Intelligence artificielle
La résolution automatique de problèmes

Hatem Ghorbel & Stefano Carrino

Ref. *Artificial Intelligence: A Modern Approach*
by Stuart Russell (Author), Peter Norvig

Introduction

- Un grand nombre de problèmes d'IA sont caractérisés par l'absence d'algorithmes permettant de construire directement une solution à partir de la donnée du problème
- On doit donc parcourir un ensemble de possibilités pour trouver
  - la meilleure. . .
  - . . . ou une pas trop mauvaise !
- Souvent, l'ensemble des solutions à parcourir est très grand et ne peut pas être considéré en entier
  - Sinon ce ne serait sans doute pas de l'IA. . .
- On doit donc utiliser des *méthodes heuristiques*.

Heuristique

- DEFINITION: Une **heuristique** est toute approche de résolution de problèmes qui emploie une méthode pratique qui n'est pas garantie d'être optimale, parfaite ou rationnelle, mais est néanmoins suffisante pour atteindre un objectif ou une approximation immédiate et à court terme.

Recherche dans un espace d'états

- Caractérisation générale d'un problème de recherche dans un espace d'états :
  - Un ensemble d'états X, partagé en états ***légaux*** et en états ***illégaux*** ($X = L \cup I,\ L \cap I = 0$)
  - Un état ***initial*** $i \in X$
  - Des états ***finaux*** $F \subset X$ (éventuellement un seul $F = \{f\}$)
  - Les états sont reliés par des ***transitions*** $T \subset X \times X$
  - Un ensemble O d'***opérateurs***. Chaque état possède un sous-ensemble d'opérateurs ***applicables*** $Op(x) \subset O$
  - À chaque opérateur applicable correspond une ***transition*** vers un autre état.

Recherche dans un espace d'états

- Le problème peut alors se poser de deux manières
  - Trouver un état final $f \in F$ atteignable depuis l'état initial i
  - Trouver une suite d'opérateurs (un chemin) permettant de passer de i à un $f \in F$

Remarques

- Les états finaux peuvent être donnés en *extension* (énumération de tous les états possibles) ou en *intension* (description des caractéristiques d'un état final)
- On ne s'intéresse pour l'instant qu'à des cas *déterministes* : dans un état donné, l'application d'un opérateur donné aura toujours le même résultat
- Les éléments ci-dessus définissent au fait un graphe appelé *graphe d'états*
  - Ce graphe n'est généralement pas représenté explicitement dans la mémoire (problème d'espace)…
  - … mais construit au fur et à mesure des besoins !

Exemple 1: Le loup, la chèvre et le chou

- État initial
  - Le loup, la chèvre, le chou et le bateau sur la rive gauche
- État final
  - Le loup, la chèvre, le chou et le bateau sur la rive droite
- Opérateurs
  - Transporter en bateau le loup, la chèvre ou le chou d'une rive à l'autre. Applicable si le bateau est sur la bonne rive
- État légal
  - Ne pas avoir le loup et la chèvre ou la chèvre et le chou sur la même rive sans le bateau.

Exemple 2: Conception d'horaires

- État initial
  - Un horaire vide
- États finaux
  - Tous les cours sont placés sans conflits
- Opérateur
  - Placer ou déplacer un cours.

- Dans ce cas, il est illusoire de parcourir l'ensemble des possibilités !

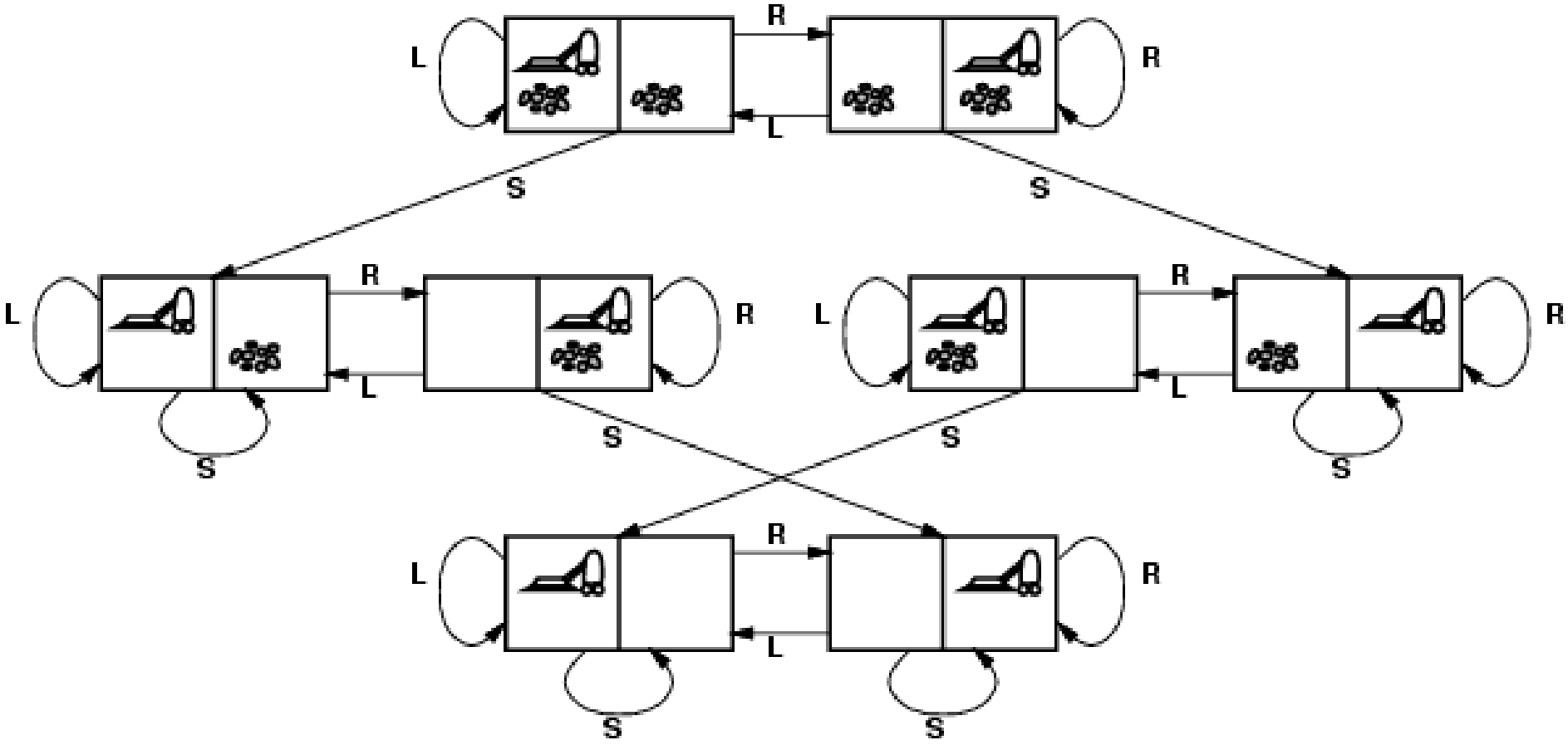
Représenter la situation

- Pour raisonner, faire des inférences, il faut une représentation manipulable.
  - **État actuel** (où se trouve l'agent, ce qui l'entoure, état de progression, ressources disponibles, etc.)
  - But à atteindre : **un état final** du monde désiré
  - États légaux : l'arbre / le réseau des **états possibles et légaux** (l'espace du problème)
  - Les moyens disponibles pour progresser (**opérations possibles**)
  - Le coût de chaque opération

Représenter la situation

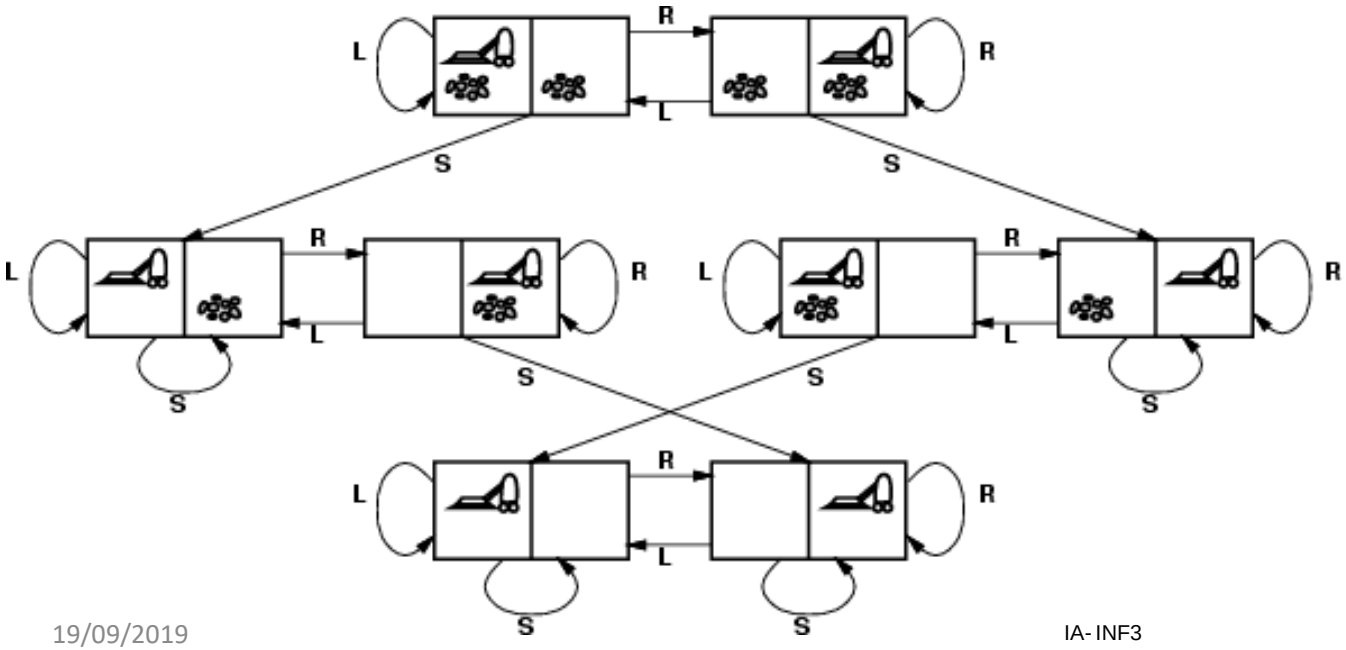
- Pour Robi, un robot balayeur dans un hôtel de deux chambres:
  - Etat actuel : deux chambres empoussiérées ← manque-t-il quelque chose ici?
    Où se trouve Robi?
  - But / état souhaité : deux chambres nettoyées
  - Moyens disponibles pour progresser : aller à gauche, aller à droite, aspirer la poussière

Espace des états d'un micro-monde: *Robi l'aspirateur*

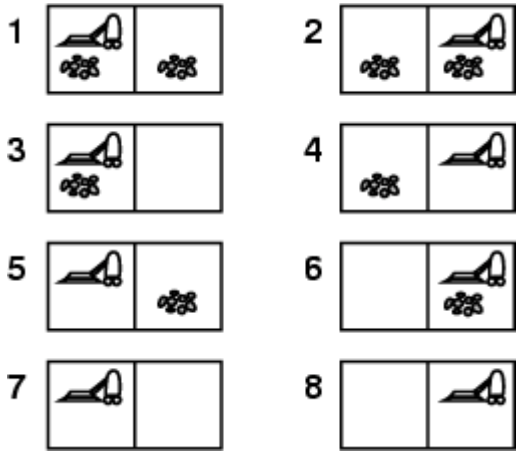


Représentation d'un problème

- Définitions
  - États légaux (possibles) du problème

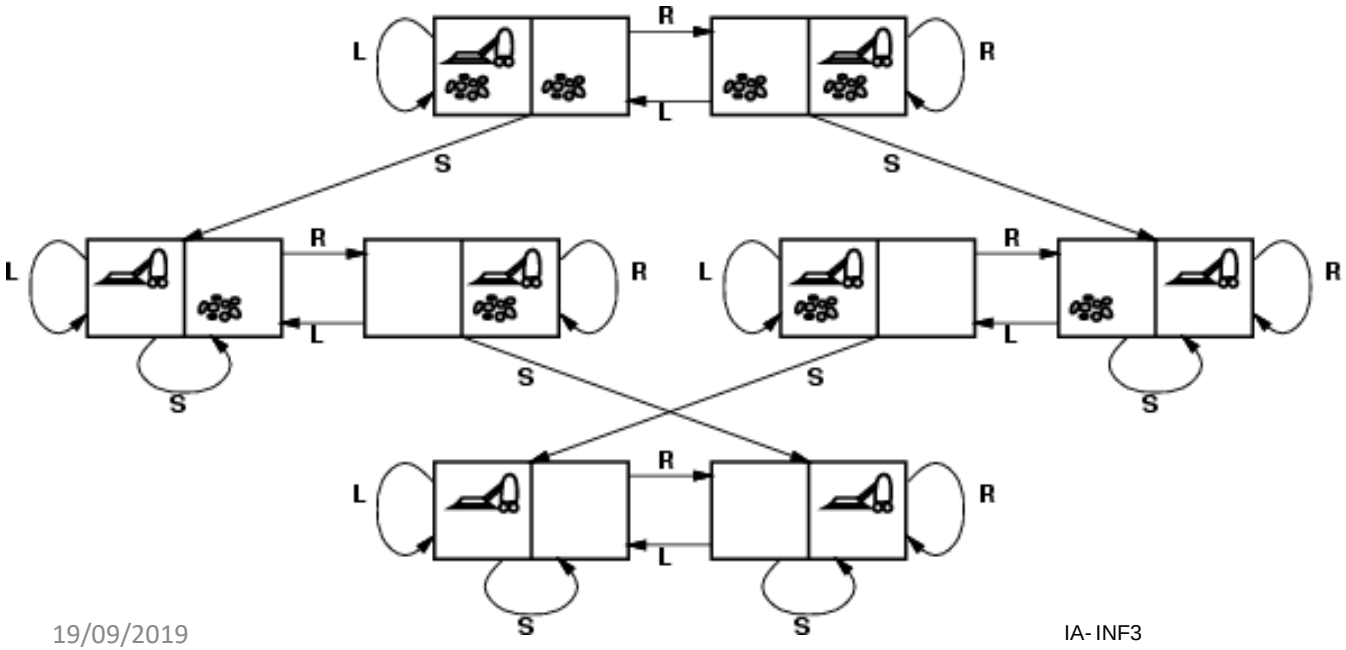


huits états légaux



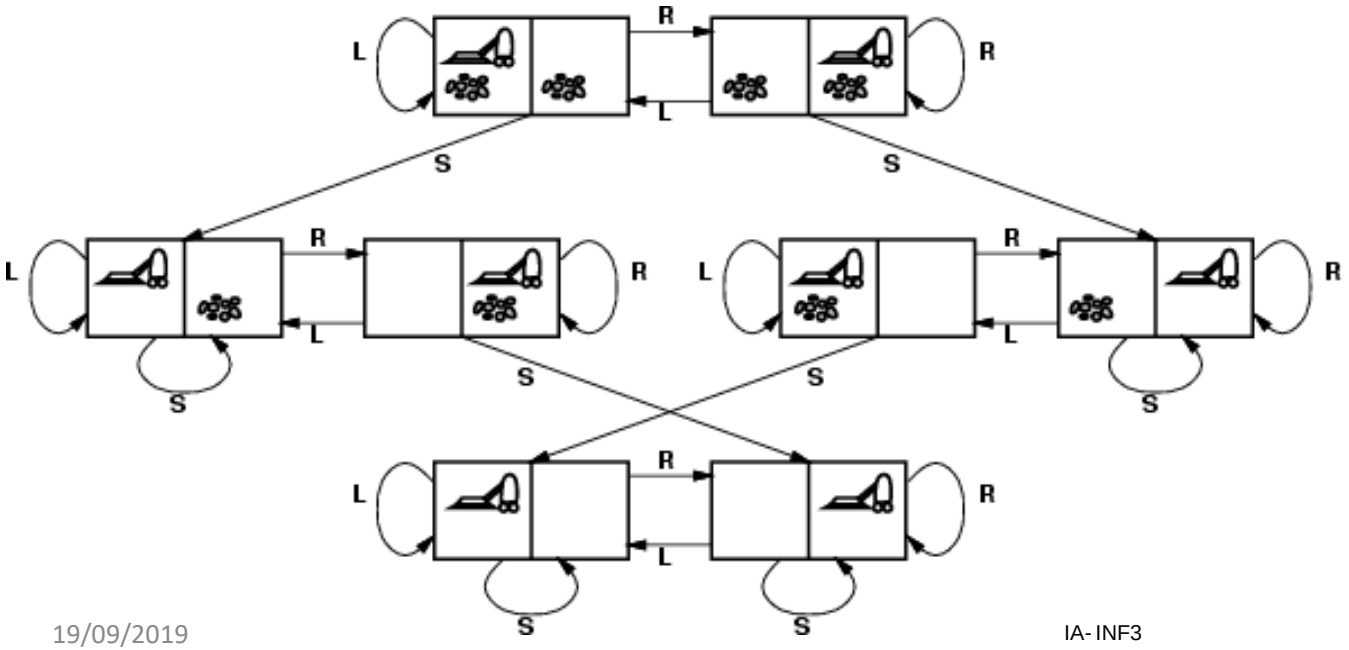
Représentation d'un problème

- Définitions
  - États légaux (possibles) du problème
  - Transitions (Opérateurs de transformation d'état)
  - État initial



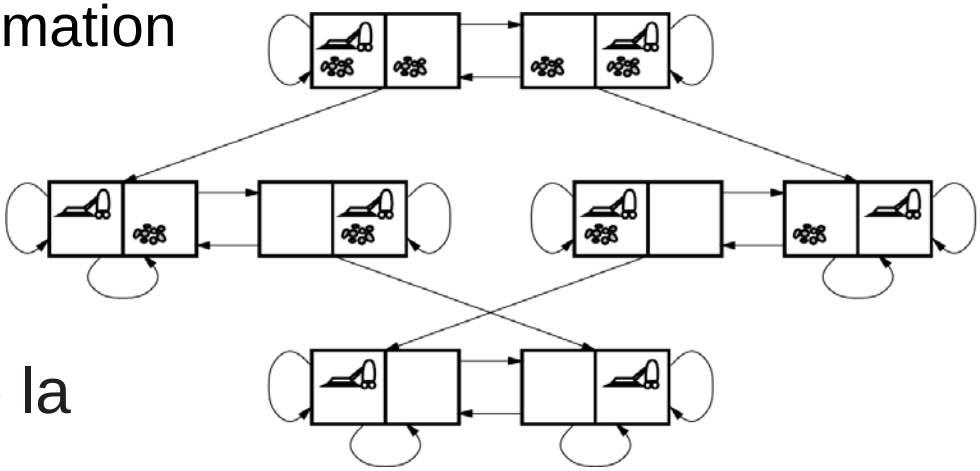
Représentation d'un problème

- Définitions
  - États légaux (possibles) du problème
  - Transitions (Opérateurs de transformation d'état)
  - État initial
  - État final (But)



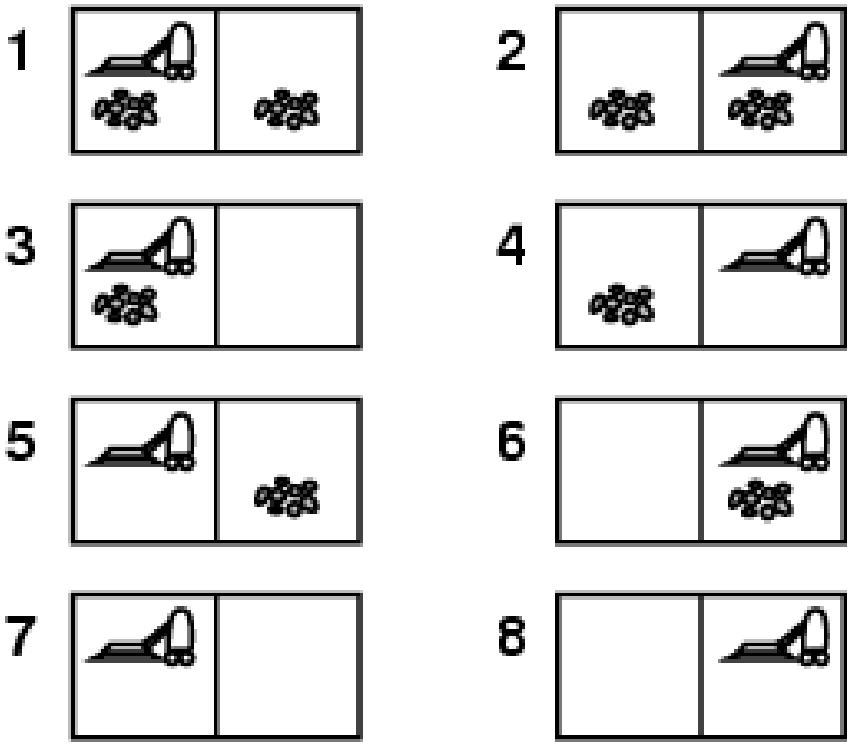
Représentation d'un problème

- Définitions
  - États légaux (possibles) du problème
  - Transitions (Opérateurs de transformation d'état)
  - État initial
  - État final (But)
- Solution au problème = Résultat de la recherche:
  - Un **chemin**: une séquence d'états allant de l'état initial à l'état final souhaité (au but)
- Lorsqu'un chemin est trouvé, l'agent l'exécute du début à la fin.



Le monde de l'agent aspirateur

- Etats légaux ??
- Etat Initial ??
- Opérateurs ??
- Etat final (but) ??
- Coût du chemin ??



Le monde de l'agent aspirateur

- Etats légaux : L'agent est dans un des deux emplacements, qui peuvent contenir ou non de la poussière (8 états en tout)
- Etat Initial : n'importe quel état
- Opérations : Gauche, Droite, Aspirer
- Etat final (but) : Vérifier que tous les emplacements sont propres
- *Coût du chemin : somme du nombre d'étapes qui composent le chemin*

Exercice: Le loup, la chèvre et le chou

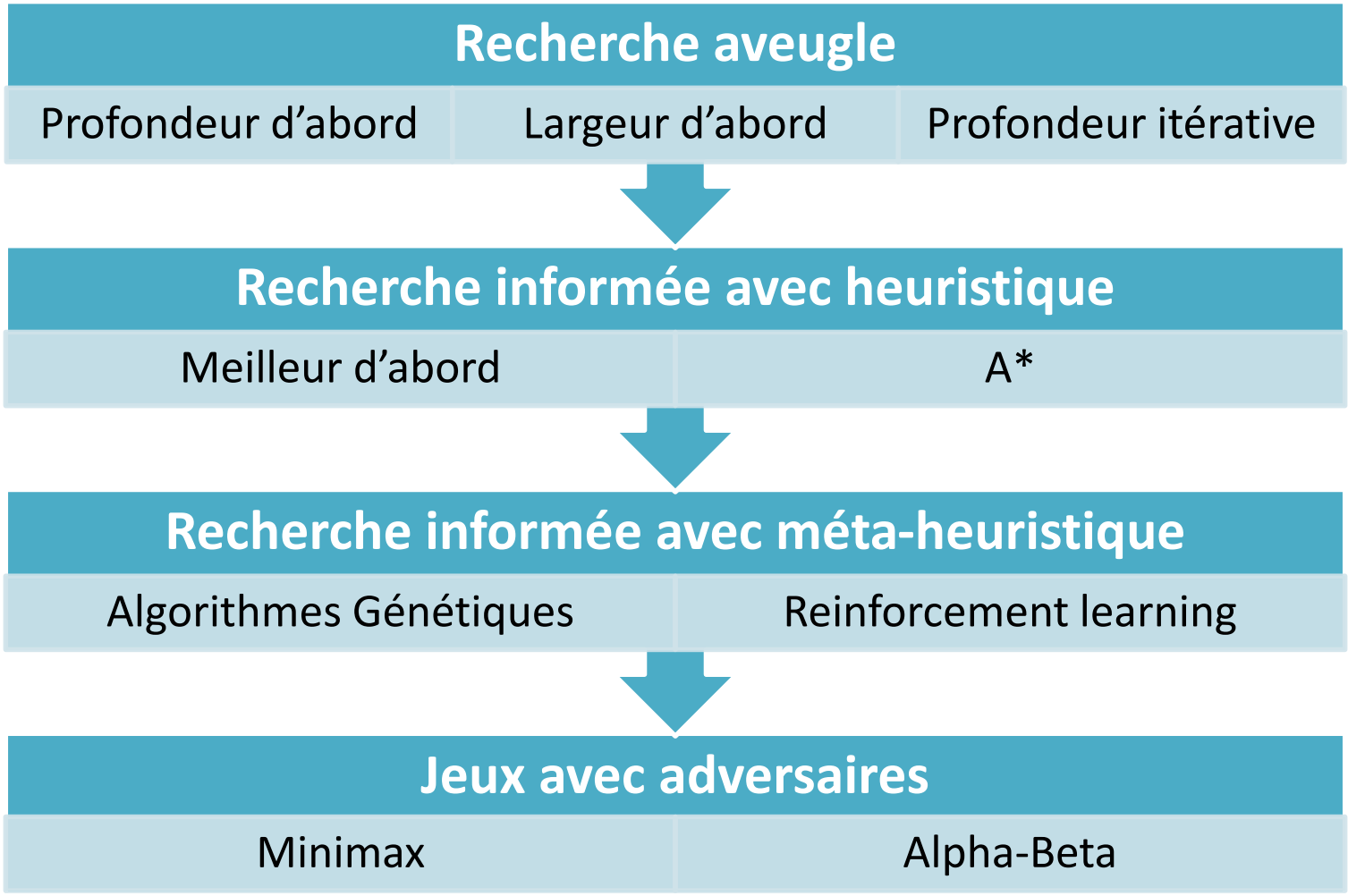
- État initial
  - Le loup, la chèvre, le chou et le bateau sur la rive gauche
- État final
  - Le loup, la chèvre, le chou et le bateau sur la rive droite
- Opérateurs
  - Transporter en bateau le loup, la chèvre ou le chou d'une rive à l'autre. Applicable si le bateau est sur la bonne rive
- État légal
  - Ne pas avoir le loup et la chèvre ou la chèvre et le chou sur la même rive sans le bateau.
- **Trouver une représentation du problème**
- **Schématiser la solution**

Des exemples de situations nécessitant la recherche de solutions

- Disposition des composantes sur un circuit intégré
  - Situation initiale : des millions de composantes non organisées
  - Opérateurs : Transitions possibles indiquées par les contraintes de connectivité
    - pas de superpositions de composantes
    - espace disponible pour passer les chemins entre les composantes
  - But : disposition qui minimise l'espace, les délais de transport de l'information, les pertes d'énergie, et les coûts de fabrication
- Assemblage robotique
- Recherche d'un trajet pour aller de Saint-Gallen à Genève
- Conception de protéines…

Stratégies de recherche

- Overview du semestre d'automne



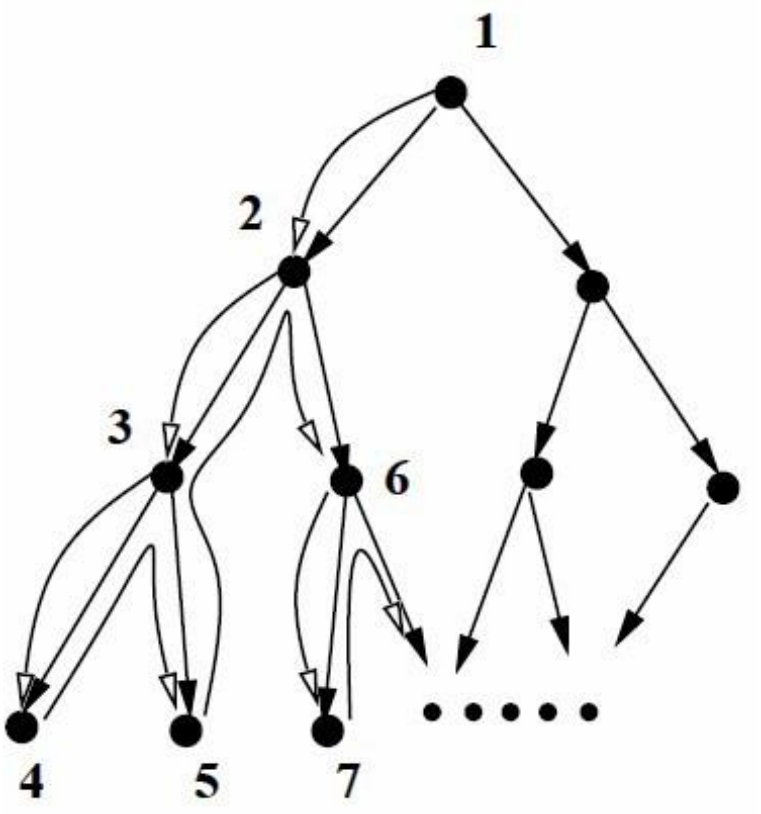
Stratégie de recherche

- La stratégie indique un ordre de développement des nœuds
- 4 dimensions d'évaluation d'une stratégie:
  - Complétude : permet-elle de toujours trouver une solution s'il en existe une ?
  - Complexité en temps : nombres de nœuds générés pour trouver la solution
    - Question : importance de ce critère => combien de temps pour trouver la solution
  - Complexité en espace : nombre maximal de nœuds mémorisés
    - Question : importance de ce critère => combien de mémoire il pour effectuer la recherche
  - Optimalité : permet-elle de toujours trouver le chemin le moins coûteux (ex. le plus court)?

Stratégie de recherche

- La complexité en temps et en espace se mesurent en termes de :
  - b : facteur de branchement (*branches*) maximal dans l'arbre de recherche
  - d : profondeur (*depth*) du but le plus "en surface"
    (du plus court chemin vers un but, de la solution la moins coûteuse)
  - m : profondeur *max*imale de l'espace d'état
    (longueur maximale de n'importe quel chemin; peut être ∞)

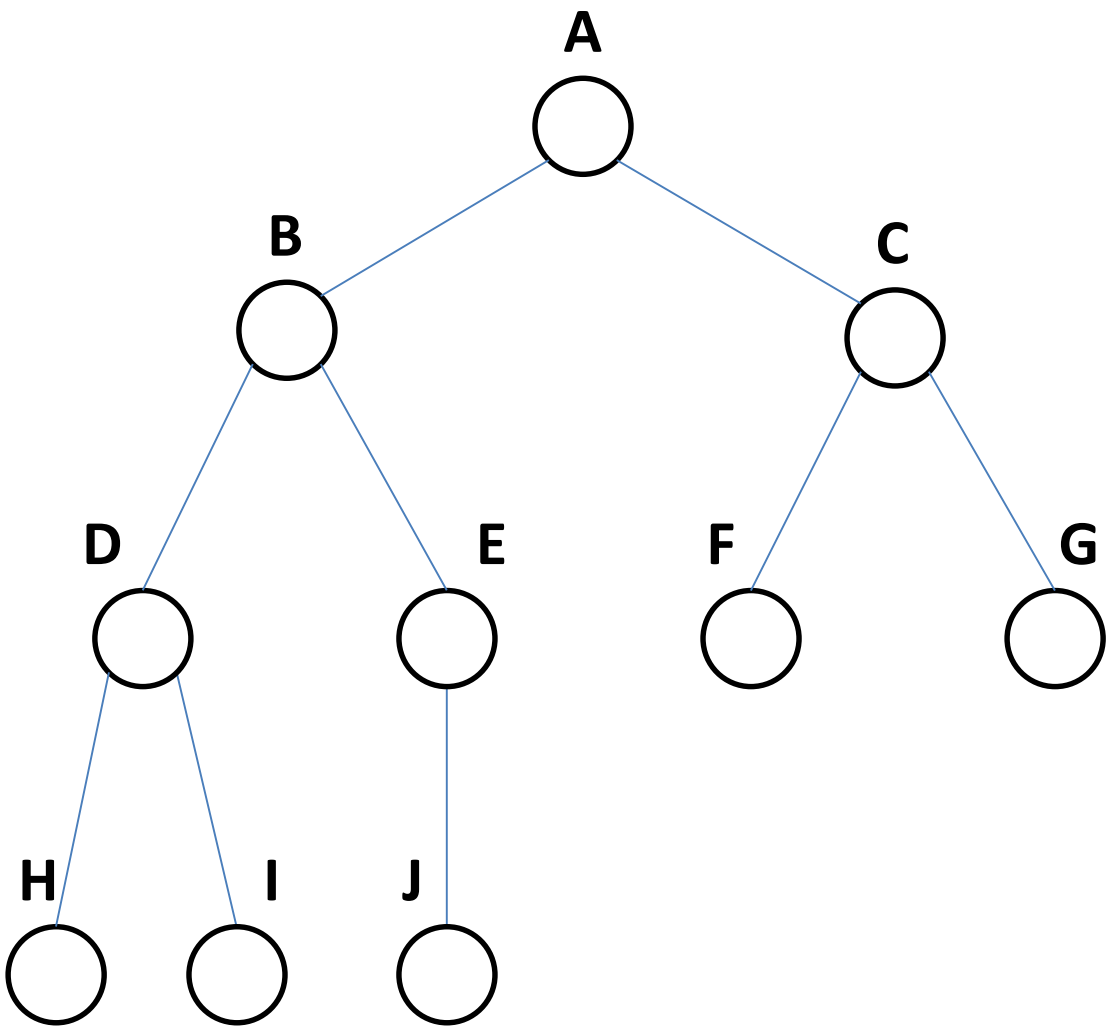
Stratégie de recherche: *En profondeur d'abord*



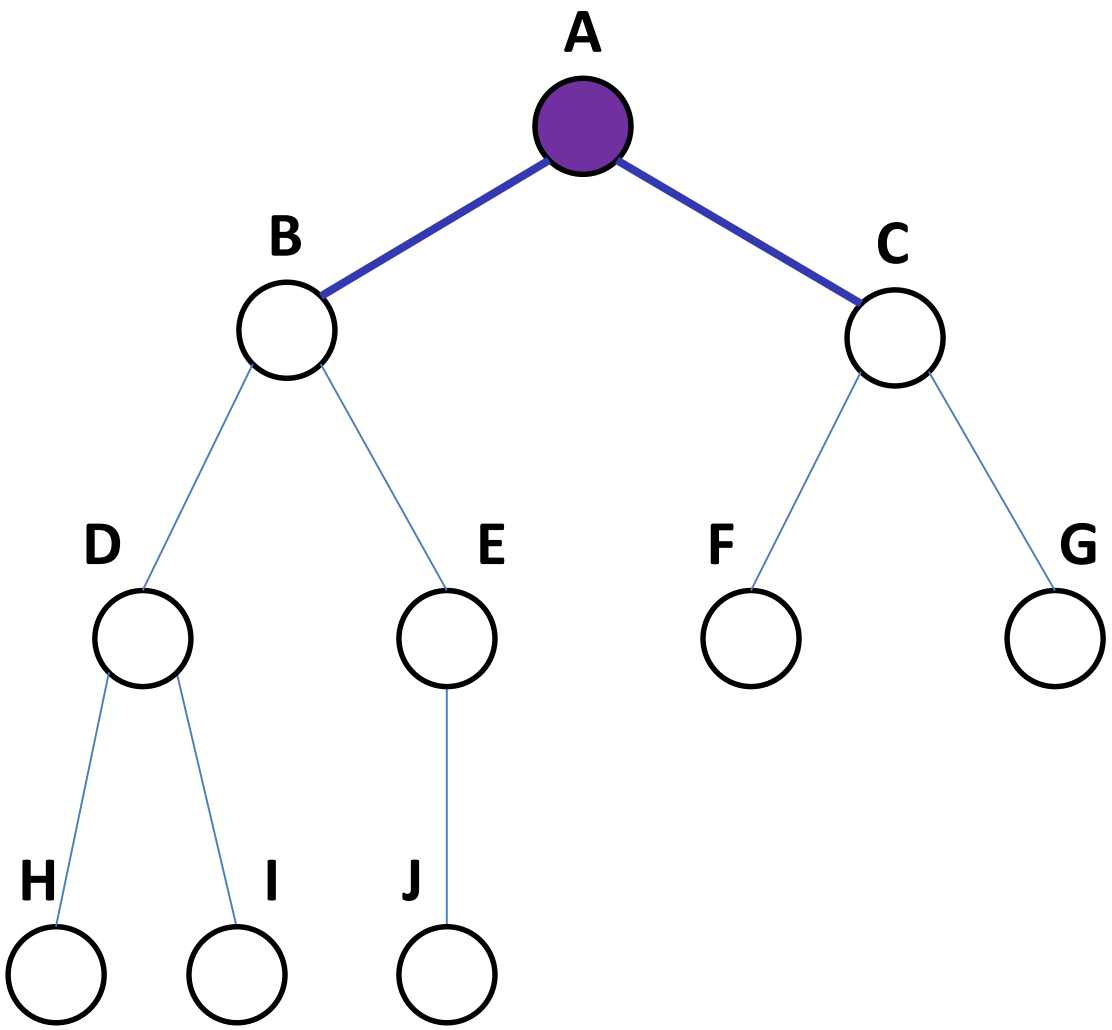
Source: École polytechnique fédérale de Lausanne

- Expansion (et examen) du premier nœud jusqu'à ce qu'il n'y ait plus de successeur
- Retour arrière au niveau précédent
- Modeste en mémoire: ne conserve en mémoire que
  - le chemin direct actuel allant de l'état initial à l'état courant,
  - les enfants (nœuds ouverts) non examinés du parent actuel
  - les nœuds en attente le long du chemin parcouru.
- La recherche s'arrête lors qu'une solution est trouvée

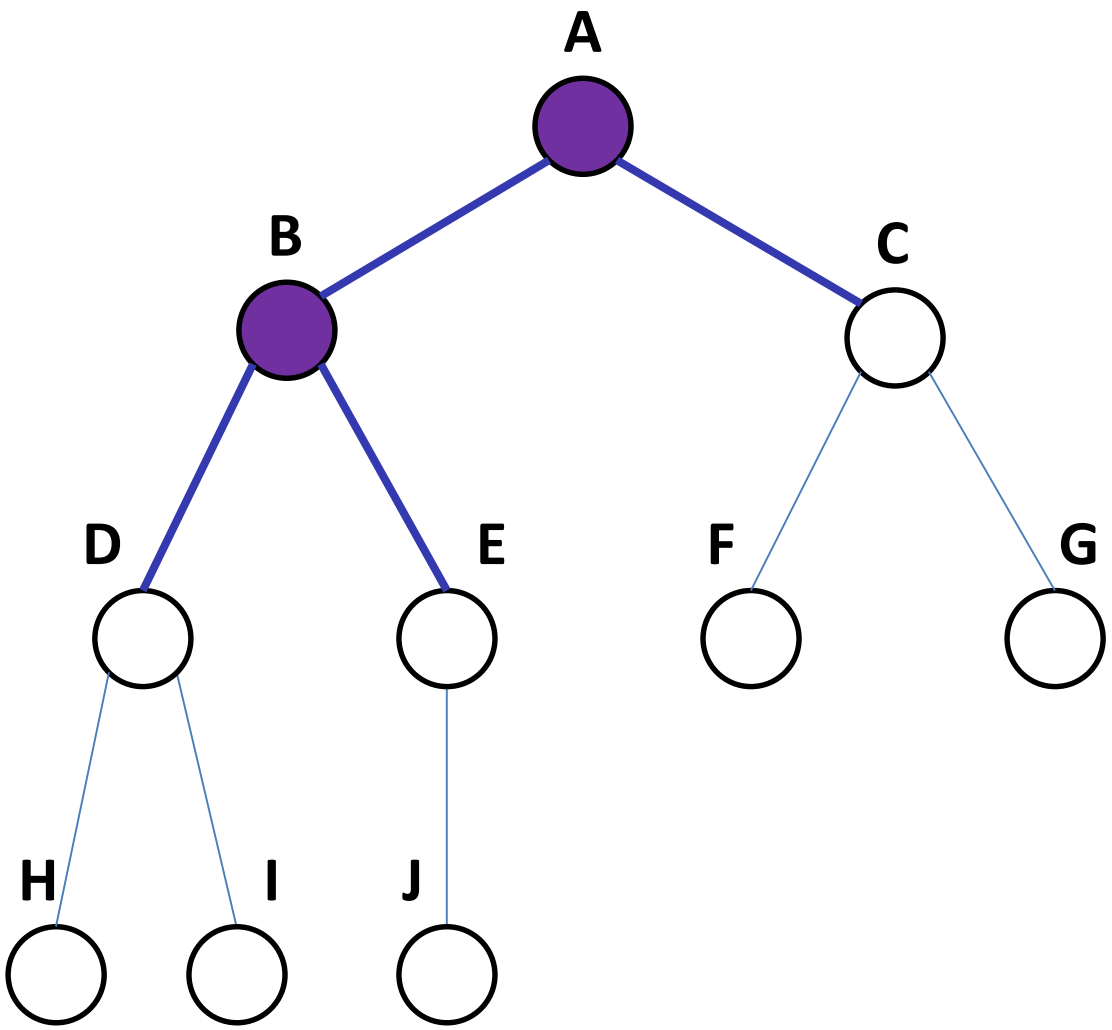
Stratégie de recherche: *En profondeur d'abord*



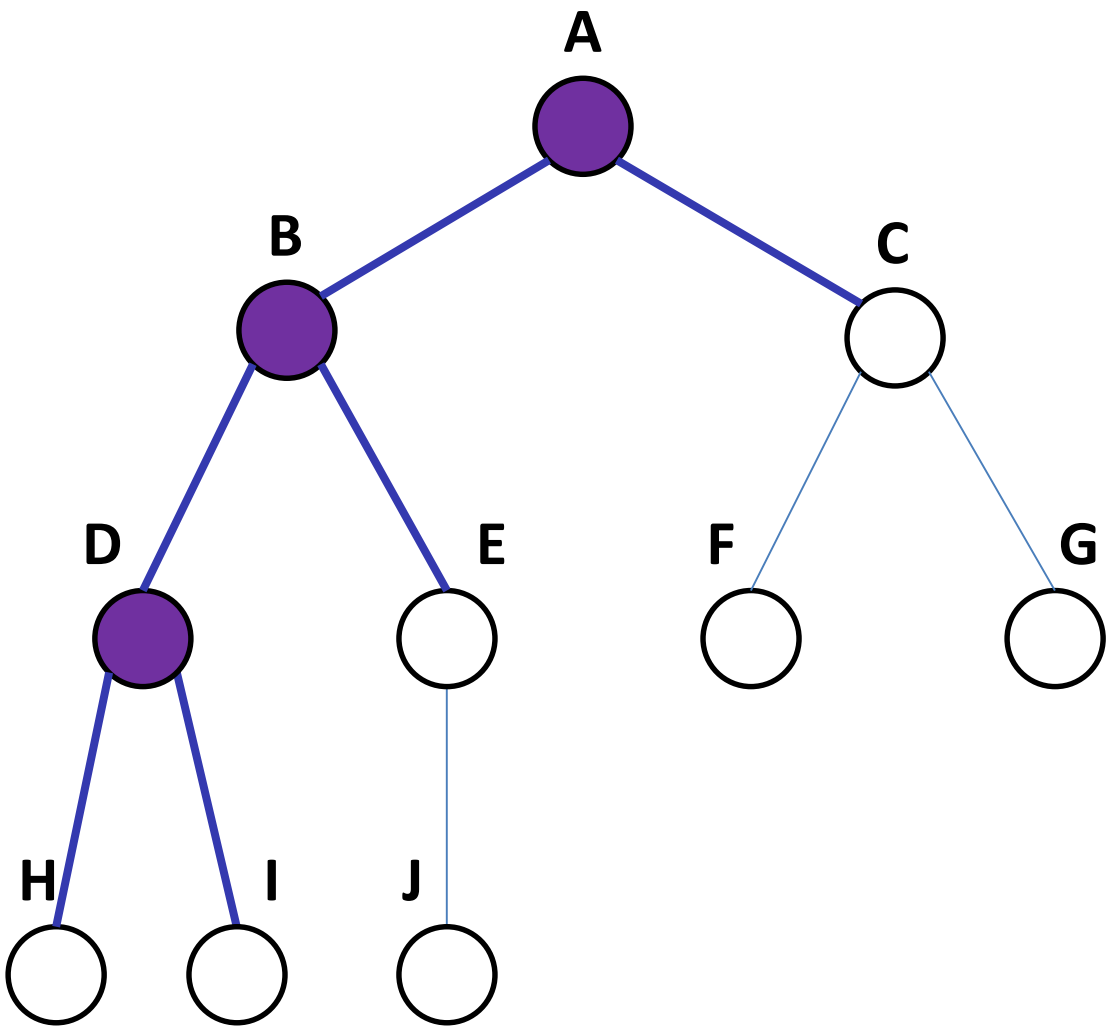
Stratégie de recherche: *En profondeur d'abord*



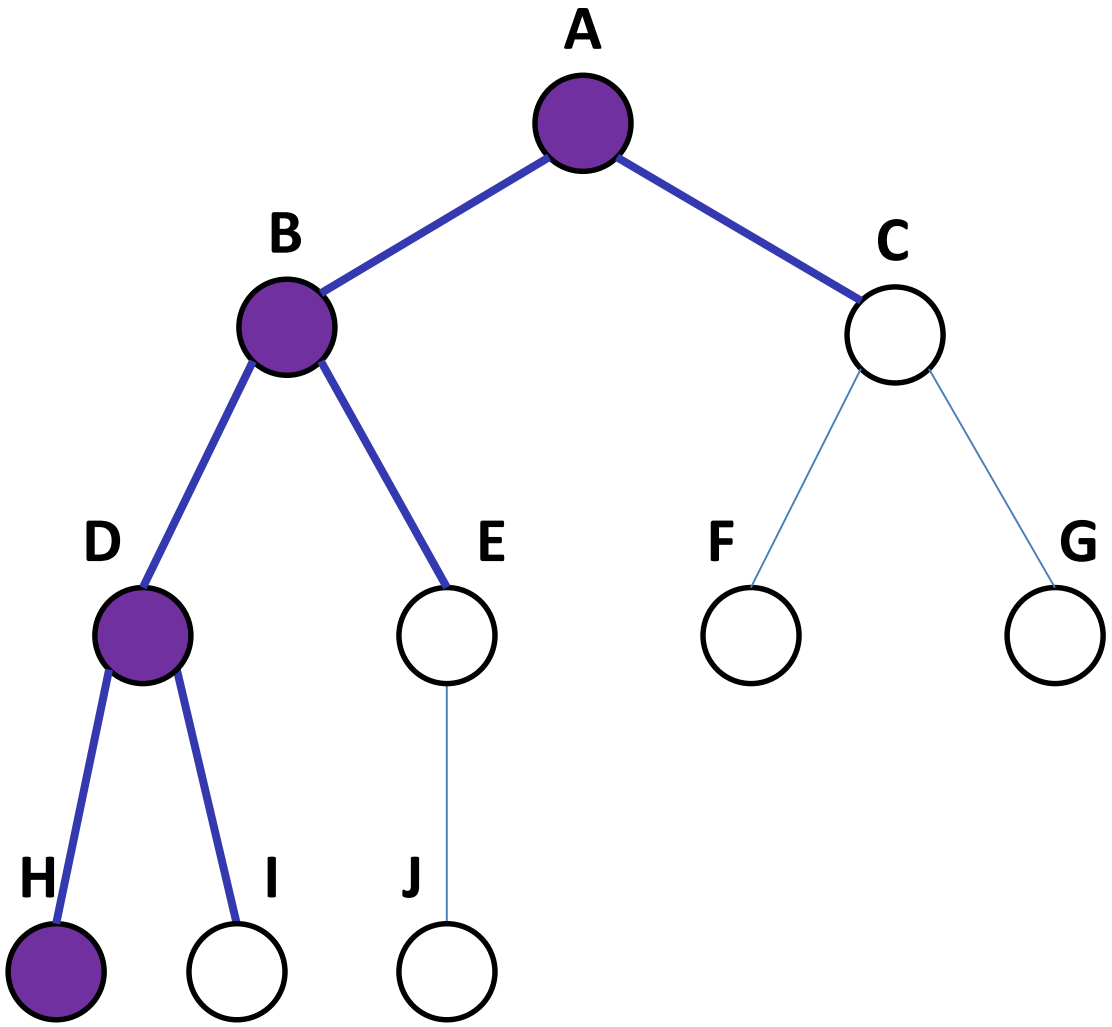
Stratégie de recherche: *En profondeur d'abord*



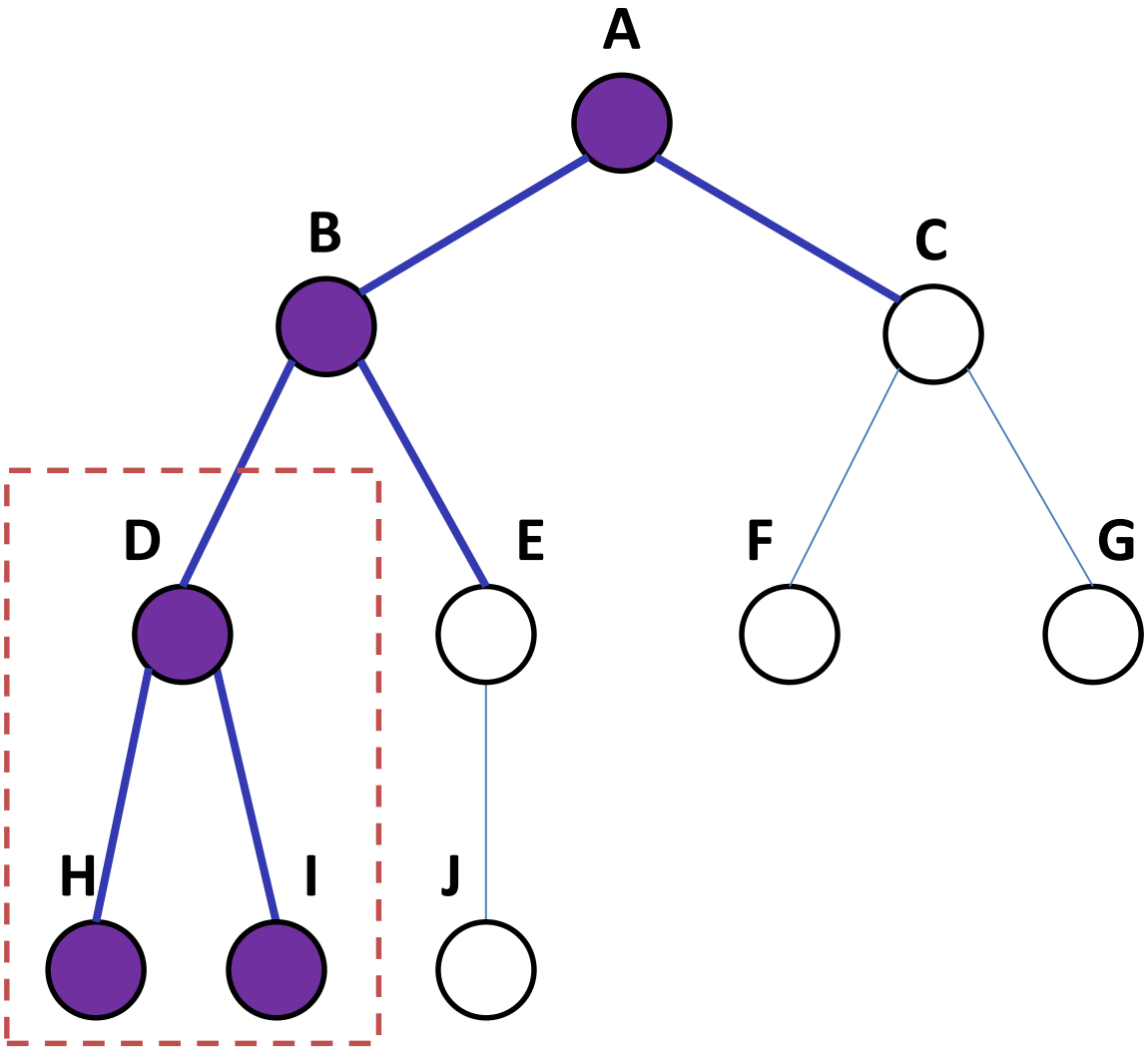
Stratégie de recherche: *En profondeur d'abord*



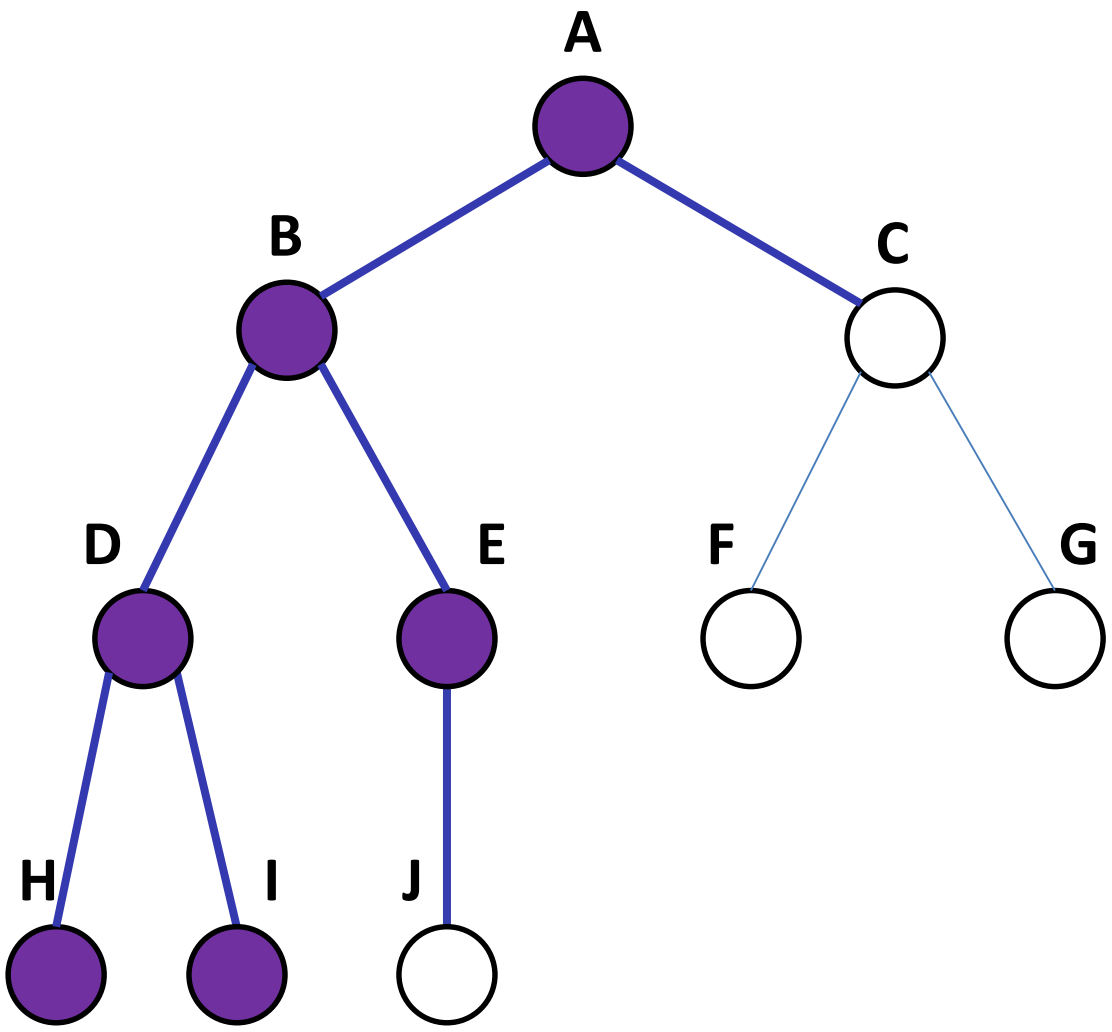
Stratégie de recherche: *En profondeur d'abord*



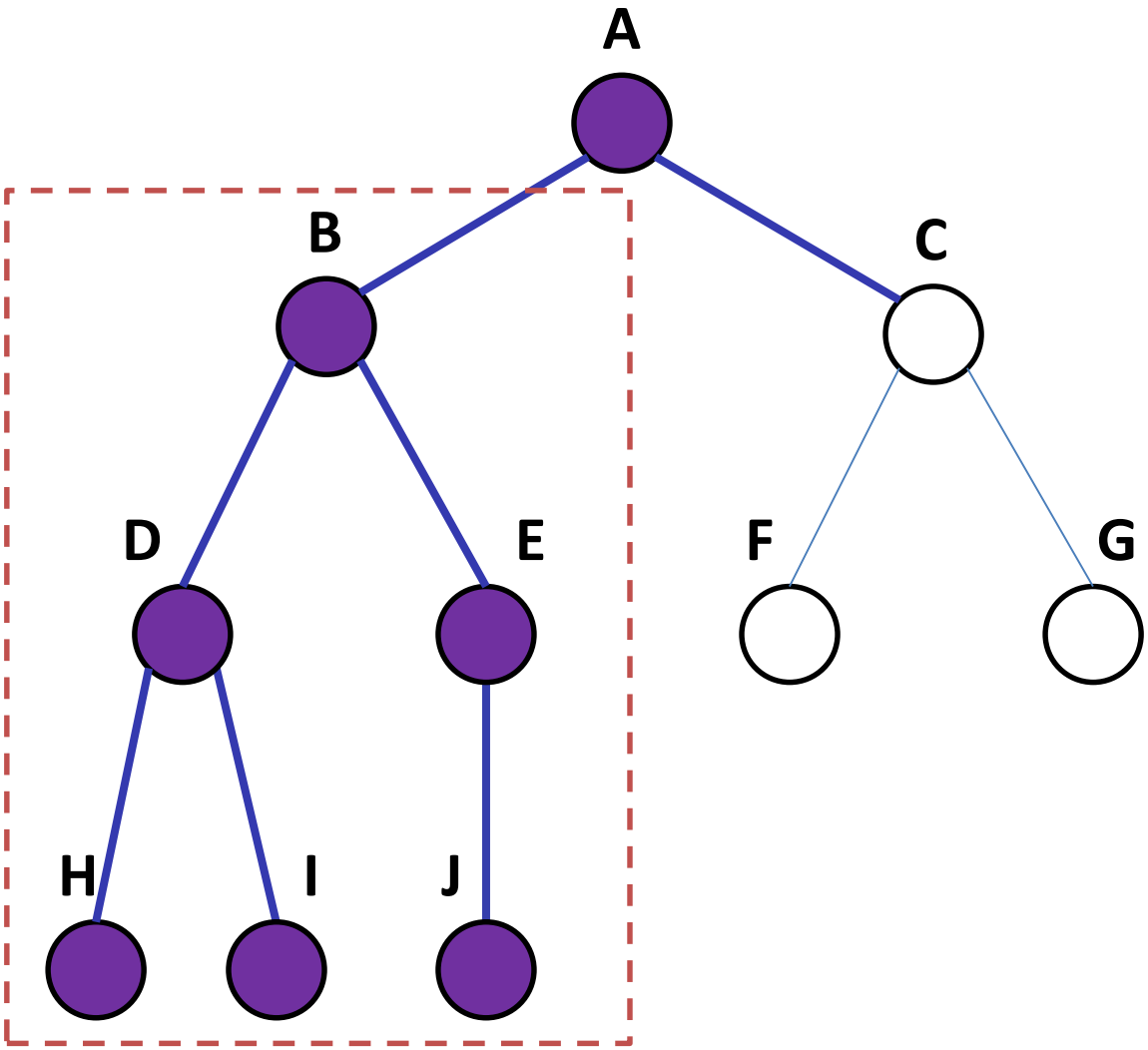
Stratégie de recherche: *En profondeur d'abord*



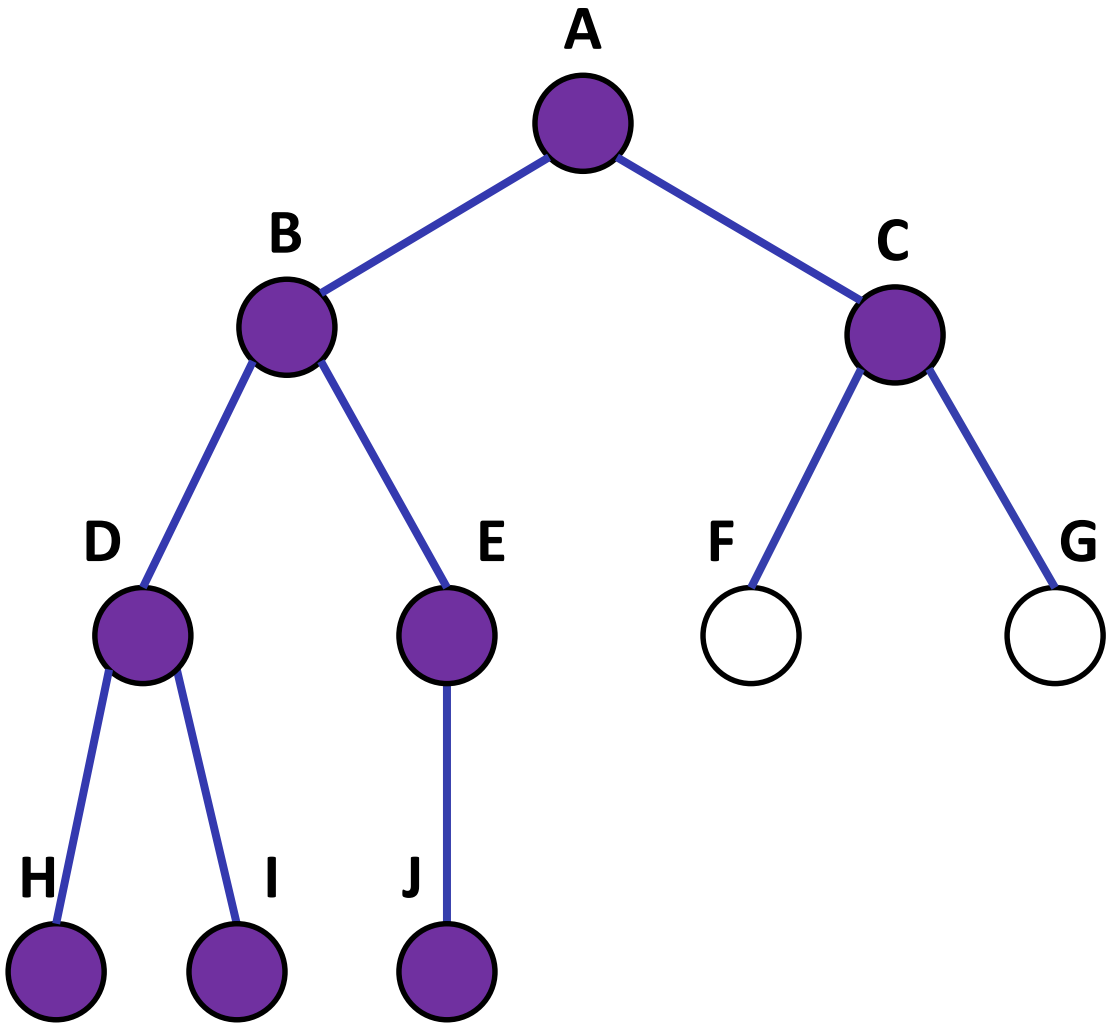
Stratégie de recherche: *En profondeur d'abord*



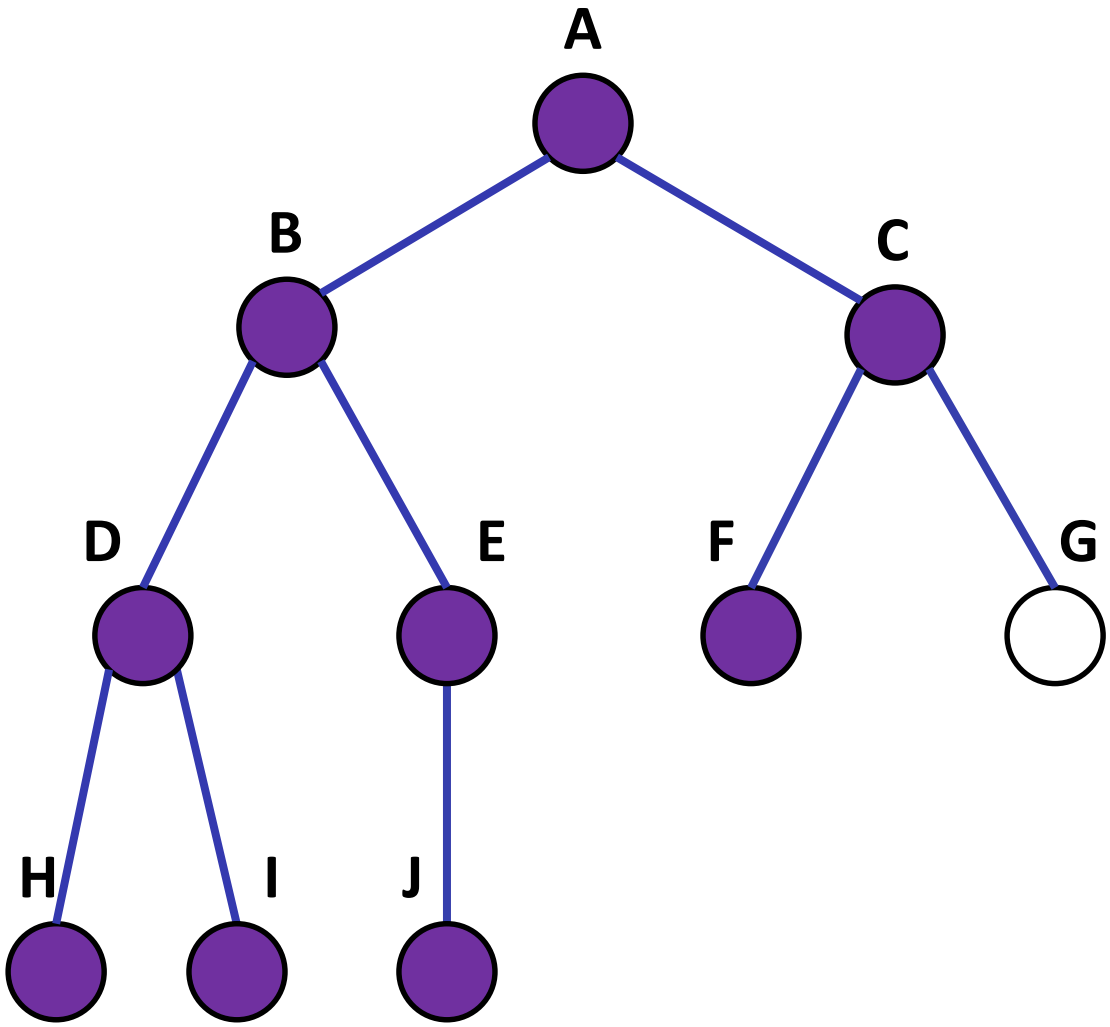
Stratégie de recherche: *En profondeur d'abord*



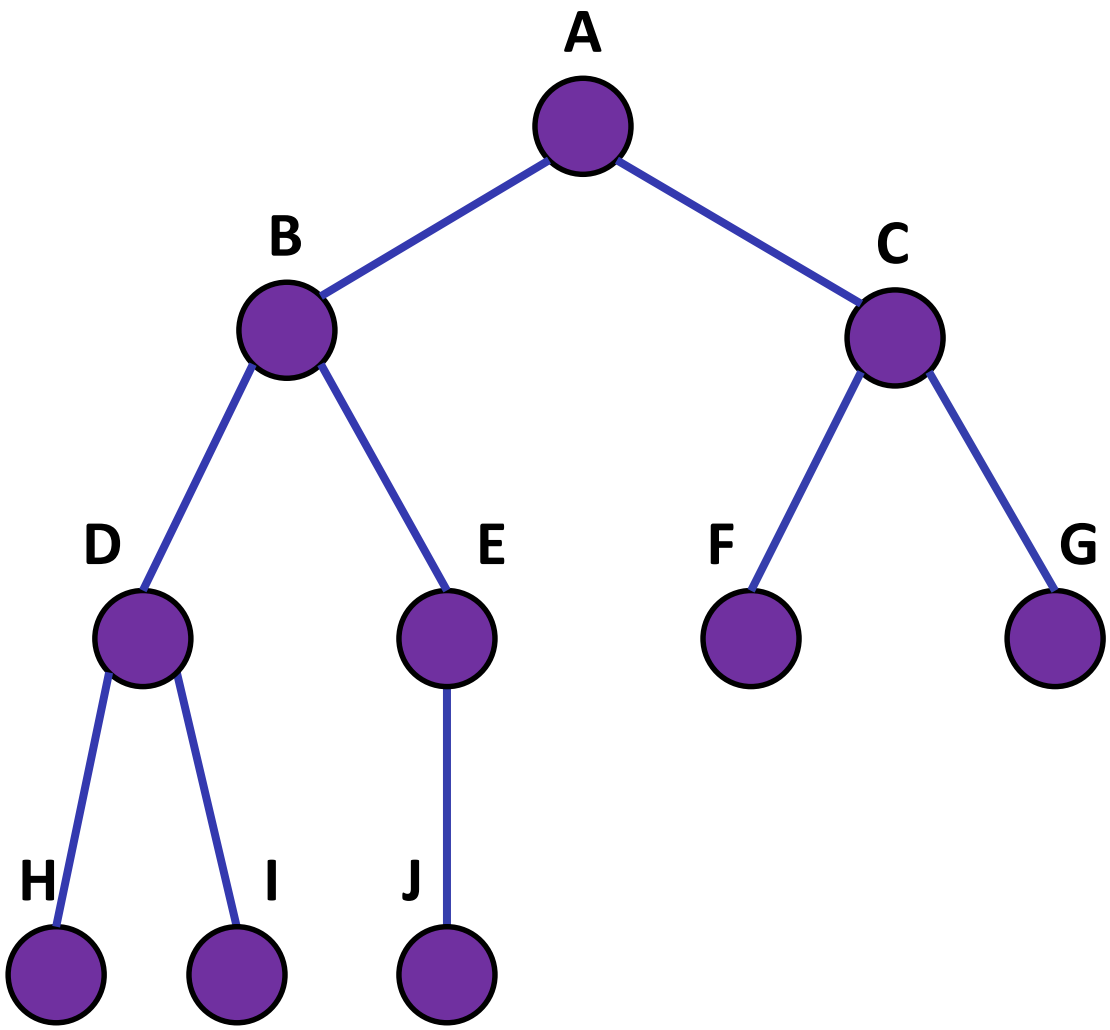
Stratégie de recherche: *En profondeur d'abord*



Stratégie de recherche: *En profondeur d'abord*



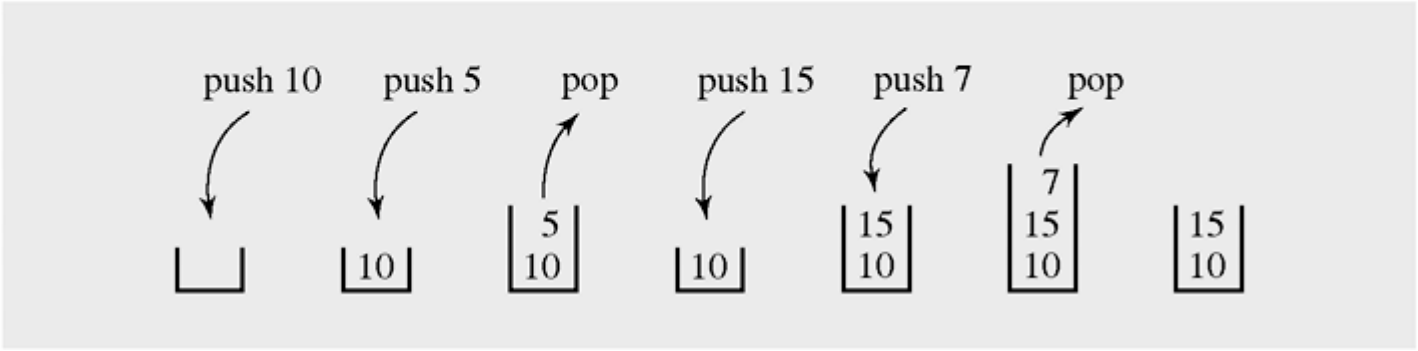
Stratégie de recherche: *En profondeur d'abord*



Stratégie de recherche: *En profondeur d'abord*

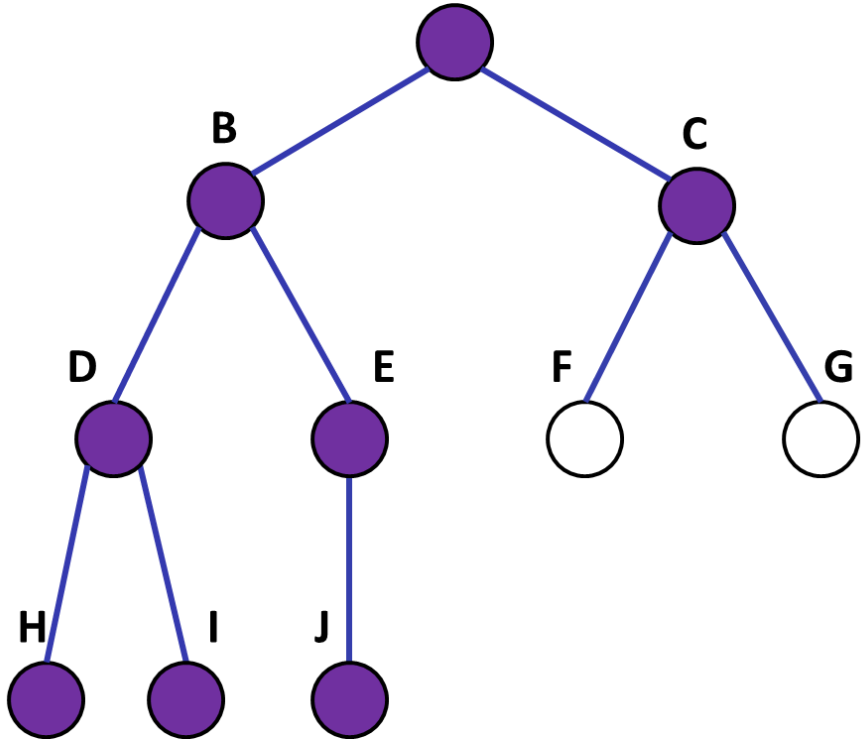
- Développer le nœud le plus profond non encore développé
- *Implémentation* : ***`Exploration_en_arbre`*** avec une pile (LIFO) => mettre les successeurs à l'avant

FIGURE 4.1 A series of operations executed on a stack.



Recherche en profondeur d'abord – Propriétés

- Complète ?
- Optimale ?
- Complexité en Temps ?
- Complexité en Espace ?



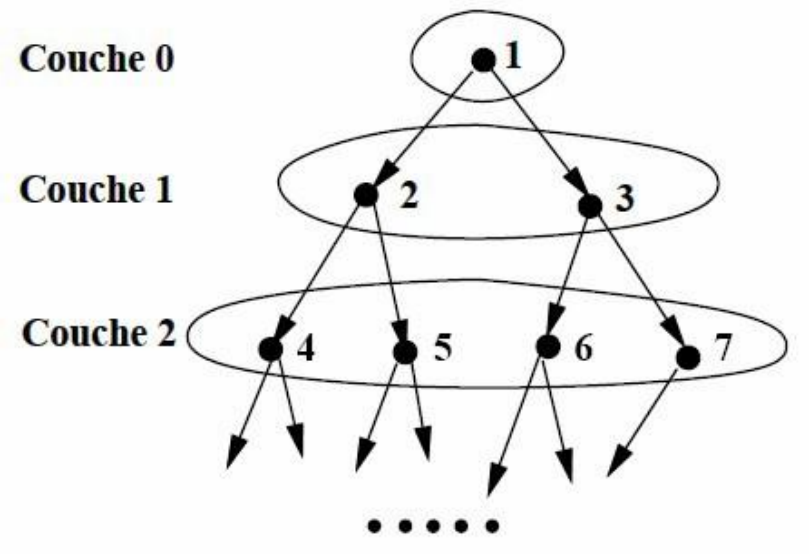
Recherche en profondeur d'abord – Propriétés

- Complète ? Non: échoue dans les cas d'espace à profondeur infinie ou d'espace avec boucles.
- Optimale ? Non
- Temps ? $O(b^m)$: peut être terrible si
  - m (profondeur de l'espace) > d (profondeur du but)

  Il peut y avoir beaucoup d'exploration en profondeur avant de trouver un but en surface, mais à la droite du graphe
- Espace ? $O(bm)$, i.e., espace linéaire

- Une solution: établir une profondeur limite d'exploration
- Une autre solution: seulement un successeur est généré à la fois (gain d'espace)

Stratégie de recherche : *En largeur d'abord*



Source: École polytechnique fédérale de Lausanne

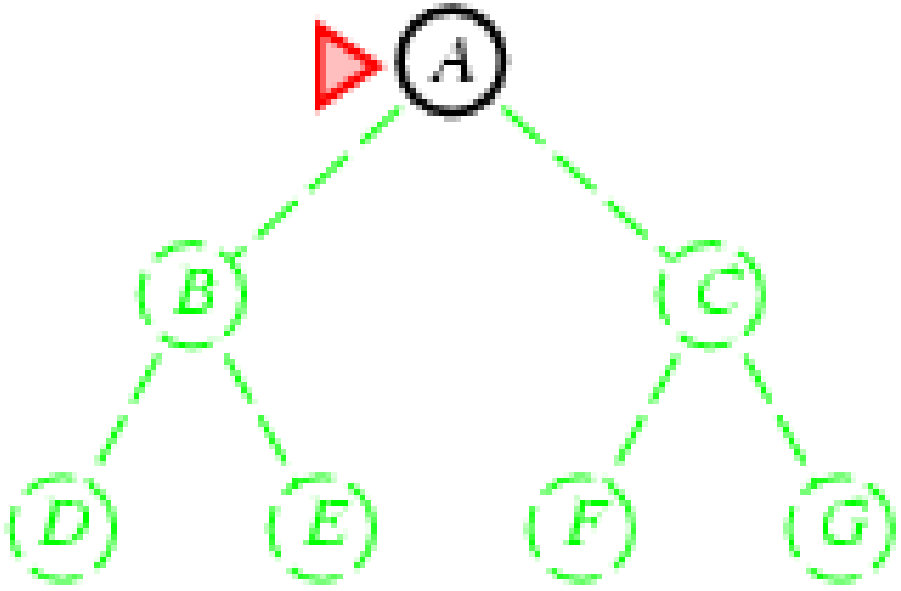
- Expansion et examen par couche
- Trouve toujours le chemin le plus court
- Exige beaucoup de mémoire pour stocker toutes les alternatives à toutes les couches.

Exploration en largeur d'abord

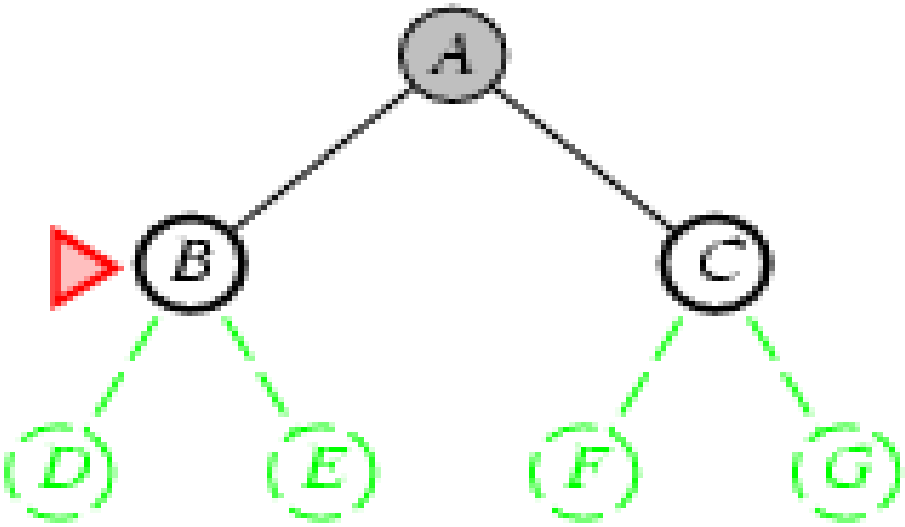
- Principe: Développer les nœuds à un même niveau de l'arbre avant d'aller au niveau suivant.
- La frontière est implantée sous forme de file FIFO (retirer le nœud le plus ancien)
- Développement: mettre les enfants à la fin de la file
- Retirer les nœuds à l'avant de la file

Exploration en largeur d'abord

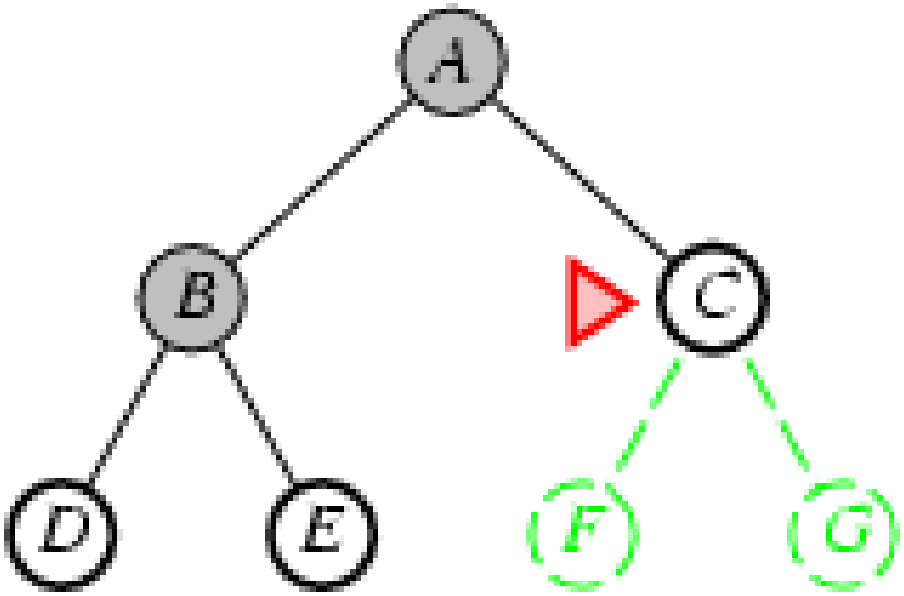
- Développer tous les nœuds d'une profondeur donnée avant de passer au niveau suivant
- Implémentation : Utiliser une file FIFO, i.e., les nouveaux successeurs sont rangés à la fin.



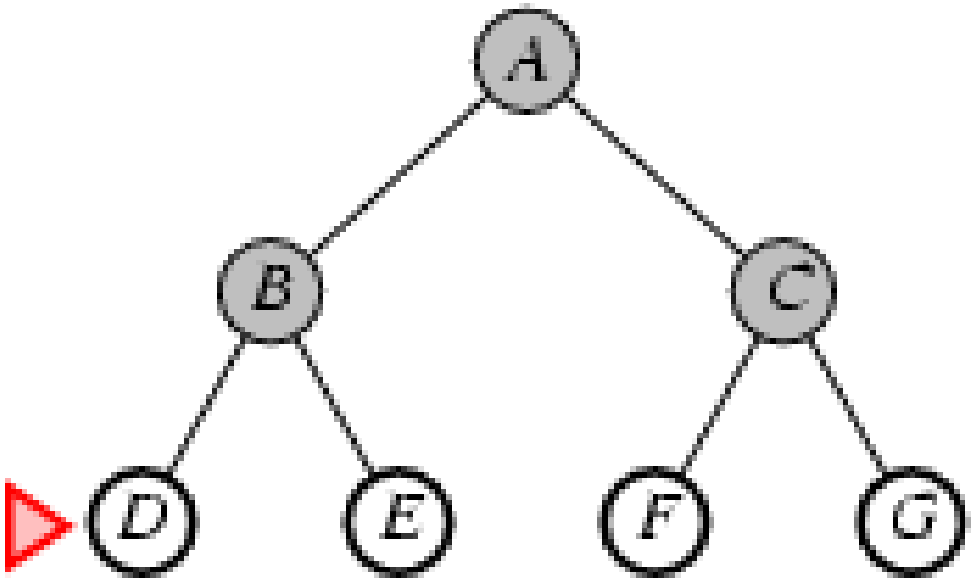
Exploration en largeur d'abord



Exploration en largeur d'abord



Exploration en largeur d'abord



Recherche en largeur d'abord Propriétés

- Complète ?
- Optimale ?
- Temps ?
- Espace ?

Recherche en largeur d'abord Propriétés

- Complète ? Oui (si *b* est fini)
- Optimale ? Oui (si le coût par étape = 1)
- Temps ? $1+b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$
- Espace ? $O(b^{d+1})$ (garde tous les nœuds en mémoire)

- L'espace est le gros problème avec cette stratégie (plus que le temps)

Le meilleur des deux mondes: Exploration Itérative en profondeur

- Profondeur d'abord est efficace en espace mais ne peut garantir un chemin de longueur minimale
- Largeur d'abord trouve le chemin le plus court (en nombre d'étapes) mais requière un espace exponentiel
- Exploration Itérative en profondeur effectue une recherche en profondeur limitée avec un niveau de profondeur croissant jusqu'à ce que le but soit trouvé.

Exploration Itérative en profondeur

- Augmentation graduelle de la limite

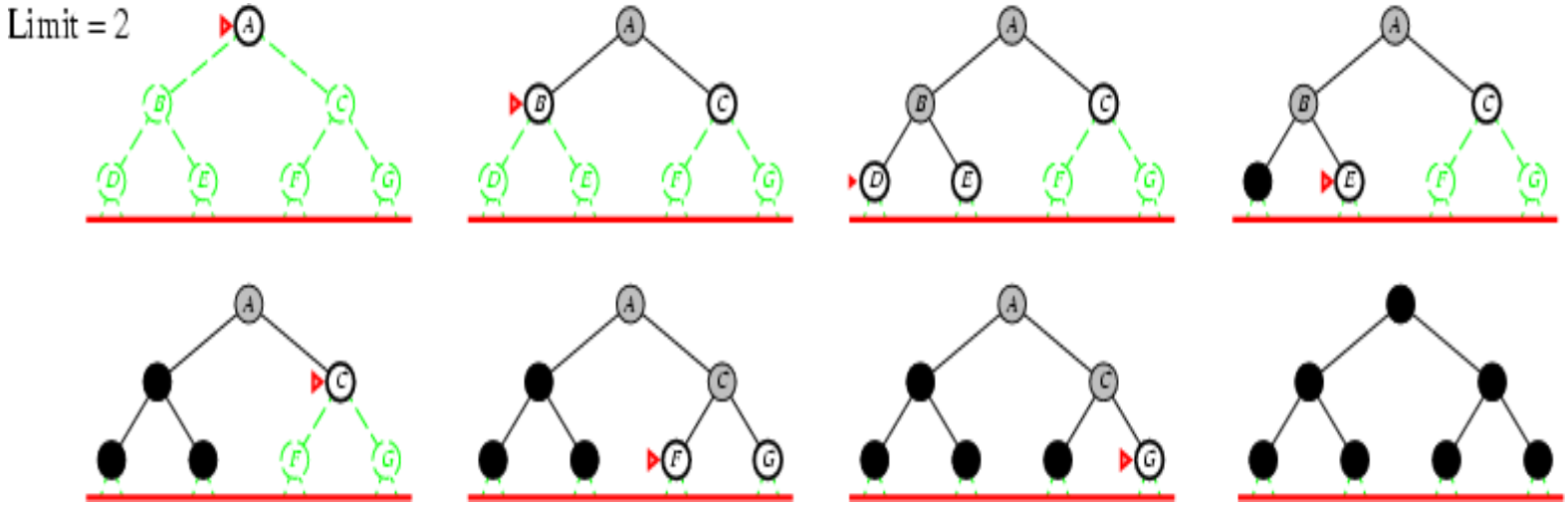
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

L=0

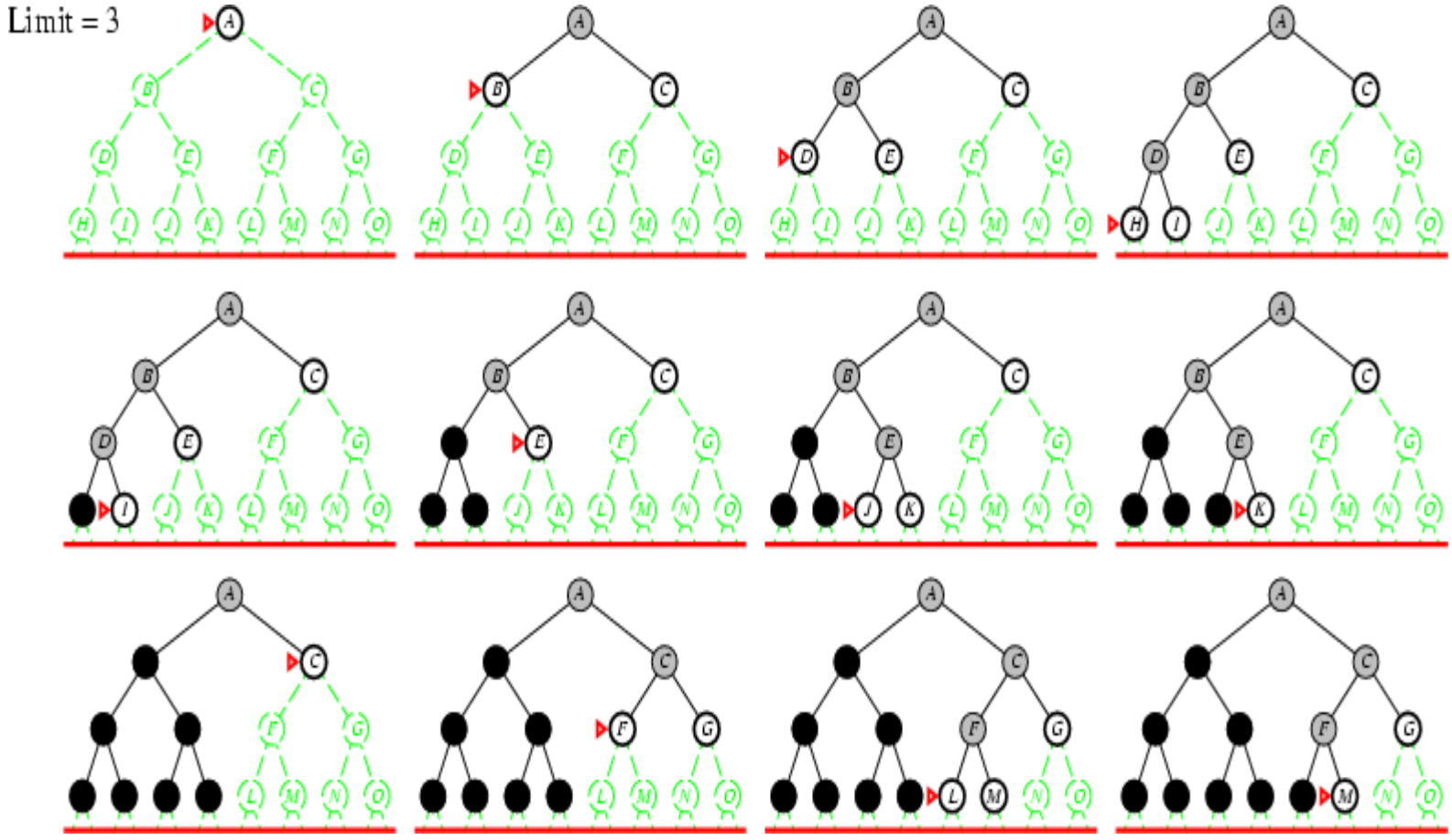
Limit = 0

L=1

L=2



L=3



Recherche en profondeur itérative
Propriétés

- Complète ?
- Optimale ?
- Temps ?
- Espace ?

Propriétés de la recherche Itérative en profondeur

- Complète ?
  - Oui (si facteur de branchement est fini)
- Temps ?
  - $(d)\ b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$
- Espace ?
  - $O(bd)$
- Optimal ?
  - Oui, si le coût de chemin = fonction non décroissante de la profondeur du nœud

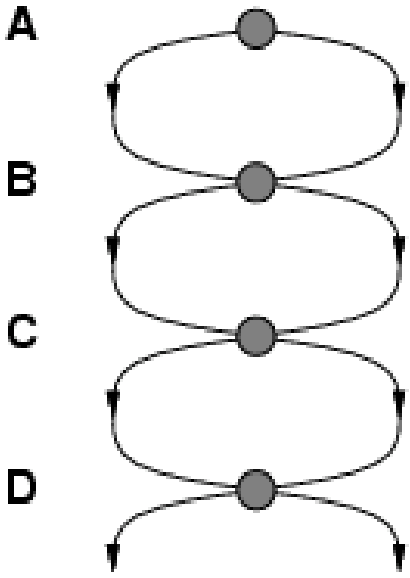
Résumé

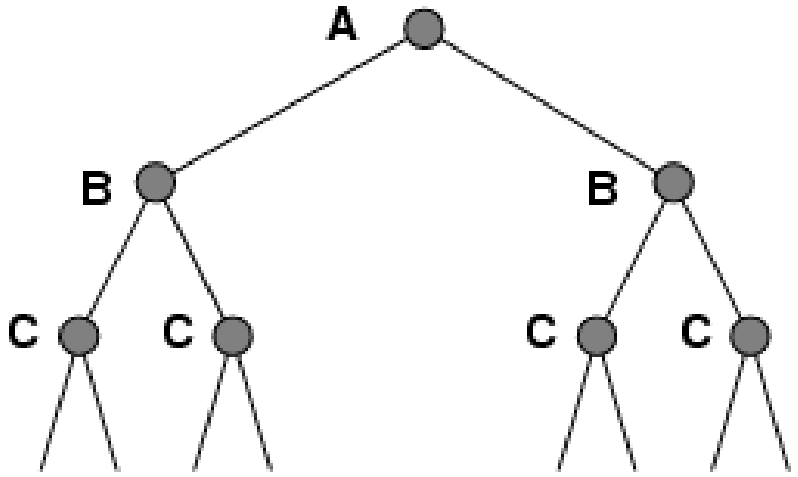
Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	No	No	Yes

b: facteur de branchement de l'arbre de recherche
d: profondeur de la solution la moins couteuse
m: prof. max de l'espace

États répétitifs

- Si ces états ne sont pas détectés la recherche peut devenir exponentielle => une recherche théoriquement possible peut devenir techniquement impossible!





- **Solution**: Garder les nœuds déjà visités dans une liste appelée histoire

Implémentation : Ingrédients

- Nous aurons besoin des fonctions suivantes :
  - `legal(state)` : vrai si un état est légal, faux sinon
  - `final(state)` : vrai si un état est final, faux sinon
  - `applicableOperators(state)` : renvoie la liste des opérateurs applicables dans l'état donné
  - `apply(state, op)` : renvoie l'état obtenu par après application de l'opérateur dans l'état donné
- Il faut aussi maintenir deux listes :
  - `frontiere` : contient les prochains états à explorer
    - Ceci suffirait dans le cas d'un arbre. . .
  - `histoire` : contient les états déjà explorés
    - Pour éviter de visiter un état déjà traité.

State

```
class State :
    def legal ( self ) :
        ...
    def final ( self ) :
        ...
    def applicableOperators ( self ) :
        ...
    def apply ( self , op ) :
        newState = ...

        # Si on veut se souvenir du chemin :
        newState.parent = self
        newState.op = op
        return newState
```

Search

```
def search ( init ) :
    frontiere = [init]
    history = []
    while frontiere:
        etat = frontiere.pop()
        history.append(etat)
        if etat.final():
            return etat
        ops = etat.applicableOps()
        for op in ops:
            new = etat.apply(op)
            if (new not in frontiere) \
            and (new not in history) \
            and new.legal():
                insert(frontiere, new) # <-- !!!
    return "Pas de solution"
```

Cas particuliers

- Suivant le type de stockage dans `frontière`, on obtient différents types de recherche
  - Pile parcours en profondeur
  - File parcours en largeur
- On peut aussi insérer les états dans `frontière` dans un ordre déterminé par une valeur : parcours par préférence
  - Si la valeur est le nombre d'opérateurs appliqués depuis l'état initial et qu'on considère les états dans l'ordre croissant de cette valeur, on retrouve le parcours en largeur
  - Le choix de cette valeur peut beaucoup influencer l'efficacité (moyenne) de la recherche.

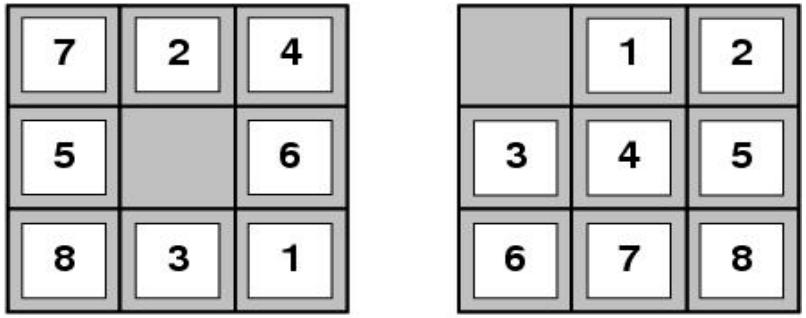
Problème de la recherche aveugle

- Ne garde pas les états du chemin solution au cours de la recherche
- Complexité en espace
  - Dans tous les cas, le nombre de nœuds à mémoriser croît de façon exponentielle (dans le cas de largeur d'abord)
  - Envisager l'élagation de l'espace de recherche par les heuristiques (recherches informées)

Exercice : Le jeu de taquin à 8 pièces

- Plateau de 3X3 cases
- 8 cases sont occupées par une pièce numérotée
- 1 case est vide
- But: Atteindre une configuration donnée à partir d'un état initial

  - Etats légaux??
  - Etat initial??
  - Opérateurs??
  - Etat final??
  - Coût du chemin??



Exercice : Le jeu de taquin à 8 pièces

- Etats légaux: L'emplacement des 8 pièces et celui de la case vide
- Etat initial: n'importe quel état
- Opérateurs: génère les états pouvant résulter de l'exécution de 4 actions (Déplacement vers Gauche, Droite, Haut ou bas)
- Etat final: l'état correspond à la configuration finale (cases en ordre)
- *Coût du chemin: nombre d'étapes qui composent le chemin (coût de chaque étape = 1)*

Exercice (code): Jeu de Taquin

1. Finir la classe *state_model.py*
2. Créer le script *play.py* pour implémenter les algorithmes (principalement la méthode *search*)
    1. Profondeur d'abord
    2. Largeur d'abord

3. Tester votre solution avec les différents niveaux de difficulté dans les exemples de problèmes donnés dans le fichier *taquin_exemple.py*
4. Utiliser la classe *TaquinViewerHTML* pour visualiser et tracer vos algorithmes