Troisième partie III

Algorithmes classiques de recherche en IA

Plan

1. Introduction à l'intelligence artificielle
2. Agents intelligents
3. Algorithmes classiques de recherche en IA
4. Algorithmes et recherches heuristiques
5. Programmation des jeux de réflexion
6. Problèmes de satisfaction de contraintes
7. Agents logiques
8. Logique du premier ordre
9. Inférence en logique du première ordre
10. Introduction à la programmation logique avec Prolog
11. Planification
12. Apprentissage

En bref ...

Agent avec objectifs (buts) explicites

Les différents types de problèmes

Exemples de problèmes

Algorithme de recherche de base (recherche aveugle)

Agent avec objectifs (buts) explicites

Agent avec objectifs (buts) explicites

Algorithme

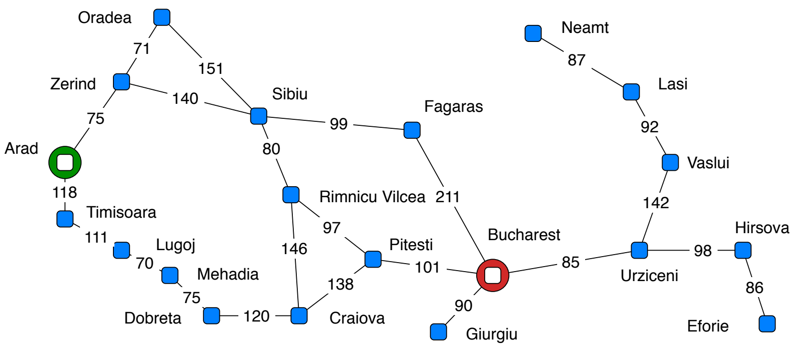
funtion SIMPLE-PROBLEM-SOLVING-AGENT(p) **returns** *an action*
 input : *p, a percept*
 static : *s, an action sequence, initially empty*
 state, some description of the current world state
 g, a goal initially null
 problem, a problem formulation

 state ← UPDATE-STATE(*state*, *p*)
 if *s is empty* **then**
  g ← FORMULATE-GOAL(*state*)
  problem ← FORMULATE-PROBLEM(*state*, *g*)
  s ← SEARCH(*problem*)
 action ← FIRST(*s*), *s* ← REST(*s*)
 return *action*

Exemple de problème : le voyage en Roumanie

- En vacances en Roumanie, actuellement dans la ville d'Arad mon vol de retour part demain de Bucharest. Comment rejoindre Bucharest ?
- Le but
  - être à Bucharest
- La formulation du problème
  - les états : les villes de Roumanie
  - les actions : déplacement de ville en ville
- Solution au problème
  - une sequence de ville me permettant d'arriver à Bucharest, par exemple Arad, Sibiu, Fagaras et Bucharest.

Exemple de problème : le voyage en Roumanie



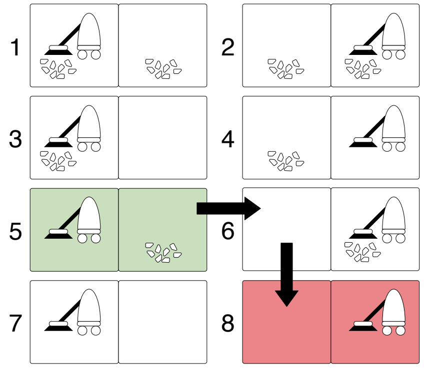
Les différents types de problèmes

Les différents types de problèmes

- Déterministe, complètement observable $\longrightarrow$ 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 $\longrightarrow$ problème sans possibilité de percevoir l'environement
  - L'agent n'a aucune idée d'ou il est réellement
  - La solution est une séquence d'actions
- Non-déterministe or partiellement observable $\longrightarrow$ 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 $\longrightarrow$ problème d'exploration

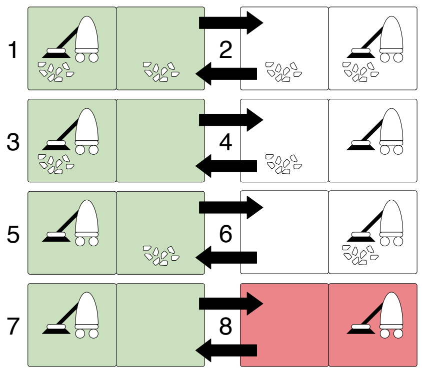
Exemple : Le monde de l'aspirateur

- État initial #5
- Solution ? $\longrightarrow$ $\langle$*droite*, *aspire*$\rangle$



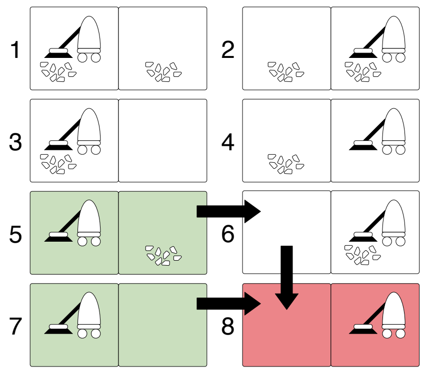
Exemple : Le monde de l'aspirateur

- États initiaux {#1,#2,#3,#4,#5,#6,#7,#8}
- Solution pour { #1,#3,#5,#7} ? $\longrightarrow$ $\langle$*droite*, *aspire*, *gauche*, *aspire*$\rangle$
- Solution pour { #2,#4,#6,#8} ? $\longrightarrow$ $\langle$*gauche*, *aspire*, *droite*, *aspire*$\rangle$



Exemple : Le monde de l'aspirateur

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



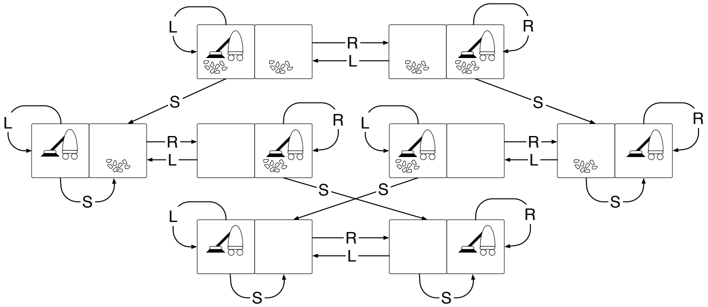
Les problèmes déterminites et complètement observables

- Un problème non-déterministe et complètement observable est défini :
  1. un état initial
     - e.g., ≪ à Arad ≫
  2. un ensemble d'actions ou une fonction de transition, $succ(x)$:
     - e.g., $succ(Arad) = \{Zerind, Timisoara\}$
  3. un test de terminaison pour savoir si le but est atteind
     - explicite, e.g., ≪ à Arad ≫
     - implicite, e.g., vérifier mat au échec
  4. un coût (additif)
     - e.g., la somme des distances, le nombre d'actions exécutées, etc.
     - e.g., $c(x, a, y)$ 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 aut but.

L'espace d'états

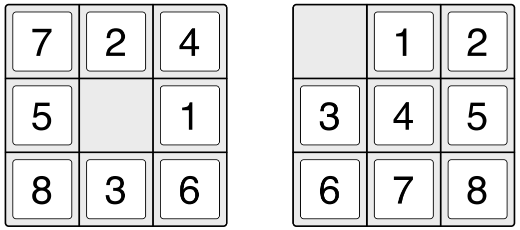
- Le monde réel est trop complexe pour être modélisé
  - L'espace de recherche modélise une vue abstraite et simplifiée du monde réel
- Un état abstrait représente un ensemble d'états réels
- Une action abstraite représente une combinaison complexe d'actions réelles
  - e.g., ≪ Arad → Zerind ≫ représente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

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



- États ? sol sale et position de l'aspirateur
- Actions ? droite, gauche, aspire
- Test du but ? toutes les positions doivent être propres
- Coût du chemin ? 1 par action

Exemple : le jeux du taquin



État initial

État but

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

Exemples de problèmes

Exemple de problème réels

- Trouver des chemins
- Trouver un tour
- Design de circuit
- Navigation de robot
- Recherche sur internet
- etc.

Exemple : le robot assembleur

- États ? coordonnées du robots, angles, position de l'objet à assemblé, etc.
- Actions ? déplacements continus
- Test du but ? objet complètement assemblé
- Coût du chemin ? le temps d'assemblable

Algorithme de recherche de base (recherche aveugle)

Algorithme de recherche de base (recherche aveugle)

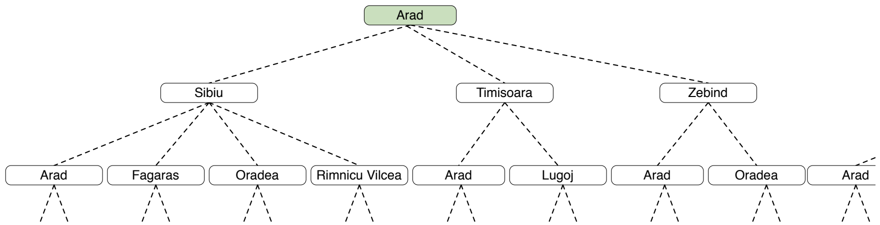
- Idée de base
  - recherche hors ligne, i.e., exploration de l'espace d'états en générant des successeurs d'états déjà explorés (développer des états)
  - Génération d'un arbre de recherche

Algorithme

funtion GENERAL-SEARCH(*problem*, *stategy*) **returns** *a solution or failure*
  initialize the search tree using 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*

- On s'arrête quand on a choisi de développer un nœud qui est un état final

Exemple : Arbre de recherche



Implémentation des algorithmes de recherche

- On définit une stucture de données nœuds qui contient état, parent, enfant, profondeur, coût du chemin noté $g(x)$
- EXPAND crée des nouveaux nœuds
- INSERT-FN insère des nœuds dans la liste des nœuds à traiter

Algorithme

funtion GENERAL-SEARCH(*problem*,INSERT-FN) **returns** *a solution or failure*
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
  loop do
    if *nodes is empty* **then return** *failure*
    node ← REMOVE-FRONT(*nodes*)
    if GOAL-TEST[*problem*] *applied to* STATE(*node*) *succeeds* **then**
      return *node*
    nodes ← INSERT-FN(*nodes*,EXPAND(*node*,OPERATORS[*problem*]))

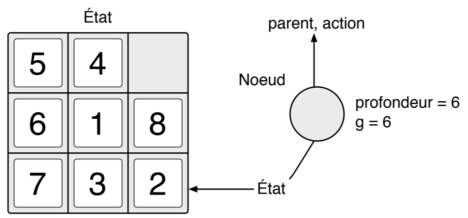
Algorithme de recherche dans les arbres

Algorithme

funtion EXPAND(*node*, *problem*) **returns** *a set of nodes*
  successors ← *the empty set*
  foreach *action*, *result in* SUCCESSOR[*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*

États versus nœuds

- Un état est une représentation d'une configuration physique du monde
- Un nœud est une structure de données partie intégrante de l'arbre de recherche incluant :
  - 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
  - la profondeur du nœud, i.e., la distance entre le nœud et la racinde de l'arbre



- Les différents attributs des nœuds sont initiallisés par la fonction EXPAND

Stratégie de recherche

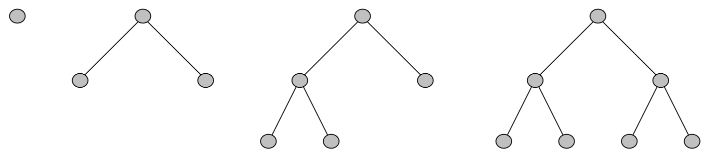
- Une stratégie de recherche est définie par l'ordre dans lequel les nœuds sont développés, i.e., la fonction INSERT-FN
- Une stratégie s'évalue en fonction de 4 dimensions :
  - la complétude : est ce que cette stratégie trouve toujours une solution si elle exise ?
  - la complexité en temps : le nombre de nœuds créés
  - la complexité en mémoire : le nombre maximum de noeuds en mémoire
  - l'optimalité : est ce que la stratégie trouve toujours la solution la moins coûteuse ?
- 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 maximum de l'arbre de recherche
    - attention peut être ∞

Stratégies de recherche non-informées (aveugle)

- Les stratégies de recherche noninformées 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

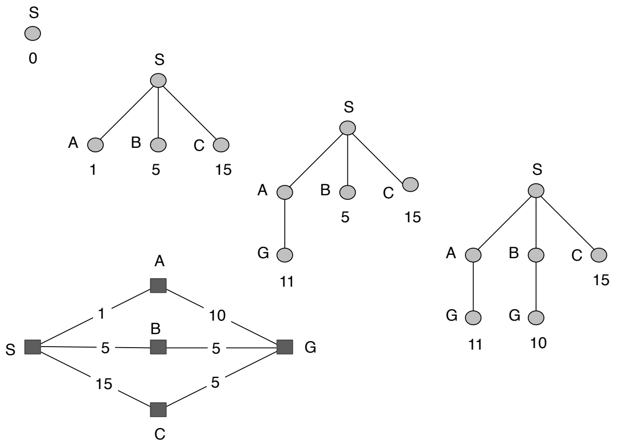
- La fonction INSERT-FN ajoute les successeurs en fin de liste



- Complet, si b est fini
- Temps : $\Sigma_{i=1}^{d} b^i + (b^{d+1} - b) = O(b^{d+1})$
- Espace : idem
- Optimale, si cout = 1 pour chaque pas, non optimale en général

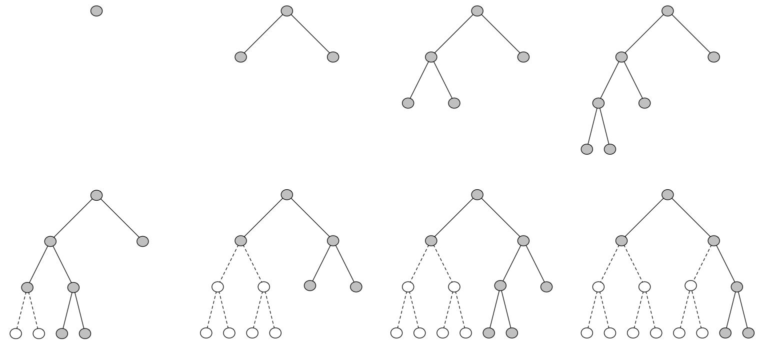
Recherche en coût uniforme

- La fonction Insert-Fn ajoute les nœuds dans l'ordre de leur coût de chemin



Recherche en profondeur d'abord

- La fonction Insert-Fn ajoute les nœuds au début de la liste

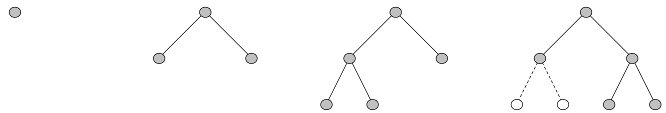


Recherche en profondeur d'abord

- Non complet dans les espaces d'états infinis ou avec boucle
  - On peut ajouter un test pour détecter les répétitions
- Temps : $O(b^m)$
- Espace : $O(bm)$
- non optimale

Recherche en profondeur limitée

- Recherche en profondeur d'abord avec limite l de profondeur
- Exemple $l = 2$:



Recherche en profondeur limitée

Algorithme

funtion DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** *a sol, fail or cutoff*
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]), *problem*, *limit*)

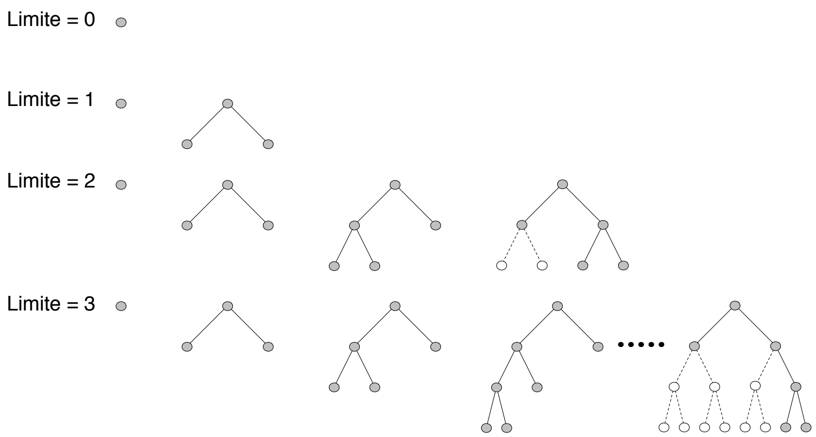
funtion RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** *a sol, fail or cutoff*
  cutoff-occured $\leftarrow$ *false*
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  else if DEPTH[*node*] $=$ *limit* **then return** *cutoff*
  else **foreach** *successors* *in* EXPAND(*node*, *problem*) **do**
    result $\leftarrow$ RECURSIVE-DLS(*successor*, *problem*, *limit*)
    if *result* $=$ *cutoff* **then** *cutoff-occured* $\leftarrow$ *true*
    else if *result* $\neq$ *failure* **then** **return** *result*
  if *cutoff-occured* **then return** *cutoff* **else return** *failure*

Recherche itérative en profondeur

Algorithme

funtion ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** *a solution sequence*
  input: *problem a problem*
  for *depth* $\leftarrow$ 0 *to* ∞ **do**
    if DEPTH-LIMITED-SEARCH(*problem*, *depth*) *succeed* **then**
      return *its result*
  return *failure*

Recherche itérative en profondeur



Recherche itérative en profondeur

- Complet
- Temps : $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Expace : $O(bd)$
- Optimalité : optimale, si coût $=1$ pour charque pas
  - peut être adapté sinon

Résumé des algorithmes de recherche

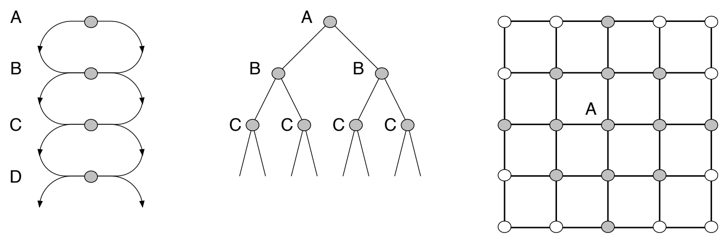
Critères	BFS	UCF	DFS	DLS	IDS
Complétude	oui	oui	non	non	oui
Temps	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Espace	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimalité	oui	oui	non	non	oui

Recherche sur des graphes

- On ajoute à chaque algorithme une liste d'états (nœuds) déjà développés
- L'optimalité n'est plus toujours assurée
- La complexité change

Recherche sur des graphes

- Souvent, on perd du temps en développant des états déjà explorés



(a) (b) (c)

- L'arbre de recherche peut être exponentiellement plus grand

Recherche sur des graphes

Algorithme

funtion GRAPH-SEARCH(*problem*) **returns** *a solution or failure*
  input: *problem a problem*
  closed ← *an empty set of nodes*
  open ← INSERT-FN(MAKE-NODE(INITIAL-STATE[*problem*]), *open*)
  loop do
    if *open is empty* **then return** *failure*
    node ← REMOVE-FRONT(*open*)
    if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
    if STATE[*node*] *is not in closed* **then**
      add STATE[*node*] *to closed*
      open ← INSERT-FN(EXPAND(*node*, *problem*), *open*)