Intelligence Artificielle
Recherche

Bruno Bouzy

http://web.mi.parisdescartes.fr/~bouzy
bruno.bouzy@parisdescartes.fr

Licence 3 Informatique

UFR Mathématiques et Informatique
Université Paris Descartes

Algorithmes de recherche en IA

- Agents avec objectifs explicites
- Les différents types de problème
- Exemples de problèmes
- Algorithme de recherche de base (recherche aveugle)

Algorithmes de recherche en IA

- Agents avec objectifs explicites
- Les différents types de problème
- Exemples de problèmes
- Algorithme de recherche de base (recherche aveugle)

Agent avec des objectifs explicites

```
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 problème: le voyage en Roumanie

Oradea
71
151
Zerind
75
Arad
140
Sibiu
99
Fagaras
80
118
Timisoara
111
Rimnicu Vilcea
Lugoj
97
70
146
Mehadia
75
Dobreta
120
Craiova
138
Pitesti
101
Bucharest
211
90
Giurgiu
Neamt
87
Las
92
Vaslui
142
Urziceni
85
98
Hirsova
80
Eforie

Algorithmes de recherche en IA

- Agents avec objectifs explicites
- Les différents types de problème
- Exemples de problèmes
- Algorithme de recherche de base (recherche aveugle)

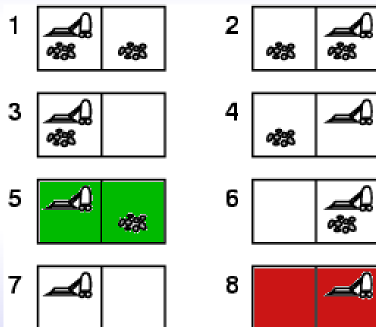
Les différents types de problèmes

- Déterministe, complètement observable $\rightarrow$ problème à un seul état
  - L'agent sait exactement dans quel état il est et dans quel état il sera
  - La solution est une séquence d'actions
- Non-observable $\rightarrow$ problème sans possibilité de percevoir l'environnement
  - L'agent n'a aucune idée d'où il est réellement
  - La solution est une séquence d'actions
- Non-déterministe ou partiellement observable $\rightarrow$ problème dans lesquels il faut gérer des éventualités
  - Les perceptions fournissent de nouvelles informations sur l'état courant
  - Souvent les phases de recherche et d'exécution sont entrelacées
- L'expace d'états est inconnu $\rightarrow$ problème d'exploration

Exemple : Le monde de l'aspirateur

Problème déterministe complètement observable

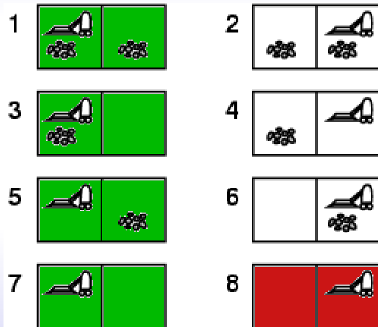
- Etat initial : #5
- Etat final : #8
- Solution : $\langle$*droite*, *aspire*$\rangle$



Exemple : Le monde de l'aspirateur

Problème déterministe non observable

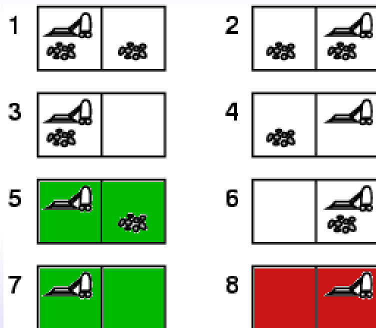
- Etats initiaux : {#1, #2, #3, #4, #5, #6, #7, #8}
- Solution pour {#1, #3, #5, #7} : ⟨*aspire*, *droite*, *aspire*⟩
- Solution pour {#2, #4 #6, #8} : ⟨*gauche*, *aspire*, *droite*, *aspire*⟩



Exemple : Le monde de l'aspirateur

Problème non déterministe partiellement observable

- Non-déterministe : aspirer ne garantit pas que le sol soit propre
- Partiellement observable : on ne sait pas si le sol à droite est propre
  $\Rightarrow$ Etats initiaux : $\{\#5, \#7\}$
- Solution :
  $\langle$*droite*, si sol sale alors *aspire*$\rangle$



Les problèmes déterministes et complètement observables

- Un problème déterministe et complètement observable est défini par :
  1. un **état initial**
     - par exemple, “à Arad”
  2. un **ensemble d'actions** ou une **fonction de transition**, $succ(x)$:
     - par exemple, $succ(Arad) = \{Zerind, Timisoara, Sibiu\}$
  3. un **test de terminaison** pour savoir si le but est atteind
     - explicite, e.g., “à Bucharest”
     - implicite, e.g., “vérifier mat au échec”
  4. un **coût** (additif)
     - par exemple la somme des distances, le nombre d'actions exécutées, etc.
     - par exemple, $c(x, a, y)$ est le coût d'une transition, $c(x, a, y) \geq 0$
- Une solution est une séquence d'actions partant de l'état initial et menant au but.

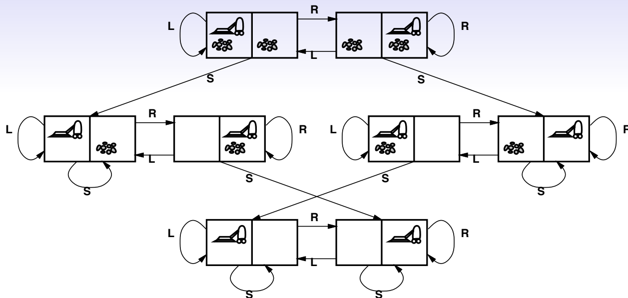
L'espace d'états

- Le **monde réel est trop complexe** pour être modélisé
  - L'espace de recherche modélise une vue abstraite et simplifiée du monde réel
- Un **état abstrait** représente un ensemble d'états réels
- Une **action abstraite** représente une combinaison complexe d'actions réelles
  - par exemple, "Arad → Zerind" représente un ensemble de routes possibles, de détours, d'arrêts, etc.
  - Une action abstraite doit être une simplification par rapport à une action réelle
- Une **solution abstraite** correspond à un ensemble de chemins qui sont solutions dans le monde réel.

Algorithmes de recherche en IA

- Agents avec objectifs explicites
- Les différents types de problème
- Exemples de problèmes
- Algorithme de recherche de base (recherche aveugle)

Exemple : Espace d'états du monde de l'aspirateur



- **Etats?** sol sale ou propre, position de l'aspirateur
- **Actions?** droite, gauche, aspire
- **Test du but?** les deux pièces doivent être propres
- **Coût du chemin?** 1 par action

Exemple : Le jeu du taquin

7	2	4
5		6
8	3	1

Etat initial

	1	2
3	4	5
6	7	8

Etat final

- **Etats?** les positions des pièces
- **Actions?** déplacement droite, gauche, haut, bas
- **Test du but?** état but donné
- **Coût du chemin?** 1 par déplacement

Exemple : Le robot assembleur

- **Etats?** coordonnées du robot, angles, position de l'objet à assembler...
- **Actions?** déplacements continus
- **Test du but?** objet complètement assemblé
- **Coût du chemin?** le temps d'assemblage

Exemple de problèmes réels

- 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, ...

Algorithmes de recherche en IA

- Agents avec objectifs explicites
- Les différents types de problème
- Exemples de problèmes
- Algorithme de recherche de base (recherche aveugle)

Algorithme de recherche de base (recherche aveugle)

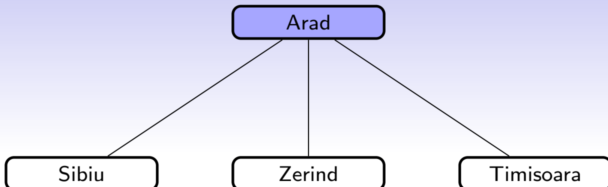
- Idée de base
  - Recherche hors ligne, i.e. exploration de l'espace d'états en générant des successeurs d'états déjà générés (**développer des états**)
  - Génération d'un arbre de recherche
- On s'arrête lorsqu'on a choisi de développer un nœud qui est un état final

```
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 : arbre de recherche

Arad

Exemple : arbre de recherche



Exemple : arbre de recherche

Arad
- Sibiu
  - Arad
  - Fagaras
  - Oradea
  - Rimnicu Vilcea
- Zerind
- Timisoara

Implémetation des algorithmes de recherche

- Structure de données **nœud** qui contient
  - état
  - parent
  - enfant
  - profondeur
  - coût du chemin (noté $g(x)$)
- `Expand` crée de nouveaux nœuds
- `InsertAll` insère de nouveaux nœuds dans la liste à traiter

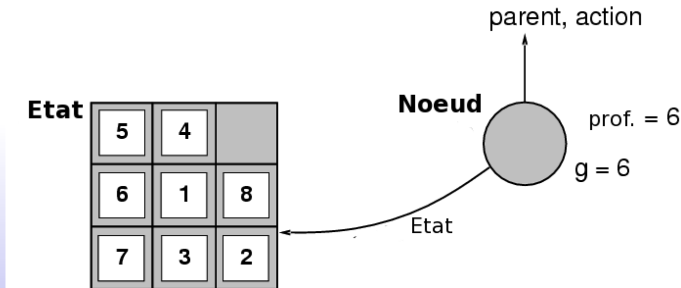
Algorithme de recherche dans les arbres

```
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
```

Etats vs Nœuds

- Un état est une représentation d'une configuration physique du monde
- Un nœud est une structure de données qui est partie intégrante de l'arbre 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 $g(x)$ 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 de l'arbre



Stratégie de recherche

- Les **différents attributs des nœuds** sont initialisés par la fonction `Expand`
- Une **stratégie de recherche** est définie par l'ordre dans lequel les nœuds sont développés, i.e., la fonction `Insert-Fn`
- Une stratégie s'évalue en fonction de 4 dimensions :
    - la **complétude** : est ce que cette stratégie trouve toujours une solution si elle exise ?
    - la **complexité en temps** : le nombre de nœuds créés
    - la **complexité en mémoire** : le nombre maximum de nœuds en mémoire
    - l'**optimalité** : est ce que la stratégie trouve toujours la solution la moins coûteuse ?

Stratégie de recherche

- La complexité en temps et en mémoire se mesure en termes de :
  - b : le **facteur maximum de branchement** de l'arbre de recherche, i.e., le nombre maximum de fils des nœuds de l'arbre de recherche
  - d : la **profondeur de la solution la moins coûteuse**
  - m : la **profondeur maximale de l'arbre de recherche**

    $\rightarrow$ attention peut être ∞

Stratégies de recherche non-informées (aveugles)

- Les stratégies de recherche non-informées utilisent seulement les informations disponibles dans le problème
- Il existe plusieurs stratégies :
  - Recherche en largeur d'abord
  - Recherche en coût uniforme
  - Recherche en profondeur d'abord
  - Recherche en profondeur limitée
  - Recherche iterative en profondeur

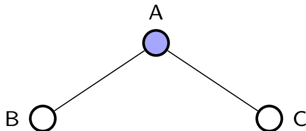
Recherche en largeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en fin de liste
- *fringe* $= [A]$

A

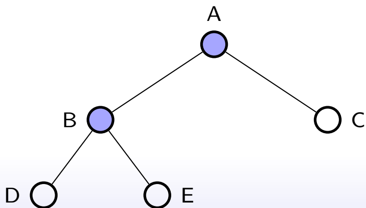
Recherche en largeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en fin de liste
- *fringe* $= [B, C]$



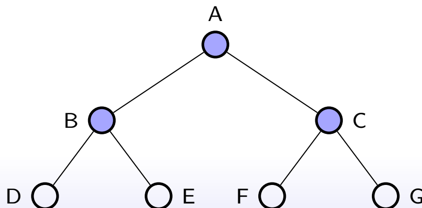
Recherche en largeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en fin de liste
- $fringe = [C, D, E]$



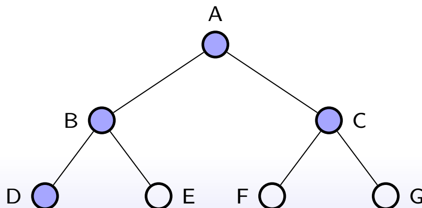
Recherche en largeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en fin de liste
- $fringe = [D, E, F, G]$



Recherche en largeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en fin de liste
- $fringe = [E, F, G]$



Exemple de problème: le voyage en Roumanie

Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Timisoara
Rimnicu Vilcea
111
Lugoj
97
211
70
146
Pitesti
101
Bucharest
Mehadia
75
138
90
Dobreta
120
Craiova
Giurgiu
Neamt
87
Las
92
Vaslui
142
Urziceni
85
98
Hirsova
80
Eforie

Recherche en largeur d'abord

- Complet, si b est fini
- Complexité en temps :

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

- Complexité en espace : $O(b^{d+1})$ (garde tous les nœuds en mémoire)
- Optimale si coût $= 1$ pour chaque pas, mais non optimale dans le cas général

$\Rightarrow$ L'espace est le plus gros problème

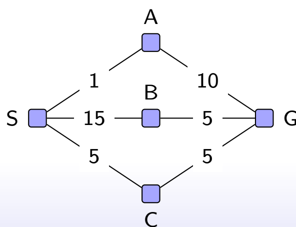
Recherche en largeur d'abord

- $b = 10$
- 10000 nœuds par seconde
- 1000 octets de mémoire pour un nœuds

Profondeur	Nœuds	Temps	Mémoire
8	10^9	31 heures	1 Teraoctet, 1024 Go
12	10^{13}	35 ans	10 Petaoctets, 1024 Teraoctet

Recherche en coût uniforme

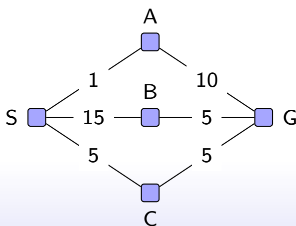
- La fonction `Insert-Fn` ajoute les nœuds dans l'ordre du coût $g(x)$
- $fringe = [(S, 0)]$

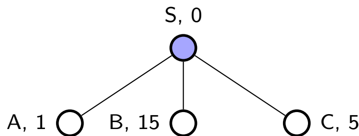


S, 0

Recherche en coût uniforme

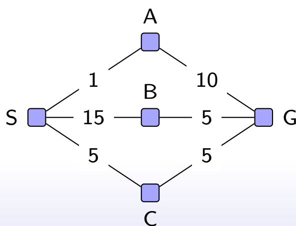
- La fonction `Insert-Fn` ajoute les nœuds dans l'ordre du coût $g(x)$
- $fringe = [(A, 1), (C, 5), (B, 15)]$

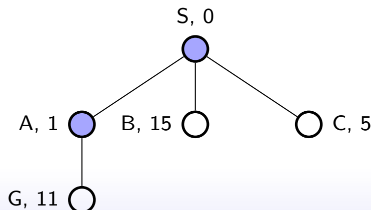




Recherche en coût uniforme

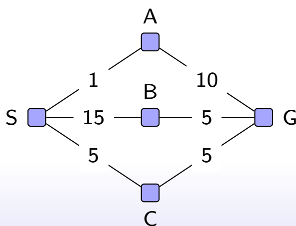
- La fonction `Insert-Fn` ajoute les nœuds dans l'ordre du coût $g(x)$
- $fringe = [(C, 5), (G, 11), (B, 15)]$

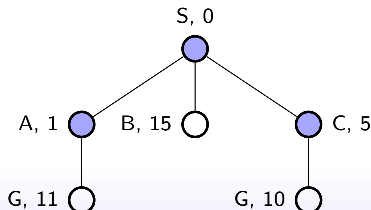




Recherche en coût uniforme

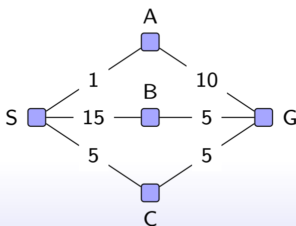
- La fonction `Insert-Fn` ajoute les nœuds dans l'ordre du coût $g(x)$
- $fringe = [(G, 10), (G, 11), (B, 15)]$

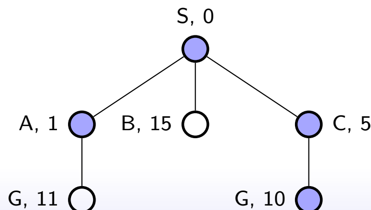




Recherche en coût uniforme

- La fonction `Insert-Fn` ajoute les nœuds dans l'ordre du coût $g(x)$
- $fringe = [(G, 11), (B, 15)]$





Recherche en coût uniforme

- Equivalent à la largeur d'abord si le coût est toujours le même
- Complet si le coût de chaque pas est strictement supérieur à 0
- Complexité en temps : nombre de nœuds pour lesquels $g \leq C^*$, où C^* est le coût de la solution optimale
- Complexité en espace : idem que la complexité en temps
- Optimale car les nœuds sont développés en fonction de g

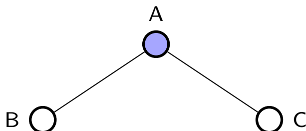
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [A]$

A

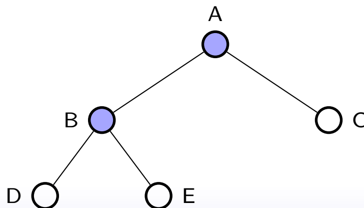
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [B, C]$



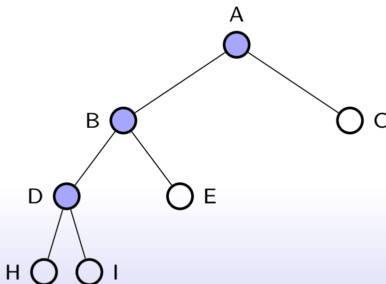
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [D, E, C]$



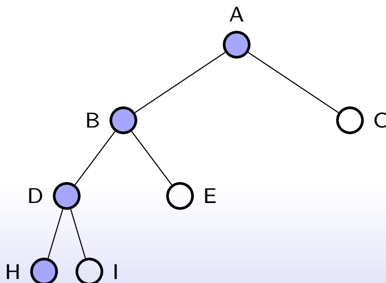
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [H, I, E, C]$



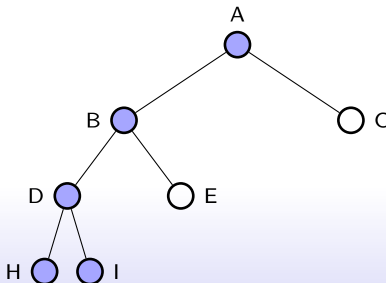
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [I, E, C]$



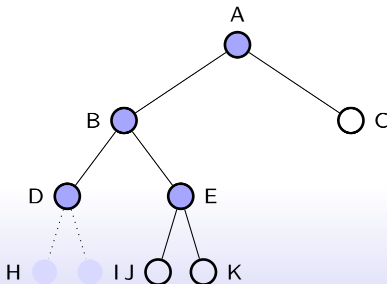
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [E, C]$



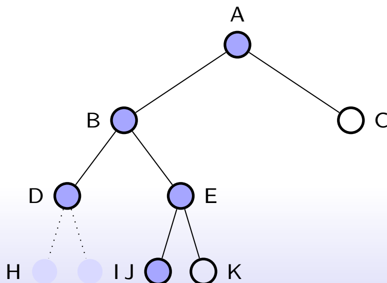
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [J, K, C]$



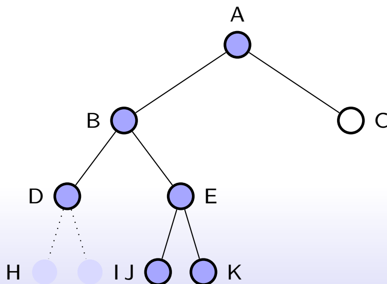
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [K, C]$



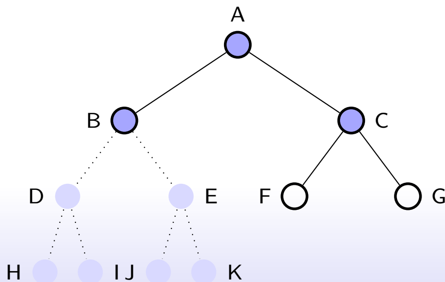
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [C]$



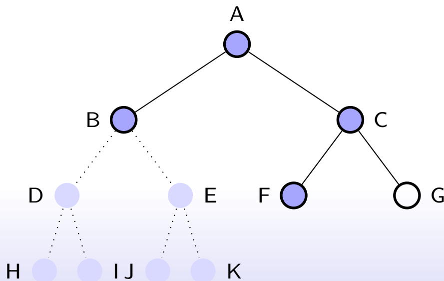
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [F, G]$



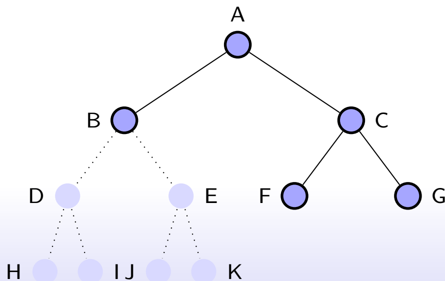
Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- $fringe = [G]$



Recherche en profondeur d'abord

- La fonction `Insert-Fn` ajoute les successeurs en début de liste
- *fringe* = []



Recherche en profondeur d'abord

- Non complet dans les espaces d'états infinis ou avec boucle
  - Il est possible d'ajouter un test pour détecter les répétitions
- Complexité en temps : $O(b^m)$
  - Très mauvais si m est beaucoup plus grand que b
- Complexité en espace : $O(bm)$
  - Linéaire!
- Non optimale

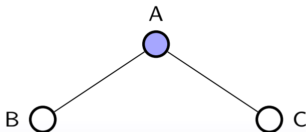
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$

A

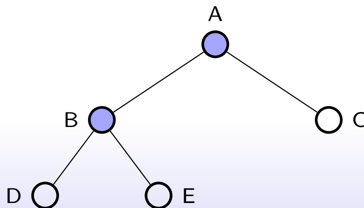
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



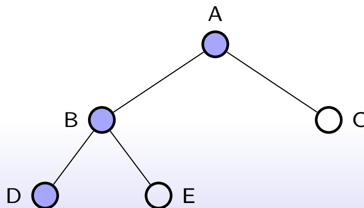
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



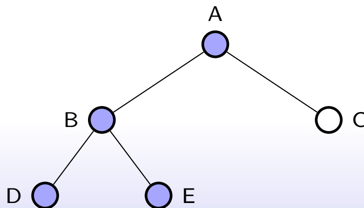
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



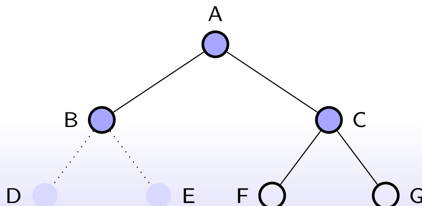
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



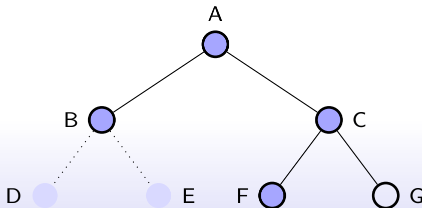
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



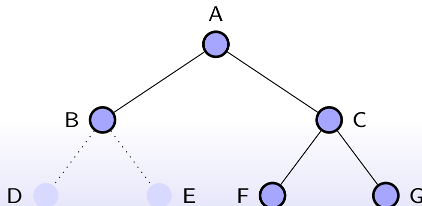
Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



Recherche en profondeur limitée

- Algorithme de recherche en profondeur d'abord, mais avec une limite l sur la profondeur
  - Les nœuds de profondeur l n'ont pas de successeurs
- Exemple pour $l = 2$



Recherche en profondeur limitée

- Complet si $l \geq d$
- Complexité en temps : $O(b^l)$
- Complexité en espace : $O(bl)$
- Non optimale

Recherche en profondeur limitée

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

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

Recherche en profondeur itérative

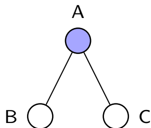
- Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, ...
- Evite le problème de trouver une limite pour la recherche profondeur limitée
- Combine les avantages de largeur d'abord (complète et optimale), mais a la complexité en espace de profondeur d'abord

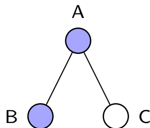
Recherche en profondeur itérative

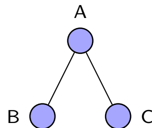
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Recherche en profondeur itérative: $l = 0$

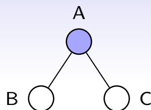
Recherche en profondeur itérative: $l = 1$

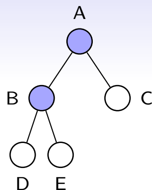


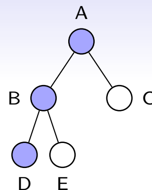


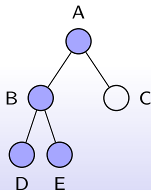


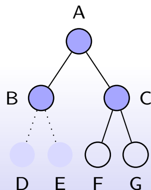
Recherche en profondeur itérative: $l = 2$

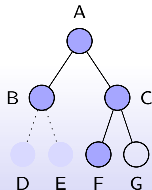


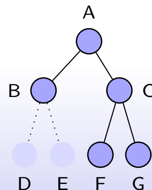












Recherche en profondeur itérative

- Peut paraître du gaspillage car beaucoup de nœuds sont étendus de multiples fois
- Mais la plupart des nouveaux nœuds étant au niveau le plus bas, ce n'est pas important d'étendre plusieurs fois les nœuds des niveaux supérieurs
- Complet
- Complexité en temps :

$$(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$$

- Complexité en espace : $O(bd)$
- Optimale : oui, si le coût de chaque action est de 1. Peut être modifiée pour une stratégie de coût uniforme

Résumé des algorithmes de recherche

Critères	Largeur d'abord	Coût uniforme	Prof. d'abord	Prof. limitée	Prof. itérative
Complétude	Oui	Oui	Non	Oui si $l \geq d$	Oui
Temps	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Espace	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimalité - coût d'une action $= 1$	Oui	Oui	Non	Non	Oui
Optimalité - cas général	Non	Oui	Non	Non	Non