Résoudre des problèmes par la recherche dans l'espace d'états

Chap. 3

1

Plan

- Agents pour résoudre des problèmes
- Types de problème
- Formulation de problème
- Problèmes exemples
- Algorithmes de recherche de base

2

Agents pour résoudre des 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
```

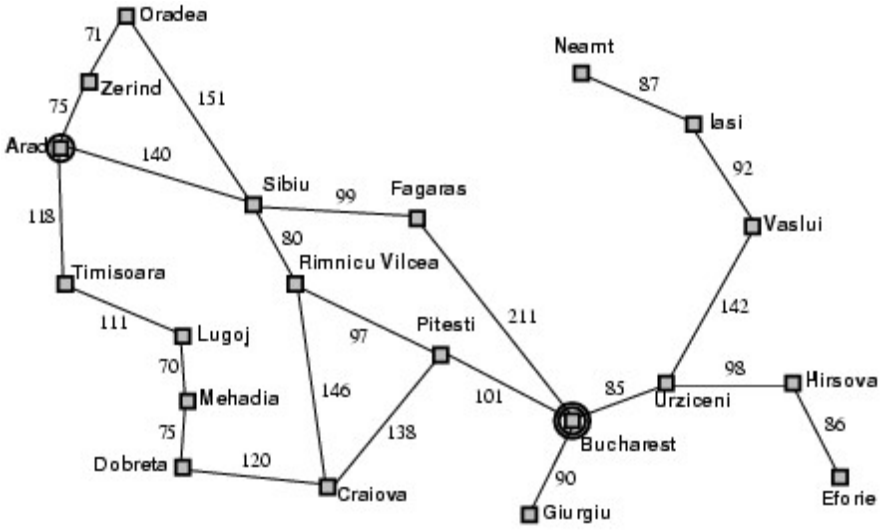
3

Exemple: Voyage en Roumanie

- En vacances en Roumanie; présentement à Arad.
- L'avion part demain de Bucharest
-
- Formuler le but :
  - Être à Bucharest
  -
- Formuler le problème :
  - États: être à différentes villes
  - actions: conduire entre les villes
  -
- Trouver une solution:
  - séquence de villes, e.g., Arad, Sibiu, Fagaras, Bucharest
  -

4

Exemple: Roumanie



5

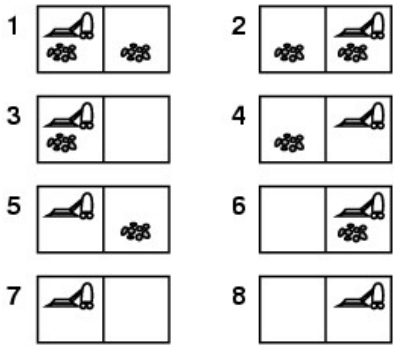
Types de problème

- Déterministe, complètement observable → problème d'un seul état
  - L'agent sait exactement où il va, et la solution est une séquence
- Non-observable → problème sans senseur (problème de conforme)
  - L'agent n'a aucune idée où il se trouve, la solution est une séquence
- Non déterministe et/ou partiellement observable → problème de contingence
  - Les perçus fournissent de nouvelles informations sur l'état courant
  - Souvent mélange la recherche et l'exécution
- Espace d'états inconnu → problème d'exploration

6

Exemple: Monde de l'aspirateur

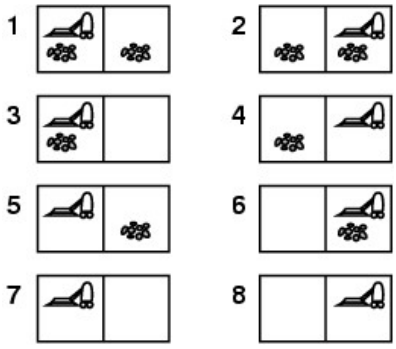
- Seul état, commencer à #5. Solution?
-



7

Exemple: Monde de l'aspirateur

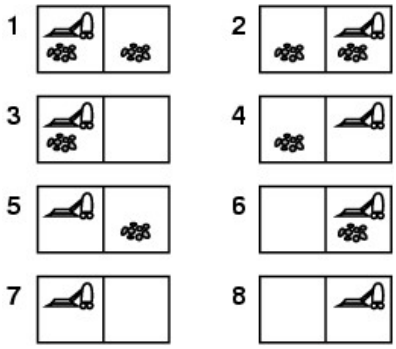
- Seul état, commencer à #5. Solution? *[Right, Suck]*
-



8

Exemple: Monde de l'aspirateur

- Seul état, commencer à #5. Solution? *[Right, Suck]*
-
- Sans senseur, commencer à {*1,2,3,4,5,6,7,8*} e.g., *Right* va à {*2,4,6,8*} Solution?
-

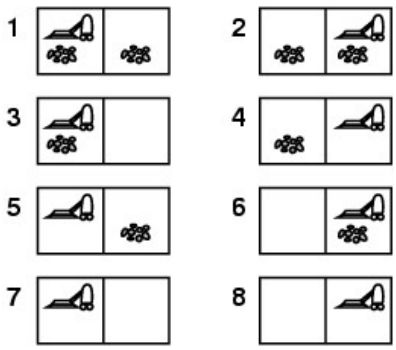


9

Exemple: Monde de l'aspirateur

- Seul état, commencer à #5. Solution? *[Right, Suck]*
-
- Sans senseur, commencer à {*1,2,3,4,5,6,7,8*} e.g., *Right* va à {*2,4,6,8*} Solution?

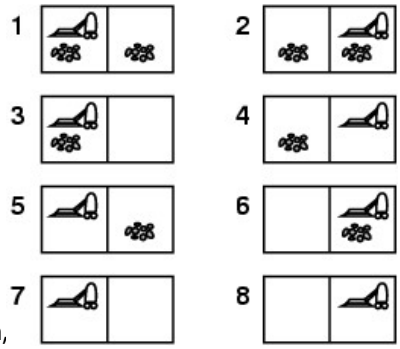
  [Right,Suck,Left,Suck]
-
-



10

Exemple: Monde de l'aspirateur

- Sans senseur, commencer à {*1,2,3,4,5,6,7,8*} e.g., *Right* va à {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]

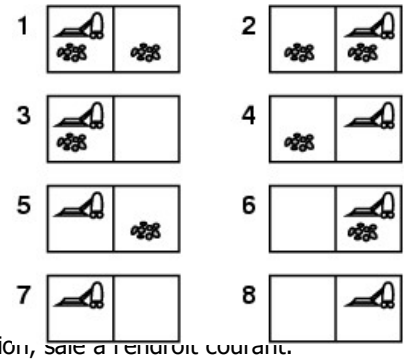


- Contingence
  - Nondéterministe: *Suck* peut salir le tapis propre
  - Partiellement observable: localisation,
  - Perçu: *[L, Clean],* i.e., commencer à #5 ou#7
    Solution?

11

Exemple: Monde de l'aspirateur

- Sans senseur, commencer à {*1,2,3,4,5,6,7,8*} e.g., *Right* va à {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



- Contingence
  - Nondéterministe: *Suck* peut salir le tapis propre
  - Partiellement observable: localisation, saleté à l'endroit courant.
  - Perçu: *[L, Clean],* i.e., commencer à #5 ou#7
    Solution? *[Right, **if** dirt **then** Suck]*

12

Formulation du problème à un seul état

Un problème est défini par 4 éléments:

1. État initial e.g., "être à Arad » (représenté par *Arad*)
2.
2. Actions ou fonction de successeur *S(x)* = ensemble de paires d'action-état
   - e.g., *S(Arad)* = { *<Arad → Zerind, Zerind>, …* }
3. Test de but, peut être
   - explicite, e.g., *x* = "être à Bucharest"
   - implicite, e.g., *Checkmate(x)*
4. Coût de chemin (additive)
   - e.g., somme de distances, nombre d'actions exécutées, etc.
   - *c(x,a,y)* est le coût d'étape, en supposant qu'il est ≥ 0

- Une solution est une séquence d'actions menant de l'état initial à un état but
-

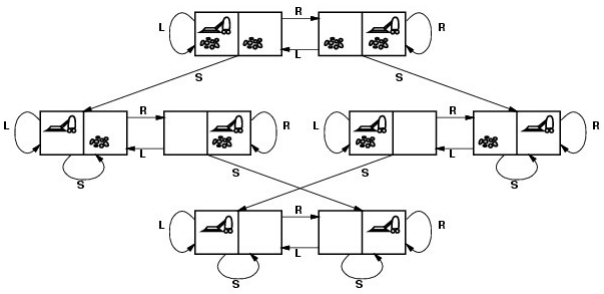
13

Sélectionner un espace d'états

- Le monde réel est trop complexe
  → Un espace d'états doit être plus abstrait
- (Abstrait) état = ensemble d'états réels
-
- (Abstrait) action = combinaison complexe d'actions réelles
  - e.g., "Arad → Zerind" représente un ensemble complexe de route, détours, pauses, etc.
- Pour garantir la faisabilité dans la réalité, chaque état réel "être à Arad" doit pouvoir aller à un certain état réel "être à Zerind"
-
- (Abstrait) solution =
  - Ensemble de chemins réels constituant une solution dans le monde réel
  -
- Chaque action abstraite doit être plus facile que le problème originel

14

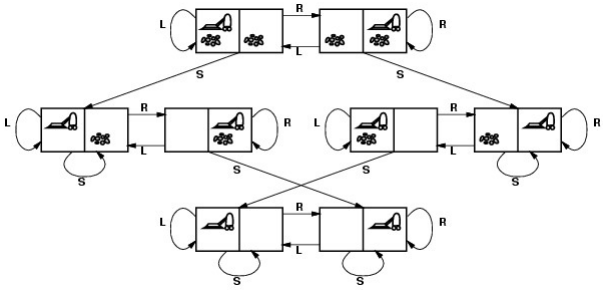
Graphe d'espace d'états



- états?
- actions?
- Test de but?
- Coût de chemin?
-

15

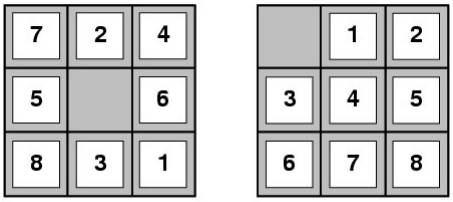
Graphe d'espace d'états



- états? indique la localisation de l'inspirateur et de saleté
- actions? *Left*, *Right*, *Suck*
- test de but? Pas de saleté dans toutes les localisations
- coût de chemin? 1 par action

16

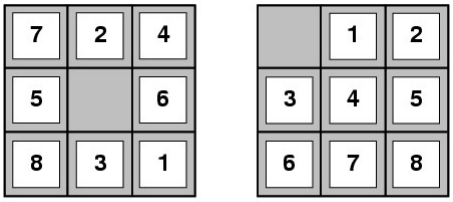
Exemple: The 8-puzzle



- états?
- actions?
- test de but?
- coût de chemin?

17

Exemple: The 8-puzzle

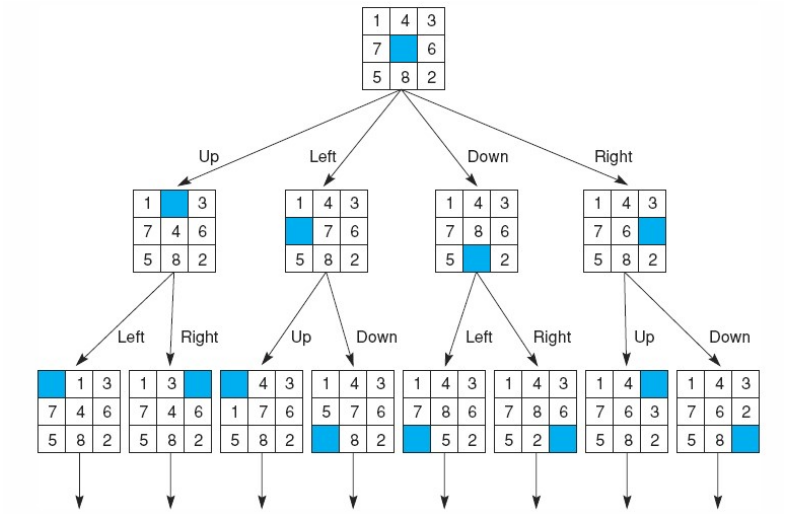


- états? Localisation des tuiles
- actions? Bouger la case blanc vers gauche, droite, haut, bas
- Tests de but? = état de but (déjà donné)
- Coût de chemin? 1 par mouvement

[Note: la solution optimale de famille de *n*-Puzzle est NP-difficile]

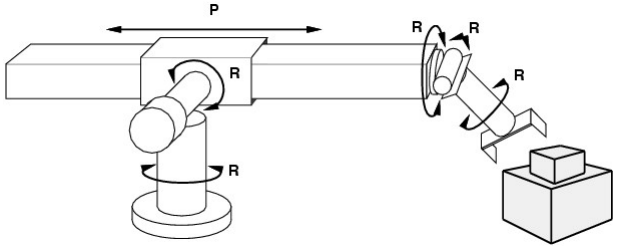
18

Espace pour 8-puzzle
(un point de départ différent)



19

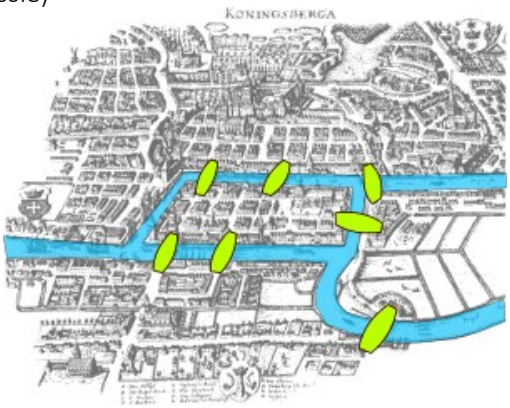
Exemple: Robot assembleur



- états?: coordonnées et angles du robot, et de la pièce à assembler
- actions?: mouvements continus des joints du robot
- tests de but?: pièce assemblée complètement
- coût de chemin?: temps d'exécution
-

20

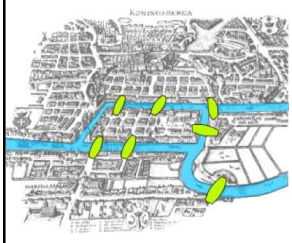
Problème de 7 ponts de Königsberg (Kaliningrad en Russie)

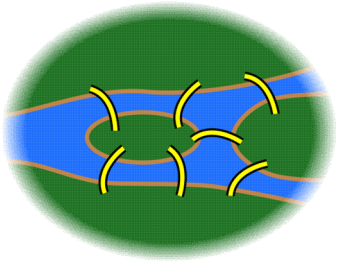


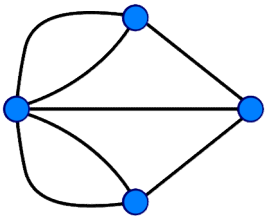
- Faire le tour de la ville en prenant chaque pont une et une seule fois
- En 1735, Euler montra que c'est impossible en utilisant la théorie de graphes et topologie.

21

Abstraction







Solution en théorie de graphes:

Pour que ceci soit possible, chaque noeud doit avoir un degré pair.

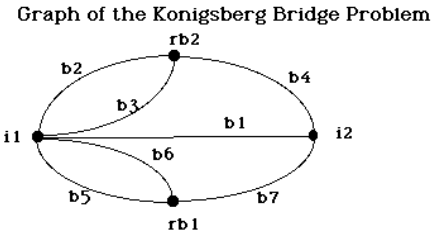
22

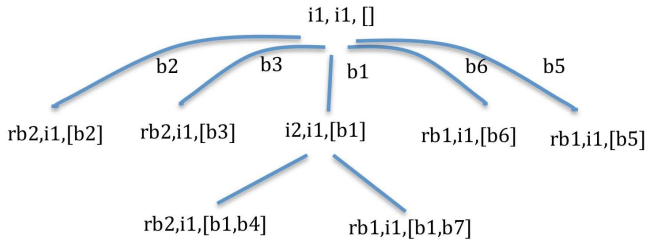
Résoudre le problème avec recherche

- État: définit un état dans le voyage
  - Position courante
  - Point de départ
  - Ponts visités
- But:
  - position courante=point de départ
  - Ponts visités = 7 ponts
- Actions:
  - aller vers le prochain point relié par un pont si le pont n'est pas déjà pris
- Coût:
  - 1 par action (sans importance dans ce problème)

23

Résoudre le problème avec recherche





24

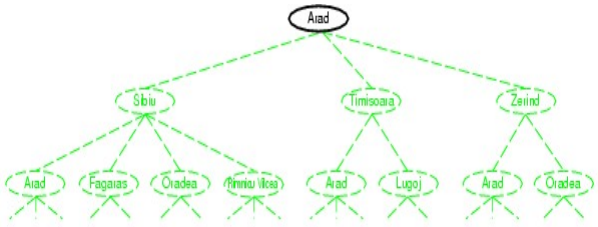
Algorithmes de recherche dans l'arbre

- Idée de base:
  - Exploration simulée dans l'espace d'états en générant les états successeurs des états déjà explorés (~expansion/développement des états)
  -

```
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
```

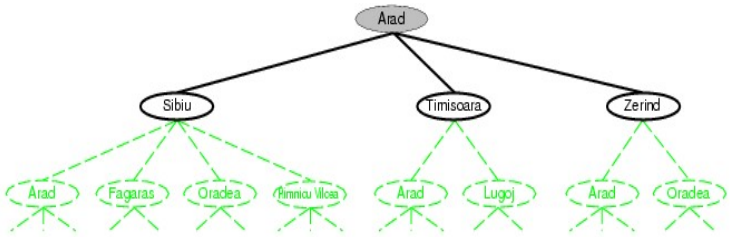
25

Exemple de recherche dans l'arbre



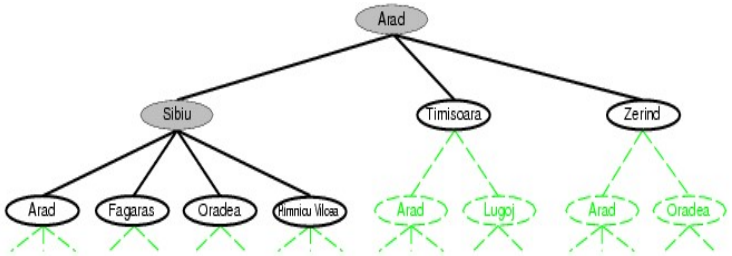
26

Exemple de recherche dans l'arbre



27

Exemple de recherche dans l'arbre



28

Implantation: recherche générale dans l'arbre

```
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
```

29

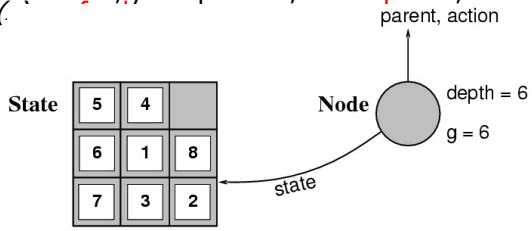
Quelques notions

- Frange (fringe): la frontière de l'exploration = les noeuds candidats à considérer pour la prochaine étape
- Coût de chemin = ancien coût de chemin + nouvelle étape (additif)

30

Implantation: état vs. noeud

- Un état est une représentation d'une configuration physique
- Un noeud est une structure de données qui constitue une partie de l'arbre de recherche, y compris état, noeud parent, action, coût de chemin *g(*



- La fonction `Expand` crée de nouveaux noeuds, remplit des champs et utilise `SuccessorFn` du problème pour créer des états corespondants

31

Stratégies de recherche

- Une stratégie de recherche est définie par l'ordre d'expansions (développements) de noeuds choisi
- Des stratégies sont évaluées selon:
  - Complétude: est-ce qu'elle trouve toujours une solution si existe?
  - Complexité en temps: nombre de noeuds générés
  - Complexité en espace: nombre max. de noeuds en mémoire
  - optimalité: est-ce qu'elle trouve toujours la solution la moins coûteuse (la plus courte)?
  -
- Les complexités en temps et en espace sont mesurées en utilisant:
  - *b:* facteur de branchement max. de l'arbre de recherche
  - *d:* profondeur de la solution la moins coûteuse
  - *m*: profondeur max. de l'arbre (peut être ∞)
  -

32

Stratégies de recherche non informées

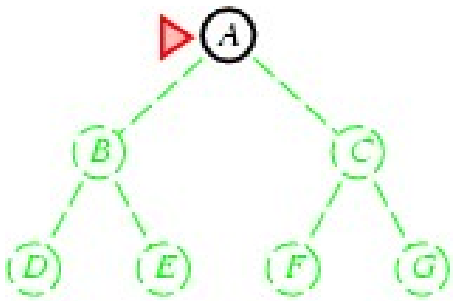
- Stratégies de recherche non informées utilisent seulement des informations disponibles dans la définition du problème
  - Utilisent la topologie de l'arbre
  - N'utilisent pas des informations internes à l'état (qualité de l'état)

- Recherche en largeur d'abord (Breadth-first search)
- Recherche de coût uniforme (Uniform-cost search)
- Recherche en profondeur d'abord (Depth-first search)
- Recherche en profondeur limitée (Depth-limited search)
- Recherche en approfondissement itératif (Iterative deepening search)
-
- Différence: Comment les noeuds sont insérés et ordonnés dans la frange

33

Recherche en largeur d'abord

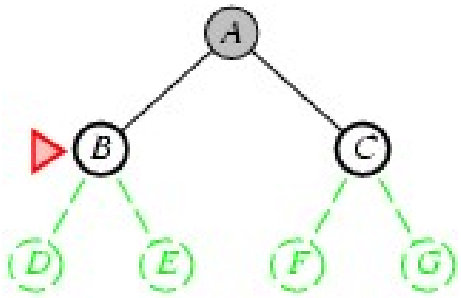
- Expansion du noeud le moins profond parmi les candidats
- Implantation:
  - *Frange* (*Fringe)* est une queue FIFO queue, i.e., nouveaux sucesseurs sont mis à la fin
  -



34

Recherche en largeur d'abord

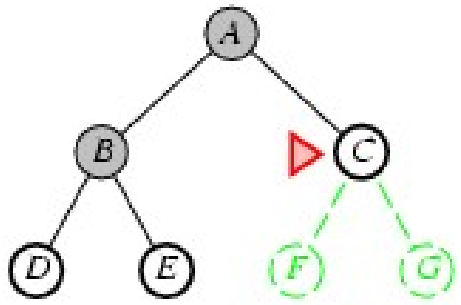
- Expansion du noeud le moins profond parmi les candidats
- Implantation:
  - *Frange* (*Fringe)* est une queue FIFO queue, i.e., nouveaux sucesseurs sont mis à la fin
  -



35

Recherche en largeur d'abord

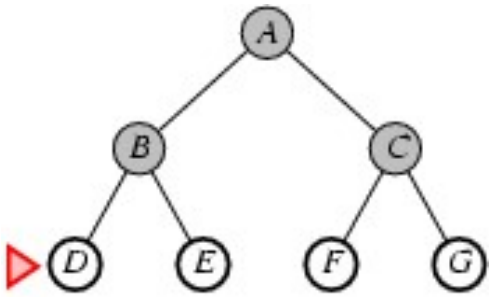
- Expansion du noeud le moins profond parmi les candidats
- Implantation:
  - *Frange* (*Fringe)* est une queue FIFO queue, i.e., nouveaux sucesseurs sont mis à la fin
  -



36

Recherche en largeur d'abord

- Expansion du noeud le moins profond parmi les candidats
- Implantation:
  - *Frange* (*Fringe)* est une queue FIFO queue, i.e., nouveaux sucesseurs sont mis à la fin
  -



37

Propriétés de la recherche en largeur d'abord

- Complète? Oui (si *b* est fini)
-
- Temps? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$
-
- Espace? $O(b^{d+1})$ (garde tous les noeuds en mémoire)
-
- Optimal? Oui (si coût = 1 par étape)
-
- Espace est le plus grand problème (plus que le temps)
-

38

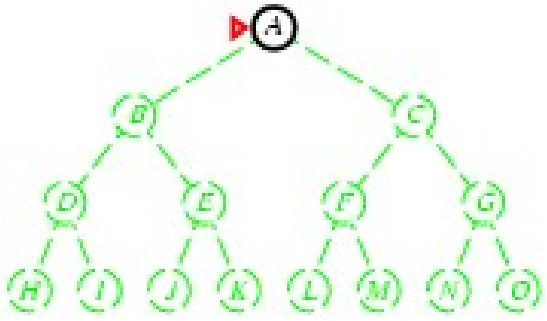
Recherche de coût uniforme

- Développer (*expand*) un noeud non-exploré le moins coûteux
- Implantation:
  - *frange* = queue ordonnée par le coût de chemin
- Equivalent à largeur d'abord si les coûts d'étape sont égaux
- Complète? Oui, si le coût d'étape ≥ ε
- Temps? # de noeuds avec $g \leq$ coût de la solution optimale
  $O(b^{\lceil C^*/\varepsilon \rceil})$ où C^* est le coût de la solution optimale
- Espace? # de noeuds avec $g \leq$ coût de la solution optimale
  $O(b^{\lceil C^*/\varepsilon \rceil})$
- Optimal? Oui – noeuds épandu dans l'ordre croissant de $g(n)$

39

Recherche en profondeur d'abord

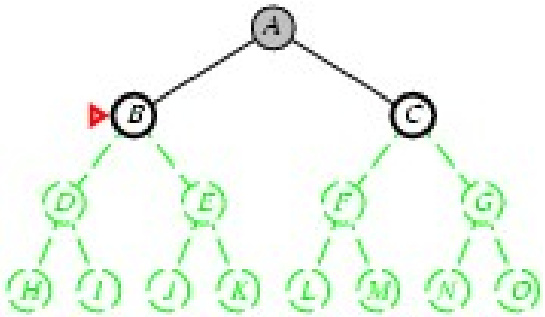
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début



40

Recherche en profondeur d'abord

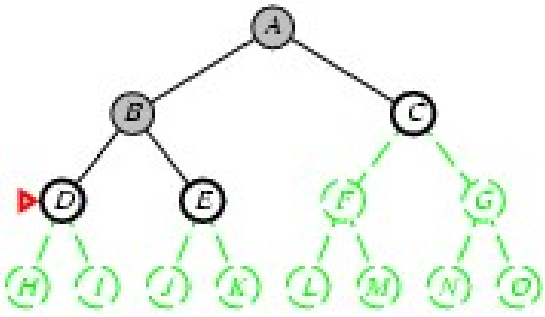
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



41

Recherche en profondeur d'abord

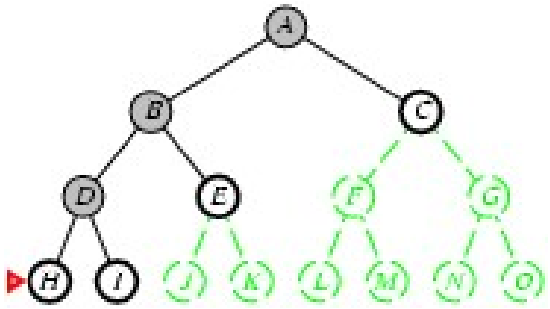
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



42

Recherche en profondeur d'abord

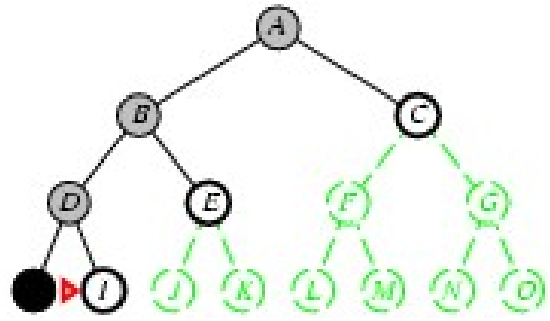
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



43

Recherche en profondeur d'abord

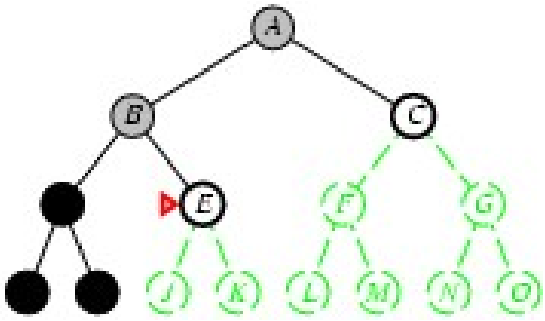
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



44

Recherche en profondeur d'abord

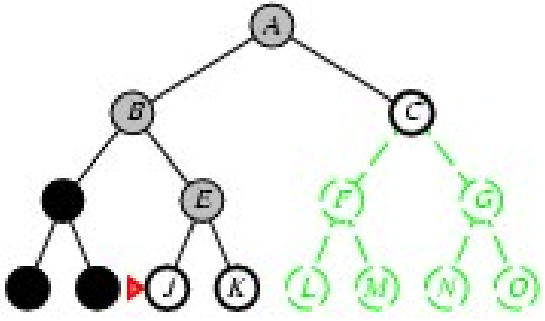
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



45

Recherche en profondeur d'abord

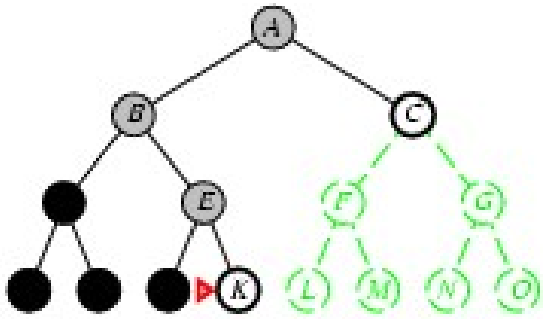
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



46

Recherche en profondeur d'abord

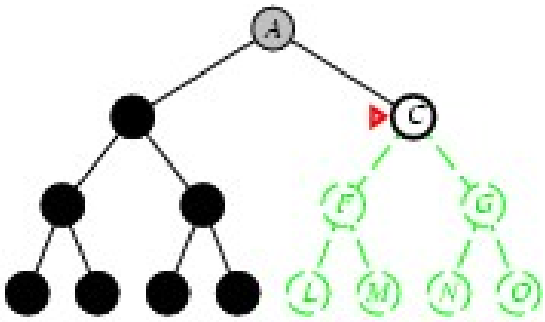
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



47

Recherche en profondeur d'abord

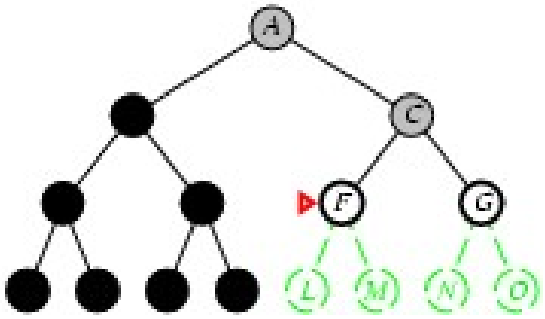
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



48

Recherche en profondeur d'abord

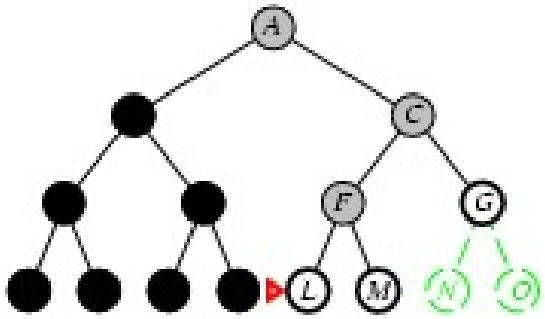
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



49

Recherche en profondeur d'abord

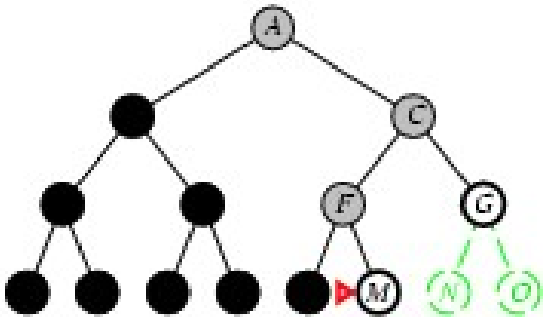
- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



50

Recherche en profondeur d'abord

- Développer le noeud non exploré le plus profond
- Implantation:
  - *frange* = queue LIFO, i.e., mettre les successeurs au début
  -



51

Propriétés de la recherche en prof.

- Complète? Non: échoue dans un espace avec profondeur infinie, avec boucles
  - Modifier pour éviter de répéter les états sur le chemin
    → complete dans un espace fini
- Temps? $O(b^m)$: terrible si m est beaucoup plus grand que d
  - Mais si les solutios sont denses, peut être plus rapide que largeur d'abord
  -
- Espace? $O(bm)$, i.e., espace linéaire !
-
- Optimal? Non
-

52

Recherche avec profondeur limitée

= profondeur d'abord avec la limite de profondeur *l*, i.e., nœuds à la profondeur *l* n'ont pas de successeurs

- Implantation récursive:

```
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
```

53

Recherche en approfondissement itératif

```
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
```

54

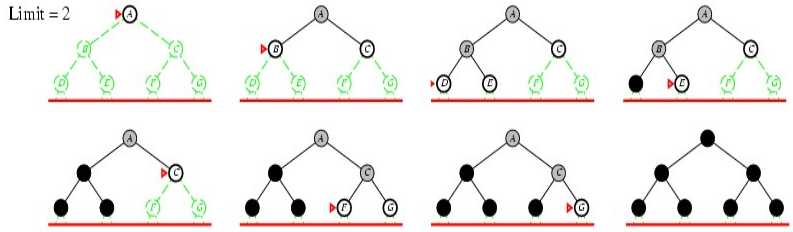
Recherche en approfondissement itératif /= 0

55

Recherche en approfondissement itératif /= 1

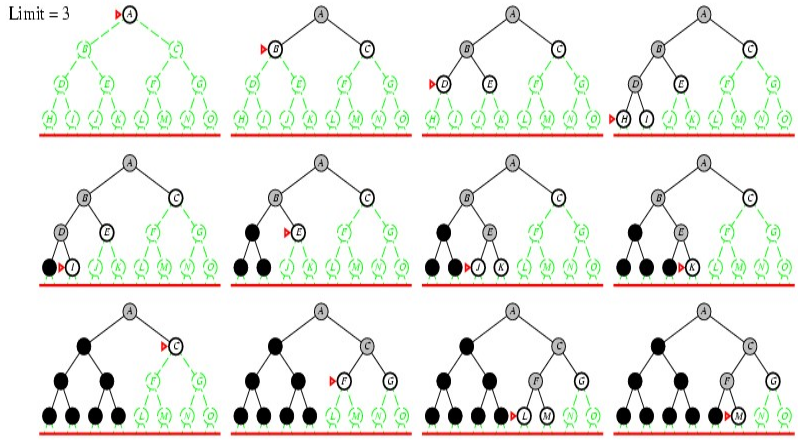
56

Recherche en approfondissement itératif /= 2



57

Recherche en approfondissement itératif /= 3



58

Recherche en approfondissement itératif

- Nombre de noeuds générés à la limite de profondeur *d* avec le facteur de branchement *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Nombre de noeuds générés dans une recherche en approfondisement itératif à la profondeur *d* avec le facteur de branchement *b*:

$$N_{IDS} = (d+1)b^0 + d\ b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- Pour *b = 10*, *d = 5*,
-
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  -
  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
- Overhead = (123,456 - 111,111)/111,111 = 11%

59

Propriétés de la recherche en approfondissement itératif

- Complète? Oui
-
- Temps? $(d+1)b^0 + d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
-
- Espace? $O(bd)$
-
- Optimal? Oui, si le coût d'étape = 1

60

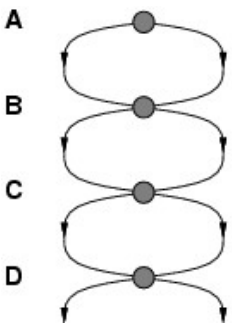
Sommaire des algorithmes

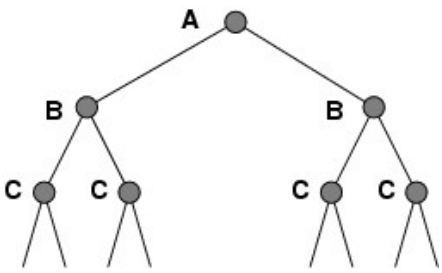
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

61

États répétés

- Échouer dans la détection des états répétés peut changer un problème linéaire en un problème exponentiel





62

Recherche dans un graphe

```
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 fringe is empty then return failure
        node ← REMOVE-FRONT(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 ← INSERTALL(EXPAND(node, problem), fringe)
```

Idée: Comparer avec les noeuds dans *Closed* pour éviter la répétition inutile

Question: Est-ce qu'il y a des répétitions utiles?
Comment gérer ça?

63

Résumé

- La formulation du problème demande généralement une abstraction des détails dans le monde réel afin de définir un espace d'états facile à explorer
- Forme de l'espace d'états peut être différente de la forme du problème (selon la définition d'état)
- Une variété de stratégies de recherche non informées
- Chaque stratégie a ses propres propriétés (à choisir selon le problème/application)
- Algorithmes non informés = utilise seulement la topologie du graphe
- Recherche en approfondissement itératif utilise seulement un espace linéaire, et ne demande pas plus de temps que les autres algorithmes non informés

64