Problèmes et algorithmes de recherche

IA– Chapitre 3

Master STIC - Université d'Orléans - Janvier 2012

Aperçu

- Agents de résolution de problèmes
- Types de problèmes
- Formulation de problèmes
- Exemples de problèmes
- Algorithmes de recherche "aveugles"
  - recherche en largeur
  - recherche en coût uniforme
  - recherche en profondeur
  - recherche en profondeur limitée
  - recherche par approfondissement itératif
  - recherche bi-directionnelle
- Comparaison des algorithmes aveugles

Agents de résolution de problèmes

Forme restreinte d'un agent général :

```
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
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← RECOMMENDATION(seq)
  seq ← REMAINDER(seq)
  return action
```

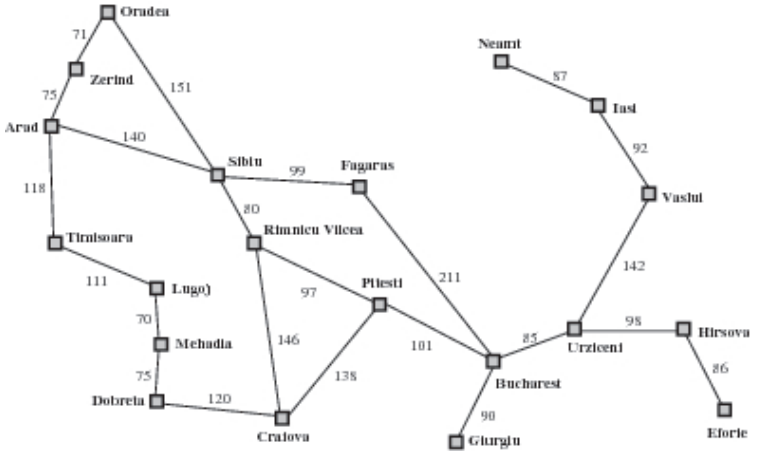

Introduction aux algorithmes de recherche

- Les algorithmes de recherche constituent l'une des approches les plus puissantes pour la résolution de problèmes en IA
- Les algorithmes de recherche sont un mécanisme général de résolution de problème qui
  - se déroule dans un espace appelé *espace d'états*
  - explore systématiquement toutes les *alternatives*
  - trouve la *séquence d'étapes* menant à la solution
- Etat initial
- Action
- Test de solution

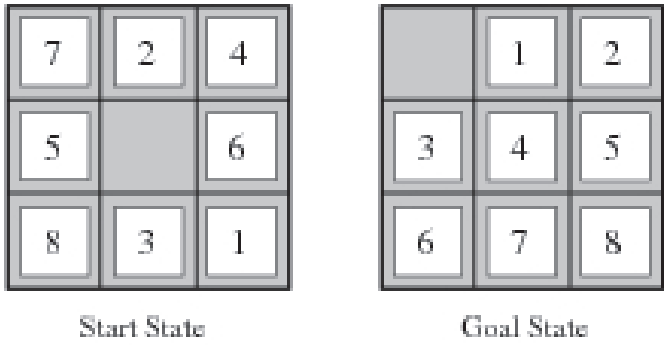
Exemple : La Roumanie

Vacances en Roumanie, position actuelle : Arad.
L'avion quitte Bucarest le lendemain.
Objectif :
Etre à Bucharest
Formulation du problème :
états : les différentes villes
état initial : Arad
actions: se déplacer d'une ville à l'autre
Solution:
une suite de villes, par exemple : Arad, Sibiu, Fagaras, Bucarest

Exemple : La Roumanie



Exemple : Le puzzle-8



Définition d'un problème de recherche

Un problème de recherche est défini par 6 éléments

- Q un ensemble non vide d'états
- $S \subseteq Q$ un ensemble non vide d'états initiaux
- $G \subseteq Q$ un ensemble non vide d'états de solutions (pouvant être définis explicitement ou implicitement par un test de solution)
- A un ensemble d'actions
- Fonction de successeur : $Q \times A \rightarrow Q$
- Fonction de coût : $Q \times A \times Q \rightarrow R^+$ (définie uniquement pour des états qui sont successeurs l'un de l'autre)

Définir un problème de recherche

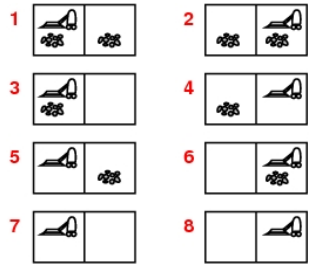
- Espace d'états
- Etat(s) initial (initiaux)
- Fonction de successeur
- Test de solution
- Coût du chemin

Problèmes "jouets"

- Le puzzle-8
- Le tour de Hanoï
- Le loup, la chèvre et le chou
- Le Wumpus
- L'aspirateur
- Le monde des blocs
- Le labyrinthe
- Les mots croisés
- L'arithmétique cryptée
- Les jeux d'echec, de dames, ...

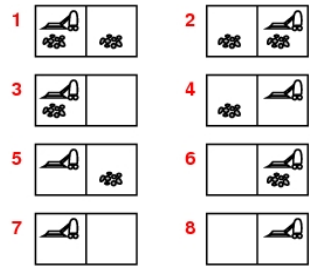
Exemple : l'aspirateur

- Le monde consiste en 2 positions
  – pièce gauche,
  – pièce droite
  - Chaque pièce peut être poussiéreuse
  - L'agent (aspirateur) est dans l'une des 2 pièces
  - Il y a au total 8 états possibles
  - 4 actions possibles :
    – aller à gauche,
    – aller à droite,
    – aspirer,
    – ne rien faire



Exemple : l'aspirateur

- Objectif :
  – éliminer toute la poussière
- Formulation du problème :
  – états : les 8 états possibles
  – actions : les 4 actions possibles
- Solution :
  – être dans l'état 7 ou 8



Types de problèmes

- Problèmes sans capteurs (conformant)
  – état connu (unique) ou dans une liste d'états possibles (multiple)
  – effets des actions connus
  – solution est une séquence

Types de problèmes

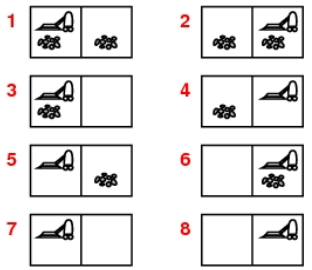
- Problèmes sans capteurs (conformant)
  – état connu (unique) ou dans une liste d'états possibles (multiple)
  – effets des actions connus
  – solution est une séquence
- Problèmes contingents :
  – effet conditionnel des actions
  – perception fournit des nouvelles infos de l'état courant
  – solution est un arbre ou une stratégie

Types de problèmes

- Problèmes sans capteurs (conformant)
  – état connu (unique) ou dans une liste d'états possibles (multiple)
  – effets des actions connus
  – solution est une séquence
- Problèmes contingents :
  – effet conditionnel des actions
  – perception fournit des nouvelles infos de l'état courant
  – solution est un arbre ou une stratégie
- Problème d'exploration :
  – exécution révèle les états
  – besoin d'expérimenter pour trouver la solution

Exemple : l'aspirateur

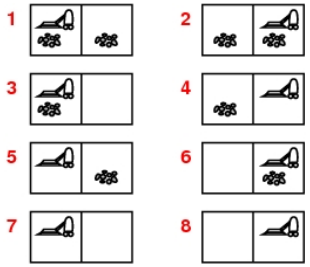
Etat unique, état initial #5. Solution ??



Exemple : l'aspirateur

Etat unique, état initial #5. Solution ??
[*Right*, *Suck*]

Etats multiples, état initial $\{1, 2, 3, 4, 5, 6, 7, 8\}$
Right produit $\{2, 4, 6, 8\}$. Solution ??

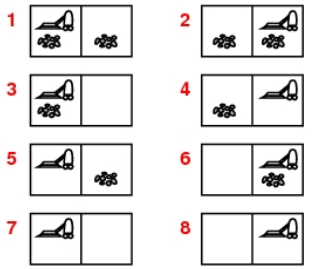


Exemple : l'aspirateur

Etat unique, état initial #5. Solution ??
[*Right*, *Suck*]

Etats multiples, état initial $\{1, 2, 3, 4, 5, 6, 7, 8\}$
Right produit $\{2, 4, 6, 8\}$. Solution ??
[*Right*, *Suck*, *Left*, *Suck*]

Contingent, état initial #5
Lois de Murphy : aspirer peut aussi salir
perception locale : poussière
Solution ??

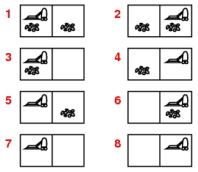


Exemple : l'aspirateur

Etat unique, état initial #5. Solution ??
[*Right*, *Suck*]

Etats multiples, état initial $\{1, 2, 3, 4, 5, 6, 7, 8\}$
Right produit $\{2, 4, 6, 8\}$. Solution ??
[*Right*, *Suck*, *Left*, *Suck*]

Contingent, état initial #5
Lois de Murphy : aspirer peut aussi salir
perception locale : poussière
Solution ??
[*Right*, if *dirt* then *Suck*]



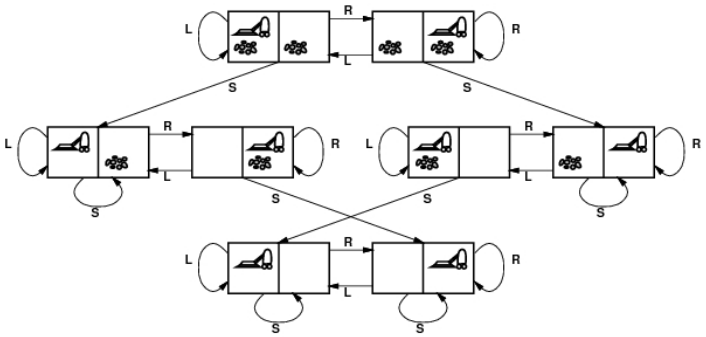
Formulation d'un problème à état unique

Un *problème* est défini par 4 éléments :
état initial ex., "être à Arad"
fonction de successeur $S(x)$ = ensemble de pairs action–état
ex., $S(Arad) = \{< Arad \rightarrow Zerind, Zerind >, \ldots\}$
test, peut être
explicite, ex., x = "être à Bucarest"
implicite, ex., $NoDirt(x)$
fonction de coût (additive)
ex., somme des distances, nombre d'actions exécutées, etc.
$c(x, a, y)$ est le coût d'un mouvement, supposition ≥ 0
Une *solution* est une séquence d'actions menant de l'état initial à l'état final

Modéliser un espace d'états

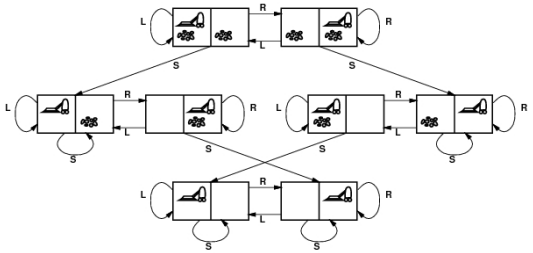
Le monde réel est particulierment complexe
$\Rightarrow$ espace d'état doit être une *abstraction*
Etat (abstrait) = ensemble d'états réels
Action (abstraite) = combinaison complexe d'actions réelles
par exemple, "Arad $\rightarrow$ Zerind" représente un ensemble complexe de routes, détours, points de pause, etc.
Pour être réalisable, chaque état réel "être à Arad" doit pouvoir mener à *un* état réel "être à Zerind"
Solution (abstraite) =
ensembles de chemins réels dans le monde réel
Chaque action abstraite doit être plus "facile" que celle dans le problème original !

Exemple : aspirateur (état unique)



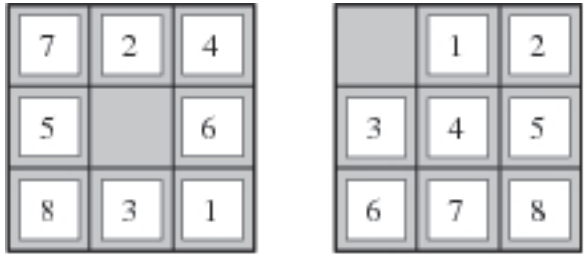
états ??
actions ??
test de solution ??
coût de chemin ??

Exemple : aspirateur (état unique)



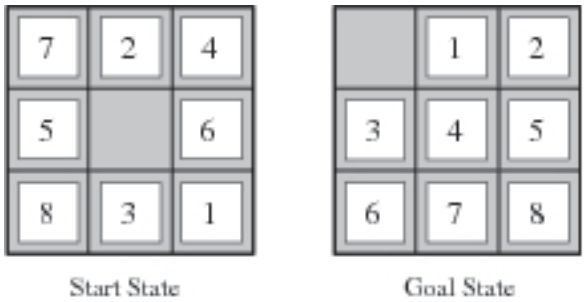
états ?? entier indiquant la position du robot et de la poussière (ignorer la quantité de poussière)
actions ?? aller à gauche (L), aller à droite (R), aspirer (S), ne rien faire (NoOp)
test de solution ?? plus de poussière nulle part
coût de chemin ?? chaque action coûte 1 unité, 0 pour NoOp

Exemple : puzzle-8



états ??
actions ??
test de solution ??
coût de chemin ??

Exemple : puzzle-8

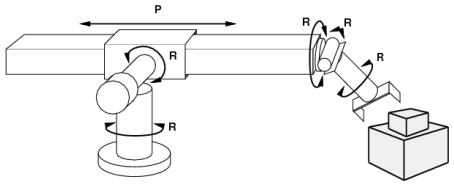


états ?? numéro des positions des plaquettes
actions ?? déplacer la case vide à gauche, à droite, en haut, en bas
goal test ?? état courant = état final
path cost ?? chaque déplacement de la case vide vaut 1
coût total = nombre total de déplacements de la case vide
Remarque : solution optimale pour puzzle-*n* est NP-difficile

Problèmes réels

- Recherche de parcours
  – itinéaires automatiques, guidage routier, planification de routes aériennes, routage sur les réseaux informatiques, ...
- Configuration de circuits VLSI
  – placement de millions d'éléments sur un chip déterminant pour le fonctionnement efficace du circuit
- Robotique
  – assemblage automatique, navigation autonome, ...
- Planification et ordonnancement
  – horaires, organisation de tâches, allocation de ressources, ...

Exemple : assemblage automatique



états ?? coordonnées des articulations du robot et des pièces à assembler
actions ?? mouvement continus des articulations du bras robotique
test de solution ??: assemblage terminé, robot en position de repos
coût ??: temps d'exécution

Typologie des problèmes de recherche

- Tous les problèmes qui peuvent être décrits par
  – un ensemble fini d'états,
  – un ensemble fini d'actions,
  – un sous-ensemble d'états initiaux et finaux,
  – une relation "successeur" définie sur l'ensemble des états et des actions dans l'ensemble des états, et
  – une fonction de coût positive.
- Principalement les problèmes dont la solution s'exprime en termes de chemin dans des graphes finis.

Hypothèses de base

- L'environnement est statique
- L'environnement est discrétisable
- L'environnement est observable
- Les actions sont déterministes

Variantes

- Un ou plusieurs états initiaux
- Un ou plusieurs états finaux
- La solution est-elle un chemin ou un état isolé ?
- Une, la meilleure ou toutes les solutions ?

Paramètres importants

- Nombre d'états dans l'espace d'états
- Distribution des états finaux
- Espace mémoire nécessaire pour stocker un état
- Temps d'exécution de la fonction de successeur

Algorithme général de recherche

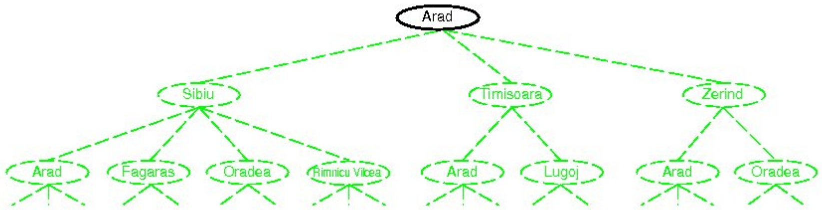
Idée de base:

exploration "offline" simulée de l'espace d'états

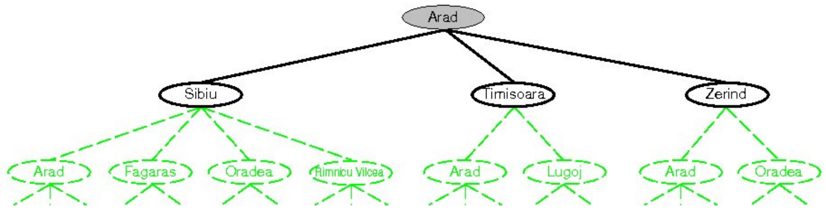
en générant les états successeurs des états déjà explorés (processus aussi appelé *expansion* d'états)

```
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
   end
```

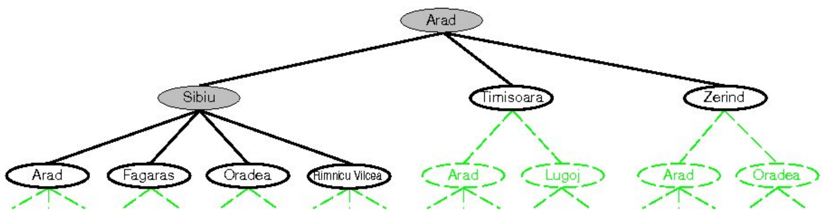
Exemple



Exemple



Exemple



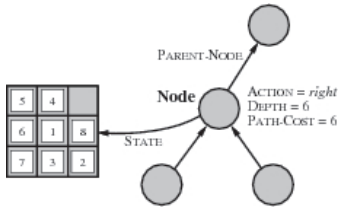
Concepts de base pour la recherche

Un algorithme de recherche repose sur les éléments suivants :

- Arbre de recherche
  l'arbre de recherche peut être infini même lorsque le graphe de recherche est fini
- Recherche d'un nœud
- Expansion d'un nœud
- Stratégie de recherche
  à chaque étape du processus de recherche, déterminer quel est le nœud à explorer

Implantation : états vs. nœuds

Un *état* est une (représentation d') une configuration réelle
Un *nœud* est un élément d'une structure de données constituant un arbre de recherche
il possède les champs : *parent*, *enfants*, *profondeur*, *coût de chemin* $g(x)$
Etats ne possèdent pas parents, enfants, depth, profondeur ou coût !



La fonction EXPAND crée de nouveaux nœuds, donne des valeurs aux différents champs et en utilisant la fonction SUCCESSORFN du problème crée les états correspondants.

File d'attente

La gestion des nœuds de l'arbre de recherche repose sur l'utilisation d'une structure de données particulière : la file d'attente

- Elle contient les nœuds de recherche pas encore explorés
- Elle est implémentée à l'aide des procédures suivantes :
  – INSERT(nœud, file) = insérer un nœud dans la file d'attente
  – REMOVE-FRONT(file) = extraire le nœud qui occupe la première position dans la file d'attente
- L'ordre dans lequel les nœuds sont arrangés dans la file d'attente détermine la stratégie de recherche

Implantation : algorithme général de recherche

```
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 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(STATE[node], action, result)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```


Stratégie de recherche

Une stratégie est définie par *l'ordre d'expansion de nœuds*
Stratégies sont évaluées suivant les critères :
complétude–la stratégie trouve-t-elle toujours une solution si elle existe ?
complexité en temps–nombre de nœuds générés/explorés
complexité en espace–nombre maximum de nœuds à garder dans la mémoire pour trouver la solution
optimalité–la stratégie trouve-t-elle toujours la meilleure solution ?

Paramètres de complexité

La complexité est mesurée en fonction de :

- b = facteur de branchement de l'arbre de recherche (= nombre maximum de successeurs pour un état)
- d = profondeur à laquelle se trouve la meilleure solution
- m = profondeur maximum de l'espace de recheche (peut être ∞)

Stratégies "aveugles" et stratégies heuristiques

- Stratégies aveugles
  n'exploitent aucune information contenue dans un nœud donné
- Stratégies heuristiques
  exploitent certaines informations pour déterminer si un nœud est "plus prometteur" qu'un autre

Remarques importantes

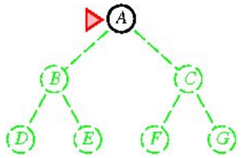
- Des problèmes formulés comme des problèmes de recherche sont NP-difficiles, par exemple puzzle-n.
- On ne peut pas espérer les résoudre en moins qu'un temps exponentiel dans le pire des cas.
- Mais on peut cependant tenter de résoudre le plus d'instances possibles de tels problèmes.

Stratégies aveugles

- Recherche en largeur (Breadth-first search)
- Recherche en coût uniforme (Uniform-cost search)
- Recherche en profondeur (Depth-first search)
- Recherche en profondeur limitée (Depth-limited search)
- Recherche par approfondissement itératif (Iterative deepening search)

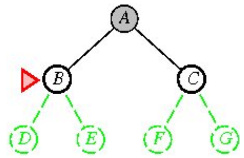
Algorithme de recherche en largeur

Stratégie : étendre le nœud le moins profond
Implémentation:
la file d'attente est FIFO, c'est-à-dire que les nouveaux successeurs sont insérés à la fin de la file



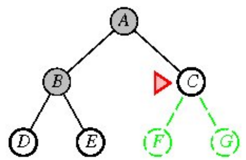
Algorithme de recherche en largeur

Stratégie : étendre le nœud le moins profond
Implémentation:
la file d'attente est FIFO, c'est-à-dire que les nouveaux successeurs sont insérés à la fin de la file



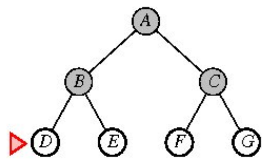
Algorithme de recherche en largeur

Stratégie : étendre le nœud le moins profond
Implémentation:
la file d'attente est FIFO, c'est-à-dire que les nouveaux successeurs sont insérés à la fin de la file



Algorithme de recherche en largeur

Stratégie : étendre le nœud le moins profond
Implémentation:
la file d'attente est FIFO, c'est-à-dire que les nouveaux successeurs sont insérés à la fin de la file



Propriétés de la recherche en largeur

- Complétude ?? Oui (si b est fini)

Propriétés de la recherche en largeur

- Complétude ?? Oui (si b est fini)
- Complexité en temps ??
  $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$,
  exponentiel en d

Propriétés de la recherche en largeur

- Complétude ?? Oui (si b est fini)
- Complexité en temps ??
  $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$,
  exponentiel en d
- Complexité en espace ?? $O(b^{d+1})$ (il faut garder tous les nœuds dans la mémoire)

Propriétés de la recherche en largeur

- Complétude ?? Oui (si b est fini)
- Complexité en temps ??
  $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$,
  exponentiel en d
- Complexité en espace ?? $O(b^{d+1})$ (il faut garder tous les nœuds dans la mémoire)
- Optimalité ?? Oui (si coût unitaire par étape) ; en général pas optimal

Propriétés de la recherche en largeur

- Complétude ?? Oui (si b est fini)
- Complexité en temps ??
  $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$,
  exponentiel en d
- Complexité en espace ?? $O(b^{d+1})$ (il faut garder tous les nœuds dans la mémoire)
- Optimalité ?? Oui (si coût unitaire par étape) ; en général pas optimal
- *Complexité en espace* pose un grand problème ; si les nœuds sont générés à la vitesse de 10MB/sec
  alors 24hrs = 860GB.
  Rappel : notation Grand O
  $g(n)$ est $O(f(n))$ si $\exists k, n_0 > 0 \ : \ \forall n > n_0, \ g(n) \leq k.f(n)$

Algorithme de recherche en coût uniforme

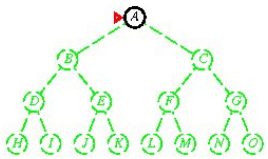
- La recherche en largeur trouve le nœud de solution le moins profond
- La recherche en coût uniforme étend systématiquement le nœud de coût le plus faible (mesuré par la fonction de coût $g(n)$)
- Le coût d'un chemin ne doit jamais décroître
  – c'est-à-dire qu'il ne doit pas y avoir de coûts partiels négatifs
  – $\forall n, \quad g(\text{Successeur}(n)) \geq g(n)$

Algorithme de recherche en coût uniforme

Stratégie : étendre le nœud de coût le plus faible
Implantation:
INSERT-ALL = insertion dans l'ordre croissant de coût de chemin
Equivalent à l'algorithme de recherche en largeur si le coût de chaque étape est unique
Complétude ?? Oui, si le coût d'une étape $\geq \epsilon$
Complexité en temps ?? $O(b^d)$
Complexité en espace ?? $O(b^d)$
Optimalité ?? Oui – nœuds sont explorés en ordre croissant de $g(n)$

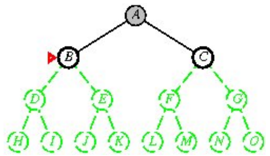
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



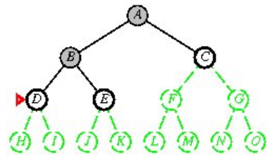
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



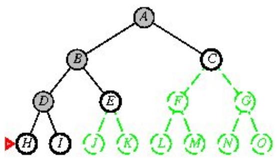
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



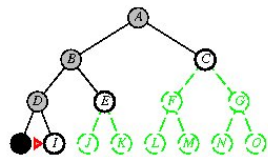
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



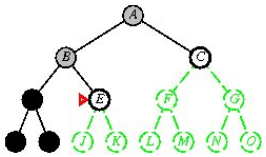
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



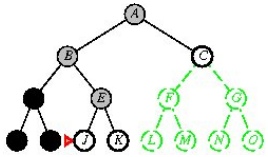
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



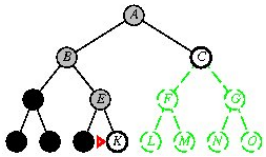
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



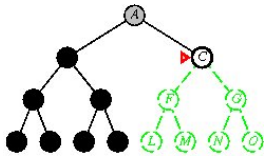
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



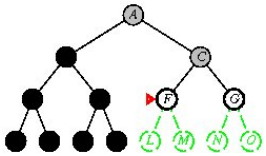
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



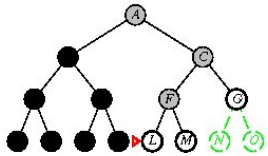
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



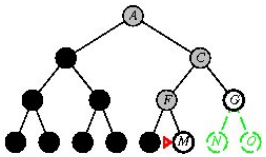
Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



Algorithme de recherche en profondeur

Stratégie : étendre le nœud le plus profond
Implantation:
file d'attente = liste LIFO, les nouveaux successeurs sont placés en tête de la file d'attente



Propriétés de l'algorithme de recherche en profondeur

- Complétude ?? Non, échoue dans les espaces infinis ou avec cycles
  Modifier pour éviter de répéter des états sur un chemin
  $\Rightarrow$ complet dans les espaces finis acycliques

Propriétés de l'algorithme de recherche en profondeur

- Complétude ?? Non, échoue dans les espaces infinis ou avec cycles
  Modifier pour éviter de répéter des états sur un chemin
  $\Rightarrow$ complet dans les espaces finis acycliques
- Complexité en temps ?? $O(b^m)$: terrible si m est beaucoup plus grand que d
  mais si les solutions sont denses, peut être beaucoup plus rapide que recherche en largeur

Propriétés de l'algorithme de recherche en profondeur

- Complétude ?? Non, échoue dans les espaces infinis ou avec cycles
  Modifier pour éviter de répéter des états sur un chemin
  $\Rightarrow$ complet dans les espaces finis acycliques
- Complexité en temps ?? $O(b^m)$: terrible si m est beaucoup plus grand que d
  mais si les solutions sont denses, peut être beaucoup plus rapide que recherche en largeur
- Complexité en espace ?? $O(bm)$ ou $O(m)$ linéaire !

Propriétés de l'algorithme de recherche en profondeur

- Complétude ?? Non, échoue dans les espaces infinis ou avec cycles
  Modifier pour éviter de répéter des états sur un chemin
  $\Rightarrow$ complet dans les espaces finis acycliques
- Complexité en temps ?? $O(b^m)$: terrible si m est beaucoup plus grand que d
  mais si les solutions sont denses, peut être beaucoup plus rapide que recherche en largeur
- Complexité en espace ?? $O(bm)$ ou $O(m)$ linéaire !
- Optimalité ?? Non

Propriétés de l'algorithme de recherche en profondeur

- Complétude ?? Non, échoue dans les espaces infinis ou avec cycles
  Modifier pour éviter de répéter des états sur un chemin
  $\Rightarrow$ complet dans les espaces finis acycliques
- Complexité en temps ?? $O(b^m)$: terrible si m est beaucoup plus grand que d
  mais si les solutions sont denses, peut être beaucoup plus rapide que recherche en largeur
- Complexité en espace ?? $O(bm)$ ou $O(m)$ linéaire !
- Optimalité ?? Non
  : besoins modestes en espace
  pour $b = 10$, $d = 12$ et 100 octets/nœuds :
  – recherche en profondeur a besoin de 12 Koctets
  – recherche en largeur a besoin de 111 Tera-octets, 10^{10} fois de plus !!!

Algorithme de recherche en profondeur limitée

Stratégie = algorithme de recherche en profondeur mais avec une limite de profondeur d'exploration l,
c'est-à-dire, les nœuds de 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 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
```


Propriétés de la recherche en profondeur limitée

Complétude ?? Oui, si $l \geq d$
Complexité en temps ?? $O(b^l)$
Complexité en espace ?? $O(b * l)$
Optimalité ?? Non
Trois possiblités :
– solution
– échec
– absence de solution dans les limites de la recherche

Recherche par approfondissement itératif

- Le problème avec la recherche en profondeur limitée est de fixer la bonne valeur de l
- Approfondissement itératif = répéter pour toutes les valeurs possibles de $l = 0, 1, 2, \ldots$
- Combine les avantages de la recherche en largeur et en profondeur
  – optimal et complet comme la recherche en largeur
  – économe en espace comme la recherche en profondeur
- C'est l'algorithme de choix si l'espace de recherche est grand et si la profondeur de la solution est inconnue

Recherche par approfondissement itératif

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

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

Recherche par approfondissement itératif $l = 0$

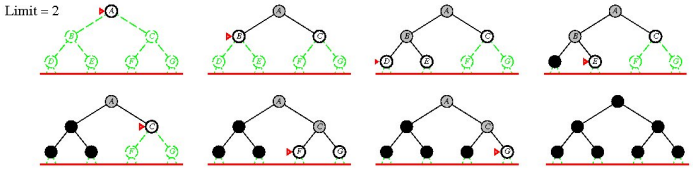
Limit = 0

Recherche par approfondissement itératif $l = 1$

Limit = 1

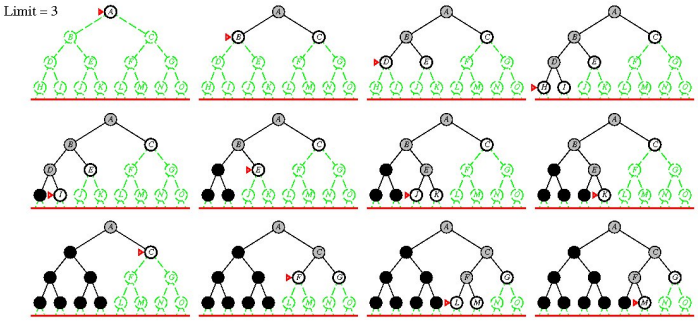
Recherche par approfondissement itératif $l = 2$

Limit = 2



Recherche par approfondissement itératif $l = 3$

Limit = 3



Propriétés de la recherche par approfondissement itératif

- Complétude ?? Oui

Propriétés de la recherche par approfondissement itératif

- Complétude ?? Oui
- Complexité en temps ??
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Propriétés de la recherche par approfondissement itératif

- Complétude ?? Oui
- Complexité en temps ??
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Complexité en espace ?? $O(bd)$

Propriétés de la recherche par approfondissement itératif

- Complétude ?? Oui
- Complexité en temps ??
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Complexité en espace ?? $O(bd)$
- Optimalité ?? Oui, si coût unitaire par étape
  Peut être modifié pour explorer l'arbre de coût uniforme

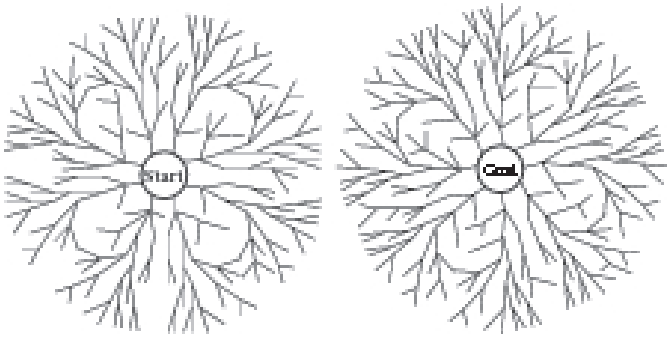
Discussion

approfondissement itératif = largeur + profondeur
– 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, seuls les nœuds des niveaux supérieus sont étendus plusieurs fois
Comparaison numérique pour $b = 10$ et $d = 5$, solution tout à droite de l'arbre :

- Complexité en espace de la recherche en largeur
  $1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d = 1.111.100$ nœuds
- Complexité en espace de la recherche par approfondissement itératif
  $(d+1)1 + (d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d$
  $= 123.456$ nœuds

Algorithme de recherche bi-directionnelle

- Recherche en largeur de l'état final à partir de l'état initial
- Recherche en largeur de l'état initial à partir de l'état final
- Arrêter la recherche lorsque les 2 processus se rencontrent



Propriétés de la recherche bi-directionnelle

Complétude ?? Oui
Complexité en temps ?? $O(b^{d/2}) << O(b^d)$
Complexité en espace ?? $O(b^{d/2}) << O(b^d)$
Optimalité ?? Oui
Nécessite :
– des états finaux sont explicitement définis
– des actions dont on connaît la fonction inverse (capable de générer l'état prédécesseur d'un état donné)

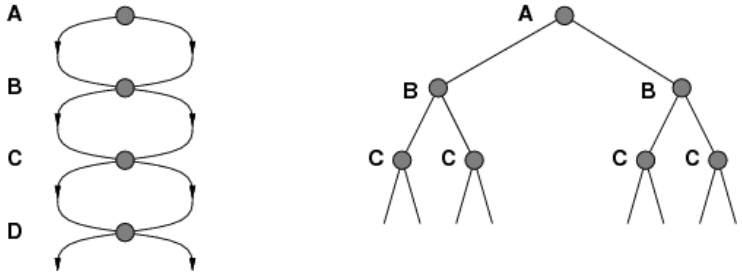
Comparaison des algorithmes

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes	Yes
Time	b^{d+1}	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes*	Yes	No	No	Yes*	Yes

b = facteur de branchement
d = profondeur de la solution
m = profondeur maximum de l'arbre de recherche
l = limite de la profondeur de recherche

Etats répétés

L'échec de détection d'états répétés peut transformer un problème linéaire en un problème exponentiel !



Eviter les états répétés

- Il faut comparer les descriptions d'états
- Recherche en largeur
  – Garder la trace de tous les états produits
  – Si l'état d'un nouveau nœud existe déjà, éliminer ce nœud
- Recherche en profondeur
  – Solution 1
  Garder la trace de tous les états associés à des nœuds dans l'arbre courant
  Si l'état d'un nouveau nœud existe déjà, éliminer ce nœud
  → Evite les boucles
  – Solution 2
  Garder la trace de tous les états produits jusqu'ici
  Si l'état d'un nouveau nœud a déjà été produit, éliminer ce nœud
  → Complexité en espace de la recherche en largeur

Algorithme général

```
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 node
      if STATE[node] is not in closed  then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```


Résumé

- La formulation de problème nécessite souvent l'abstraction des détails dans le monde réel pour définir un espace d'états pouvant être exploré en pratique
- Plusieurs algorithmes de recherche aveugles
- La recherche par approfondissement itératif utilise un espace linéaire et n'est pas beaucoup plus lent par rapport aux autres algorithmes de recherche aveugles