

## INTELLIGENCE ARTIFICIELLE

BASÉ SUR "ARTIFICIAL INTELLIGENCE : A MODERN APPROACH" DE RUSSEL ET NOWIG

ENSISA 2A

---

Jonathan Weber  
Automne 2024

RECHERCHE

---

## 1. Recherche

Problème de Recherche

Recherche de solutions

Recherche non-informées (aveugles)

RECHERCHE

---

PROBLÈME DE RECHERCHE

1. Définir le problème :

1. Définir le problème :
  - Formulation de l'objectif

## 1. Définir le problème :

- ▷ Formulation de l'objectif
- ▷ Formulation du problème

1. Définir le problème :
  - ▷ Formulation de l'objectif
  - ▷ Formulation du problème
2. Résoudre le problème en "deux" étapes :



1. Définir le problème :
  - ▷ Formulation de l'objectif
  - ▷ Formulation du problème
2. Résoudre le problème en "deux" étapes :
  - ▷ Recherche : exploration des différentes possibilités

1. Définir le problème :
  - ▷ Formulation de l'objectif
  - ▷ Formulation du problème
2. Résoudre le problème en "deux" étapes :
  - ▷ Recherche : exploration des différentes possibilités
  - ▷ Exécuter la solution trouvée

- ▷ **État initial** : état de départ de l'agent

- ▷ **État initial** : état de départ de l'agent
- ▷ **États** : Ensemble des états atteignables depuis l'état initial par une séquence d'actions (espace des états)

- ▷ **État initial** : état de départ de l'agent
- ▷ **États** : Ensemble des états atteignables depuis l'état initial par une séquence d'actions (espace des états)
- ▷ **Actions** : Ensemble des actions possibles pour l'agent (espace des actions).

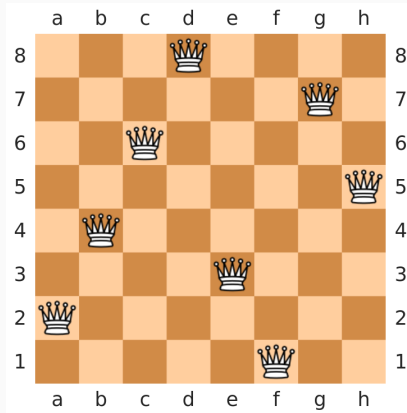
- ▷ **État initial** : état de départ de l'agent
- ▷ **États** : Ensemble des états atteignables depuis l'état initial par une séquence d'actions (espace des états)
- ▷ **Actions** : Ensemble des actions possibles pour l'agent (espace des actions).
- ▷ **Modèle de transition** : description de ce que fait chaque action

- ▷ **État initial** : état de départ de l'agent
- ▷ **États** : Ensemble des états atteignables depuis l'état initial par une séquence d'actions (espace des états)
- ▷ **Actions** : Ensemble des actions possibles pour l'agent (espace des actions).
- ▷ **Modèle de transition** : description de ce que fait chaque action
- ▷ **Test de l'objectif** : détermine si un état est un objectif

- ▷ **État initial** : état de départ de l'agent
- ▷ **États** : Ensemble des états atteignables depuis l'état initial par une séquence d'actions (espace des états)
- ▷ **Actions** : Ensemble des actions possibles pour l'agent (espace des actions).
- ▷ **Modèle de transition** : description de ce que fait chaque action
- ▷ **Test de l'objectif** : détermine si un état est un objectif
- ▷ **Coût du chemin** : fonction qui évalue numériquement le coût d'un chemin (mesure de performance)

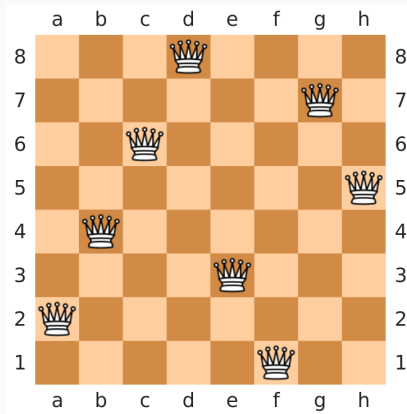


## EXEMPLE : PROBLÈME DES HUIT REINES



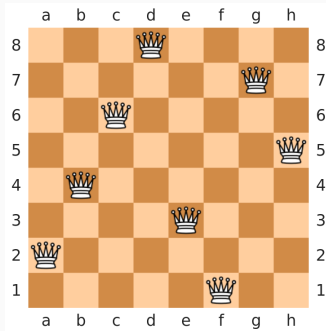
- **Énoncé** : placer huit reines telles qu'aucune ne puisse attaquer une autre reine

## EXEMPLE : PROBLÈME DES HUIT REINES



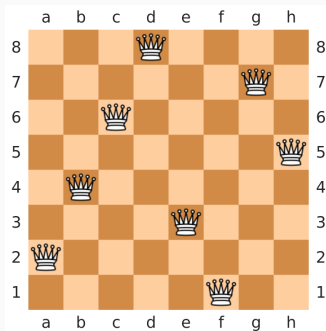
- ▷ **Énoncé** : placer huit reines telles qu'aucune ne puisse attaquer une autre reine
- ▷ **Combinaisons possibles** :  $64 \times 63 \times 62 \times \dots \times 57 \Rightarrow 1,8 \times 10^{14}$

## EXEMPLE : PROBLÈME DES HUIT REINES



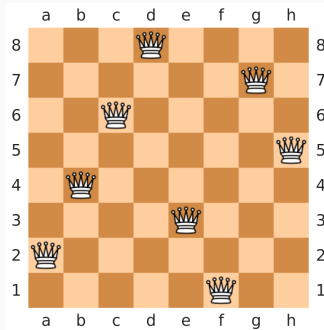
▷ État initial : échiquier vide

## EXEMPLE : PROBLÈME DES HUIT REINES



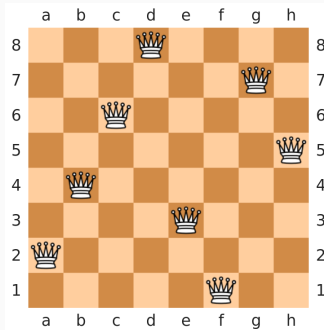
- **État initial** : échiquier vide
- **États** : Toutes les combinaisons de 0 à 8 reines sur l'échiquier

## EXEMPLE : PROBLÈME DES HUIT REINES



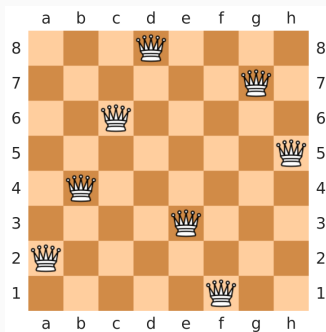
- ▷ **État initial** : échiquier vide
- ▷ **États** : Toutes les combinaisons de 0 à 8 reines sur l'échiquier
- ▷ **Actions** : Ajouter une reine sur une case vide

## EXEMPLE : PROBLÈME DES HUIT REINES



- ▷ **État initial** : échiquier vide
- ▷ **États** : Toutes les combinaisons de 0 à 8 reines sur l'échiquier
- ▷ **Actions** : Ajouter une reine sur une case vide
- ▷ **Modèle de transition** : Mettre à jour l'échiquier

## EXEMPLE : PROBLÈME DES HUIT REINES



- ▷ **État initial** : échiquier vide
- ▷ **États** : Toutes les combinaisons de 0 à 8 reines sur l'échiquier
- ▷ **Actions** : Ajouter une reine sur une case vide
- ▷ **Modèle de transition** : Mettre à jour l'échiquier
- ▷ **Test de l'objectif** : 8 reines sur l'échiquier sans qu'aucune ne soit menacée

## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- **État initial** : N'importe quel état



## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- **État initial** : N'importe quel état
- **États** : Position de chaque tuile dans la grille  $3 \times 3$

## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- **État initial** : N'importe quel état
- **États** : Position de chaque tuile dans la grille  $3 \times 3$
- **Actions** : Gauche, Droite, Haut et Bas

## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- **État initial** : N'importe quel état
- **États** : Position de chaque tuile dans la grille  $3 \times 3$
- **Actions** : Gauche, Droite, Haut et Bas
- **Modèle de transition** : Selon état et action, calculer le nouvel état

## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- ▷ **État initial** : N'importe quel état
- ▷ **États** : Position de chaque tuile dans la grille  $3 \times 3$
- ▷ **Actions** : Gauche, Droite, Haut et Bas
- ▷ **Modèle de transition** : Selon état et action, calculer le nouvel état
- ▷ **Test de l'objectif** : État courant est-il état objectif?

## EXEMPLE : JEU DU TAQUIN

7	2	4
5		6
8	3	1

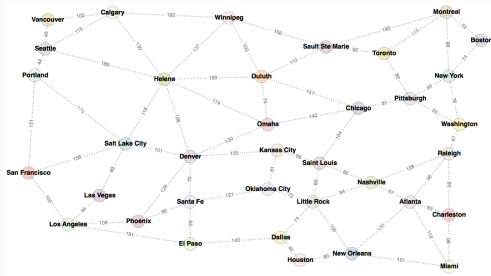
Start State

	1	2
3	4	5
6	7	8

Goal State

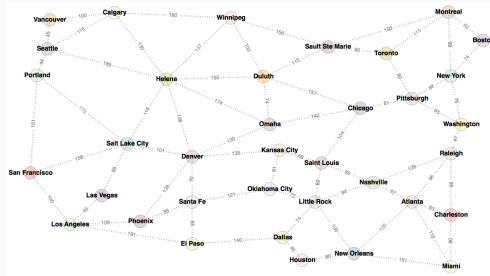
- **État initial** : N'importe quel état
- **États** : Position de chaque tuile dans la grille  $3 \times 3$
- **Actions** : Gauche, Droite, Haut et Bas
- **Modèle de transition** : Selon état et action, calculer le nouvel état
- **Test de l'objectif** : État courant est-il état objectif?
- **Coût du chemin** : Chaque action coûte 1 point

# EXEMPLE : ITINÉRAIRE



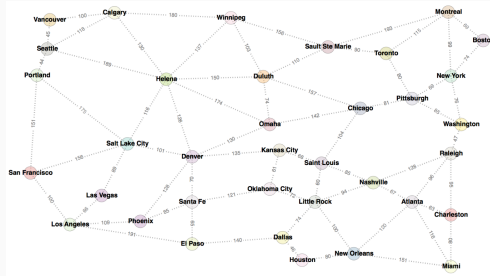
▷ **État initial** : Dans n'importe quelle ville

# EXEMPLE : ITINÉRAIRE



- ▷ **État initial** : Dans n'importe quelle ville
- ▷ **États** : Dans une ville

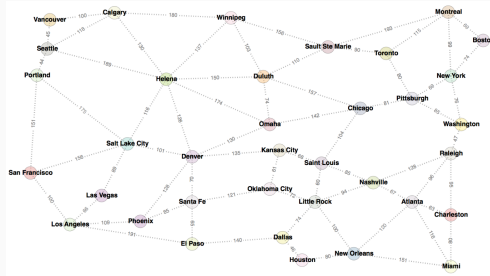
# EXEMPLE : ITINÉRAIRE



- ▷ **État initial** : Dans n'importe quelle **ville**
- ▷ **États** : Dans une **ville**
- ▷ **Actions** : Aller dans une **ville voisine**

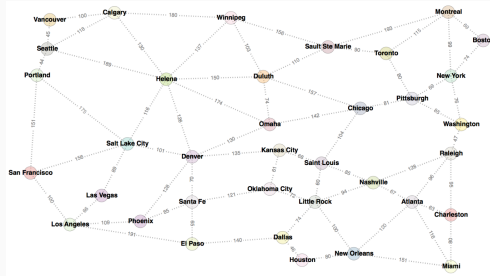


# EXEMPLE : ITINÉRAIRE



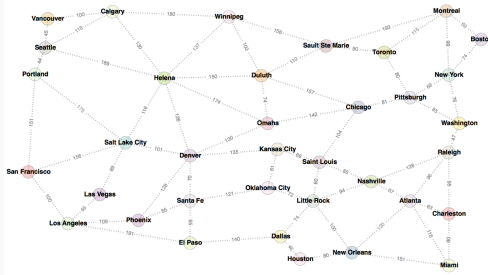
- ▷ **État initial** : Dans n'importe quelle **ville**
- ▷ **États** : Dans une **ville**
- ▷ **Actions** : Aller dans une **ville voisine**
- ▷ **Modèle de transition** : Dans **ville**  $V_1$  + Aller **ville**  $V_2 \Rightarrow$  Dans **ville**  $V_2$

# EXEMPLE : ITINÉRAIRE



- ▷ **État initial** : Dans n'importe quelle **ville**
- ▷ **États** : Dans une **ville**
- ▷ **Actions** : Aller dans une **ville voisine**
- ▷ **Modèle de transition** : Dans **ville**  $V_1$  + Aller **ville**  $V_2 \Rightarrow$  Dans **ville**  $V_2$
- ▷ **Test de l'objectif** : Dans **ville** d'arrivée?

## EXEMPLE : ITINÉRAIRE



- ▷ **État initial** : Dans n'importe quelle **ville**
- ▷ **États** : Dans une **ville**
- ▷ **Actions** : Aller dans une **ville voisine**
- ▷ **Modèle de transition** : Dans **ville**  $V_1$  + Aller **ville**  $V_2 \Rightarrow$  Dans **ville**  $V_2$
- ▷ **Test de l'objectif** : Dans **ville** d'arrivée?
- ▷ **Coût du chemin** : Coût du chemin en kilomètre

- ▷ Recherche de parcours :
  - ▷ Itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...

- ▷ Recherche de parcours :
  - ▷ Itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...
- ▷ Robotique :
  - ▷ Assemblage automatique, navigation autonome, ...

- ▷ Recherche de parcours :
  - ▷ Itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...
- ▷ Robotique :
  - ▷ Assemblage automatique, navigation autonome, ...
- ▷ Planification et ordonnancement
  - ▷ Horaires, organisation de tâches, allocation de ressources, ...

- ▷ Recherche de parcours :
  - ▷ Itinéraires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...
- ▷ Robotique :
  - ▷ Assemblage automatique, navigation autonome, ...
- ▷ Planification et ordonnancement
  - ▷ Horaires, organisation de tâches, allocation de ressources, ...
- ▷ ...

RECHERCHE

---

RECHERCHE DE SOLUTIONS



▷ Idée :

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

- ▷ Idée :
  - ▷ Recherche **hors ligne**, i.e. exploration de l'espace d'états en générant des successeurs d'états déjà générés (développer des états)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

- ▷ Idée :
  - ▷ Recherche **hors ligne**, i.e. exploration de l'espace d'états en générant des successeurs d'états déjà générés (développer des états)
  - ▷ Génération d'un **arbre de recherche** (ou d'un graphe)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

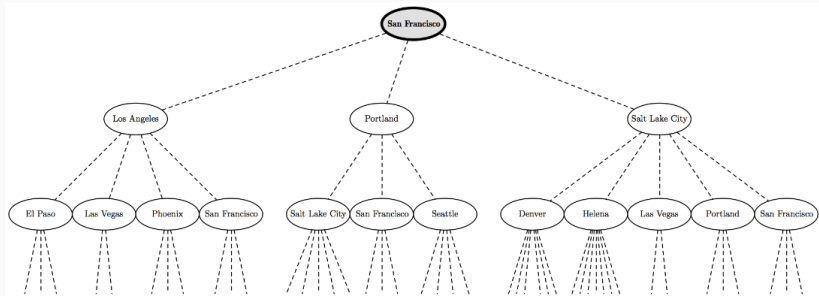
▷ Idée :

- ▷ Recherche **hors ligne**, i.e. exploration de l'espace d'états en générant des successeurs d'états déjà générés (développer des états)
- ▷ Génération d'un **arbre de recherche** (ou d'un graphe)
- ▷ Fin : développement d'un nœud qui est un **état objectif**

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

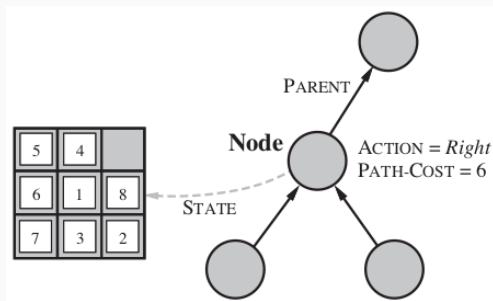
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```





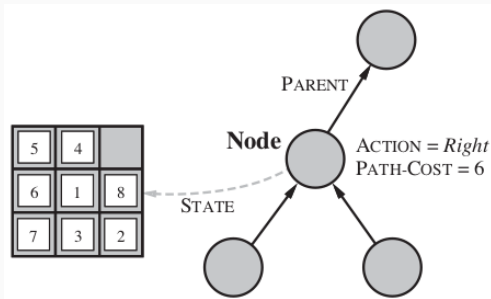


- État : représentation d'une configuration physique du monde

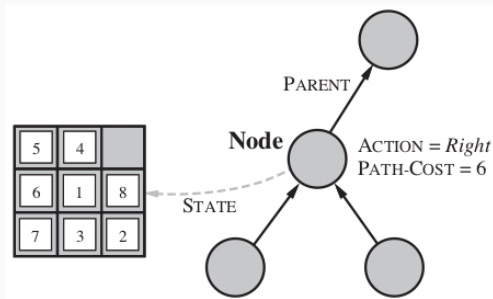




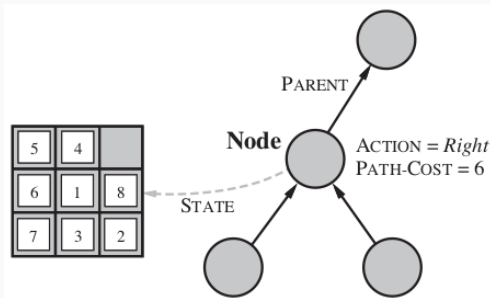
- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :



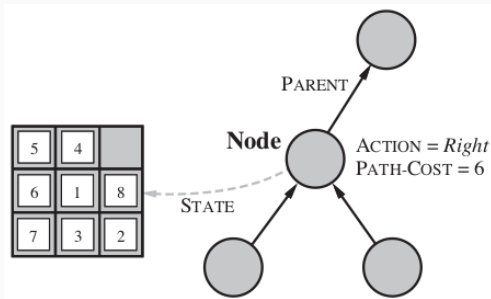
- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :
  - ▷ l'état



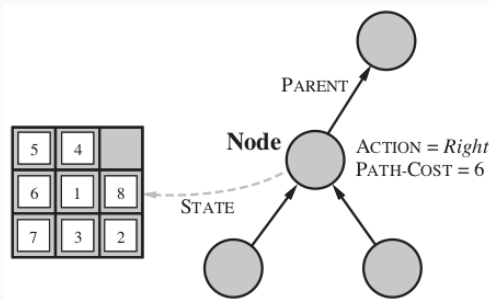
- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :
  - ▷ l'état
  - ▷ le parent, i.e. le nœud père



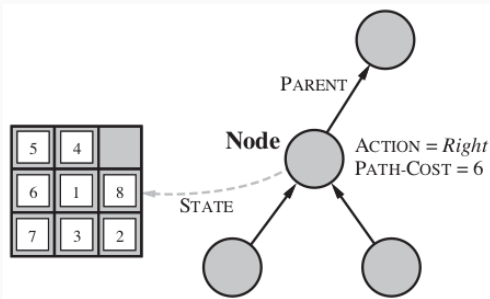
- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :
  - ▷ l'état
  - ▷ le parent, i.e. le nœud père
  - ▷ l'action réalisée pour obtenir l'état contenu dans le nœud



- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :
  - ▷ l'état
  - ▷ le parent, i.e. le nœud père
  - ▷ l'action réalisée pour obtenir l'état contenu dans le nœud
  - ▷ le coût pour atteindre l'état contenu dans le nœud depuis l'état initial



- ▷ État : représentation d'une configuration physique du monde
- ▷ Nœud : structure de données qui est partie intégrante de l'arbre/graphes de recherche et qui inclue :
  - ▷ l'état
  - ▷ le parent, i.e. le nœud père
  - ▷ l'action réalisée pour obtenir l'état contenu dans le nœud
  - ▷ le coût pour atteindre l'état contenu dans le nœud depuis l'état initial
  - ▷ la profondeur du nœud, i.e., la distance entre le nœud et la racine



- Stratégie de recherche = ordre dans lequel les nœuds sont développés

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :



- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- ▷ Complexité en temps et en mémoire se mesure en termes de :

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- ▷ Complexité en temps et en mémoire se mesure en termes de :
  - ▷ **b : facteur maximum de branchement** de l'arbre de recherche, i.e., le nombre maximum de fils des nœuds de l'arbre de recherche

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- ▷ Complexité en temps et en mémoire se mesure en termes de :
  - ▷ **b : facteur maximum de branchement** de l'arbre de recherche, i.e., le nombre maximum de fils des nœuds de l'arbre de recherche
  - ▷ **d : profondeur de la solution la moins coûteuse**

- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- ▷ Complexité en temps et en mémoire se mesure en termes de :
  - ▷ **b** : **facteur maximum de branchement** de l'arbre de recherche, i.e., le nombre maximum de fils des nœuds de l'arbre de recherche
  - ▷ **d** : **profondeur de la solution la moins coûteuse**
  - ▷ **m** : **profondeur maximale de l'arbre de recherche**



- ▷ Stratégie de recherche = ordre dans lequel les nœuds sont développés
- ▷ Une stratégie s'évalue en fonction de 4 dimensions :
  - ▷ **complétude** : est ce que cette stratégie trouve toujours une solution si elle existe ?
  - ▷ **complexité en temps** : le nombre de nœuds créés
  - ▷ **complexité en mémoire** : le nombre maximum de nœuds en mémoire
  - ▷ **optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- ▷ Complexité en temps et en mémoire se mesure en termes de :
  - ▷ **b** : **facteur maximum de branchement** de l'arbre de recherche, i.e., le nombre maximum de fils des nœuds de l'arbre de recherche
  - ▷ **d** : **profondeur de la solution la moins coûteuse**
  - ▷ **m** : **profondeur maximale de l'arbre de recherche**
    - ⚠ peut-être  $\infty$

RECHERCHE

---

RECHERCHE NON-INFORMÉES (AVEUGLES)

- ▷ Aucune connaissance du domaine!

- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :

- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :
  - ▷ Breadth-first search (BFS)
    - ⇒ parcours en largeur

- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :
  - ▷ Breadth-first search (BFS)
    - ⇒ parcours en largeur
  - ▷ Depth-first search (DFS)
    - ⇒ parcours en profondeur

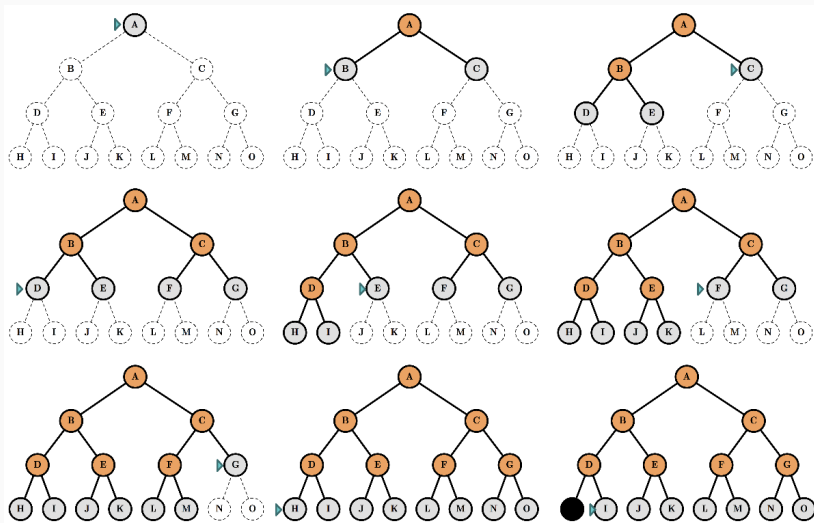
- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :
  - ▷ Breadth-first search (BFS)
    - ⇒ parcours en largeur
  - ▷ Depth-first search (DFS)
    - ⇒ parcours en profondeur
  - ▷ Depth-limited search (DLS)
    - ⇒ parcours en profondeur limitée

- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :
  - ▷ Breadth-first search (BFS)
    - ⇒ parcours en largeur
  - ▷ Depth-first search (DFS)
    - ⇒ parcours en profondeur
  - ▷ Depth-limited search (DLS)
    - ⇒ parcours en profondeur limitée
  - ▷ Iterative-deepening search (IDS)
    - ⇒ parcours itératif en profondeur



- ▷ Aucune connaissance du domaine!
- ▷ Stratégies :
  - ▷ Breadth-first search (BFS)
    - ⇒ parcours en largeur
  - ▷ Depth-first search (DFS)
    - ⇒ parcours en profondeur
  - ▷ Depth-limited search (DLS)
    - ⇒ parcours en profondeur limitée
  - ▷ Iterative-deepening search (IDS)
    - ⇒ parcours itératif en profondeur
  - ▷ Uniform-cost search (UCS)
    - ⇒ parcours à coût uniforme

# BREADTH-FIRST SEARCH (BFS)



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure  
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  frontier  $\leftarrow$  a FIFO queue with node as the only element  
  explored  $\leftarrow$  an empty set  
  loop do  
    if EMPTY?(frontier) then return failure  
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */  
    add node.STATE to explored  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      if child.STATE is not in explored or frontier then  
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)  
        frontier  $\leftarrow$  INSERT(child, frontier)
```

- ▷ **complétude** : Oui (si b est fini)

- ▷ complétude : Oui (si  $b$  est fini)
- ▷ complexité en temps :  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$

## BREADTH-FIRST SEARCH (BFS)

- ▷ complétude : Oui (si b est fini)
- ▷ complexité en temps :  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$
- ▷ complexité en mémoire :  $O(b^d)$

- ▷ **complétude** : Oui (si b est fini)
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$
- ▷ **complexité en mémoire** :  $O(b^d)$
- ▷ **optimalité** : Oui (si coût=1 pour toutes les actions)

## BREADTH-FIRST SEARCH (BFS)

- ▷ **complétude** : Oui (si b est fini)
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$
- ▷ **complexité en mémoire** :  $O(b^d)$
- ▷ **optimalité** : Oui (si coût=1 pour toutes les actions)

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

b=10; 1 million nœuds/seconde; 1 000 octets par nœuds



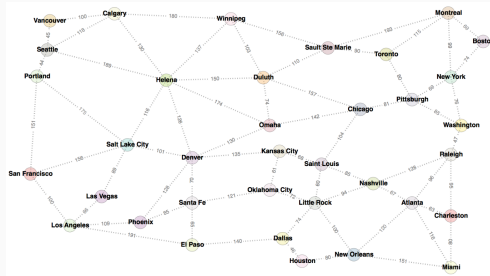
- ▷ **complétude** : Oui (si b est fini)
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$
- ▷ **complexité en mémoire** :  $O(b^d)$
- ▷ **optimalité** : Oui (si coût=1 pour toutes les actions)

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

b=10; 1 million nœuds/seconde; 1 000 octets par nœuds

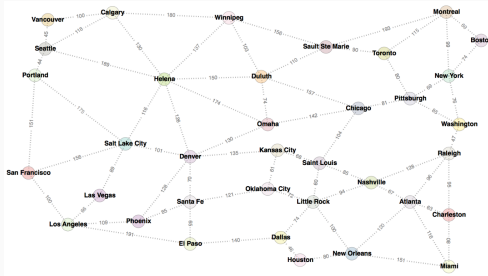
⚠ complexité en temps et en mémoire beaucoup trop élevées!

# UNIFORM-COST SEARCH (UCS)



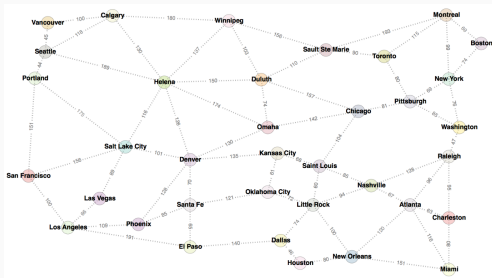
- Les arcs de l'arbre/graphes de recherche peuvent avoir des coûts différents

## UNIFORM-COST SEARCH (UCS)



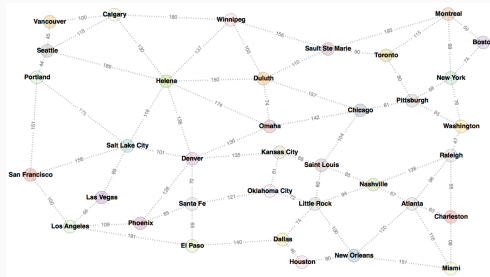
- ▷ Les arcs de l'arbre/graphe de recherche peuvent avoir des coûts différents
- ▷ BFS va trouver le plus court mais il peut être coûteux

# UNIFORM-COST SEARCH (UCS)



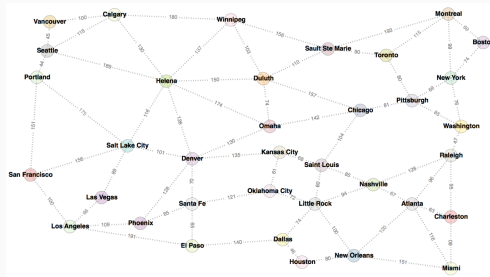
- Les arcs de l'arbre/graphes de recherche peuvent avoir des coûts différents
- BFS va trouver le plus court mais il peut être coûteux
- Nous voulons le moins coûteux :

# UNIFORM-COST SEARCH (UCS)



- Les arcs de l'arbre/graphes de recherche peuvent avoir des coûts différents
- BFS va trouver le plus court mais il peut être coûteux
- Nous voulons le moins coûteux :
  - ⇒ Modification de BFS : On étend le nœud avec le coût de chemin le plus faible

# UNIFORM-COST SEARCH (UCS)



- Les arcs de l'arbre/graphe de recherche peuvent avoir des coûts différents
- BFS va trouver le plus court mais il peut être coûteux
- Nous voulons le moins coûteux :
  - ⇒ Modification de BFS : On étend le nœud avec le coût de chemin le plus faible
  - Exploration par coût croissant

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

- ▷ **complétude** : Oui (si la solution a un coût fini)



- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :

- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale

- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale
  - ▷ Chaque action coûte au moins  $\epsilon$  (coût minimum)

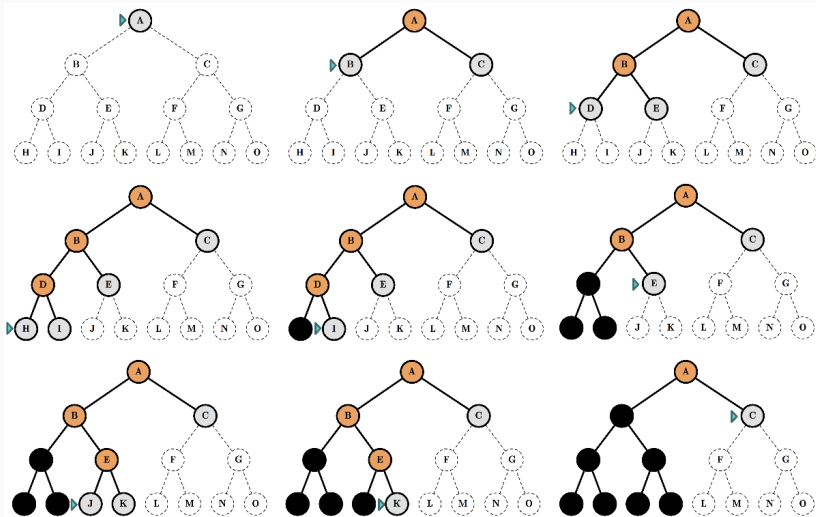
- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale
  - ▷ Chaque action coûte au moins  $\epsilon$  (coût minimum)
  - ▷ La profondeur maximale de la solution est au pire  $C^*/\epsilon$

- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale
  - ▷ Chaque action coûte au moins  $\epsilon$  (coût minimum)
  - ▷ La profondeur maximale de la solution est au pire  $C^*/\epsilon$
  - ▷  $O(b^{\frac{C^*}{\epsilon}})$

- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale
  - ▷ Chaque action coûte au moins  $\epsilon$  (coût minimum)
  - ▷ La profondeur maximale de la solution est au pire  $C^*/\epsilon$
  - ▷  $O(b^{\frac{C^*}{\epsilon}})$
- ▷ **complexité en mémoire** :  $O(b^{\frac{C^*}{\epsilon}})$

- ▷ **complétude** : Oui (si la solution a un coût fini)
- ▷ **complexité en temps** :
  - ▷ Soit  $C^*$  : coût de la solution optimale
  - ▷ Chaque action coûte au moins  $\epsilon$  (coût minimum)
  - ▷ La profondeur maximale de la solution est au pire  $C^*/\epsilon$
  - ▷  $O(b^{\frac{C^*}{\epsilon}})$
- ▷ **complexité en mémoire** :  $O(b^{\frac{C^*}{\epsilon}})$
- ▷ **optimalité** : Oui

# DEPTH-FIRST SEARCH (DFS)





▷ complétude : Non

- ▷ complétude : Non
  - ▷ Problème dans les espaces de profondeur infini

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^m = O(b^m)$

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^m = O(b^m)$ 
  - ▷ Mauvais si  $m > d$

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^m = O(b^m)$ 
  - ▷ Mauvais si  $m > d$
- ▷ **complexité en mémoire** :  $O(bm)$  complexité mémoire linéaire

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^m = O(b^m)$ 
  - ▷ Mauvais si  $m > d$
- ▷ **complexité en mémoire** :  $O(bm)$  complexité mémoire linéaire
  - ▷ Stocke seulement un chemin depuis la racine (+les nœuds non développés)

- ▷ **complétude** : Non
  - ▷ Problème dans les espaces de profondeur infini
  - ▷ Problème dans les espaces avec boucles
- ▷ **complexité en temps** :  $1 + b + b^2 + b^3 + \dots + b^m = O(b^m)$ 
  - ▷ Mauvais si  $m > d$
- ▷ **complexité en mémoire** :  $O(bm)$  complexité mémoire linéaire
  - ▷ Stocke seulement un chemin depuis la racine (+les nœuds non développés)
- ▷ **optimalité** : Non



- ▷ DFS avec une profondeur maximum  $l$

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff  
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  else if limit = 0 then return cutoff  
  else  
    cutoff_occurred?  $\leftarrow$  false  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)  
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true  
      else if result  $\neq$  failure then return result  
  if cutoff_occurred? then return cutoff else return failure
```

- ▷ DFS avec une profondeur maximum  $l$ 
  - ▷ Nœuds à la profondeur  $l$  n'ont pas de successeur

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff  
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  else if limit = 0 then return cutoff  
  else  
    cutoff_occurred?  $\leftarrow$  false  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)  
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true  
      else if result  $\neq$  failure then return result  
  if cutoff_occurred? then return cutoff else return failure
```

- ▷ **complétude** : Seulement si  $l \geq d$

- ▷ complétude : Seulement si  $l \geq d$
- ▷ complexité en temps :  $O(b^l)$

- ▷ complétude : Seulement si  $l \geq d$
- ▷ complexité en temps :  $O(b^l)$
- ▷ complexité en mémoire :  $O(bl)$

- ▷ complétude : Seulement si  $l \geq d$
- ▷ complexité en temps :  $O(b^l)$
- ▷ complexité en mémoire :  $O(bl)$
- ▷ optimalité : Non

- ▷ Profondeur limitée, mais en essayant toutes les profondeurs : 0, 1, 2, 3, .  
..

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

- ▷ Profondeur limitée, mais en essayant toutes les profondeurs : 0, 1, 2, 3, .  
..
- ▷ Évite le problème de trouver une limite pour la recherche profondeur limitée

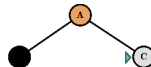
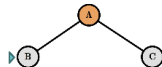
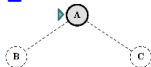
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```



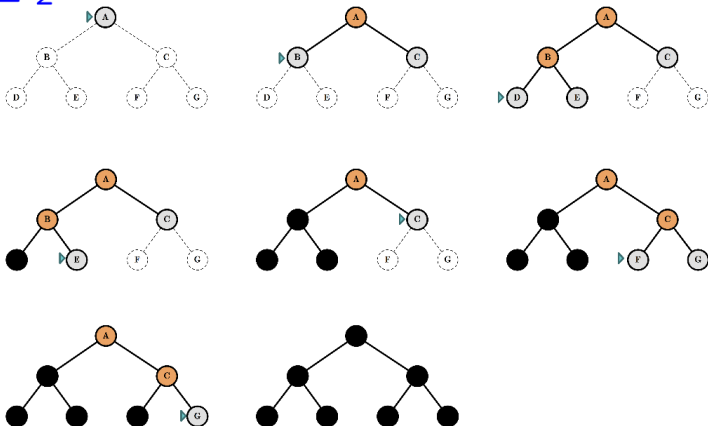
- ▷ Profondeur limitée, mais en essayant toutes les profondeurs : 0, 1, 2, 3, . . .
- ▷ Évite le problème de trouver une limite pour la recherche profondeur limitée
- ▷ Combine les avantages du parcours en largeur d'abord (complète et optimale), mais a la complexité du parcours en profondeur d'abord

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

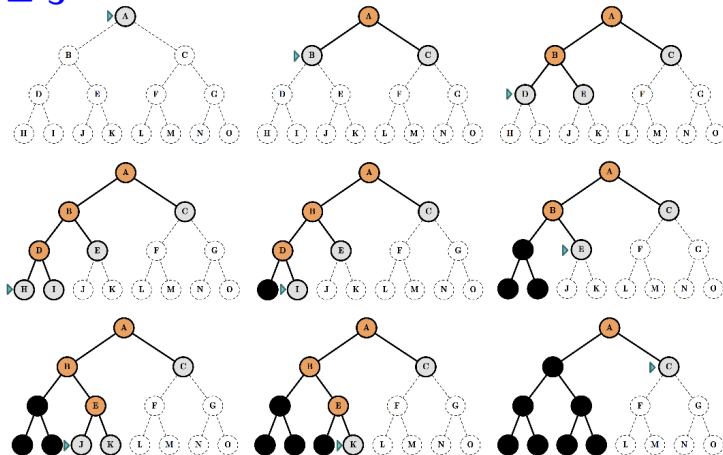
Limit = 1



Limit = 2



Limit = 3



▷ complétude : Oui

- ▷ complétude : Oui
- ▷ complexité en temps :  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

- ▷ complétude : Oui
- ▷ complexité en temps :  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- ▷ complexité en mémoire :  $O(bd)$

- ▷ complétude : Oui
- ▷ complexité en temps :  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- ▷ complexité en mémoire :  $O(bd)$
- ▷ optimalité : Oui (si coût=1 pour toutes les actions)



- ▷ **complétude** : Oui
- ▷ **complexité en temps** :  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- ▷ **complexité en mémoire** :  $O(bd)$
- ▷ **optimalité** : Oui (si coût=1 pour toutes les actions)
- ⚠ impression de gaspillage car de nombreux nœuds seront développés plusieurs fois

- ▷ complétude : Oui
- ▷ complexité en temps :  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- ▷ complexité en mémoire :  $O(bd)$
- ▷ optimalité : Oui (si coût=1 pour toutes les actions)
- ⚠ impression de gaspillage car de nombreux nœuds seront développés plusieurs fois
  - ⇒ Nœuds proches de la racine donc coût "faibles"