Intelligence Artificielle

GL4

Introduction

- Qu'est-ce que l'IA ? Difficulté de définir l'intelligence.

- But de l'IA: remplacer l'expertise humaine dans des tâches demandant un raisonnement.

Intelligence artificielle

- Terme créé par John McCarthy (aussi développeur du langage LISP) autour de 1956

> *Construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisantes par des êtres humains car elles demandent des processus mentaux de haut niveau tels que l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique. (Marvin Minsky)*

Le test de Turing

- Turing (1950) “Computing machinery and intelligence”

- Le test est créé pour donner une définition opérationnelle satisfaisante de l’intelligence

- Un ordinateur passe ce test si un homme, après avoir posé des questions écrites ne sait pas s’il s’adresse à un autre humain ou à une machine.

Exemples d'utilisation de l'IA

- Traduction automatique (par exemple Mandarin → Français)
- Identification d'objet dans des images (ex : chaises, visages, etc)
- Démontrer ou aider à démontrer de nouveaux théorèmes
- Assistants(médical, juridique, ex IBMWatson)
- Robots (aides aux personnes âgées, aides musées)
- Voitures autonomes
- Jouer au jeu de Go, au Pocker etc.

Tracks at IJCAI 2017

- **Machine Learning** (Classification, Feature Selection, Data Mining, Learning, Graphical Model, Active Learning, Relational learning, Time series and Data Stream, Classification, Kernel Methods, Deep Learning, Data Mining and Personalization, Semi supervised Learning, Reinforcement learning
- **Constraint Satisfaction** (Constraint Satisfaction 2, Solvers and tools,)
- **Uncertainty in AI** (Approximate Probabilistic inference 2, MDP)
- **Robotique** (Vision and Perception, motion and path planning, robotics and vision)
- **Search** (Search in planning and scheduling, Heuristic search, planning algorithms, Satisfiability, Activity and plan recognition)
- **Systèmes multiagents** (Agent theories, Cooperative Games, Noncooperative games, Economic Paradigms, Agent based simulations, etc)
- **Traitement automatique des langues** (Natural Language semantics, Natural Language processing, applications and tools, Discourse, Sentiment analysis and Text mining, Machine Translation)
- **Knowledge representation** (Common sense reasoning, automated reasoning and theorem proving, Description Logics and Ontologies, Geometric spatial and temporal reasoning, Computational Complexity of Reasoning, Belief change)

Plan

- Recherche dans les graphes d'états (recherche aveugle et recherche informée)
- Problème de satisfaction de contrainte (CSP)
- Jeux avec Minmax et élagage alpha-beta
- Jeux stratégiques (stratégies pures et mixte)
- Apprentissage d'arbres de décision (ID3)

Références

- Le livre référence pour ce cours est : "Artificial Intelligence: A Modern Approach" de Stuart Russel et Peter Norvig (éditions Pearson).

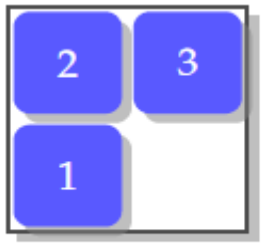
Résolution d’un problème grâce à la recherche dans un graphe

Principe général

- On représente un problème par un espace d'états (arbre/graphe).
- Chaque état est une configuration possible du problème.
- Résoudre le problème consiste à trouve un chemin dans le graphe.
  - Parcours aveugles non informés : profondeur, largeur.
  - Parcours informés.

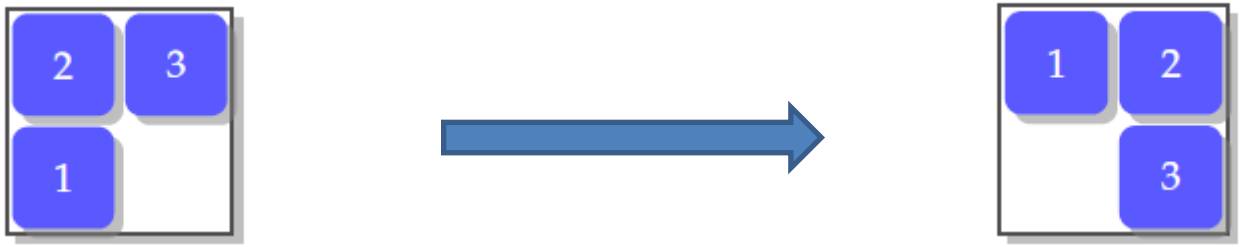
Exemple d'un jeu: le taquin

- Un taquin $n \times n$ est constitué de n^2-1 jetons carrés à l'intérieur d'un carré pouvant contenir n^2 jetons : on a donc une **case vide**.



taquin 2×2

- Les coups permis sont ceux qui font glisser dans la case vide un des 2, 3 jetons contigus.
- **But** : partir d'un état initial du taquin pour arriver à un état final du taquin.



Un programme peut-il résoudre ce type de problèmes pour n=2,3,4,5. . . ?

Autres exemples de jeux

- Poser n reines sur un échiquier de taille $n \times n$ sans qu'aucune reine n'attaque une autre reine.
- trouver un itinéraire d'un point A à un point B à l'aide d'une carte
- résoudre un "rubik's cube"

On peut aussi se poser des question **d'optimisation** :

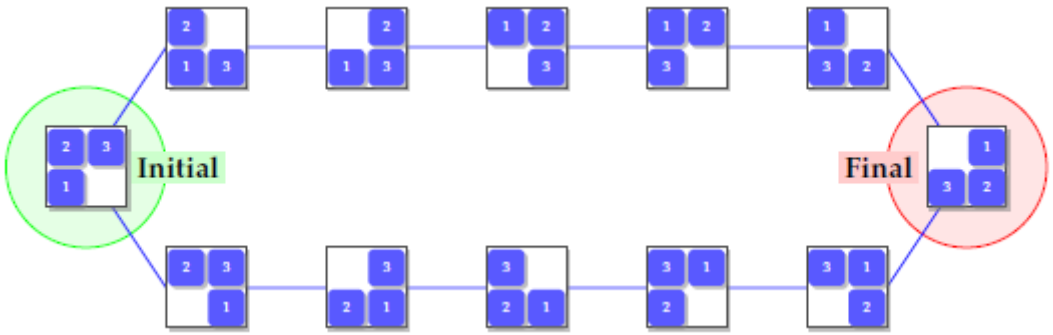
- résoudre le taquin en un minimum de coups
- trouver un itinéraire qui minimise le temps de parcours

Définition du problème : la modélisation

- 1ère étape : décider ce que sont les états et les actions.
- 2nde étape : bien définir le problème avec
  - un état de **départ**.
  - les **actions** possibles pour chaque état
  - le modèle de **transition** : c'est une fonction qui donne l'état qui résulte après avoir effectué une action depuis un état.

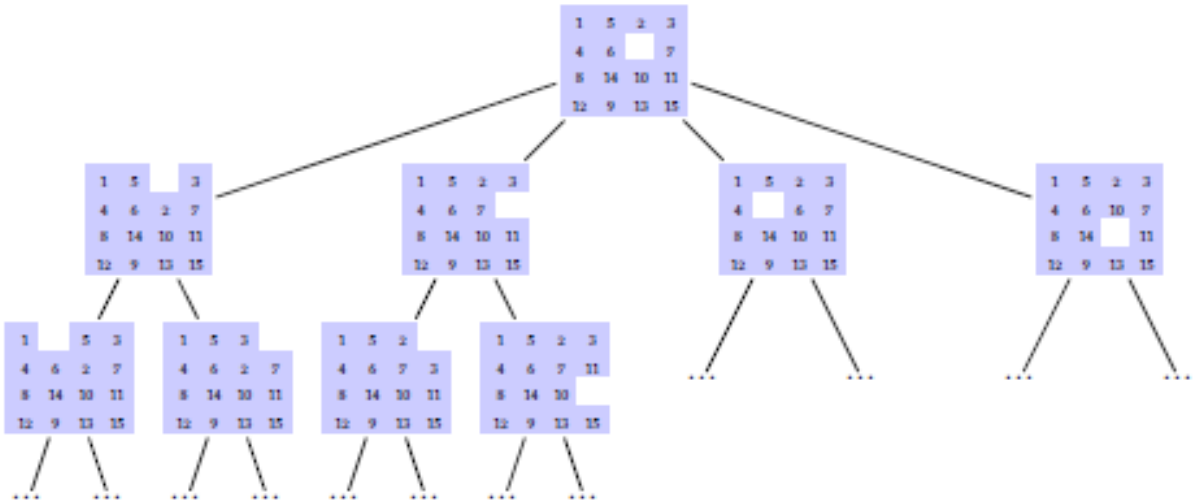
On peut représenter l'état de départ, les actions et le modèle de transition par :

- un **graphe**.
- une fonction qui teste si le but est **atteint**.
- éventuellement une fonction de coût pour chaque action dans chaque état.



Résolution

- Les noeuds/états du graphe sont les configurations possibles du jeu.
- Les arcs du graphe sont les actions possibles
- Il y a un état initial



- Trouver une solution consiste à trouver un chemin allant d'un état initial vers un état final (le but)

Remarques sur la résolution

- Pour expliquer un algorithme, on dessine un graphe **mais** lors de l'exécution de l'algorithme, tout le graphe n'est pas forcément stocké en mémoire → on utilise une liste d'états à examiner appelée "ouverts"

- On peut répéter des noeuds → on peut tourner en boucle (sauf si on se rappelle des états visités) ! → on utilise une liste d'états visités appelée « fermés»

- sans mémoire : **"tree-search"** risque que les états se répètent → risque de boucle

- avec mémoire : **"graph-search"** nécessite des ressources de stockage, évite les boucles !

- Résoudre un problème de taquin 2x2 a l'air facile !
- Considérons le taquin 4x4:
  - le nombre de sommets exactement est un entier proche de 2.10^{13}
  - Certains problèmes sont impossibles (il n'existe pas de chemin de l'état initial à l'état final). Pour le taquin 4x4, le graphe possède deux composantes connexes: si on tire au sort l'état initial et l'état final on a 50% de chance qu'une solution existe.
- Peut-on utiliser des algorithmes de la théorie des graphes ? Problème de stockage des nœuds en mémoire!
- Mais on peut utiliser des algorithmes du cours d'algorithmique

Principe général de la recherche

1. Démarrer la recherche avec la liste contenant l'état initial du problème.
2. Si la liste n'est pas vide alors :
   a. Choisir (à l'aide d'une stratégie) un état e à traiter.
   b. Si e est un état final alors retourner recherche positive
   c. Sinon, rajouter tous les successeurs de e à la liste d'états à traiter et recommencer au point 2.
3. Sinon retourner recherche négative.

Stratégie

- C'est un critère qui permet de choisir un ordre pour traiter les états du problème. On tiendra compte de :
  - La complétude
  - L'optimalité (selon le coût de chaque action).
  - La complexité (en temps et en espace) mesurée par :
    - b : branchement maximal de l'arbre de recherche
    - d : profondeur de la meilleure solution
    - m : profondeur maximale de l'espace d'états

Méthodes de recherche

- Recherche aveugle ou non-informée: Aucune information additionnelle. Elles ne peuvent pas dire si un nœud est meilleur qu'un autre. Elles peuvent seulement dire si l'état est un but ou non. Exemples: Largeur d'abord, profondeur d'abord, etc.

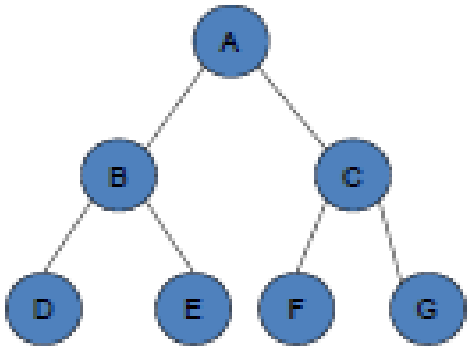
- Recherche informée: Elles peuvent estimer si un nœud est plus prometteur qu'un autre. Exemples: Meilleur d'abord, A*, algorithmes génétiques, etc.

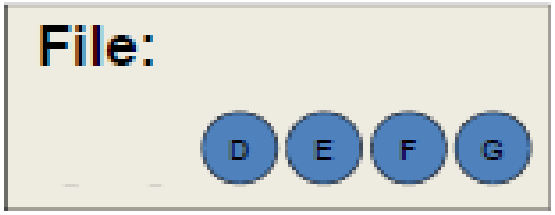
Algorithmes de recherche aveugle (ou non informée)

- expansion d'un graphe en largeur d'abord: **Breadth-first search (BFS)**
- expansion d'un graphe en profondeur d'abord: **Depth-first search (DFS)**
- expansion d'un graphe avec coût uniforme: **Uniform-cost search**
- développer le noeud qui a le coût le plus bas.
- recherche en profondeur limitée : **Depth-limited search**
- recherche en profondeur itérative: **Iterative deepening DFS**
- recherche bidirectionnelle : **Bidirectional search**

Recherche en largeur d'abord

- Développement de tous les noeuds au niveau *i* avant de développer les noeuds au niveau *i+1*
- Implémenté à l'aide d'une file; les nouveaux successeurs vont à la fin.





Ordre de visite: A – B – C – D – E – F - G

Propriétés de la recherche en largeur d'abord

- Complétude : oui, si *b* est fini
- Complexité en temps : $b^0 + b^1 + b^2 + b^3 + \ldots + b^d = O(b^d)$
- Complexité en espace : $O(b^d)$ (Garde tous les noeuds en mémoire)
- Optimal : non en général, oui si le coût des actions est le même pour toutes les actions

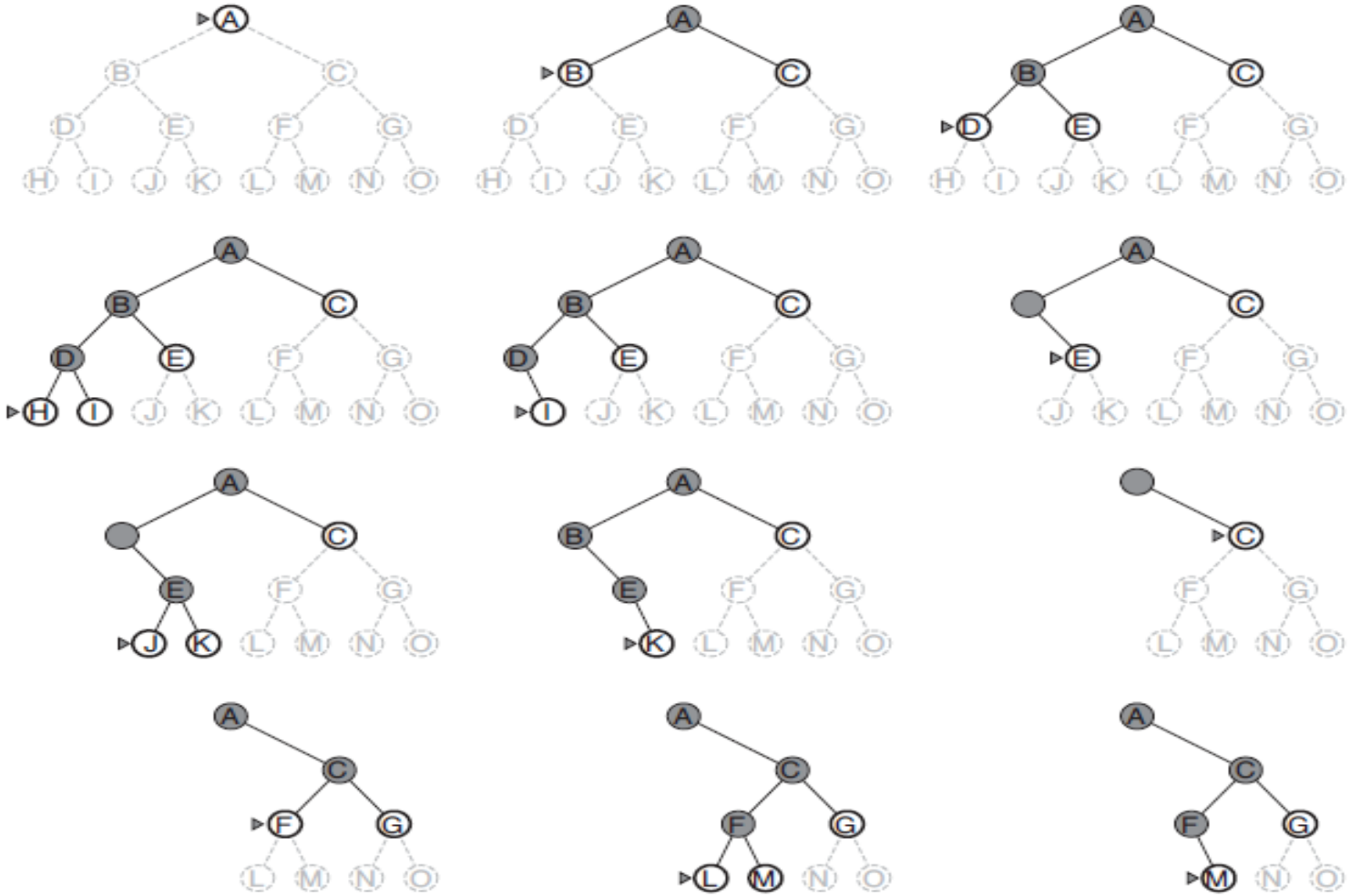
Exemple:

Profondeur	Nœuds	Temps	Mémoire
8	10^9	31 heures	1 teraoctet
12	10^{13}	35 ans	10 pétaoctets

- Les temps sont à 10 000 noeuds par seconde
- 1000 octets de mémoire pour un noeud.
- Téraoctet ~ 1 000 milliards, soit 10^{12} octets
- Pétaoctets = 1 000 téraoctets, soit 10^{15} octets

Recherche en profondeur d'abord

- Développe le noeud le plus profond.
- Implémenté à l'aide d'une pile: les nouveaux noeuds générés vont sur le sommet.



Propriétés de la recherche en profondeur d'abord

- Complétude : non, si la profondeur est infinie, s'il y a des cycles. Oui, si on évite les états répétés ou si l'espace de recherche est fini.

- Complexité en temps : $O(b^m)$, très mauvais si m est plus grand que d.

- Complexité en espace : $O(bm)$

- Optimal : Non

Recherche à coût uniforme

- Développe le nœud ayant le coût le plus bas.

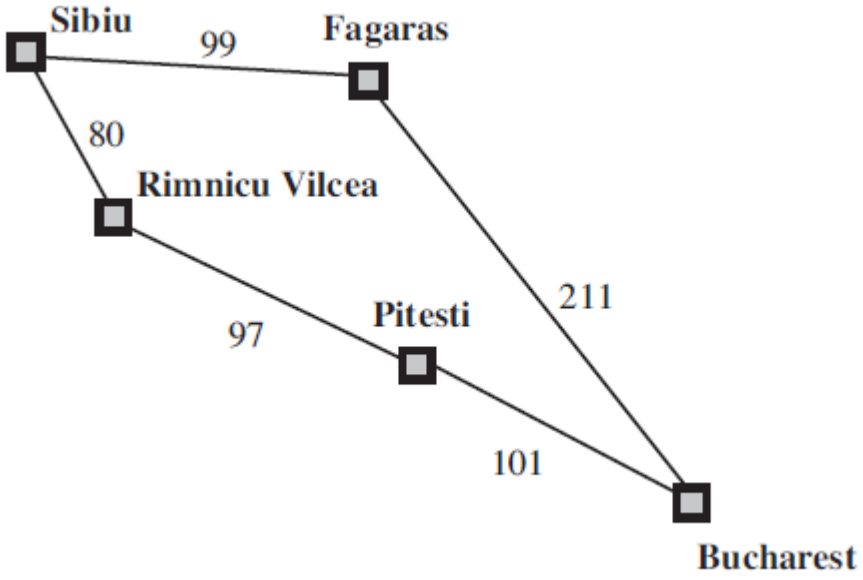
- File triée selon le coût

- Équivalent à largeur d'abord si le coût des actions est toujours le même.

Exemple pour coût uniforme

Aller de Sibiu à Bucarest:

- On développe le noeud à moindre coût, soit RV, ce qui ajoute Pitesti et au total ça donne 80 +97 = 177
- Le noeud de moindre coût est maintenant Fagaras; il est donc développé, ce qui ajoute Bucarest avec un coût de 99 + 211 = 310
- Un noeud but à été produit
- Le nœud Pitesti (177) continue et produit 177 + 101 = 278
- Le moindre coût vérifie lequel est meilleur et celui-ci est retenu. Les autres nœuds restants donnent un coup supérieur.



Propriétés de la recherche à coût uniforme

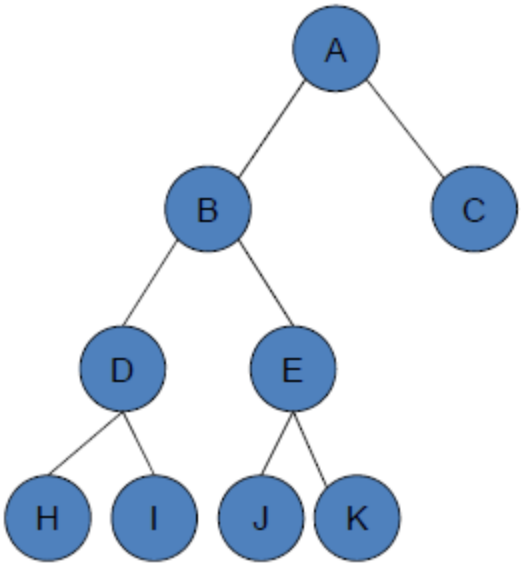
- Complète : oui, si le coût de chaque action ≥ ε
- Complexité en temps : nombre de noeuds n avec *g(n)* ≤ au coût de la solution optimale

$$O(b^{\lceil C^*/\epsilon \rceil})$$

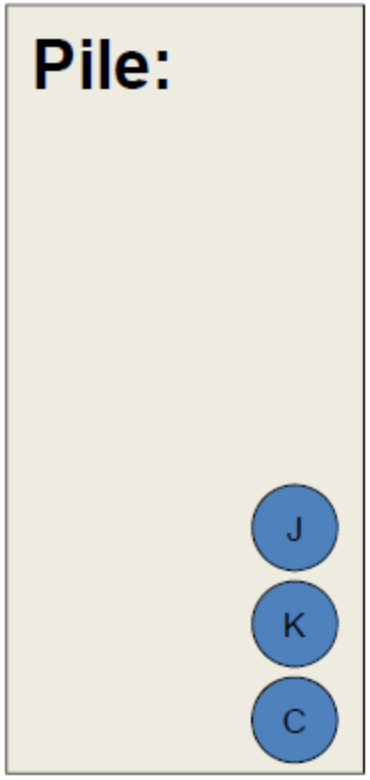
où *C** est le coût de la solution optimale. Risque d'être plus grand que $O(b^d)$

- Complexité en espace : même que celle en temps
- Optimal : oui, parce que les noeuds sont développés en ordre de *g*(*n*).

Recherche en profondeur d'abord avec retour-arrières(Backtracking)



Ordre de visite: A – B – D – H – I – E – J – K - C

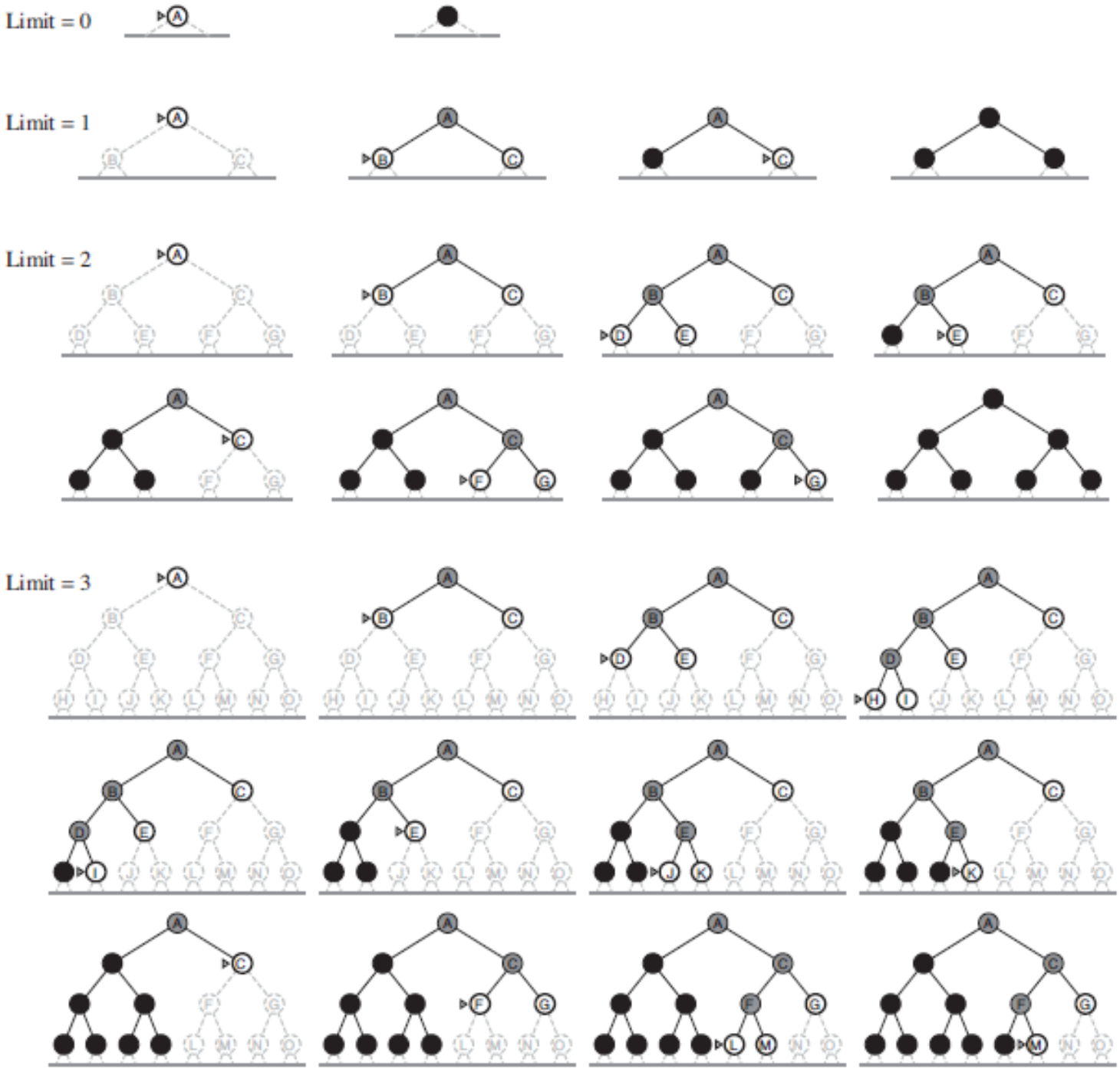


Recherche en profondeur limitée

- Est basée sur l'algorithme de profondeur d'abord, mais avec une limite de l sur la profondeur. Les nœuds de profondeur l n'ont pas de successeurs.
- Complétude : oui, si $l \geq d$
- Complexité en temps : $O(b^l)$
- Complexité en espace : $O(\mathrm{b}l)$
- Optimal : Non

Recherche en profondeur itérative

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


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Propriétés de la recherche en profondeur itérative

- Complétude : Oui
- Complexité en temps : $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Complexité en espace : $O(bd)$
- Optimal : Oui, si le coût de chaque action est de 1. Peut être modifiée pour une stratégie de coût uniforme.

Recherche bidirectionnelle

- Recherche simultanée du départ vers le but et du but vers le départ, afin de se rejoindre au milieu.

- Applicable si on peut faire une recherche à partir du but.

- Besoin d'une vérification efficace de l'existence d'un nœud commun aux deux arbres. Il faut conserver tous les nœuds d'au moins un des arbres.

Propriétés de la recherche bidirectionnelle

- Complétude : Oui
- Complexité en temps : $O(b^{d/2})$
- Complexité en espace : $O(b^{d/2})$
- Optimale : Oui

Recherche informée

- Les stratégies de recherche non-informée ne sont pas très efficaces dans la plupart des cas. Elles sont aveugles car elles ne savent pas si elles s'approchent du but.

- Les stratégies de recherche informée utilisent une fonction d'estimation (heuristique) pour choisir les nœuds à visiter.

Recherche informée

Meilleur d'abord:

- Meilleur d’abord glouton
- L'algorithme *A**

Meilleur d'abord

- L'idée est d'utiliser une fonction d'évaluation qui estime l'intérêt des noeuds et de développer le nœud le plus intéressant.
- Le noeud à développer est choisi selon une fonction d'évaluation $f(n)$.
- Une composante importante de ce type d'algorithme est une fonction heuristique $h(n)$ qui estime le coût du chemin le plus court pour se rendre au but.
- Deux types de recherche meilleur d'abord: A* et meilleur d'abord glouton.

Meilleur d'abord gloutonne

- $f(n) = h(n)$

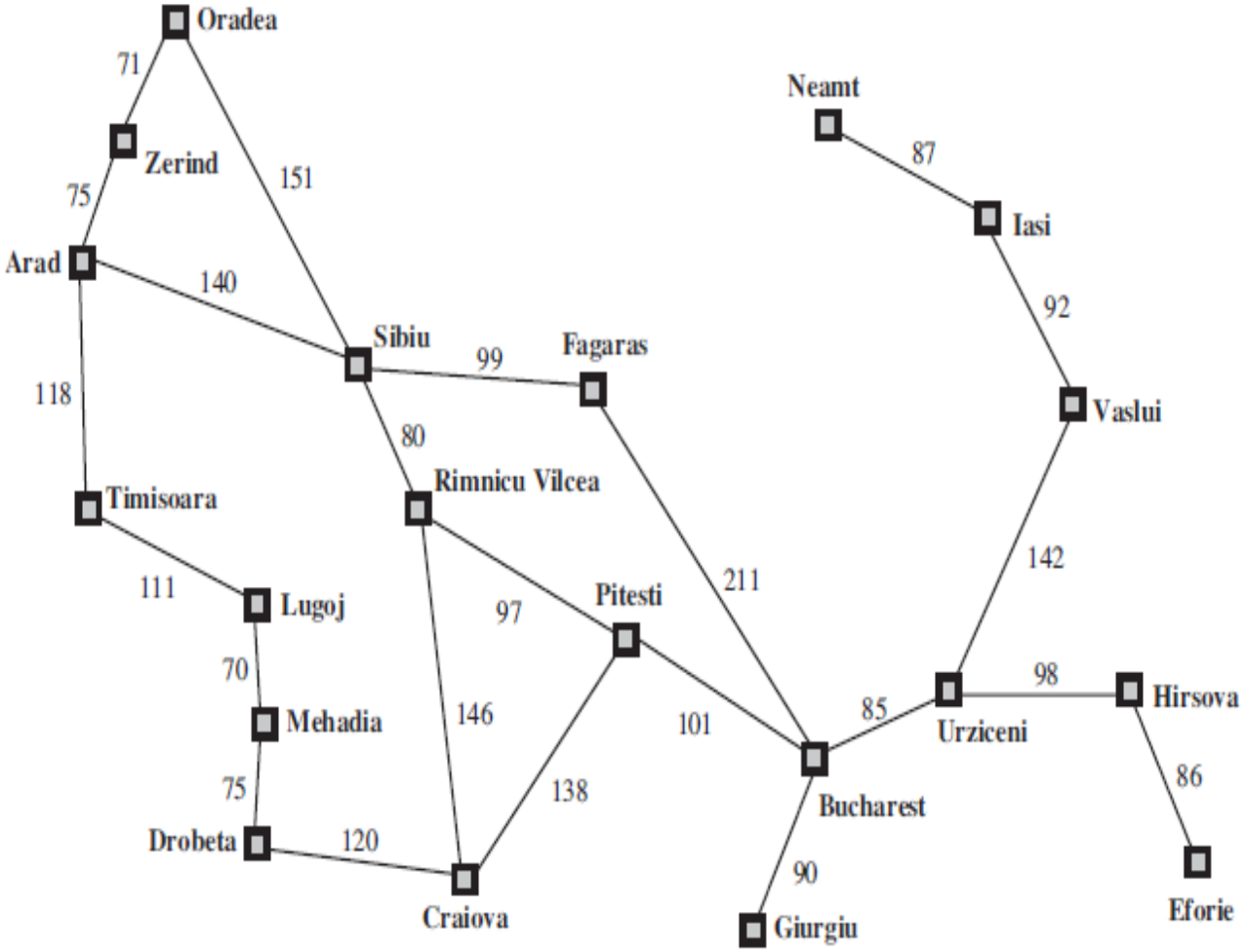
- Choisit toujours de développer le nœud le plus proche du but.

- Par exemple, dans le pb du chemin le plus court entre deux villes on peut prendre $h(n)$ = distance directe entre n et la ville destination.

Exemple

Distance à vol d'oiseau jusqu'à Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



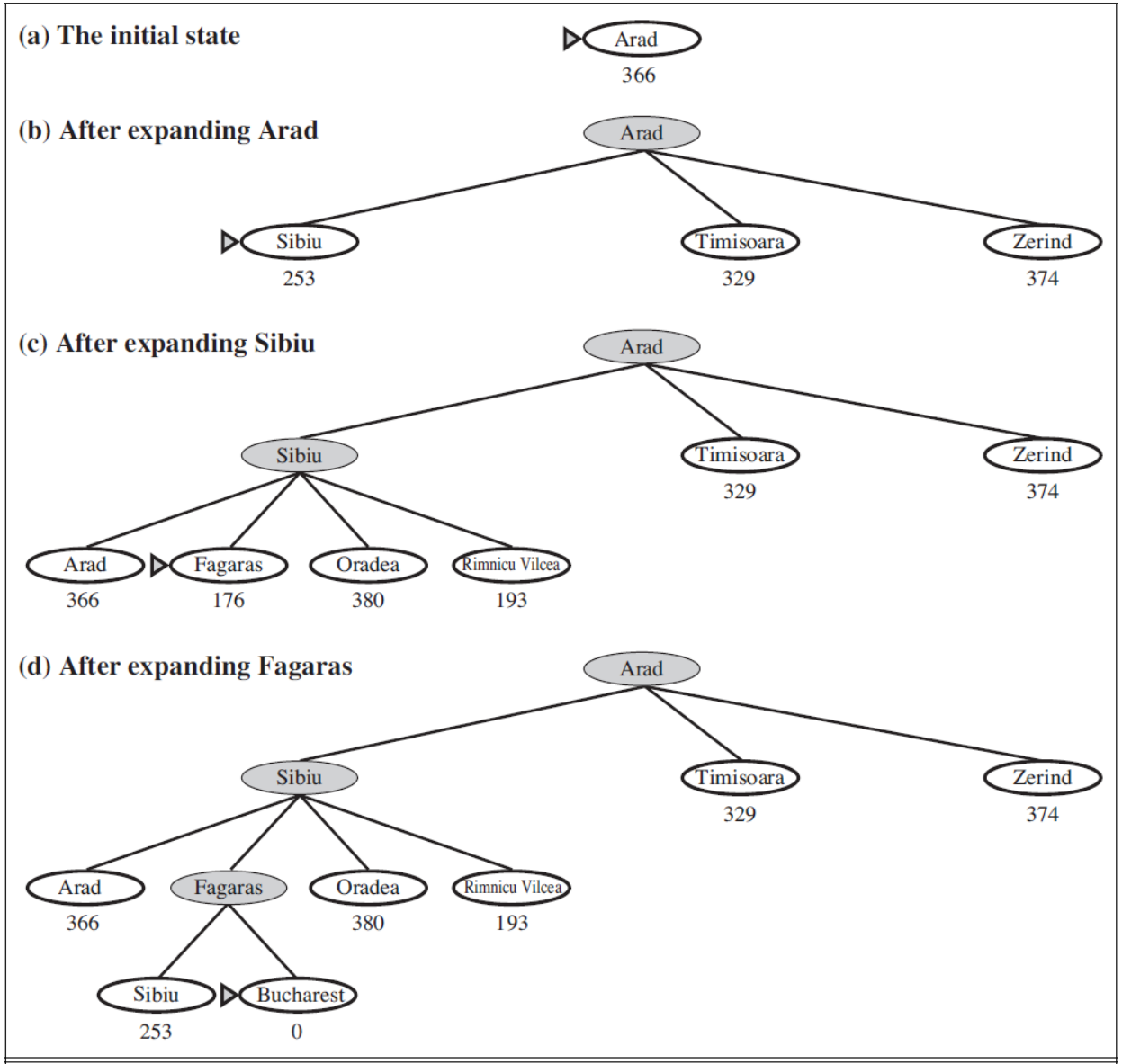


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- La solution n’est pas optimale: elle a un coût de 450, et on peut trouver plus court !
- Pourquoi? parfois, la solution optimale doit “s’éloigner” du but temporairement pour aller plus vite ensuite.
- Il faut trouver une meilleure fonction d’évaluation

Propriétés de la recherche Meilleur d'abord gloutonne

- Complétude : Non, elle peut être prise dans des cycles. Oui, si l'espace de recherche est fini avec vérification des états répétés (pas de cycles).

- Complexité de temps : $O(b^m)$, mais une bonne fonction heuristique peut améliorer grandement la situation.

- Complexité d'espace : $O(b^m)$, elle retient tous les nœuds en mémoire.

- Optimale : non, elle s'arrête à la première solution trouvée.

Algorithme A*

- Fonction d'évaluation: $f(n) = g(n) + h(n)$
  - *g(n)*: coût du noeud de départ jusqu'au noeud *n*
  - *h(n)*: coût estimé du noeud *n* jusqu'au but
  - *f(n)*: coût total estimé du chemin passant par *n* pour se rendre au but.
- Demande aussi que $h(n) \geq 0$ et que $h(G) = 0$ pour tous les buts G.
- On a toujours deux versions:
  - **tree-search** : on ne se rappelle pas des états visités
  - **graph-search** : on se rappelle des états visités

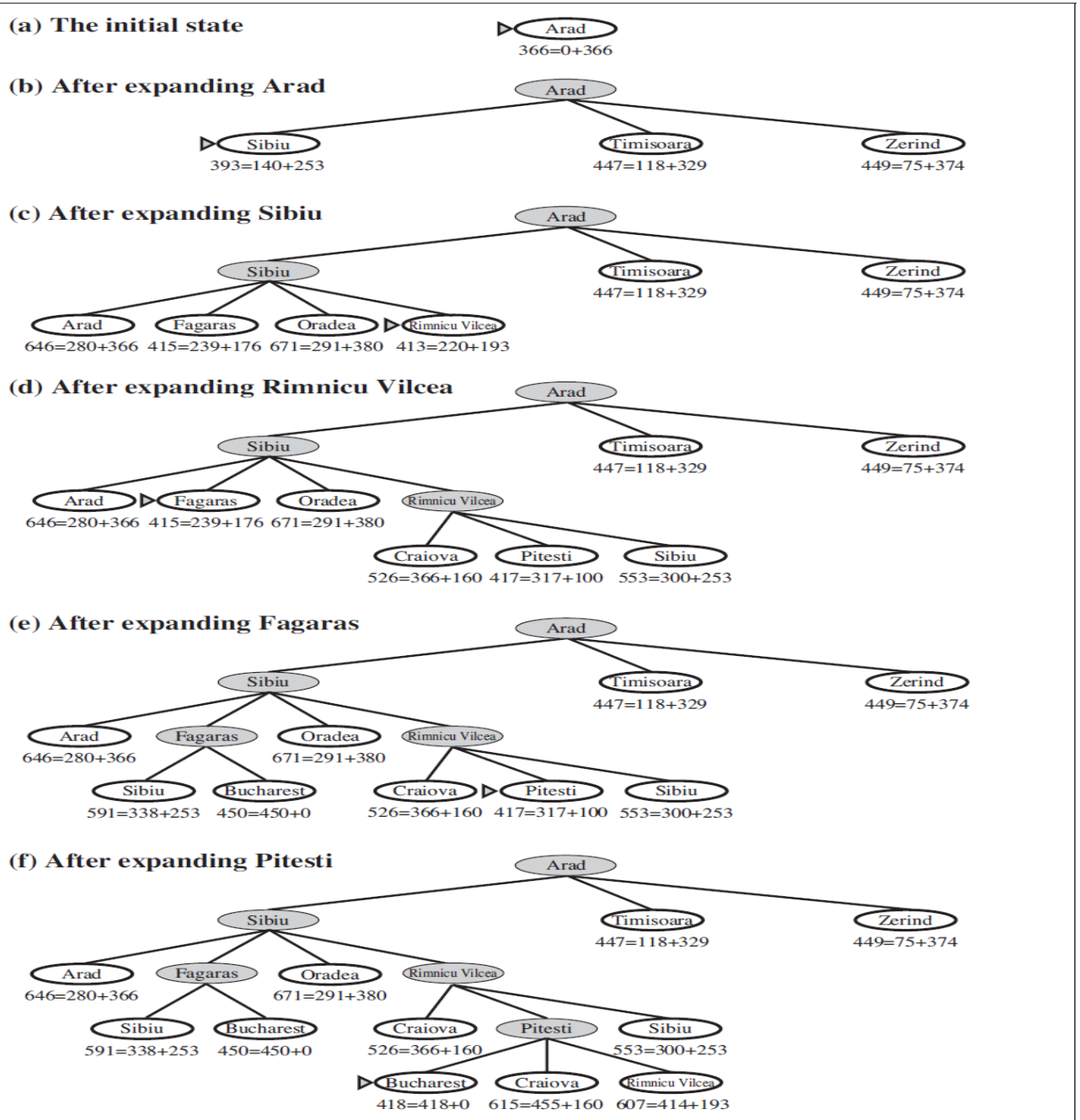


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

*Condition d'optimalité de A** (version tree-search)

Définition:

Une heuristique est **admissible** ssi elle **sous estime** toujours le coût

Théorème:

Dans la version **tree-search, A*** est optimal si la fonction heuristique h est admissible

Preuve:

Supposons qu'un but $G2$ non optimal soit dans la queue.

Soit n un noeud dans la queue qui se trouve dans un chemin vers un but optimal $G1$.

$f(G2) = g(G2)$ car $h(G2) = 0$

$\geq g(G1)$ car $G1$ est optimal

$\geq f(n)$ car h admissible implique

$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G1)$

Comme $f(G2) > f(G1)$, l'algorithme A^* ne va jamais choisir $G2$.

*Condition d'optimalité de A** **(version graph-search)**

Définition:

Une heuristique est monotone si pour tout noeud n, tout successeur n' de n on a: $h(n) \leq c(n,n')+h(n')$.

Théorème:

Dans la version **graph-search, A*** est optimal si la fonction heuristique h est monotone

Remarque:

Une heuristique monotone est admissible

Preuve

Supposons que h est monotone. Soit un chemin dans le graphe de recherche et n et n' deux nœuds successifs.

Alors $f(n) = g(n)+h(n) \leq g(n)+c(n,n') + h(n')$ (monotonicité de h)

$f(n) \leq g(n') \quad +h(n') = f(n')$.

On en conclut donc que f est croissante le long de tout chemin.

Démontrons maintenant qu'au moment où A* choisit de développer le nœud n, le chemin optimal menant à n a déjà été trouvé. Supposons que ce ne soit pas le cas: il doit exister un nœud n' sur le chemin optimal menant à n qui n'a pas été développé. Comme f est croissante, $f(n') \leq f(n)$, et donc n' aurait dû être développé, on arrive à une contradiction.

Propriétés de A*

- Aucun autre algoritme utilisant une même heuristique h ne peut développer moins de noeuds que A*: ne pas développer un noeud n tel que f(n) < C* présente le risque de manquer un chemin optimal.
- Pour des problèmes de grandes tailles, c'est généralement la mémoire qui limite l'utilisation de A*.
- il existe des variantes pour limiter l'utilisation de la mémoire (ex: iterative-deepening A* qui utilise l'idée de iterative-deepening avec un coût fixé : on cherche une solution au plus d'un coût fmax, si on ne trouve pas, on cherche avec un fmax plus grand)

Trouver des heuristiques admissibles

Considérer une version simplifiée du problème:

→ L'idée est qu'une solution optimale du problème simplifié est une heuristique admissible pour le problème original.

Exemples dans le jeu du Taquin:

- on cherche à relaxer certaines contraintes → pas de contraintes sur le déplacement → h1
- on a le droit de superposer les jetons → h2

Comment choisir entre deux heuristiques admissibles?

- On dit qu'une heuristique $h2$ domine $h1$ ssi:
  $h2(n) \geq h1(n)$ pour tout n. Dans ce cas, on choisit l'heuristique dominante.

- Une heuristique dominante peut réduire l'espace de recherche considérablement.

Exemple: jeu du taquin

Deux heuristiques admissibles :

$h1(n)$ = nb de pièces mal placées

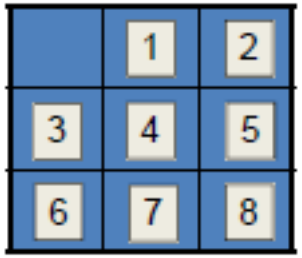
$h2(n)$ = Σ distances de chaque pièce à sa position finale

État initial

$h_1(n) = 8$

$h_2(n) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2$

$= 18$



État but