Résolution de problème par exploration:
Partie1: Formalisation d'un problème

Master 1 SID

Module CRP

Enseignant : Dr H.Belleili

Plan

- Étapes de résolution
- Formulation du problème
- Exemples: problèmes jouets
  - L'aspirateur
  - Jeu du taquin
  - Les 8 reines
- Exemples: Problèmes du monde réel
  - Problème de recherche d'un trajet
  - é
- Stratégies d'exploration
  - Définition
  - Évaluation d'une stratégie
- exercice

Étapes de résolution

1. Formulation d'un but : un ensemble d'états à atteindre.
2. Formulation du problème : les états et les actions à considérer.
3. Recherche de solution : examiner les différentes séquences d'actions menant à un état but et choisir la meilleure.
4. Exécution : accomplir la séquence d'actions sélectionnée.

Formulation du problème

- Un problème sera défini par les 5 éléments suivants :
- 1. un état initial
- 2. un ensemble d'actions
- 3. une fonction successeur, qui définit l'état résultant de l'exécution d'une action dans un état
- 4. un ensemble d'états buts
- 5. une fonction coût, associant à chaque action un nombre non-négatif (le coût de l'action)

Formulation du problème

- un problème est représenté comme un graphe orienté où les noeuds sont des états accessibles depuis l'état initial et où les arcs sont des actions.
- Nous appellerons ce graphe l'espace des états.
- Une solution sera un chemin de l'état initial à un état but.
- On dit qu'une solution est optimale si la somme des coûts des actions du chemin est minimale parmi toutes les solutions du problème.

Exemple de problème d'exploration

- On distingue 2 types de problèmes:
  - Des problèmes dits jouets: servent à illustrer ou à expérimenter des méthodes de résolution de problèmes
  - Des problèmes du monde réel: ce sont des problèmes qui intéressent vraiment les gens.

Problèmes jouets (exemple 1)

- Agent aspirateur (2 emplacements)
- Formulation:
  - États : l'agent peut être dans l'un des deux emplacements, les emplacements peuvent contenir ou pas de la poussière (2 x 2^2 états possibles),
  - État initial: n'importe quel état peut être pris comme état initial,
  - Fonction successeur: elle génère les états pouvant résulter de l'exécution des trois actions (gauche, droite, aspirer)
  - Test de l'état final: il vérifie que tous les carrés sont propres
  - Coût du chemin: le coût de chaque étape est 1, le coût du chemin est égal au nombre d'étapes que composent le chemin.
- Hypothèses: emplacements discrets, les saletés discrètes, un nettoyage fiable, jamais resali une fois nettoyé.

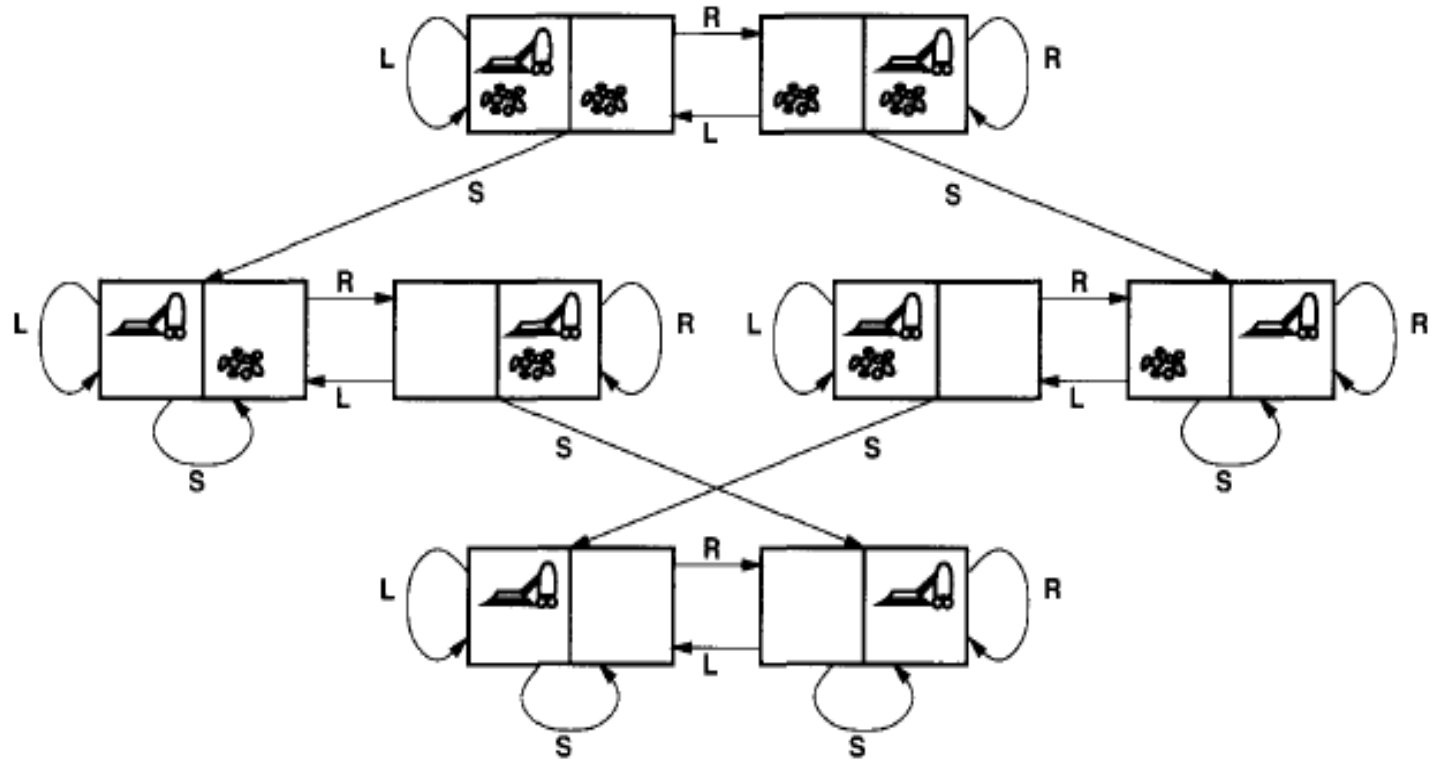
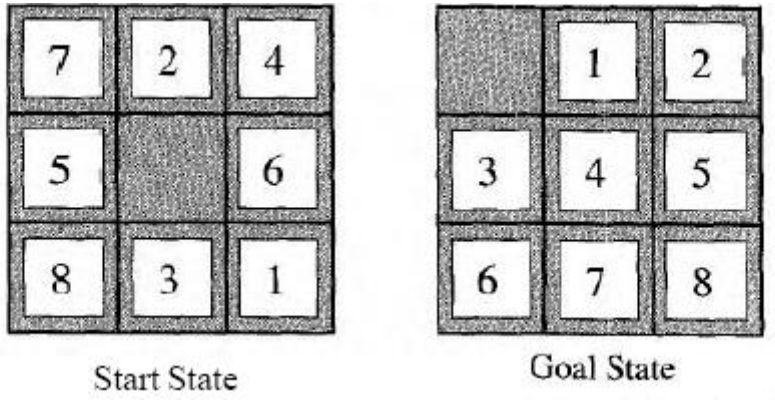


Figure 3.6 Diagram of the simplified vacuum state space. Arcs denote actions. L = move left, R = move right, S = suck.

Exemple 2: le jeu de taquin à 8 pièces

- Se compose dôun plateau 3 x 3 cases dont huit sont occupés par des numéros et une case est vide



Le taquin est souvent utilisé pour tester les algorithmes de recherche.

Taquin : Formulation

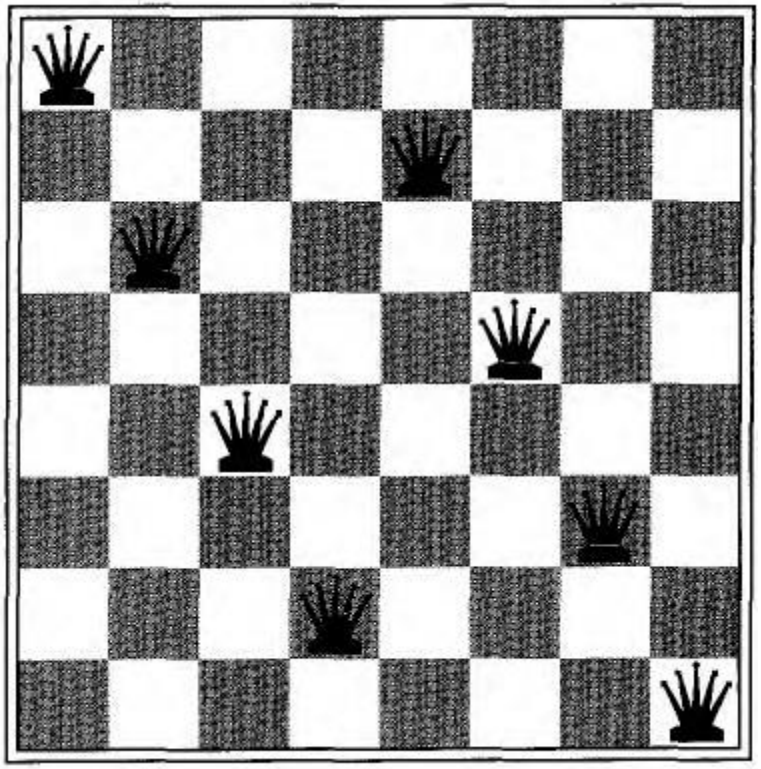
- Etats : Ce sont des configurations des huit tuiles dans les neuf cases de la grille.
- Etat initial : N'importe quel état pourrait être choisi comme l'état initial.
- Actions: Il y aura 4 actions possibles correspondant aux quatre façons de changer la position du carré vide : haut, bas, gauche, droite
- Fonction de successeur : Cette fonction spécifie les états résultants des différentes actions.
- Test de but: L'état but est unique et fixé au début du jeu
- Coût des actions: Chaque déplacement d'une tuile a un coût de 1 (pour trouver une solution avec le moins de déplacements).

Taquin 8 pièces (suite)

- Classe des problèmes NP-complets,
- Le problème à 8 pièces compte 9!/2=181 440 états possibles et est facilement résolu,
- Avec 15 pièces (16 cases) ~ 1,3 milliard d'états accessibles on arrive à le résoudre en quelques millièmes de seconde
- Avec 24 pièces ~ 10^{25} états accessibles et il est essez difficile à résoudre d'une manière optimale

Le problème des huit reines

- Objectif: placer huit reines sur un échiquier (une grille 8 × 8) tel quôaucune reine attaque une autre reine,(il nôy a pas deux reines sur la même colonne, la même ligne, ou sur la même diagonale).



8 reines: Formulation

- Etats : Toute configuration de 0 à 8 reines sur la grille.
- Etat initial: La grille vide.
- Actions : Ajouter une reine sur nôimporte quelle case vide de la grille.
- Fonction de successeur : La configuration qui résulte de lâajout dôune reine à une case spécifiée à la configuration courante.
- Test de but: Une configuration de huit reines avec aucune reine sous attaque.
- Coûts des actions: Ce pourrait être 0, ou un coût constant pour chaque action - nous nous intéressons pas au chemin, seulement lôétat but obtenu.
- Questions:
  1. quel est le nombre dôétats possibles avec cette formulation?
  2. Existe-t-il une manière de réduire ce nombre dôétat?

Taquin vs 8 reines

- Ces deux problèmes sont de nature assez différentes.
- Avec le taquin, nous savons depuis le début quel état nous voulons, et la difficulté est de trouver une séquence d'actions pour l'atteindre.
- le problème des huit reines, nous ne sommes pas intéressés par le chemin mais seulement par l'etat but obtenu.
- Ces deux jeux sont des exemples de deux grandes classes de problèmes étudiés en IA : des problèmes de planification et des problèmes de satisfaction de contraintes.

Problèmes du monde réel

- Problème de recherche d'un trajet,
- Le problème du voyageur du commerce (TSP) qui une variante du problème de recherche d'un itinéraire dans lequel chaque ville doit être visitée une seule fois
- Navigation d'un robot une généralisation du problème de recherche d'itinéraire (déplacement dans un espace continu ayant un ensemble infini d'actions et d'états possibles)
- Ordonnancement automatique d'assemblage d'objets complexes (le robot FREDDY)
- Conception de protéine de synthèse (bioinformatique)
- Recherches internet

Exemple

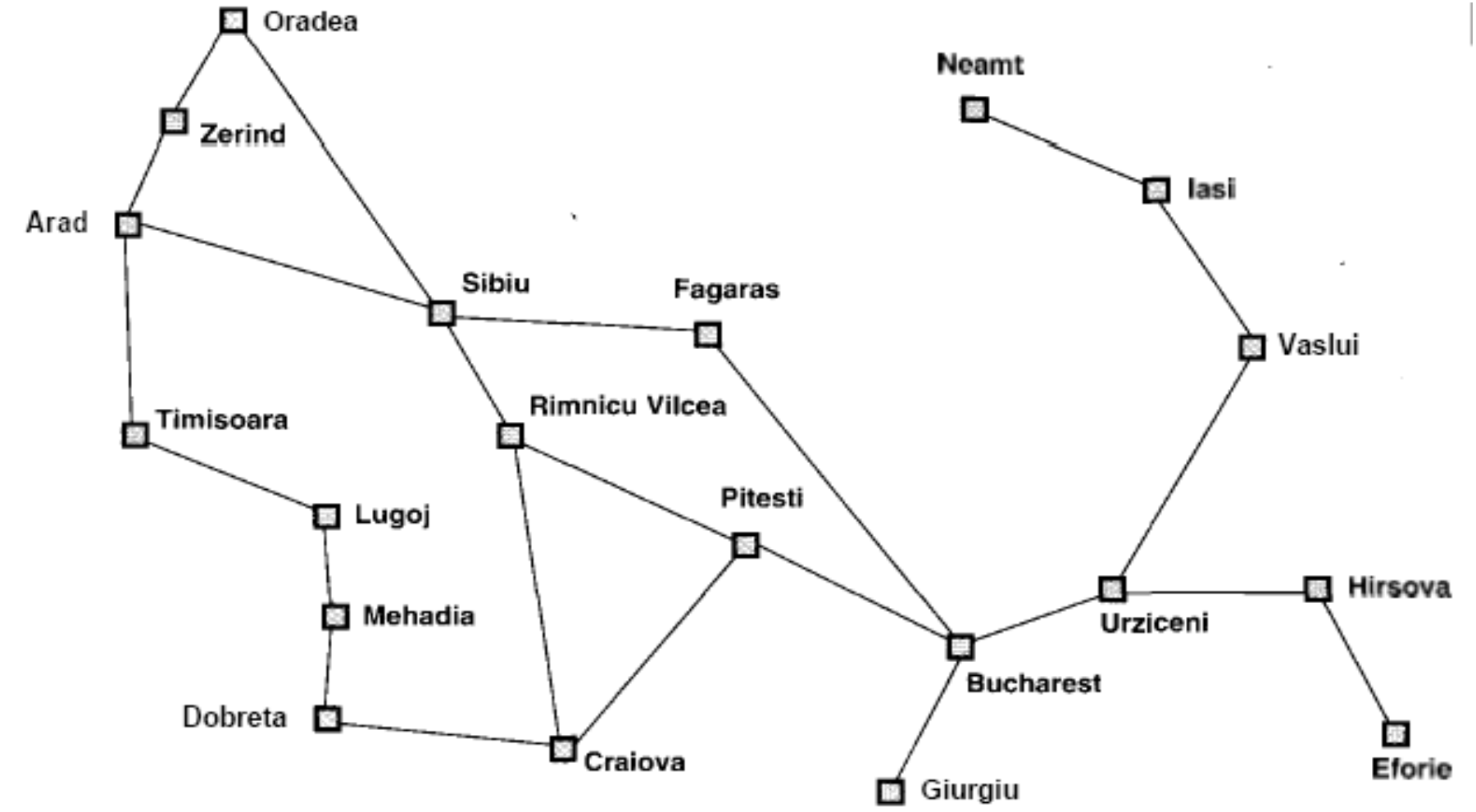


Figure 3.3 A simplified road map of Romania.

Exemple

Å On est à Arad et on veut aller à Bucharest

- ï But: être à Bucharest
- ï Problème:
  - ï États: villes
  - ï Actions: aller dûune ville à une autre.
- ï Solution: Une séquence de villes.
  - ï Ex: Arad, Sibiu, Fagaras, Bucharest
  - ï Environnement très simple
- ï statique, observable, discret et déterministe

Formulation du problème

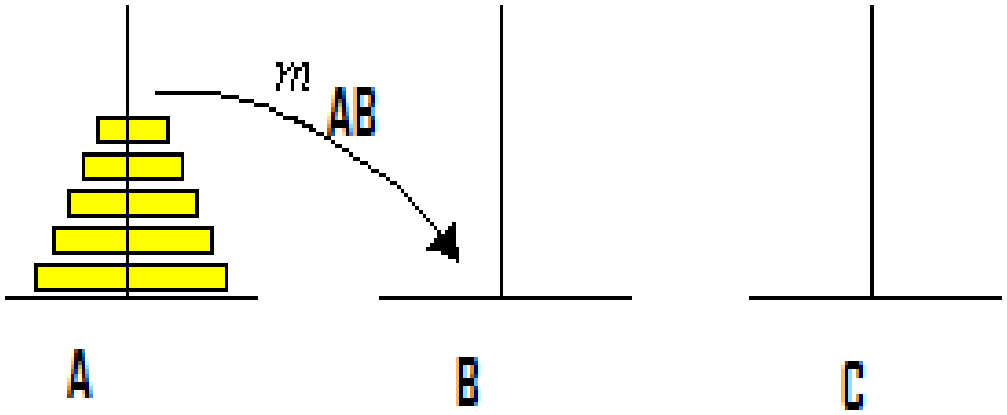
- État initial : lœétat *x* dans lequel se trouve lôagent
  - ***Dans*(*Arad*)**
- Actions : Une fonction *S*(*x*) qui permet de trouver lœensemble des états successeurs, sous forme de pair (action, successeur)
  S(*Dans* (*Arad*)) = {((*Aller*(*Sibiu*), *Dans(Sibiu*)), ((aller(Timisoara), Dans(Timisoara)), é .} etc.

- Lœétat initial et la fonction successeur définissent ensemble **lœespace des états du problème (états accessibles à partir de lœétat initial)**
- Test de but : un test afin de déterminer si le but est atteint
  - But: {*Dans (Bucharest*)}

- Coût du chemin : noté C(x,a,y) . une fonction qui assigne un coût à un chemin. Elle reflète la mesure de performance de lôagent. La valeur est non négative.

Exercices

1. On considère le problème des Tours de Hanoi avec trois tours. On demande de déplacer les disques situés sur la première tour sur une autre, de façon qu'un disque repose toujours sur un autre disque de taille supérieure ou sur une tour libre.



Exercice (suite)

- 2- Une chèvre, un chou et un loup se trouvent sur la rive dû un fleuve ; un passeur souhaite les transporter sur lâ autre rive mais, sa barque étant trop petite, il ne peut transporter quû un seul dê entre eux à la fois. Comment doit-il procéder afin de ne jamais laisser ensemble et sans surveillance le loup et la chèvre, ainsi que la chèvre et le chou ?
- 3- On souhaite prélever 4 litres de liquide dans un tonneau. Pour cela, nous avons à notre disposition deux récipients (non gradués !), lû un de 5 litres, lâ autre de 3 litres... Comment doit-on procéder ?

Partie 2: Recherche aveugle (non-informée) de solutions

Fonction AGENT-SIMPLE-RESOLUTION-PROBLEME (*percept*)
retourner une action

entrées : *percept*, un percept
Variables statiques : *seq*, une séquence dôactions, initialement vide
état, une description de lôétat courant du problème
but, un but, initialement nul
problème, la formulation dôun problème

état := ACTUALISER-ETAT (*état*, *percept*)
si *seq* est vide alors faire
    but := FORMULER-BUT (état)
    seq := EXPLORER (problème)
    action := PERMIER (*seq*)
    seq := RESTE(*seq*)
retourner *action*

Introduction

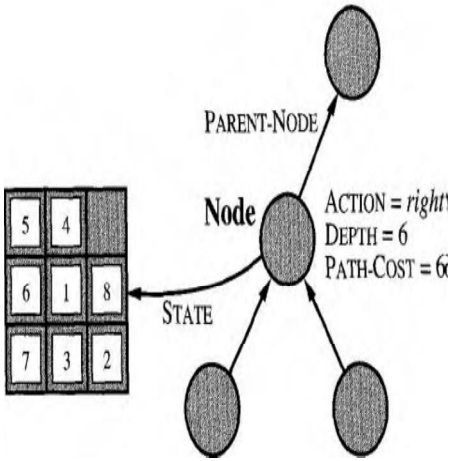
- Après avoir formulé le problème, il faut le résoudre.
- On effectue une exploration de l'espace des états
- L'exploration se fait soit
  - Dans un arbre généré explicitement par l'état initial et la fonction successeur
  - Soit dans un graphe lorsque le même état peut être atteint à partir de plusieurs chemins

Évaluation des algorithmes de recherche

- Dans la suite, nous allons voir différents dâalgorithmes de recherche.
- Comment pouvons-nous les comparer ?
- Voici quatre critères que nous allons utiliser pour comparer les différents algorithmes de recherche :
  - **Complexité en temps** : Combien de temps prend lâalgorithme pour trouver la solution ?
  - **Complexité en espace** : Combien de mémoire est utilisée lors de la recherche dâune solution ?
  - **Complétude:** Est-ce que lâalgorithme trouve toujours une solution sâil y en a une ?
  - **Optimalité:** Est-ce que lâalgorithme renvoie toujours des solutions optimales ?

Recherche de solutions dans un arbre

- Simuler lœexploration de lœespace dœétats en générant des successeurs pour les états déjà explorés.
- Noeud de recherche
  - État : lœétat dans lœespace dœétat.
  - Noeud parent : Le noeud dans lœarbre de recherche qui a généré ce noeud.
  - Action : Lœaction qui a été appliquée à lœétat du nî ud parent pour générer lœétat de ce noeud.
  - Coût du chemin : Le coût du chemin à partir de lœétat initial jusquœà ce noeud : *g(n)*
  - Profondeur : Le nombre dœétapes dans le chemin à partir de lœétat initial.



Nî ud vs état

- Il est important de distinguer entre les nî uds et les états
- Un nî ud est une structure de données de mémorisation qui est utilisé pour représenter lœarbre de recherche
- Un état correspond à une configuration du monde
- DONC:
  - Les nî uds sont sur des chemins particuliers, ça nœest pas le cas des états
  - Deux nî uds différents peuvent contenir le même état (si cet état est généré via deux chemins différents)

Principe Exploration

- Commencer par le nî ud racine
- Développer (expand) le nî ud en lui appliquant la fonction successeur
- On obtient tous les nî uds successeurs ces nî uds sont dits des nî uds générés
- Les nî uds générés mais non encore développés sont mis dans une structure de données appelée frontière
- Chaque élément de la frontière est un nî ud feuille (ie sans les nî uds successeurs)
- La stratégie dœexploration est la fonction qui sélectionne le nî ud suivant à développer dans cet ensemble de nî uds de la frontière.
- La fonction dœexploration doit examiner chacun des éléments de la frontière afin de choisir le meilleur (à développer)

Structure générale dôun algorithme de recherche

- La plupart des algorithmes de recherche suivent à peu près le même schéma :
- Fonction Exploration-En-Arbre (problème, stratégie) retourne une solution, ou echec
  - **Initialiser lôarbre de recherche avec lôétat initial du problème**
  - **itérer**
    - **si** il nôy a plus de nîuds candidats à développer **retourner échec**
    - **choisir un nîud feuille (dans la frontière) à développer en appliquant la stratégie**
    - **si** le nîud choisi contient un état final **alors retourner la solution correspondante**
    - **sinon**, développer le nîud et ajouter les nîuds du résultat dans la frontière

Frontière

Å La Frontière est une file ayant des opérations:

- ï Créer file (élémenté)
- ï Vide(file) retourne vrai ou faux
- ï Premier(file) retourne le premier élément de la file
- ï Supp-premier(file) retourne le premier(file) et le supprime de la file
- ï Insérer (élément,file) insére un nî ud dans la file et retourne file
- ï Insérer-tout(éléments,file) insère un ensemble dœéléments dans la file et retourne la file résultante

Retour sur la mesure de performance

- La mesure de performance dûun algorithme de recherche dans un graphe est liée à la taille du graphe de lœespace dœétats
- On exprime la complexité de la recherche en fonction de 3 critères:
  - Le facteur de branchement b: cœest le nombre maximal de successeurs dûun nî ud donné
  - La profondeur du nî ud but le moins éloigné d
  - La longueur maximale dûun chemin dans lœespace des états m
- Complexité de temps: le nombre de nî uds générés pendant la recherche.
- Complexité dœespace: le nombre maximum de nî uds conservés en mémoire.

Stratégies de recherche

- Détermine l'ordre de développement des nœuds.
- Recherches non-informées : Aucune information additionnelle. Elles ne peuvent pas dire si un noeud est meilleur qu'un autre.Elles peuvent seulement dire si l'état est un but ou non.
- Recherches informées (heuristiques) : Elles peuvent estimer si un noeud est plus prometteur qu'un autre.

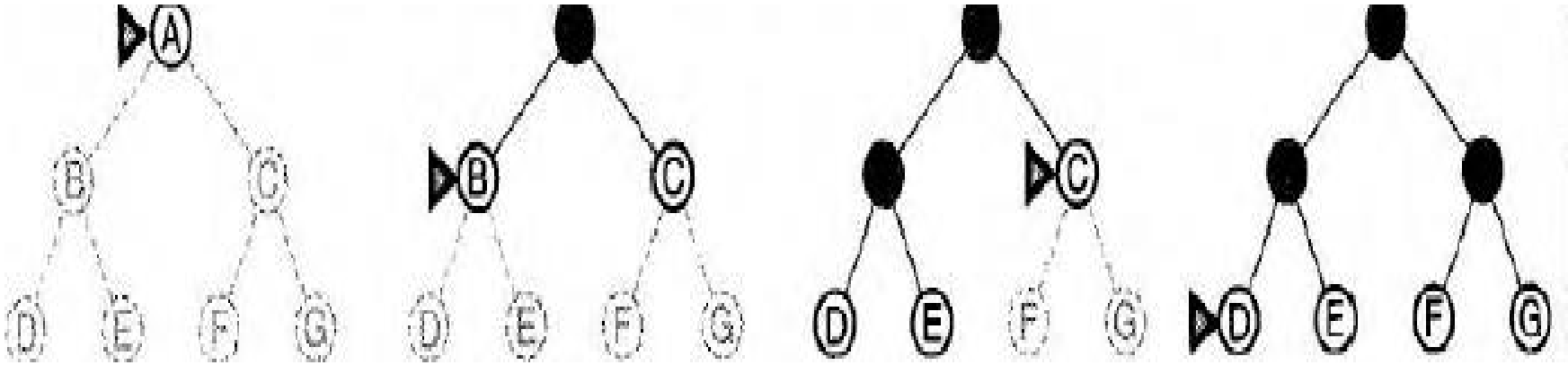
Stratégies de recherche non informées

- Largeur dôabord (Breath-first)
- Coût uniforme (Uniform-cost)
- Profondeur dôabord (Depth-first)
- Profondeur limitée (Depth-limited)
- Profondeur itératif (Iterative deepening)
- Recherche bidirectionnelle (Bidirectional search)

Parcours en largeur

- Le parcours en largeur est un algorithme de recherche très simple : nous examinons d'abord lœtat initial, puis ses successeurs, puis les successeurs des successeurs, etc.
- Tous les noeuds d'une certaine profondeur sont examinés avant les noeuds de profondeur supérieure.
- Pour implementer cet algorithme, il suffit de placer les nouveaux noeuds systématiquement à la fin de la liste de noeuds à traiter.
- La frontière est gérée en FIFO qui assure que les nî uds visités en premier seront développés dœabord
- Dans une file FIFO tous les successeurs nouvellement générés sont placés à la fin, ce qui a pour effet de développer les nî uds en surface avant ceux qui sont plus en profondeur.

Parcours en largeur (exemple)



Le parcours en largeur pour un arbre binaire simple. Nous commençons par l'état initial A, puis nous examinons les noeuds B et C de profondeur 1, puis les noeuds D, E, F, et G de profondeur 2.

Largeur dôabord

- Complètude: oui, si b est fini
- Optimalité :
  - Oui, Si le coût du chemin est une fonction non décroissante de la profondeur du nî ud alors la stratégie est optimale
  - non, le nî ud but le moins profond peut ne pas être optimal.
- La complexité en temps Nombre total de nî uds générés si tous les nî uds ont « b » successeurs et que le but soit à la profondeur « d » au pire cas

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

- La complexité en espace est la même que la complexité en temps: chaque nî ud généré doit rester en mémoire soit par parce quôil appartient à la Frontière non encore développé) soit parce quôil est un ancêtre dôun nî ud sur la frontière.

Largeur dâabord (suite)

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Facteur de branchement b=10, 10 000 nî uds/s et 1000 octets/nî ud

1 terabyte= 1000 Milliards dôctets

Largeur dôabord (conclusion)

- Les besoins en mémoire posent un problème plus important que le temps dôexécution
- Il est possible dôattendre 31 heures la solution mais il est difficile dôavoir un ordinateur de 1 térabyte de mémoire centrale
- Les problèmes dont la complexité est exponentielle ne peuvent pas être résolus par des méthodes non informées, sauf pour les petites instances

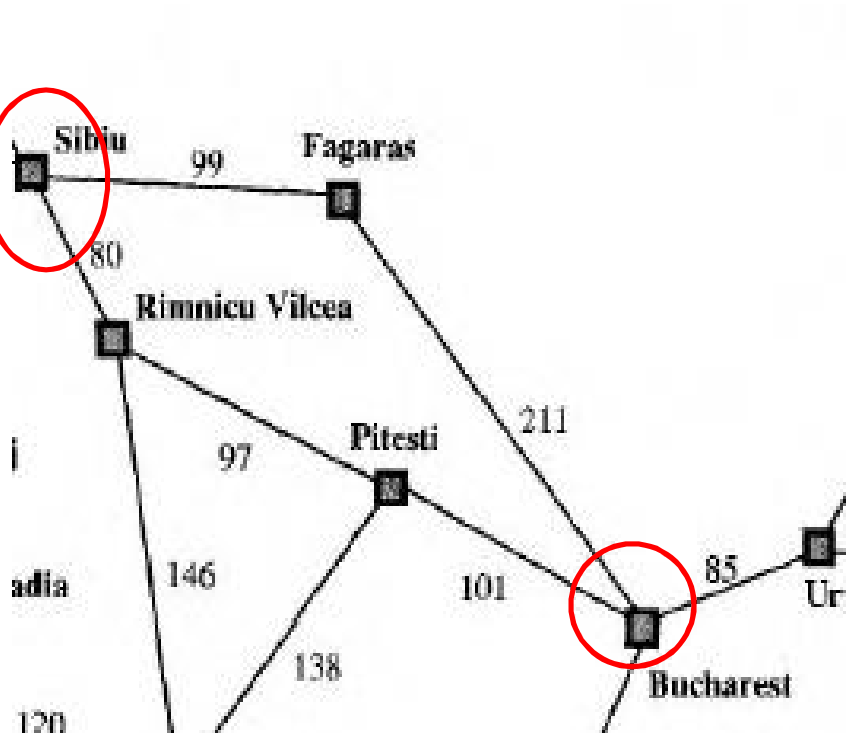
Coût uniforme

- Développe le noeud ayant le coût g(n) le plus bas.
- File triée selon le coût
- Équivalent à largeur d'abord si le coût des actions est toujours le même.
- L'exploration à coût uniforme ne tient pas compte du nombre d'étapes que compte un chemin mais uniquement de leur coût total

Exemple

- Succ(Sibiu) = Rimnicu Vilcea (80); Fagaras (99)
- Min(Succ(Rimnicu Vilcea)):
- Pitesti (80+97=177)
- On choisit fagaras (99<177)
- succ(fargaras): bucarest (99+211=310) **lœexploration continue**
- Min(Succ(pitesti)): bucarest (80+97+101=278)

- Le chemin
- Sibiu-RV-pitesti-Bucarest (278) < sibiu-fagaras-Bucarest (310)



Coût uniforme

- Questions:

1. si un nî ud a une action à coût nul qui le ramène au même état que se passe t-il ?
   Boucle infinie
2. À quelle condition la complétude est garantie?

Le coût de chaque étape doit dépasse une certaine constante positive ε

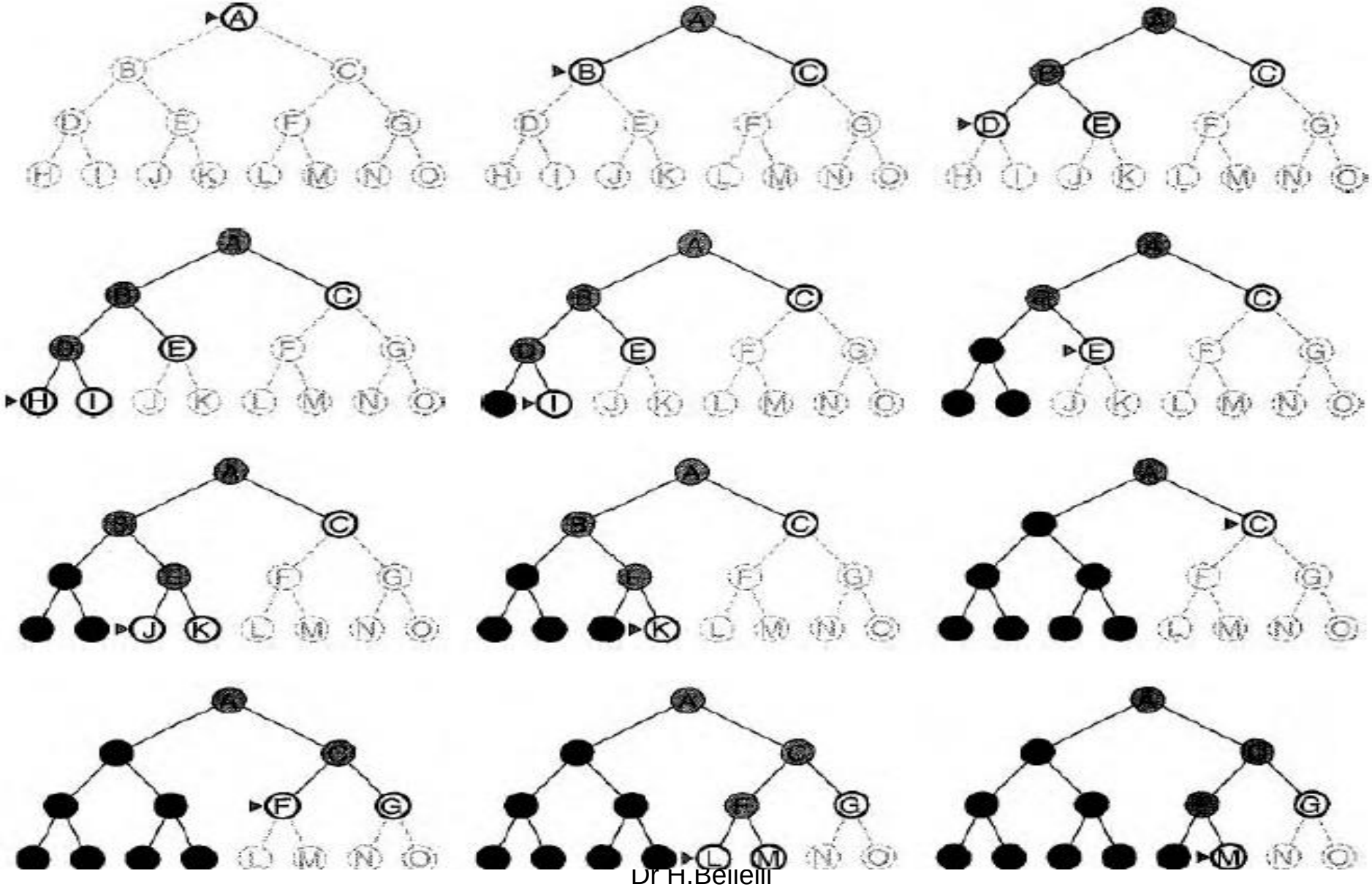
- Il est difficile de caractériser la complexité de cet algorithme en termes de b et d puisqu'il est guidé par le cout des chemins plutôt que par la profondeur.
- On pose C* la solution optimale et le coût minimum de chaque action est ε
- Alors dans le pire cas la complexité en temps et en espace est $(b^{1+\lfloor C^*/\varepsilon \rfloor}) >> b^d$
- Exploration CU explore de très grand nombre d'étapes avant d'explorer des chemins plus intéressants
- Lorsque les coût des étapes sont égaux $(b^{1+\lfloor C^*/\varepsilon \rfloor}) = b^d + 1$ semblable à largeur d'abord seulement que cette dernière s'arrête dès qu'elle génère un but et l'autre examine tous les à la profondeur du but les moins couteux

- **L'exploration à CU fait plus de travail en développant inutilement des noeuds de profondeur d**

Profondeur dôabord (1)

- Développe le noeud le plus profond.
- Implémenté à lôaide dôune pile (LIFO). Les nouveaux noeuds générés vont sur le dessus.

Profondeur dâabord (2)



Profondeur dôabord (3)

- Complétude: si la profondeur dôun sous arbre nôest pas limitée et sôil nôy a pas de solution lôexploration ne se terminera jamais.
- Optimalité: non optimal (pourquoi?)
- Compléxité en temps: $O(b^m)$, très mauvais si m est plus grand que d, (m profondeur maximale)
- Complexité en espace mémoire : necessite de conserver en mémoire que bm+1 nî uds. Soit une complexité $O(bm)$, linéaire

Exploration en profondeur limitée (DLS)

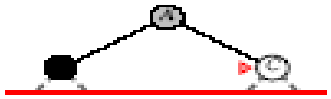
- On peut remédier au problème des arbres infinis en spécifiant une profondeur déterminée *l* et en imposant une limite de profondeur à l'exploration en profondeur d'abord
- Les nœuds situés à la profondeur *l* seront traités comme s'ils n'avaient pas de successeurs
- La connaissance du problème permet de déterminer la limite de profondeur
- Si *l* < d incomplétude
- Complexité en temps $O(b^l)$
- Complexité en espace est $O(bl)$

Exploration itérative en profondeur (IDS)

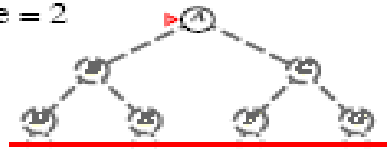
- Combine les avantages de largeur dôabord (complète et optimale) et ceux de profondeur dôabord (complexité en espace).
- Principe:
  - Profondeur limitée, mais en essayant toutes les profondeurs: 0, 1, 2, 3, é
  - Évite le problème de trouver une limite pour la recherche profondeur limitée .

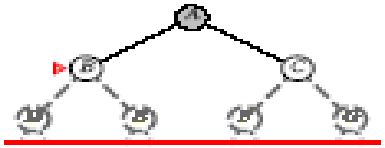
Bome = 0

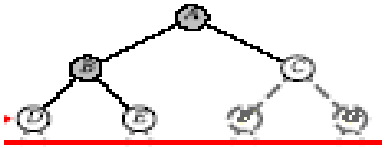
Bome = 1

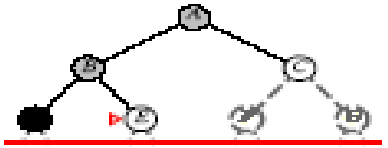


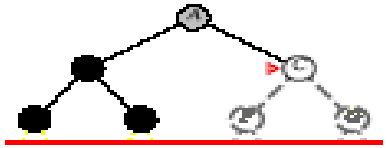
Bome = 2

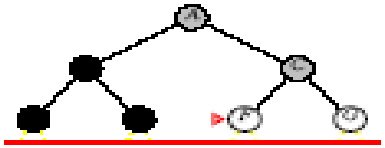


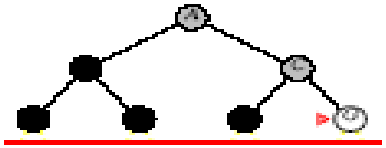


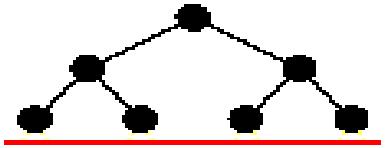




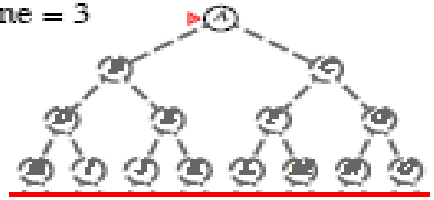


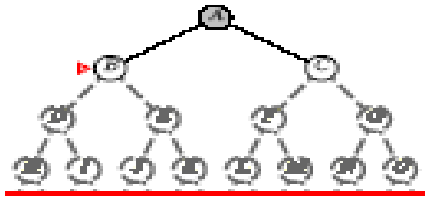


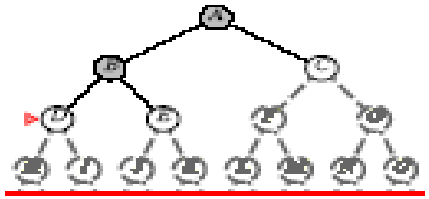


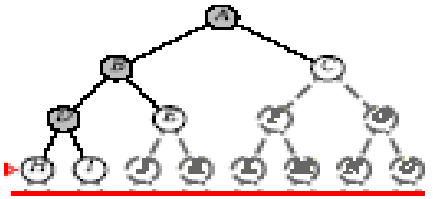


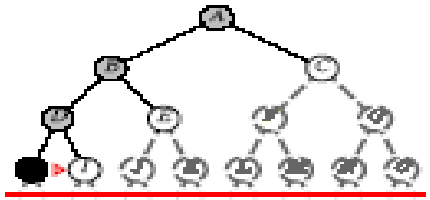
Bome = 3

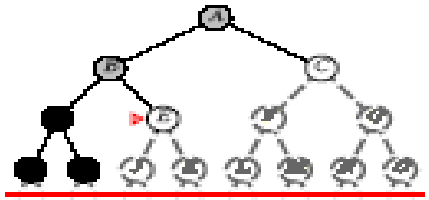


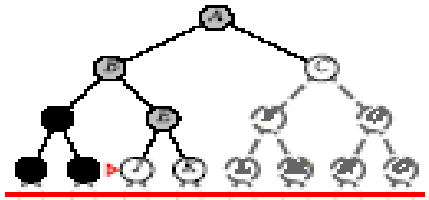


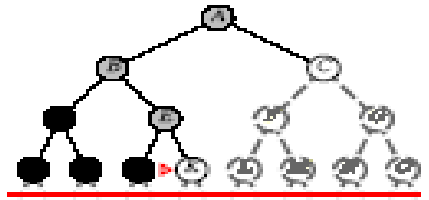


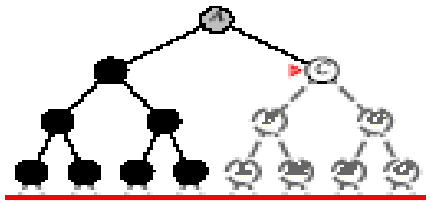


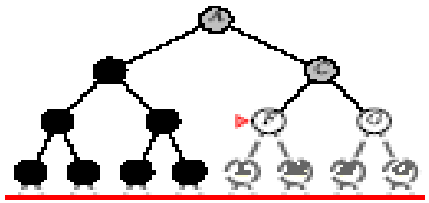


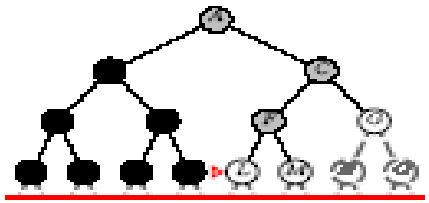


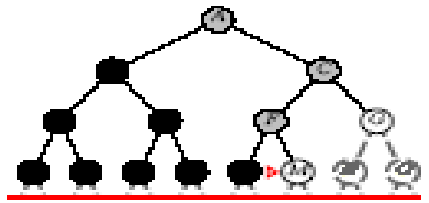












Propriétés IDS

- Peut paraître dispendieuse, les états étant générés plusieurs fois, ça n'est pas le cas.
- À chaque niveau la majorité des nœuds sont situés au niveau le plus bas
- Donc le fait de générer plusieurs fois les niveaux supérieurs a un impact négligeable

Complexité IDS

- Complexité en temps Les nî uds du niveau le plus bas (prof. d) sont générés une seule fois (d)b, ceux du niveau immédiatement supérieur sont générés 2 fois (d-1)bé jusquâaux enfants du nî ud racine qui sont générés d fois : N(IDS)= db^1+ $(d-1)b^2$+é . +b^d= $O(b^d)$
- Complexité en espace: $O(bd)$
- Complétude: Oui
- Optimal? Oui, si le coût de chaque action est de 1.
- IDS est utilisée lorsque le nombre dœtats est important et que la profondeur de la solution est inconnue

Comparaison IDS, BFS

- Nombre de nî uds générés
- En largeur dâ abord:
- N(BFS)= $b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b)$
- Itérative en profondeur:
- N(IDS)= $db^1 + (d-1)b^2 + \text{é} \ . \ + b^d$

Pour b=10 et d=5

N(BFS)=10+100+1000+10000+100000+999999=1 111 100

N(IDS)=50+400+3000+20000+100000=123 450

Exploration bidirectionnelle

- Exécuter deux explorations simultanées:
  - L'une en aval depuis l'état initial
  - Et l'autre en amont à partir du but
  - Puis d'arrêter lorsque elles se rencontrent au milieu
- Intêret: $b^{d/2} + b^{d/2}$ est très inférieur à b^d

- Principe: l'une ou l'autre des deux recherches vérifient chaque nî ud avant de le développer afin de voir s'il appartient à la frontière de l'autre arbre; si c'est le cas la solution a été trouvée
- Exemple: pour d=6 , on exécute une exploration en largeur dans chaque sens alors dans le pire des cas, les deux explorations se rejoignent lorsque chacune aura développé tous les nî uds sauf un situé à la profondeur 3

Exploration bidirectionnelle

- La diminution de la complexité en temps rend lœexploration bidirectionnelle attrayante,
- Le problème est comment effectuer lœexploration en arrière
- Pas si simple!!!
- Supposons que les prédécesseurs dœun nî ud n, Pred(n) soient tous les états ayant n pour successeur
- Lœexploration bidirectionnelle nécessite que Pred(n) soit calculable
- Le cas le plus facile est celui où toutes les actions de lœespace des états sont réversibles, ie Pred(n)=Succ(n)
- Dans les autres cas il faut faire preuve dœingénioité

Exploration bidirectionnelle

- Pour b=10, il en résulte 22 000 nœuds générés à comparer avec 11 111 100 nœuds d'une recherche en largeur standard
- Complexité en temps: $O(b^{d/2})$
- Complexité en espace: $O(b^{d/2})$
- Complétude: oui, Optimalité: oui si les deux explorations sont en largeur d'abord
- D'autres combinaisons sont susceptibles de sacrifier la complétude ou l'optimalité ou les deux à la fois

Exploration bidirectionnelle

- Lorsque il y a un seul but (problème du taquin ou recherche dôun itinéraire) il nôy a quôun état but: lôexploration en arrière est similaire à lôexploration en avant
- Sôil y a plusieurs état buts que faut-il faire?
- Le cas le plus difficile: cas du jeu dôechec ou il y a plusieurs états qui satisfont au test « échec et mat »

Comparaison des stratégies d'exploration non informée

Critère	Largeur d'abord	Coût uniforme	Profondeur d'abord	Profondeur limitée	Itérative en profondeur	Bidirectionnelle (si applicable)
Complète	oui[1]	oui[1,2]	non	non	oui[1]	oui[1,4]
Temps	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Espace	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimale	oui[3]	oui	non	non	oui[3]	oui[3,4]

b : facteur de branchement
d : profondeur de la solution la moins profonde
m : profondeur maximale de l'arbre de recherche
l : profondeur limite
[1] si b est fini
[2] si les coûts des étapes sont $\geq \varepsilon$ avec ε positif
[3] si les coûts d'étapes sont tous identiques
[4] si les deux directions utilisent un exploration en largeur d'abord

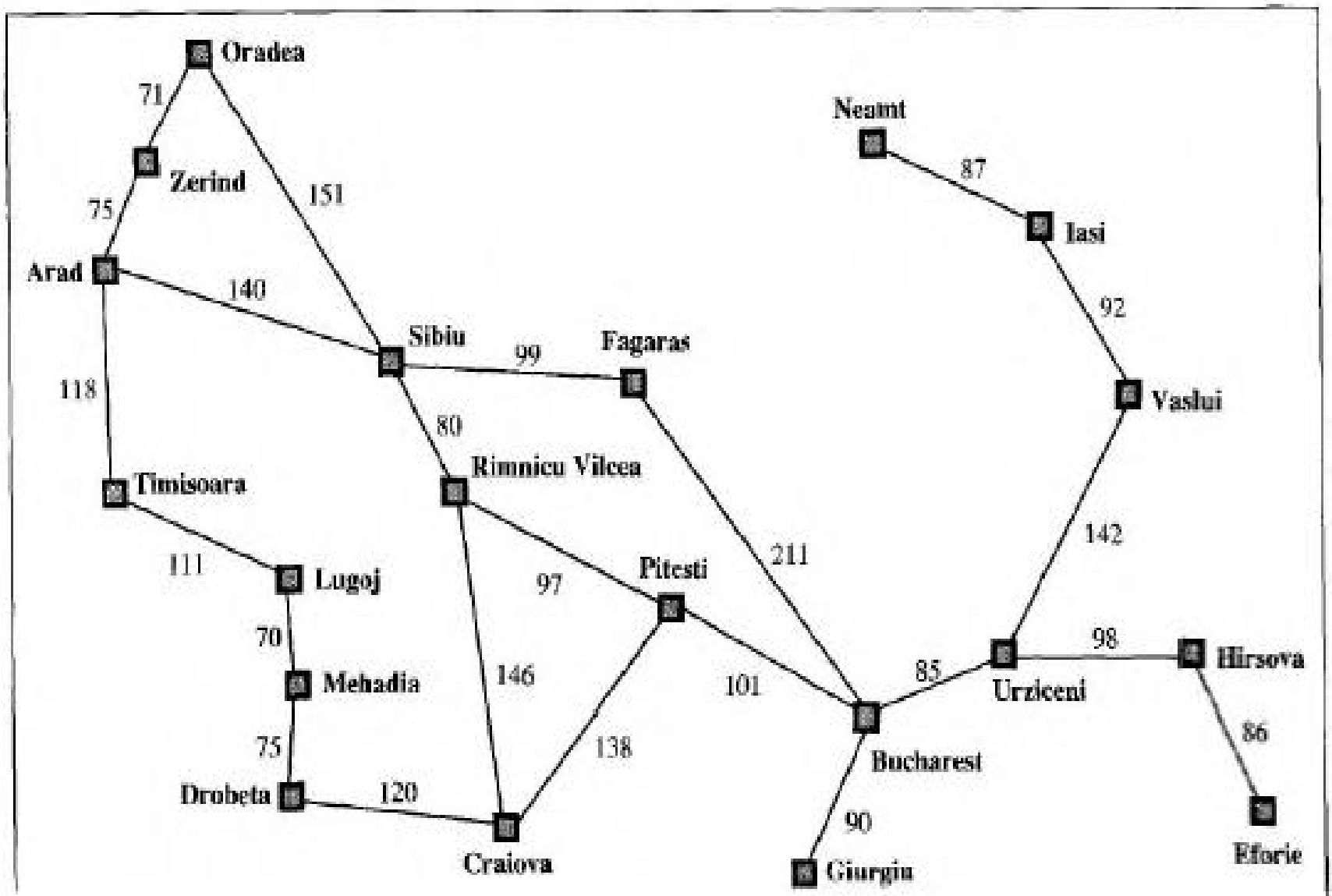
Répétitions dœétats

- Possibilité de perdre du temps à développer des états déjà rencontrés et développés
- Parfois inévitable (actions réversibles)
- exemple dans la détermination dôun itinéraire ou dans le taquin ou les arbres sont infinis
- Solution: restreindre lôarbre à une profondeur déterminée
- Lœélimination des répétition dœétats entraine une réduction exponentielle du coût de lœexploration
- Les répétition dœétats peuvent rendre un problème insoluble en un problème soluble
- Comment? comparer le nî ud sur le point dœêtre développer à ceux déjà développés
- Cela revient à utiliser plus de mémoire pour gagner en temps:
- *Les algorithmes qui oublient leur histoire sont condamnés à la répéter*

Algorithme de recherche dans un graphe

function GRAPH-SEARCH(*problème*)
noeuds_fermés Ŷ Ø
noeuds_ouverts Ŷ {CRÉÉR-NOEUD-ÉTAT-INITIAL(*problème*)}
loop
if VIDE(*noeuds_ouverts*) **then return** *échec*
noeud Ŷ CHOISIR(*noeuds_ouverts*)
if TEST-ÉTAT-FINAL(*problème*,ÉTAT(*noeud*)) **then**
return SOLUTION(*noeud*)
if *noeud* ∉ *noeuds_fermés* **then**
noeuds_fermés Ŷ *noeuds_fermés* ∪ {*noeud*}
noeuds_ouverts Ŷ *noeuds_ouverts* ∪
DÉVELOPPER(*noeud*,*problème*)

Fin
Exploration non informée


Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Pitesti
Lugoj
97
98
70
Hirsova
Mehadia
146
101
85
Urziceni
75
86
138
Bucharest
Drobeta
120
90
Craiova
Eforie
Giurgiu