Résoudre des problèmes par recherche

- Agent avec objectifs (buts) explicites
- Types de problèmes
- Formaliser un problème
- Exemples de problèmes
- Algorithmes de recherche de base (recherche aveugle)
- Algorithmes de recherche informés
- Algorithmes de recherche locale
- Algorithmes génétiques

Agent avec objectifs (buts) explicites

function SIMPLE-PROBLEM-SOLVING-AGENT(p) **returns** an action
  inputs: 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

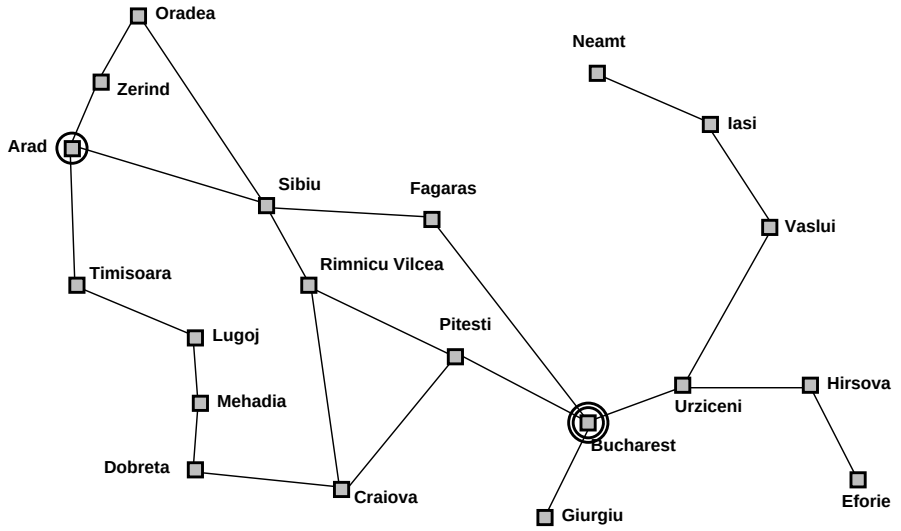
  $state \leftarrow$ UPDATE-STATE($state, p$)
  if s is empty **then**
    $g \leftarrow$ FORMULATE-GOAL($state$)
    $problem \leftarrow$ FORMULATE-PROBLEM($state, g$)
    $s \leftarrow$ SEARCH(*problem*)
  $action \leftarrow$ RECOMMENDATION($s, state$)
  $s \leftarrow$ REMAINDER($s, state$)
  return *action*

Recherche de solutions offline

Exemple

- L'agent est en vacances en Roumanie.
- L'avion part demain de Bucharest.
- L'agent est à Arad en ce moment.
- Formaliser but: être à Bucharest.
- Formaliser le problème:
  - états: les villes
  - actions: conduire entre les villes
- Trouver une solution: séquence de villes, par exemple: Arad, Sibiu, Fagaras, Bucharest

Exemple



Problème à information parfaite

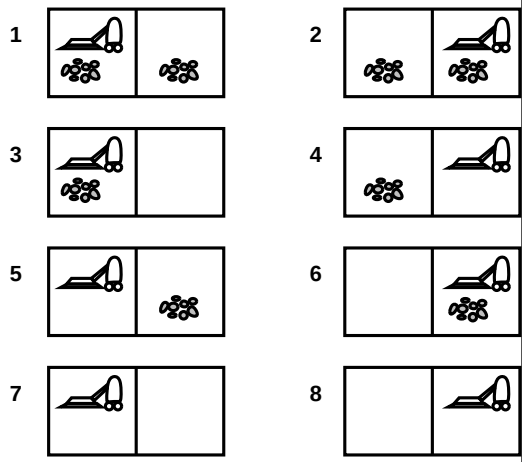
- Problème à état simple
- déterministe
- accessible

Problèmes à information incomplète

- Problème à états multiples:
  - déterministe
  - inaccessible (en partie)
- Problème de l'imprévu:
  - non déterministe
  - inaccessible (en partie)
  - On doit utiliser des capteurs pendant l'exécution
  - La solution est un arbre
  - On alterne souvent recherche et exécution
- Problème d'exploration:
  - L'espace d'états est inconnu

Exemple: Le monde de l'aspirateur

- état simple: départ dans l'état 5.
- état multiple: départ dans $\{1, 2, 3, 4, 5, 6, 7, 8\}$
- L'imprévu: Départ dans l'état 5, mais aspirer peut salir un sol propre. Capteur local seulement: poussière, position



Formalisation du problème à état simple

Un problème est défini par:

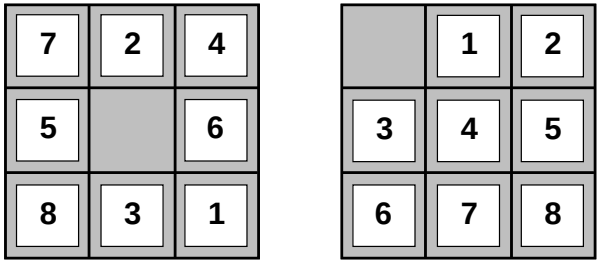
- Un espace d'états (p.e.: les villes)
- Un état initial (p.e.: "à Arad")
- Les actions disponibles: Il suffit d'avoir une fonction de successeurs S(x)
  - p.e.: Arad $\rightarrow$ Zerind, Arad $\rightarrow$ Sibiu, etc.
- Un test état final
  - explicite: x = "à Bucharest"
  - implicite: PasdePoussière(x)
- Coût de chemin:
  - p.e.: Somme de distance, nombre d'action exécutés, etc.

Une solution est une séquence d'action de l'état initial vers un état final.

Choisir un espace d'états

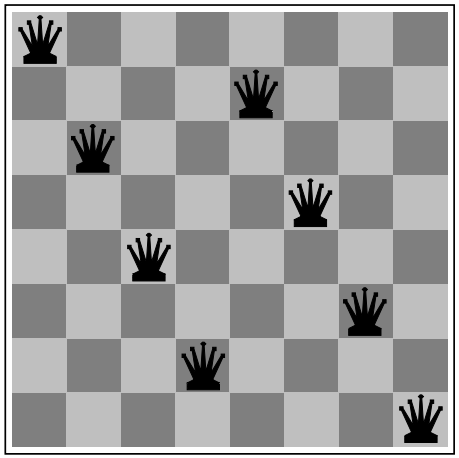
- Le monde réel est trop complexe
- Il faut l'abstraire
- état abstrait: ensemble d'états réels
- action abstraite: combinaison d'actions réelles
- Solution abstraite: Ensemble de chemins réels qui sont des solutions dans le monde réel.

Exemple: 8-puzzle



- états: coordonnées des pièces
- actions: Déplacer le blanc vers la gauche, la droite, le haut, le bas
- test d'état final: donné
- Coût de chemin: Nombre d'actions

Exemple: 8 reines



- Placer 8 reines sur un échiquier de sorte qu'aucune reine attaque une autre.
- Deux descriptions possibles: incrémental, complète

8 reines

Description complète:

- États: Tous les arrangements de 8 reines sur l'échiquier.
- État initial: Choisi au hasard
- Fonction de successeur: Changer la position d'une reine.
- Test d'état final: 8 reines sur l'échiquier, aucune attaquée

Description incrémentale:

- États: Tous les arrangements de 0 à 8 reines sur l'échiquier.
- État initial: Pas de reines sur l'échiquier
- Fonction de successeur: Ajouter une reine sur une case vide.
- Test d'état final: 8 reines sur l'échiquier, aucune attaquée

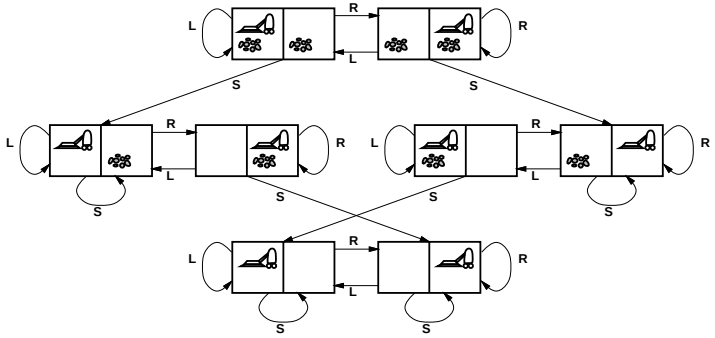
Exemple: Monde des blocs

- états: l'état des trois piles
- actions: Enlever le bloc le plus haut d'une pile et le mettre sur une autre pile
- test d'état final: donné
- Coût de chemin: Nombre d'actions

Exemple de problèmes réels

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

Exemple: Monde de l'aspirateur, graphe d'états

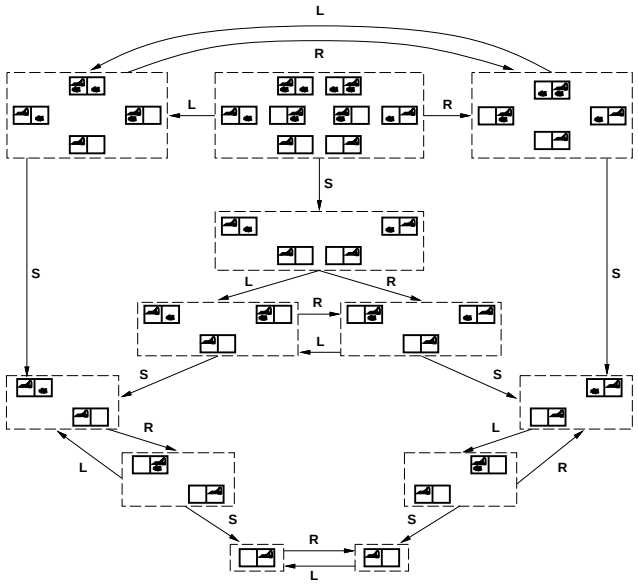


- états: coordonnés de l'aspirateur et de la poussière
- actions: droite, gauche, aspirer
- test d'état final: pas de poussière
- Coût de chemin: Nombre d'actions

Exemple: Monde de l'aspirateur sans capteurs

- L'aspirateur ne sait ni où il est, ni où est la poussière.
- Ensemble d'états: les sous-ensembles de l'ensemble des états de l'exemple d'avant
- Même sans capteurs il est possible d'atteindre le but (un état final) !

Exemple: Monde de l'aspirateur sans capteurs



Algorithme de recherche

- Idée de base: offline, exploration de l'espace d'états en générant des successeurs d'états déjà explorés (développer des états)
- Génération d'un arbre de recherche

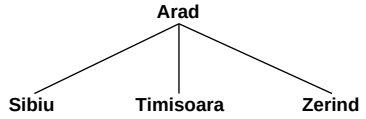
function GENERAL-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

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

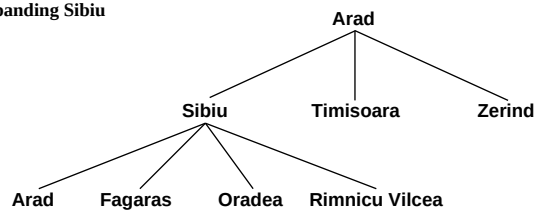
Exemple

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu



Implémentation des algorithmes de recherche

- On définit une structure de donnée nœuds (**node**) qui contient état, parent, enfants, profondeur, coût du chemin: $g(x)$
- EXPAND crée des nouveaux nœuds.
- QUEUING-FN est une fonction qui insère des nœuds dans la liste des nœuds à traiter.

function GENERAL-SEARCH(*problem*, QUEUING-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 ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))
  end

Stratégies de recherche

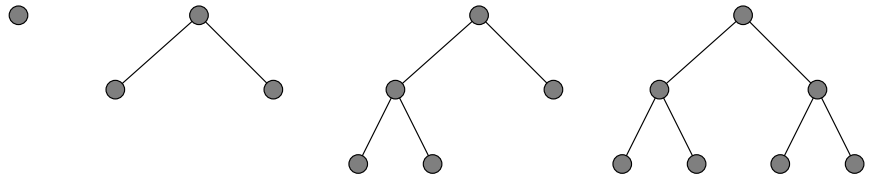
- Une stratégie est définie en choisissant un ordre dans lequel les états sont développés.
- Les stratégies sont évaluées selon les critères suivants:
  - Complétude
  - Complexité en temps
  - Complexité en espace
  - Optimalité
- La complexité en temps et en espace est mesurée selon
  - b : facteur de branchement maximal de l'arbre de recherche
  - d : profondeur de la meilleur solution
  - m : profondeur maximale de l'espace d'états (peut être ∞)

Stratégies de recherche non-informés (aveugles)

- Les stratégies aveugles utilisent seulement l'information disponible dans la définition du problème
- Recherche en largeur d'abord
- Recherche à coût uniforme
- Recherche en profondeur d'abord
- Recherche en profondeur limité
- Recherche en profondeur itérative (RPI)

Recherche en largeur d'abord

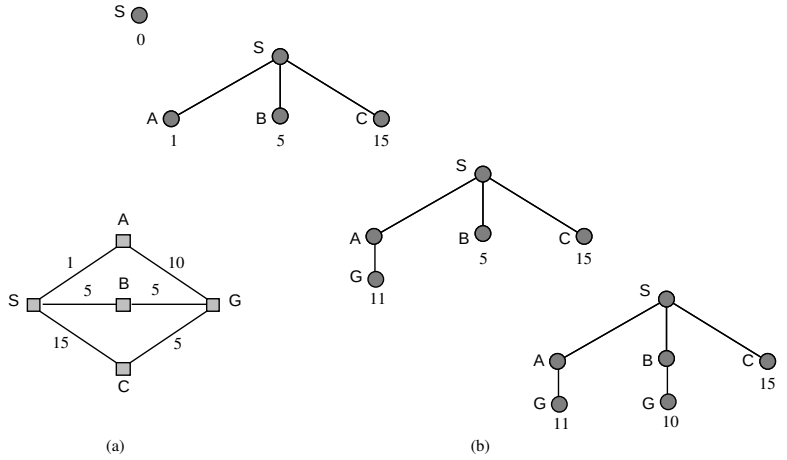
- QUEING-FN = met les successeurs en fin de liste



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

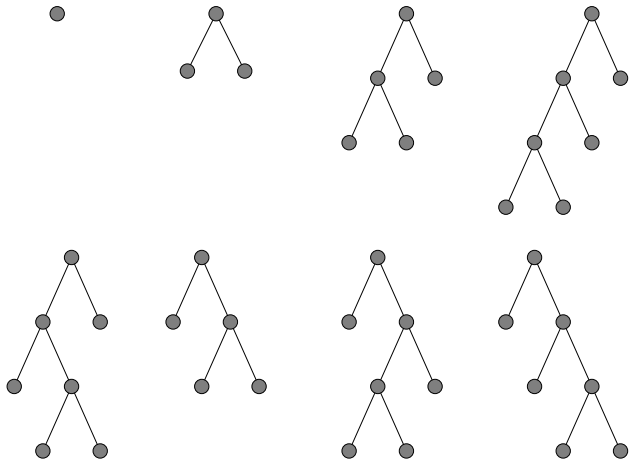
Recherche à coût uniforme

QUEING-FN = insérer les nœuds dans l'ordre de leur coût de chemin



Recherche en profondeur d'abord

QUEING-FN = met nœuds au début de la liste



Recherche en profondeur d'abord

- non complet: dans espace d'états infini ou avec boucle.
  - On peut ajouter test pour éviter des répétitions
- Temps: $O(b^m)$
- Espace: $O(bm)$
- non optimale

Recherche en profondeur limité

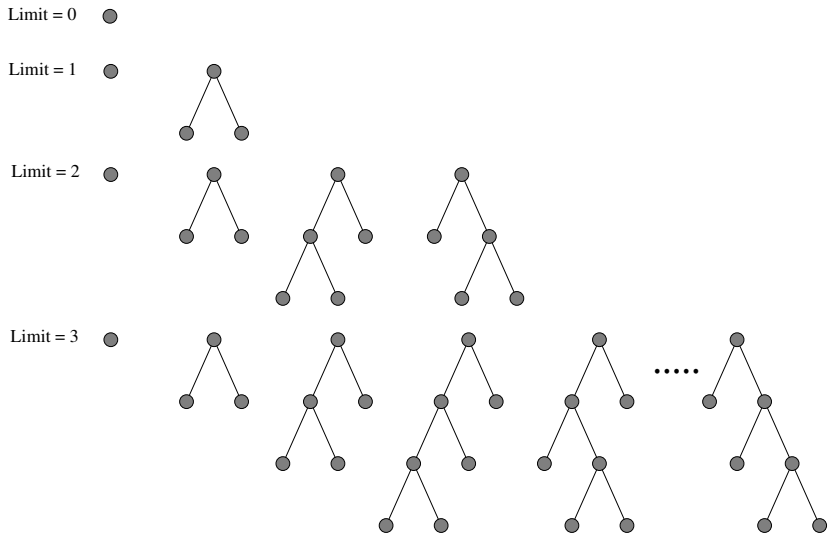
Recherche en profondeur d'abord avec limite l de profondeur

Recherche en profondeur itérative

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

for *depth* $\leftarrow$ 0 **to** ∞ **do**
if DEPTH-LIMITED-SEARCH(*problem*, *depth*) succeeds **then return** its result
end
return failure

Recherche en profondeur itérative

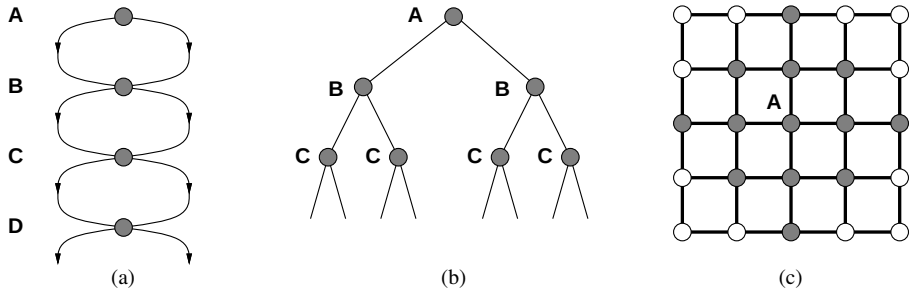


Recherche en profondeur itérative

- complet
- Temps: $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots b^d = O(b^d)$
- Espace: $O(bd)$
- optimale, si coût $= 1$ pour chaque pas
  - peut être adapté sinon

Éviter des états répétés

- 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

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