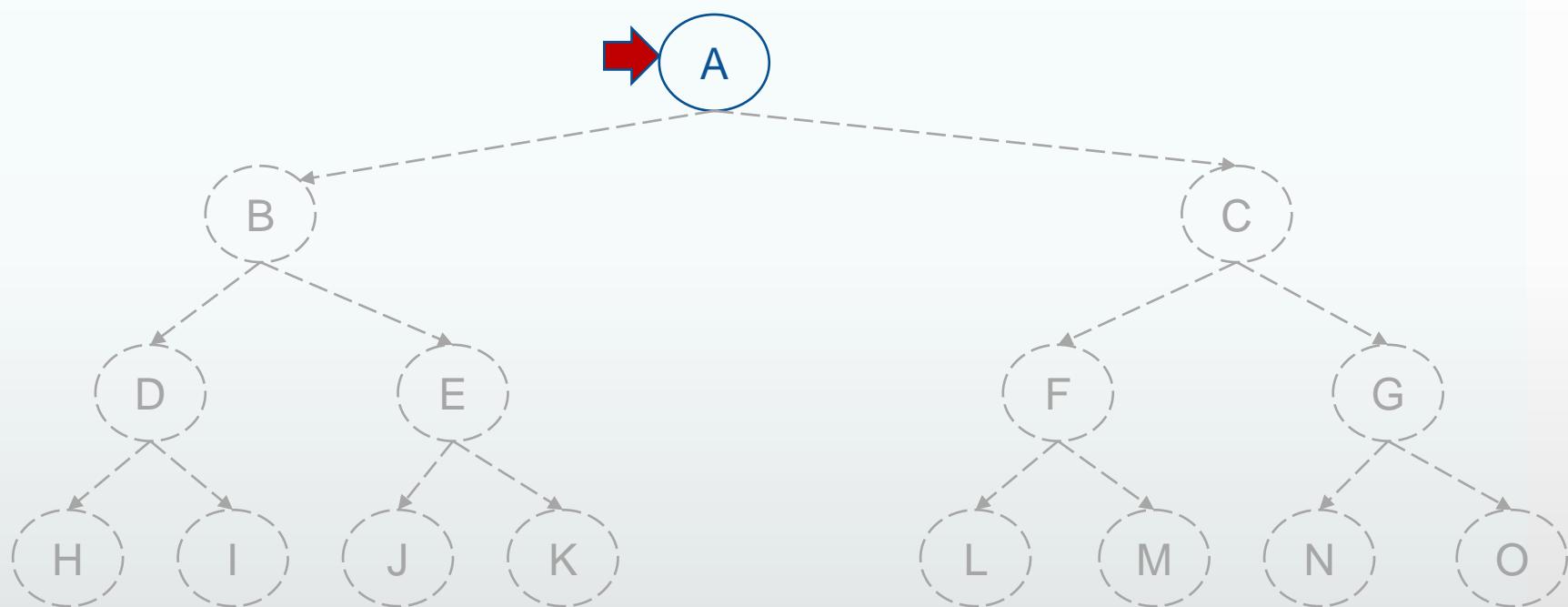
Boucle dans l'exploration en arbre



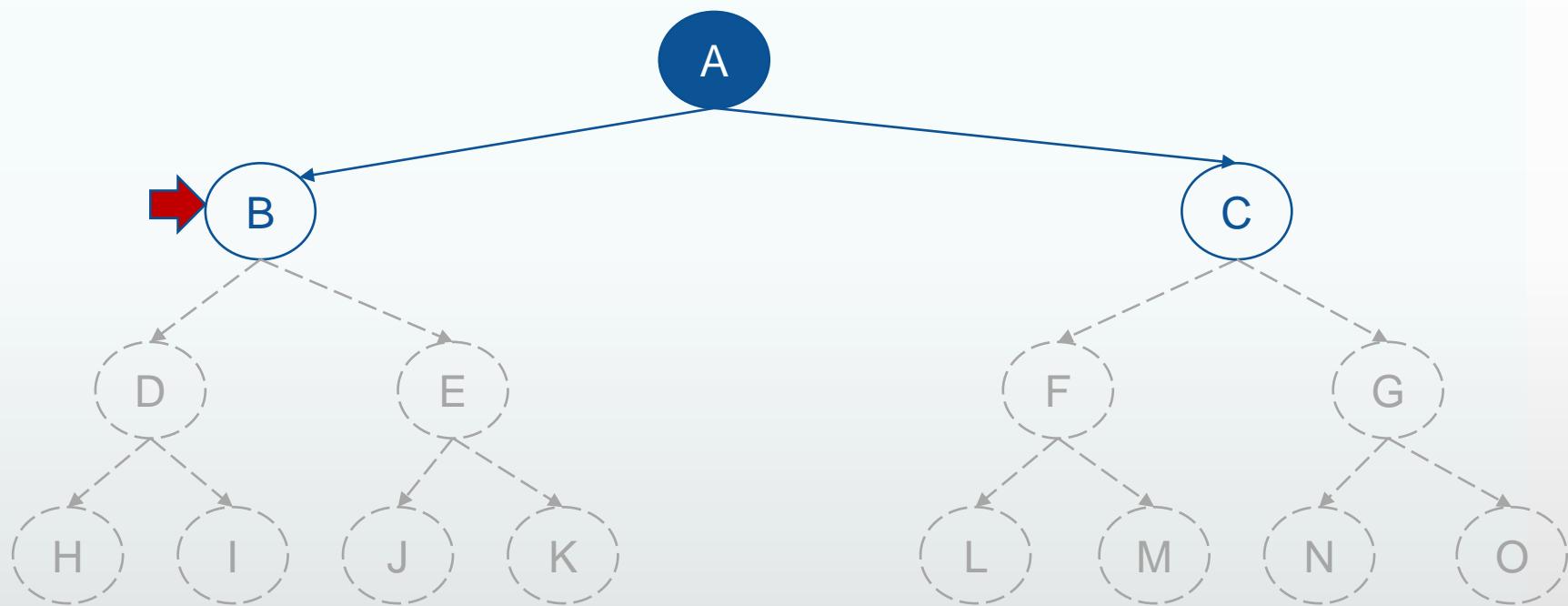
Performances

- ❑ Complétude : oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre).
- ❑ Optimalité : non (imaginez que C ou J soit un nœud but).
- ❑ Complexité temporelle : $O(b^m)$ *m peut être >d.*
- ❑ Complexité spatiale :
 - ❑ $O(bm)$, linéaire pour l'exploration en arbre

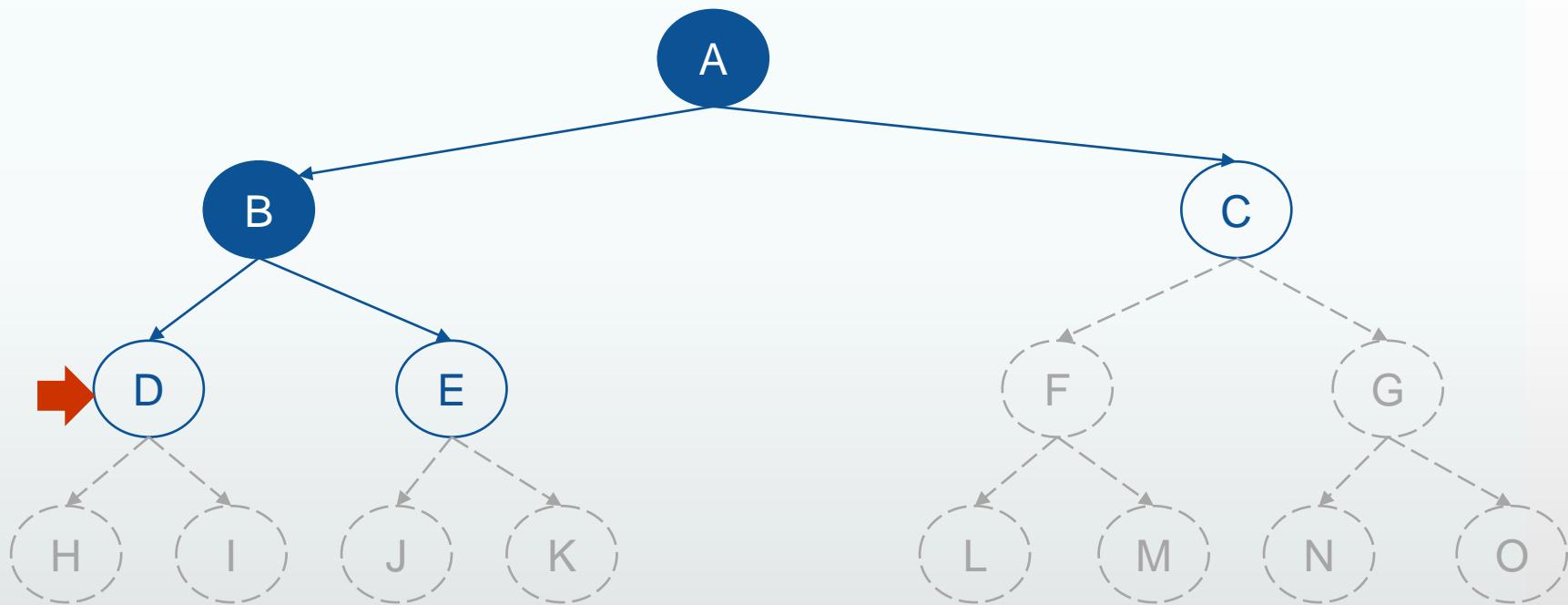
Exploration en profondeur d'abord en arbre avec libération de la mémoire



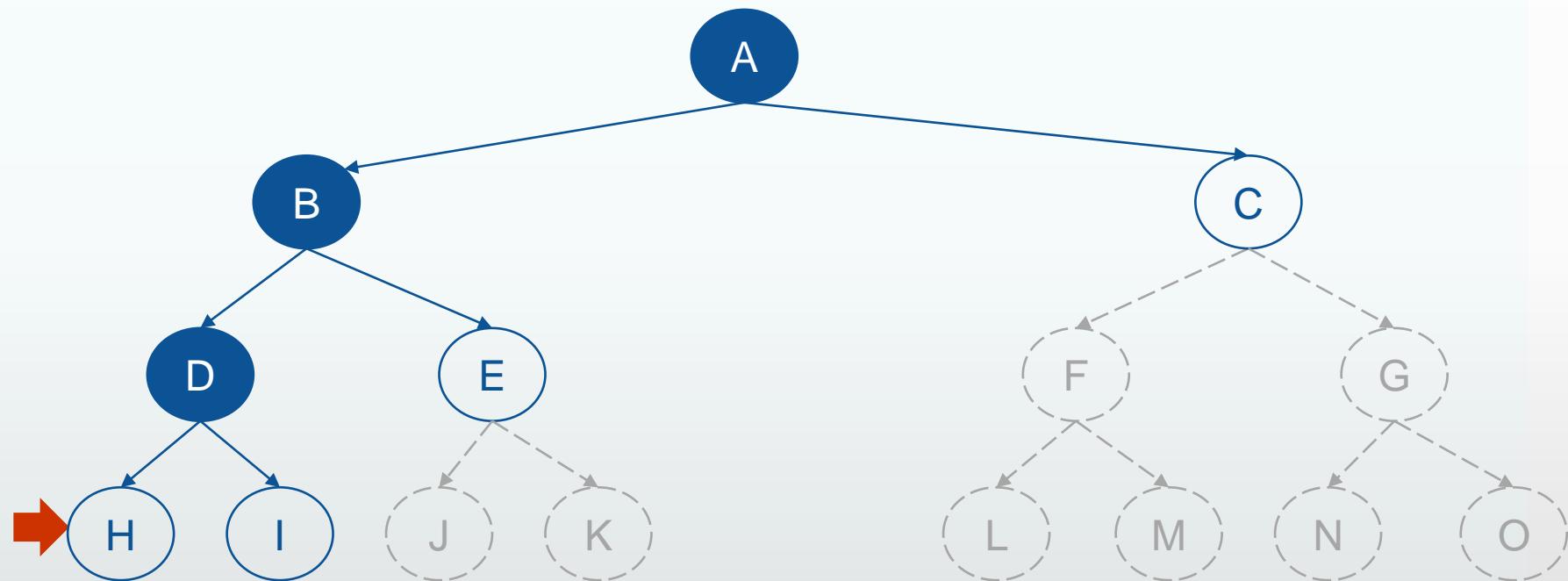
Exploration en profondeur d'abord en arbre avec libération de la mémoire



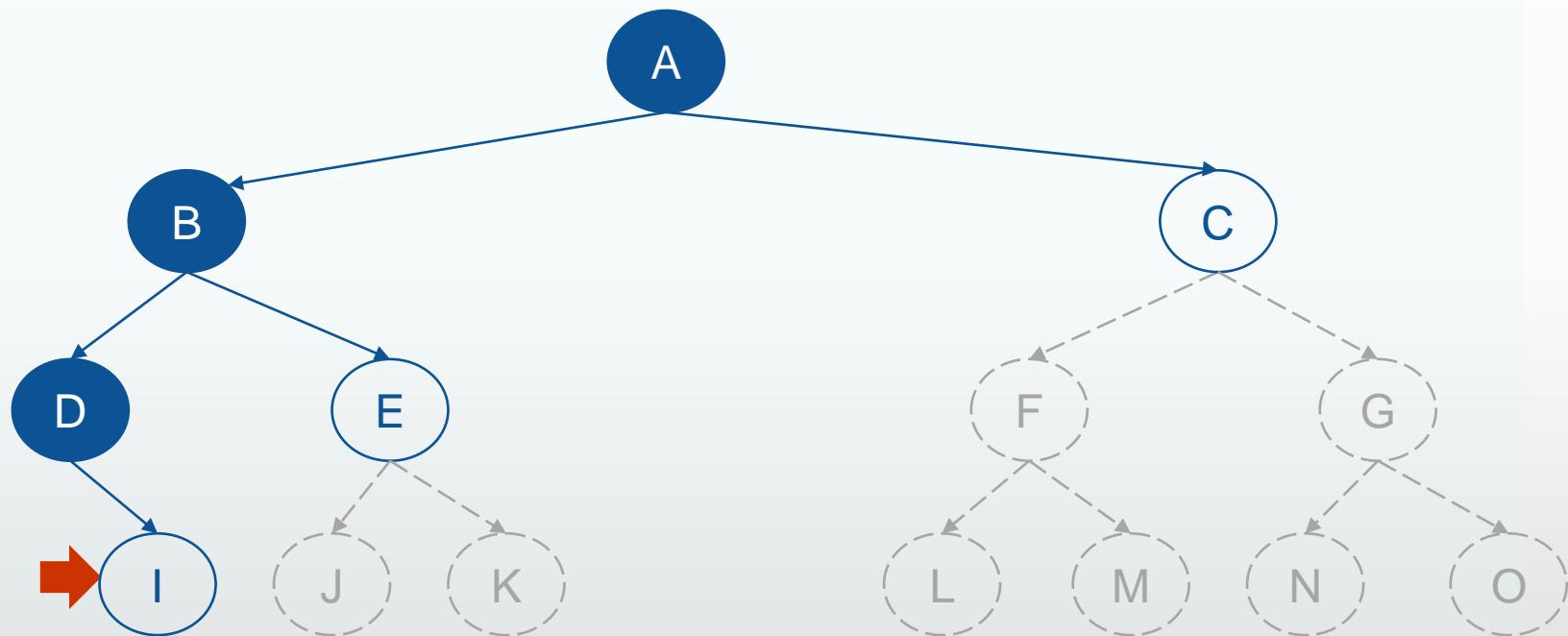
Exploration en profondeur d'abord en arbre avec libération de la mémoire



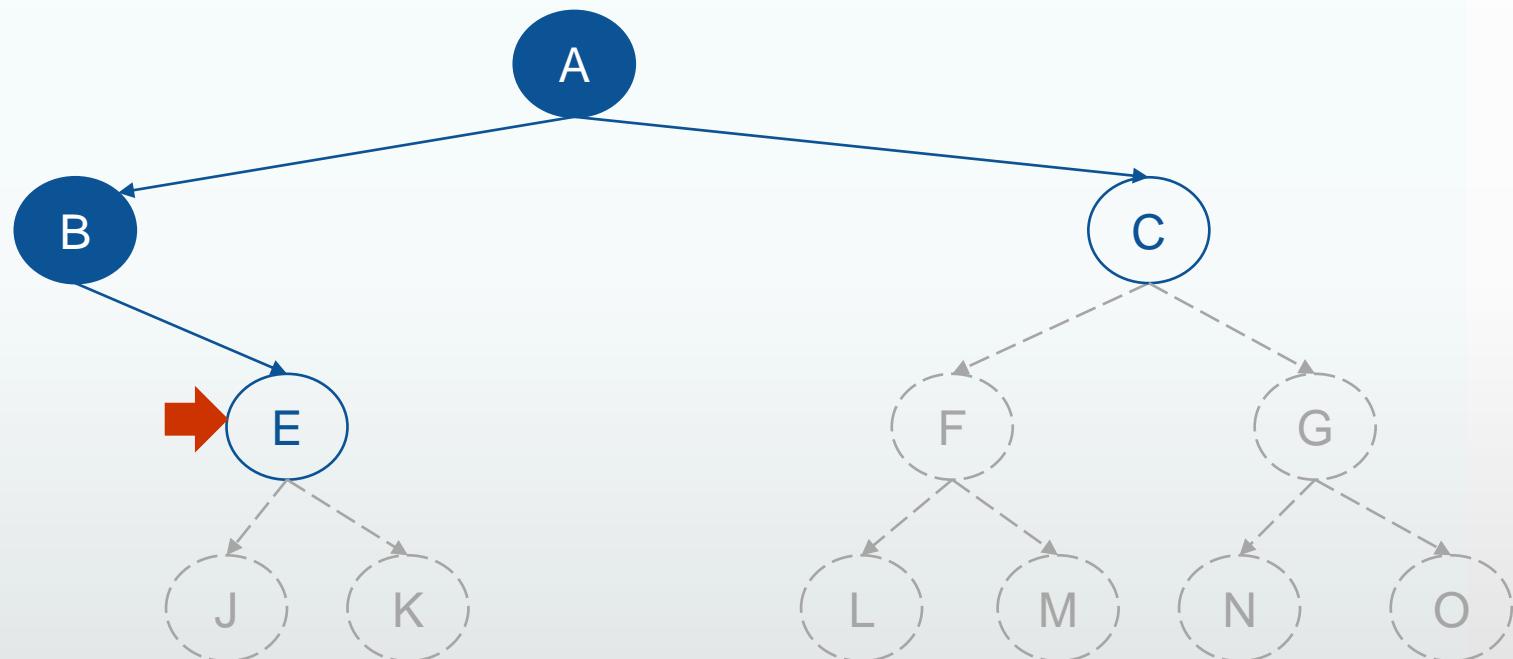
Exploration en profondeur d'abord en arbre avec libération de la mémoire



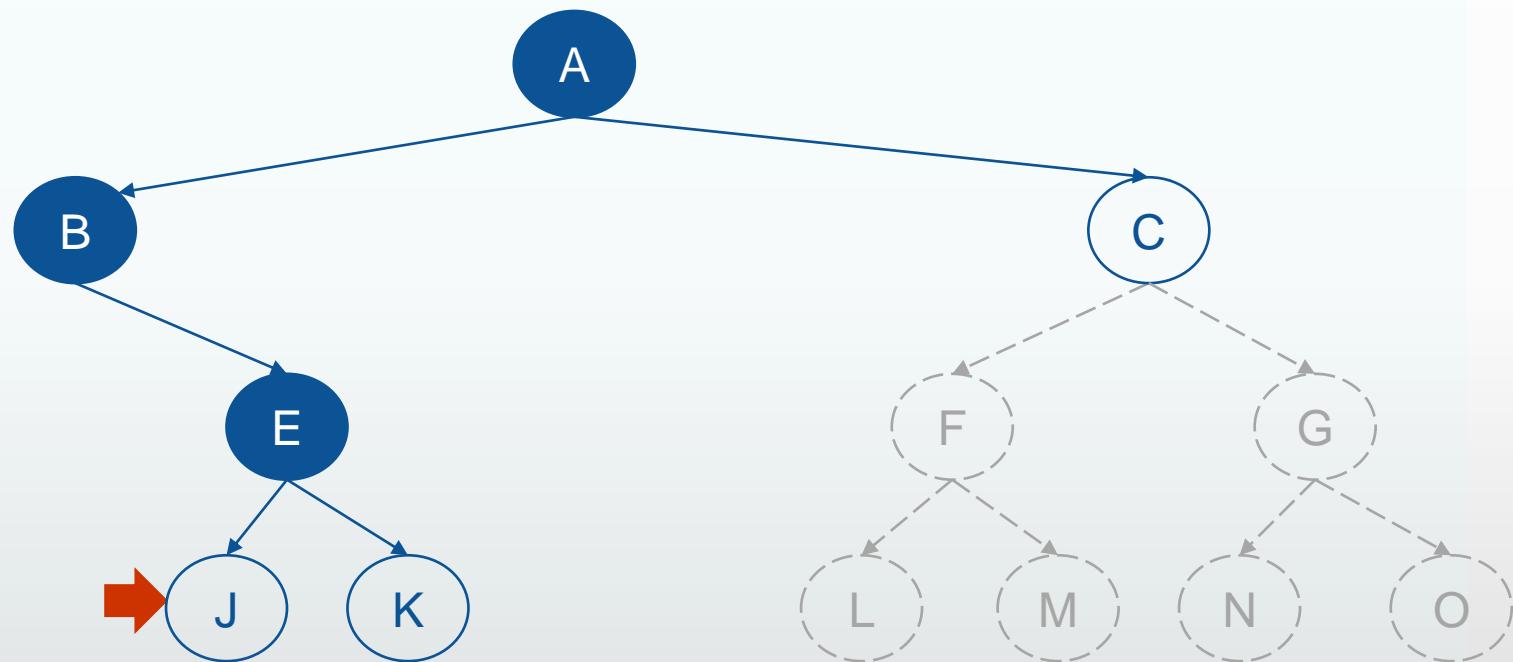
Exploration en profondeur d'abord en arbre avec libération de la mémoire



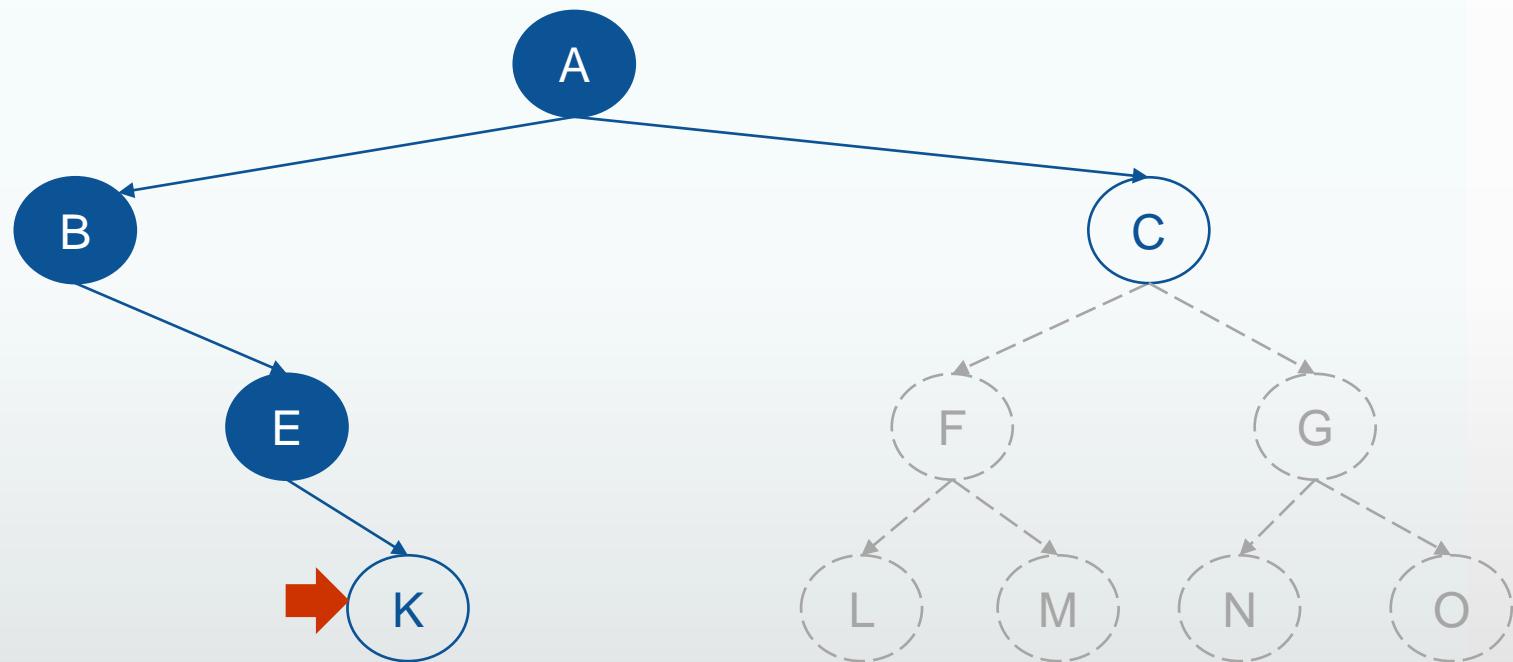
Exploration en profondeur d'abord en arbre avec libération de la mémoire



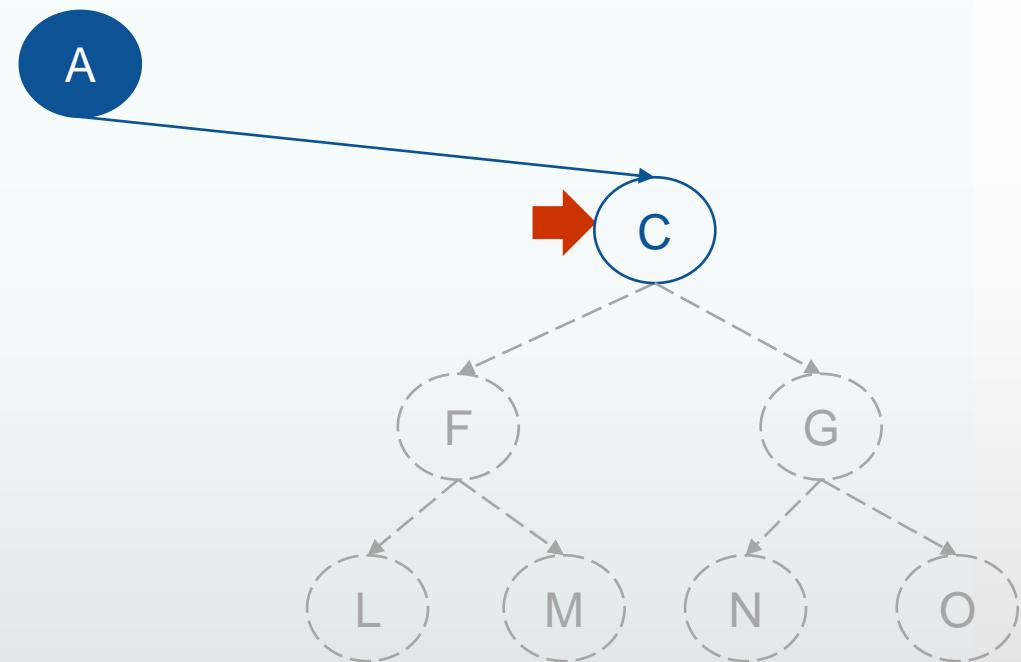
Exploration en profondeur d'abord en arbre avec libération de la mémoire

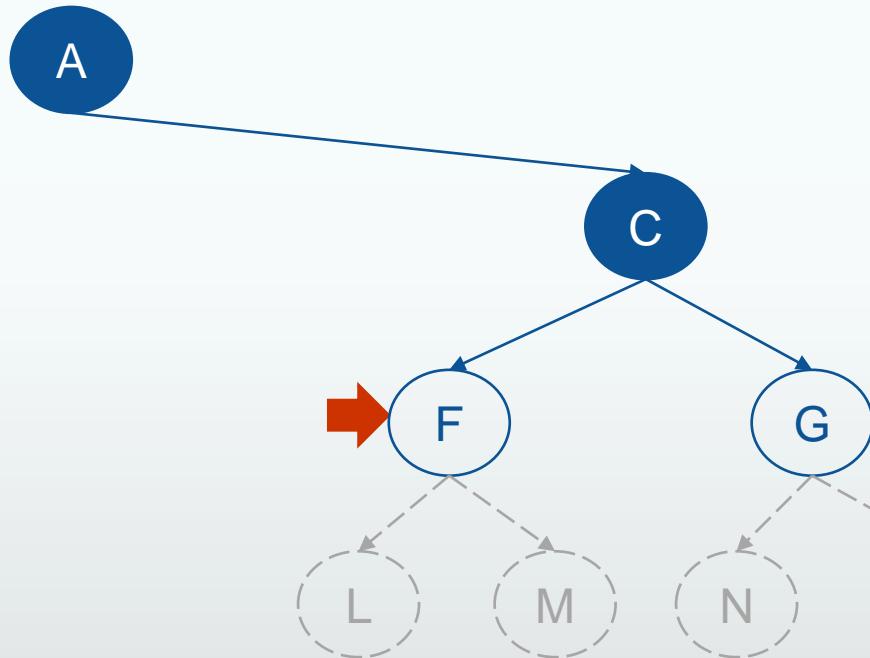


Exploration en profondeur d'abord en arbre avec libération de la mémoire

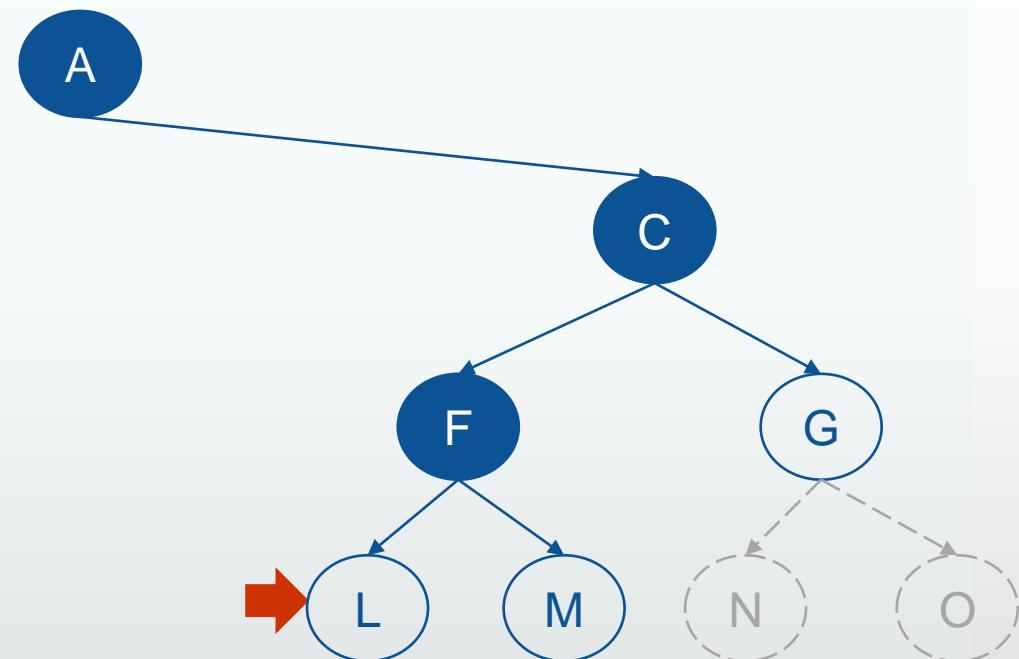


Exploration en profondeur d'abord en arbre avec libération de la mémoire

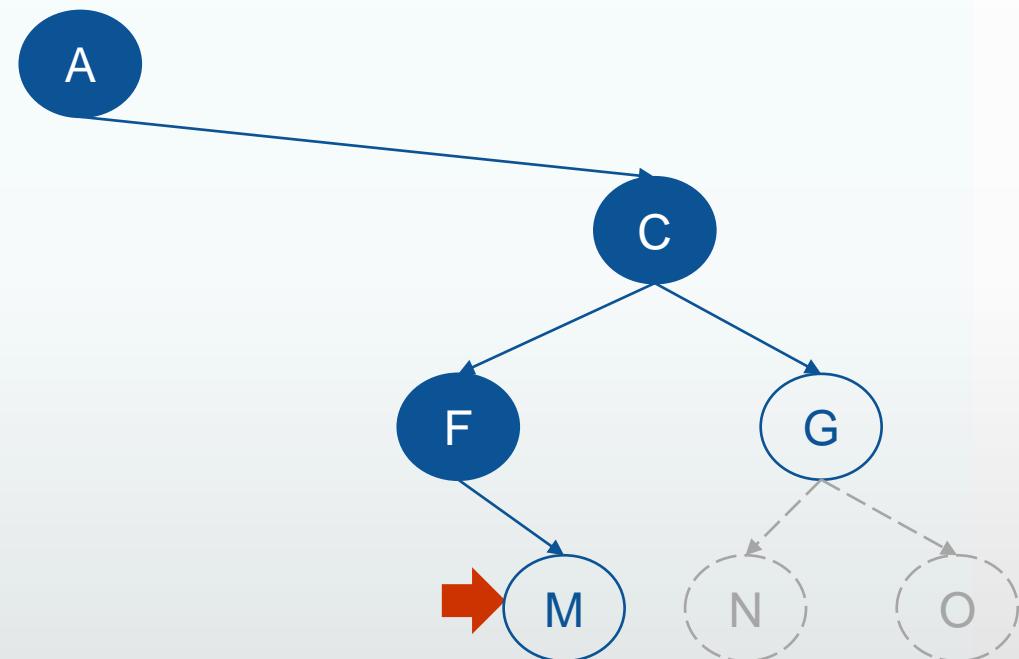




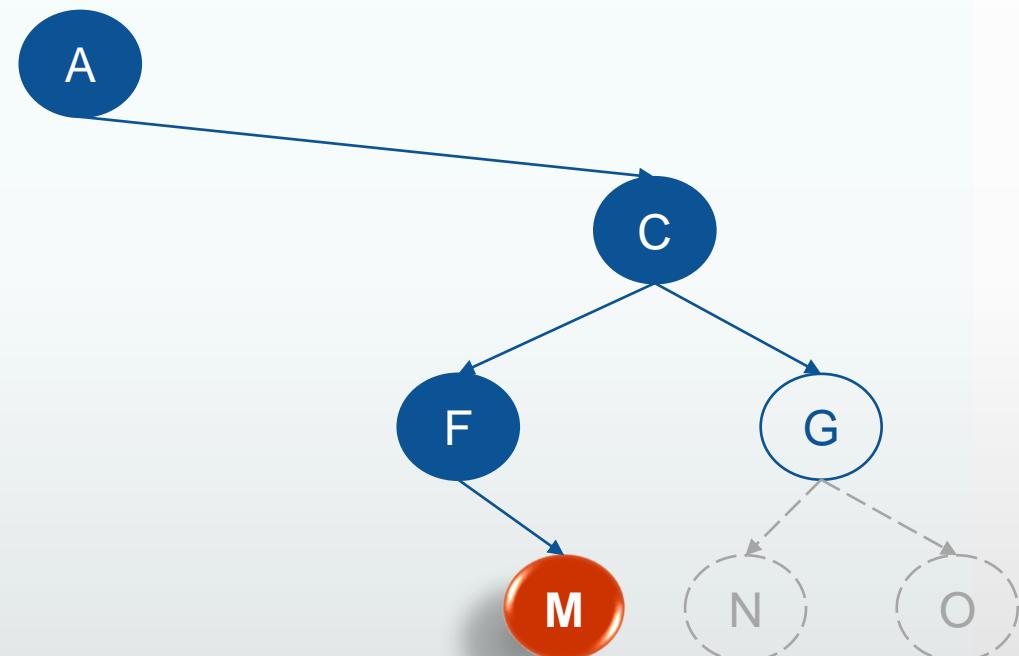
Exploration en profondeur d'abord en arbre avec libération de la mémoire



Exploration en profondeur d'abord en arbre avec libération de la mémoire



Exploration en profondeur d'abord en arbre avec libération de la mémoire



Performances

- ❑ Complétude : oui, si l'espace d'états est fini et pas de chemins redondants (recherche en arbre).
- ❑ Optimalité : non (imaginez que C ou J soit un nœud but).
- ❑ Complexité temporelle : $O(b^m)$ m peut être $>d$.
- ❑ Complexité spatiale :
 - ❑ $O(bm)$, *linéaire* pour l' exploration en arbre.
 - ❑ $O(b^m)$ pour l'exploration en graphe.
 - ❑ $O(m)$ en arbre avec backtraking.

Exploration en profondeur limitée (*Depth Limited Search DLS*)

Principe de l'exploration en profondeur limitée

- ❑ Profondeur limitée : l .
- ❑ Nœuds à la profondeur l sont considérés sans successeurs.
- ❑ Résout le problème du chemin infini.
- ❑ Équivalent à profondeur d'abord si l est infinie.

Performances

- ❑ Complétude : uniquement lorsque $\lVert I \rVert \geq d$.
- ❑ Complexité temporelle : $O(b^l)$.
- ❑ Complexité spatiale : $O(bl)$.
- ❑ On peut améliorer l'efficacité avec une bonne connaissance du problème.

Exploration itérative en profondeur IDS (ou
exploration par approfondissement itératif)

Principe de l'exploration itérative en profondeur

- ❑ Variante de l'exploration en profondeur d'abord.
- ❑ Détermine la meilleure profondeur limite.
- ❑ Essaye toutes les profondeurs graduellement (0, puis 1, puis 2, ...) jusqu'à atteindre le but (profondeur optimale d).
- ❑ Combine les avantages de :
 - ❑ l'exploration en largeur d'abord (complète et optimale) et
 - ❑ l'exploration en profondeur d'abord (complexité temporelle).
- ❑ Recommandée lorsque l'espace d'états est important et la profondeur de la solution est inconnue.

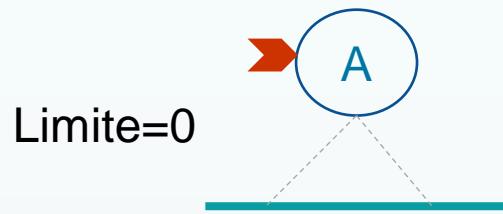
Algorithme d'exploration itérative en profondeur

```
fonction EXPLORATION-ITÉRATIVE-PROFONDEUR(problème) retourne une solution, ou échec  
    pour profondeur = 0 jusqu'à  $\infty$  faire  
        résultat  $\leftarrow$  EXPLORATION-LIMITÉE-PROFONDEUR(problème, profondeur)  
        si résultat  $\neq$  arrêt alors retourner résultat
```

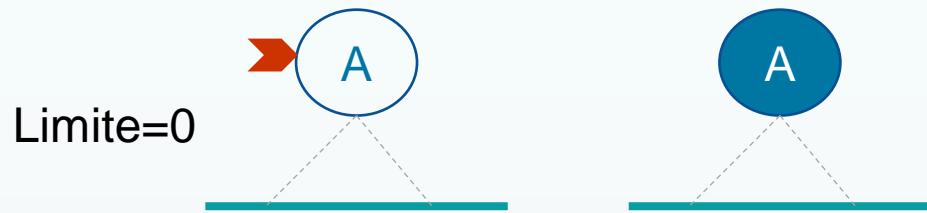
Exemple de l'exploration itérative en profondeur



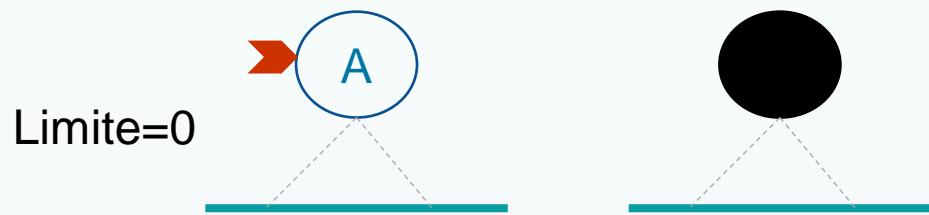
Exemple de l'exploration itérative en profondeur



Exemple de l'exploration itérative en profondeur

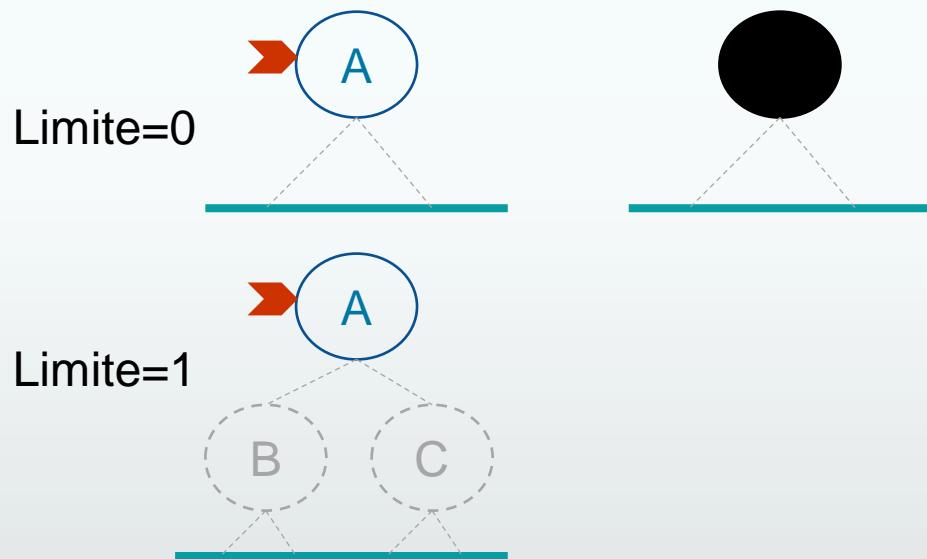


Exemple de l'exploration itérative en profondeur

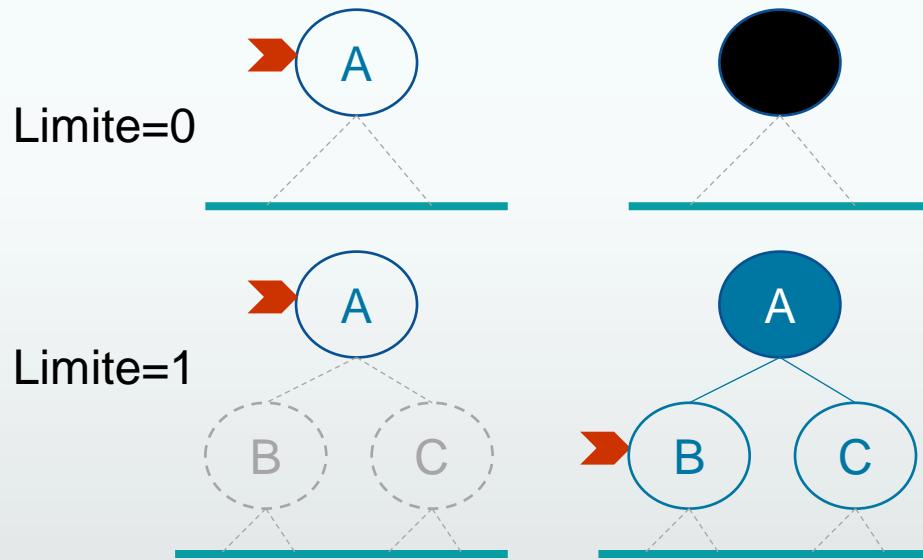


Limite=1

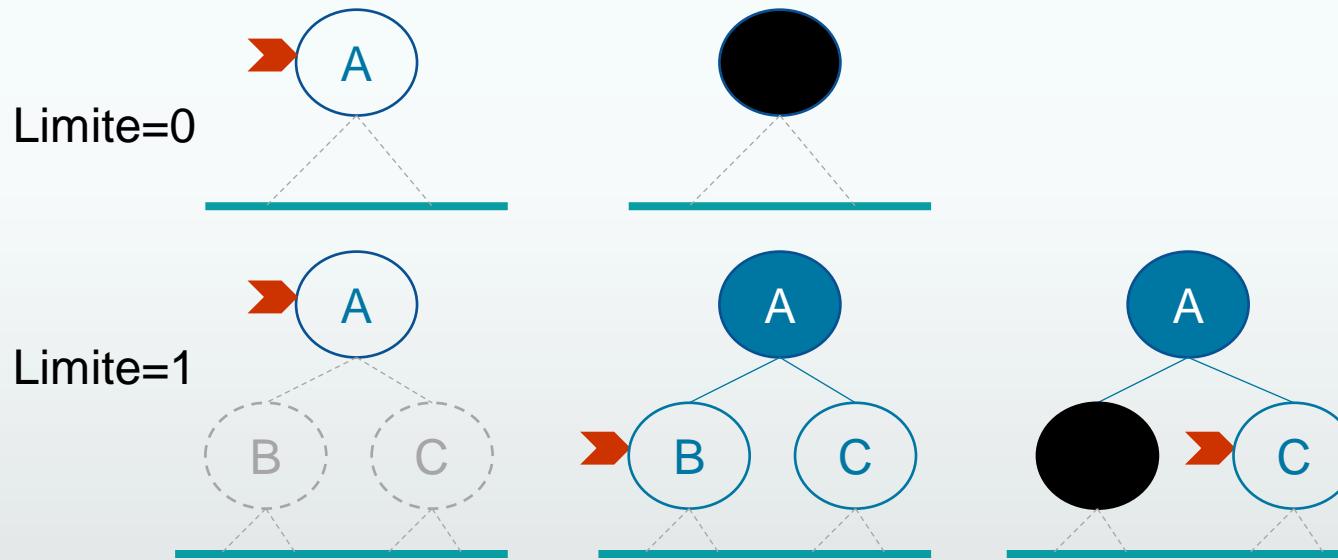
Exemple de l'exploration itérative en profondeur



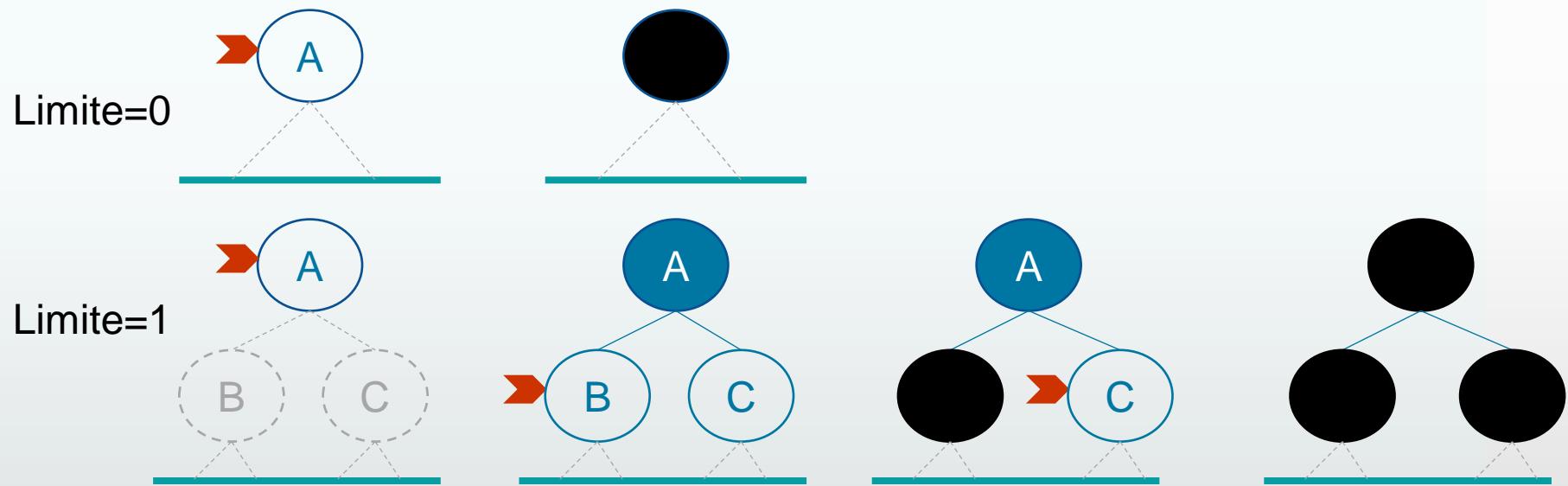
Exemple de l'exploration itérative en profondeur



Exemple de l'exploration itérative en profondeur



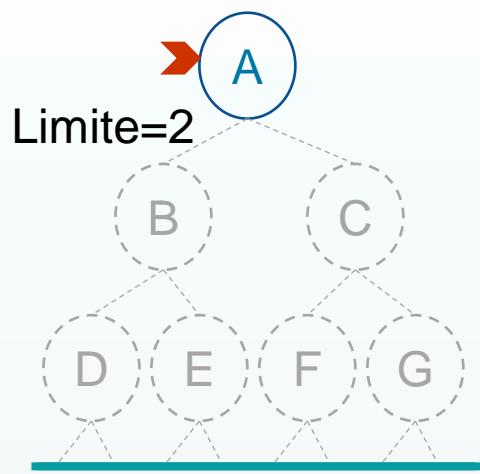
Exemple de l'exploration itérative en profondeur



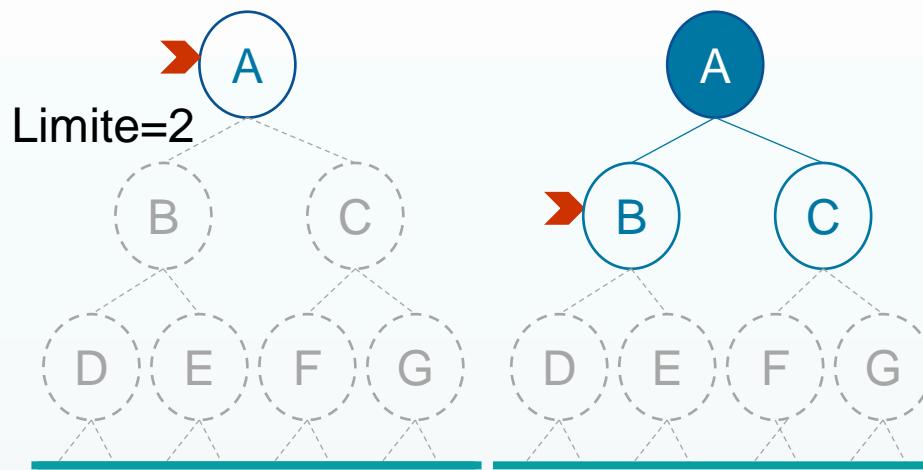
Exemple de l'exploration itérative en profondeur

Limite=2

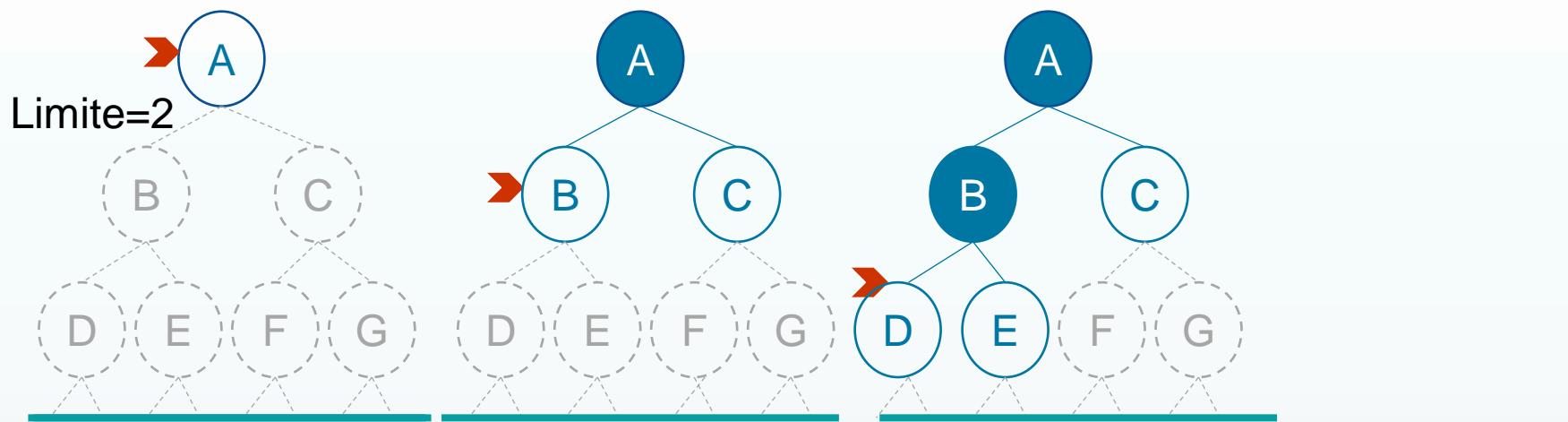
Exemple de l'exploration itérative en profondeur



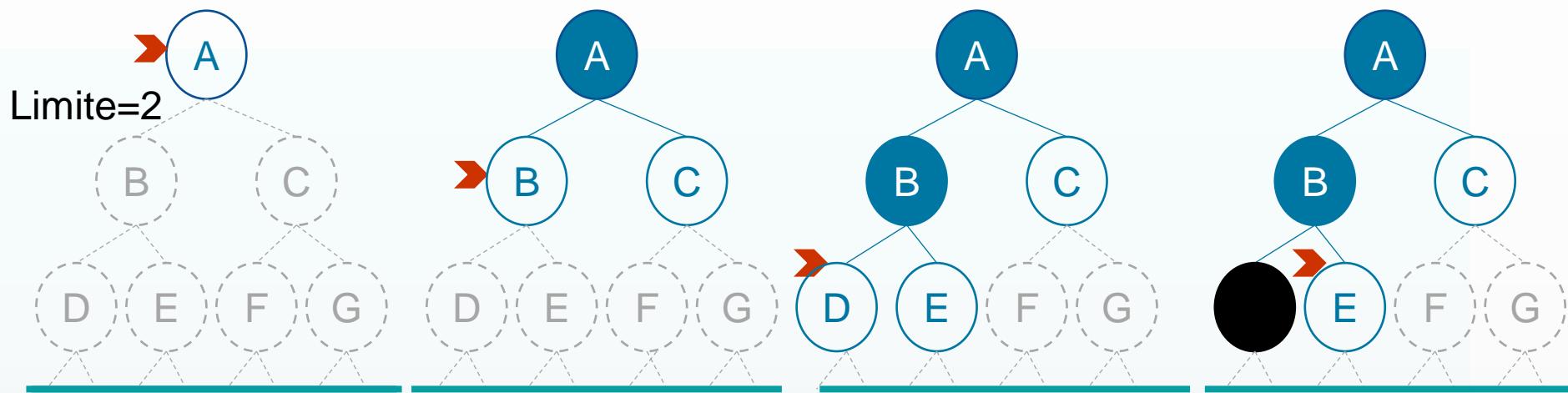
Exemple de l'exploration itérative en profondeur



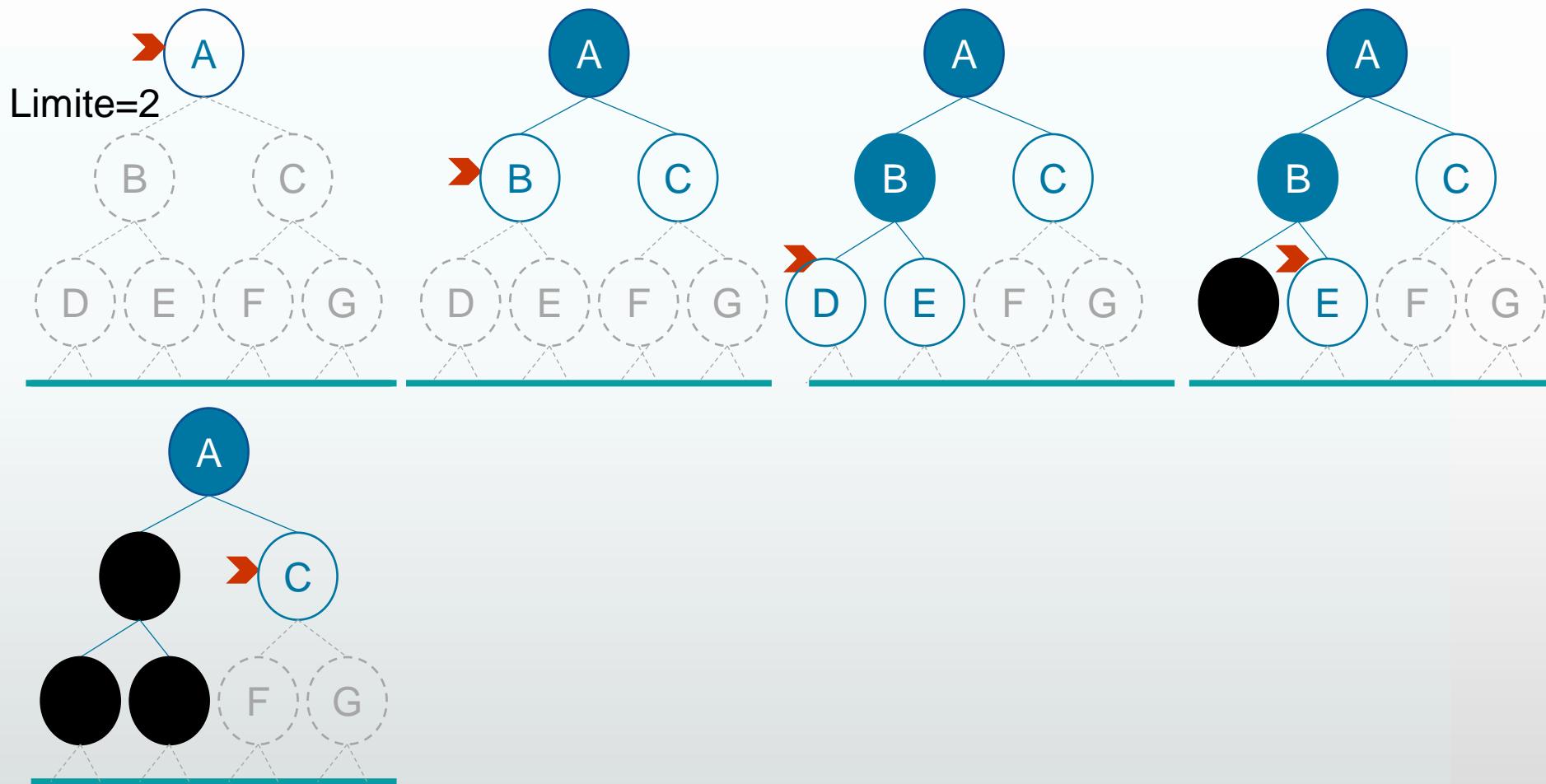
Exemple de l'exploration itérative en profondeur



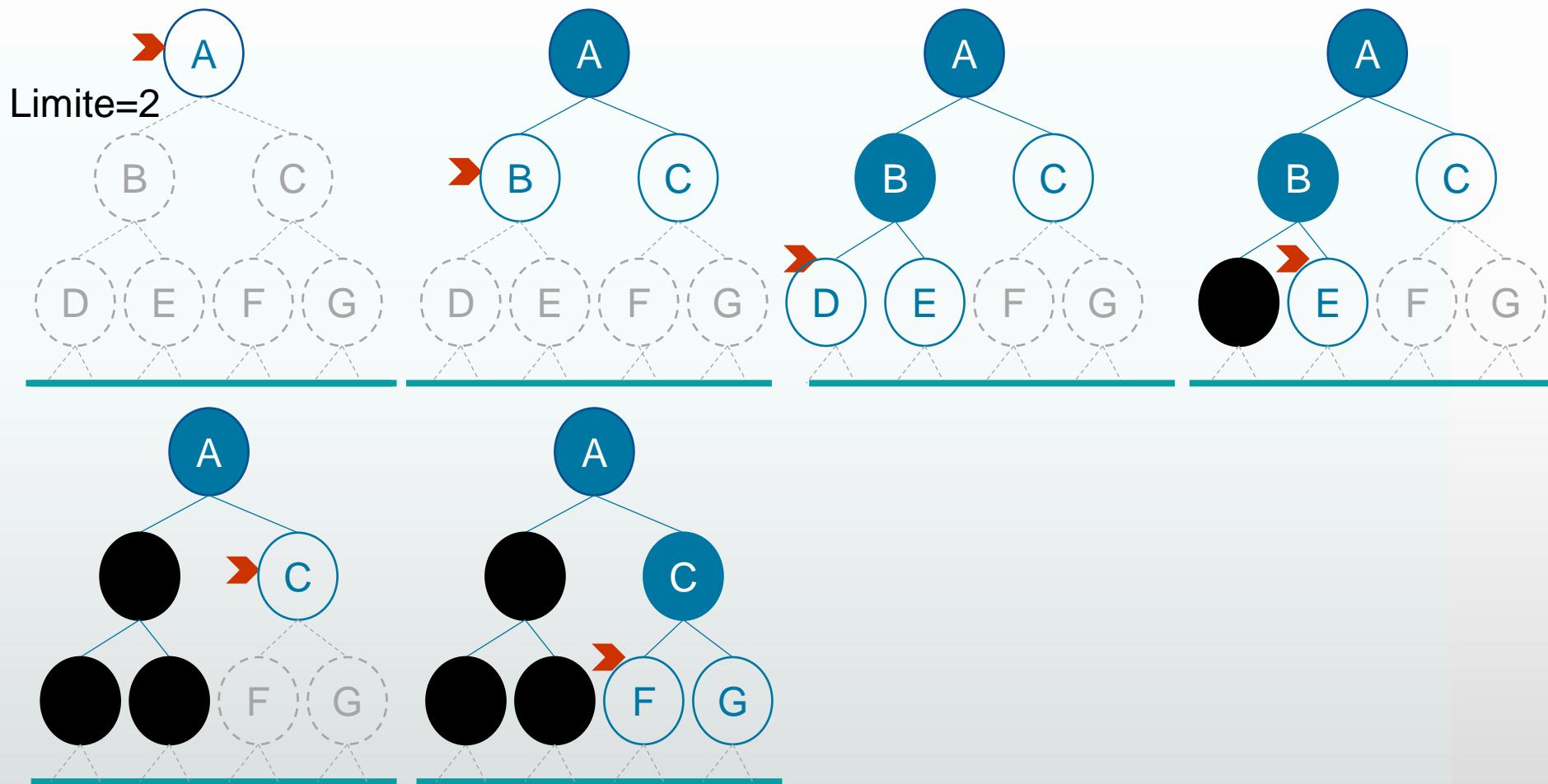
Exemple de l'exploration itérative en profondeur



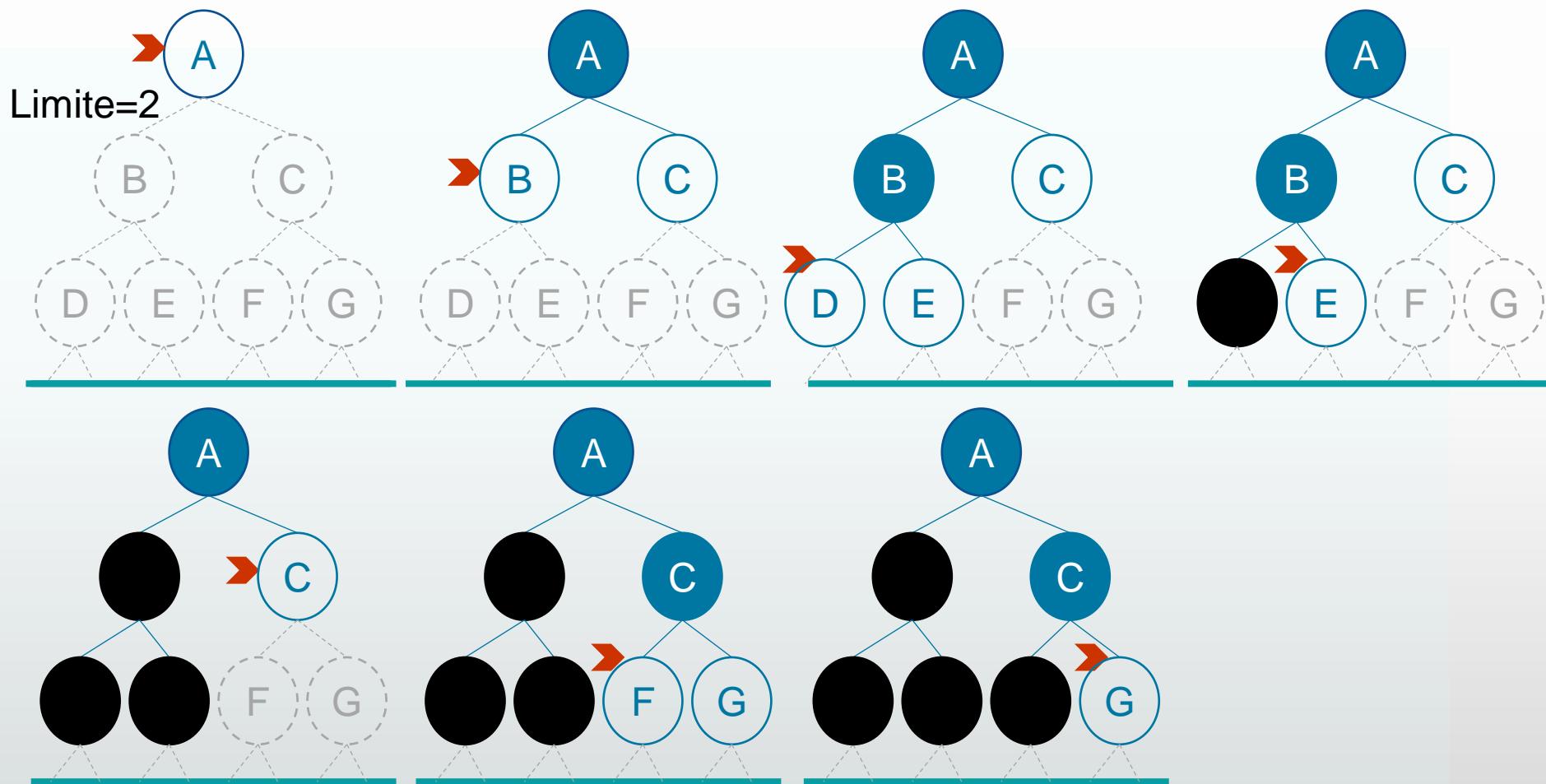
Exemple de l'exploration itérative en profondeur



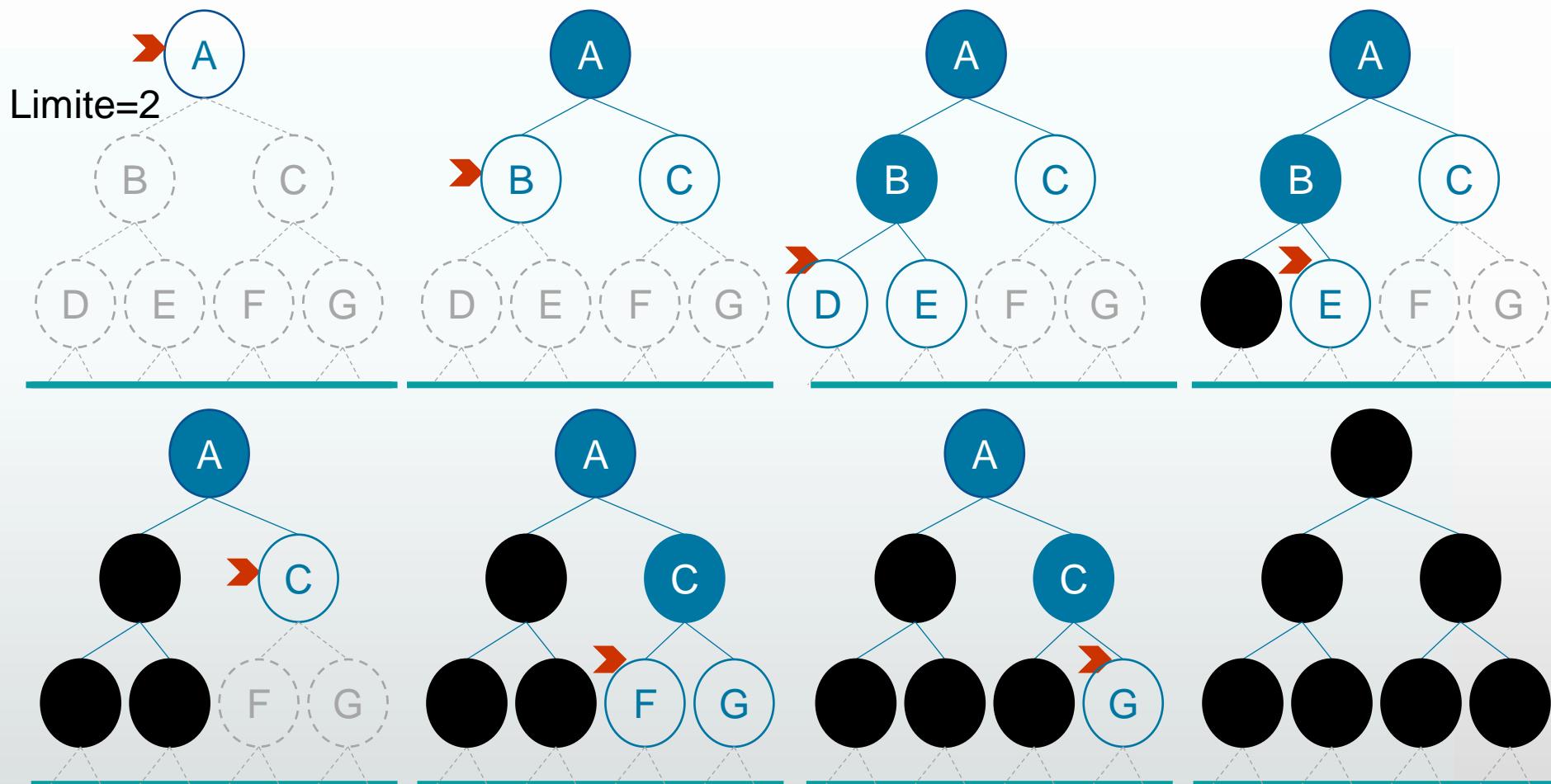
Exemple de l'exploration itérative en profondeur



Exemple de l'exploration itérative en profondeur

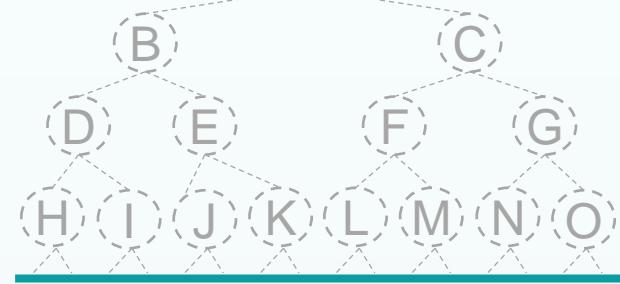


Exemple de l'exploration itérative en profondeur



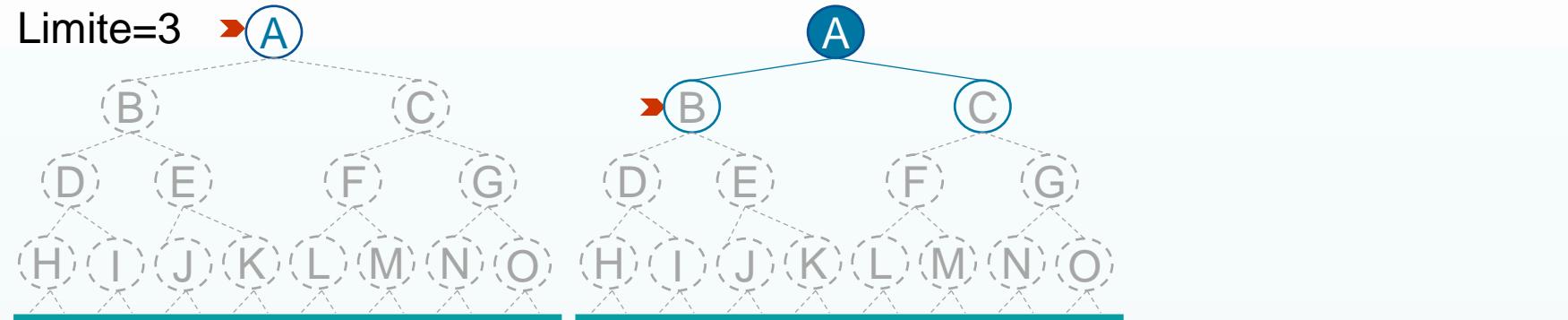
Exemple de l'exploration itérative en profondeur

Limite=3 ➔ A



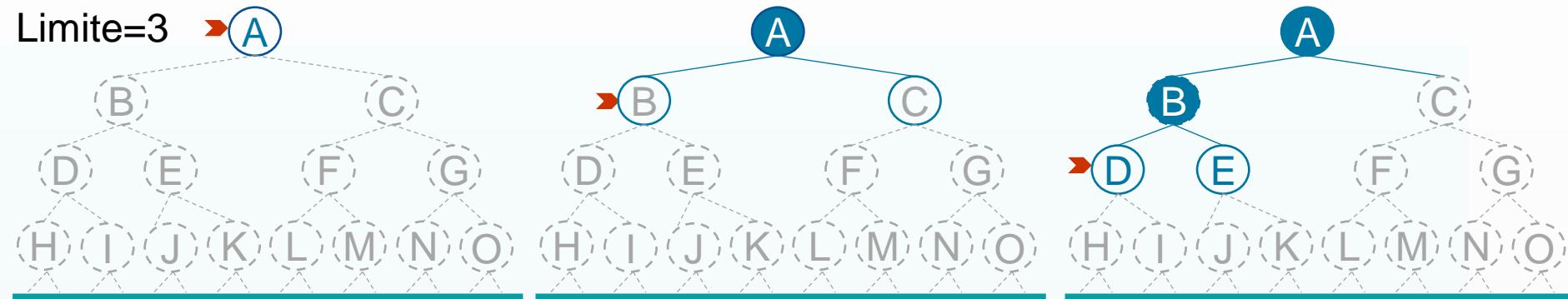
Exemple de l'exploration itérative en profondeur

Limite=3 ➔ A

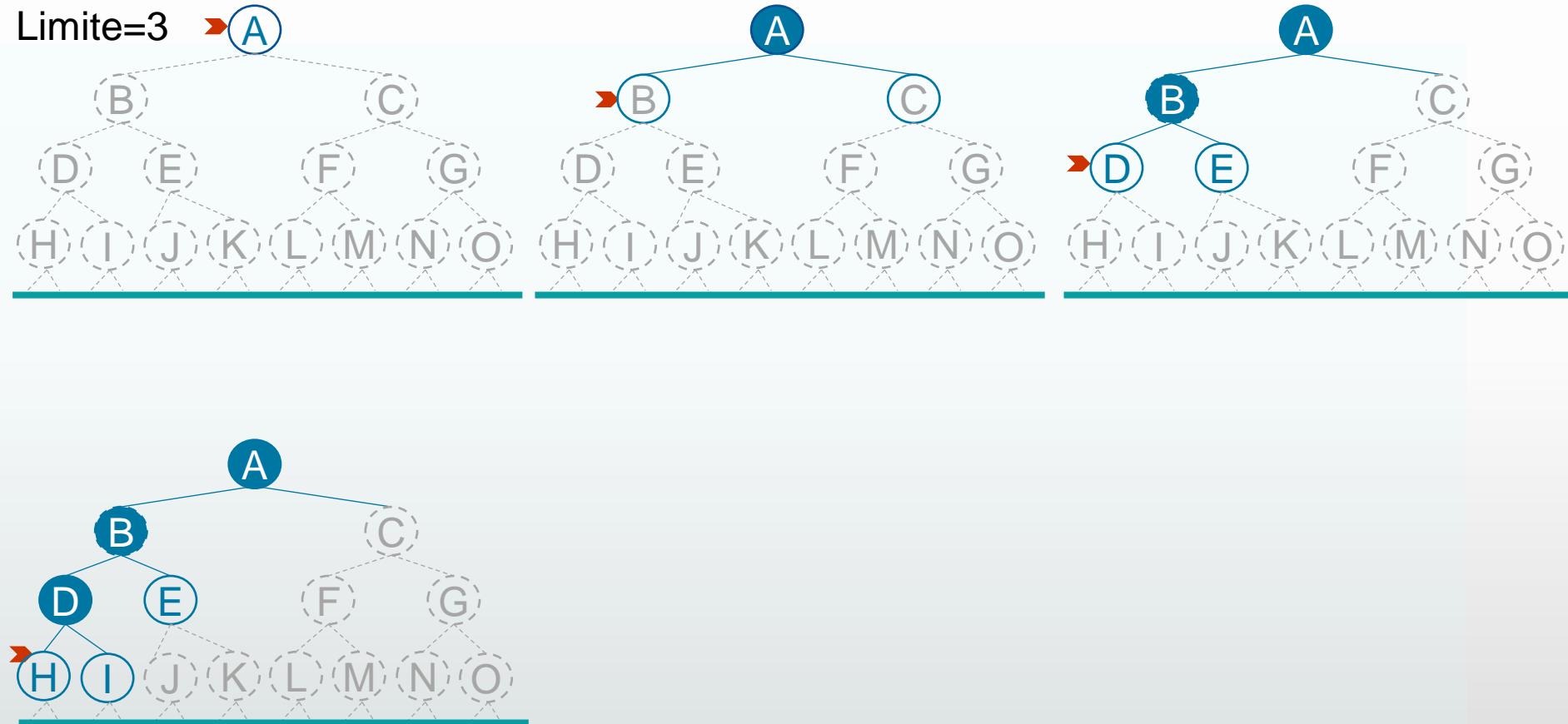


Exemple de l'exploration itérative en profondeur

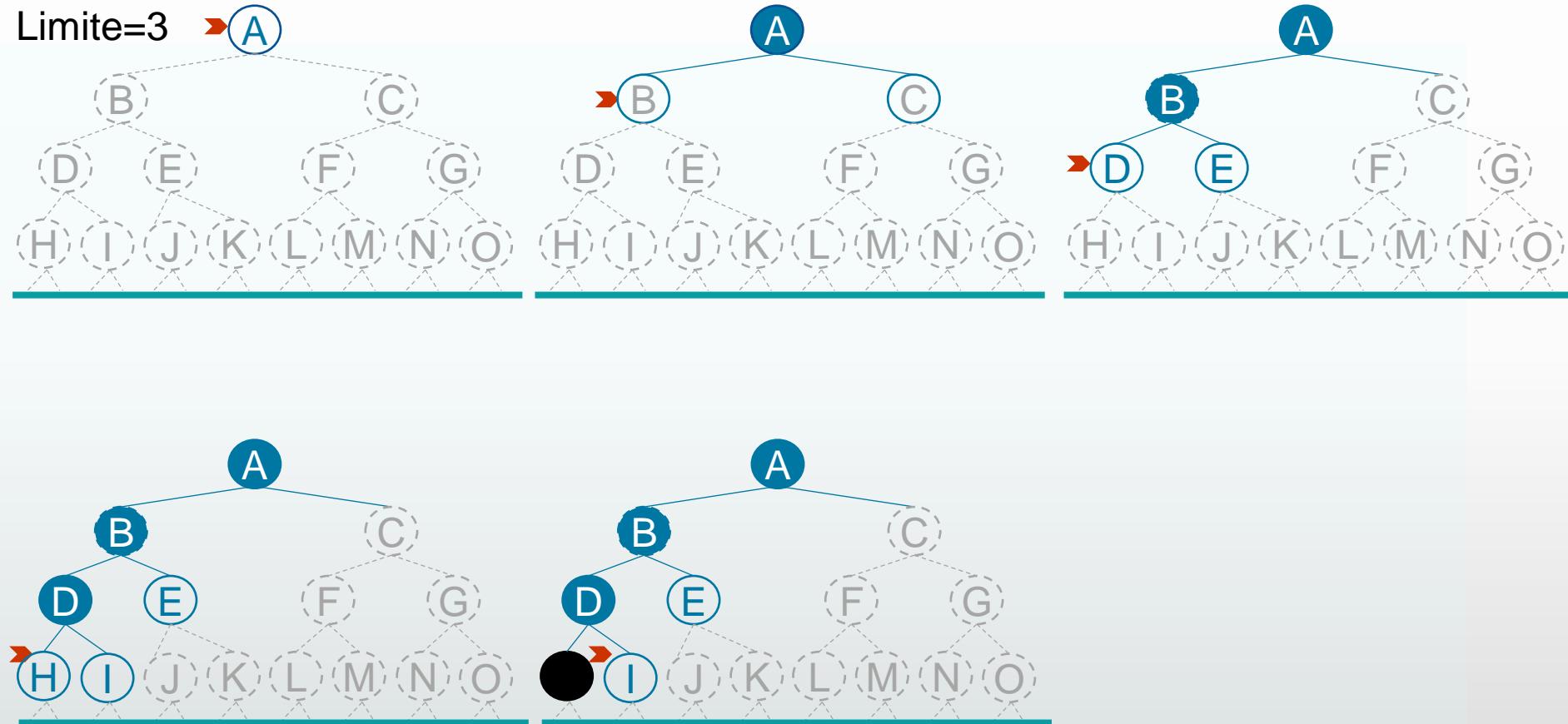
Limite=3 ➔ A



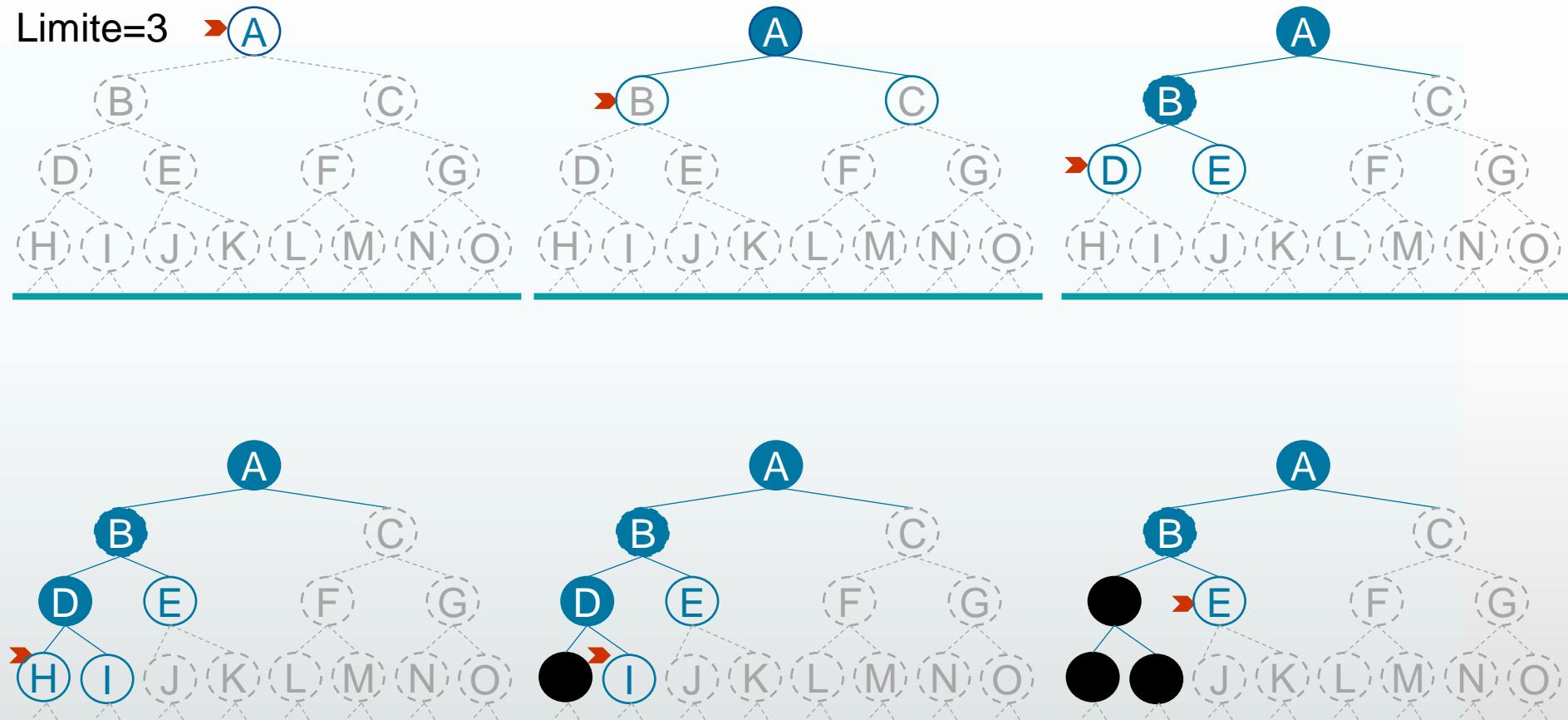
Exemple de l'exploration itérative en profondeur



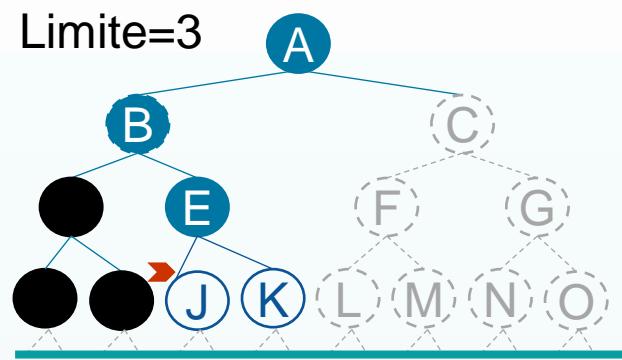
Exemple de l'exploration itérative en profondeur



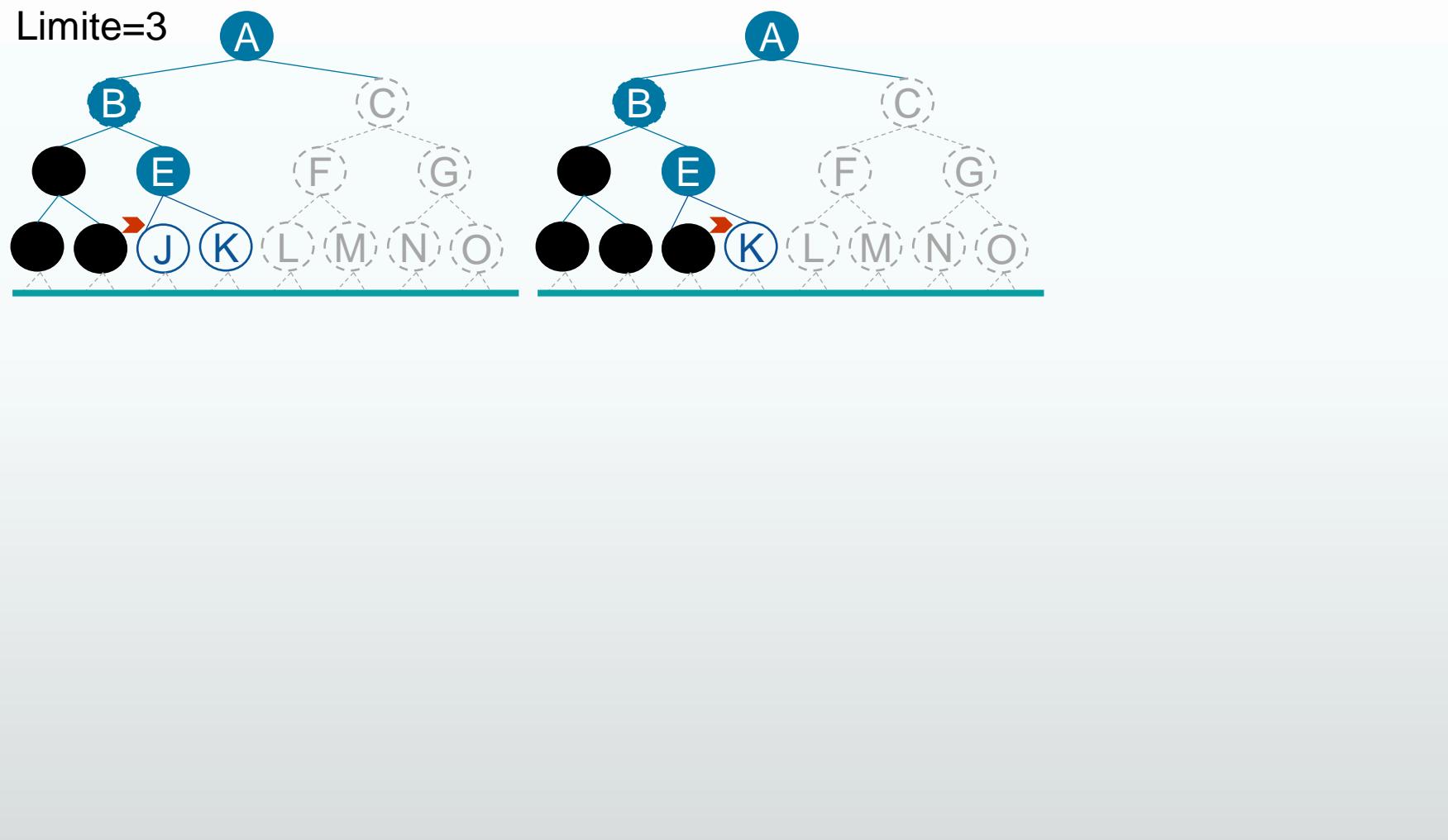
Exemple de l'exploration itérative en profondeur



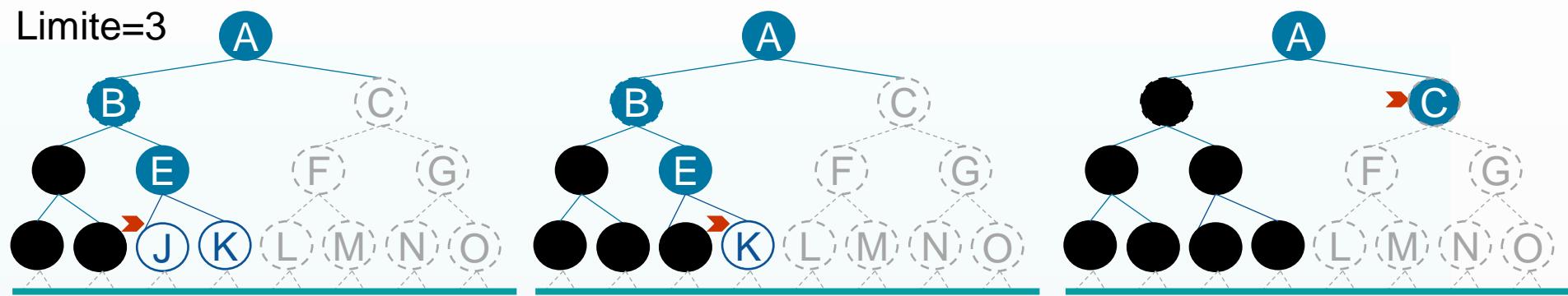
Exemple de l'exploration itérative en profondeur



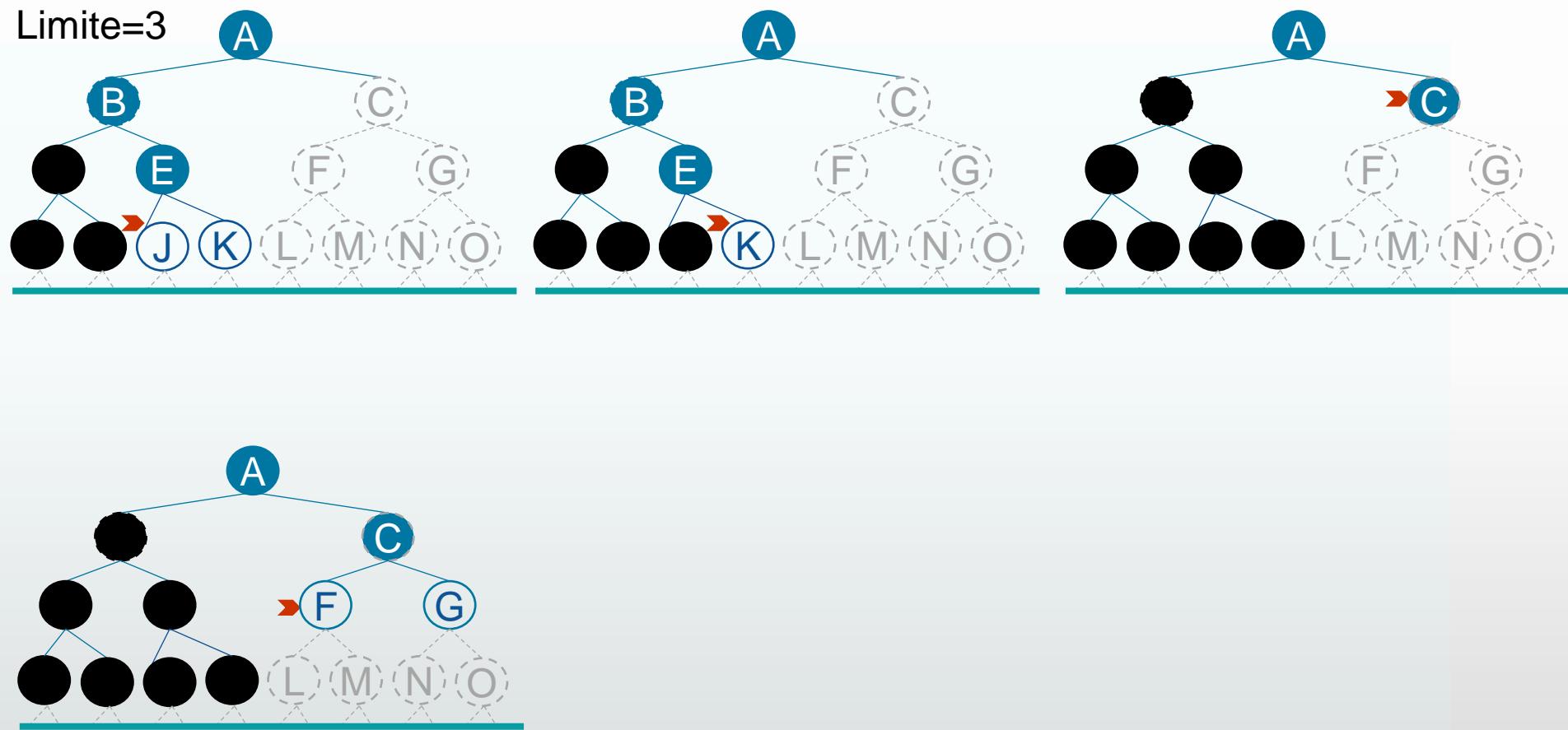
Exemple de l'exploration itérative en profondeur



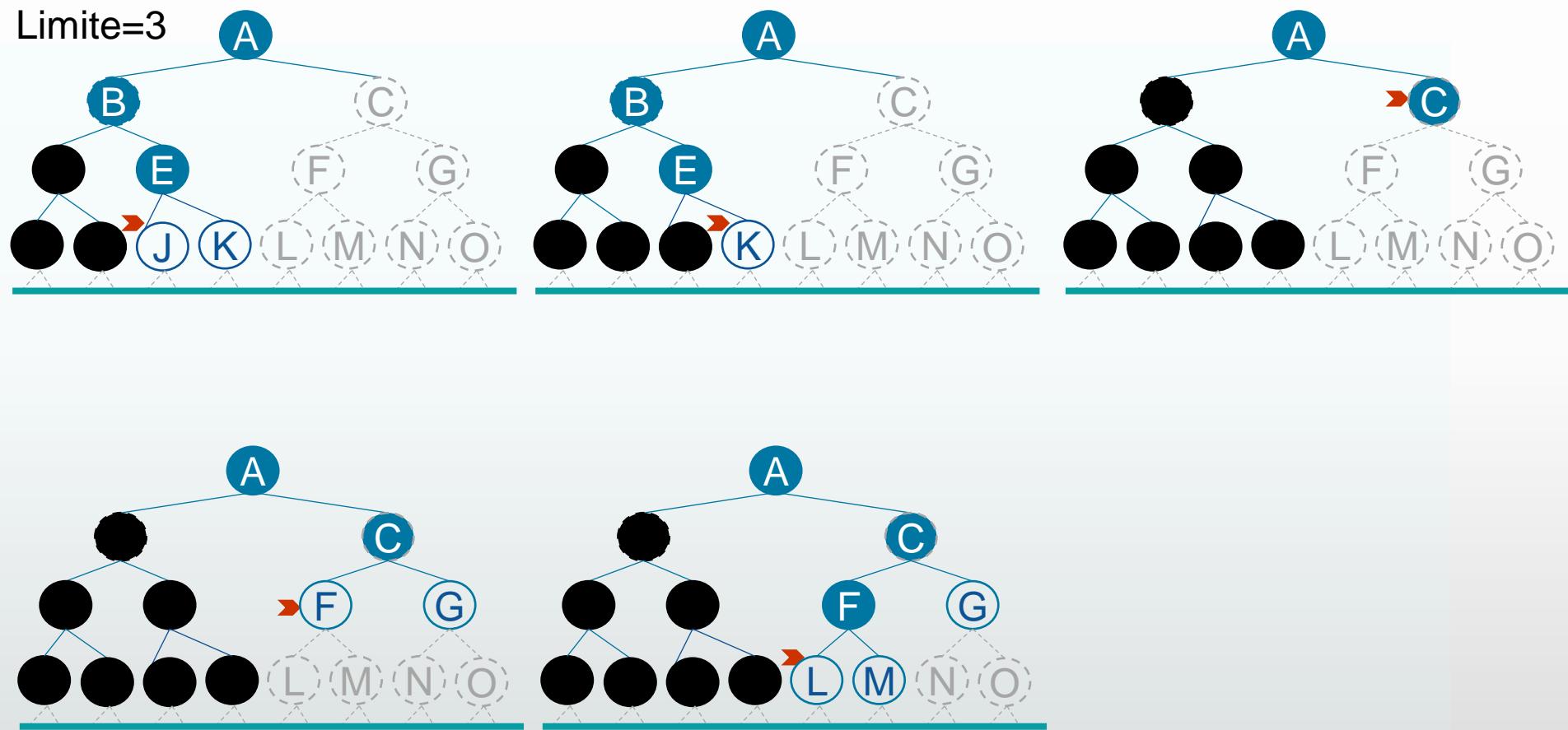
Exemple de l'exploration itérative en profondeur



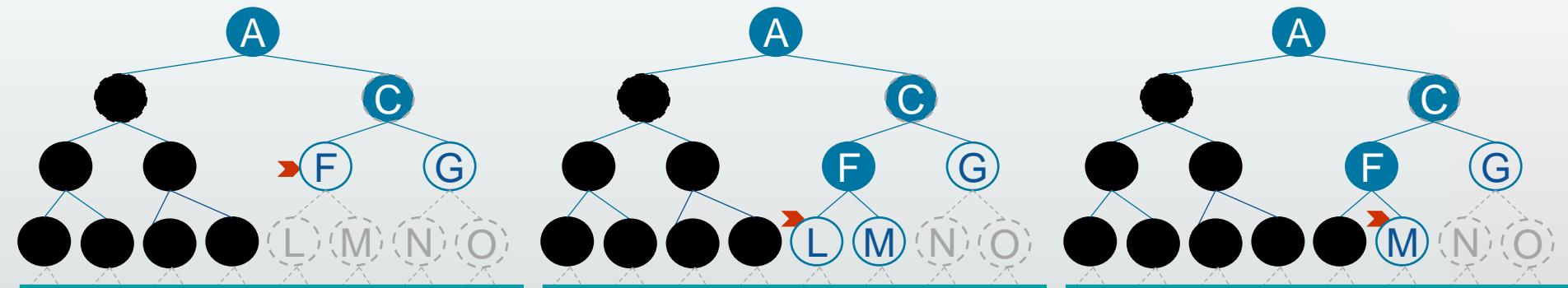
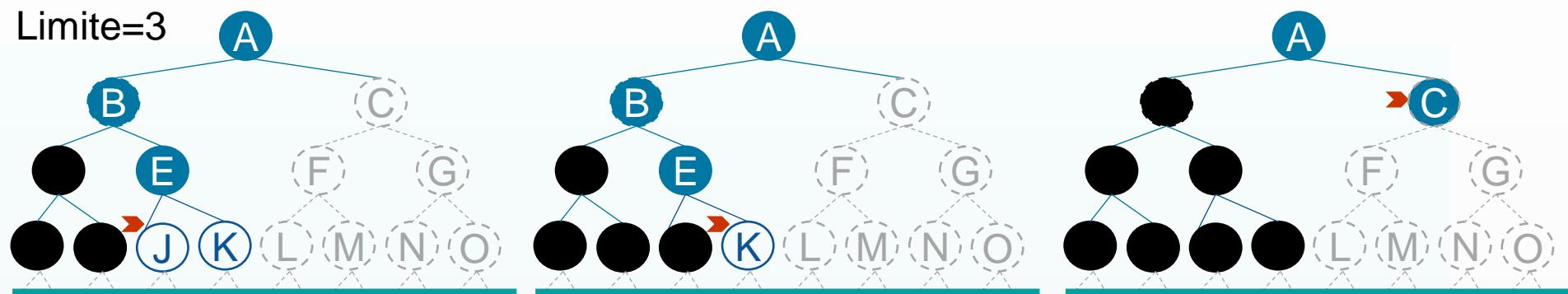
Exemple de l'exploration itérative en profondeur



Exemple de l'exploration itérative en profondeur



Exemple de l'exploration itérative en profondeur



$l = 3$

Performances

- ❑ Complétude : oui si b est fini.
- ❑ Optimalité : oui si coût est une fonction non décroissante de la profondeur du nœud.
- ❑ Complexité en temps :

$$\square (d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d).$$

	Largeur d'abord	Itérative en profondeur
$b=10$ $d=5$	$N=1+10+100+1000+10000+100000+999990=1111101$	$N=6+50+400+3000+20000+100000=123456$

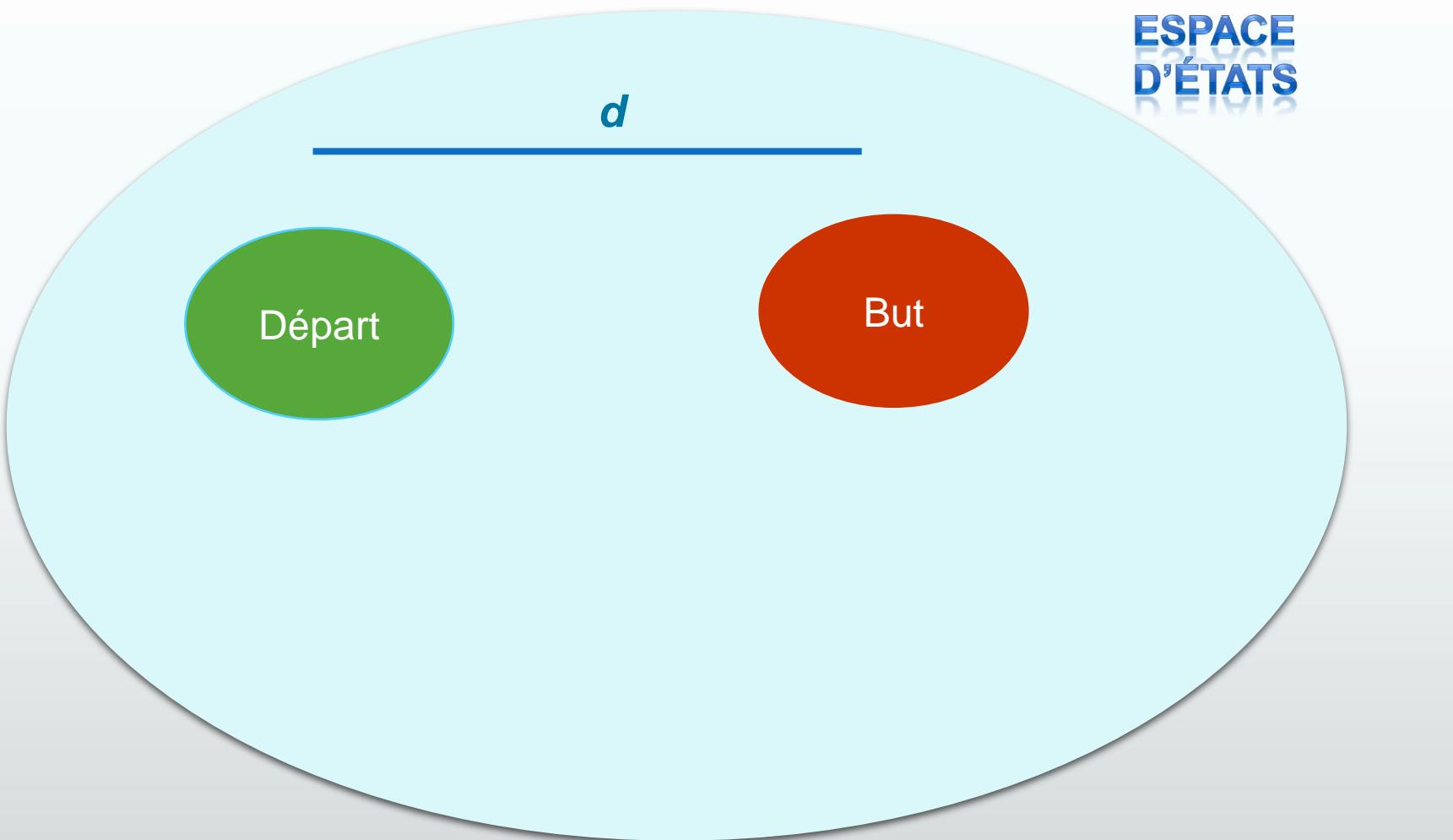
- ❑ Complexité en espace : $O(bd)$.
- ❑ Peut être modifiée pour une stratégie de coût uniforme.

Exploration bidirectionnelle

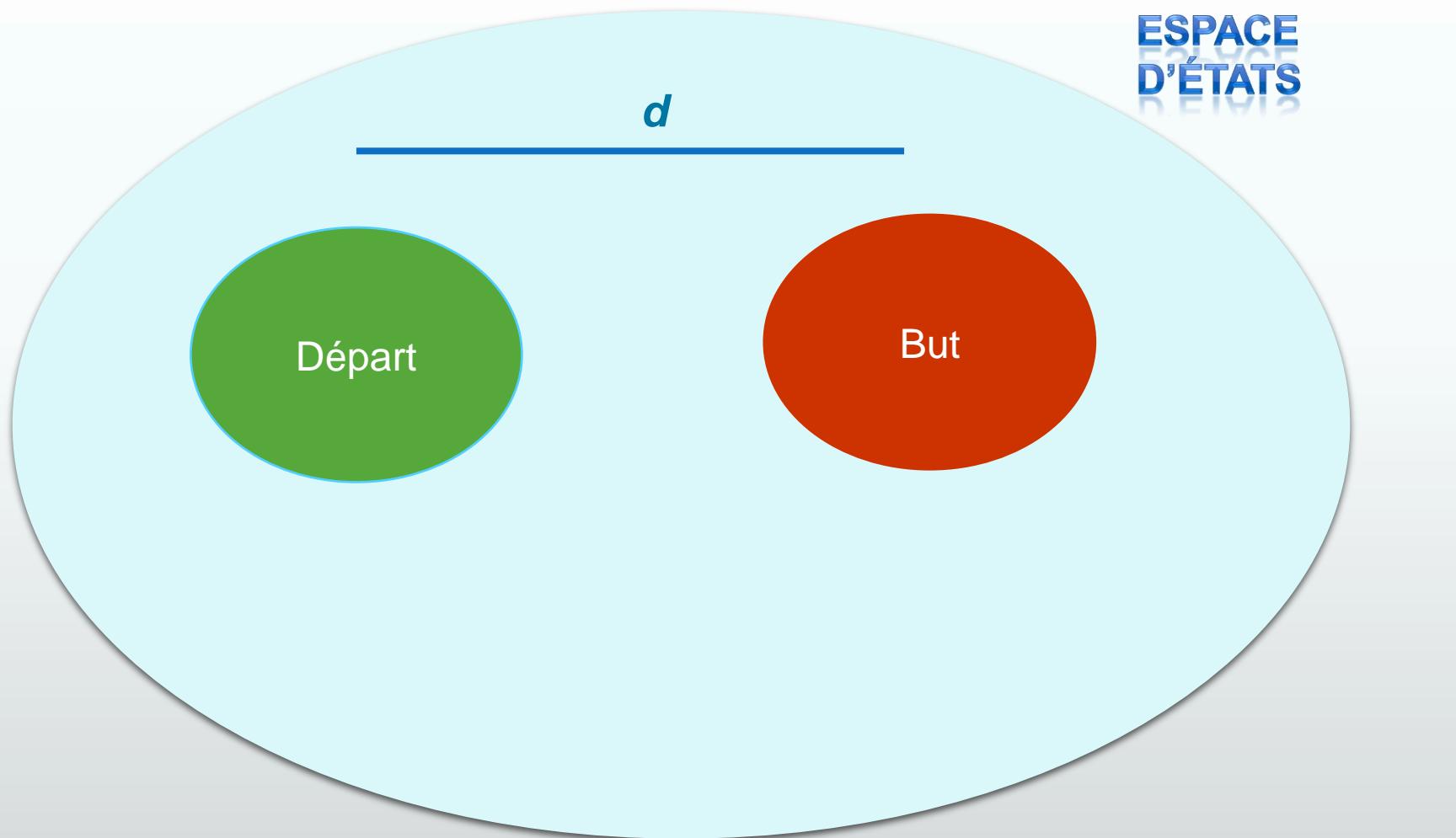
Principe de l'exploration bidirectionnelle

- Exécution de deux explorations simultanément :
 - en aval depuis l'état initial
 - en amont à partir du but
- Arrêt lorsque les deux frontières se rencontrent au milieu.
- Réduction de moitié de la profondeur explorée.
- Vérification de l'existence d'un nœud commun aux deux arbres
 - Tenir une table de hachage.
 - Conserver tous les nœuds d'un des arbres.
- Difficile à appliquer si le but est une description abstraite (n reines).

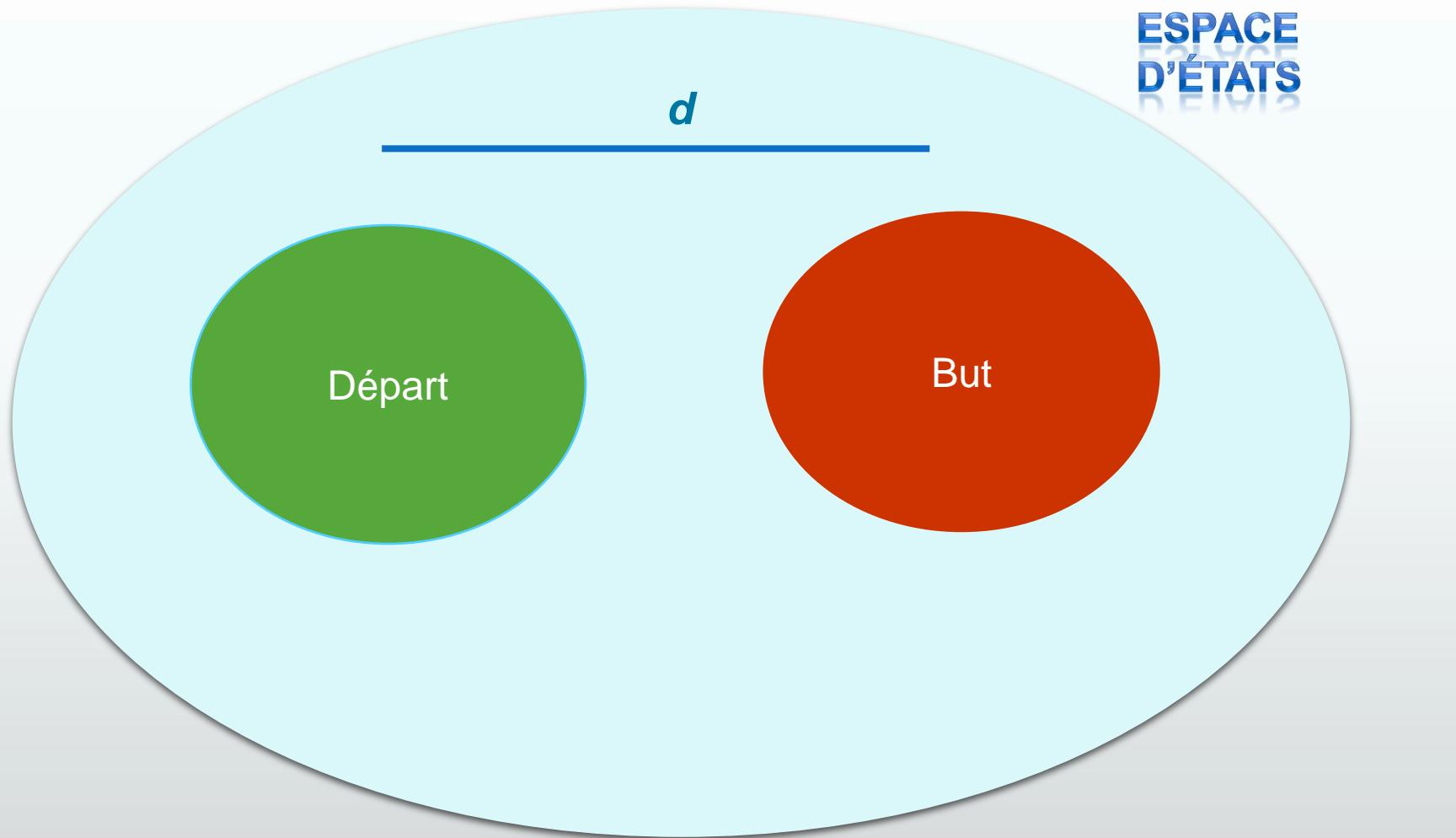
Schématisation de l'exploration bidirectionnelle



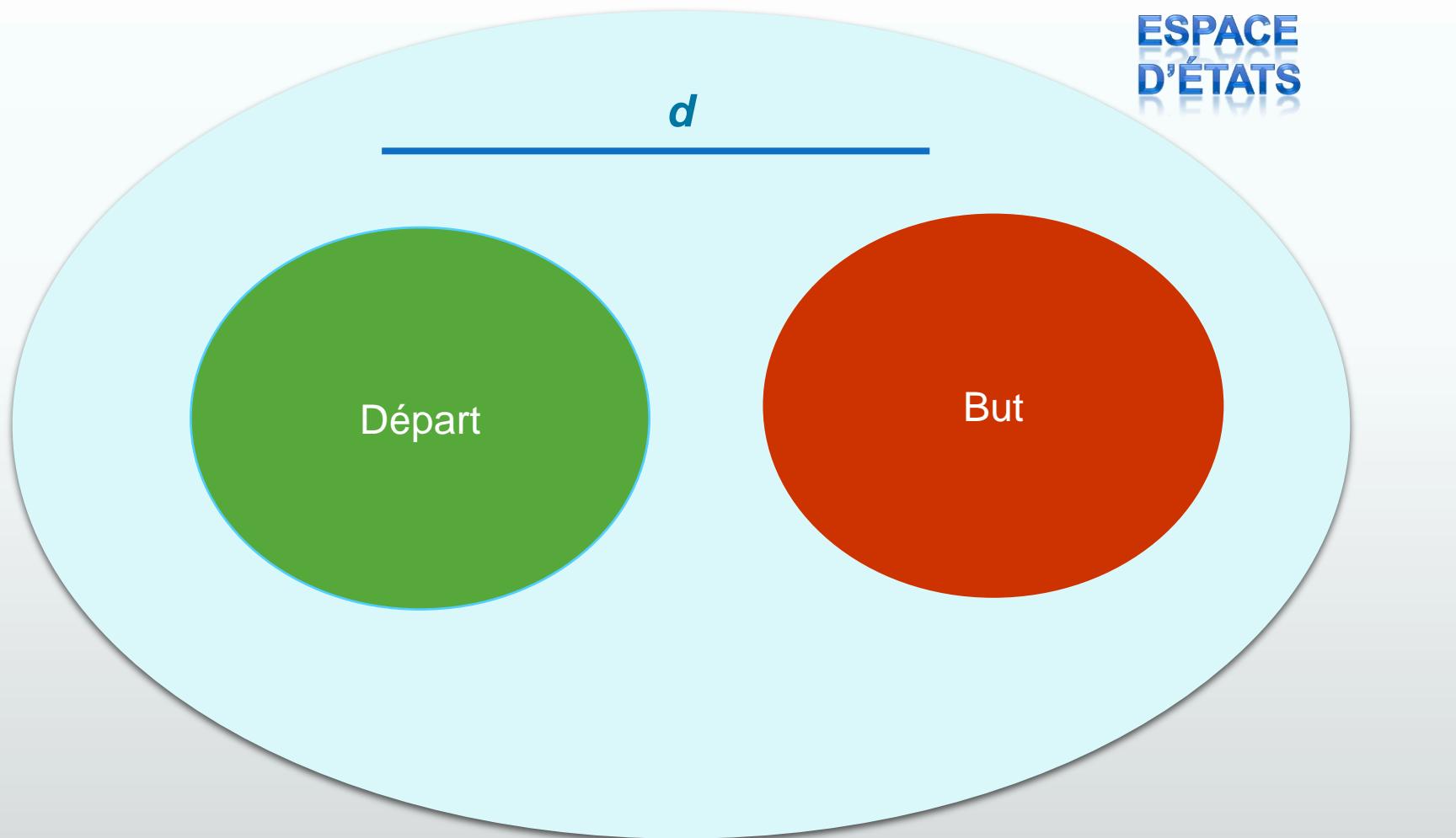
Schématisation de l'exploration bidirectionnelle



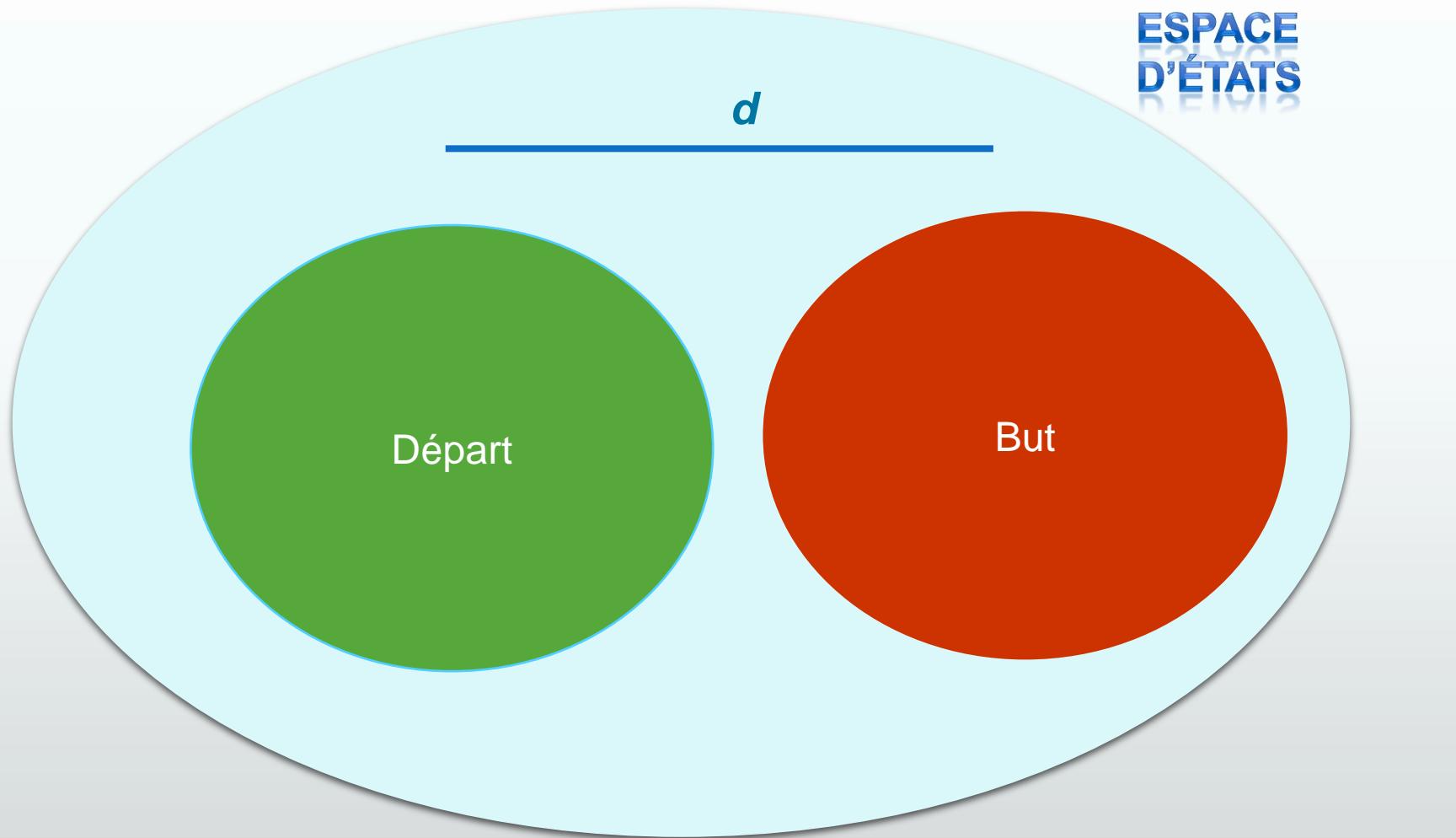
Schématisation de l'exploration bidirectionnelle



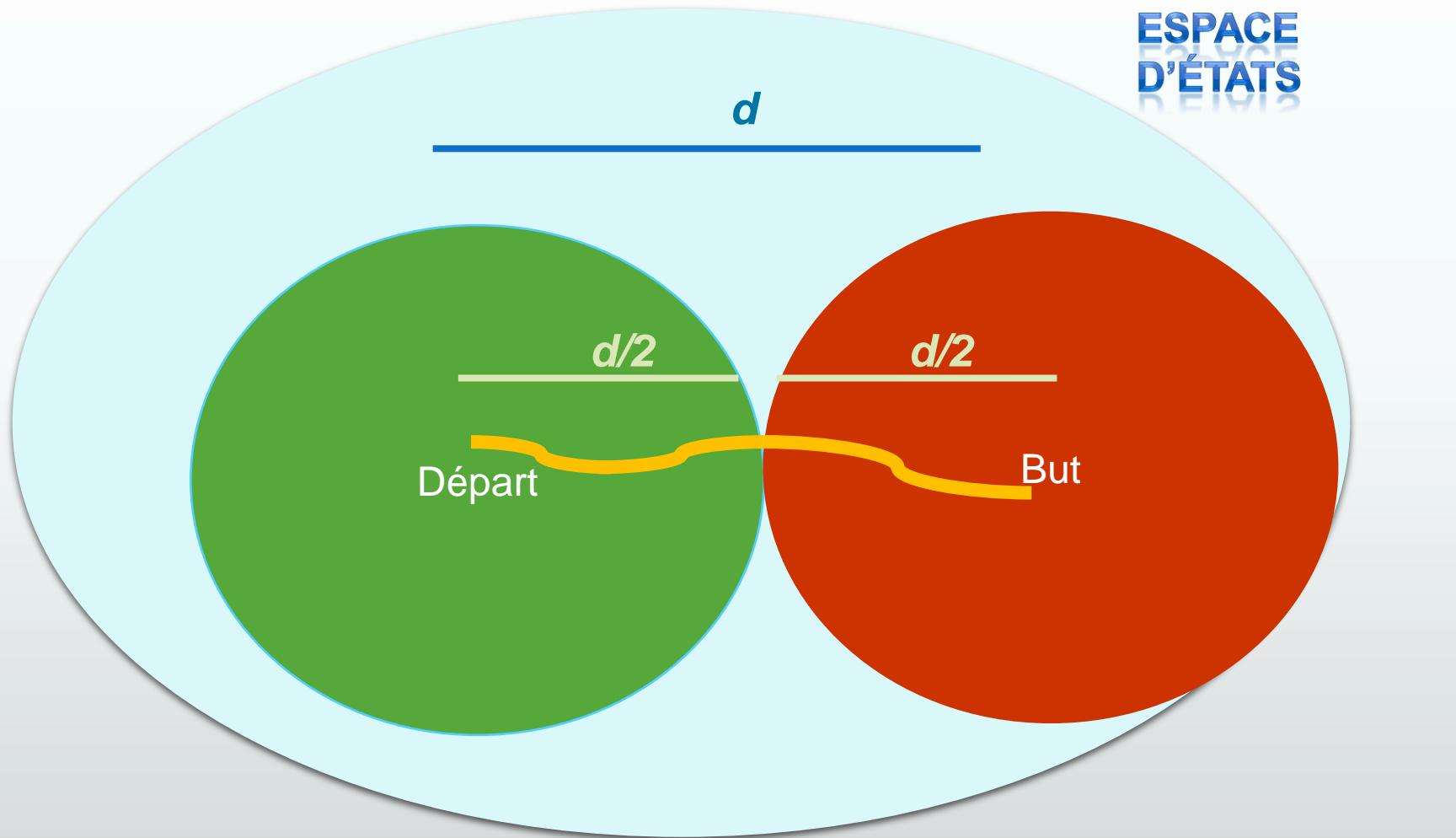
Schématisation de l'exploration bidirectionnelle



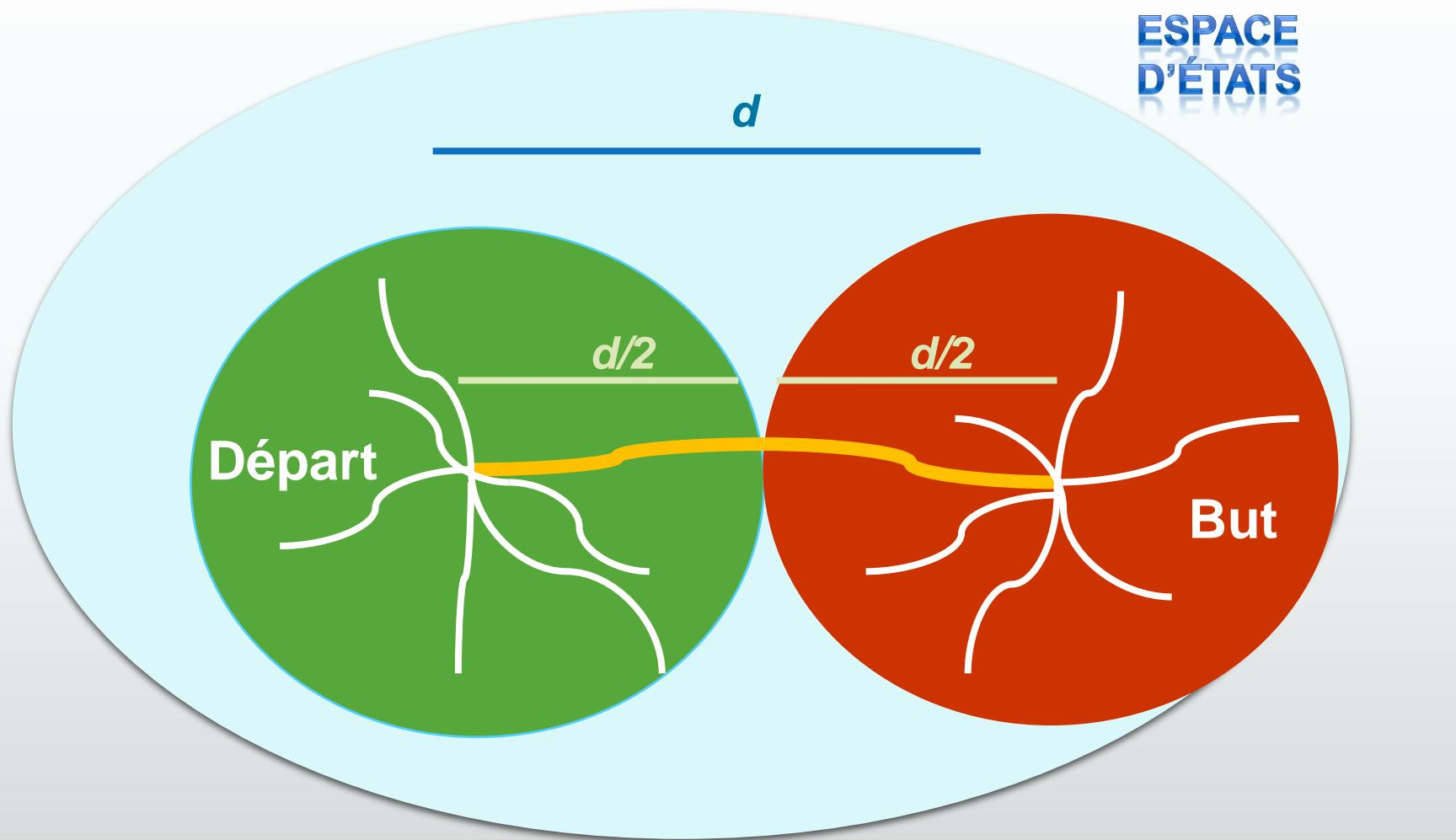
Schématisation de l'exploration bidirectionnelle



Schématisation de l'exploration bidirectionnelle



Schématisation de l'exploration bidirectionnelle



Performances

- ❑ Complétude : oui.
- ❑ Optimalité : oui si les coûts des étapes sont identiques et les deux directions utilisent une exploration en largeur d'abord.
- ❑ Complexité en temps : $O(b^{d/2})$.
- ❑ Complexité en espace : $O(b^{d/2})$.

Comparaison des stratégies d'exploration d'arbres

Critère	Largeur d'abord	Coût uniforme	Profondeur d'abord	Profondeur limitée	Profondeur itérative	Bidirectionnelle (si applicable)
Complète?	Oui ^a	Oui ^{a,b}	Non	Non	Oui ^a	Oui ^{a,d}
Temps	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espace	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Oui ^c	Oui	Non	Non	Oui ^c	Oui ^{c,d}

b : facteur de branchement

d : profondeur de la solution la moins profonde

m : profondeur maximale de l'arbre d'exploration

l : profondeur limite

^a si b est fini

^b si les coûts des étapes sont $\geq \epsilon$ avec ϵ positif

^c si les coûts d'étapes sont tous identiques

^d si les deux directions utilisent une exploration en largeur d'abord

Comparaison des stratégies d'exploration de graphes

- Même tableau sauf :
 - La recherche en profondeur d'abord est complète pour les espaces d'états finis.
 - Les complexités temporelle et spatiale sont limitées par la taille de l'espace d'états.

Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ **Stratégies d'explorations :**
 - ❑ Non informées,
 - ❑ **Informées (heuristiques)**
- ❑ Fonctions heuristiques

Stratégies d'exploration informées (heuristiques)

Pourquoi une exploration informée ?

- Stratégies d'exploration non informées sont généralement très peu efficaces.
- Stratégies d'exploration informées :
 - Utilisent des connaissances du problème : une fonction **heuristique**.
 - Plus efficaces que l'exploration aveugle.

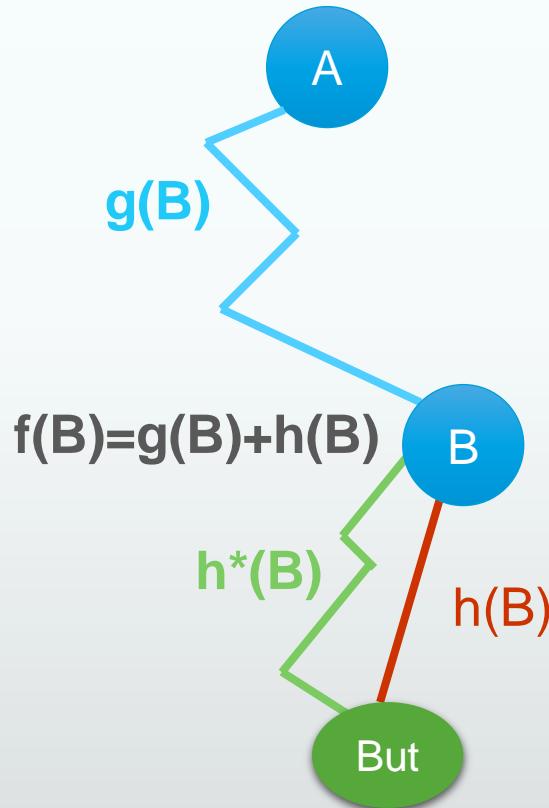
Stratégies d'exploration informée

□ Exploration par la meilleur d'abord

Stratégie d'exploration par le meilleur d'abord

- ❑ Basée sur l'exploration en arbre et en graphe.
- ❑ Le nœud à développer est choisi selon une fonction d'évaluation $f(n)$:
 - ❑ La fonction $f(n)$ est une estimation du coût.
 - ❑ Le nœud qui a un $f(n)$ le plus faible est développé en premier.
 - ❑ Comme l'exploration à coût uniforme mais f est utilisée au lieu de g pour ordonner la file.

Fonction d'évaluation $f(n)$

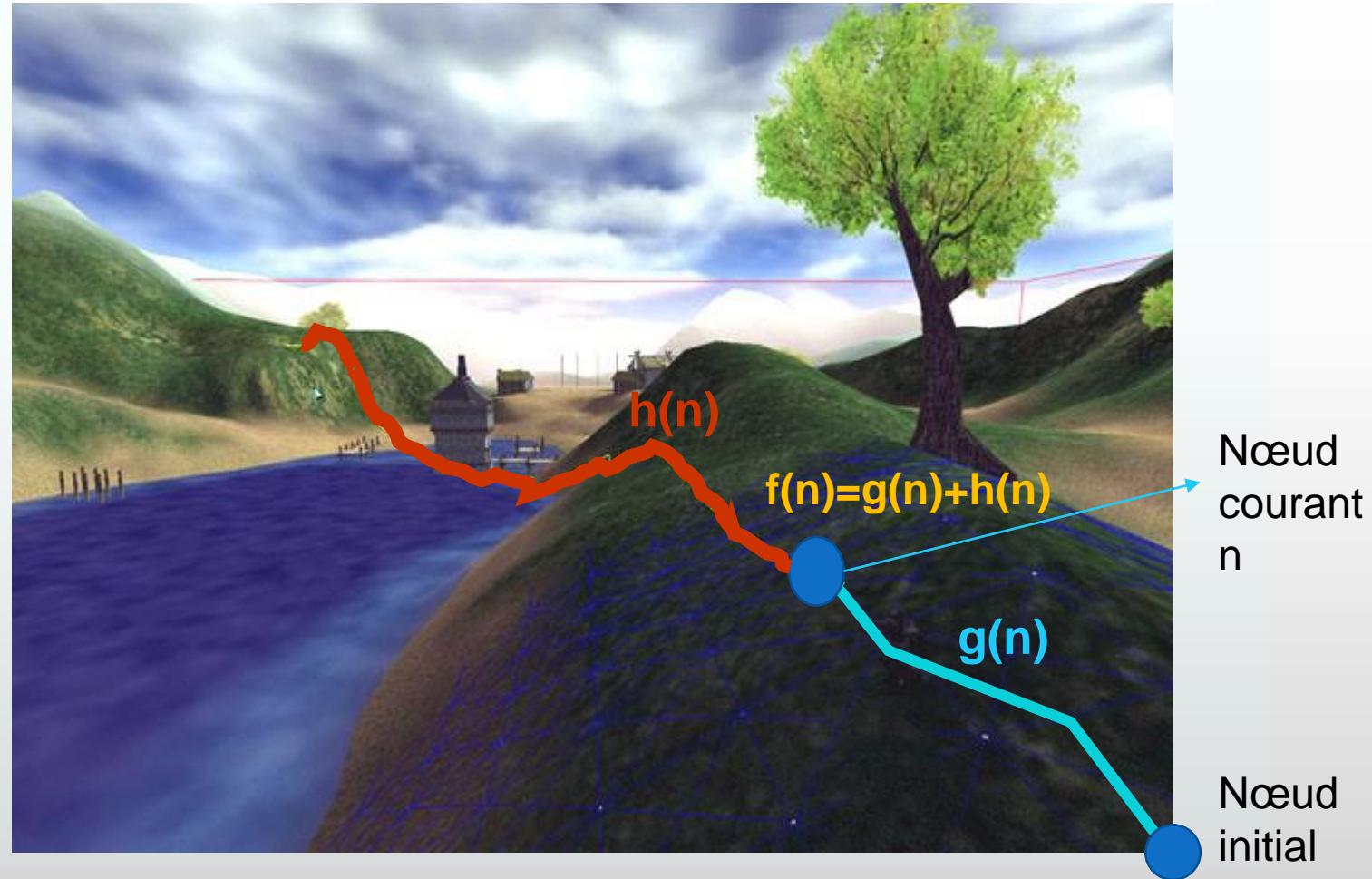


- $g(n)$: *coût réel* (somme des meilleurs coûts du nœud *initial* à n).
- $h^*(n)$: *coût réel* (somme des coûts du nœud n au nœud *but*).
- $h(n)$: *coût heuristique* (estimation du chemin le moins coûteux de l'état du nœud n à nœud *but*).

Fonction d'évaluation $f(n)$

$h(n)$:
heuristique
distance à vol
de oiseau

Le prochain
nœud est choisi
selon la valeur
de $f(n)$



Types d'exploration par le meilleur d'abord

- Exploration gloutonne par le meilleur d'abord.
- Exploration A*.

Exploration gloutonne par le meilleur d'abord

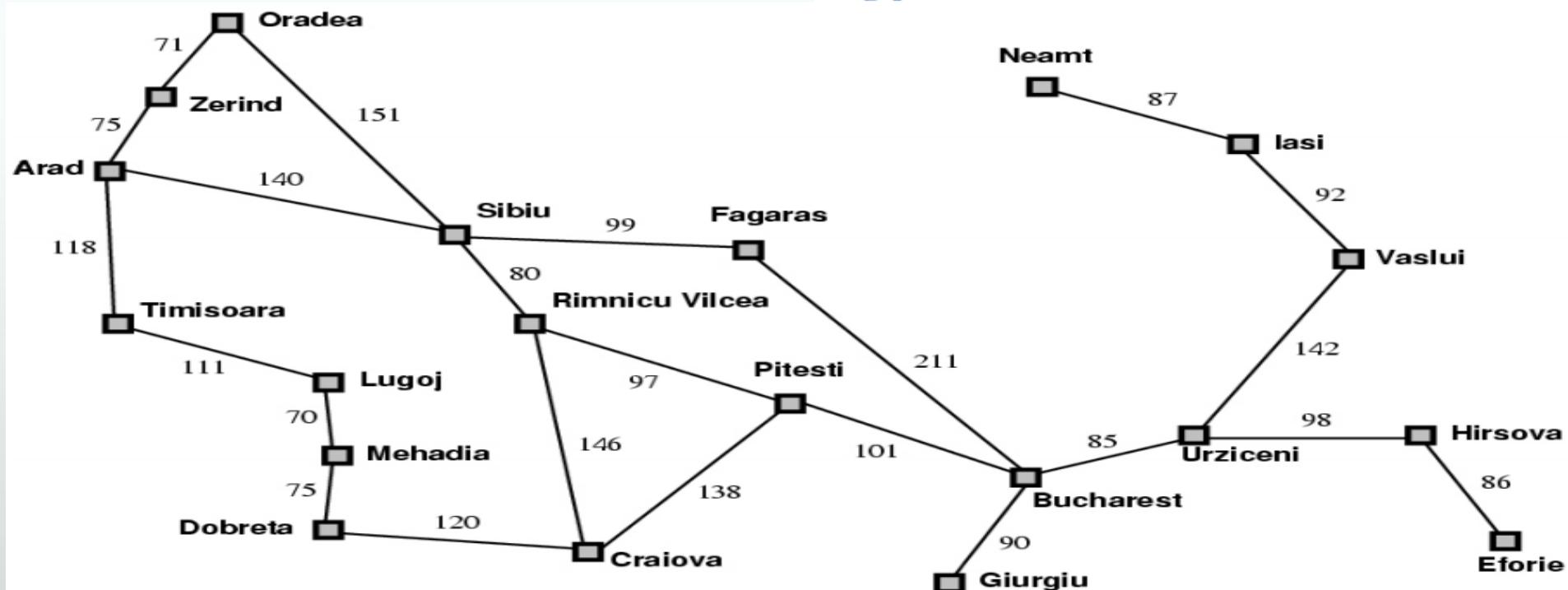
Principe de l'exploration gloutonne *meilleur d'abord*

- Développe le nœud le plus proche du but (*glouton*).
- $f(n) = h(n)$.
- Exemple du voyageur en Roumanie :
 - Heuristique h_{SLD} : distance à vol d'oiseau (Straight-line distance SLD).

Heuristique h_{SLD}

h_{SLD} distance à vol d'oiseau
jusqu'à Bucarest

Arad	366	Mehadia	241
Bucarest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	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

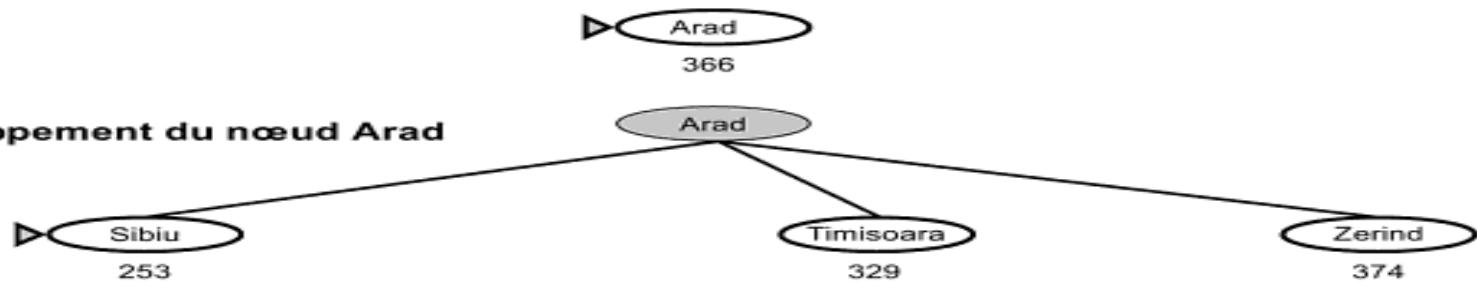


Exploration d'arbre gloutonne par le meilleur d'abord

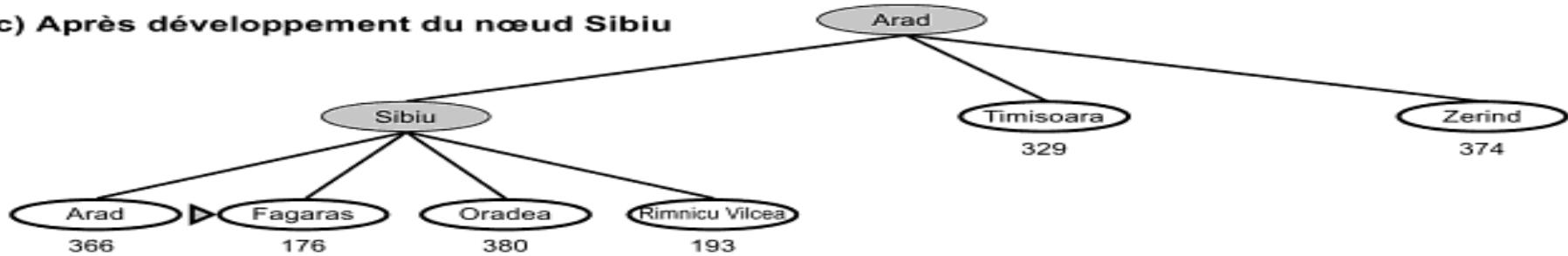
(a) L'état initial



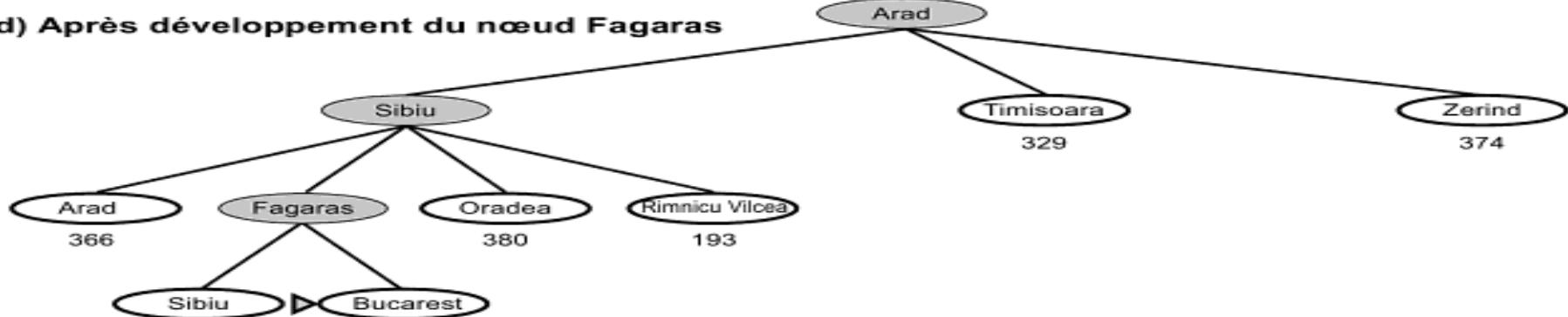
(b) Après développement du nœud Arad



(c) Après développement du nœud Sibiu



(d) Après développement du nœud Fagaras



Performances de l'exploration gloutonne par le meilleur d'abord

- Complétude :
 - Non pour l'exploration d'arbres (exemple chemin entre Iasi et Fagaras), possibilité de boucle.
 - Oui pour l'exploration de graphes, mais l'espace des états doit être fini.
- Optimalité : non, l'optimisation est locale (le chemin Arad-Sibiu-Rimnicu Vilcea-Pietesti-Bucarest est meilleur).
- Complexité en temps : $O(b^m)$ en arbre, mais peut être améliorée par le choix d'une bonne fonction heuristique.
- Complexité d'espace : $O(b^m)$, elle garde tous les nœuds en mémoire.

Exploration A* : A étoile

Principe de l'exploration A*

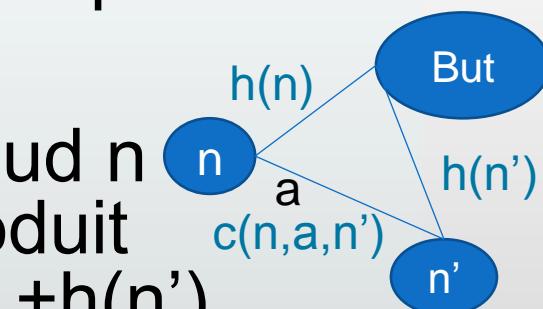
□ Fonction d'évaluation f :

$$\square f(n) = g(n) + h(n).$$

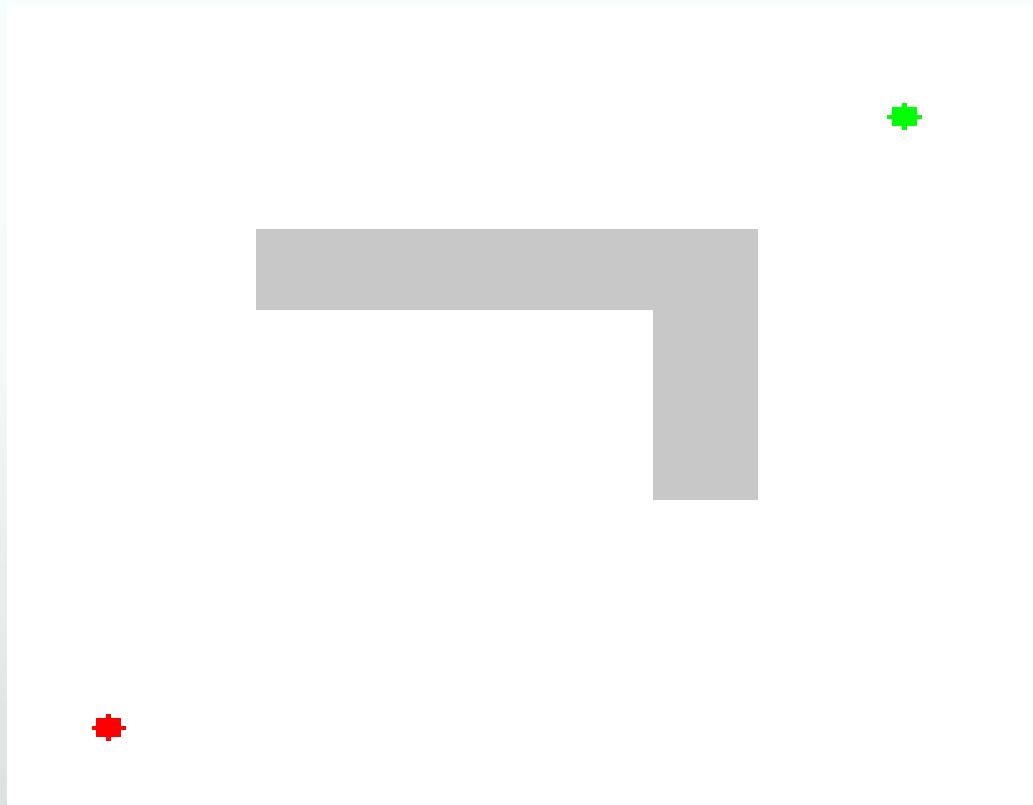
□ $f(n)$ est le coût estimé de la solution la moins couteuse passant par n .

Conditions d'optimalité de A*

- A* doit utiliser une **heuristique $h(n)$** **admissible**.
- Une heuristique h est admissible si *elle ne surestime jamais le coût réel* pour atteindre le but : $h(n) \leq h^*(n)$
- **Consistance** (monotonie) : pour l'exploration de graphes :
 - $h(n)$ est consistante si pour nœud n chaque successeur n' de n , produit par une action a , $h(n) \leq c(n,a,n') + h(n')$



A* : illustration

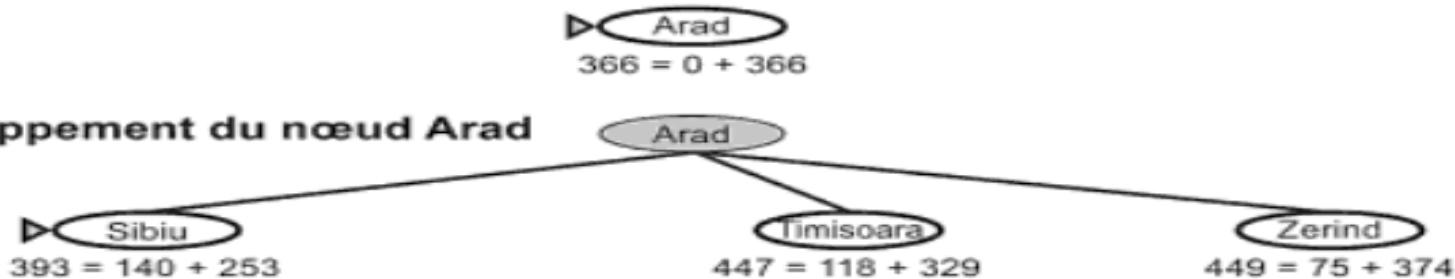


Etapes pour une exploration A* vers Bucarest

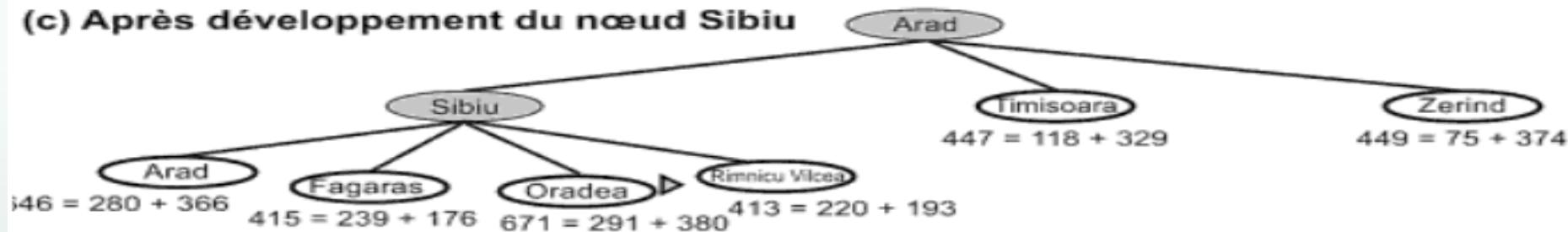
(a) L'état initial



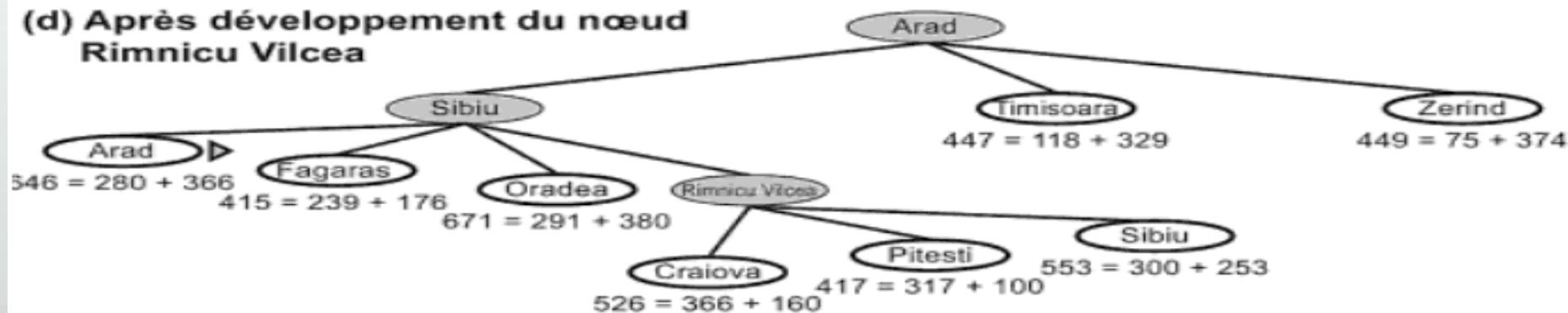
(b) Après développement du nœud Arad



(c) Après développement du nœud Sibiu

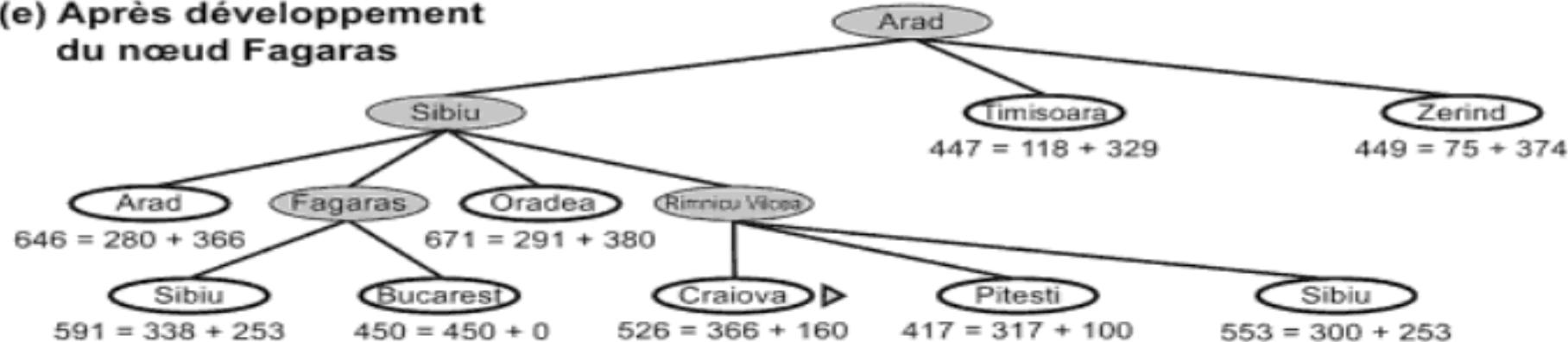


(d) Après développement du nœud Rimnicu Vilcea

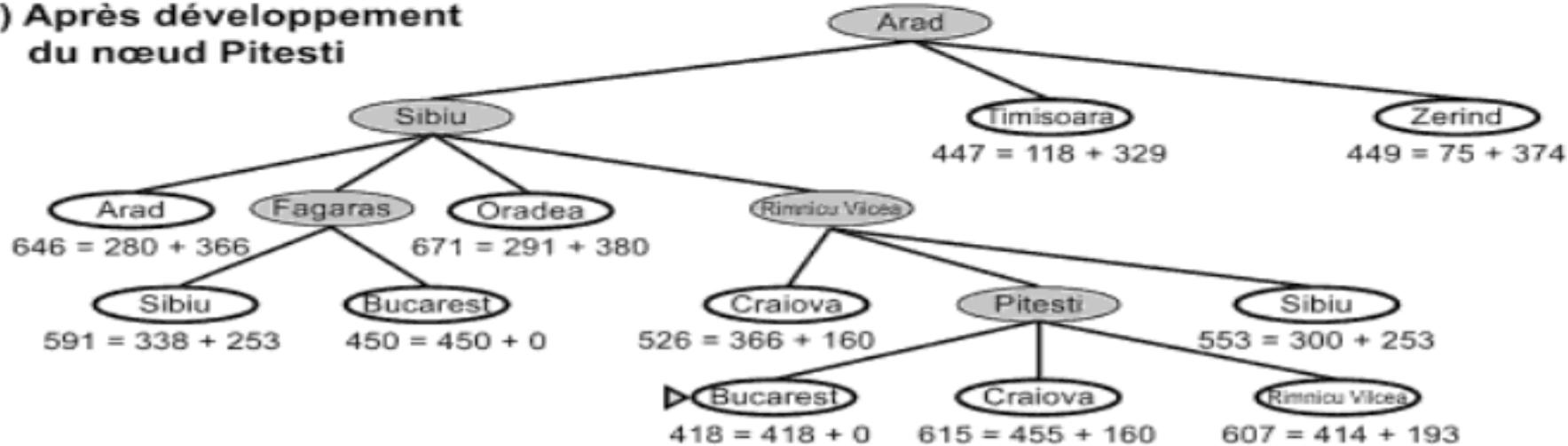


Etapes pour une exploration A* vers Bucarest

(e) Après développement du nœud Fagaras



(f) Après développement du nœud Pitesti



Performances de A*

- ❑ Complétude : oui si b est fini et le coût des actions est supérieur à une constante positive.
- ❑ Optimale :
 - ❑ Arbre : oui si $h(n)$ est admissible.
 - ❑ Graphe : oui si $h(n)$ est consistante.
- ❑ Complexité de temps : exponentielle, selon la longueur de la solution optimale.
- ❑ Complexité en espace : exponentielle, selon la longueur de la solution optimale , elle garde tous les nœuds en mémoire.
- ❑ Habituellement, on manque d'espace bien avant de manquer de temps.

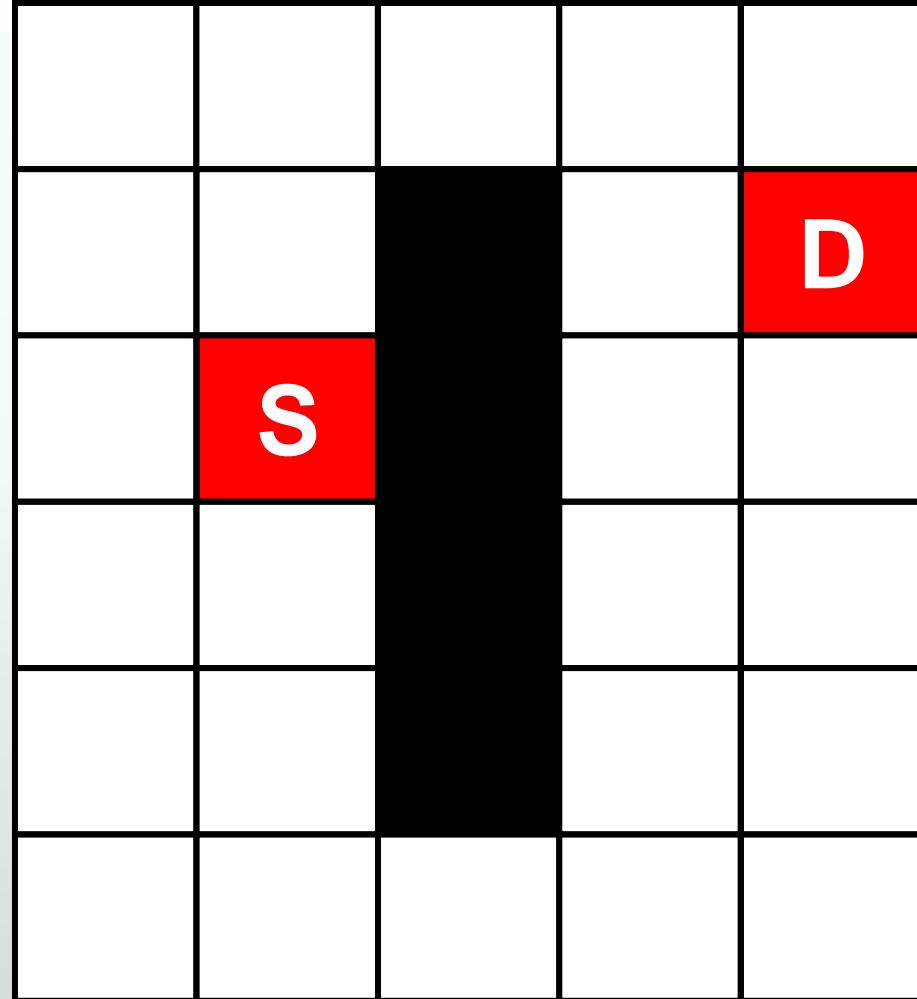
Exemples d'application de A*

Exemple 1

 Nœud source

 Nœud destination

 Obstacle



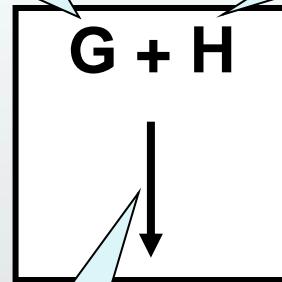
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

		$10 + 30$ ↓		D
$10 + 50$ →	S			
	↑	$10 + 50$		

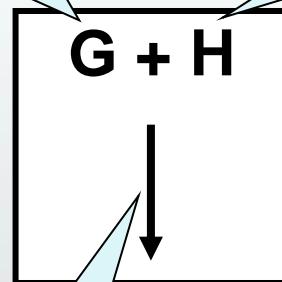
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

	$20 + 40$ ↓			
$20 + 40$ →	$10 + 30$ ↓	S		D
$10 + 50$ →			↑	$10 + 50$

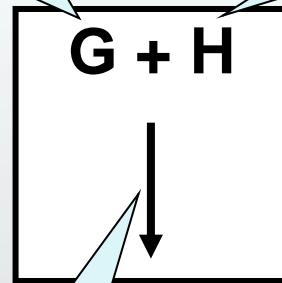
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

	$20 + 40$ ↓			
$20 + 40$ →	$10 + 30$ ↓			D
$10 + 50$ →	S			
↑	↑			
$20 + 60$	$10 + 50$			

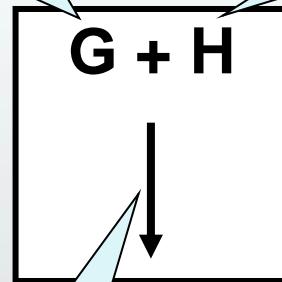
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

	$20 + 40$ ↓			
$20 + 40$ →	$10 + 30$ ↓			D
$10 + 50$ →	S			
↑	↑			
$20 + 60$	$10 + 50$			
	$20 + 60$			

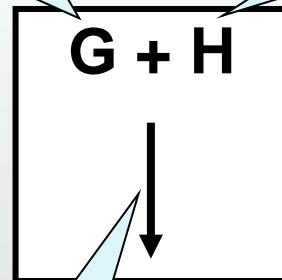
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

$30 + 50$ 	$20 + 40$ 			
$20 + 40$ 	$10 + 30$ 			D
$10 + 50$ 	S			
				
$20 + 60$	$10 + 50$			
		$20 + 60$		

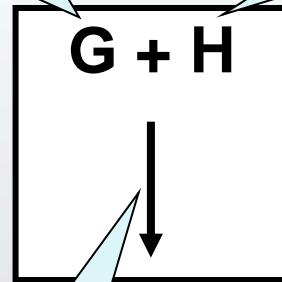
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

30 + 50	20 + 40	30 + 30		
20 + 40	10 + 30			D
10 + 50	S			
20 + 60	10 + 50			
20 + 60				

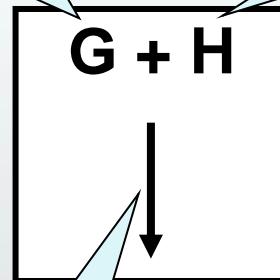
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

$30 + 50$ ↓	$20 + 40$ ↓	$30 + 30$ ←	$40 + 20$ ←	
$20 + 40$ →	$10 + 30$ ↓			D
$10 + 50$ →	S			
↑	↑			
$20 + 60$	$10 + 50$			
	↑			
	$20 + 60$			

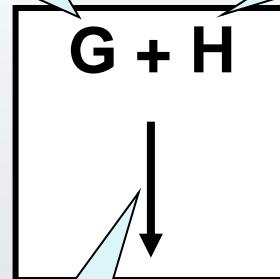
Exemple 1

 Nœud développé

 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent

$30 + 50$	$20 + 40$	$30 + 30$	$40 + 20$	$50 + 10$
$20 + 40$	$10 + 30$			D
$10 + 50$	S			$50 + 10$
$20 + 60$	$10 + 50$			
	$20 + 60$			

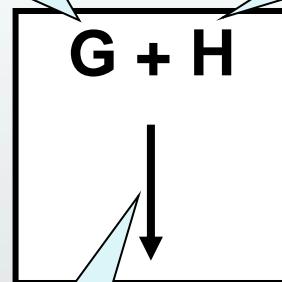
Exemple 1

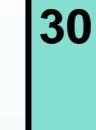
 Nœud développé

 Nœud généré

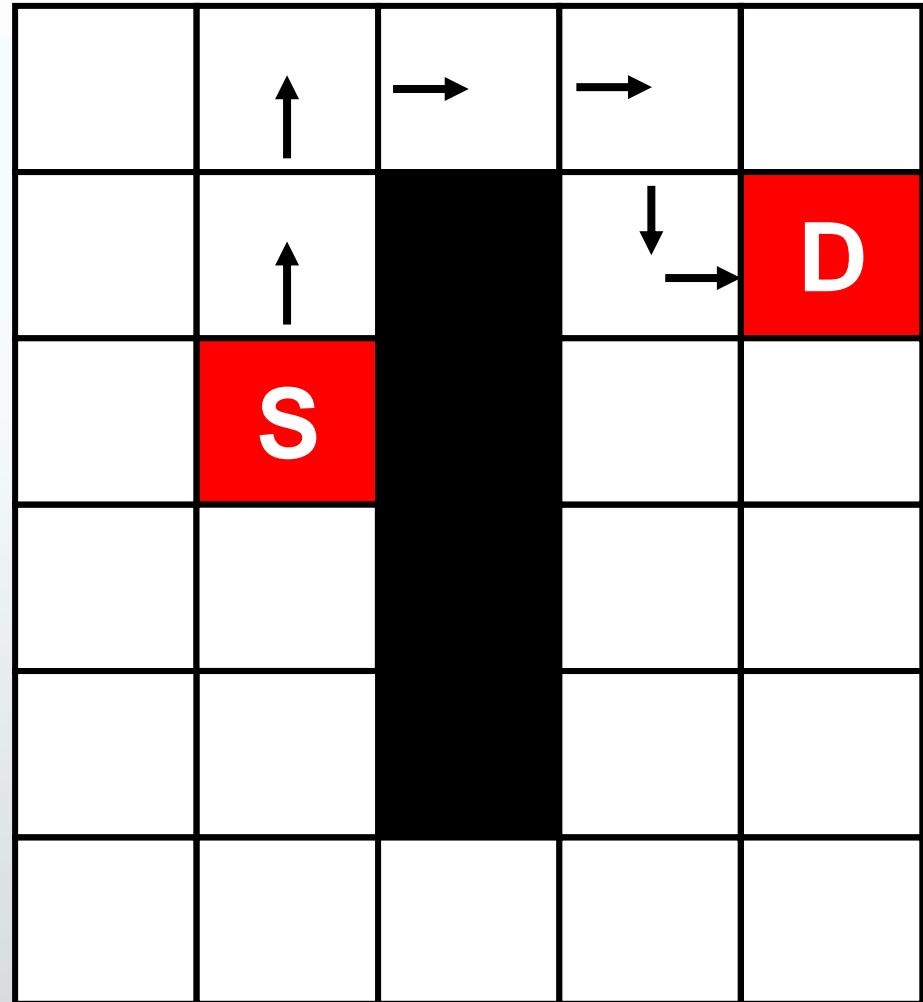
Coût depuis
la source

Coût vers
la destination



$30 + 50$	$20 + 40$	$30 + 30$	$40 + 20$	$50 + 10$
				
$20 + 40$	$10 + 30$			$60 + 0$
				
$10 + 50$		S	$50 + 10$	$60 + 20$
				
$20 + 60$	$10 + 50$			
			$20 + 60$	

Exemple 1 : chemin complet

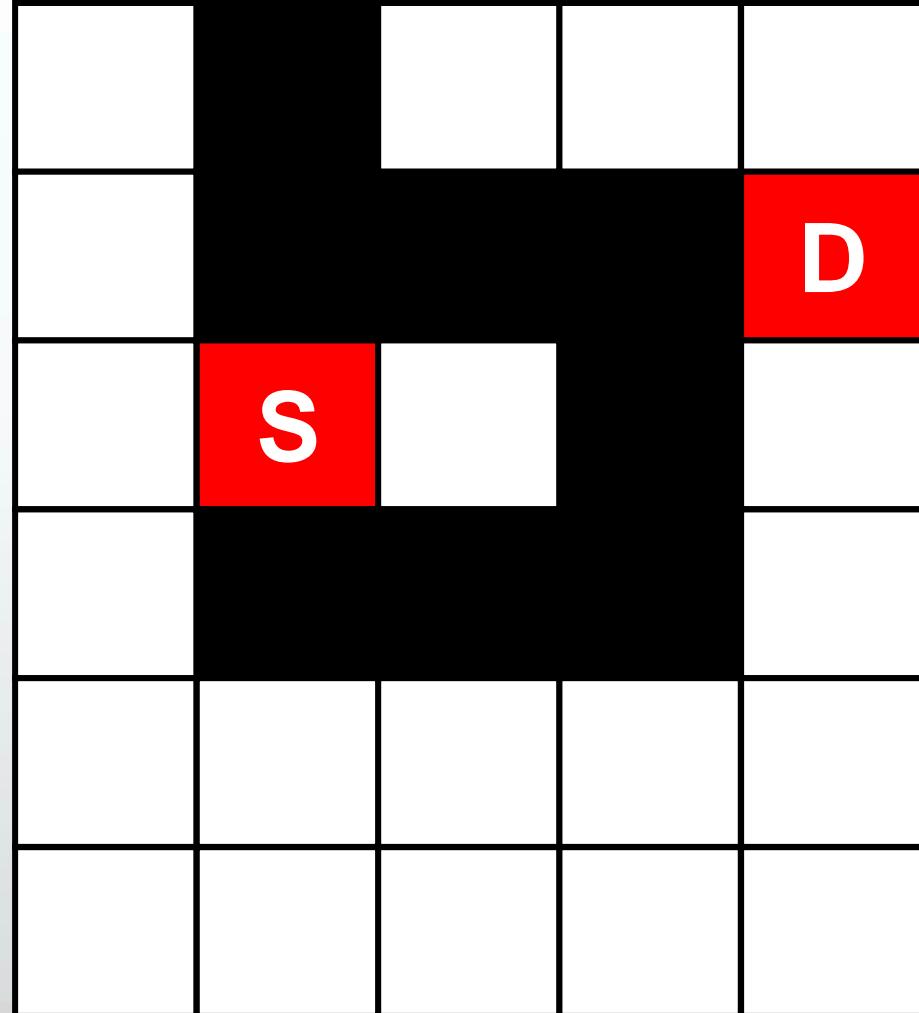


Exemple 2

 Nœud source

 Nœud destination

 Obstacle



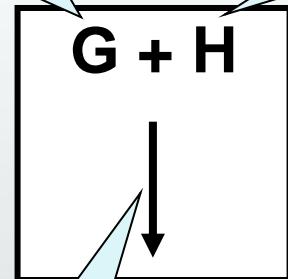
Exemple 2

 Nœud développé

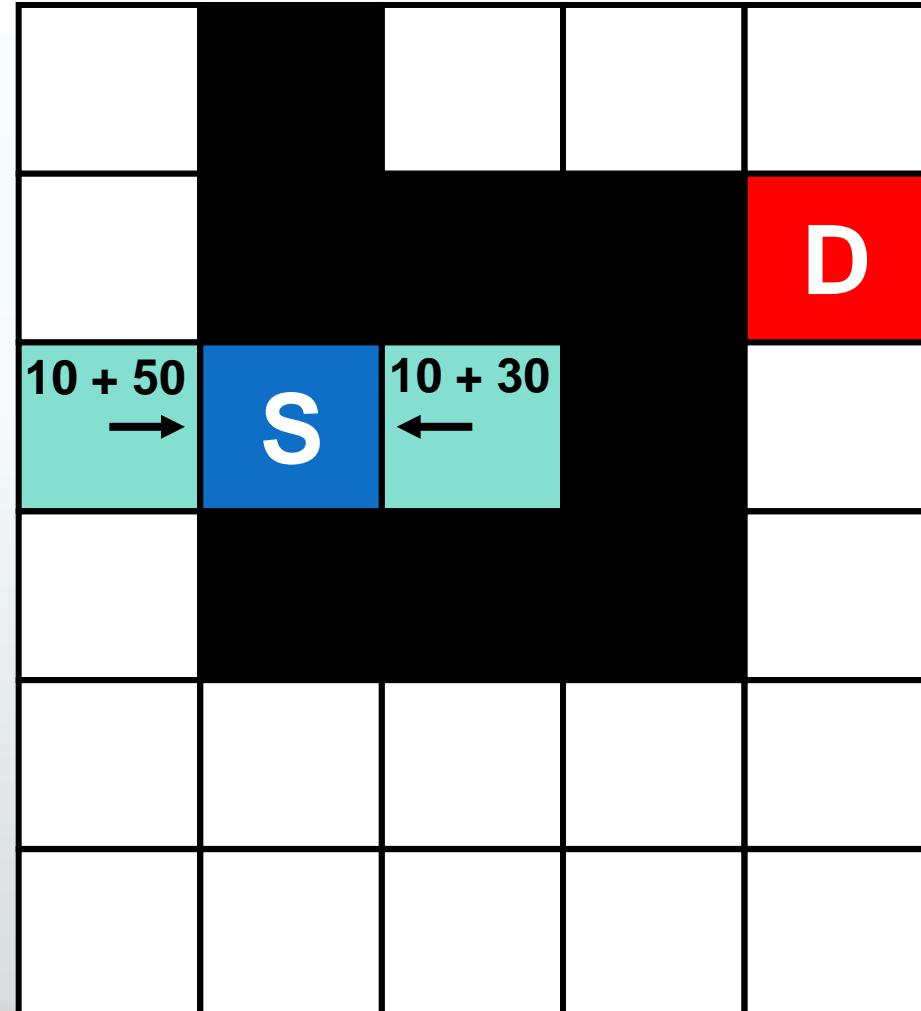
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



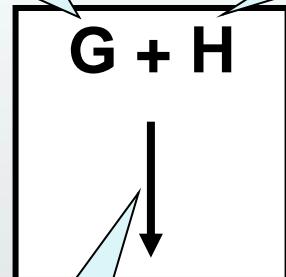
Exemple 2

 Nœud développé

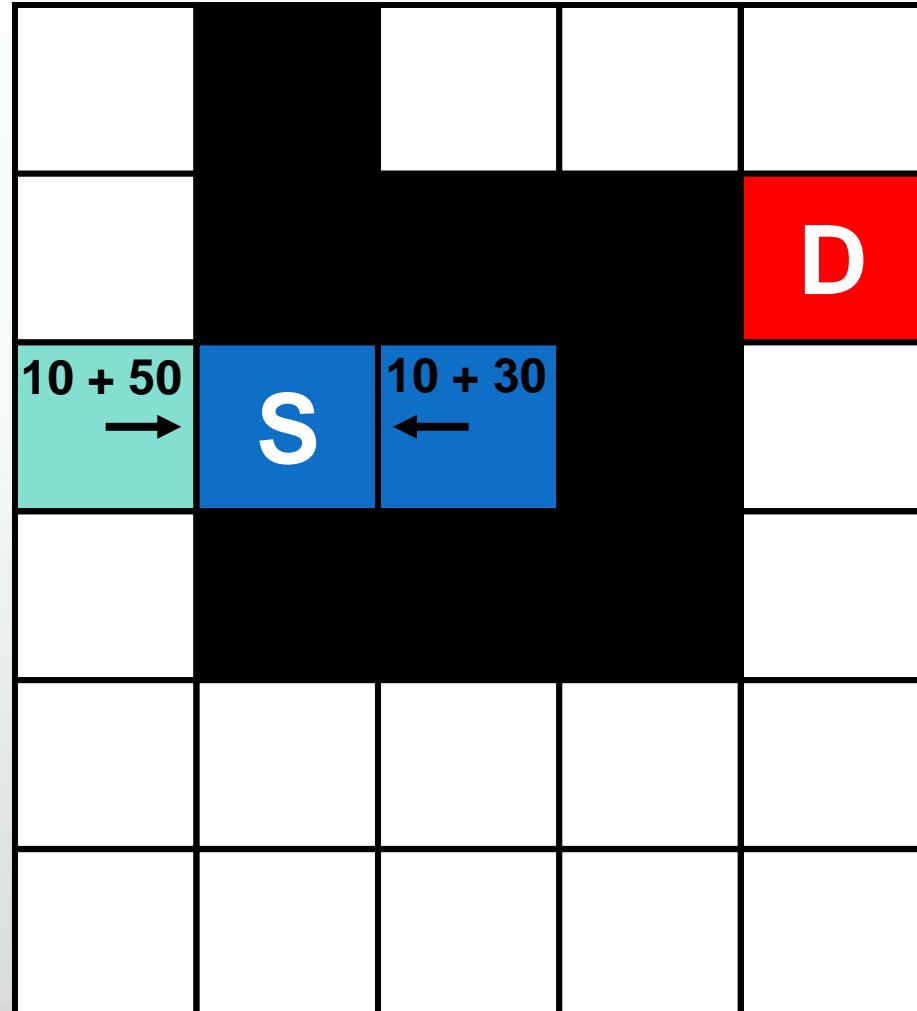
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



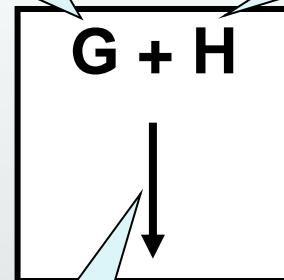
Exemple 2

 Nœud développé

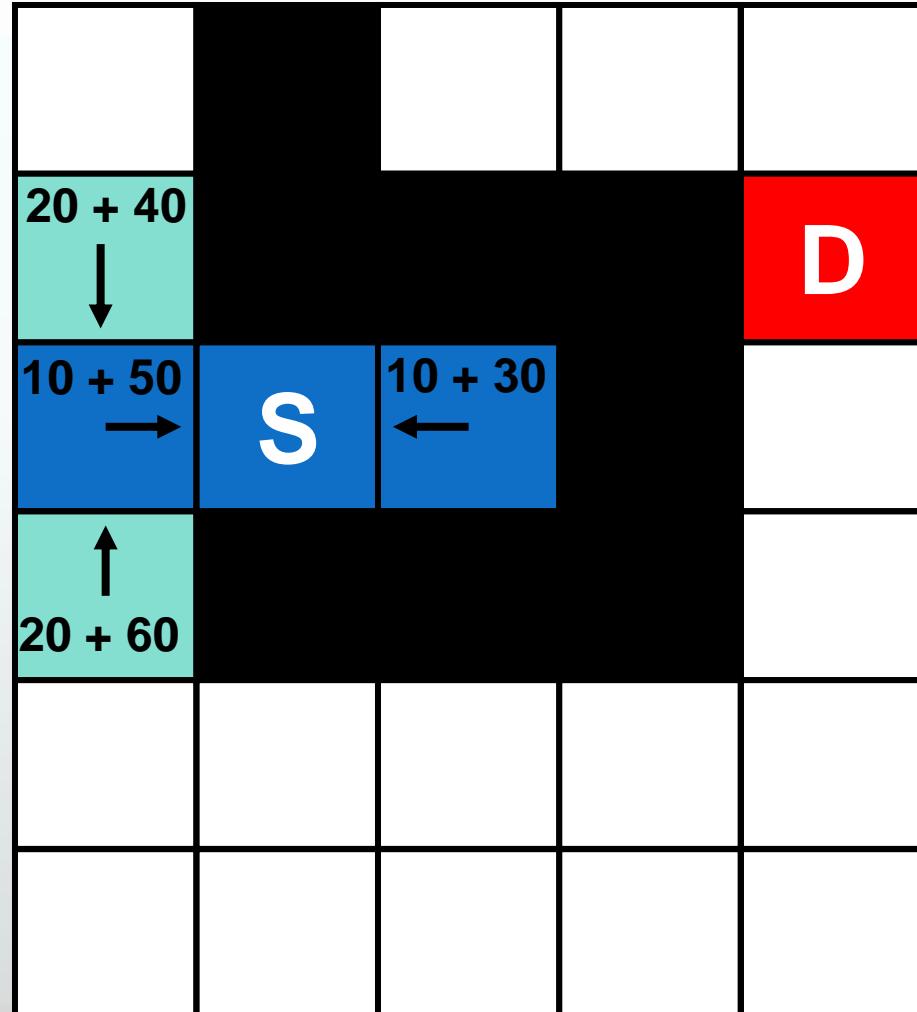
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



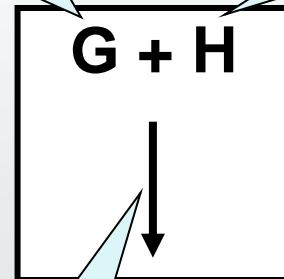
Exemple 2

 Nœud développé

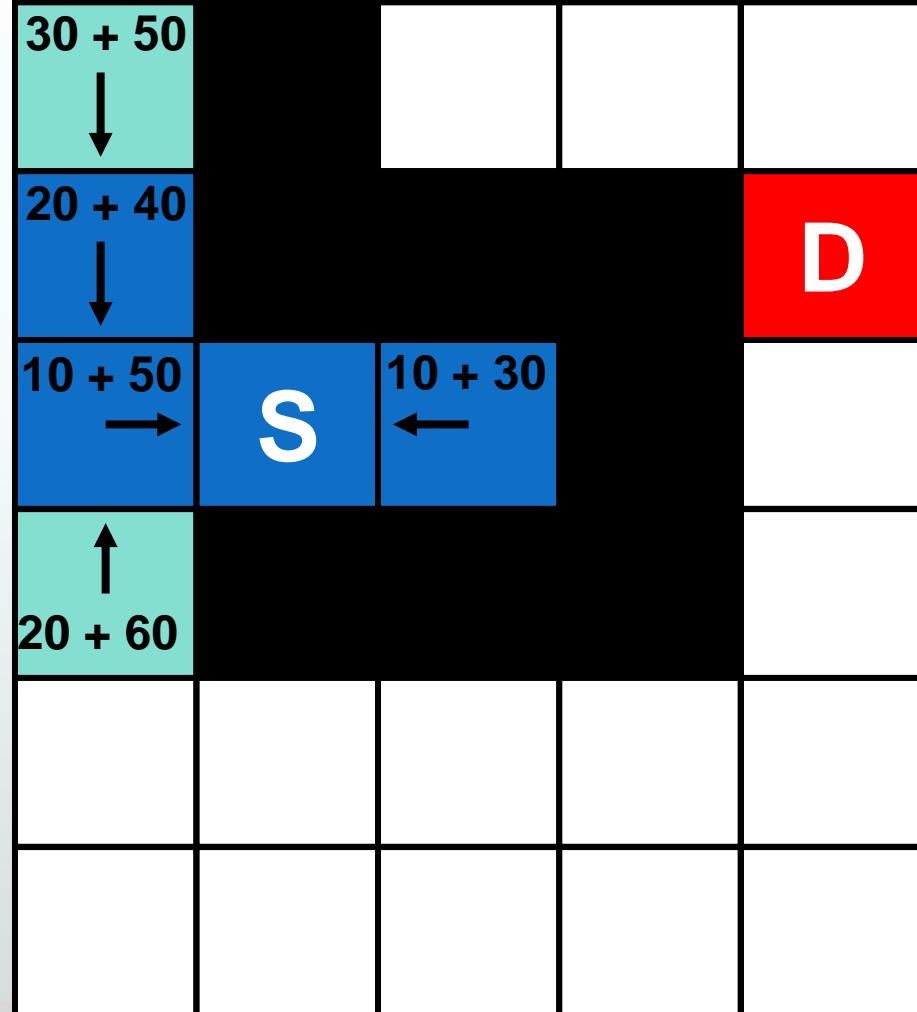
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



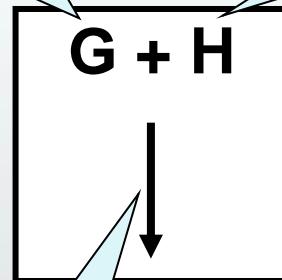
Exemple 2

 Nœud développé

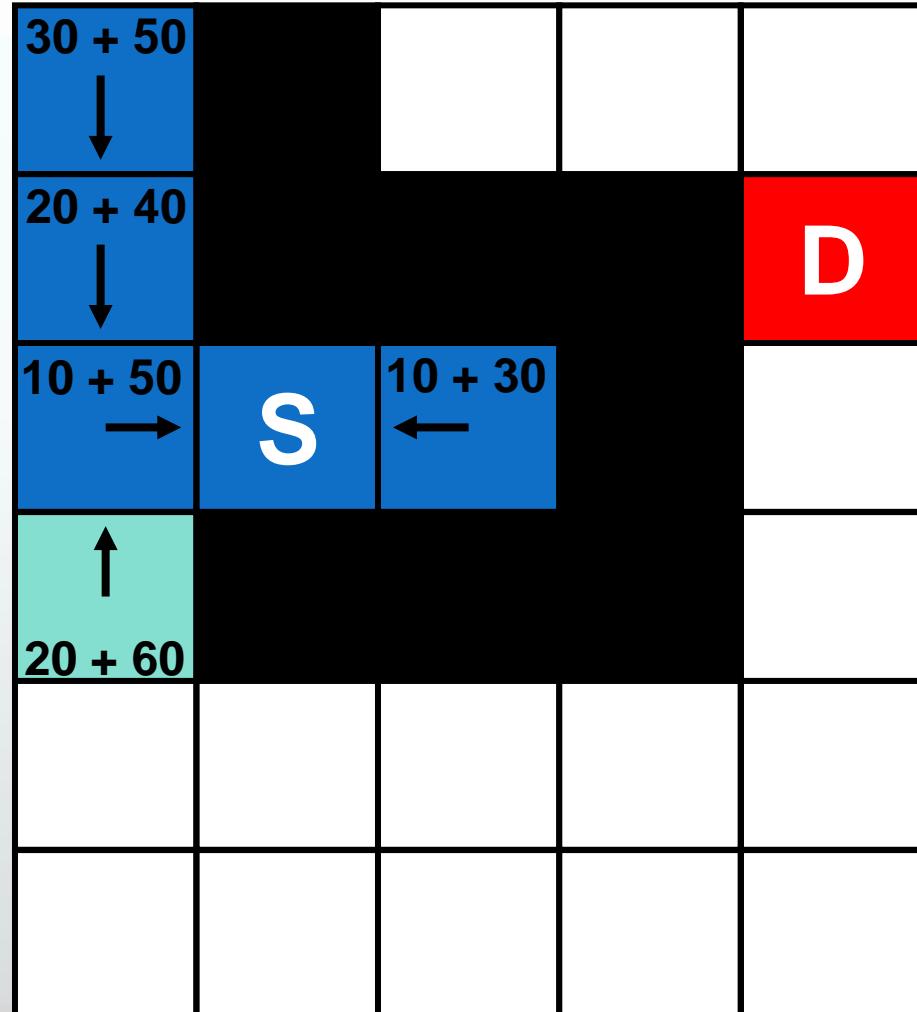
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



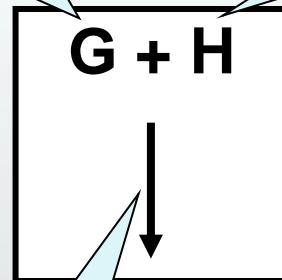
Exemple 2

 Nœud développé

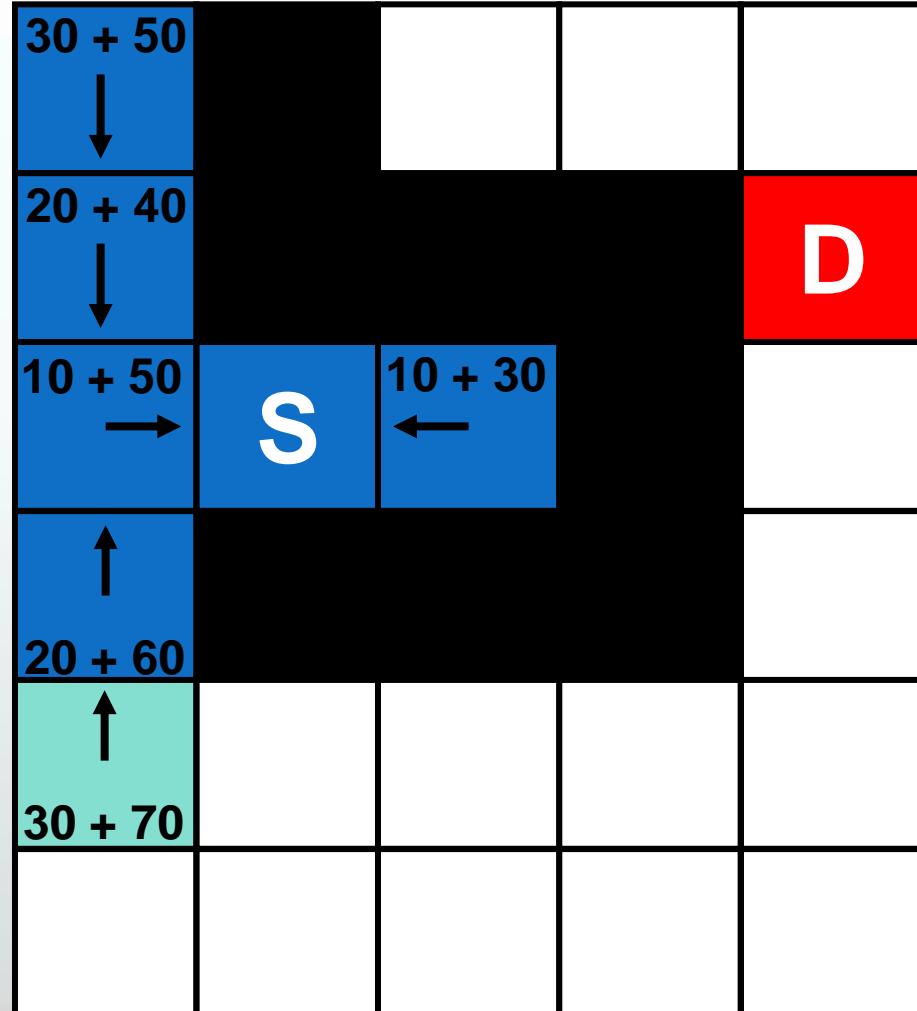
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



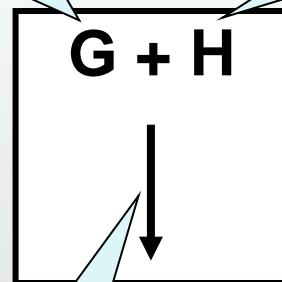
Exemple 2

 Nœud développé

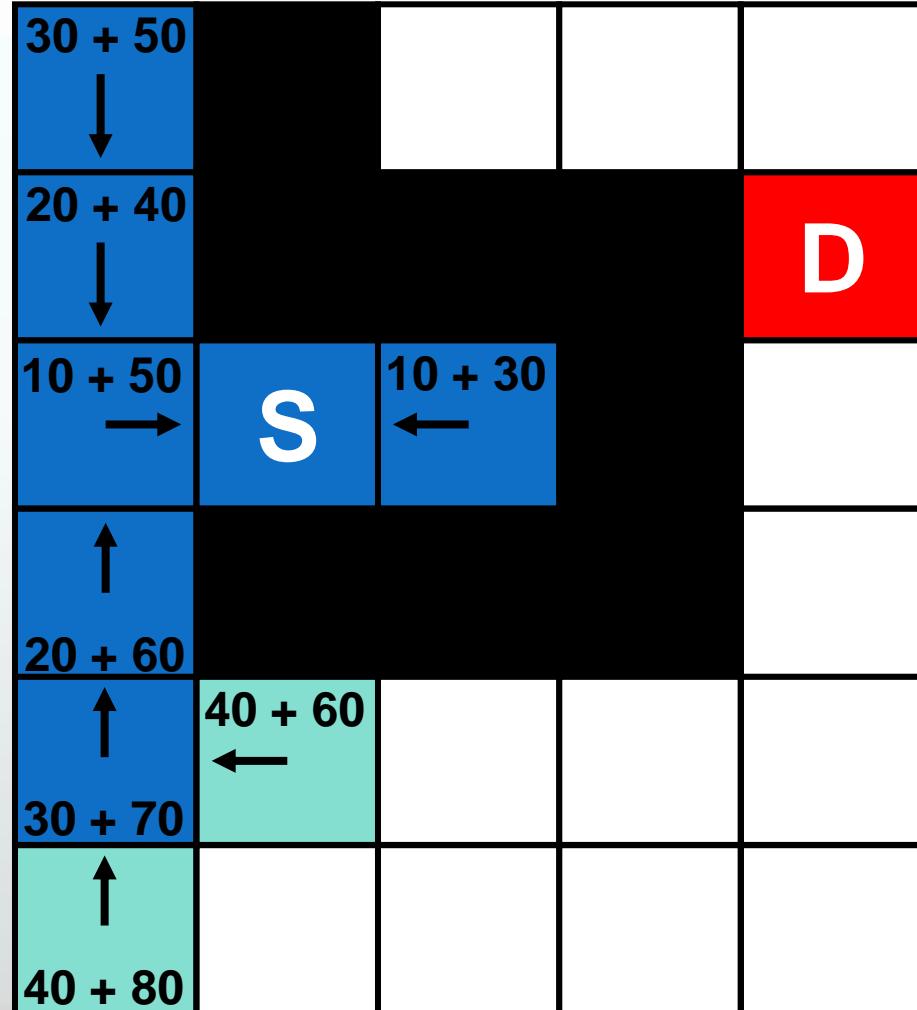
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



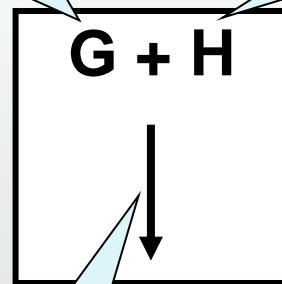
Exemple 2

 Nœud développé

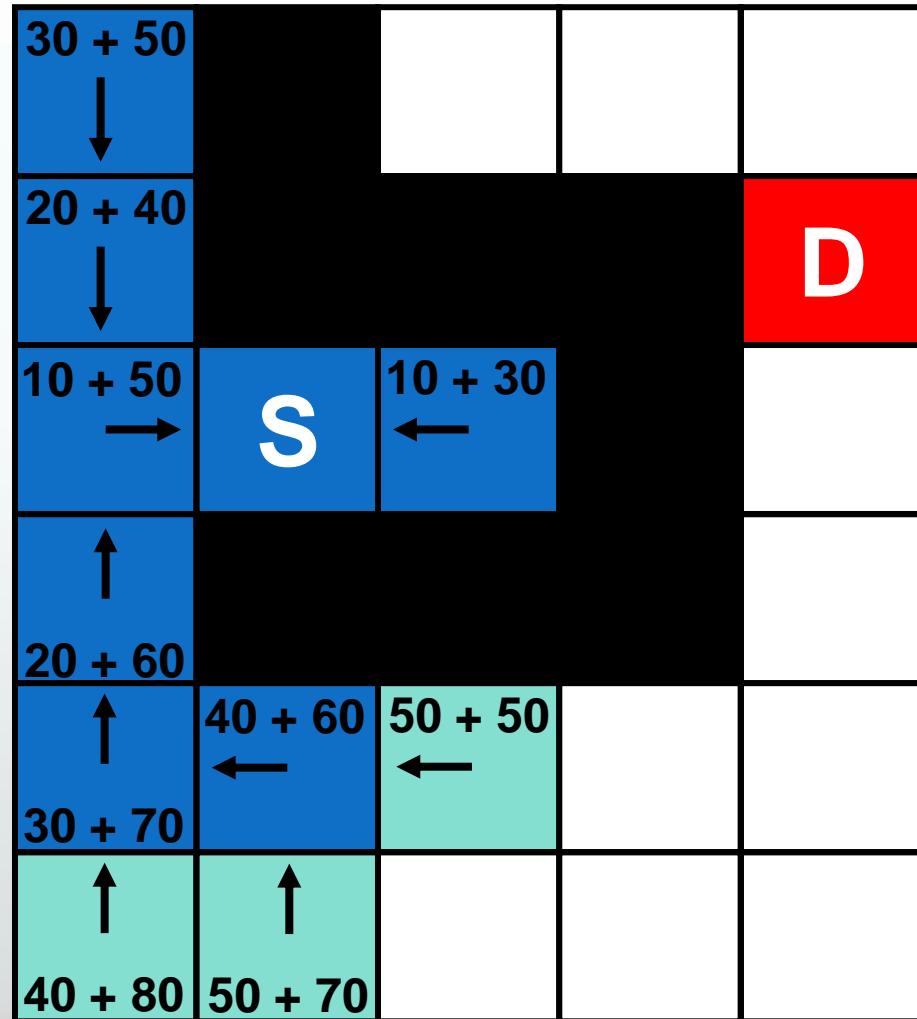
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



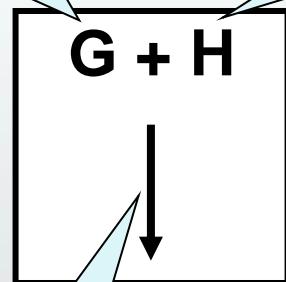
Exemple 2

 Nœud développé

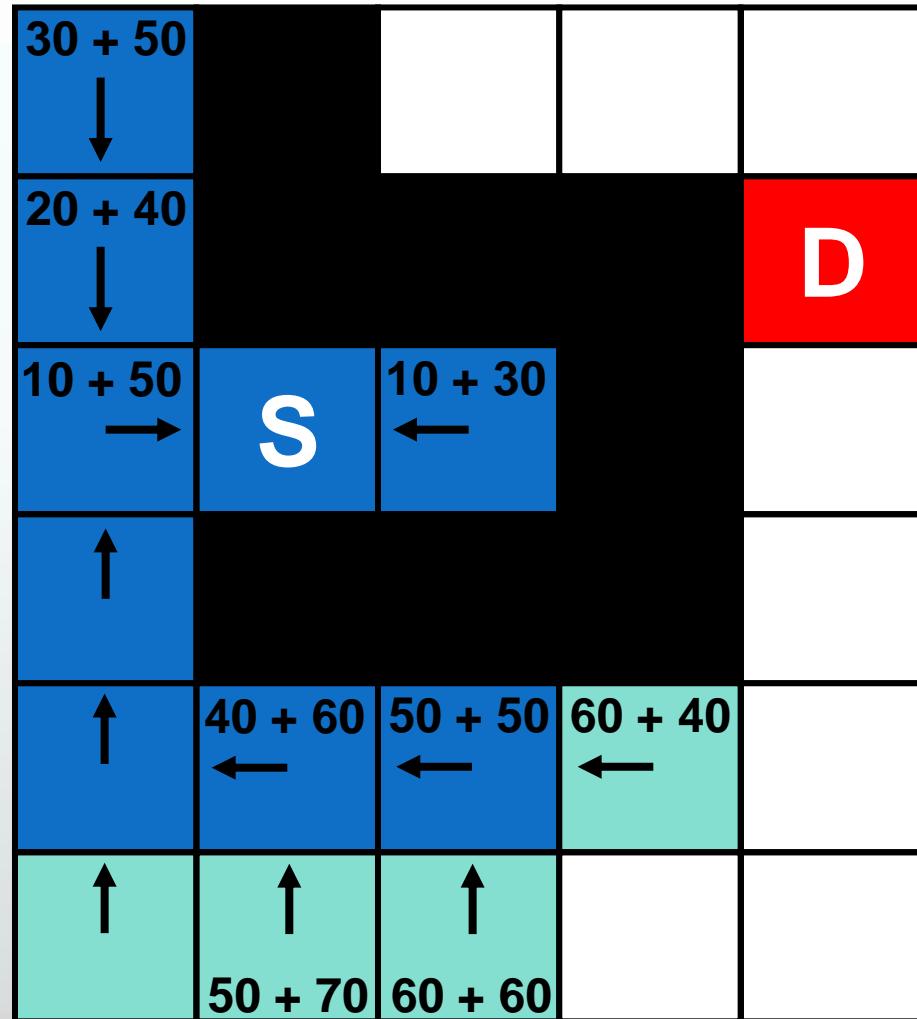
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



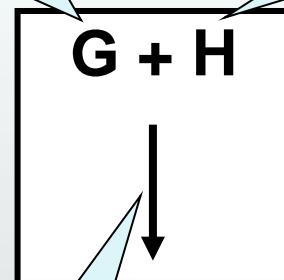
Exemple 2

 Nœud développé

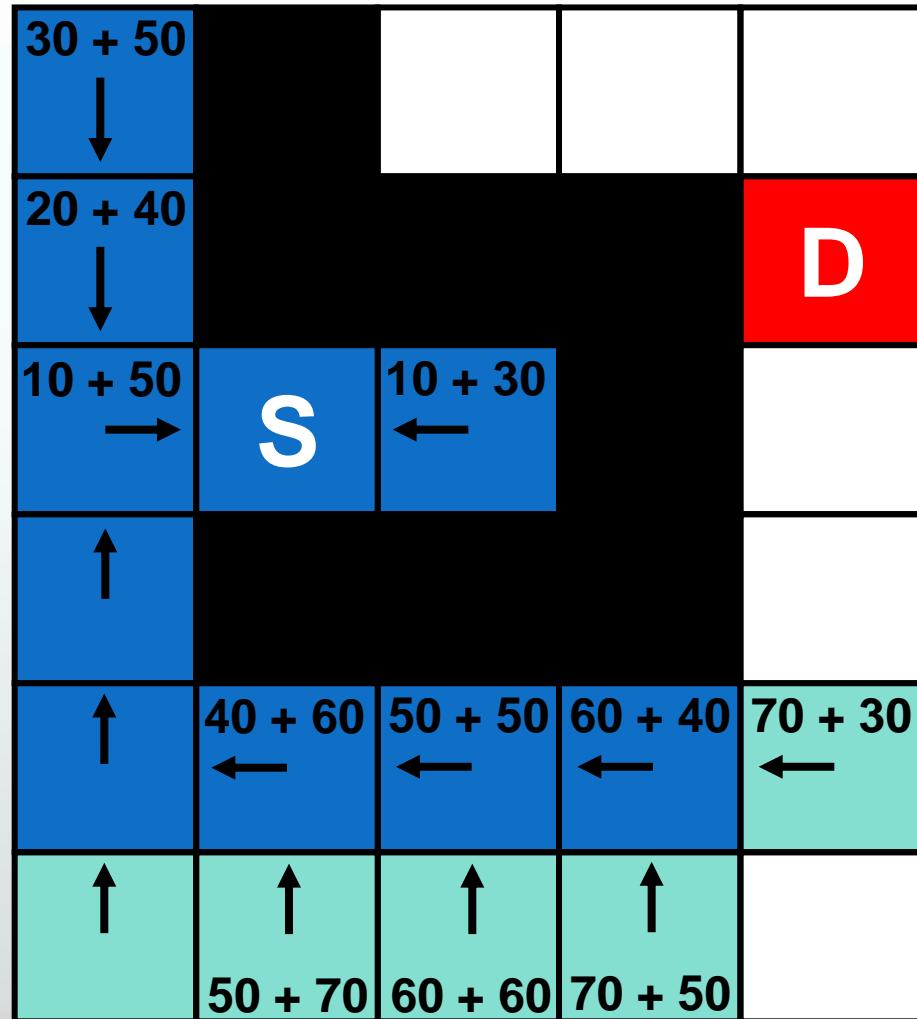
 Nœud généré

Coût depuis
la source

Coût vers
la destination



Référence au
parent



A grid-based search space diagram. The columns are labeled with costs: $30 + 50$, $20 + 40$, $10 + 50$, S , $10 + 30$, $40 + 60$, $50 + 50$, $60 + 40$, and $70 + 30$. The rows are labeled with costs: $50 + 70$, $60 + 60$, $70 + 50$, and D . Blue squares represent developed nodes, teal squares represent generated nodes, and black squares represent empty slots. Arrows indicate the direction of cost calculations.

$30 + 50$	$20 + 40$	$10 + 50$	S	$10 + 30$	$40 + 60$	$50 + 50$	$60 + 40$	$70 + 30$	D
$50 + 70$	$60 + 60$	$70 + 50$							

Exemple 2

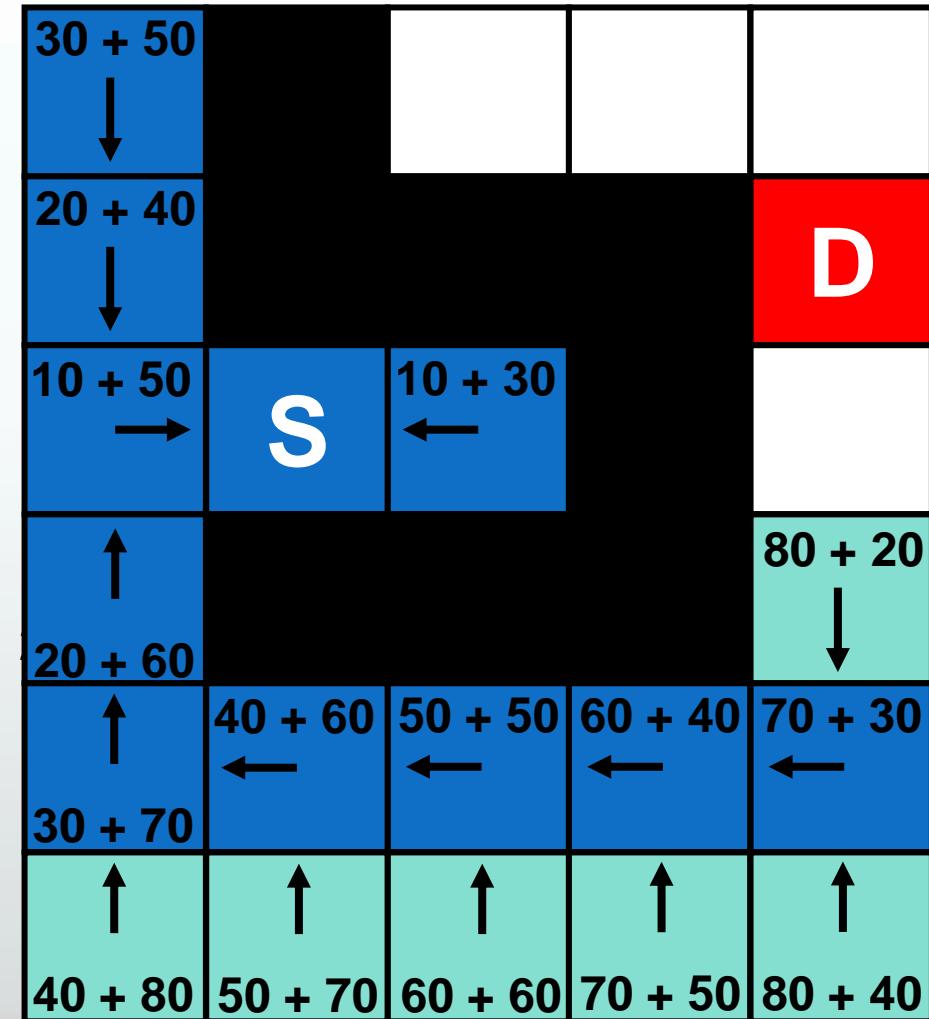
- Nœud développé
- Nœud généré

Coût depuis
la source

Coût vers
la destination

G + H

Référence au
parent



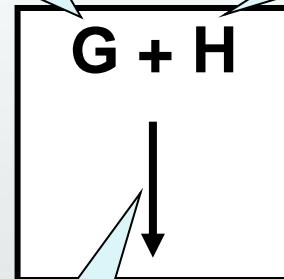
Exemple 2

 Nœud développé

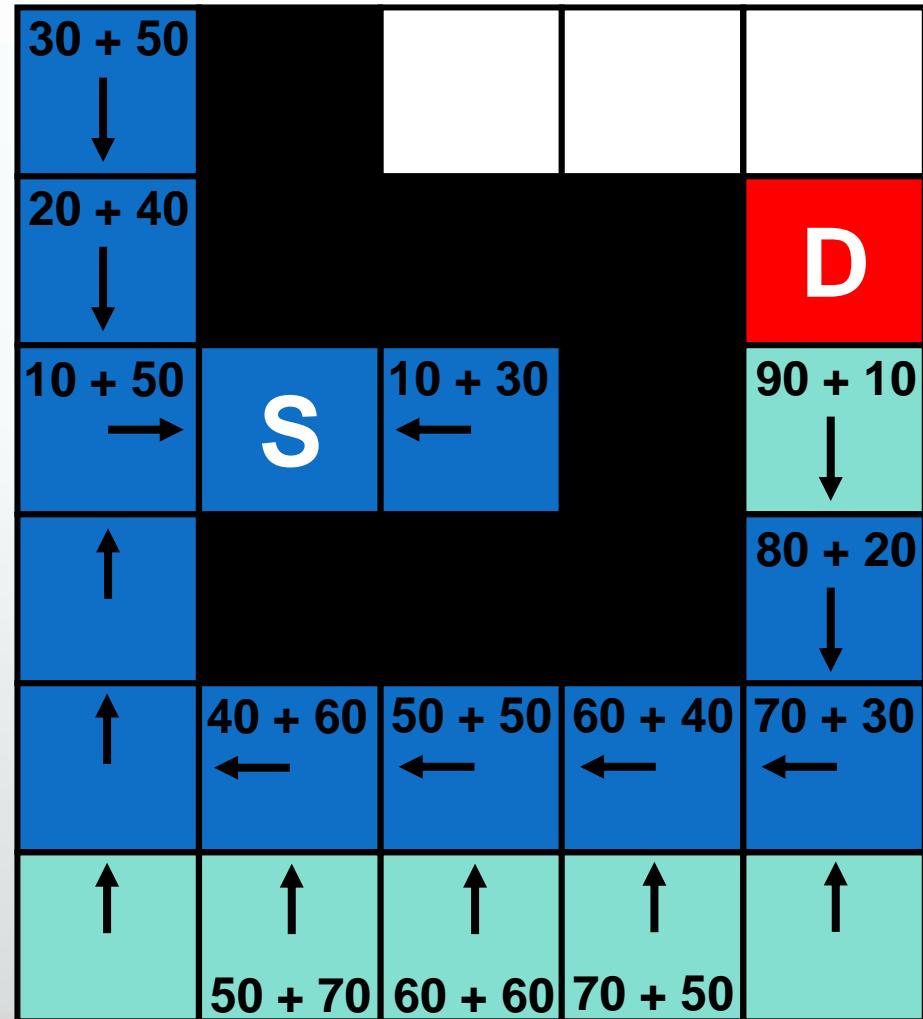
 Nœud généré

Coût depuis
la source

Coût vers
la destination



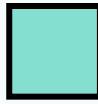
Référence au
parent



Exemple 2



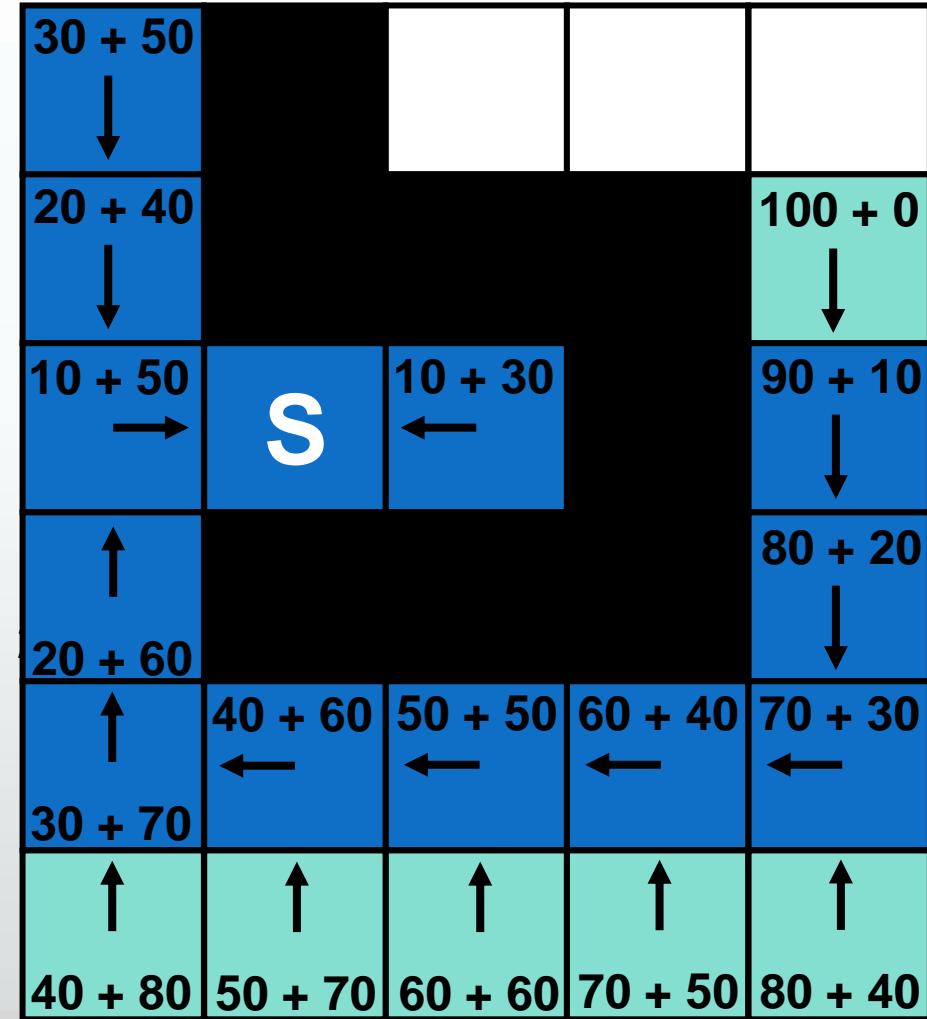
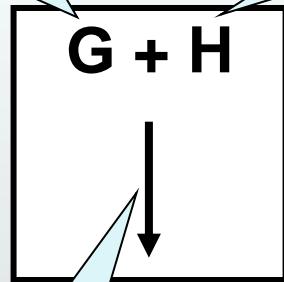
Nœud développé



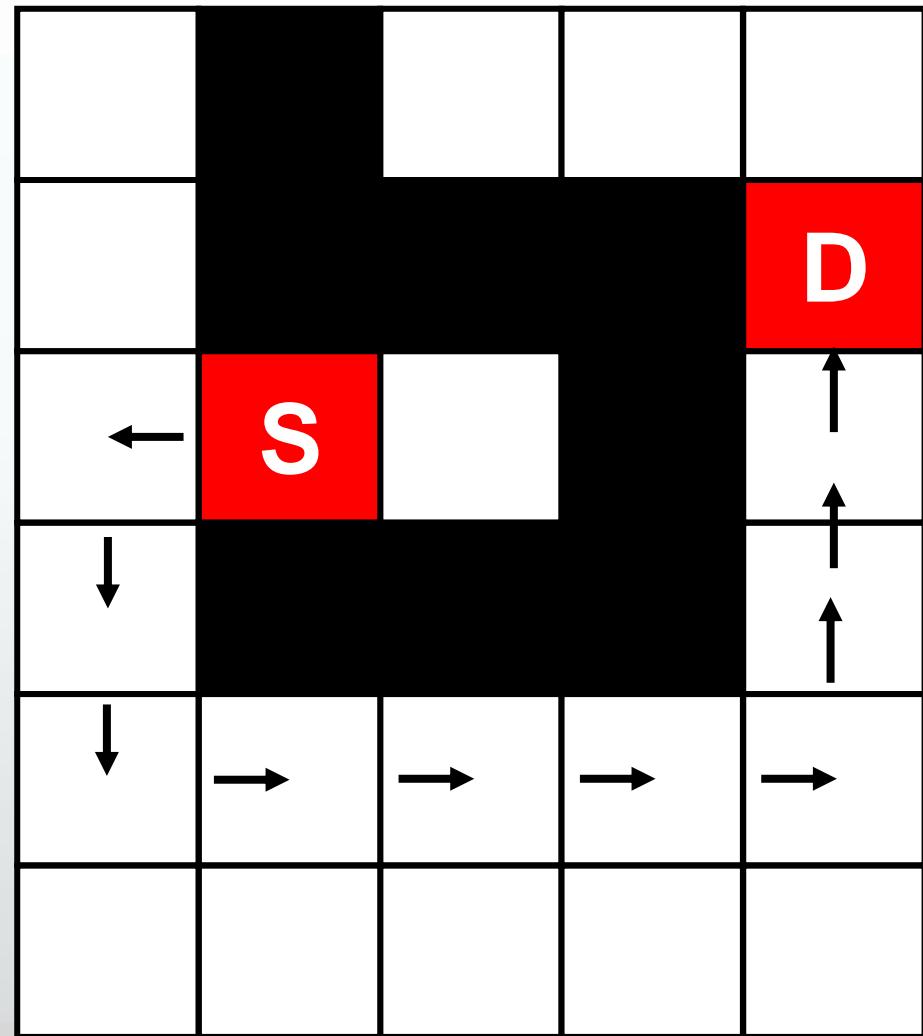
Nœud généré

Coût depuis
la source

Coût vers
la destination



Exemple 2 : chemin complet

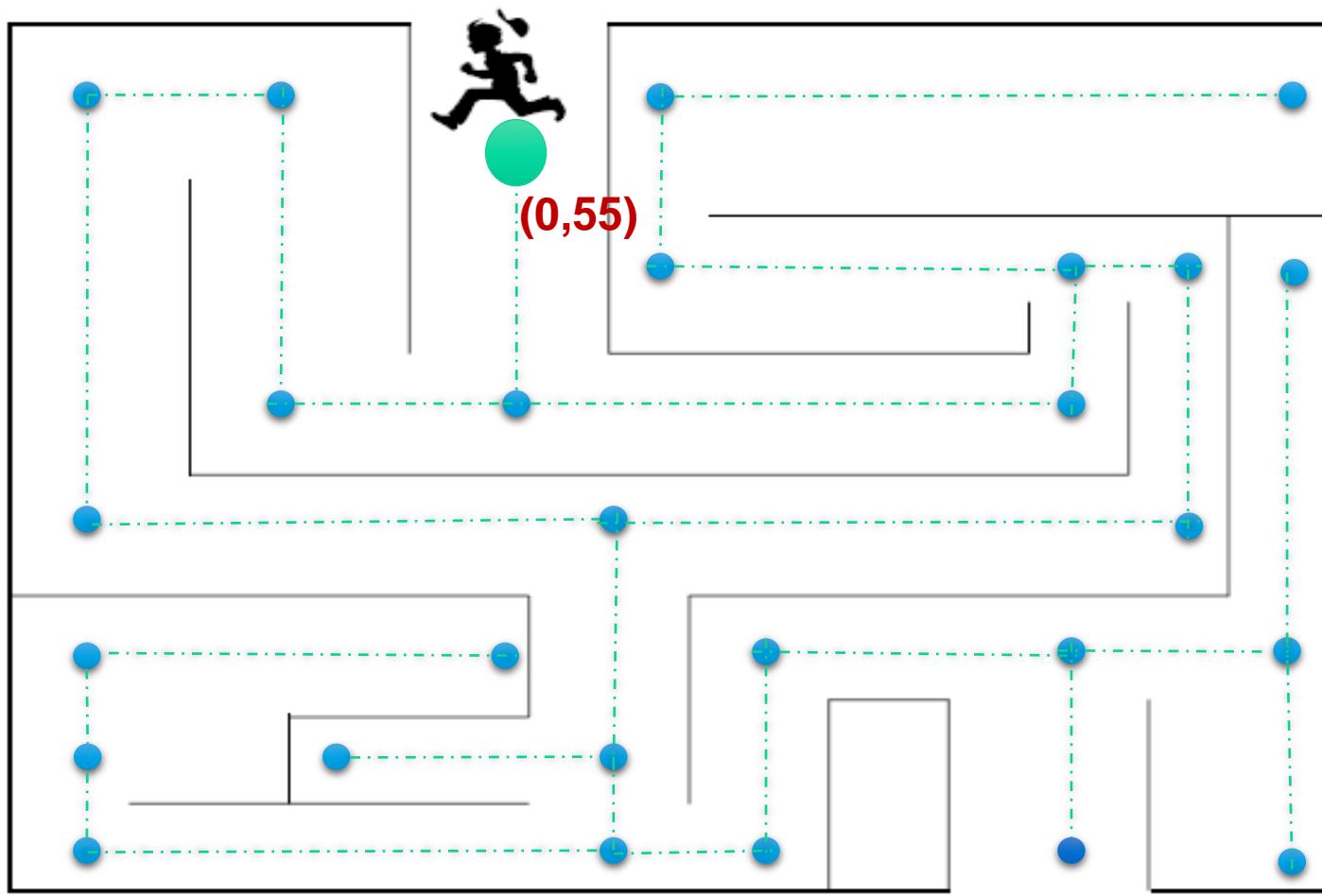


Application (labyrinthe)

Application (labyrinthe)

Au début la distance parcourue est 0

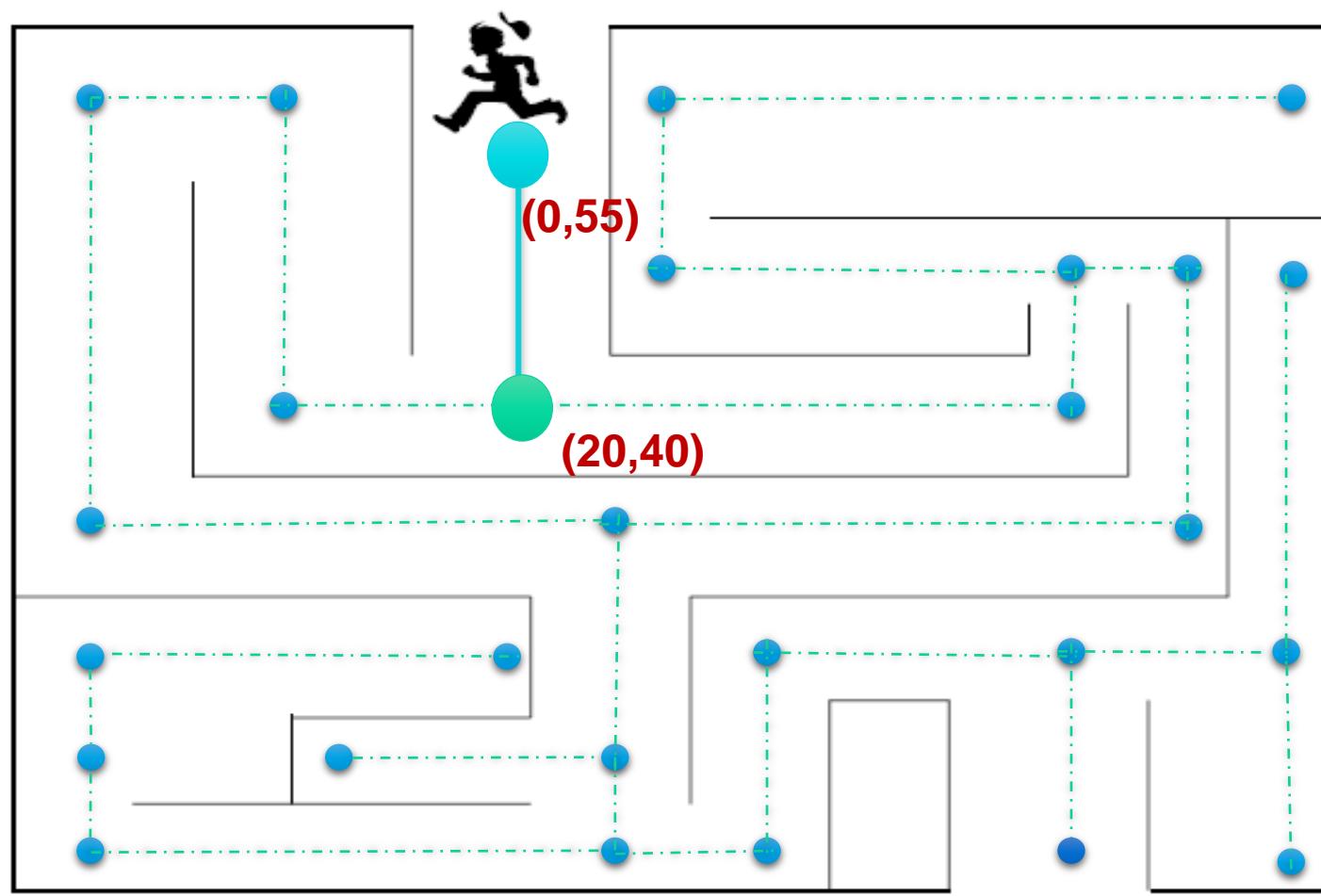
(Distance parcourue, Distance estimée)



- Nœud non encore généré
- Nœud généré
- Nœud développé

Application (labyrinthe)

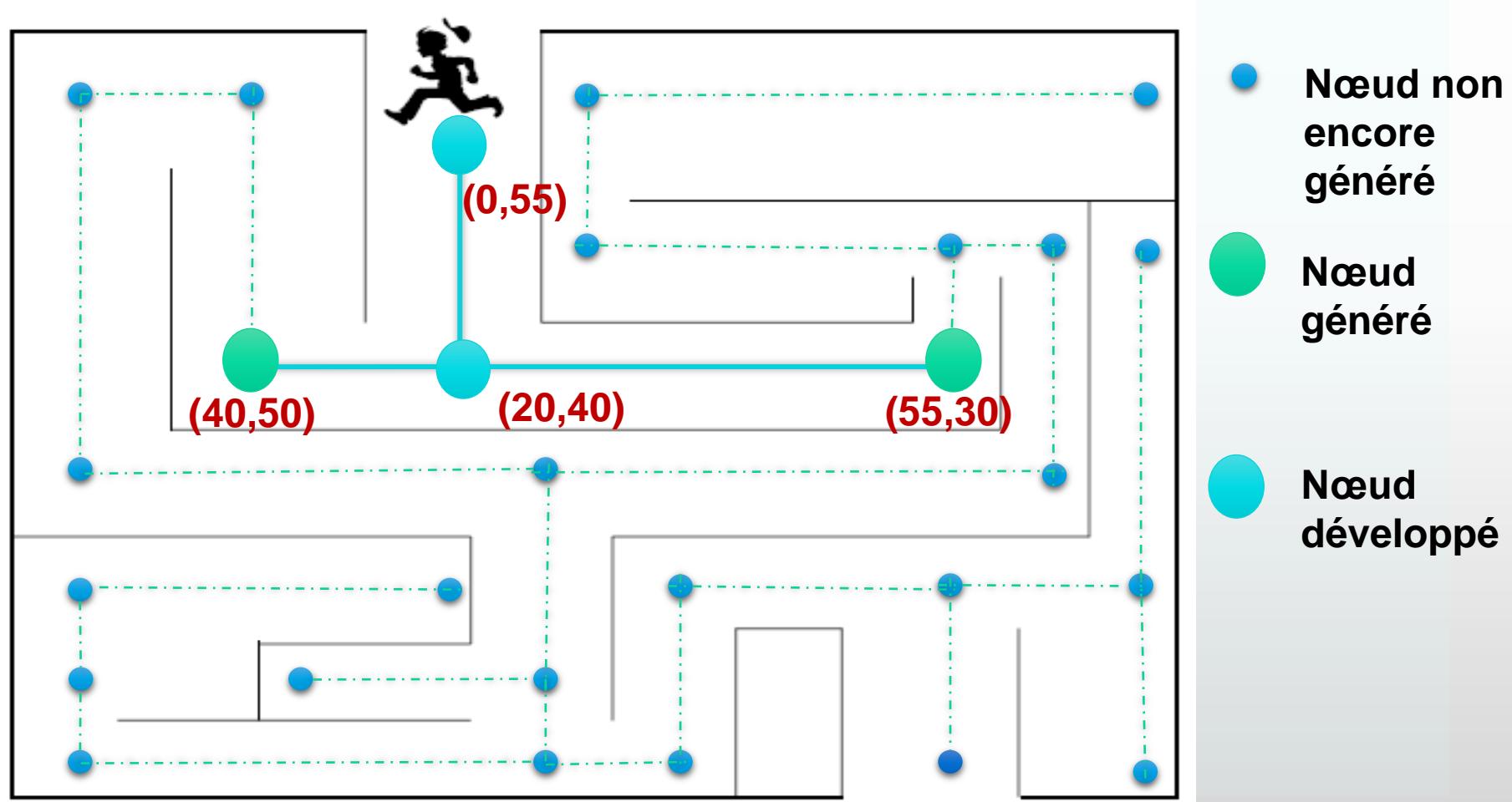
(Distance parcourue, Distance estimée)



- Nœud non encore généré
- Nœud généré
- Nœud développé

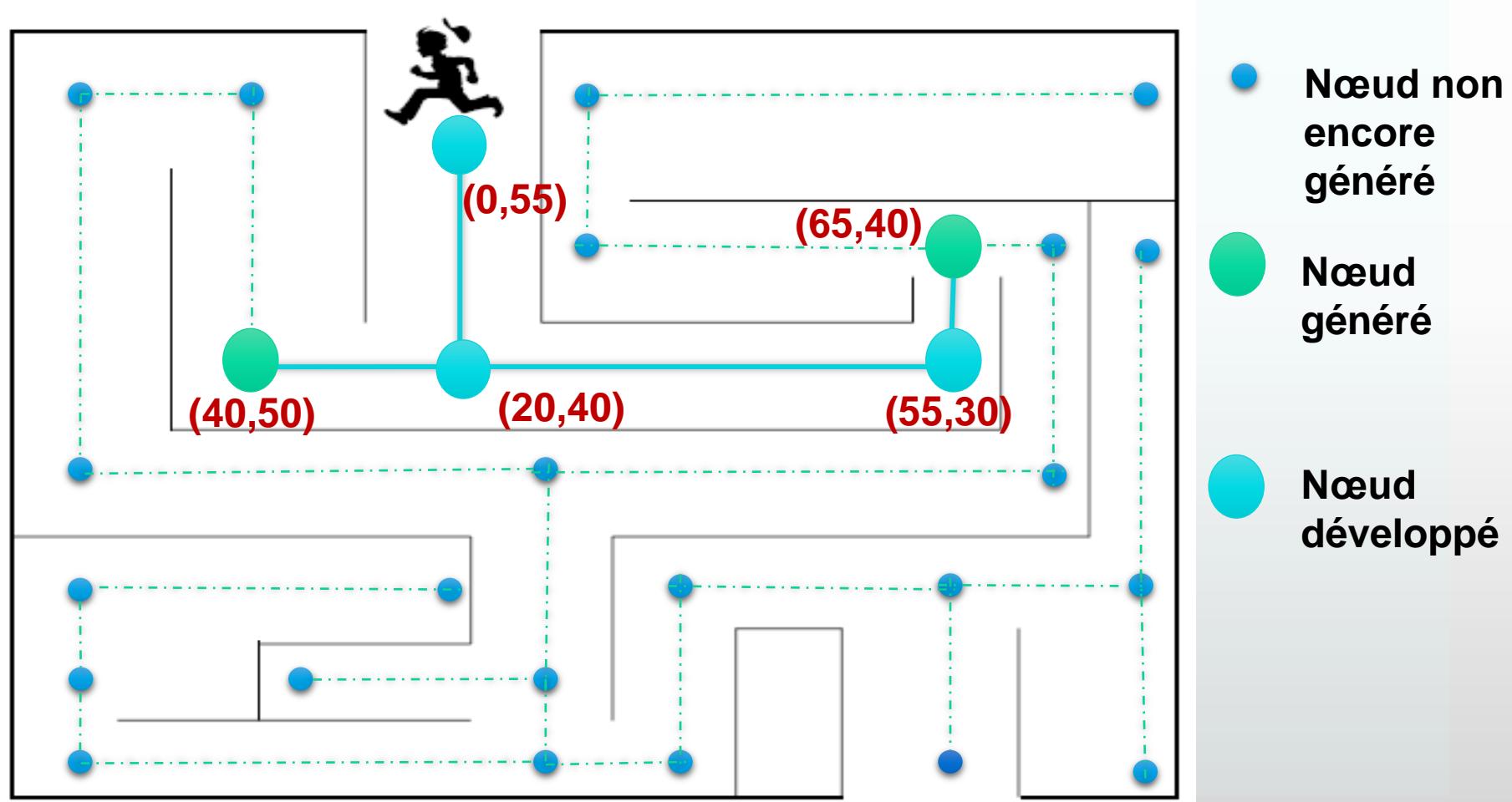
Application (labyrinthe)

(Distance parcourue, Distance estimée)



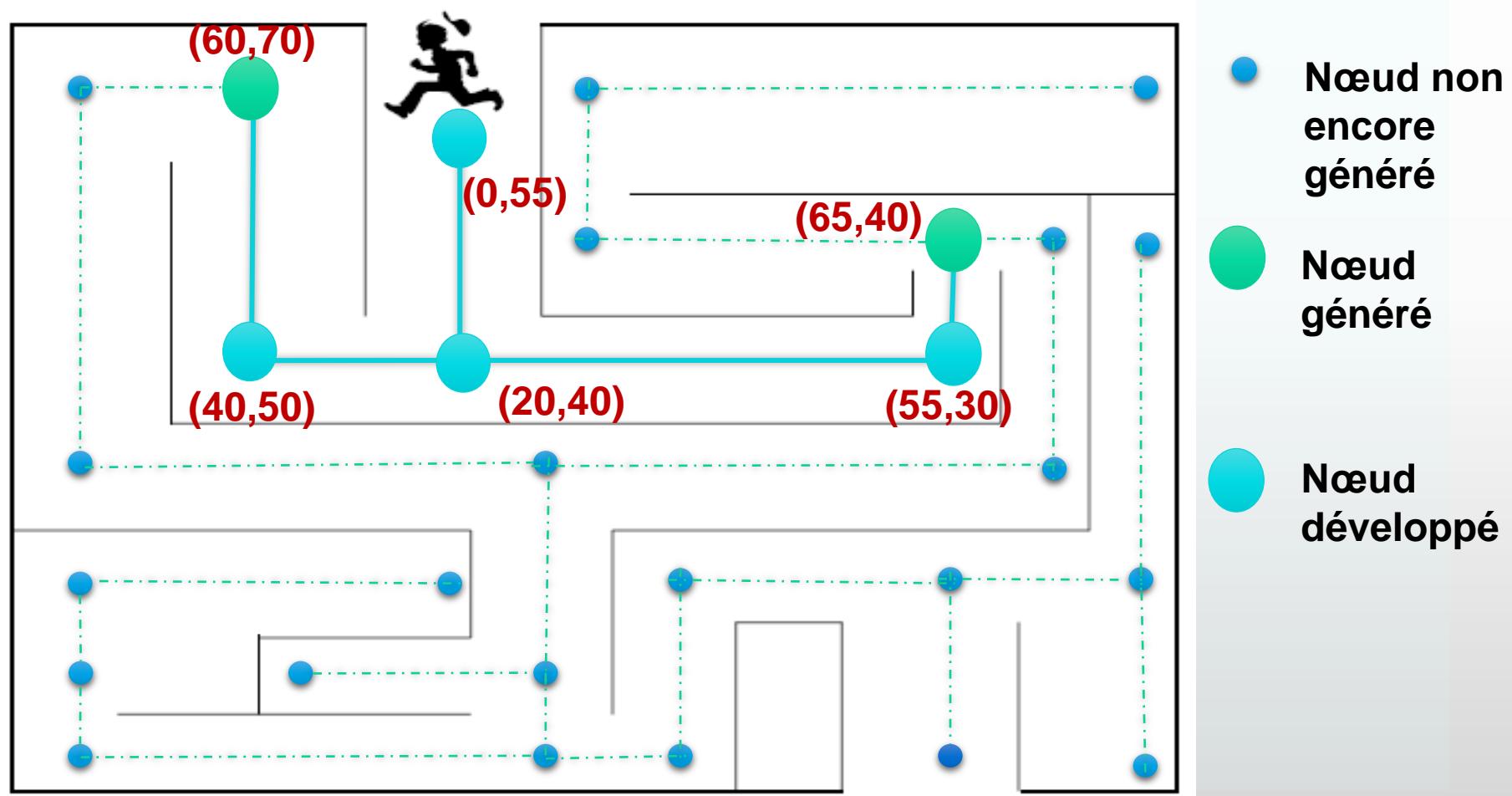
Application (labyrinthe)

(Distance parcourue, Distance estimée)



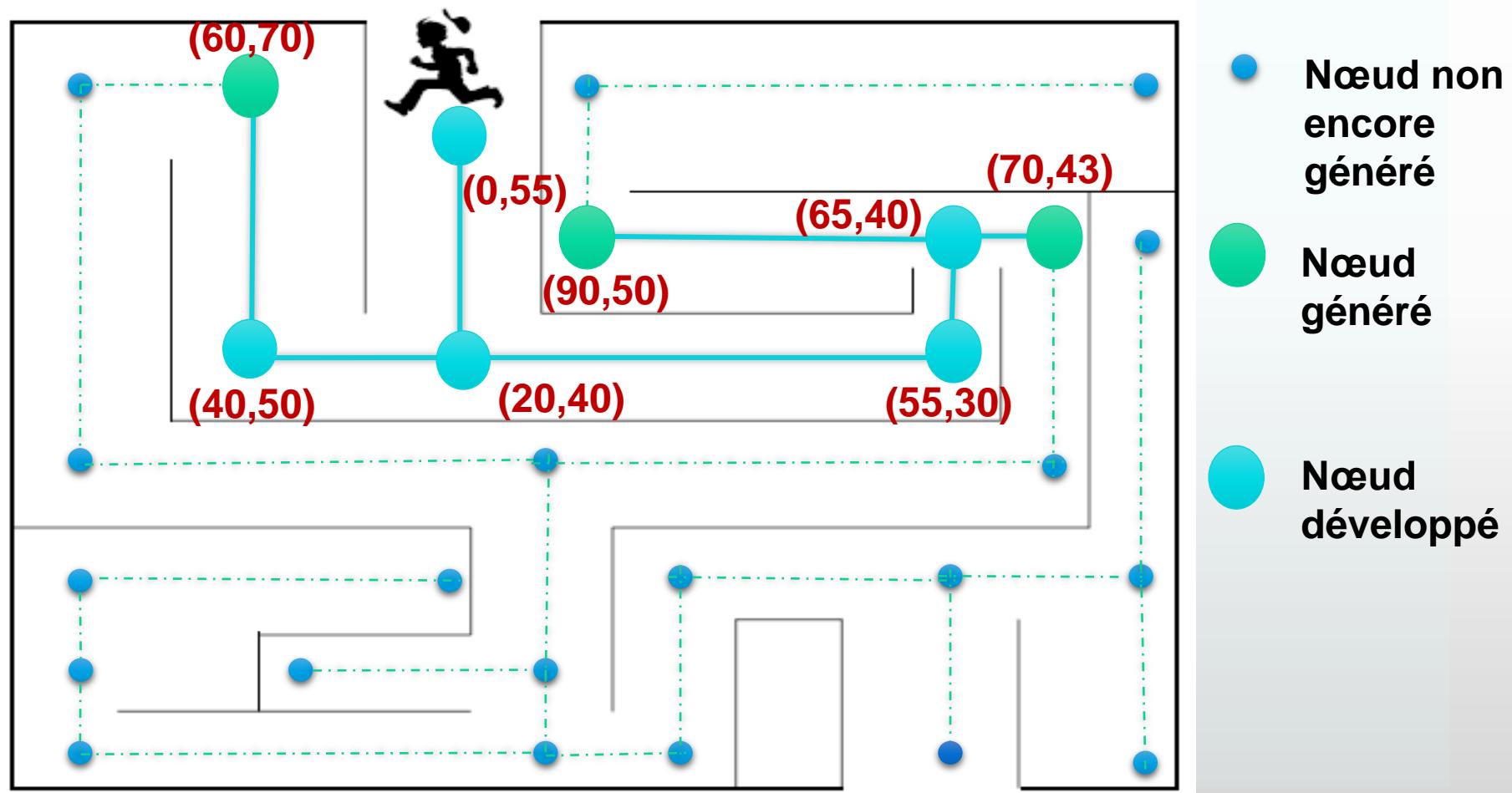
Application (labyrinthe)

(Distance parcourue, Distance estimée)



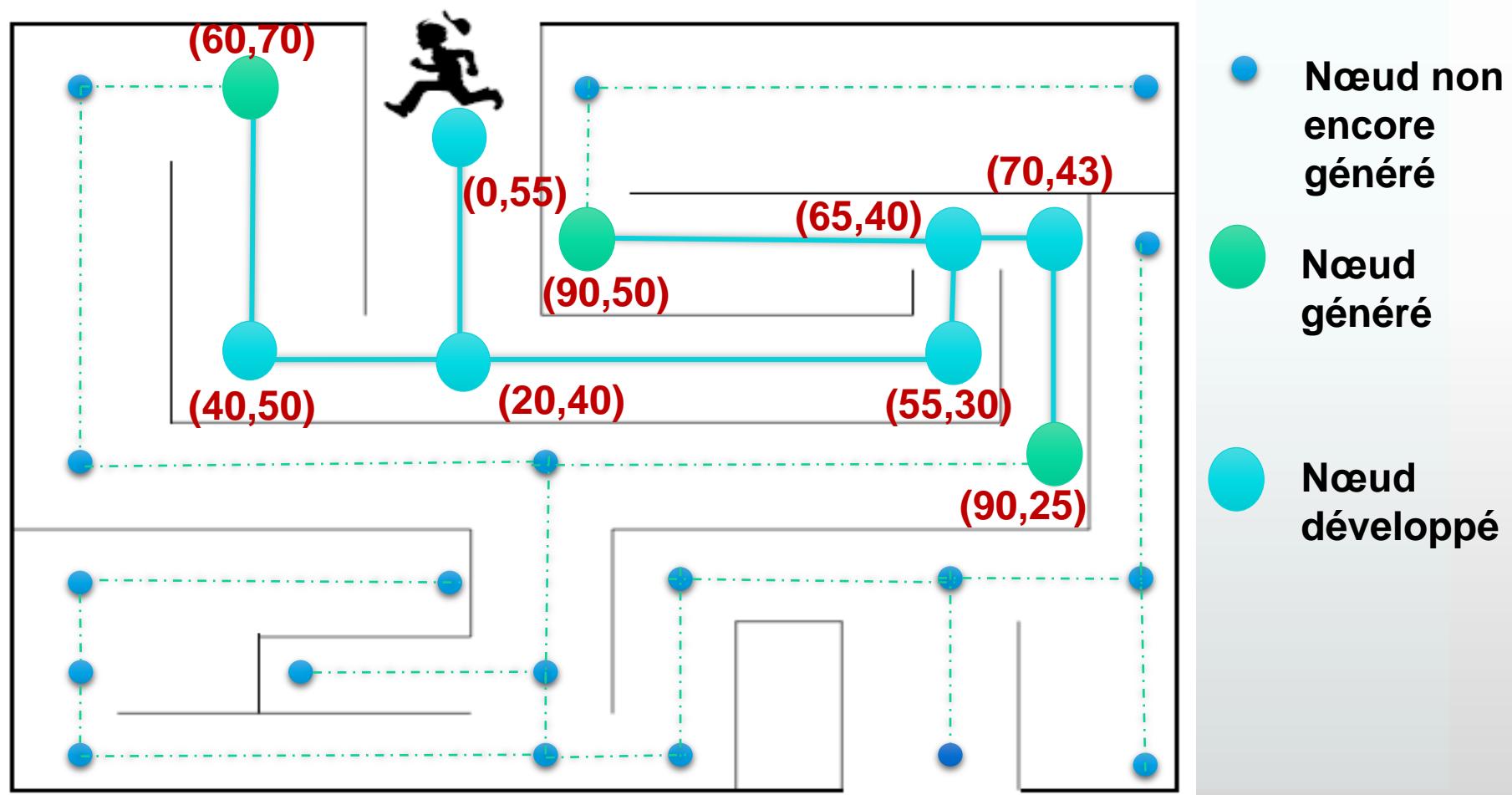
Application (labyrinthe)

(Distance parcourue, Distance estimée)



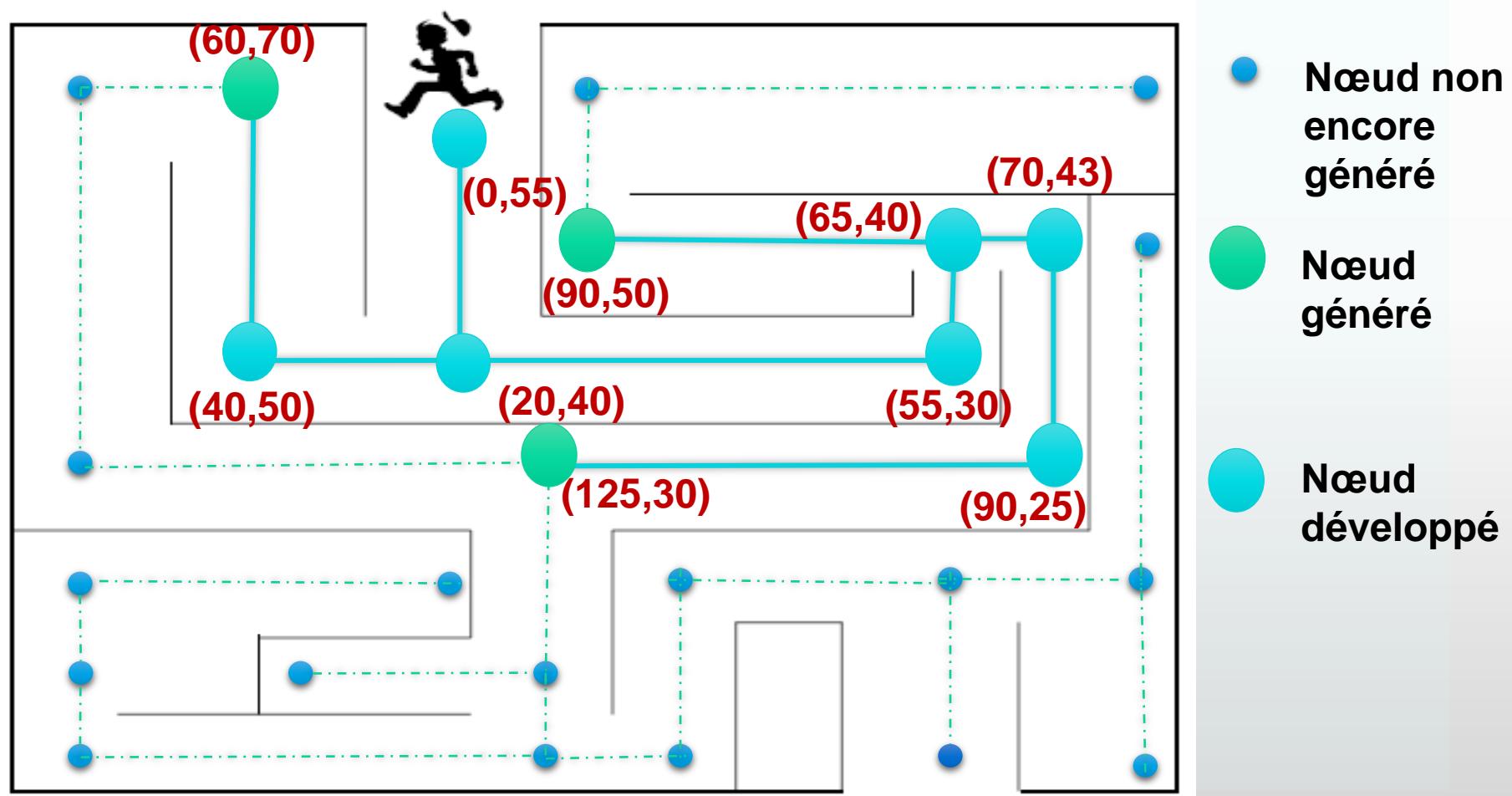
Application (labyrinthe)

(Distance parcourue, Distance estimée)



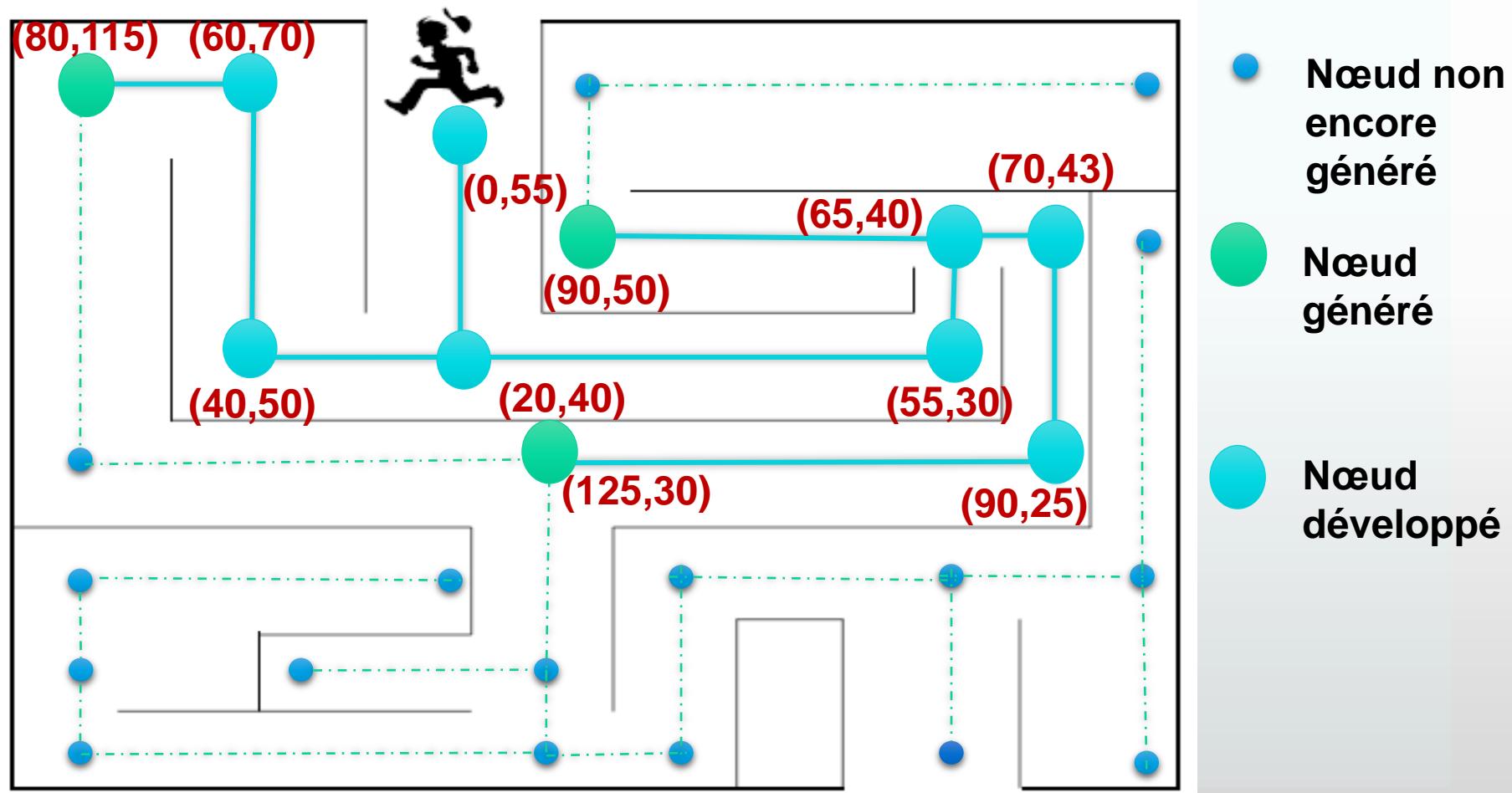
Application (labyrinthe)

(Distance parcourue, Distance estimée)



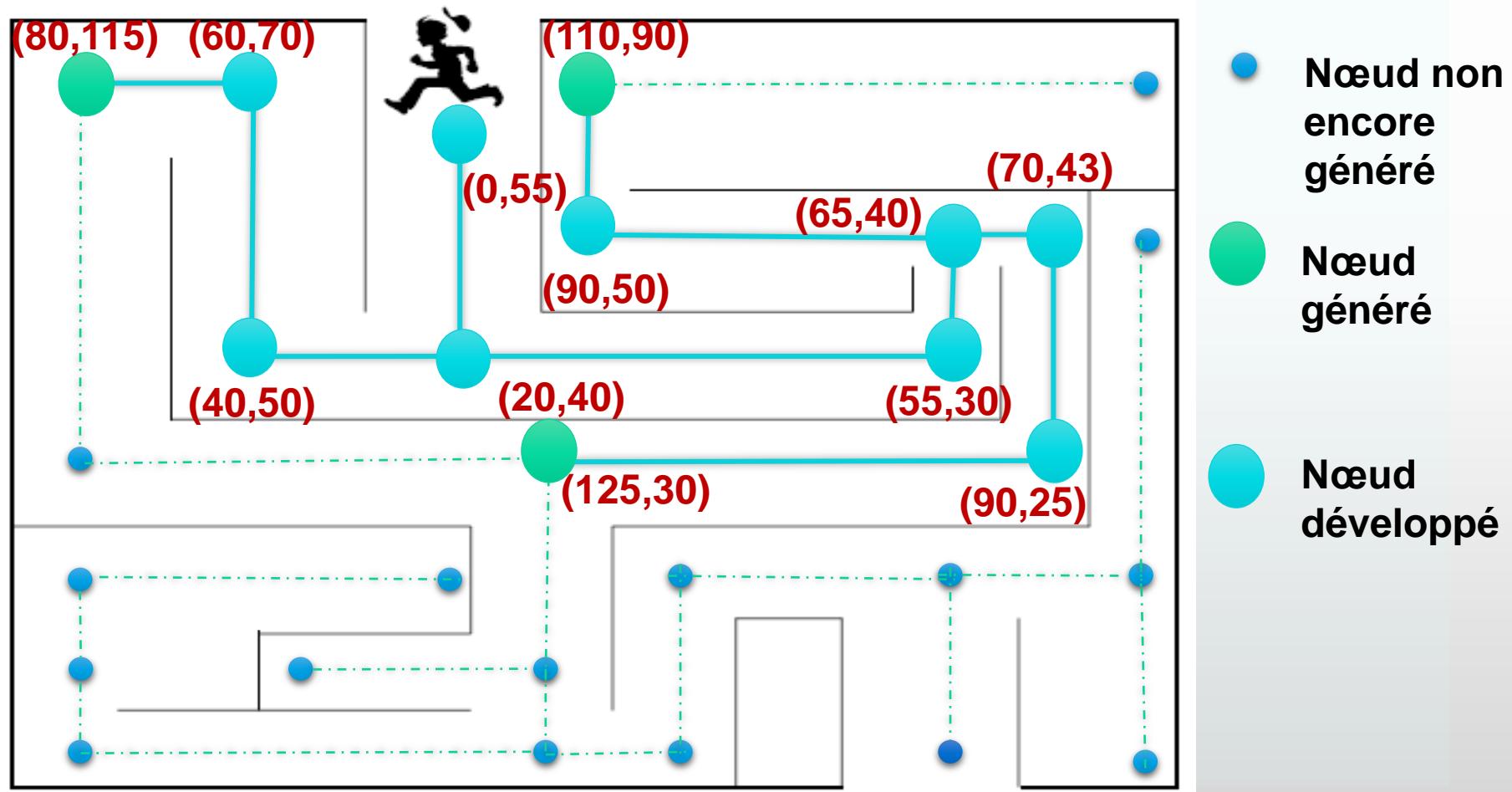
Application (labyrinthe)

(Distance parcourue, Distance estimée)



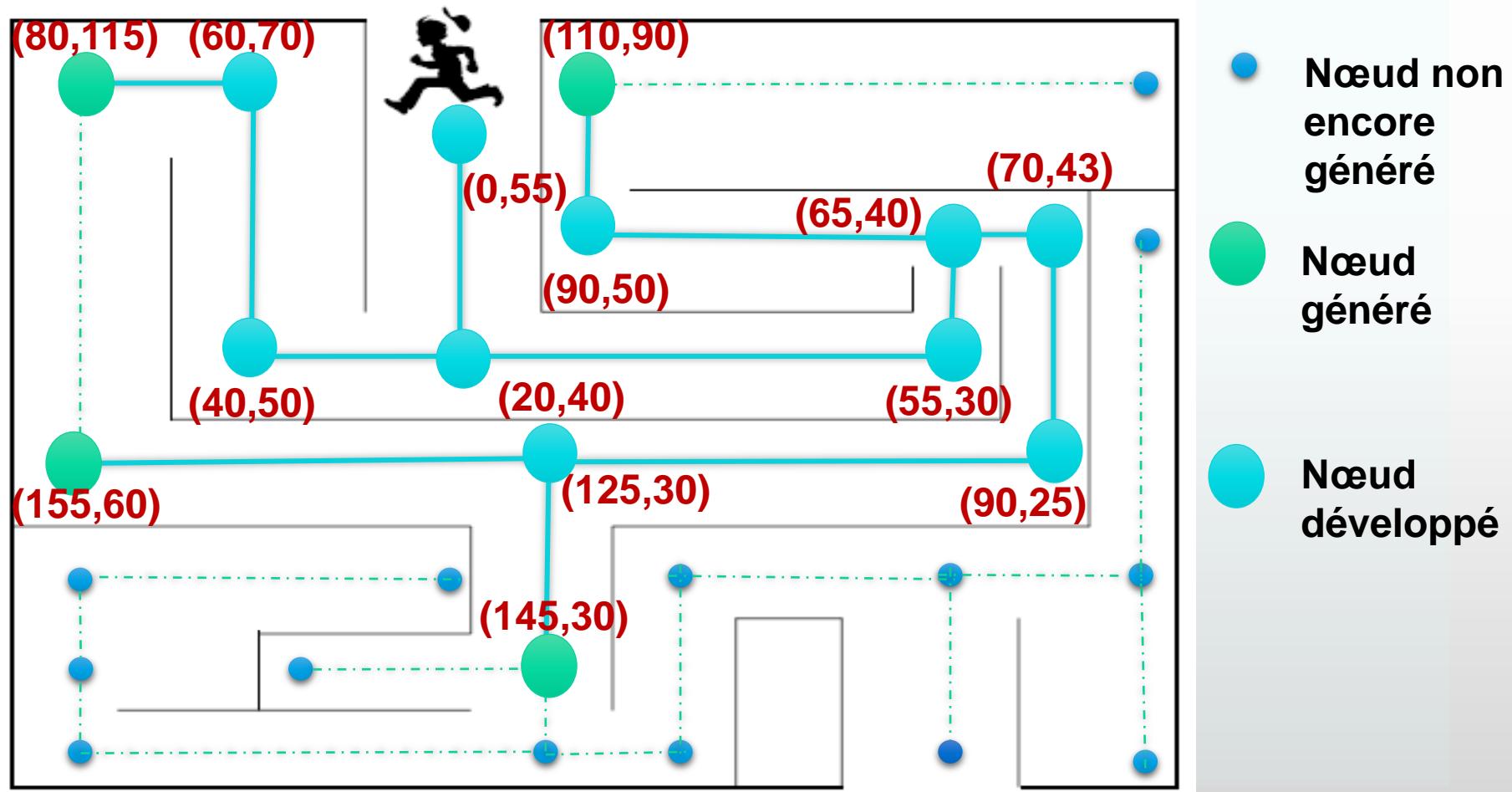
Application (labyrinthe)

(Distance parcourue, Distance estimée)



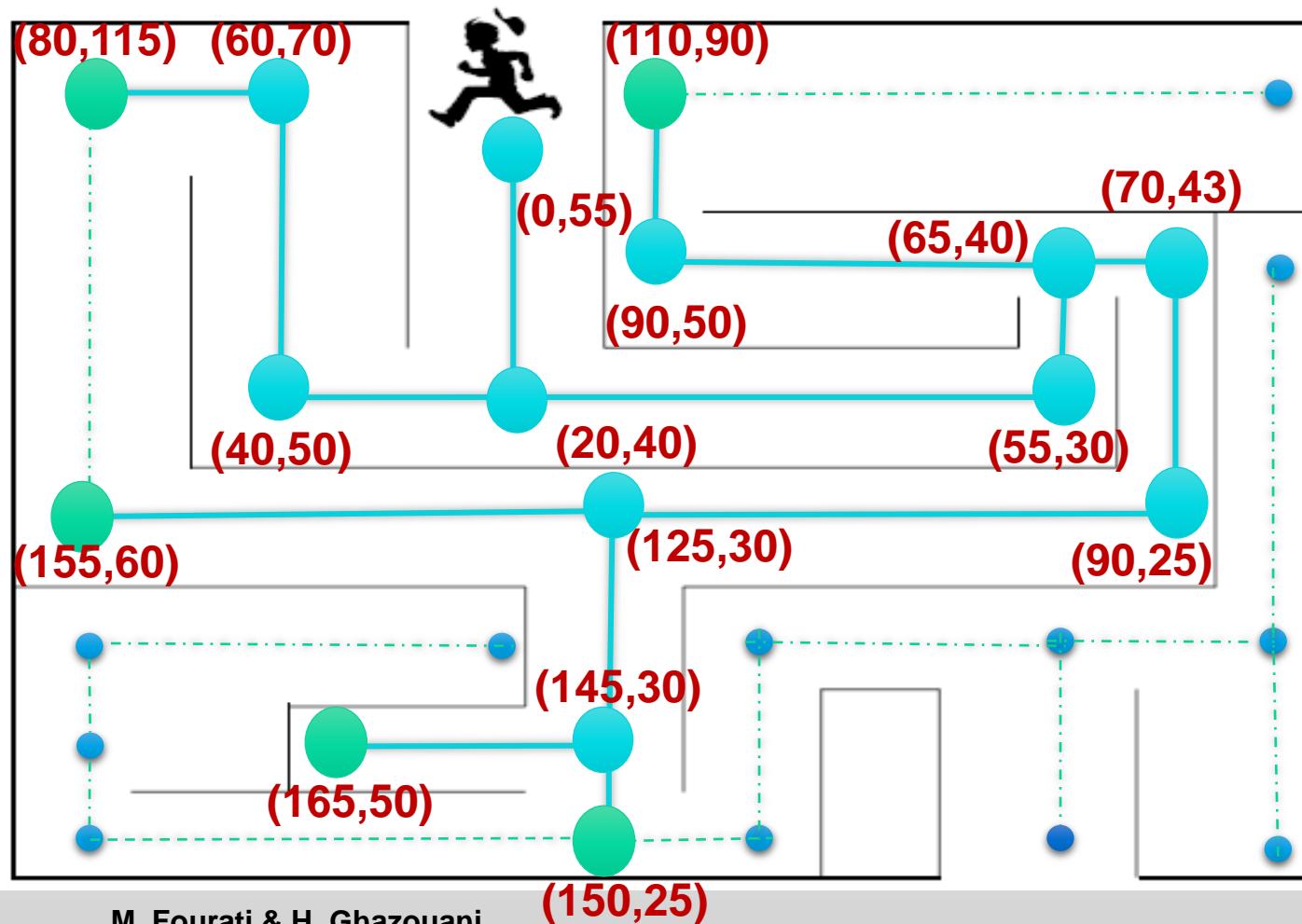
Application (labyrinthe)

(Distance parcourue, Distance estimée)



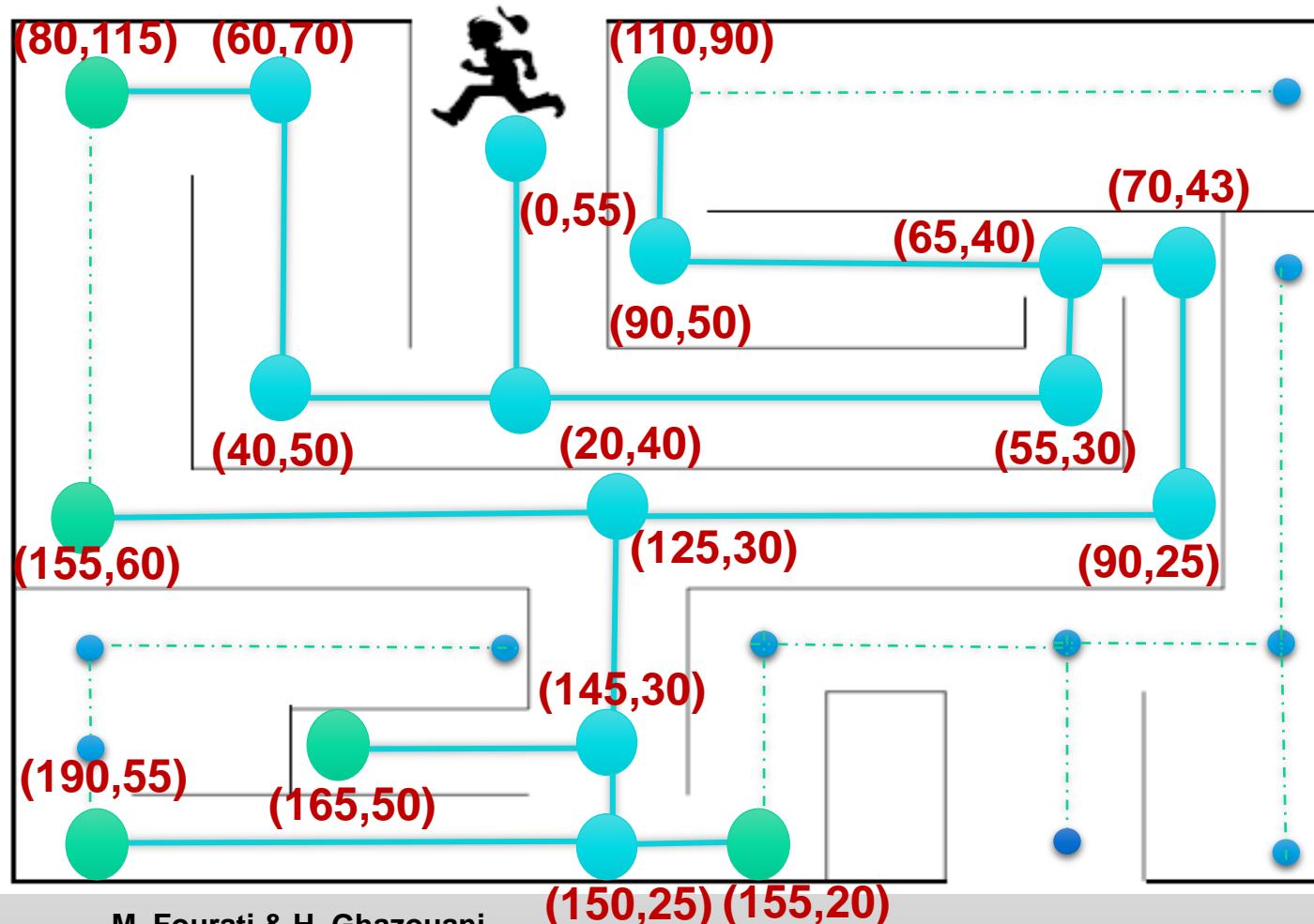
Application (labyrinthe)

(Distance parcourue, Distance estimée)



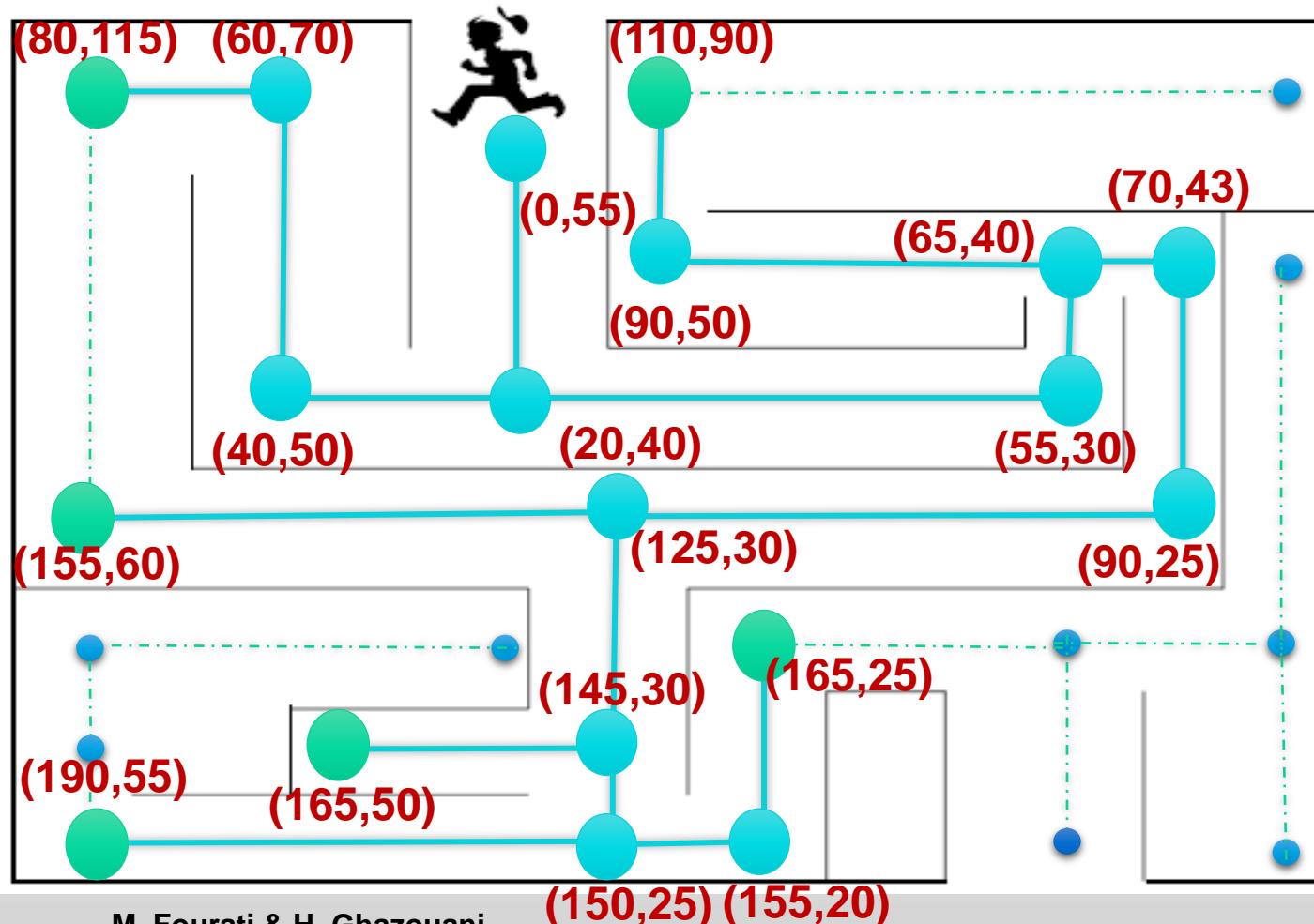
Application (labyrinthe)

(Distance parcourue, Distance estimée)



Application (labyrinthe)

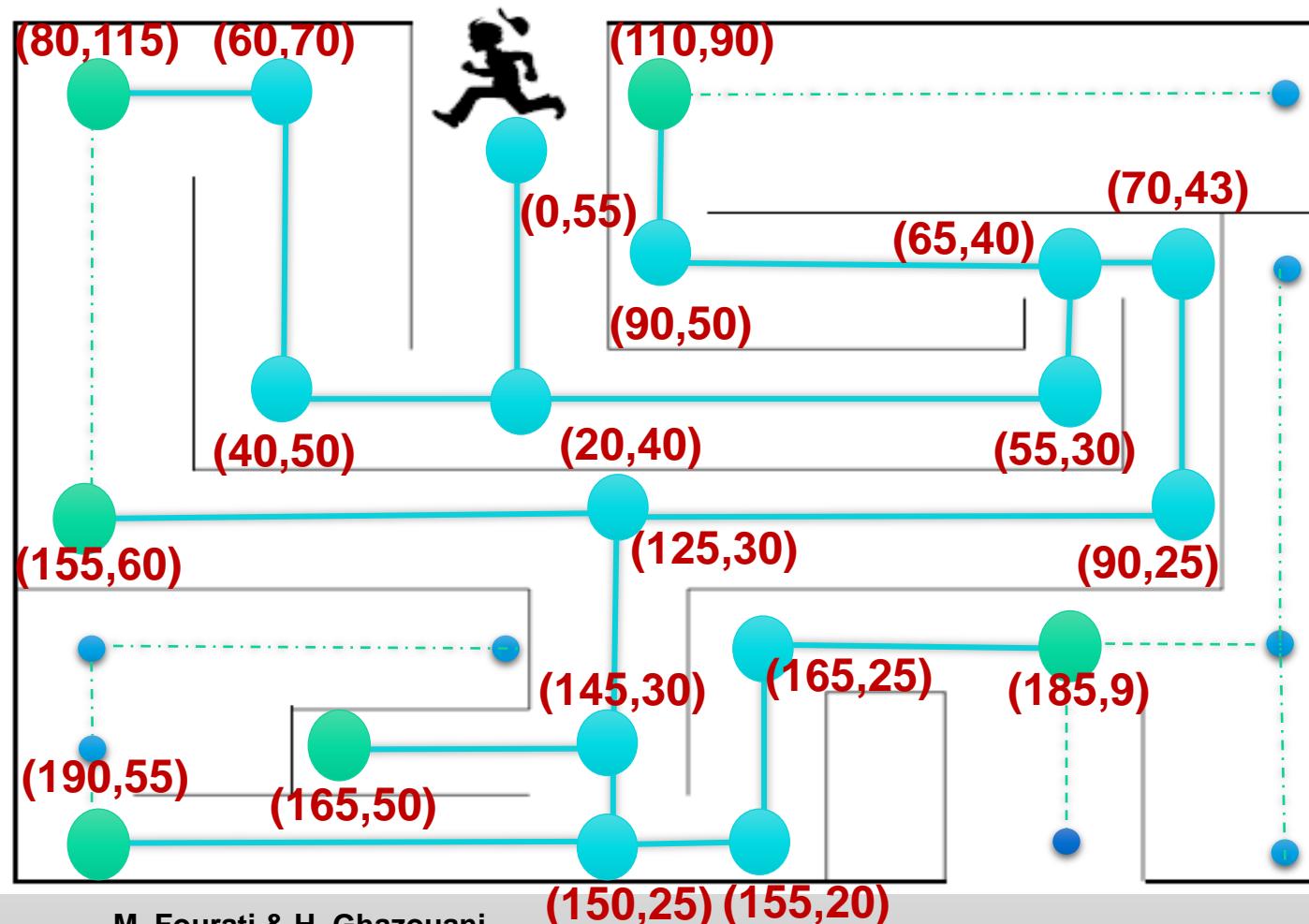
(Distance parcourue, Distance estimée)



- Nœud non encore générée
- Nœud générée
- Nœud développé

Application (labyrinthe)

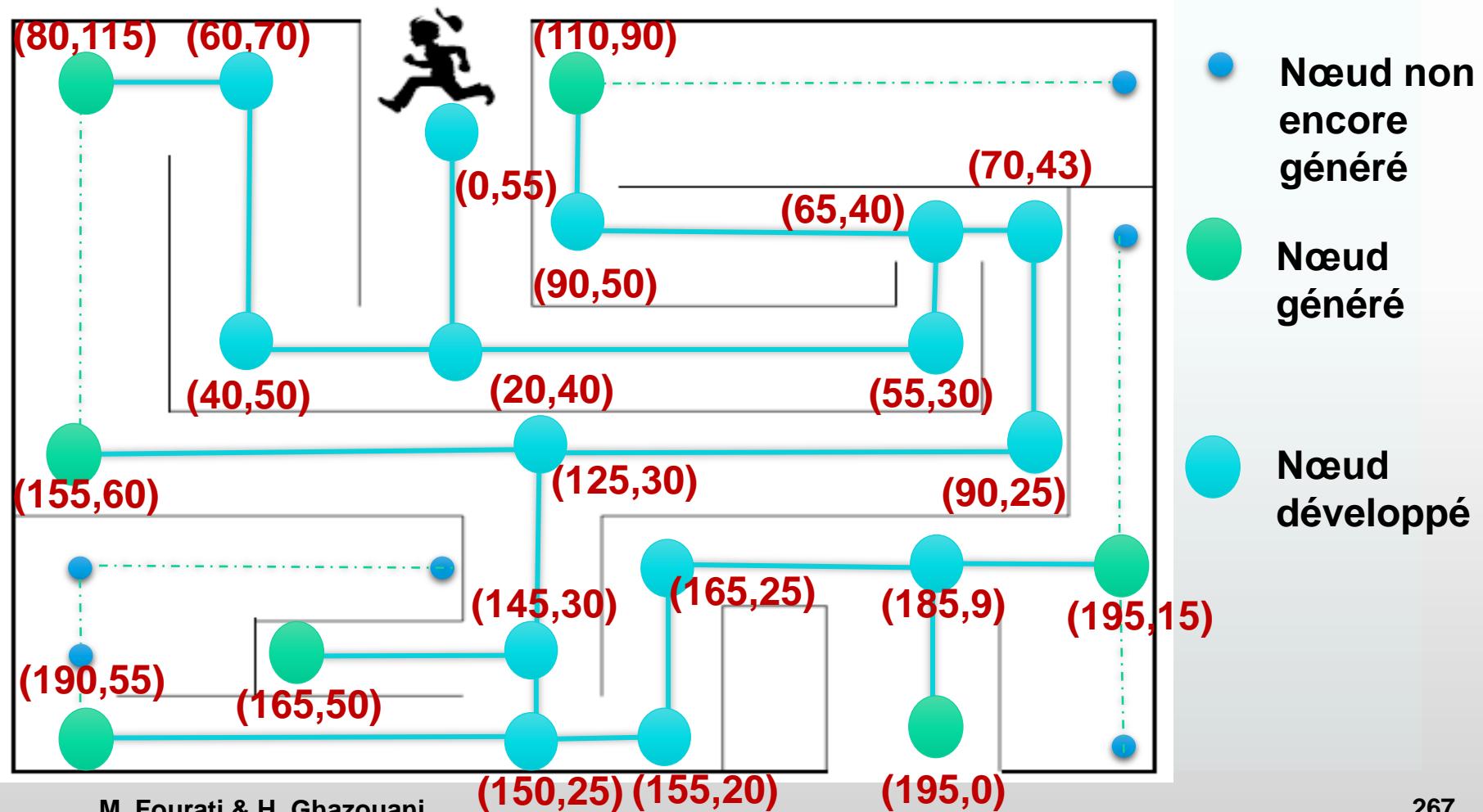
(Distance parcourue, Distance estimée)



- Nœud non encore générée
- Nœud générée
- Nœud développé

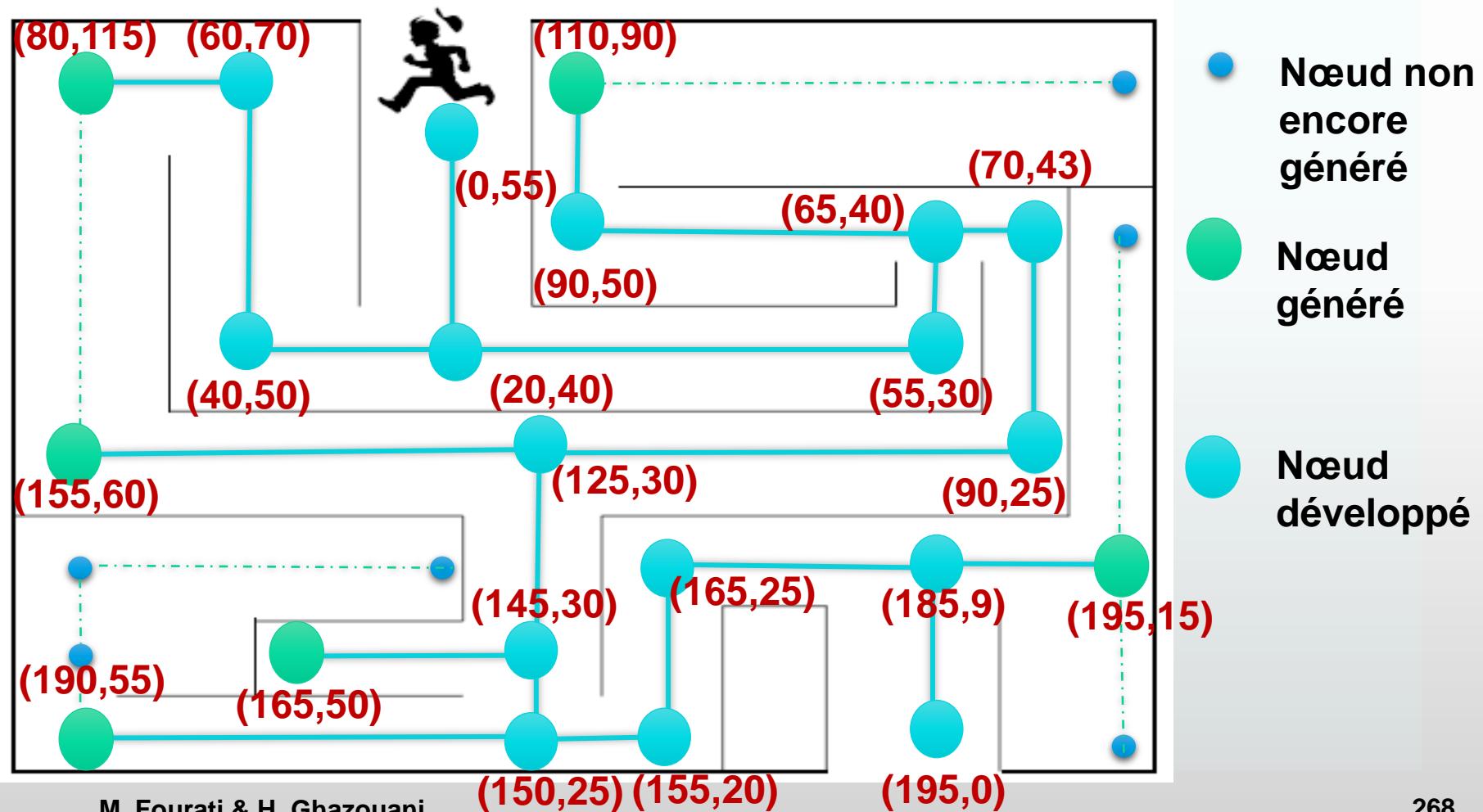
Application (labyrinthe)

(Distance parcourue, Distance estimée)



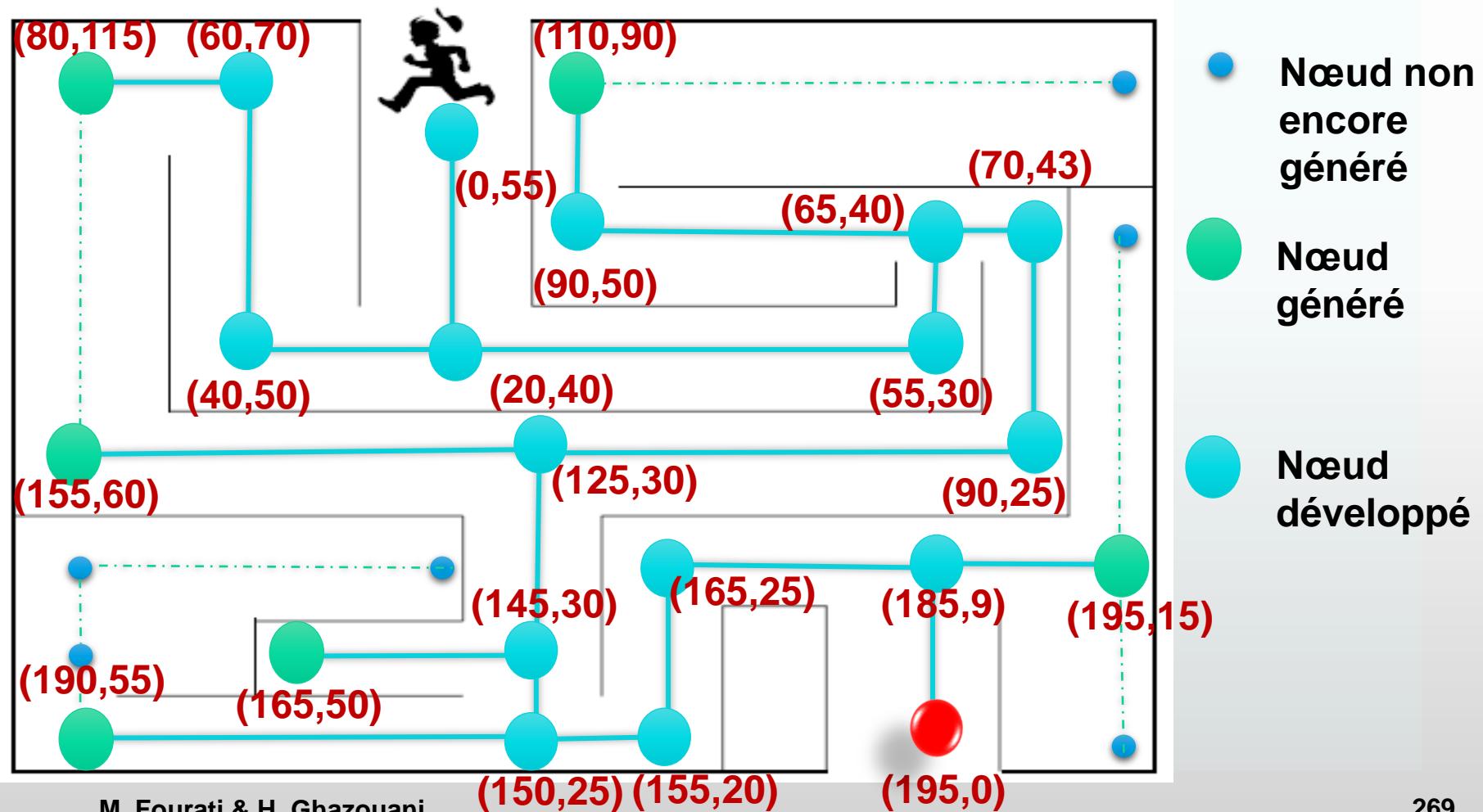
Application (labyrinthe)

(Distance parcourue, Distance estimée)



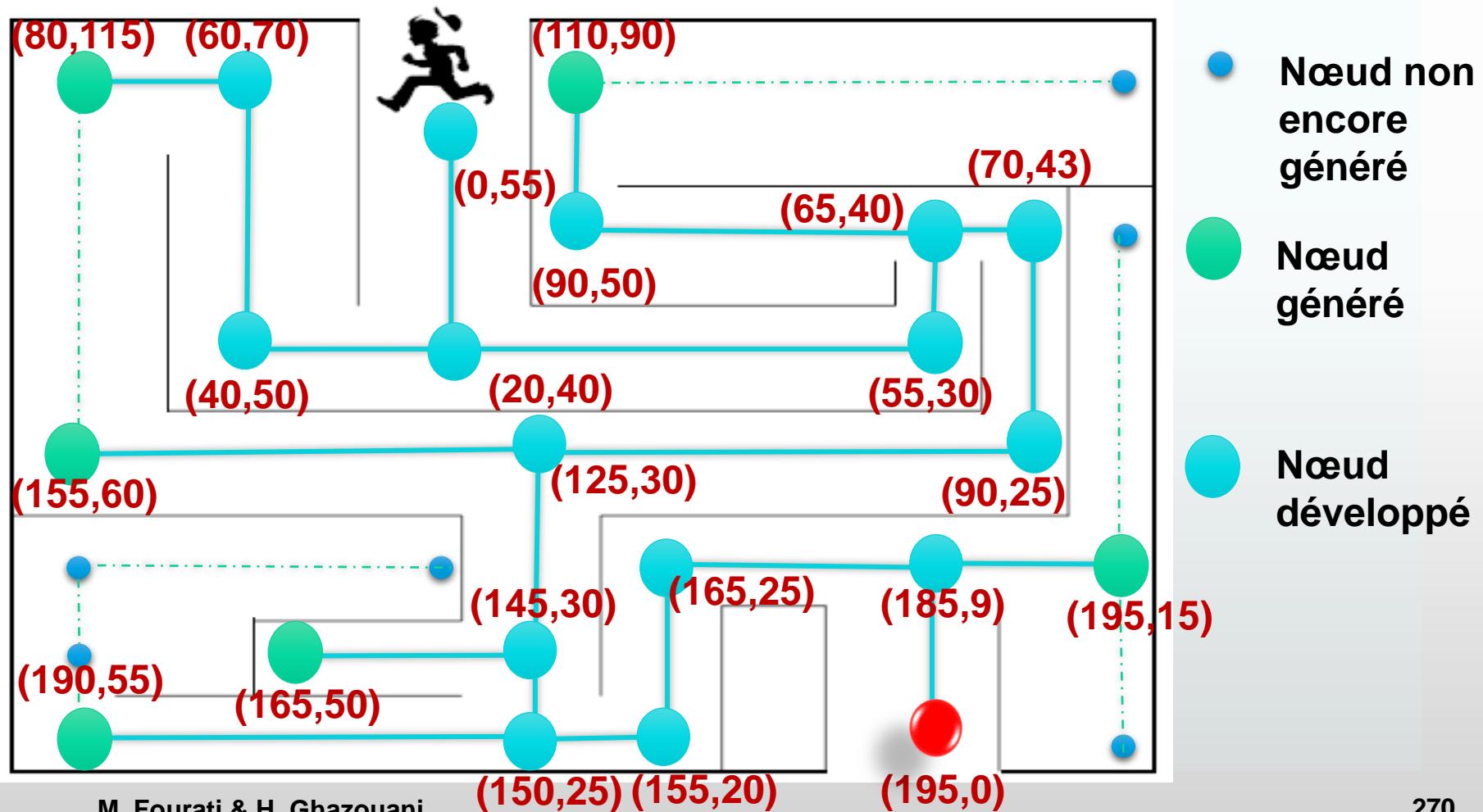
Application (labyrinthe)

(Distance parcourue, Distance estimée)



Application (labyrinthe)

(Distance parcourue, Distance estimée)



Exploration heuristique à mémoire limitée :

IDA*

RBFS,

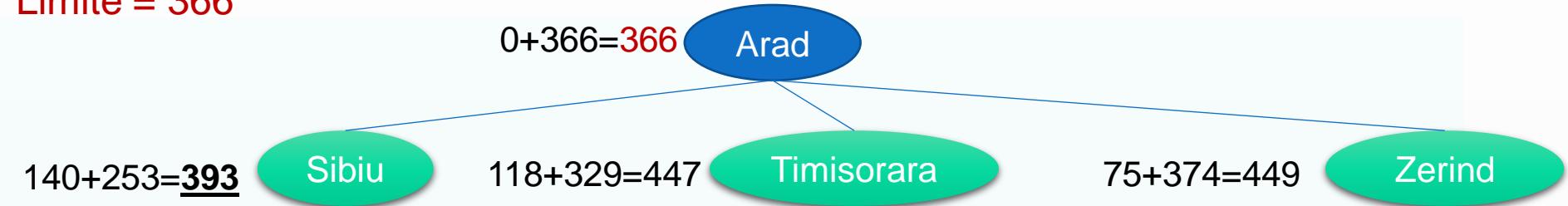
MA*

A* avec approfondissement itératif (IDA*)

- Algorithme par approfondissement itératif avec la limite est le coût f et non la profondeur.
- A chaque itération la profondeur est fixée par la valeur de f la plus basse, mais qui dépasse la limite de l'itération précédente.
- Prend moins de mémoire, comparable à l'exploration en profondeur d'abord et évite le maintien d'une liste triée.

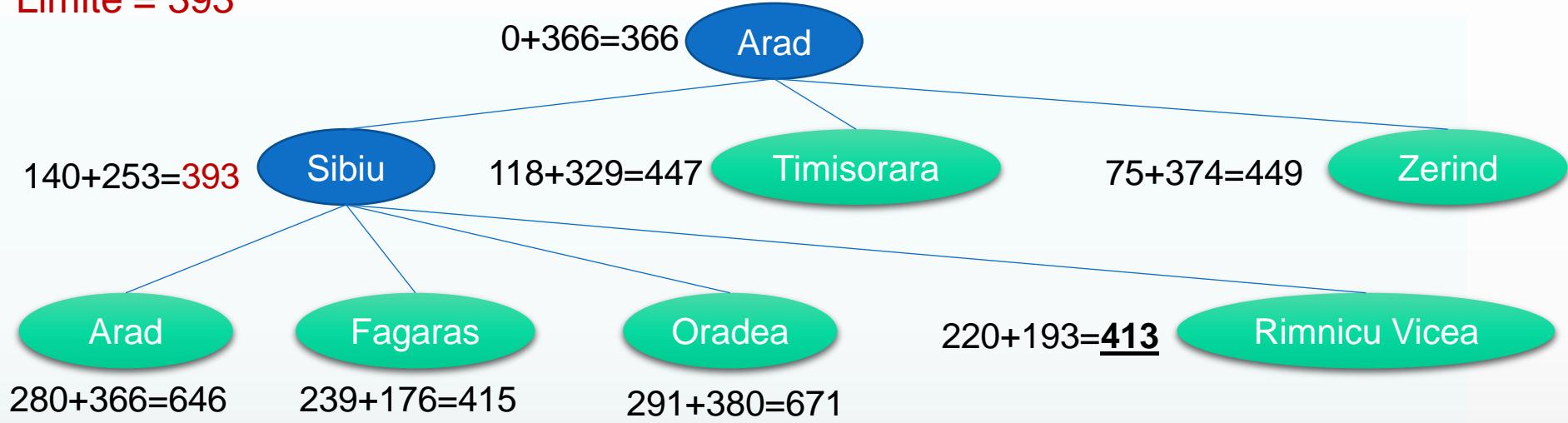
Application IDA*

Limite = 366



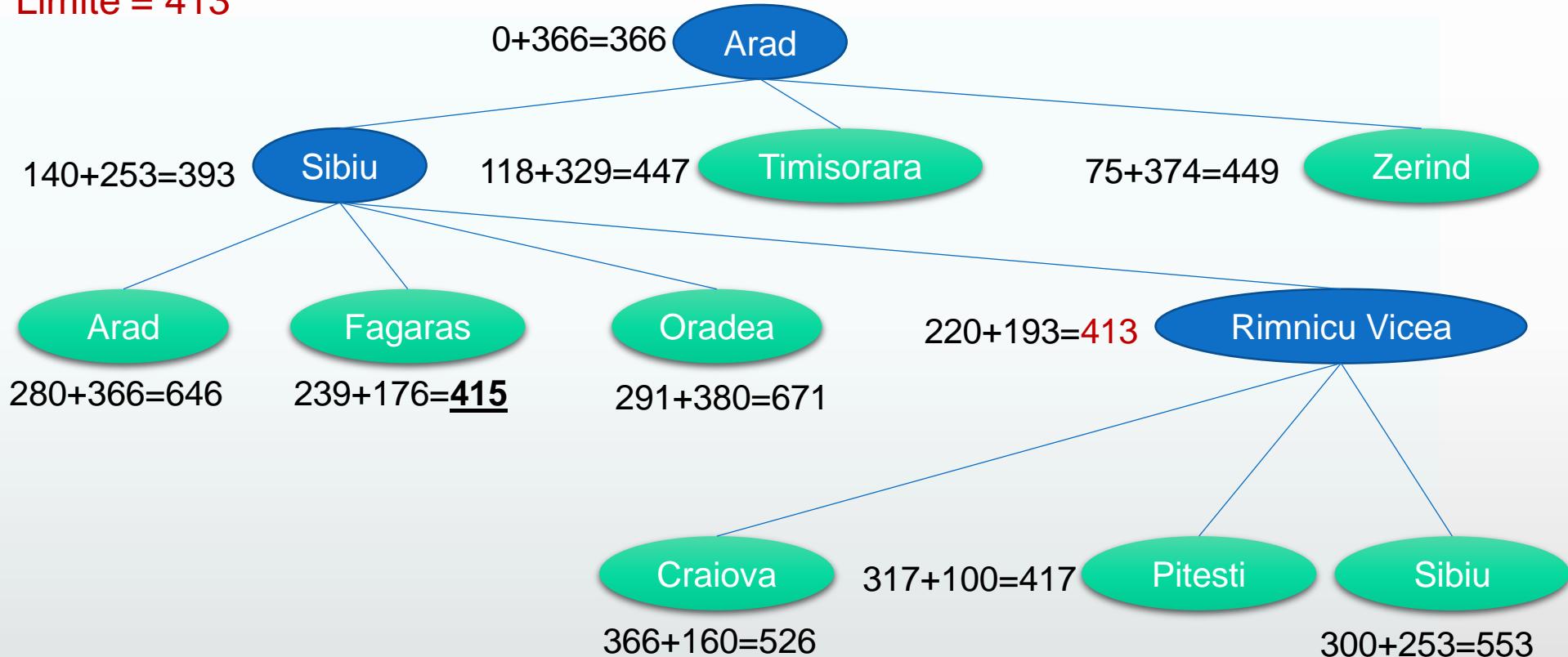
Application IDA*

Limite = 393



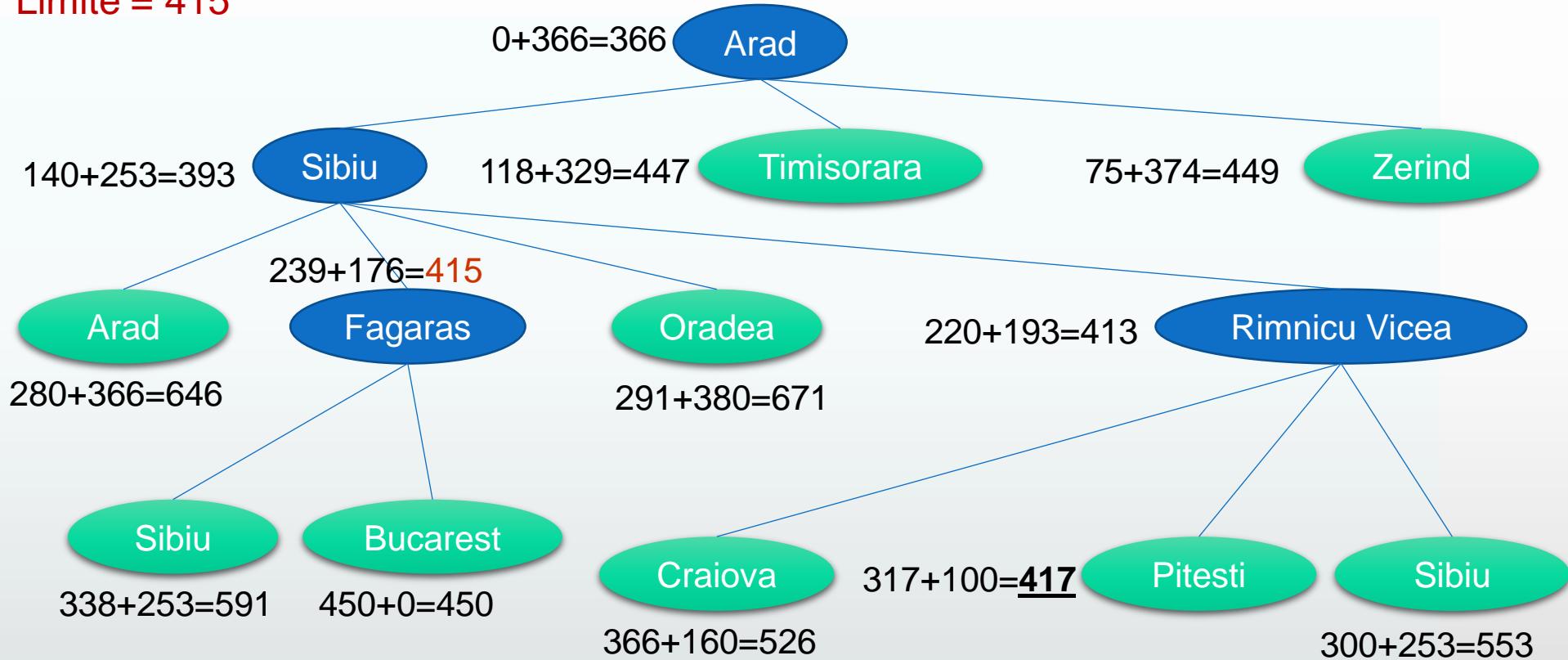
Application IDA*

Limite = 413



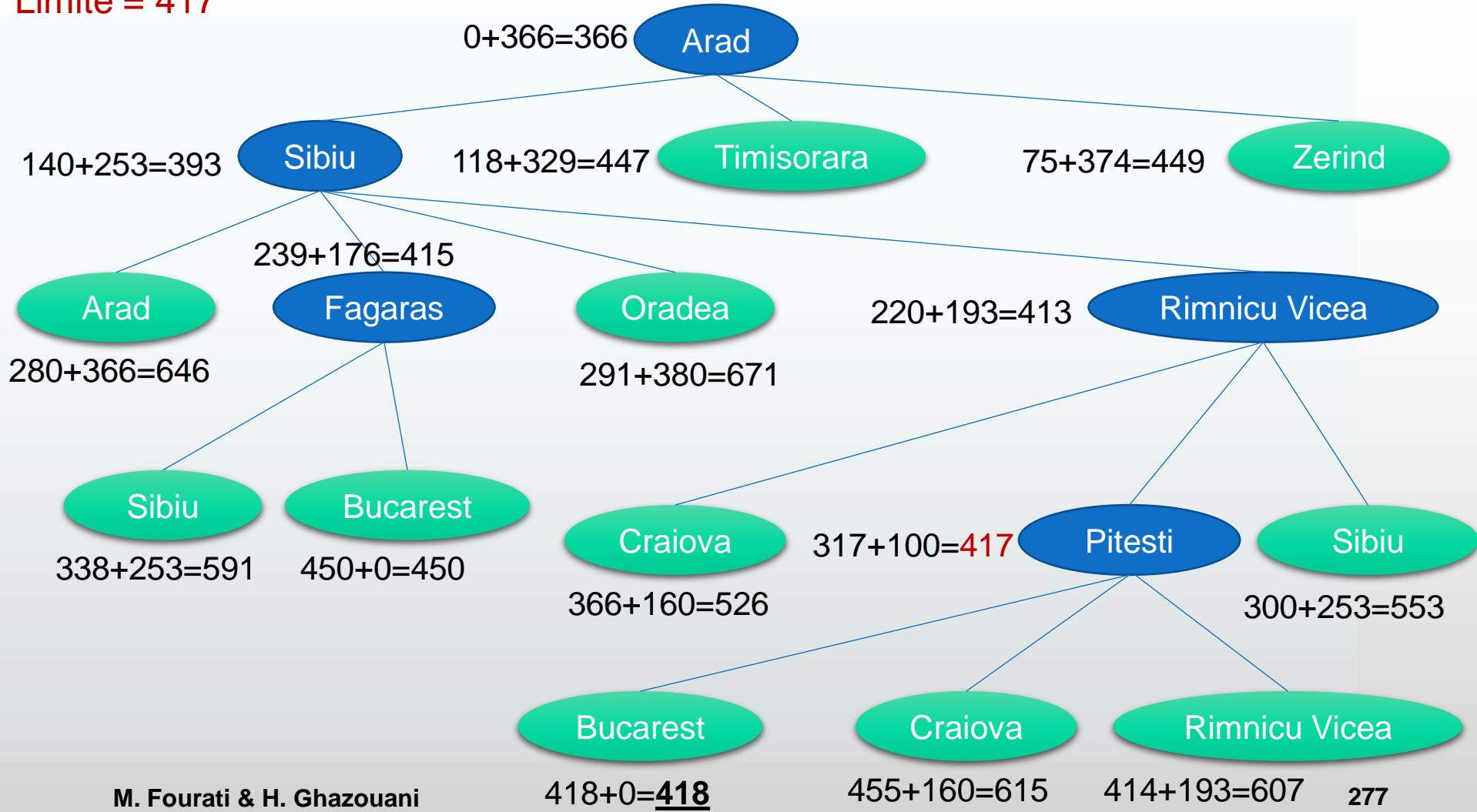
Application IDA*

Limite = 415



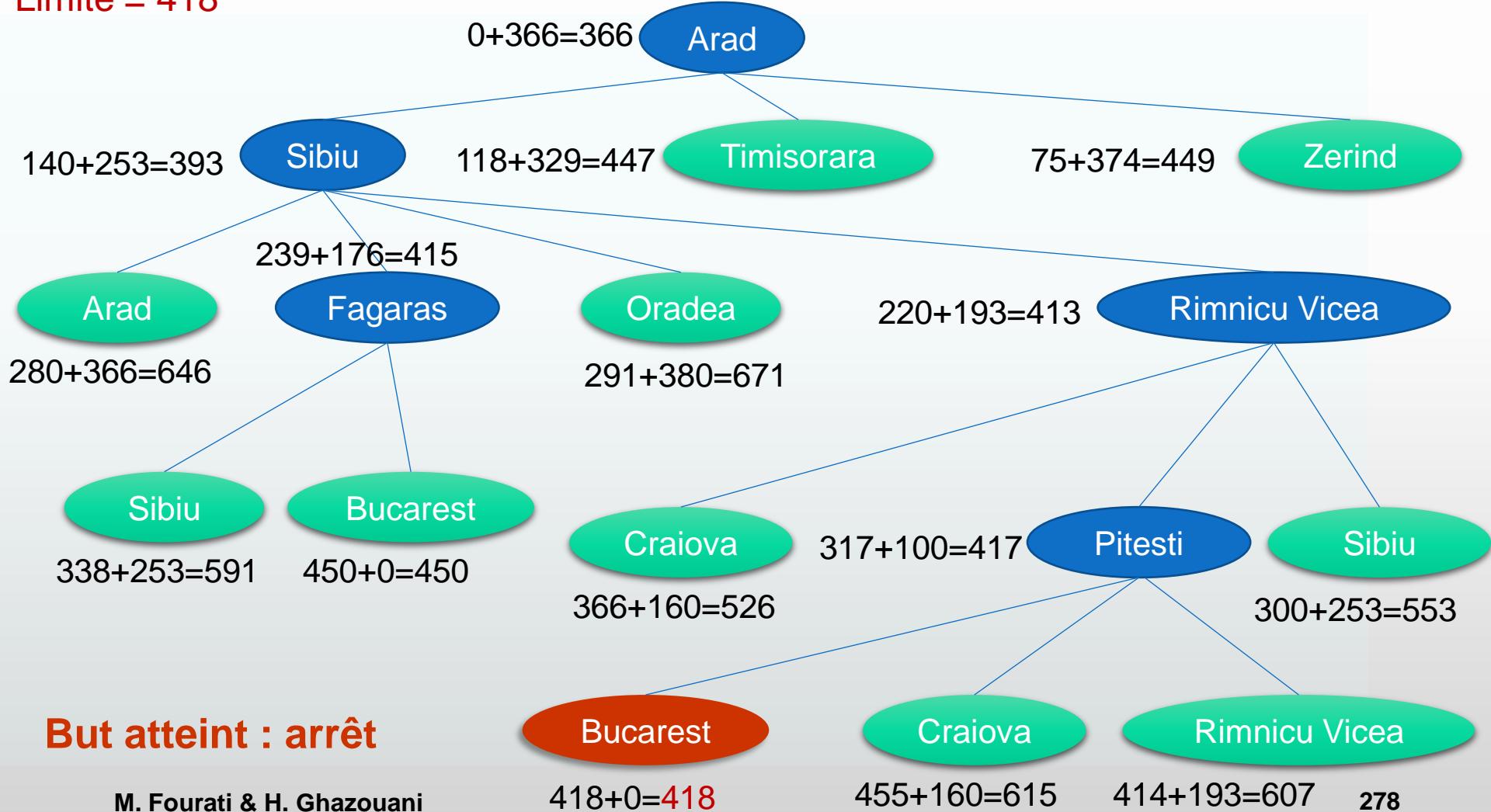
Application IDA*

Limite = 417



Application IDA*

Limite = 418

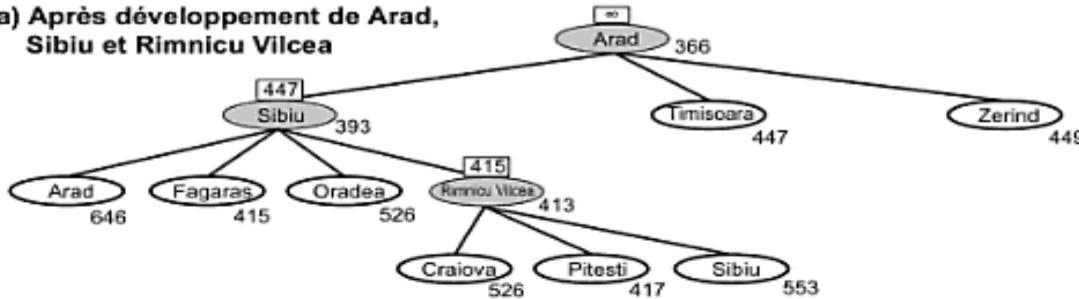


Exploration récursive par le meilleur d'abord (RBFS)

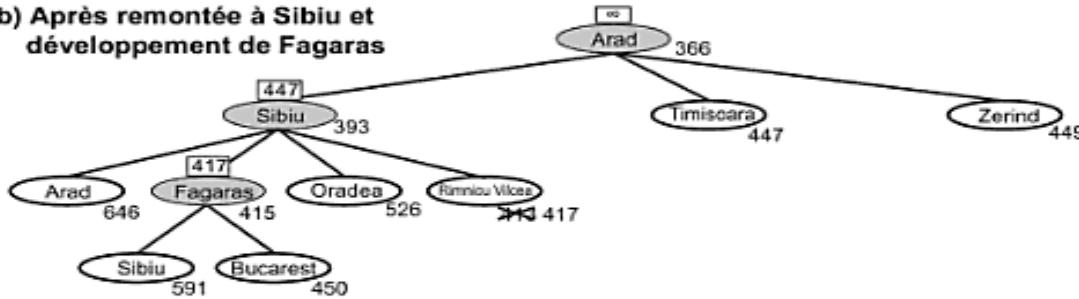
- Variante de l'exploration par le meilleur d'abord.
- Utilise une espace de temps linéaire.
- L'exploration se fait récursivement en profondeur d'abord.
- Mais ne continue pas indéfiniment dans un chemin, elle pause une limite correspondant à la valeur f du meilleur chemin alternatif parmi les ancêtres.

Exploration RBFS pour Bucarest

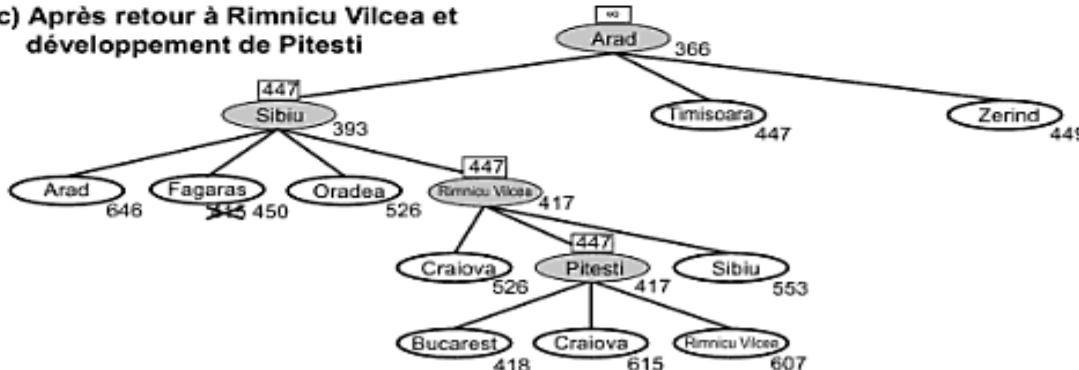
(a) Après développement de Arad, Sibiu et Rimnicu Vilcea



(b) Après remontée à Sibiu et développement de Fagaras



(c) Après retour à Rimnicu Vilcea et développement de Pitesti



Performances de l'exploration RBFS

- ❑ Un peu plus efficace que IDA*; mais toujours génération excessive de nœuds.
- ❑ Optimale si $h(n)$ est admissible.
- ❑ Complexité en espace linéaire : $O(bd)$.
- ❑ Complexité en temps : dépend de l'exactitude de la fonction heuristique et de la fréquence des changements de chemins.
- ❑ Mais comme IDA* il sous-utilise l'espace mémoire.

Exploration SMA* (Simple Memory-Bounded A*)

- ❑ Meilleure utilisation de la mémoire disponible.
- ❑ Pareil que A* sauf qu'il développe la meilleure feuille jusqu'à ce que la mémoire soit pleine.
- ❑ Pour libérer de l'espace, elle supprime le nœud feuille le moins favorable.
- ❑ Mémorise la valeur du nœud oublié au niveau du parent, il pourra alors régénérer ce sous-arbre s'il redeviendra le meilleur.

Performances de l'exploration SMA*

- Complétude : oui s'il existe une solution accessible ($d \leq \text{la capacité de stockage}$).
- Optimalité : oui une solution optimale est accessible.

Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ Stratégies d'explorations systématiques :
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ **Fonctions heuristiques**
- ❑ Stratégies d'explorations locales

Fonction heuristique

Jeu de Taquin

- ❑ Coût moyen pour la solution : 22 étapes.

- ❑ $b = \text{en moyenne } 3.$

7	2	4
5		6
8	3	1

État initial

	1	2
3	4	5
6	7	8

État final

- ❑ Pour une exploration d'arbre exhaustive à une profondeur 22, examine $3^{22} \approx 3.1 \times 10^{10}$ états.
- ❑ Pour une exploration en graphes : une diminution par un facteur de 170000 états.
- ❑ Un nombre raisonnable, mais pour le taquin 15, il faut penser à utiliser une bonne fonction heuristique.

Heuristiques pour le jeu de Taquin

- Deux heuristiques admissibles pour le jeu de taquin :
 - h_1 =nombre de pièces mal placées :
 $h_1=8$ pour l'état initial.
 - h_2 =somme des distances des pièces par rapport à leur position but
(distance de Manhattan ou distance city-block) :
($h_2=3+1+2+2+2+3+3+2=18$ pour l'état initial).

Dominance

- On dit que h_2 **domine** h_1 si h_1 et h_2 sont admissibles et que $h^*(n) \geq h_2(n) \geq h_1(n)$.
- h_2 est alors meilleure pour la recherche, car généralement A^* avec h_2 ne développera pas plus de nœuds que A^* avec h_1 .
- Que peut ont dire des heuristiques de l'exemple précédent ?

Production d'heuristiques admissibles

- Problèmes relaxés
- Base de données de motifs
- Apprentissage

Production d'heuristiques pour les problèmes relaxés

- ❑ L'heuristique est une solution exacte pour le problème simplifié.
- ❑ Exemple du jeu de Taquin :
 - ❑ Problème réel : une pièce peut passer de la case A à la case B si A est adjacent à B et si B est vide.
 - ❑ Problèmes relaxés :
 - une pièce peut passer de la case A à B si A est adjacent a B (h2).
 - une pièce peut passer de la case A à la case B si B est vide.
 - une pièce peut passer de la case A a la case B (h1).
- ❑ Le programme **OBSERVER** produit automatiquement des heuristiques en se basant sur des problèmes relaxés, il a produit une meilleure heuristique pour le taquin 8 et la première heuristique pour le Rubik's cube.



Production d'heuristiques à partir de sous-problèmes

- ❑ L'heuristique est obtenue à partir du coût de la solution d'un sous-problème.
- ❑ Stocker, dans une base de données de motifs, des coûts exacts de solutions qui sont des instances de sous-problèmes.

7	2	4
5		6
8	3	1

État initial

	1	2
3	4	5
6	7	8

État final

Sous
problème
de
l'instance



*	2	4
*		*
*	3	1

État initial

	1	2
3	4	*
*	*	*

État final

Production d'heuristiques à partir de sous-problèmes

- La base de données est construite par exploration en arrière à partir du but et en enregistrant le coût de chaque nouveau motif rencontré.
- Cette heuristique réduit le nombre de nœuds générés d'un facteur 1000 pour le taquin à 15 pièces par rapport à la distance de Manhattan.

Apprentissage d'heuristique

- Développer des heuristiques à partir du coût des solutions optimales.
- Appliquer des algorithmes d'**apprentissage** afin de construire une fonction $h(n)$ qui prédit les coûts des solutions pour d'autres états qui se produisent pendant l'exploration.
- Des techniques basées sur les réseaux de neurones, d'arbres de décision et d'autres méthodes d'apprentissage sont adoptées.

Plan

- ❑ Agents de résolution de problèmes
- ❑ Problèmes bien définis
- ❑ Exemples de problèmes
- ❑ Recherche de solutions
- ❑ Stratégies d'explorations systématiques :
 - ❑ Non informées,
 - ❑ Informées (heuristiques)
- ❑ Fonctions heuristiques
- ❑ **Stratégies d'explorations locales**

Exploration systématique

- Parcours d'espaces d'états pour trouver un chemin vers la solution (le chemin est aussi une solution).
- L'environnement est observable, déterministe, statique et entièrement connu.
- Il est possible d'explorer **systématiquement** tous les chemins à partir de l'état initial.

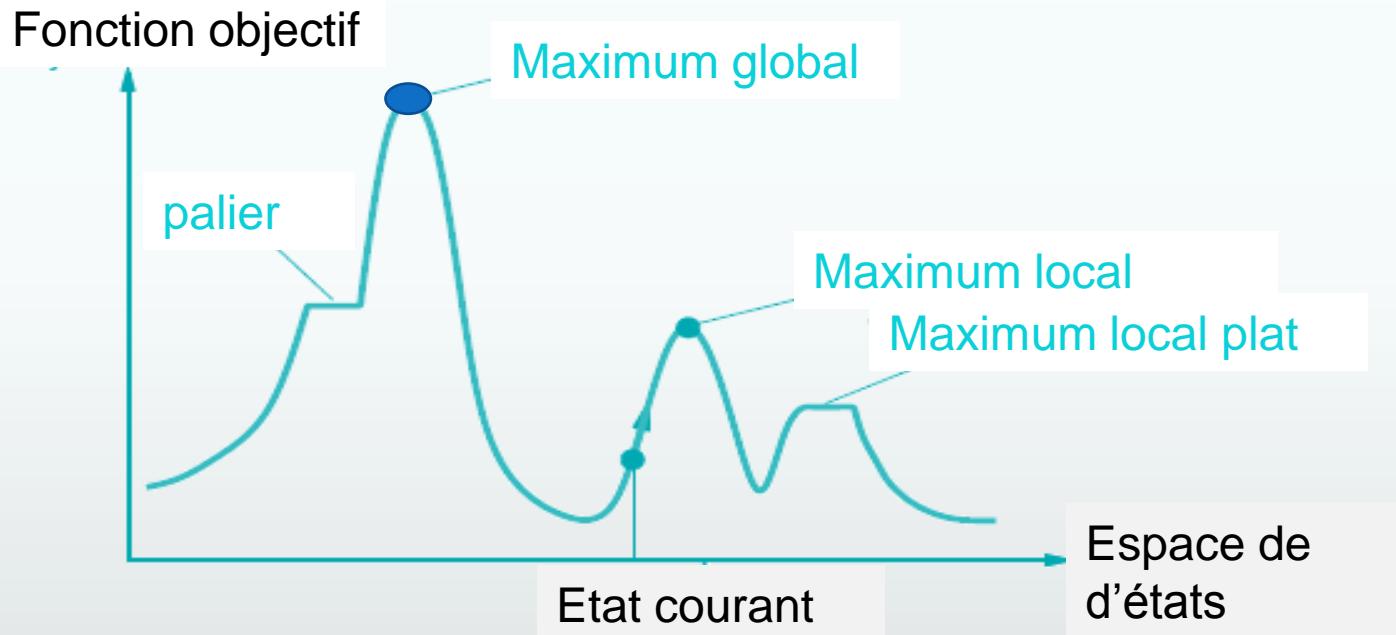
Exploration locale

- ❑ Si la solution recherchée est l'état optimal (ou proche) et non le chemin qui y mène, le chemin vers le but n'est pas important.
- ❑ Problème de jouets : jeu des 8 dames.
- ❑ Problèmes réels : Conception de circuits intégrés, ordonnancement, optimisation de réseaux de télécommunication, routage.

Exploration locale

- ❑ Elle opère en ne considérant qu'un seul nœud courant et non plusieurs chemins.
- ❑ Ne s'occupe que des voisins du nœud choisi.
- ❑ Les chemins suivis ne sont généralement pas mémorisés.
- ❑ Utiles pour résoudre des problèmes d'**optimisation** où l'objectif est de trouver la meilleure solution selon une **fonction objectif** (à minimiser ou à maximiser).
- ❑ Consomment peu de mémoire et trouvent souvent des solutions raisonnables dans des espaces d'états infinis.

Paysage de l'espace d'états pour une exploration locale



- Un algorithme d'exploration locale est complet s'il trouve toujours un but et est optimal s'il trouve un minimum ou un maximum global.

Algorithmes pour l'exploration locale

- Hill-climbing
- Recuit simulé
- Locale en faisceau
- Génétiques

Exploration par escalade « Hill-climbing »

Exploration par escalade « Hill-climbing »

□ But : « *Atteindre le sommet du Mont Everest dans un épais brouillard tout en souffrant d'amnésie* ».

fonction ESCALADE(*problème*) **retourne** un état qui est un maximum local
courant \leftarrow CRÉER-NŒUD(*problème.ÉTAT-INITIAL*)
faire en boucle
 voisin \leftarrow un successeur de *courant* ayant la valeur la plus élevée
 si *voisin.VALEUR* \leq *courant.VALEUR* **alors retourner** *courant.ÉTAT*
 courant \leftarrow *voisin*

Fonctionnement

❑ Entrée :

- ❑ État initial.
- ❑ Fonction à optimiser (fonction objectif/coût).
- ❑ Fonction permettant de générer les états successeurs.

❑ Algorithme :

- ❑ Le nœud courant est initialisé à l'état initial.
- ❑ Itérativement, le nœud courant est comparé à ses successeurs immédiats.
 - Le meilleur voisin immédiat et ayant la plus grande valeur (selon la fonction objectif) que le nœud courant, devient le nœud courant.
 - Si un tel voisin n'existe pas, on arrête et on retourne le nœud courant comme solution.

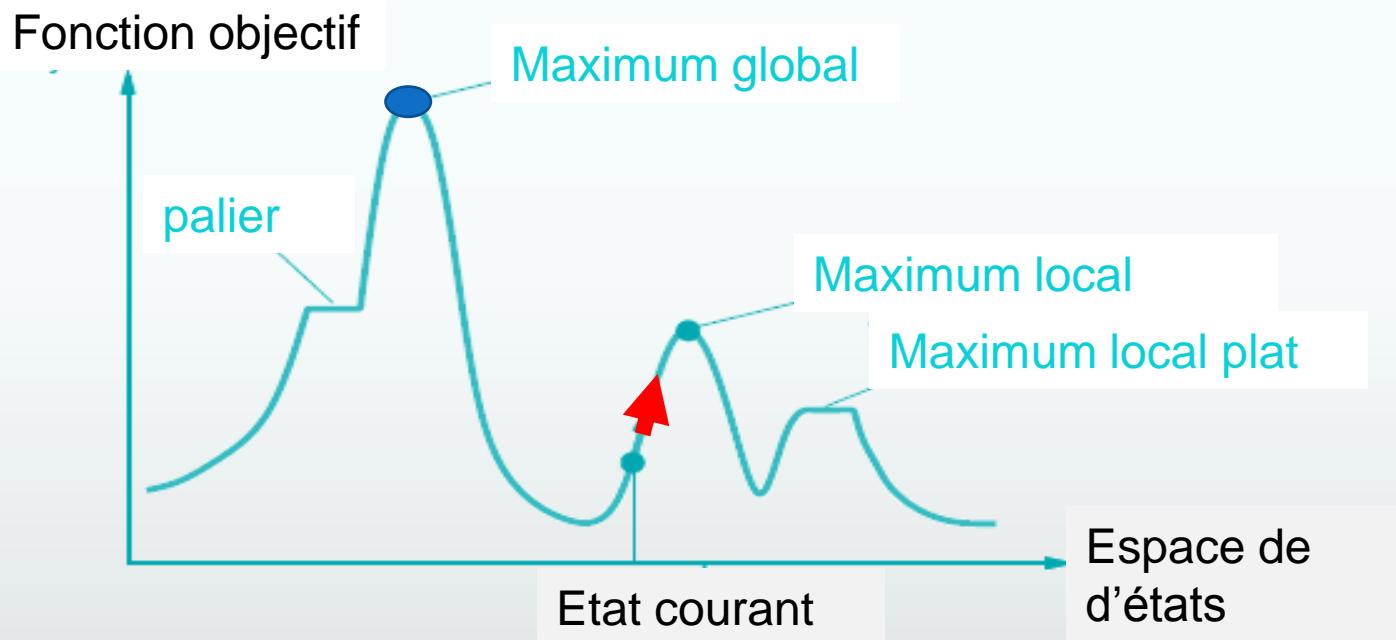
Exploration par escalade

- ❑ Une boucle qui se déplace vers les valeurs croissantes guidée par la fonction objectif.
- ❑ S'arrête quant elle atteint un pic avec aucun voisin avec une valeur plus élevée.
- ❑ Pas de maintien d'un arbre d'exploration, uniquement l'état courant et la fonction objectif sont enregistrés.
- ❑ Uniquement les voisins immédiats sont observés.

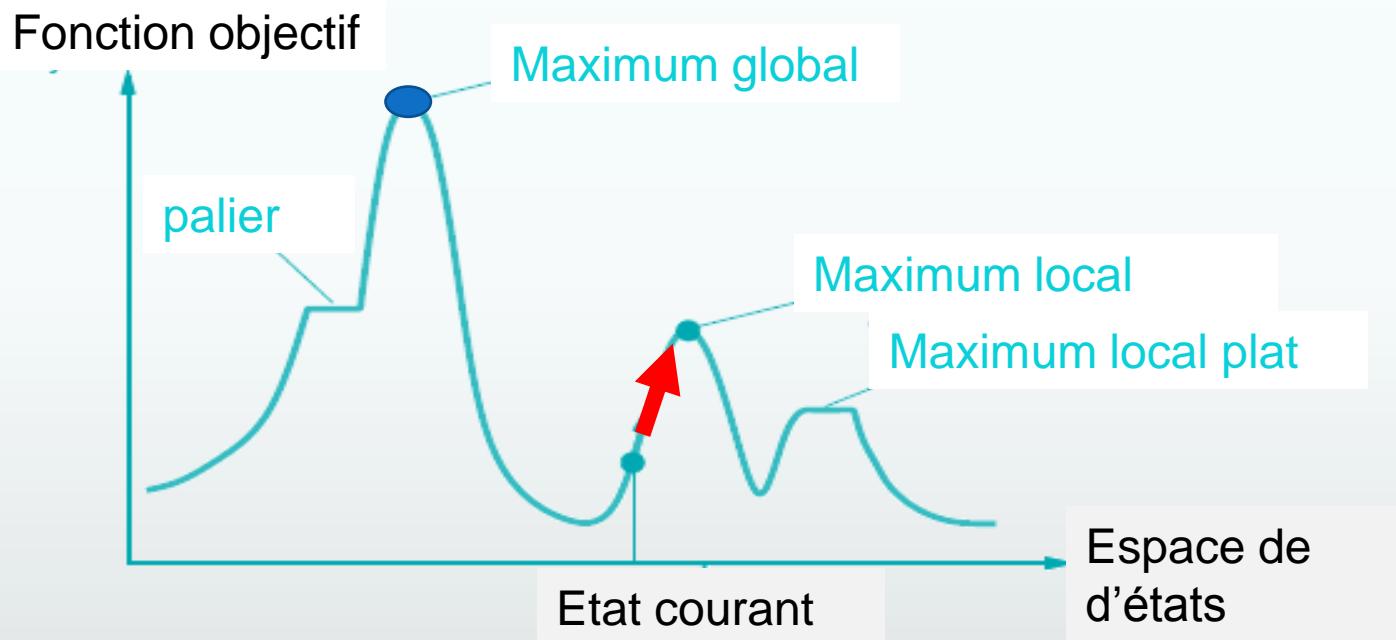
Exploration par escalade

- ❑ L'escalade est qualifiée de gloutonne parce qu'elle choisit un bon état voisin sans considérer le déplacement suivant.
- ❑ Pourtant elle a fréquemment de bonnes performances.
- ❑ Mais échoue souvent :
 - ❑ Maxima locaux.
 - ❑ Crêtes.
 - ❑ Plateaux.

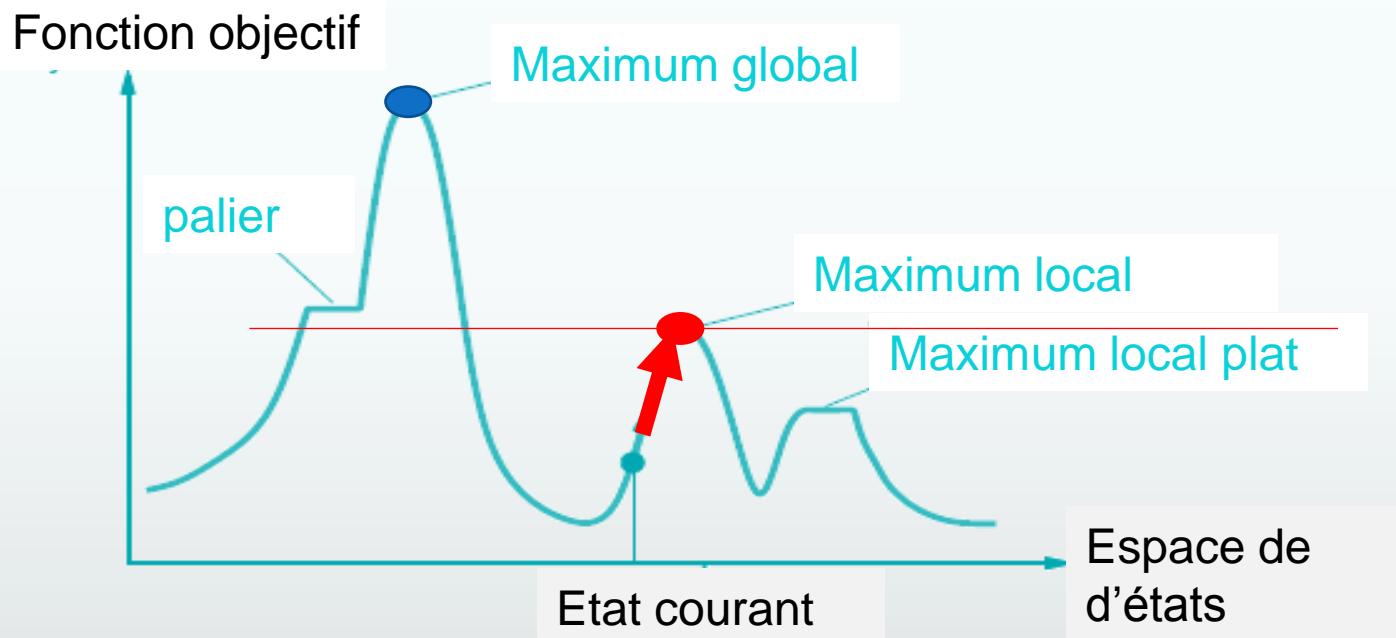
Maxima locaux



Maxima locaux



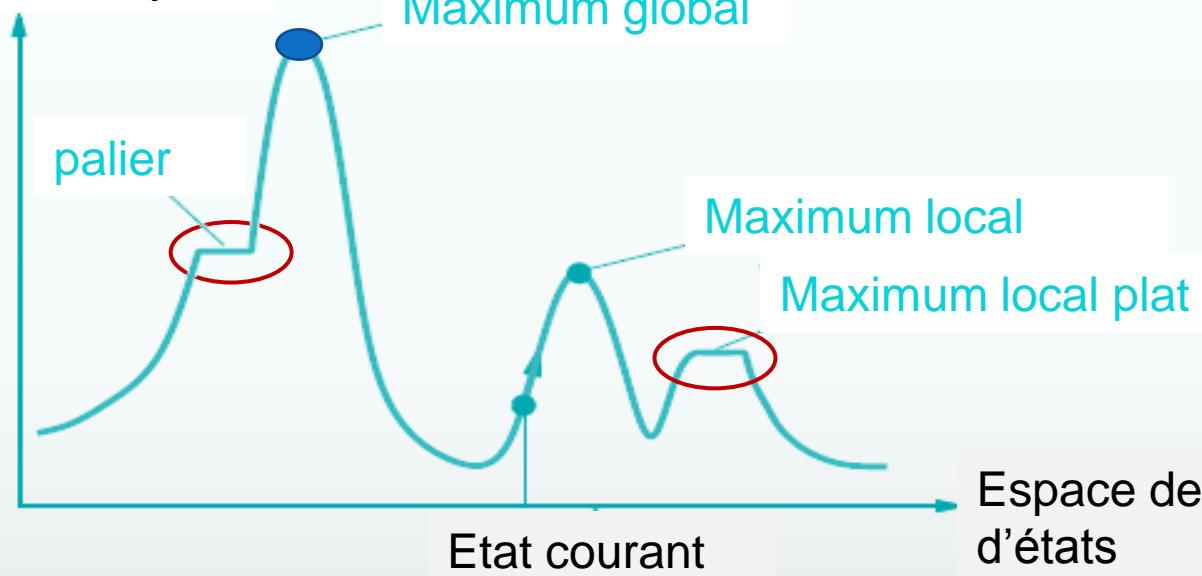
Maxima locaux



- L'algorithme *hill-climbing* risque d'être piégé dans des optimums locaux : s'il atteint un nœud dont ses voisins immédiats sont moins bons, il s'arrête !

Plateaux

Fonction objectif

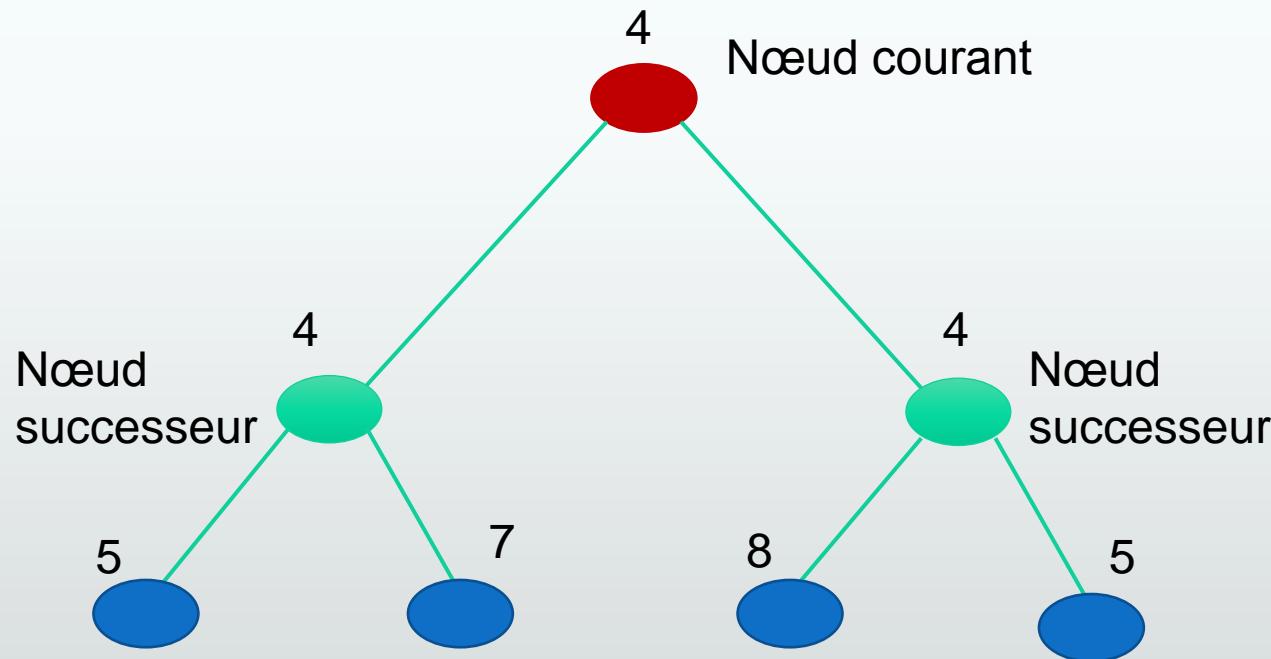


❑ Région plate :

- ❑ Maximum local plat : rien à faire.
- ❑ Palier : possibilité de progresser.

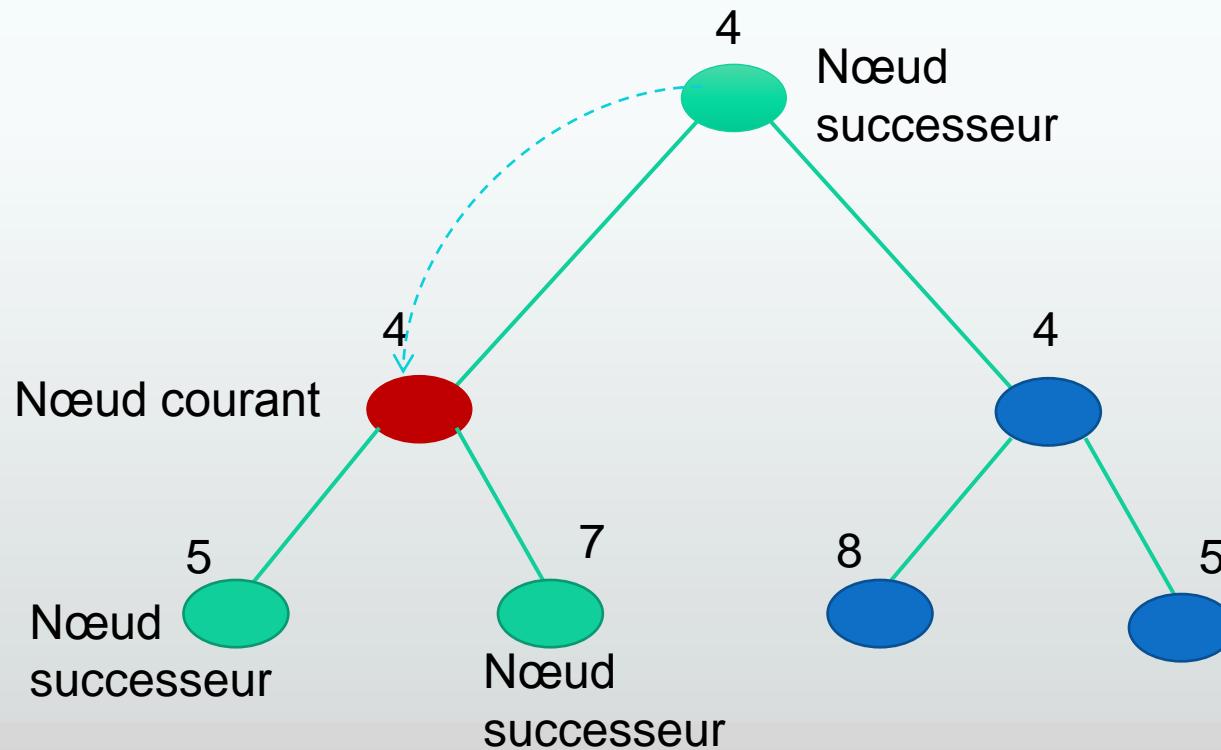
Minimum local plat

- On cherche à minimiser la fonction objectif.



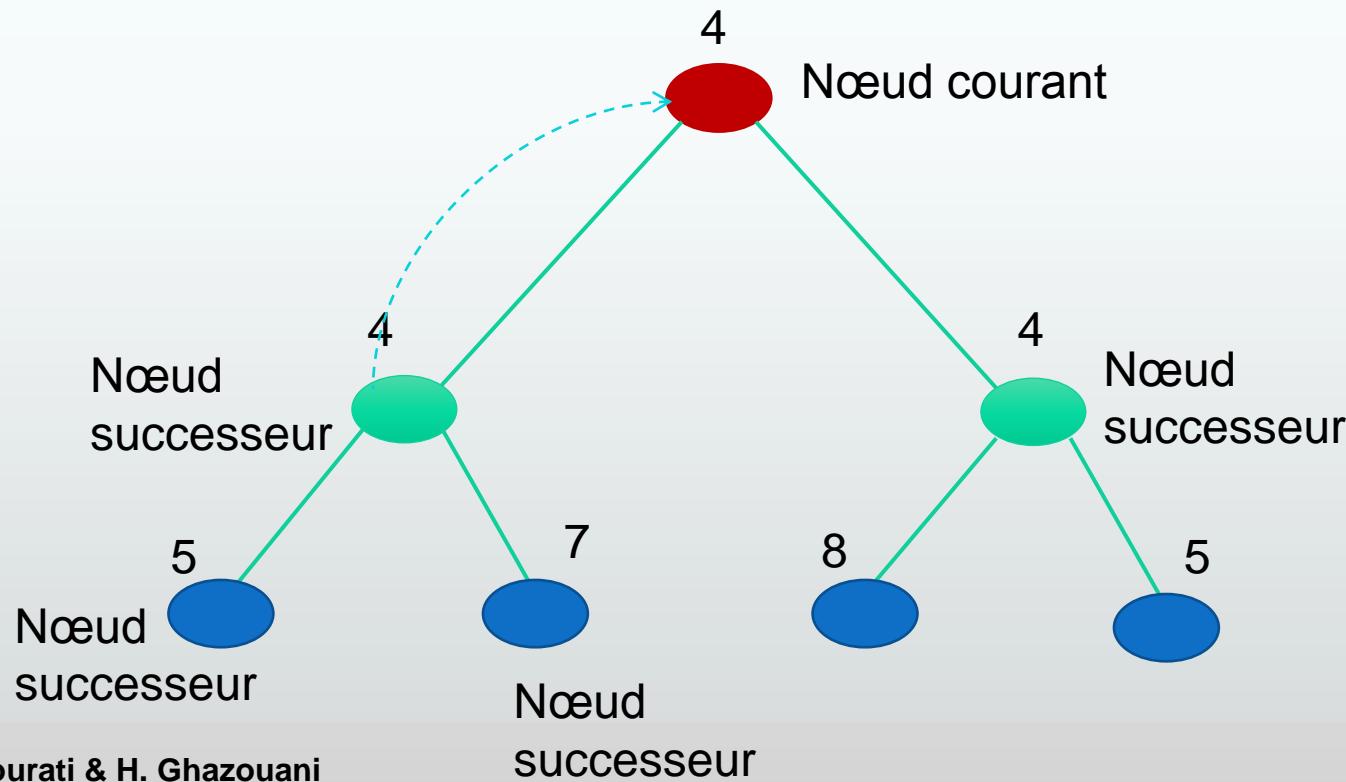
Minimum local plat

- Si on arrive à dépasser l'espace plat et on choisit un état successeur de façon aléatoire.



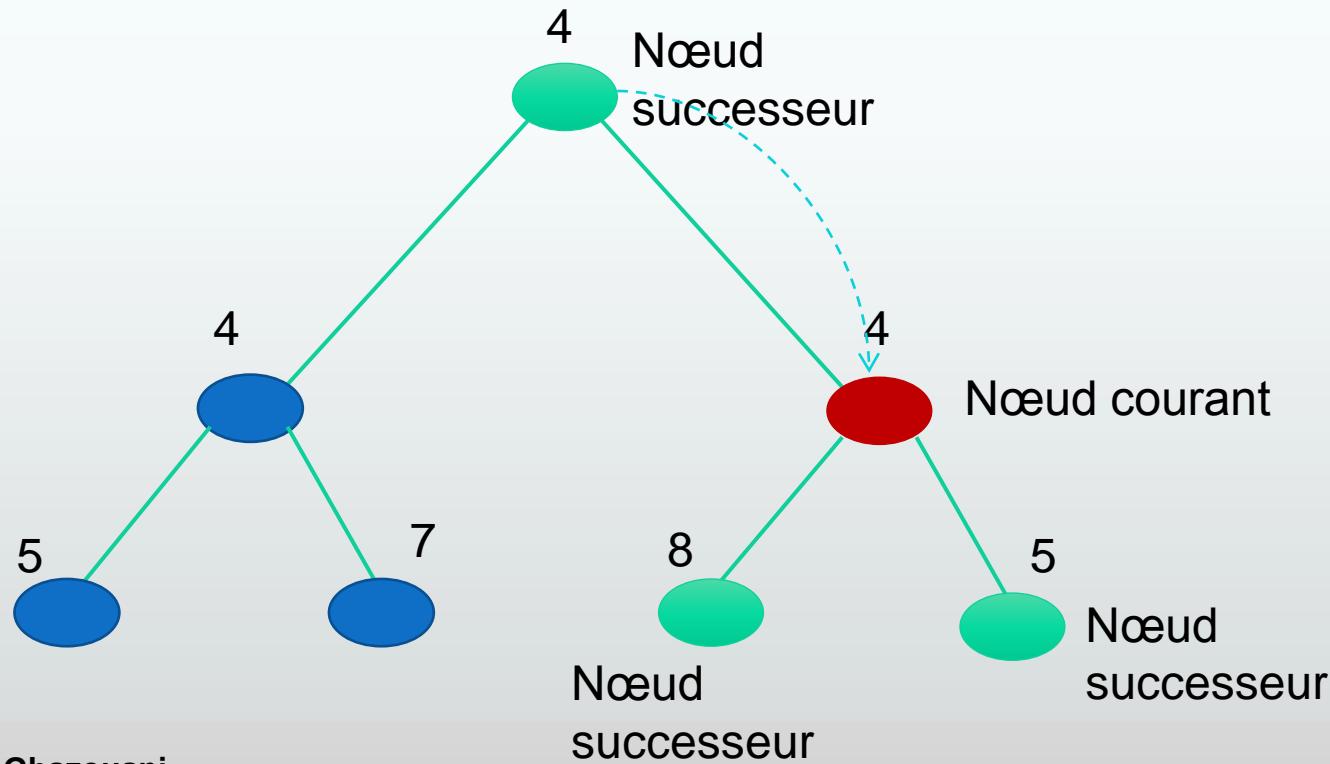
Minimum local plat

□ Ici il va revenir à l'état initial puisque c'est le meilleur choix.



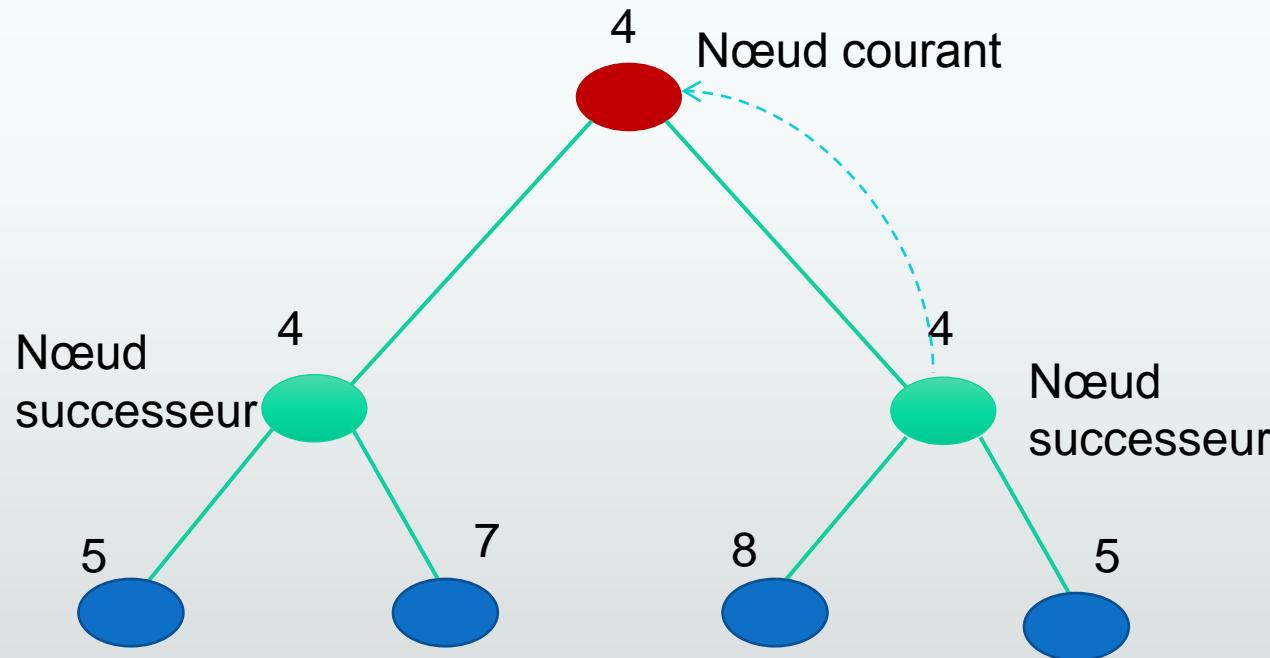
Minimum local plat

- Il peut ou bien boucler entre le nœud initial et son successeur gauche ou passer au successeur droit d'une façon aléatoire.



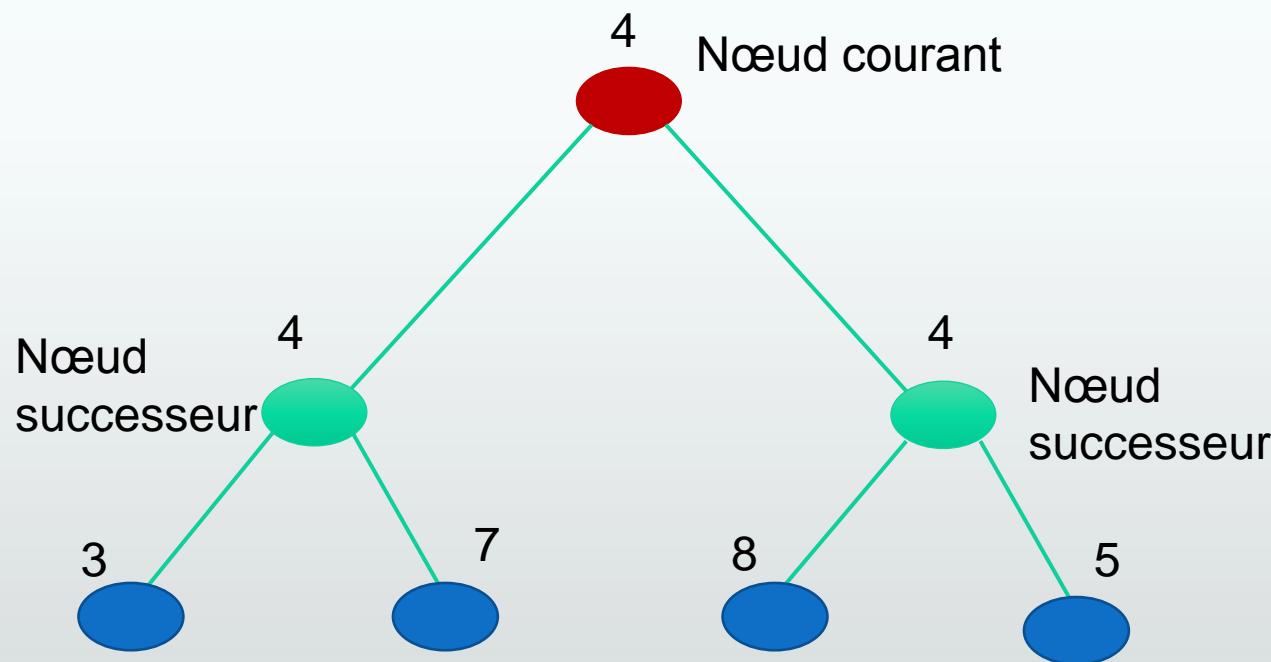
Minimum local plat

- Il va encore revenir à l'état initial puisque c'est le meilleur choix. Ainsi il bouclera à l'infini entre ses deux successeurs.



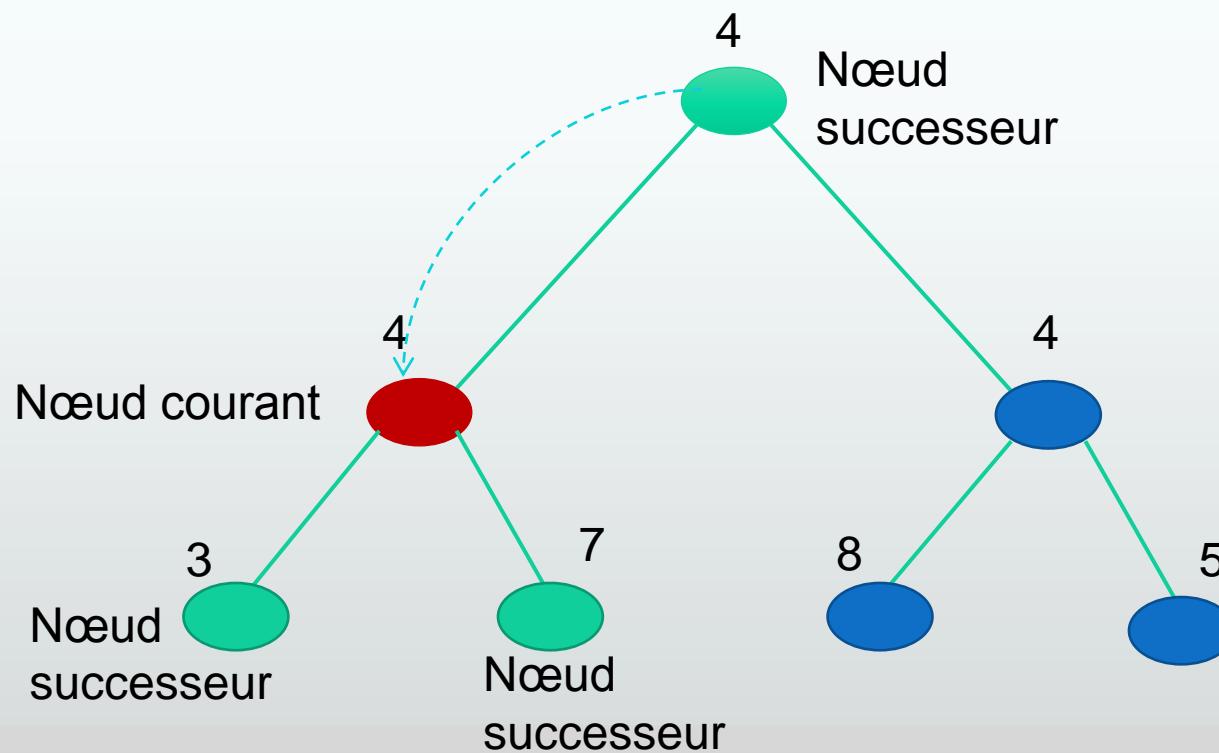
Palier

- On cherche à minimiser la fonction objectif.



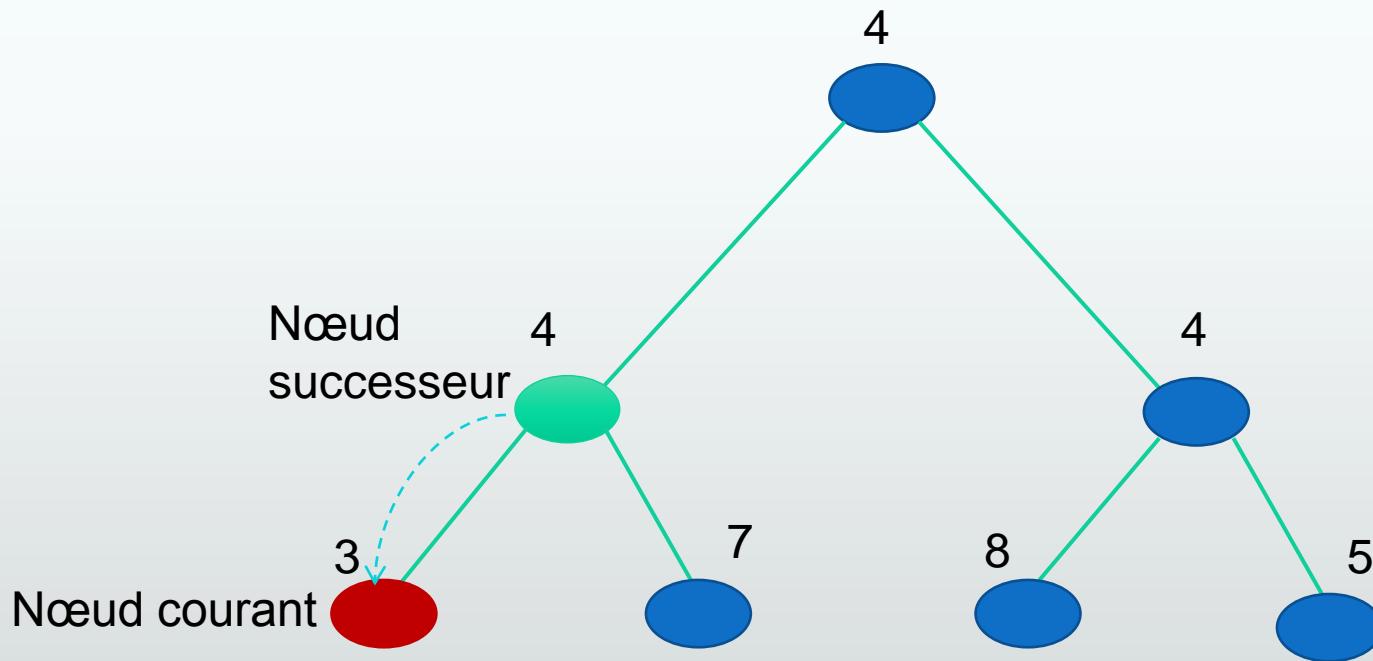
Palier

- Si on arrive à dépasser l'espace plat et on choisit un état successeur de façon aléatoire.



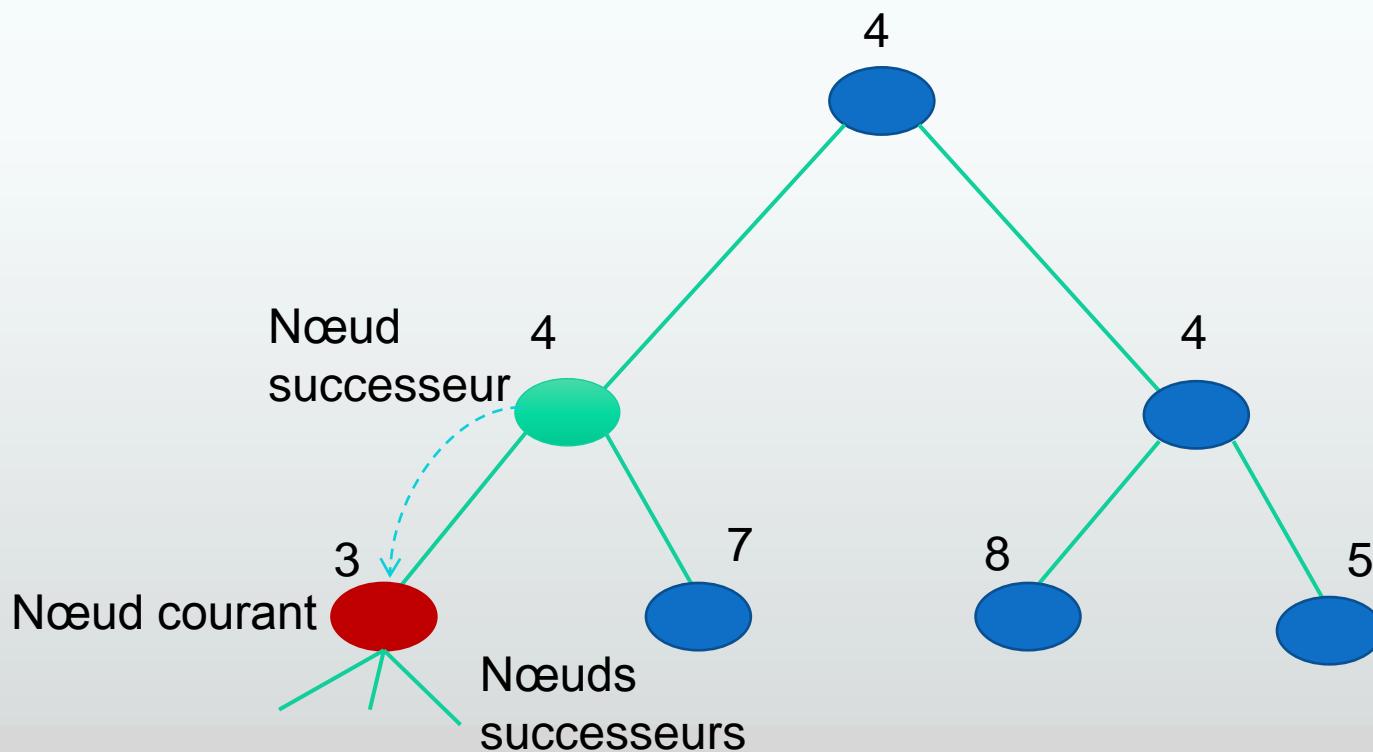
Palier

- Le successeur gauche est le meilleur choix puisqu'il minimise la fonction objectif.



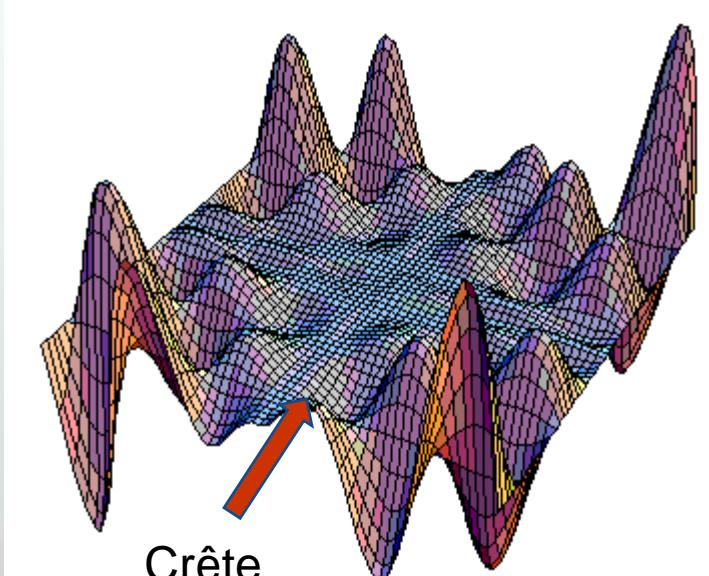
Palier

□ Il peut ainsi quitter le plateau et il pourra peut être arriver à la solution.



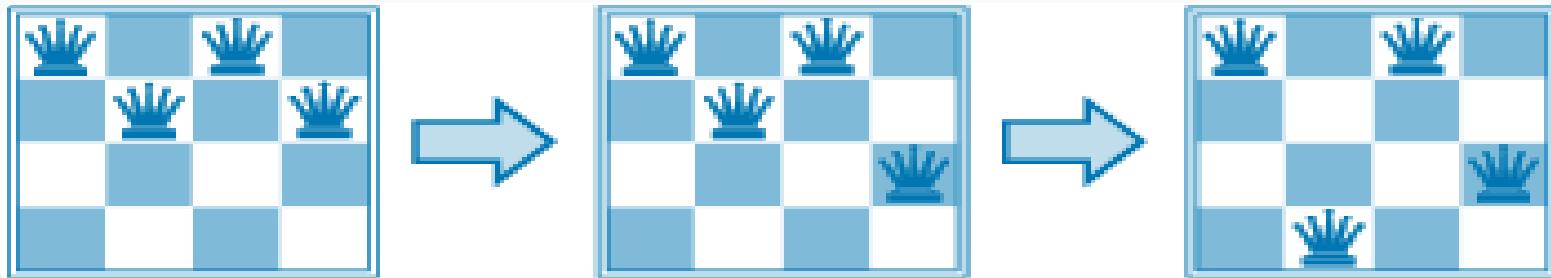
Crête (ridge)

- ❑ Ressemble à un plateau, qui a tendance à baisser vers les extrémités
- ❑ Mais on n'est pas en présence ni d'un maximum local ni global et une exploration d'autres directions peut trouver un point plus élevé.



Exemple : n reines

- Il faut placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines ne s'attaquent mutuellement : même diagonale, ligne ou colonne.



- $n=4$: 256 configurations.
- $n=8$: 16 777 216 configurations.
- $n=16$: 18,446,744,073,709,551,616 configurations.

Hill-climbing pour 8-reines

- Chaque état a 8 reines sur l'échiquier, avec 8×7 successeurs.
- Fonction heuristique h : nombre de paires de reines qui s'attaquent.
- Ici, pour l'état $n=[5,6,7,4,5,6,7,6]$ on a $h=17$.
- Le successeur est celui qui réduit le nombre d'attaques ($h=12$).

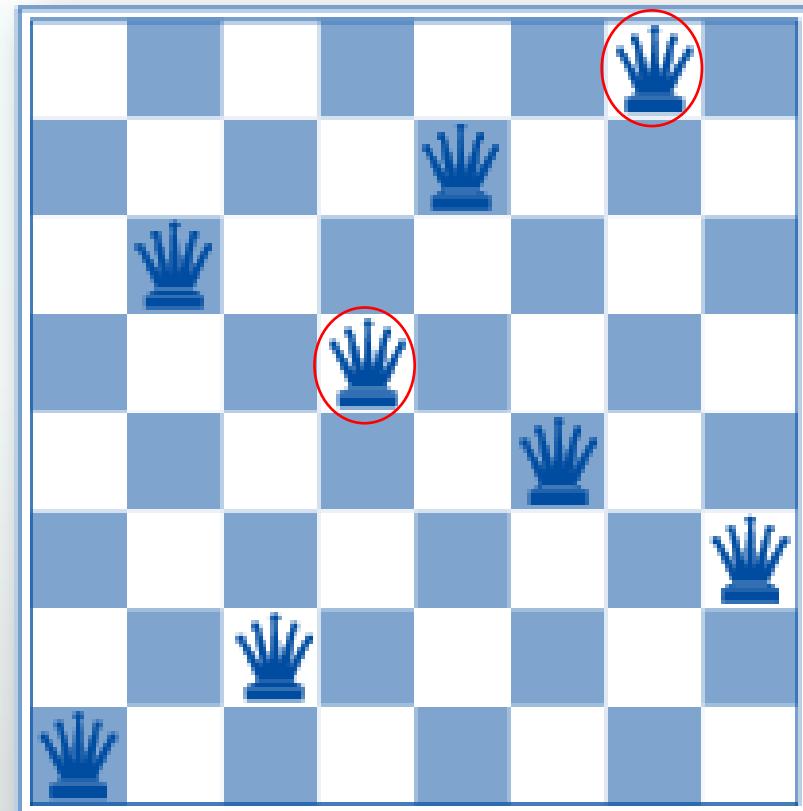


Minimum local dans l'espace des états des 8-reines

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	15	13	16	16
15	14	14	14	15	13	16	16
17	14	17	15	15	14	16	16
17	15	16	18	15	15	15	15
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

$h=17$

Après
5
étapes



$h=1$ (minimum local)

Surmonter le plateau

- L'algorithme s'arrête en présence d'un plateau.
- Une amélioration vise à se déplacer latéralement dans l'espoir que le plateau s'avère être un palier : mais attention au bouclage.
- Une solution : imposer une limite au nombre de déplacements.

Variantes de l'exploration par escalade

- **Escalade stochastique** : choisit par hasard parmi les mouvements ascendants avec une certaine probabilité qui dépend de l'importance du mouvement.
- **Escalade du premier choix** : génère des successeurs au hasard jusqu'à ce que l'un d'eux soit meilleur que l'état courant (bonne en présence de plusieurs voisins).
- **Escalade avec reprise aléatoire**.

Escalade avec reprise aléatoire

- Dans le cas où aucun avancement n'est possible, des explorations par escalade sont démarrées à partir d'états initiaux générés aléatoirement.
- Pour le problème des reines, même pour trois millions de reines, des solutions sont trouvées en moins d'une minute.
- Le succès de l'escalade dépend du paysage :
 - peu de maxima locaux et de plateaux: la solution sera trouvée rapidement.
 - Si la surface est dentelée : la complexité temporelle est exponentielle, mais il est souvent possible de trouver une solution raisonnable après un petit nombre de reprises.

Exercice

□ h : nombre de cases mal placées

2	8	3
1	6	4
7		5

État initial
 $h=4$

1	2	3
8		4
7	6	5

État but
 $h=0$

Hill-Climbing : verdict

- ❑ Il peut trouver une solution très rapidement.
- ❑ Il peut être **piégé** dans des maximum/minimum locaux.
- ❑ Peut **osciller** indéfiniment en revenant à un état antérieurement visité : cas du maximum local plat.

Exploration par Recuit simulé (*Simulated Annealing*)

Exploration par recuit simulé

- ❑ Amélioration de *hill-climbing* pour minimiser le risque d'être piégé dans des minimums locaux.
- ❑ Compromis entre un parcours qui n'effectue jamais de déplacements ascendants (efficace mais incomplet) et un parcours purement aléatoire (complet mais inefficace).
- ❑ Inspiré de la métallurgie :
 - ❑ Durcissement des métaux par chauffage à haute température.
 - ❑ Refroidissement progressif pour atteindre la cristallisation.

Principe de l'exploration par recuit simulé

- Au lieu de choisir le meilleur voisin, un nœud successeur est généré au hasard.
- S'il améliore la situation, il est toujours accepté.
- Sinon, le nœud est choisi selon une probabilité < 1 .
- La probabilité que le déplacement soit choisi décroît avec la mauvaise qualité du nœud.

Principe de l'exploration par recuit simulé

- ❑ Un paramètre T dit température, qui tend vers zéro avec le temps, influence aussi la probabilité :
 - ❑ La probabilité décroît à mesure que T baisse.
 - ❑ Plus T est grande plus un mauvais mouvement a des chances d'être pris.
- ❑ Le nombre d'itérations et la diminution des probabilités sont définis à l'aide d'un schéma (*schedule*) de températures, en ordre décroissant
- ❑ Lorsque T s'approche de 0, l'algorithme se comporte comme celui de l'escalade (aucun dégradation du coût n'est acceptée).

Algorithme d'exploration par recuit simulé

fonction RECUIT-SIMULÉ(*problème, schéma*) **retourne** un état solution
entrées: *problème*, un problème
 schéma, une fonction du temps sur les « températures »

```
courant ← CRÉER-NŒUD(problème.ÉTAT-INITIAL)
pour  $t = 1$  jusqu'à  $\infty$  faire
     $T \leftarrow schéma(t)$ 
    si  $T = 0$  alors retourner courant
    suivant ← un successeur de courant choisi au hasard
     $\Delta E \leftarrow suivant.VALEUR - courant.VALEUR$ 
    si  $\Delta E > 0$  alors courant ← suivant
    sinon courant ← suivant seulement avec une probabilité de  $e^{\Delta E/T}$ 
```

Exploration locale en faisceau (*local beam search*)

Principe de l'exploration locale en faisceau

- Conserve k nœuds en mémoire au lieu d'un.
- Commence avec k états générés aléatoirement :
 - A chaque étape, tous les successeurs des k états sont générés.
 - Si l'un d'eux est un but, l'algorithme s'arrête.
 - Sinon, il sélectionne les k meilleurs successeurs et recommence.
- Les k explorations se partagent les informations utiles.
- **Exploration en faisceau stochastique** : sélectionne les k successeurs au hasard avec une probabilité qui est une fonction croissante de sa valeur (répond à l'absence de diversité).

Algorithmes génétiques

Principe de l'algorithme génétique

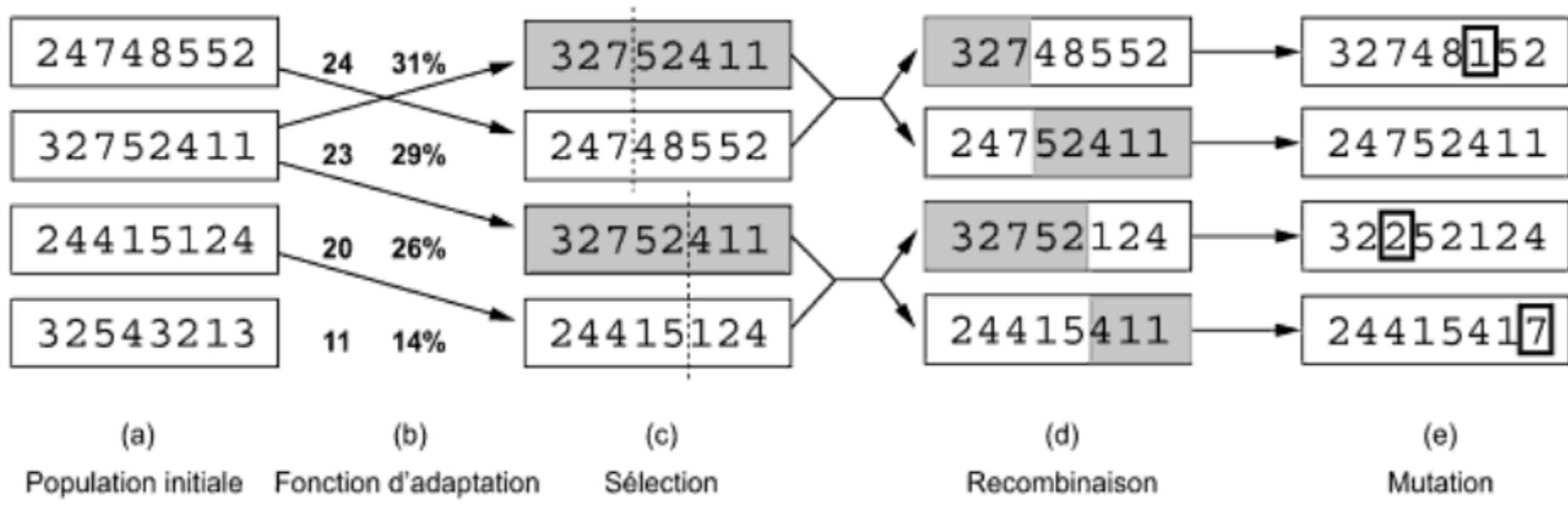
- ❑ Variante de l'exploration en faisceau stochastique, mais les états successeurs sont générés en combinant deux parents au lieu de modifier un seul état.
- ❑ Il utilise des principes de l'évolution naturelle :
 - ❑ La loi du plus fort.
 - ❑ Deux individus génétiquement favorisés donnent généralement des enfants génétiquement favorisés.
 - ❑ Existence de mutations.

Principe de fonctionnement des algorithmes génétiques

- Commence par k états générés aléatoirement : la **population**.
- Chaque état, individu, est représenté par des chaînes de caractères : chaînes de bits ou des expressions symboliques.
- La génération suivante est obtenue par **reproduction** ou **mutation** de ces individus.
- Les meilleurs individus ont de meilleures chances d'être choisis selon une fonction objectif : **fonction d'adaptation** (*fitness function*).
- Avec une faible probabilité, on applique une mutation à l'individu qui sera ajouté à la nouvelle population.
- Les reproductions et mutations continuent jusqu'à ce que l'individu solution apparait dans la population.

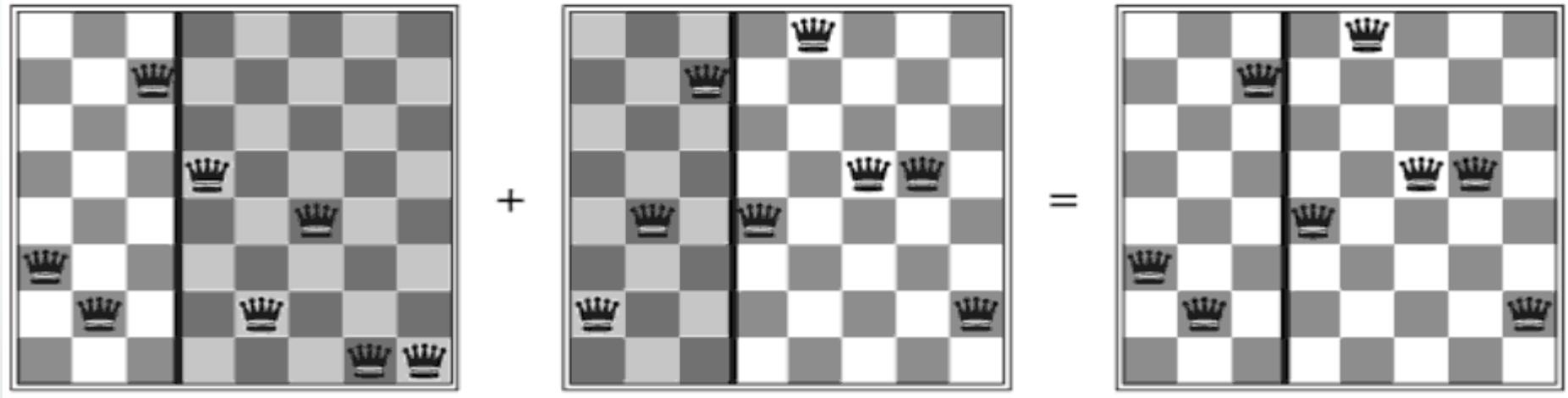
Illustration de l'algorithme

Population de quatre chaînes de huit chiffres



Nombre de reines
qui ne sont pas en
prise

Croisement



Fonctionnement de l'algorithme génétique

1. Générer aléatoirement une population de N chromosomes.
2. Calculer la valeur d'adaptabilité (*fitness*) de chaque chromosome x .
3. Créer une nouvelle population de taille N .
 - 3.1 Sélectionnant 2 parents chromosomes (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité.
 - 3.2. Mutant les deux enfants obtenus avec une certaine probabilité.
 - 3.3 Plaçant les enfants dans la nouvelle population.
 - 3.4 Répéter à partir de l'étape 3.1 jusqu'à avoir une population de taille N .
4. Si la population satisfait le critère d'arrêt, arrêter.
Sinon, recommencer à l'étape 2.

Algorithme génétique

fonction ALGORITHME-GÉNÉTIQUE(*population*, FN-ADAPTATION) **retourne** un individu
entrées: *population*, un ensemble d'individus
FN-ADAPTATION, une fonction qui mesure l'adaptation d'un individu

répéter

nouvelle_population \leftarrow ensemble vide
 pour *i* = 1 **jusqu'à** TAILLE(*population*) **faire**
 x \leftarrow SÉLECTION-ALÉATOIRE(*population*, FN-ADAPTATION)
 y \leftarrow SÉLECTION-ALÉATOIRE(*population*, FN-ADAPTATION)
 enfant \leftarrow REPRODUIRE(*x*, *y*)
 si (petite probabilité aléatoire) **alors** *enfant* \leftarrow MUTATION(*fils*)
 ajouter *enfant* à *nouvelle_population*
 population \leftarrow *nouvelle_population*
 jusqu'à ce qu'un individu soit suffisamment adéquat,
 ou que suffisamment de temps se soit écoulé
 retourner le meilleur individu de la *population*, selon FN-ADAPTATION

fonction REPRODUIRE(*x*, *y*) **retourne** un individu
entrées: *x*, *y*, individus parents

n \leftarrow LONGUEUR(*x*); *c* \leftarrow nombre aléatoire compris entre 1 et *n*
 retourner AJOUTER(SOUS-CHAÎNE(*x*, 1, *c*), SOUS-CHAÎNE(*y*, *c* + 1, *n*))

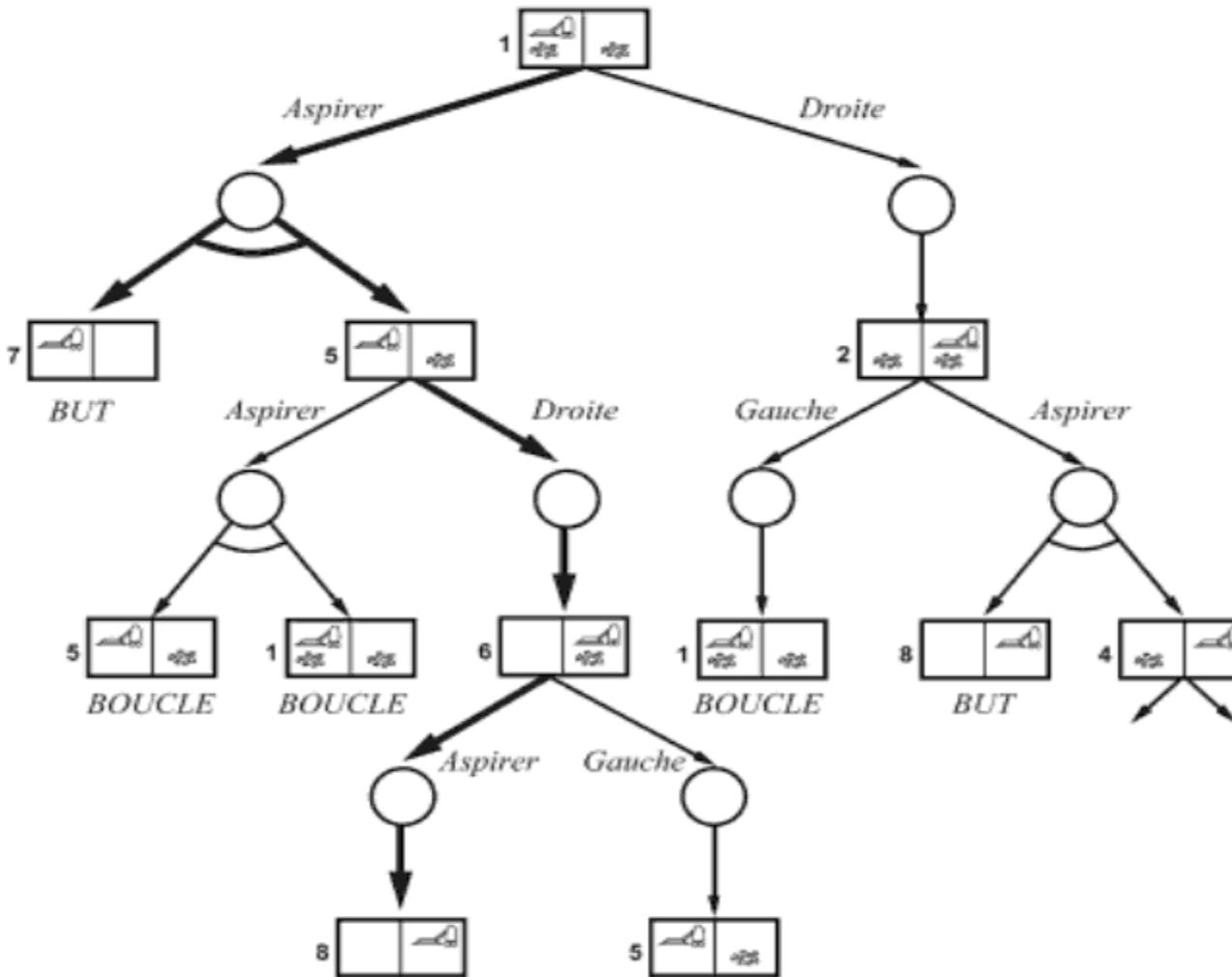
Exploration avec actions non déterministes

- Lorsque l'environnement est non déterministe, l'agent ne connaît pas à l'avance le résultat de ses actions.
- Après chaque action, les percepts indiquent ses effets réels.
- La solution n'est plus une séquence mais un plan contingent (stratégie) qui indique quoi faire selon les percepts reçus.

Aspirateur capricieux

- Exemple d'un aspirateur capricieux qui suite à l'action Aspirer :
 - Dans un carré sale, l'action le nettoie mais parfois elle nettoie aussi le carré adjacent.
 - Dans un carré propre, l'action peut déposer de la saleté.
- Une solution contingente à ce problème peut être représentée par un arbre ET-OU.

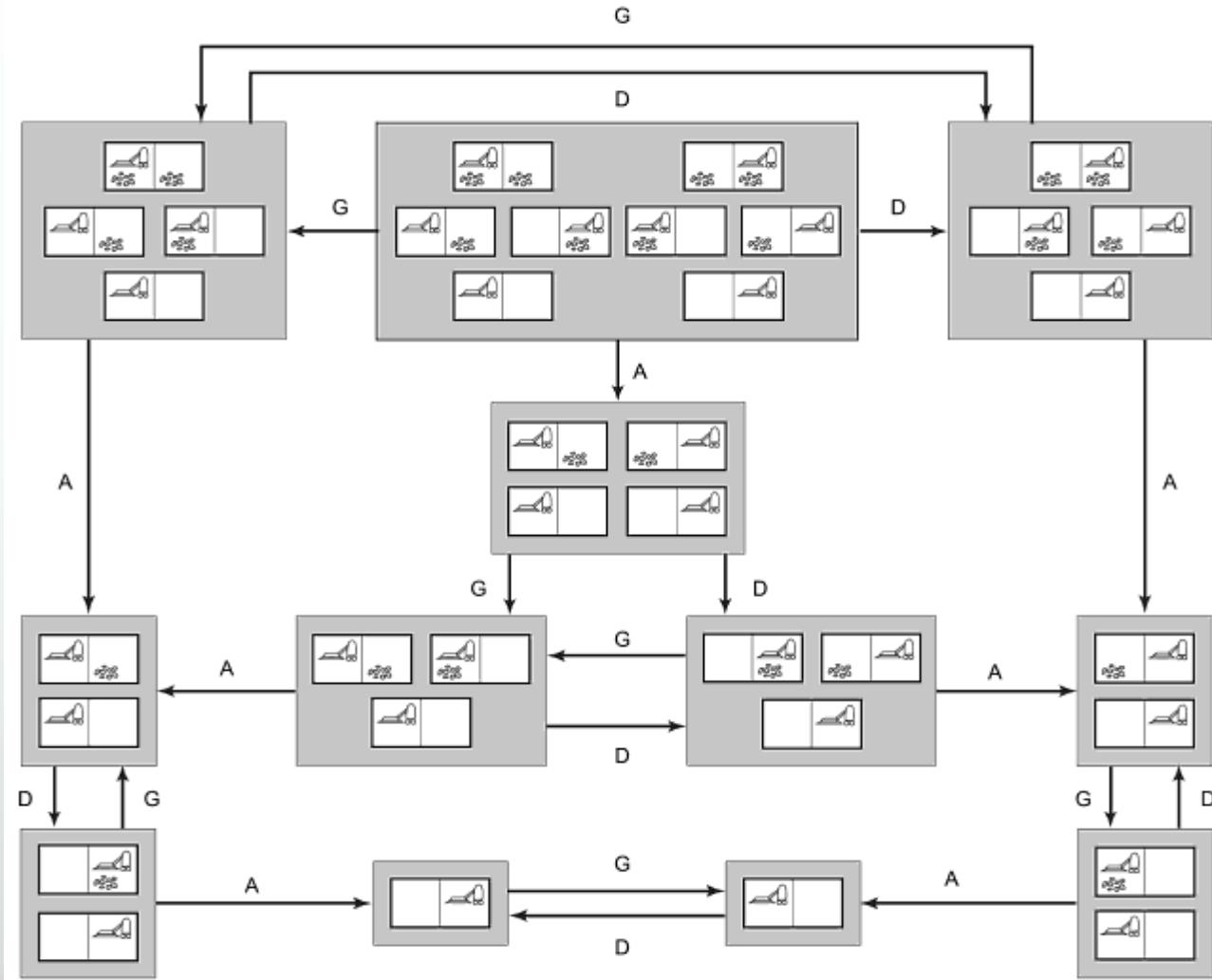
Arbre d'exploration ET-OU pour l'aspirateur capricieux



Exploration sans observation

- ❑ Les percepts de l'agent ne fournissent aucune information : sans capteurs.
- ❑ Exemple de l'aspirateur qui connaît son monde mais ignore son emplacement actuel qui peut être n'importe quel état parmi [1,2,3,4,5,6,7,8].
- ❑ Ce problème est résolu par exploration des états de croyance au lieu des états physiques.
- ❑ Rarement faisable en pratique : taille de l'état de croyance.

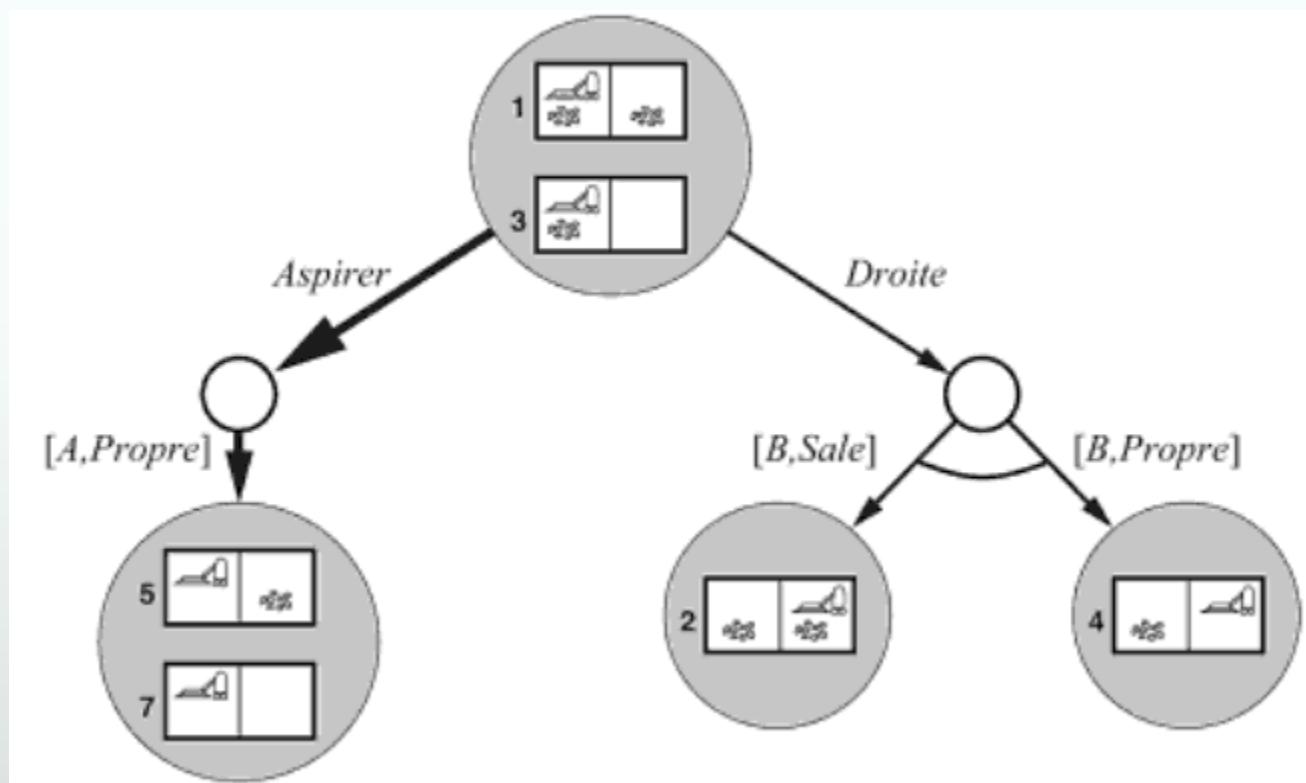
Espace des états de croyance pour l'aspirateur déterministe et sans capteurs



Résolution de problèmes partiellement observables

- ❑ Exemple d'aspirateur à détection locale : détecteurs de position et de saleté locale.
- ❑ Une fonction percepts renvoie le percept reçu dans un état.
- ❑ Exemple : le percept dans les états 1 et 3 sont *[A,sale]*.
- ❑ Sachez que dans un environnement :
 - ❑ Entièrement observable : $\text{Percept}(s)=s$.
 - ❑ Sans capteur (non observable) : $\text{Percept}(s) = \text{nil}$.

Un niveau de l'arbre d'exploration ET-OU à détection locale



Recherche *hors ligne*

- ❑ Les algorithmes vus sont des algorithmes d'exploration hors ligne.
- ❑ Ils calculent la solution complète avant d'affronter l'environnement.
- ❑ Ils exécutent les actions sans utiliser leurs percepts.

Recherche en ligne

- Nécessaire pour les environnement inconnus : problèmes de découverte.
- Les algorithmes d'exploration en ligne entrelacent calcul et action : l'agent doit exécuter des actions et non réaliser uniquement des calculs.
- Il ne connaît pas le résultat de ses actions avant de les effectuer réellement.
- Après chaque action l'agent reçoit un percept lui indiquant l'état atteint et enrichir sa carte de l'environnement.
- Contrairement à un algorithme hors ligne comme A* il ne peut pas développer un nœud dans une partie de l'espace ensuite développer un autre nœud dans une autre partie : les actions sont simulées et non réelles.

Fonctionnement de la recherche *en ligne*

- ❑ Le développement des nœuds se fait dans un ordre local : l'exploration en profondeur possède cette propriété.
- ❑ L'état résultant de l'exécution d'une action est enregistré.
- ❑ Chaque fois qu'une action de l'état courant n'a pas été explorée, l'agent l'essaie.
- ❑ Avec une exploration en profondeur hors ligne l'état visité est supprimé de la file, mais dans l'exploration en ligne l'agent revient en arrière physiquement.

Exploration en ligne locale

- L'exploration par escalade possède la propriété de localité et est donc un exploration en ligne.
- Mais elle n'est pas intéressante puisque l'agent reste bloqué dans des maximum/minimum locaux.
- Les reprises aléatoires ne sont pas applicables : impossible de se transporter dans un nouvel état.
- Une solution est le parcours aléatoire : sélection au hasard d'une action disponible.
- Une meilleure solution consiste à mémoriser la meilleure estimation courante du coût pour atteindre le but à partir de chaque état visité : l'agent est appelé LRTA*.