Résolution de problèmes par l'exploration

IFT-17587

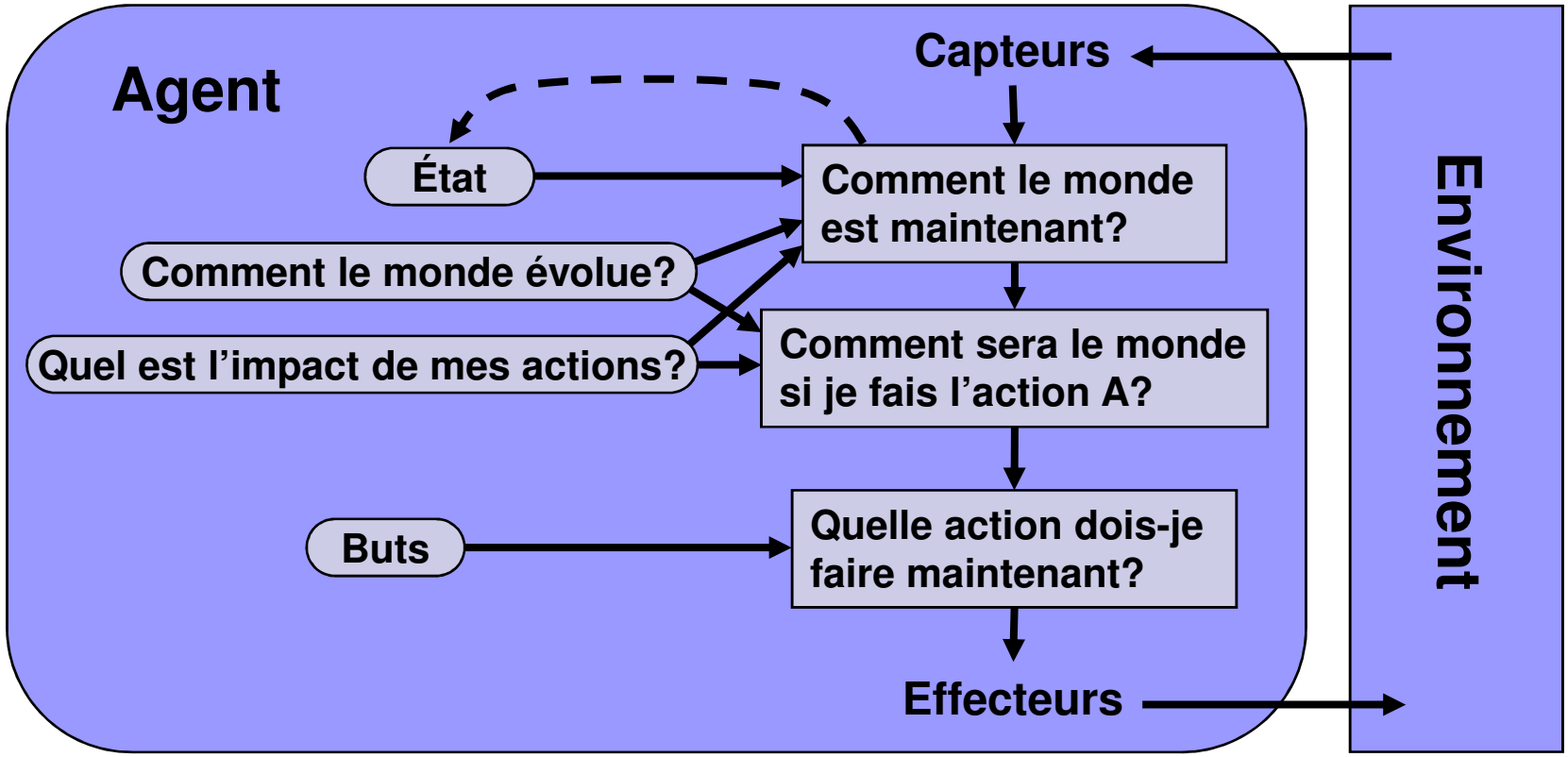
Concepts avancés pour systèmes intelligents

Luc Lamontagne

Plan

- Agent de résolution de problèmes
- Formulation de problèmes
- Stratégies d'exploration
  - Exploration non informée
    - Largeur d'abord, profondeur d'abord, etc.
  - Exploration informée
    - Meilleur d'abord, A*, recuit simulé, algorithmes génétiques, etc.

Rappel - Agent basé sur les buts



Goal-based agent

Agent basé sur les buts :

Agent de résolution de problèmes

1. Formulation d'un but:
   - Un ensemble d'états à atteindre.
2. Formulation du problème:
   - Les états et les actions à considérer.
3. Exploration de solution:
   - Examiner les différentes séquences d'actions menant à un état but;
   - Et choisir la meilleure.
4. Exécution:
   - Accomplir la séquence d'actions sélectionnées.

Agent basé sur les buts :

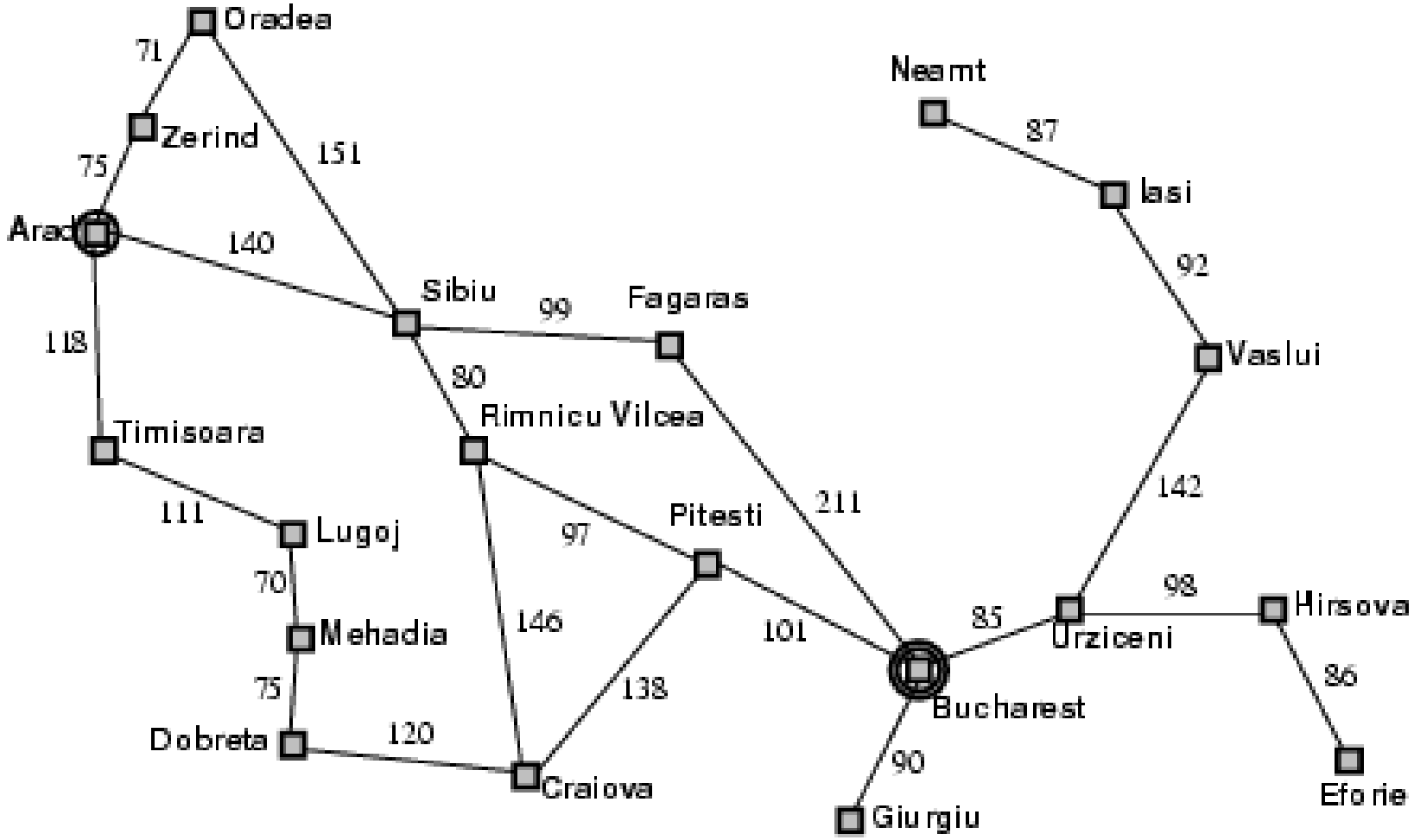
Agent de résolution de problèmes

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Exemple de formulation de problèmes :

Planification de route



Exemple de formulation de problèmes :

Planification de route

- On est à Arad et on veut aller à Bucharest
  - But :
    - Être à Bucharest
  - Problème:
    - États : villes
    - Actions : aller d'une ville à une autre.
  - Solution :
    - Une séquence de villes.
    - Par ex. Arad, Sibiu, Fagaras, Bucharest
  - Coût :
    - Distance entre les deux villes (en km)
- Environnement simple
  - statique, observable, discret et déterministe

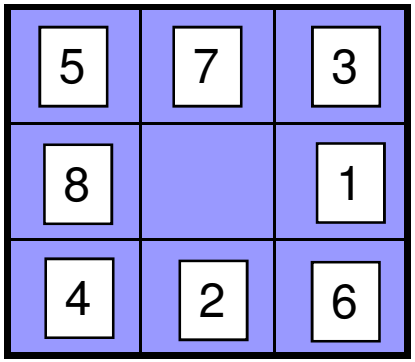
Formulation d'un problème

- État initial
  - L'état *x* dans lequel se trouve l'agent
    - *Par ex.* *x* = À(Arad)
- Actions
  - Une fonction successeur *S*(*x*) qui permet de trouver l'ensemble des états adjacents (*action*, *état_successeur*)
  - *Par ex. S*(*Arad*) = { (*AllerÀ*(*Sibiu*), *À*(*Sibiu*)) , (*AllerÀ*(*Zerind*), *À*(*Zerind*)) ...}
- Test de but
  - Un test afin de déterminer si le but est atteint
    - Par ex. x = À(*Bucharest)*
- Coût du chemin
  - Une fonction qui assigne un coût à un chemin.
  - Elle reflète la mesure de performance de l'agent.
  - La valeur est non-négative.

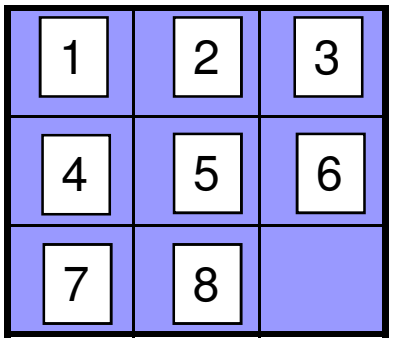
Exemple de formulation de problèmes :

8-puzzle

- États :
  - Positions des huit tuiles dans les cases.
- État initial :
  - Les huit tuiles dans n'importe quelle case.
- Actions :
  - Déplacement du trou
    - droite, gauche, haut, bas.
- Test de but :
  - Un état qui correspond à l'état final.
- Coût :
  - Chaque action coûte 1.



État initial

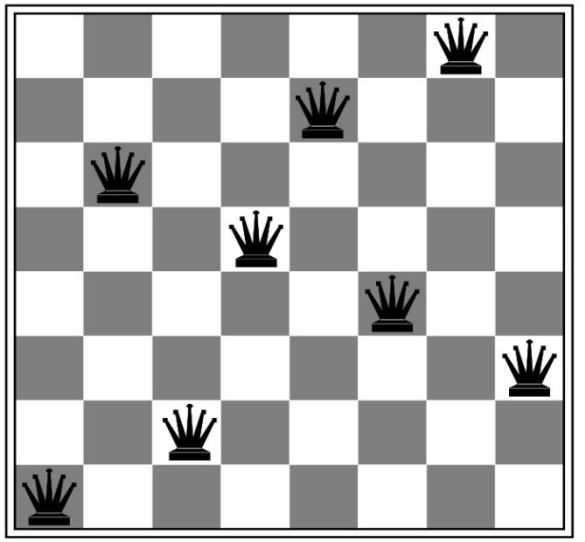


État but

Exemple de formulation de problèmes :

8-reines

- États
  - Une configuration de 0 à 8 reines sur l'échiquier.
- État initial
  - Aucune reine sur l'échiquier.
- Actions
  - Ajouter une reine sur une case vide.
- Test de but
  - Les 8 reines sont placés sur l'échiquier sans attaque.
- Coût
  - Chaque action coûte 1.

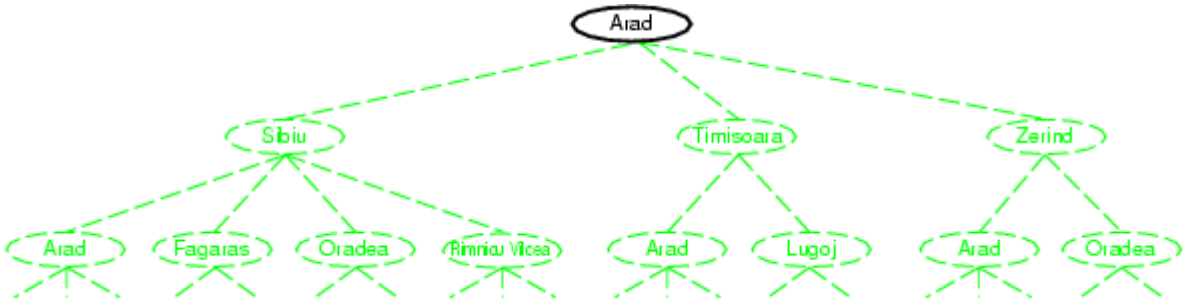


Exploration de solutions dans un arbre

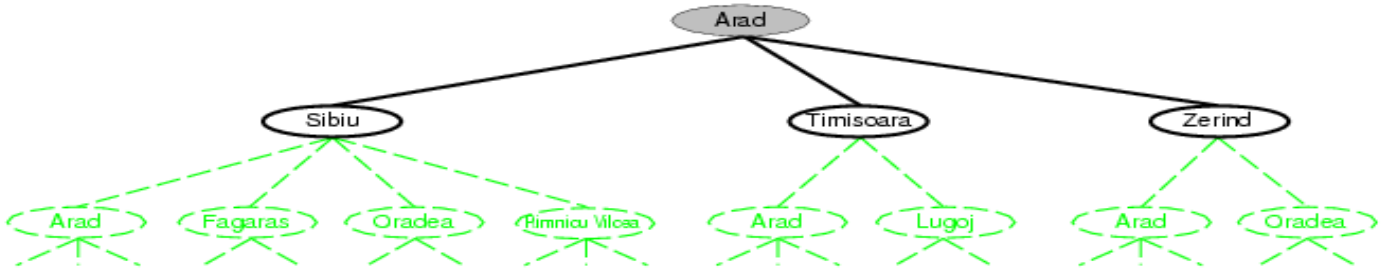
- Simuler l'exploration de l'espace d'états en générant des successeurs pour les états déjà explorés.
  - Exploration simulée et *offline*

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

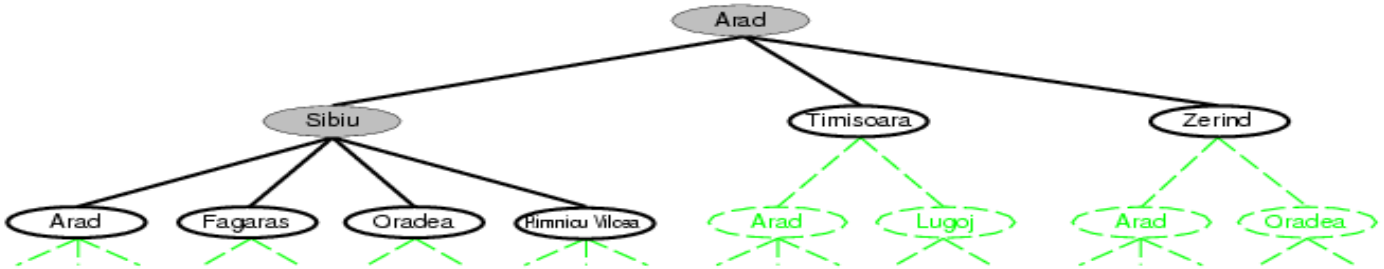
Exemple d'exploration dans un arbre



Exemple d'exploration dans un arbre



Exemple d'exploration dans un arbre



Exploration dans un arbre :

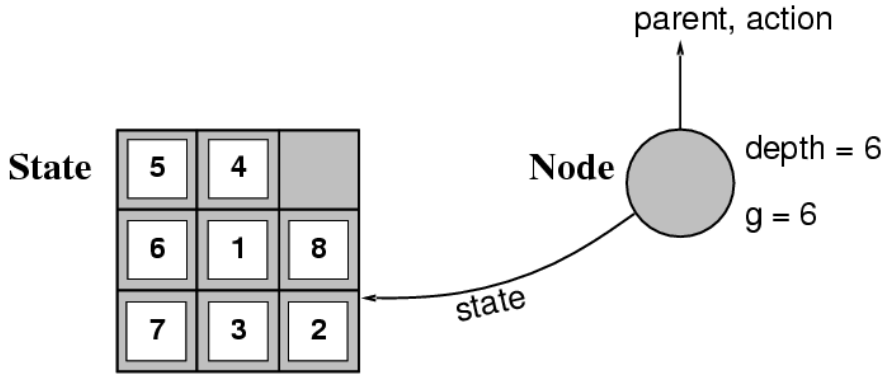
Mise en oeuvre

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Exploration de solutions dans un arbre

- Simuler l'exploration de l'espace d'états en générant des successeurs pour les états déjà explorés.
- Nœud de recherche
  - État: l'état dans l'espace d'état.
  - Nœud parent: Le nœud dans l'arbre de recherche qui a généré ce nœud.
  - Action: L'action qui a été appliquée au parent pour générer ce nœud.
  - Coût du chemin: Le coût g(n) du chemin à partir de l'état initial jusqu'à ce nœud.
  - Profondeur: Le nombre d'étapes dans le chemin à partir de l'état initial.



Stratégies d'exploration

- Détermine l'ordre de développement des nœuds.
- Explorations non informées
  - Aucune information additionnelle.
  - Elles ne peuvent pas dire si un nœud est meilleur qu'un autre.
  - Elles peuvent seulement dire si l'état est un but ou non.
- Explorations informées (heuristiques):
  - Elles peuvent estimer si un nœud est plus prometteur qu'un autre.

Évaluation des stratégies

- Complétude:
  - Est-ce que l'algorithme garantit de trouver une solution s'il y en a une?
- Optimalité:
  - Est-ce que la stratégie trouve la solution optimale?
- Complexité en temps:
  - Combien de temps pour trouver une solution?
- Complexité en espace:
  - Quelle quantité de mémoire a-t-on besoin?

Complexité

- Elle est exprimée en utilisant les quantités suivantes
  - *B :* le facteur de branchement
    - c.-à-d. le nombre maximum de successeurs à un nœud.
  - *D :* la profondeur du nœud but le moins éloigné.
  - *M :* la longueur maximale d'un chemin dans l'espace d'états.
- Complexité en temps
  - le nombre de nœuds générés pendant la recherche.
- Complexité en espace
  - le nombre maximum de nœud en mémoire.

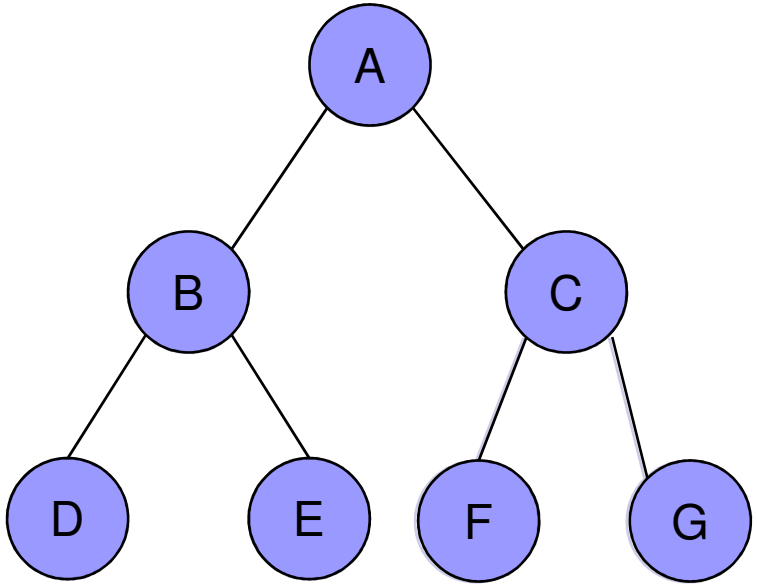
Stratégies d'exploration non informées

- Largeur d'abord (*Breath-first - BFS)*
- Coût uniforme (*Uniform-cost - UFS*)
- Profondeur d'abord (*Depth-first - DFS*)
- Profondeur limitée (*Depth-limited - DLS*)
- Itérative en profondeur (*Iterative deepening - IDS*)
- Bidirectionnelle (*Bidirectional search*)

Largeur d'abord (BFS)

- Approche
  - Développer tous les noeuds au niveau *i*
  - Développer par la suite tous les nœuds au niveau *i+1*
  - Et ainsi de suite…
- Implémenté à l'aide d'une file.
  - Les nouveaux successeurs vont à la fin.

Exemple largeur d'abord



File:

Ordre de visite: A – B – C – D – E – F - G

Propriétés de largeur d'abord

- Complétude : oui, si b est fini
- Complexité en temps : $O(b^{d+1})$
  - $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$
- Complexité en espace : $O(b^{d+1})$
  - Garde tous les nœuds en mémoire
- Optimal : non en général.
  - Oui si le coût des actions est le même pour toutes les actions

Profondeur	Nœuds (b=10)	Temps (10 000 nœuds/sec)	Mémoire (1000 octets/ nœuds)
8	10^9	31 heures	1 teraoctet
12	10^{13}	35 ans	10 pétaoctets

Coût uniforme (UCS)

- Développe le nœud ayant le coût le plus faible.
  - *g(n)* := coût du nœud initial au nœud développé.
- File triée selon le coût.
- Si le coût des actions est toujours le même
  - Équivalent à largeur d'abord !

Coût uniforme

- Complète : oui, si le coût > ε
- Complexité en temps :

  nombre de nœuds avec $g(n) \leq$ coût(solution optimale)

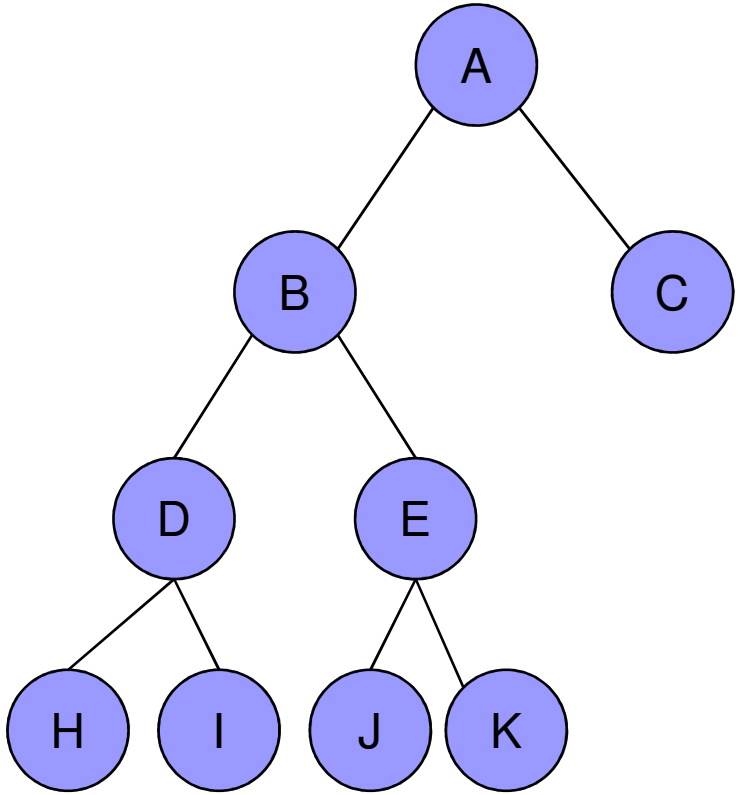
  $$O(b^{\lceil C^*/\epsilon \rceil})$$

  où C^* est le coût de la solution optimale.
- Complexité en espace : même que celle en temps
- Optimal : oui
  - Les nœuds sont développés en ordre de $g(n)$.

Profondeur d'abord (DFS)

- Développe le nœud le plus profond.
- Implémenté à l'aide d'un pile.
  - Les nouveaux nœuds générés vont sur le dessus.

Exemple profondeur d'abord



Pile:

Ordre de visite: A – B – D – H – I – E – J – K - C

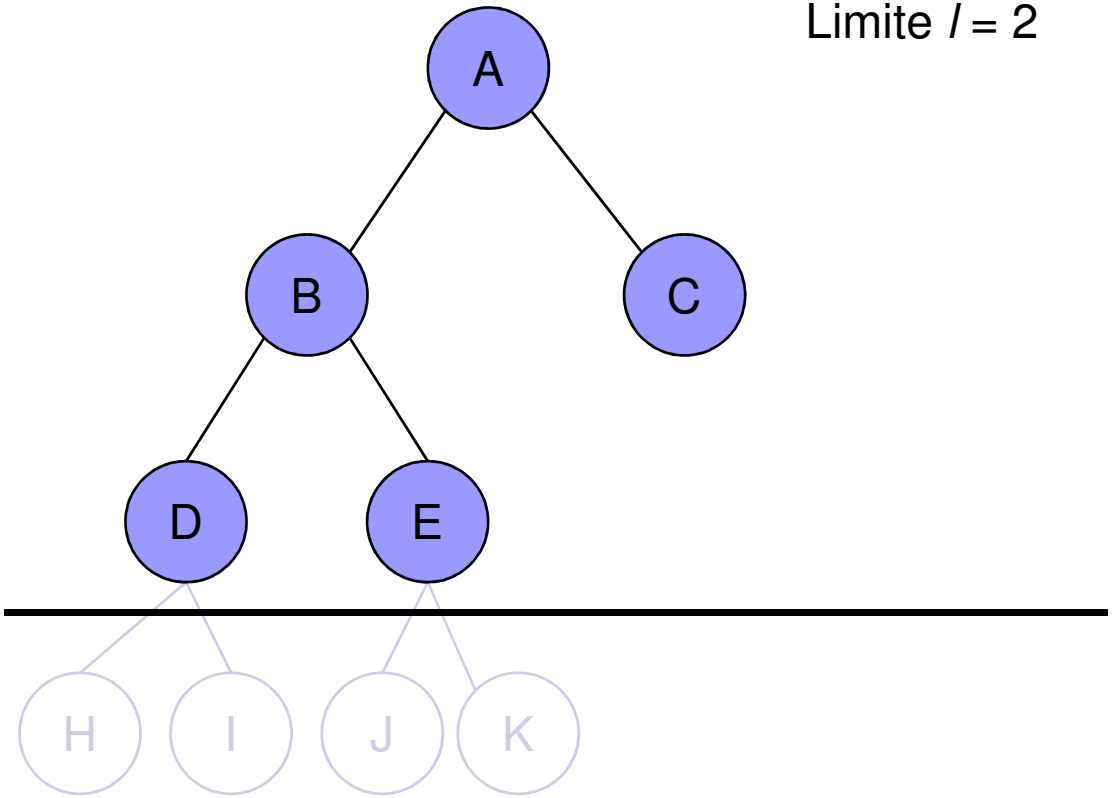
Propriétés de profondeur d'abord

- Complétude :
  - Non si la profondeur est infinie, s'il y a des cycles.
  - Oui, si on évite les états répétés ou si l'espace de recherche est fini.
- Complexité en temps : $O(b^m)$
  - Très mauvais si m est plus grand que d.
  - Mais si les solutions sont denses, il peut être beaucoup plus rapide que largeur d'abord.
- Complexité en espace : $O(bm)$, linéaire
- Optimal : Non

Profondeur limitée (DLS)

- L'algorithme de profondeur d'abord, mais avec une limite de *l* sur la profondeur.
  - Les nœuds de profondeur *l* n'ont pas de successeurs.
- Complétude : Seulement si $l > d$
- Complexité en temps : $O(b^l)$
- Complexité en espace : $O(bl)$, linéaire
- Optimal : Non!

Exemple profondeur limité



Limite *l* = 2

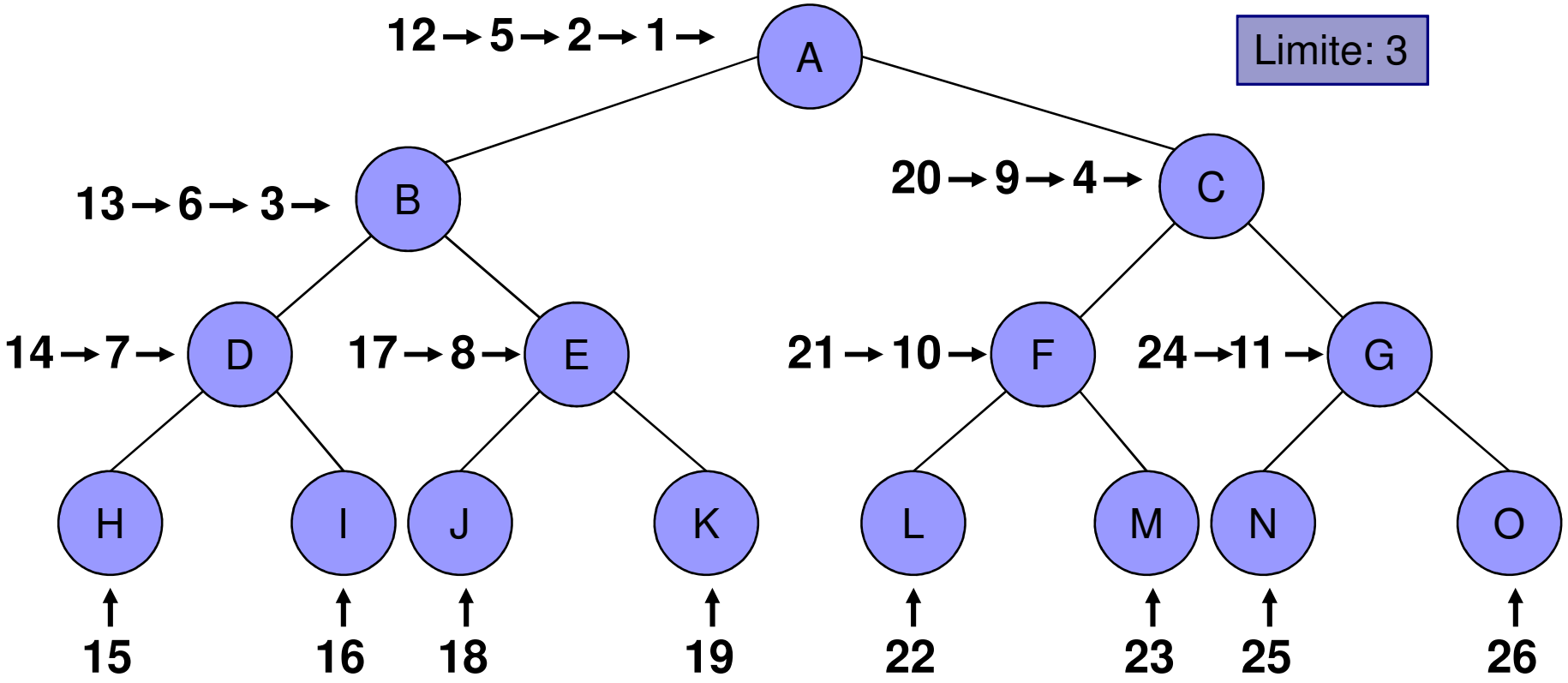
Pile:

Ordre de visite: A – B – D – E - C

Itérative en profondeur (IDS)

- Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, …
- Évite le problème de trouver une limite pour la recherche profondeur limitée.
- A les avantages de largeur d'abord (complète et optimale),
  - Mais a la complexité en espace de profondeur d'abord.
  - Donc une combinaison des deux stratégies.

Exemple - itérative en profondeur



Ordre de visite: A – A – B – C – A – B – D – E – C – F – G – A – B – D – H – I – E – J – K – C – F – L – M – G – N – O

Propriétés itérative en profondeur

- Complétude: Oui
- Complexité en temps:
  - $(d+1)b^0 + db^1 + (d-1)b^2 + b^d = O(b^d)$
- Complexité en espace: $O(bd)$
- Optimal?
  - Oui, si le coût de chaque action est de 1.
  - Peut être modifiée pour une stratégie de coût uniforme.

Comparaison:

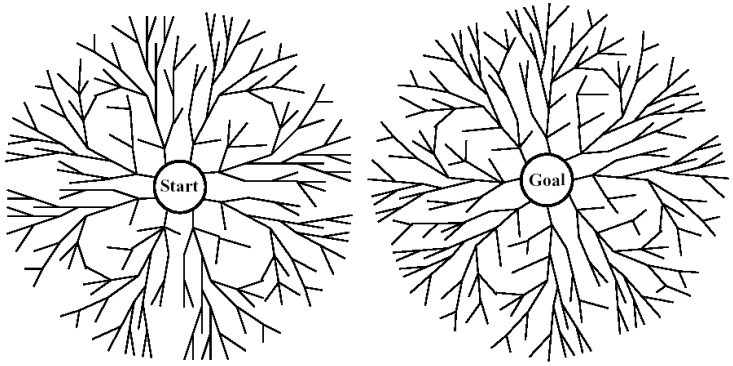
Itérative en profondeur et largeur d'abord

- Comparaison pour b = 10 et d = 5, et la solution à l'extrémité droite

N(IDS) = 6 + 50 + 400 + 3000 + 20 000 + 100 000
= 132 456 nœuds

N(BFS) = 1 + 10 + 100 + 1000 + 10 000 + 100 000 + 999 990
= 1 111 101 noeuds

Recherche bidirectionnelle



- Recherche simultanée
  - Du départ vers le but
  - Et du but vers le départ
  - Afin de se rejoindre au milieu.
- Applicable si on peut faire une cherche à partir du but.
- Besoin d'une vérification efficace de l'existence d'un nœud commun aux deux arbres
  - Table de hachage.
  - Il faut conserver tous les nœuds d'au moins un des arbres.

Propriété bidirectionnelle

- Complétude: Oui
- Complexité en temps: $O(b^{d/2})$
- Complexité en espace: $O(b^{d/2})$
- Optimale: Oui

Répétition d'états

- La répétition d'états fait perdre du temps
  - Dans le pire cas, la recherche tourne en rond dans les cycles créés.
- Détection de répétitions
  - Normalement faite en comparant les nouveaux nœuds aux nœuds déjà développés.
- Avant d'éliminer le nouveau nœud
  - On doit vérifier s'il est meilleur que le nœud que l'on a déjà.

Répétition d'états :

Exploration de type *Graph-Search*

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```