EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE FAKULTÄT

Artificial Intelligence
Chapter 3: Solving Problems by Searching

Andreas Zell

After the Textbook: Artificial Intelligence,
A Modern Approach
by Stuart Russel and Peter Norvig (3rd Edition)

Solving Problems by Searching

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Reflex agents are too simple and have great difficulties in learning desired action sequences
- Goal-based agents can succeed by considering future actions and the desirability of their outcomes.
- We now describe a special type of goal-based agents called problem-solving agents, which try to find action sequences that lead to desirable states.
- These are uninformed algorithms, they are given no hints or heuristics for the problem solution other than its definition.

3.1 Problem-Solving Agents

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- We first need a goal formulation, based on the current situation and the performance measure.
- Problem formulation is the process of deciding what actions and states to consider, given a goal.
- In general, an agent with several options for action of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.
- A search algorithm takes a problem as input and returns a solution in form of an action sequence.

Simple Problem-Solving Agent

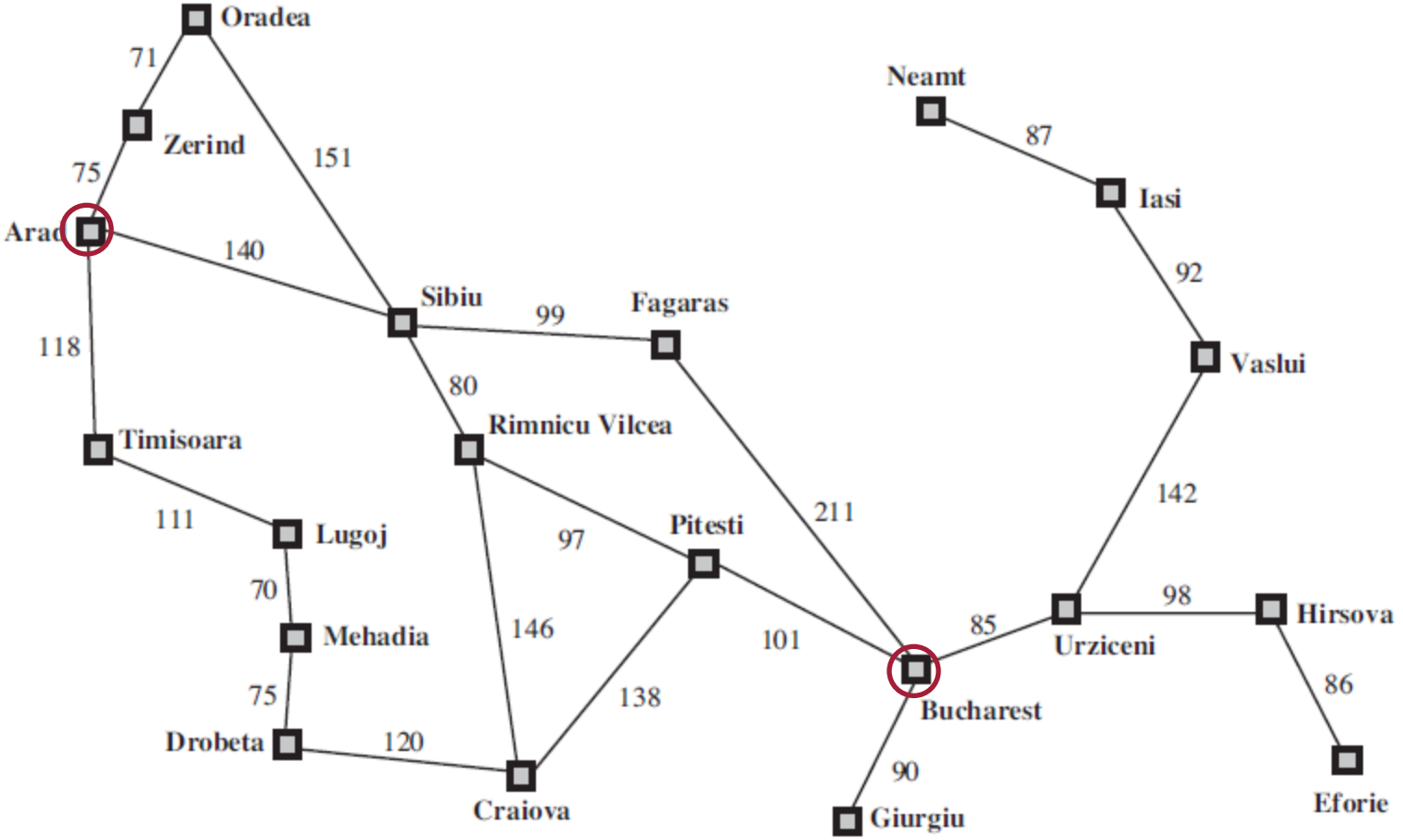
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: 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)
        if seq = failure then return a null action
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

3.2 Example Problem: Romania Tour

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example Problem: Romania Tour



Single-State Problem Formulation

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

A problem is defined by four items:

1. Initial state e.g., "at Arad"
2. Actions or successor function
   - S(x) = set of action–state pairs
   - e.g., S(Arad) = {<Arad → Zerind, Zerind>, … }
3. Goal test, can be
   - explicit, e.g., x = "at Bucharest"
   - implicit, e.g., Checkmate(x)
4. Path cost function (additive)
   - e.g., sum of distances, # actions executed, etc.
   - c(x,a,y) is the step cost, assumed to be ≥ 0

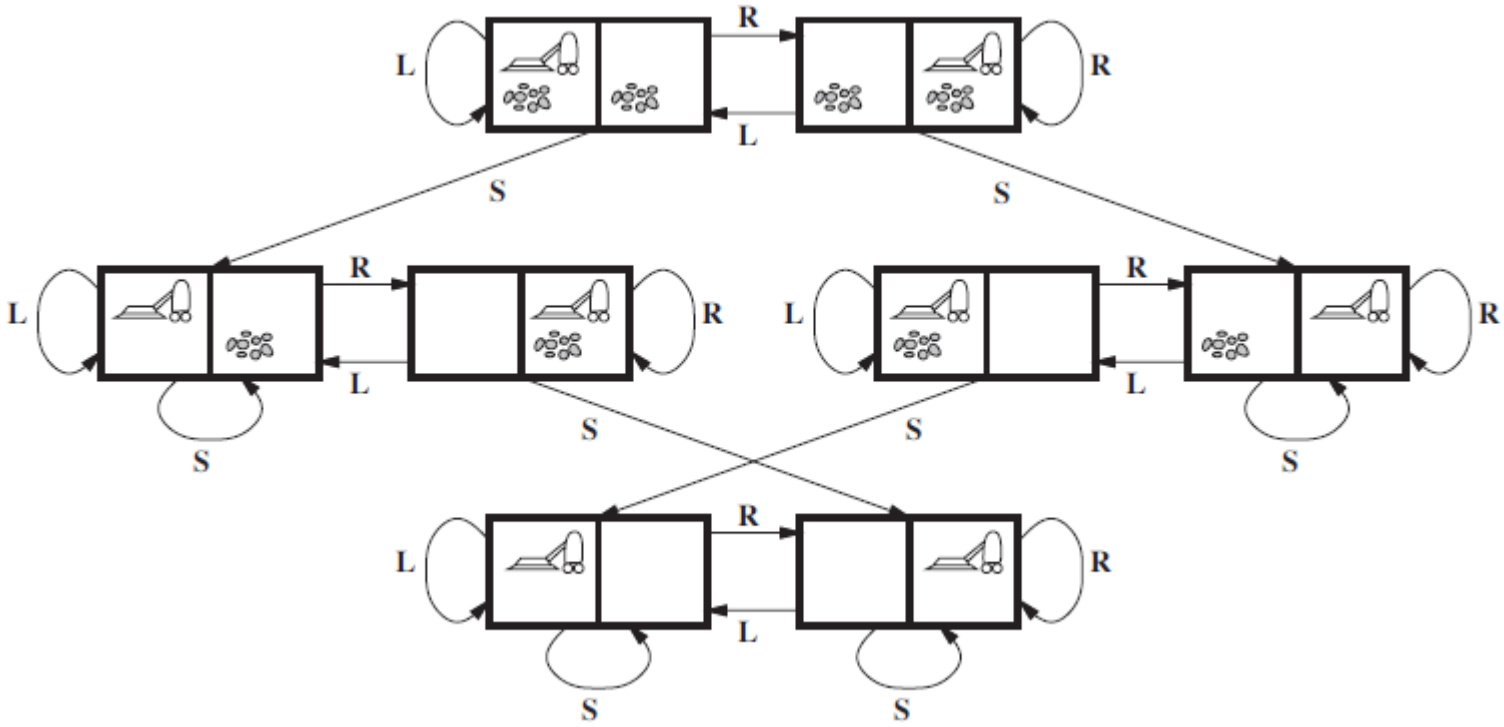
- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a State Space

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

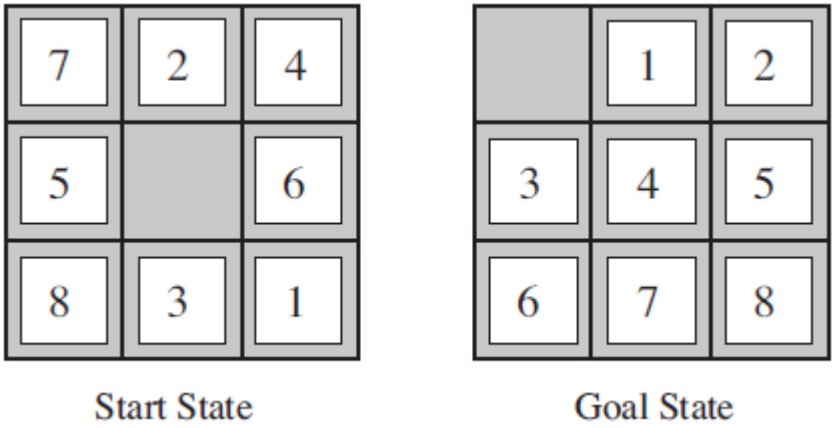
- Real world is absurdly complex, therefore state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum World State Space Graph



- States: dirt and robot location
- Actions: *Left*, *Right*, *Suck*
- Goal test: no dirt at all locations
- Path cost: 1 per action

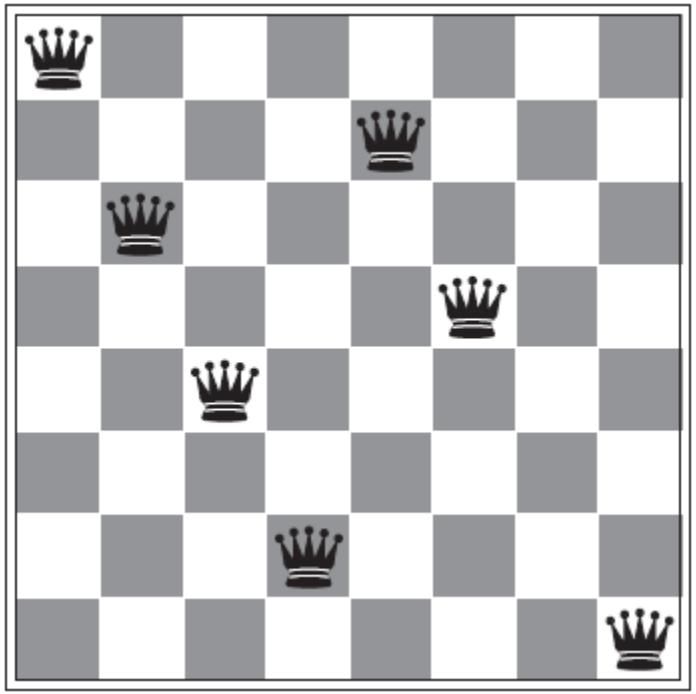
Example: The 8-Puzzle



- States: locations of tiles
- Actions: move blank left, right, up, down
- Goal test: state matches goal state (given)
- Path cost: 1 per move

(Note that the sliding-block puzzles are NP-hard)

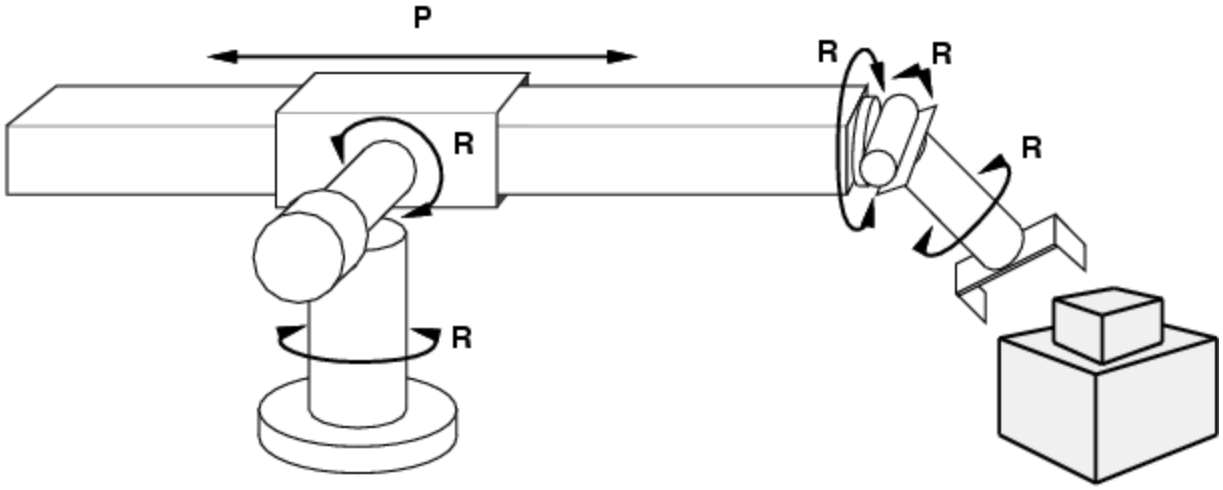
8-Queens Problem



Almost a solution (because of white diagonal)

- States: any arrangement of 0-8 queens on the board
- Actions: add a queen to an empty square
- Goal test: 8 queens on board, none attacked
- Path cost: 1 per move

Example: Robotic Assembly



- States: coordinates of robot joint angles, parts of object to be assembled
- Actions: continuous motions of robot joints
- Goal test: complete assembly
- Path cost: time to execute

Real-World Search Problems

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Route-finding problems
  - GPS-based navigation systems, Google maps
- Touring problems
  - TSP problem
- VLSI layout problems
- Robot navigation problems
- Automatic assembly sequencing
- Internet searching
- Searching paths in metabolic networks in bioinformatics

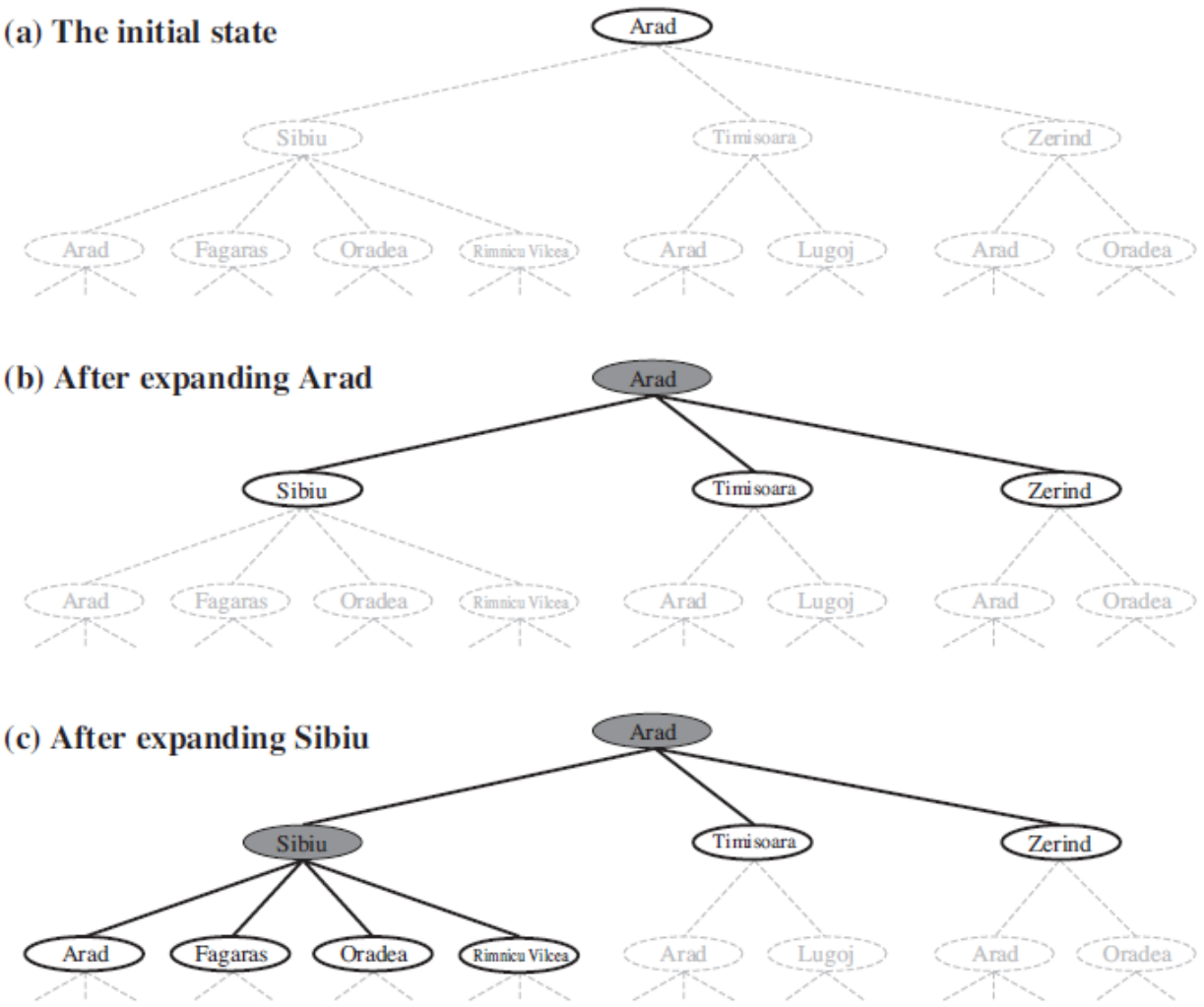
3.3 Searching for Solutions

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

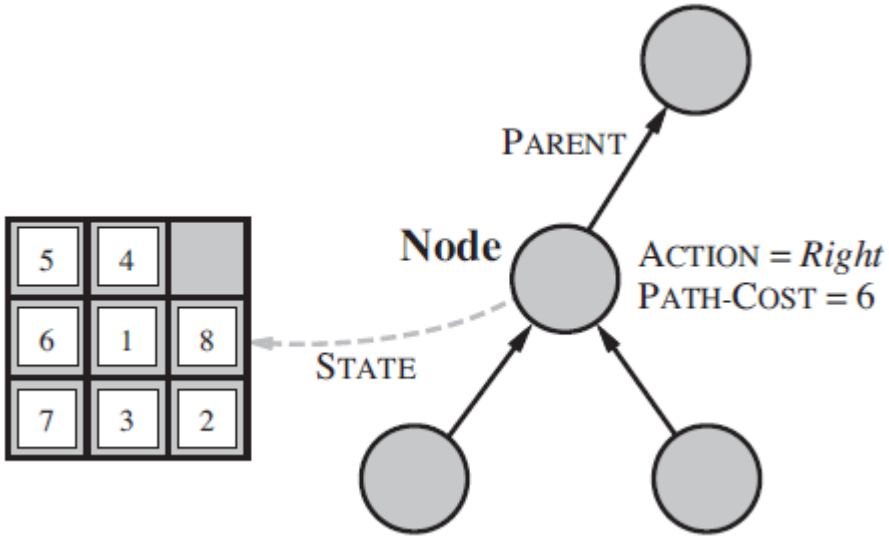
- Basic idea of tree search algorithms:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Search Tree, Example Romania



Search Tree Data Structures



Nodes are the data structures from which the search tree is constructed

Queue data structure to store frontier of unexpanded nodes:

- Make-Queue(element, …) creates queue w. given elements
- Empty?(queue) returns true iff queue is empty
- Pop(queue) returns first elem. and removes it
- Insert(element, queue) inserts elem. in queue, returns q.

Tree Search and Graph Search Algorithms

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Search Strategies

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

3.4 Uninformed Search Strategies

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

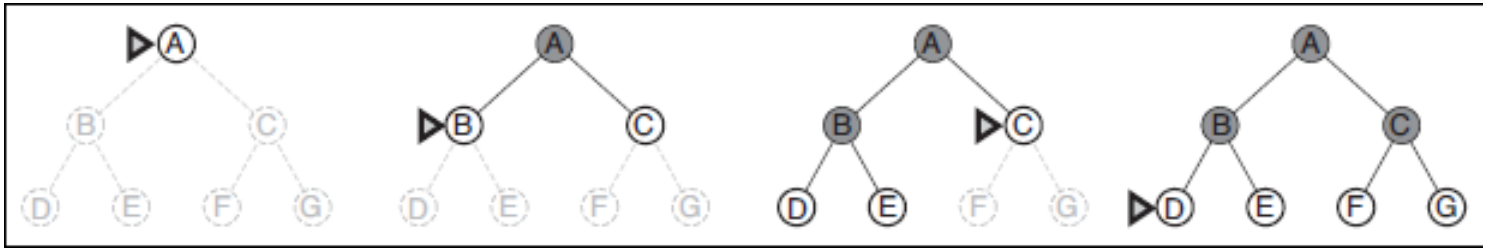
- Uninformed (blind) search strategies use only the information available in the problem definition

- Breadth-first search (BFS)
- Uniform-cost search
- Depth-first search (DFS)
- Depth-limited search
- Iterative deepening search

3.4.1 Breadth-First Search (BFS)

- The root node is expanded first
- Then all successors of the root node are expanded
- Then all their successors
- … and so on

- In general, all the nodes of a given depth are expanded before any node of the next depth is expanded.
- Uses a standard queue as data structure

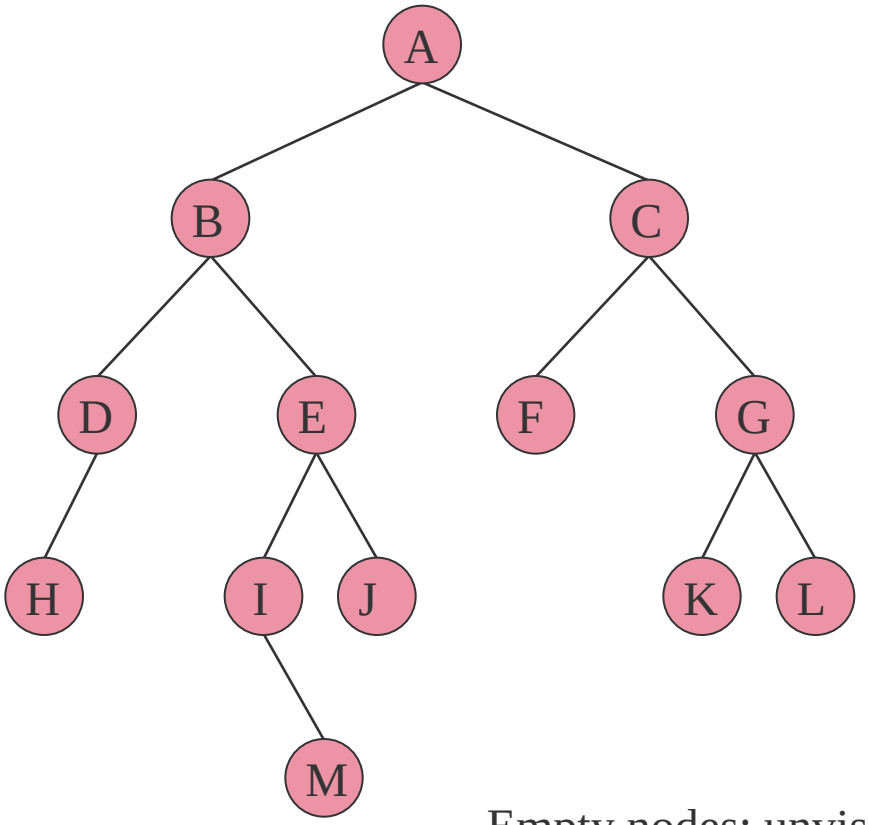


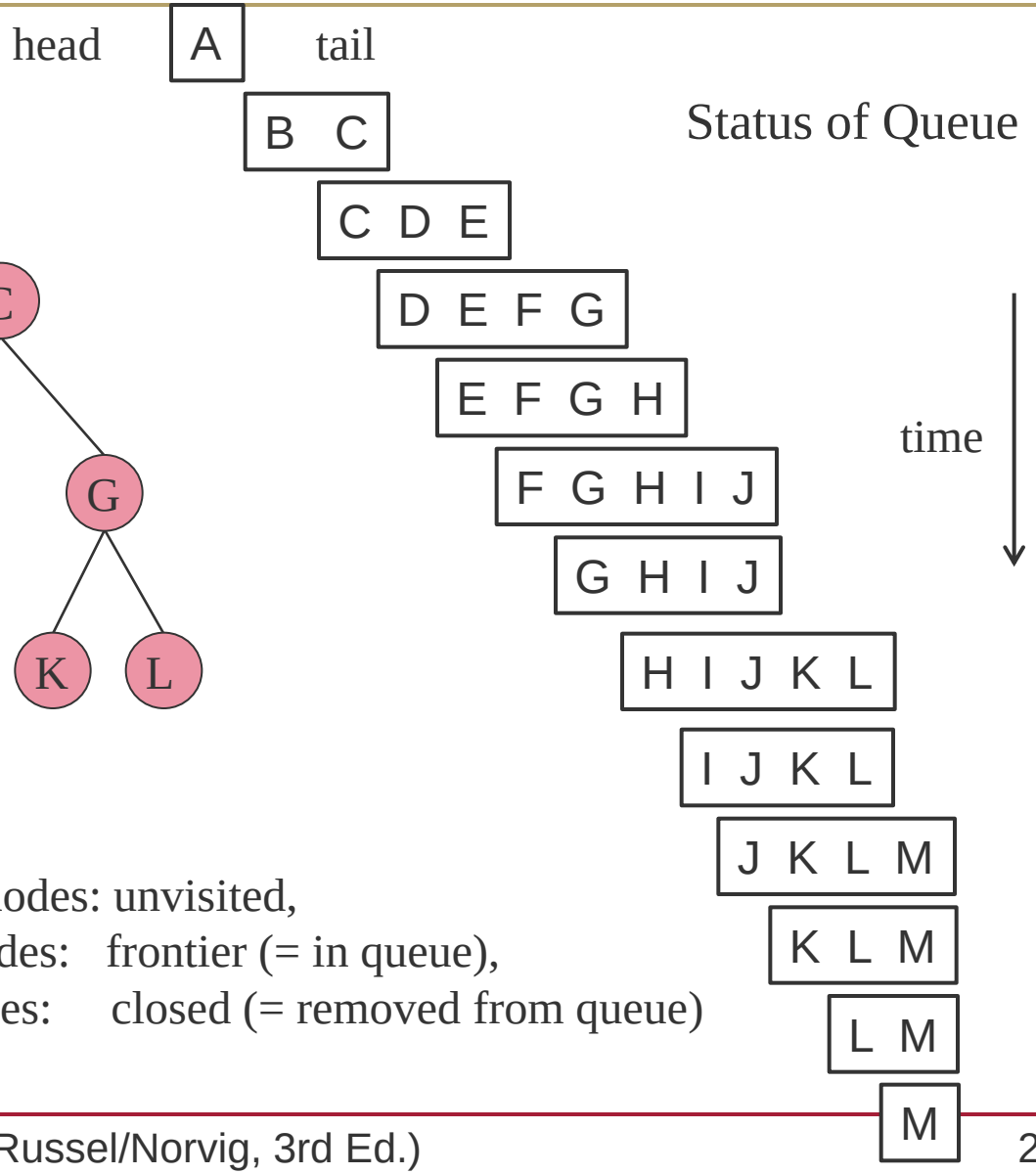
Breadth-First Search (BFS)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Example Breadth-First Search





Empty nodes: unvisited,
Grey nodes: frontier (= in queue),
Red nodes: closed (= removed from queue)

Breadth-First Search Properties

- BFS is complete (always finds goal if one exists)
- BFS finds the shallowest path to any goal node. If multiple goal nodes exist, BFS finds the shortest path.
- If tree/graph edges have weights, BFS does not find the shortest length path.
- If the shallowest solution is at depth d and the goal test is done when each node is generated then BFS generates $b + b^2 + b^3 + \ldots + b^d = O(b^d)$ nodes, i.e. has a time complexity of $O(b^d)$.
- If the goal test is done when each node is expanded the time complexity of BFS is $O(b^{d+1})$.
- The space complexity (frontier size) is also $O(b^d)$. This is the biggest drawback of BFS.

3.4.2 Uniform-Cost Search (UCS)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

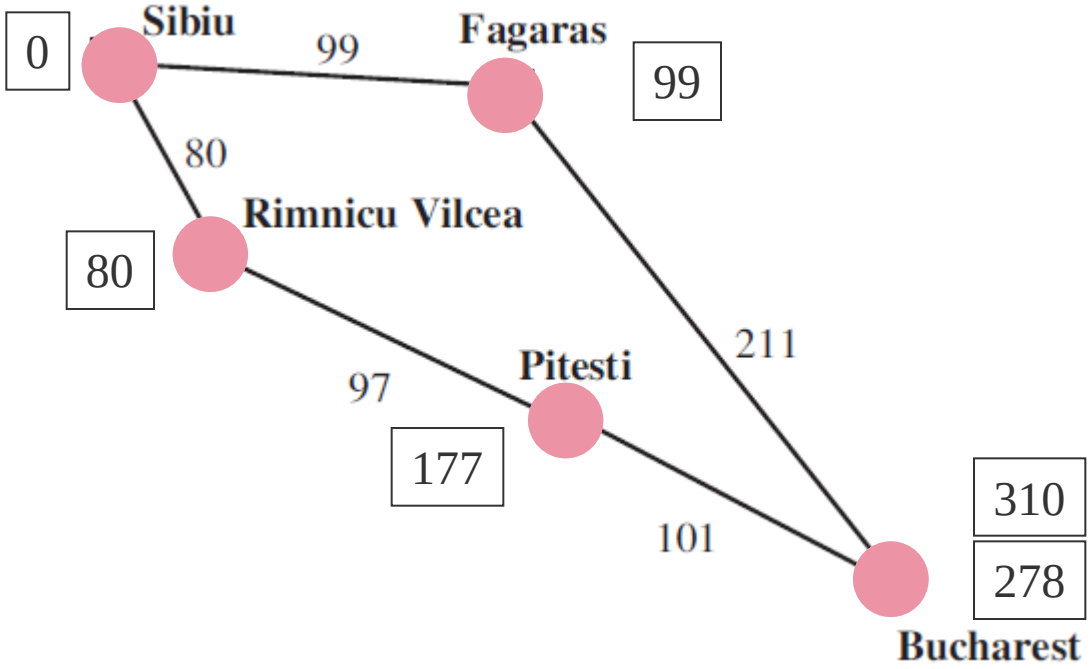
- Modification to BFS generates Uniform-cost search, which works with any step-cost function (edge weights/costs):
- UCS expands the node *n* with lowest summed path cost *g*(*n*).
- To do this, the frontier is stored as a priority queue. (Sorted list data structure, better heap data structure).
- The goal test is applied to a node when selected for expansion (not when it is generated).
- Also a test is added if a better node is found to a node on the frontier.

Uniform-Cost Search

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform-Cost Search

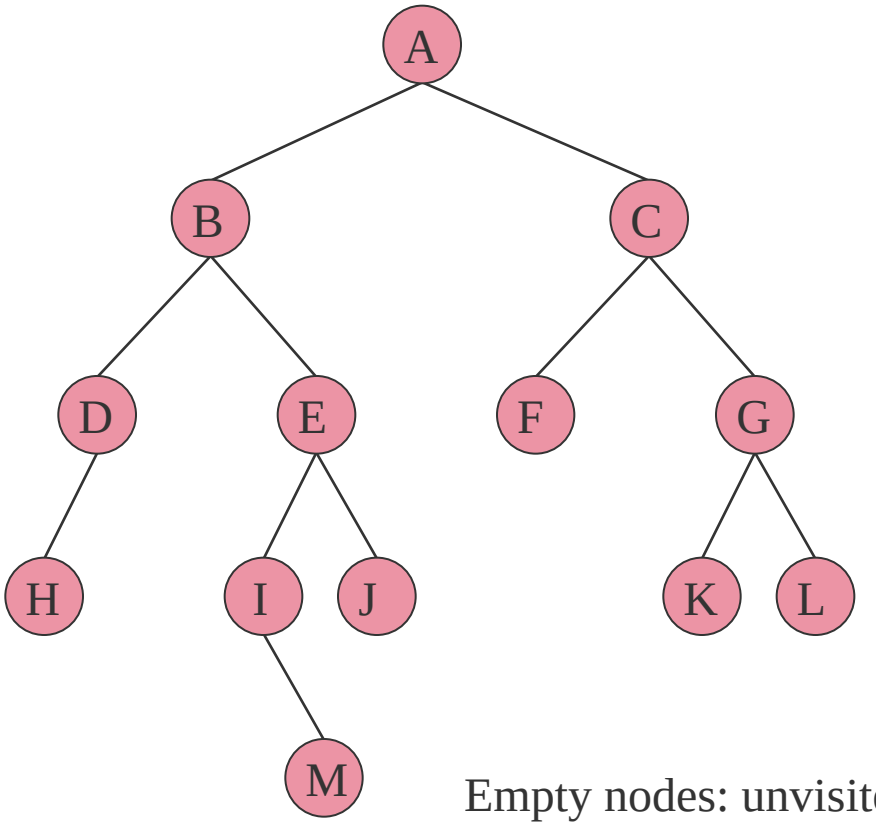


- Uniform-cost search is similar to Dijkstra's algorithm!
- It requires that all step costs are non-negative
- It may get stuck if there is a path with an infinite sequence of zero cost steps.
- Otherwise it is complete

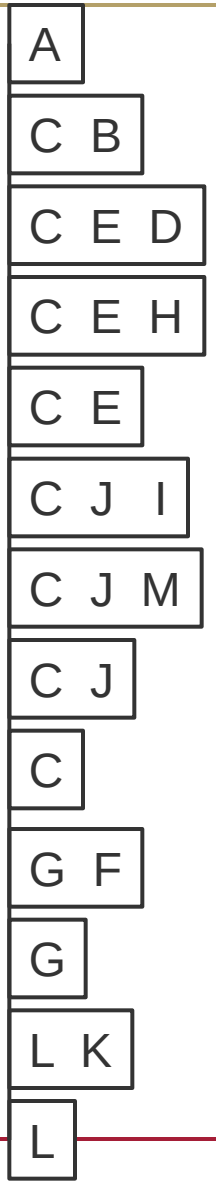
3.4.3 Depth-First Search (DFS)

- DFS alwas expands the deepest node in the current frontier of the search tree.
- It uses a stack (LIFO queue, last in first out)
- DFS is frequently programmed recursively, then the program call stack is the LIFO queue.
- DFS is complete, if the graph is finite.
- The tree search version of DFS is complete on a finite graph, if a test is included whether the node has already been visited
- DFS is incomplete on infinite trees or graphs.

Example Depth-First Search



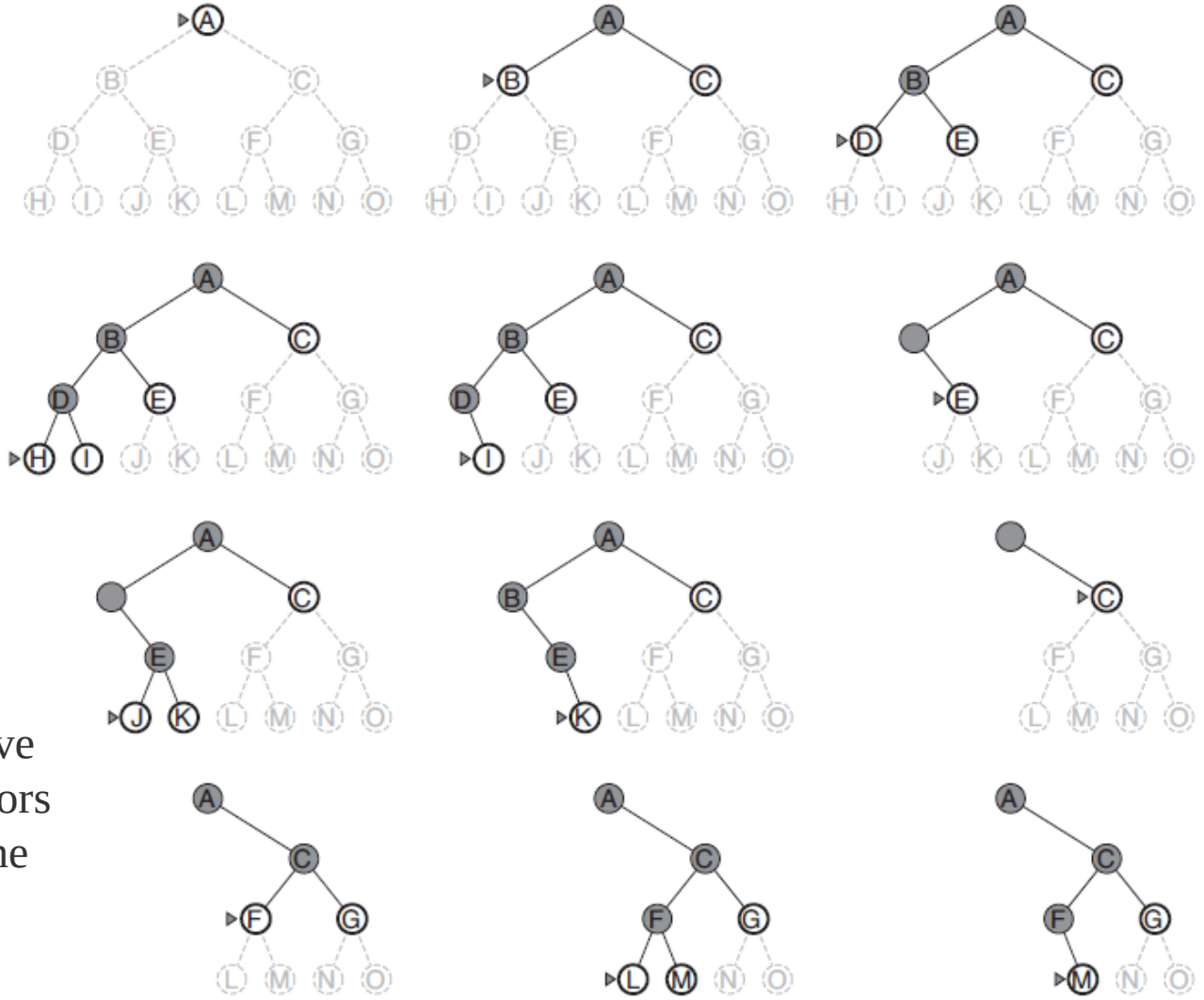
Empty nodes: unvisited,
Grey nodes: frontier (= on stack),
Red nodes: closed
(= removed from stack)



status of stack

time

DFS Example, AIMA book fig. 3.16



Nodes at depth 3 have no successors and M is the only goal node.

Depth-First Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- DFS has time complexity $O(b^m)$, if m is the maximum depth of any node (may be infinite).
- DFS has space complexity of $O(b\ m)$.
- In many AI problems space complexity is more severe than time complexity
- Therefore DFS is used as the basic algorithm in
  - Constraint satisfaction (chapter 6)
  - Propositional satisfiability (chapter 7)
  - Logic programming (chapter 9).
- Backtracking search, a variant of DFS, uses still less memory, only $O(m)$, by generating successors not at once, but one at a time.

3.4.4 Depth-Limited Search

- The failure of DFS in infinite search spaces can be prevented by giving it a search limit l.
- This approach is called depth-limited search.
- Unfortunately, it is not complete if we choose $l < d$, where d is the depth of the goal node.
- This happens easily, because d is unknown.
- Depth-limited search has time complexity $O(b^l)$.
- It has space complexity of $O(b\ l)$.
- However, in some applications we know a depth limit (# nodes in a graph, maximum diameter, …)

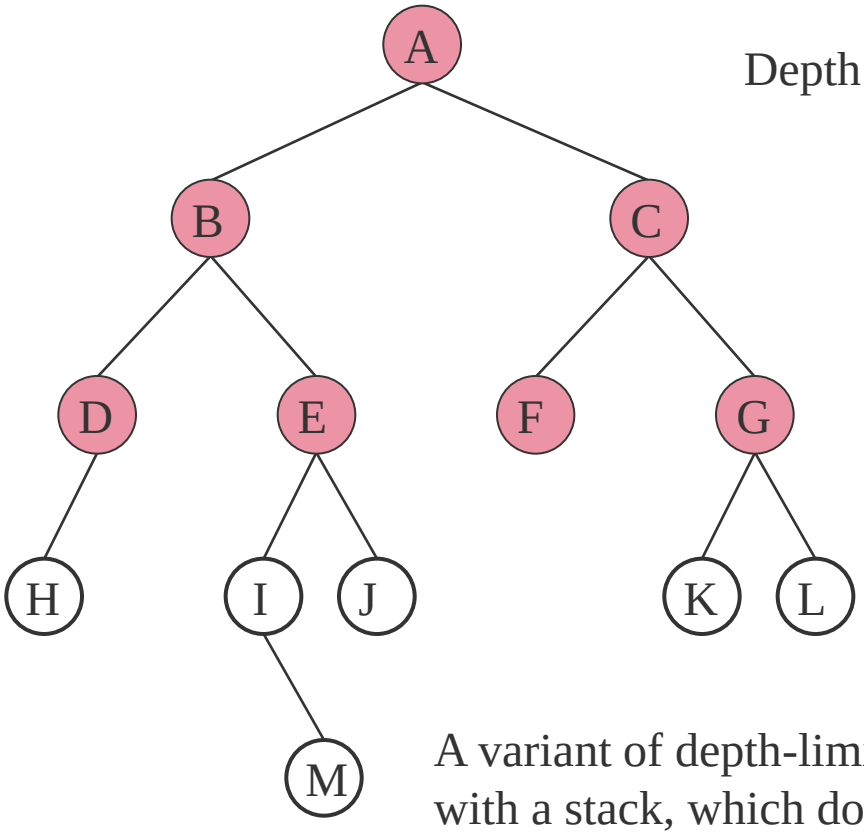
Depth-Limited Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

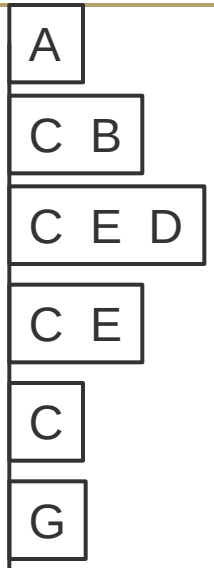
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.16 A recursive implementation of depth-limited tree search.

Example Depth-Limited Search



Depth limit = 2



A variant of depth-limited search with a stack, which does the depth limit test of children before putting them on the stack. This is like the recursive version in the AIMA book.

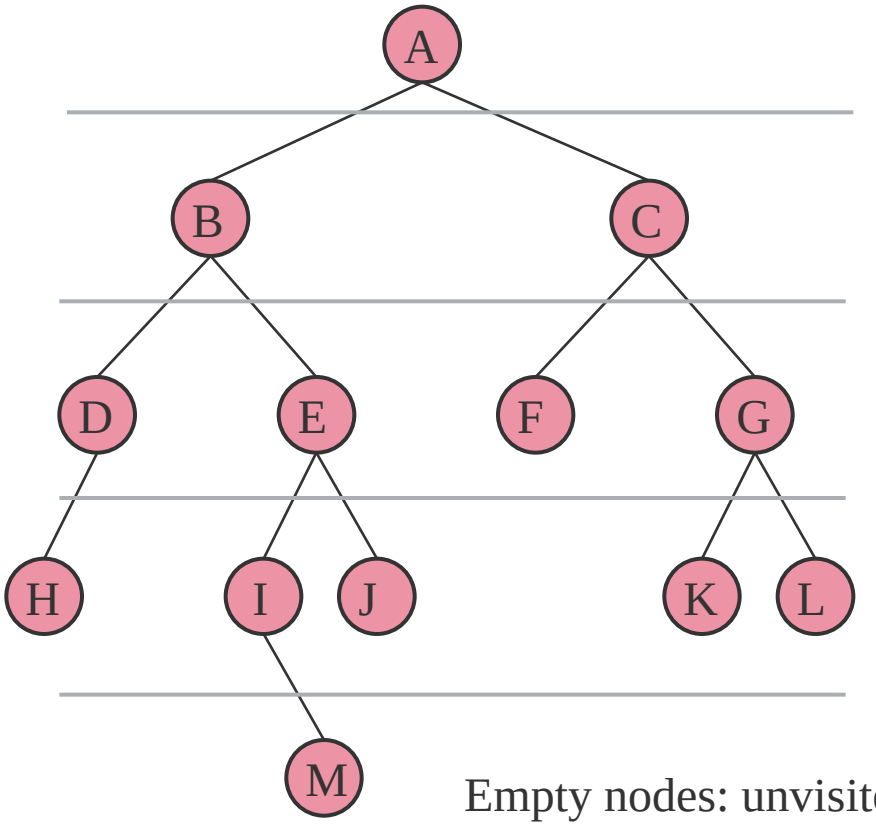
3.4.5 Iterative Deepening Search

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

- The iterative deepening search algorithm repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns failure, meaning that no solution exists.
- This search is also frequently called depth-first iterative deepening (DFID)

Example Iterative Deepening Search



Depth 0

Depth 1

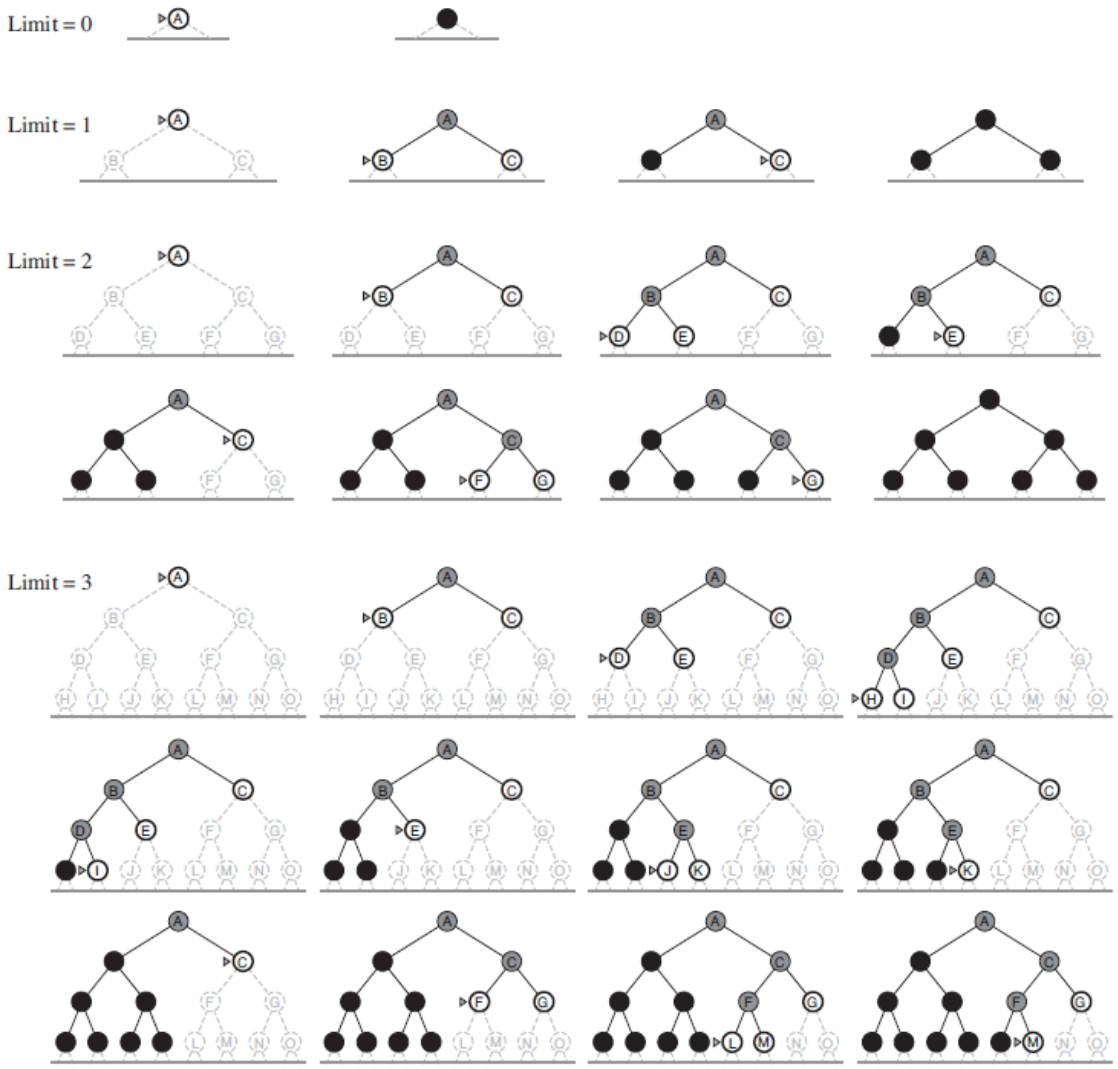
Depth 2

Depth 3

Depth 4

Empty nodes: unvisited,
Grey nodes: frontier (= on stack),
Red nodes: closed
(= removed from stack)

Iterative Deepening, AIMA ex. fig. 3.19

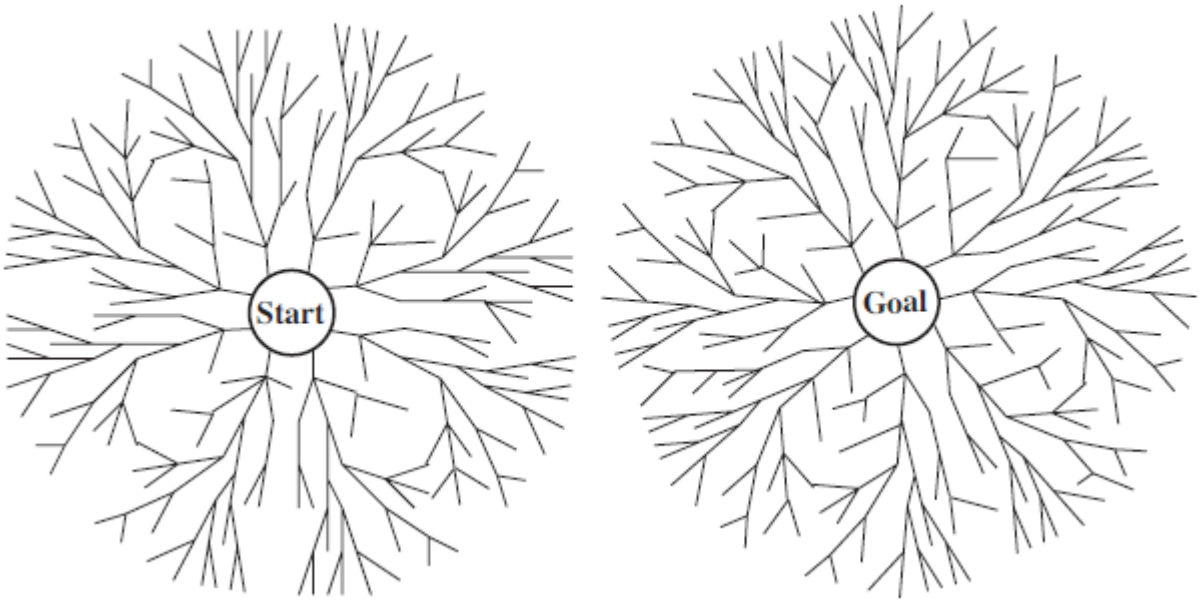


Iterative Deepening Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Repeatedly visiting the same nodes seems like a waste of time (not space). How costly is this?
- This heavily depends on the branching factor *b* and the sparseness of the search tree.
- Assume $b = 2$, $d = 10$, full binary tree, then
  - $N(IDS) = 11*2^0 + 10*2^1 + 9*2^2 + \ldots + 1*2^{10} = 4083$
  - $N(DFS) = 1*2^0 + 1*2^1 + 1*2^2 + \ldots + 1*2^{10} = 2047$
  - i.d. IDS generates at most twice the #nodes of DFS or BFS
- Assume $b = 10$, $d = 10$, full 10-ary tree, then
  - $N(IDS) = 11*10^0+10*10^1+9*10^2+\ldots+1*10^{10} = 12.345.679.011$
  - $N(DFS) = 1*10^0 + 1*10^1+1*10^2+\ldots+1*10^{10} = 11.111.111.111$
  - i.d. IDS generates only 11% more nodes than DFS or BFS
- IDS is the preferred uninformed search method when the search space is large and the solution depth is unknown.

3.4.6 Bidirectional search



- Advantage: delays exponential growth by reducing the exponent for time and space complexity in half
- Disadvantage: at every time point the two fringes must be compared. This requires an efficient hashing data structure
- Bidirectional search also requires to search backward (predecessors of a state). This is not always possible

3.4.7 Comparing Uninformed Search Strategies

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Criterion	Breadth-first	Uniform-cost	Depth-first	Depth-limited	Iterative deepen.	Bidirectional
Complete	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b\,m)$	$O(b\,l)$	$O(b\,d)$	$O(b^{d/2})$
Optimal	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

- Evaluation of tree-search strategies. *b* is the branching factor, *d* is the depth of the shallowest solution, *m* is the maximum depth of the search tree, *l* is the depth limit.
- [a] complete, if *b* is finite;
  [b] complete, if step costs >= ε > 0;
  [c] optimal, if step costs are all identical;
  [d] if both directions use BFS

3.5 Informed (Heuristic) Search Strat.

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- A search strategy that uses problem-specific knowledge can find solutions more efficiently than an uninformed strategy.
- We use best-first search as general approach
- A node is selected for expansion based on an evaluation function $f(n)$.
- Informed search algorithms include a heuristic function $h(n)$ as part of $f(n)$.
- Often $f(n) = g(n) + h(n)$
- $h(n)$ = estimate of the cheapest cost from the state at node n to a goal state; $h(goal) = 0$.

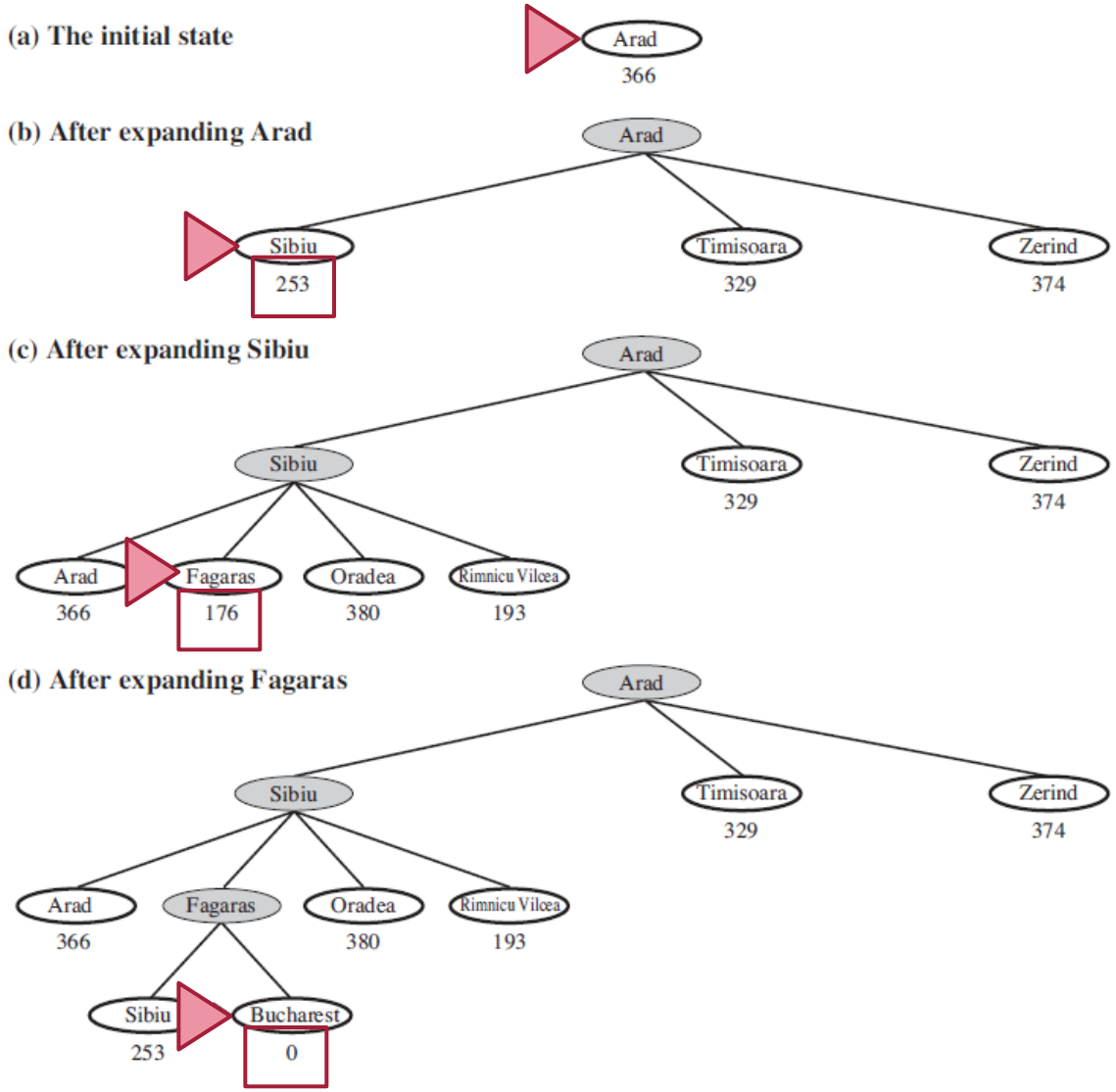
3.5.1 Greedy Best-First Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Greedy best-first search tries to expand the node that it estimates as being closest to the goal.
- It uses only the heuristic function $h(n)$,
- $f(n) = h(n)$.
- We use a straight-line distance heuristic h_{SLD} for the route-finding in Romania: straigt line distances to the goal Bucharest (as given in the table below).

h_{SLD}( ) =			
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

Greedy Best-First Search



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

Greedy Best-First Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

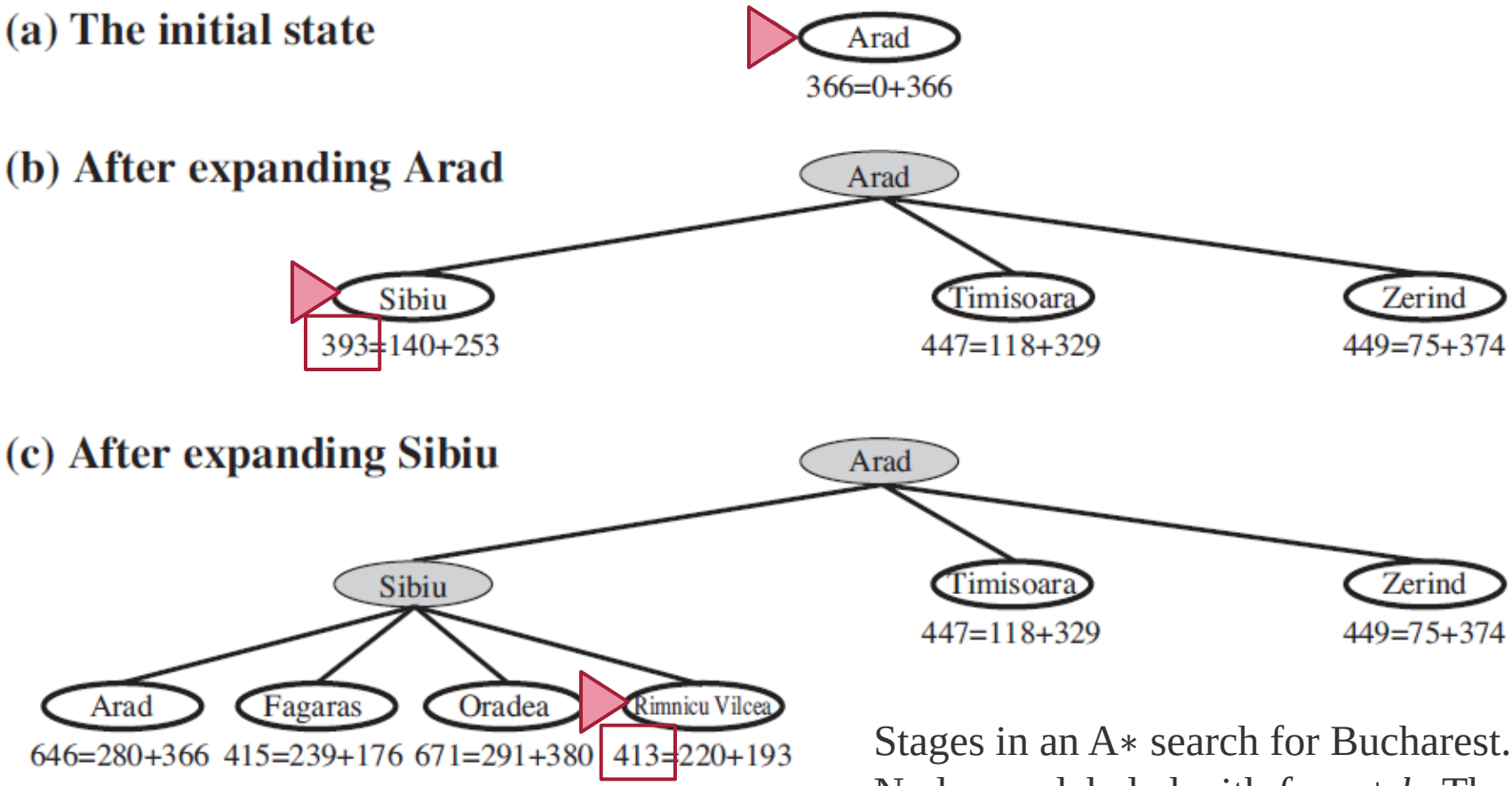
- The algorithm is called “greedy”, because in each step the algorithm greedily tries to get as close to the goal as possible.
- GBFS as tree is not complete, even in a finite state space (it may not find the shortest solution to the goal).
- The graph search version of GBFS is complete in finite search spaces, but not in infinite ones.
- Suggestion for improvement: Use the accumulated path distance $g(n)$ plus a heuristic $h(n)$ as cost function $f(n)$. This leads to A*

3.5.2 A* Search

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

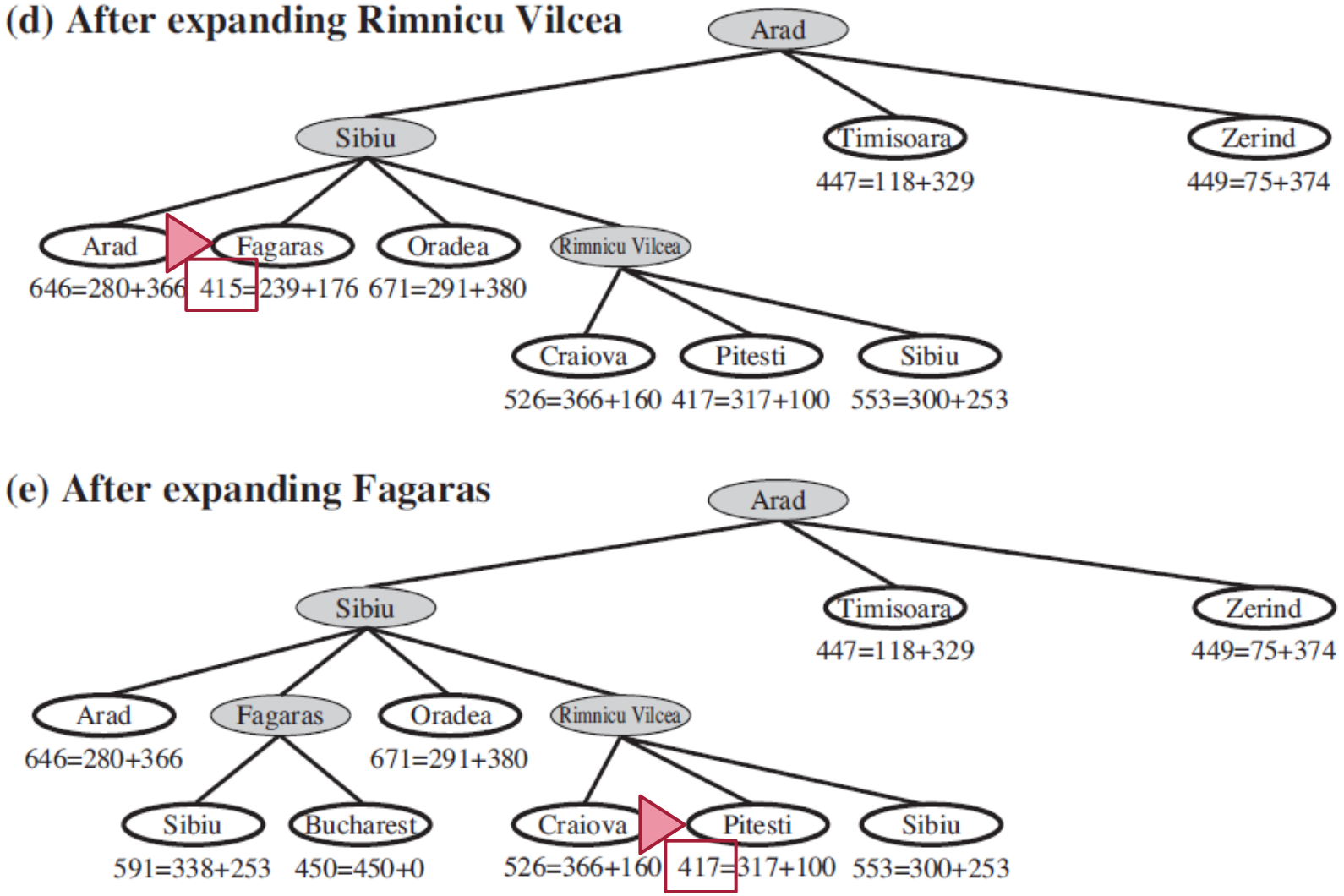
- A* search is one of the most popular AI search algorithms
- It combines two path components:
  - $g(n)$, the travelled path component from the start node to the node n, and
  - $h(n)$, a heuristic component, the estimated cost to reach the goal.
- $f(n) = g(n) + h(n)$
- $f(n)$ = estimated cost of the cheapest solution to the goal passing through node n.
- Under certain conditions for $h(n)$, A* is both complete and optimal.

A* Search

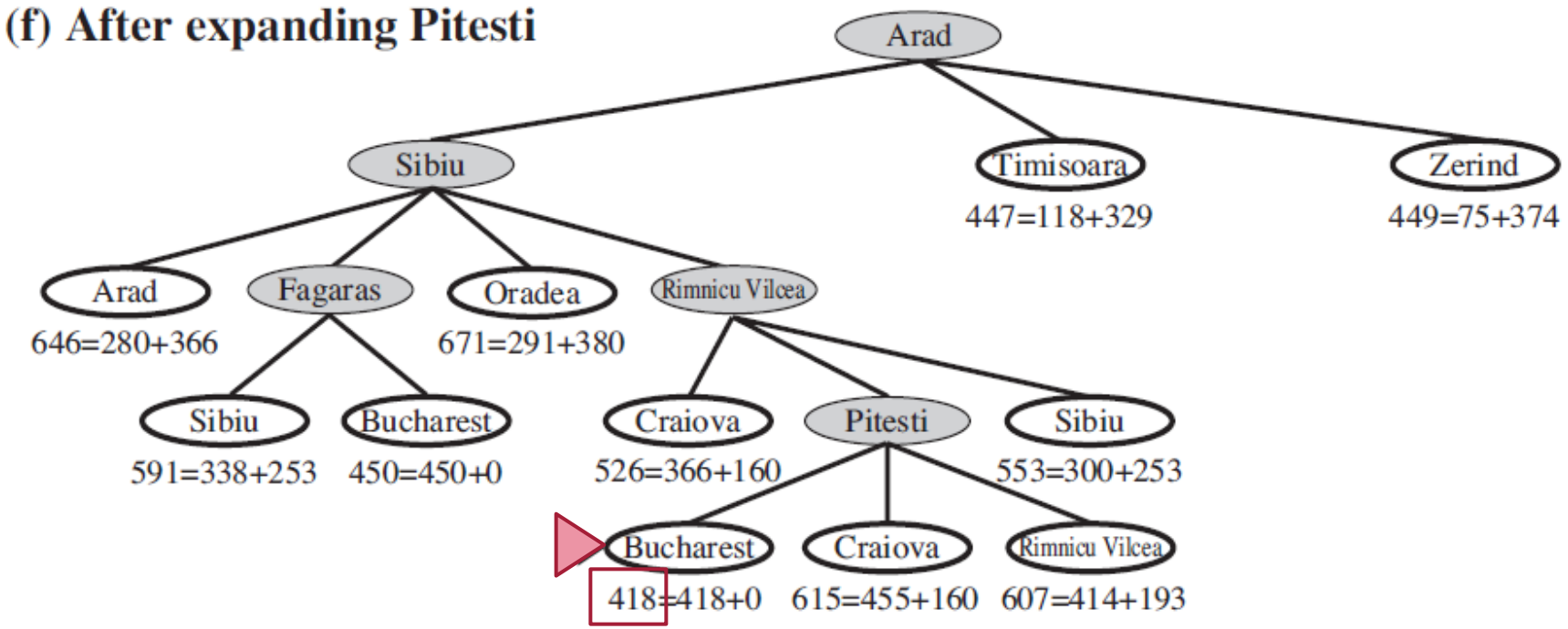


Stages in an A∗ search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest

A* Search


(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search

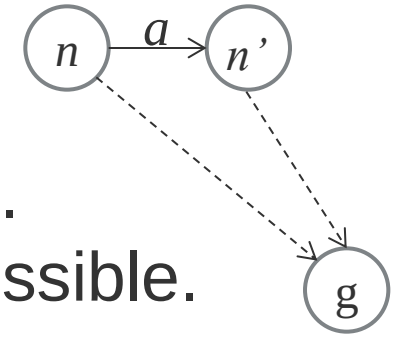

(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Conditions for Optimality of A*

In order for A* to be optimal

- $h(n)$ must be admissible, i.e. it never overestimates the cost to reach the goal.
- Then, as a consequence, $f(n) = g(n) + h(n)$ never overestimates the true cost of a solution along the current path through n.
- $h(n)$ must be consistent (monotonic) in graph search, i.e. for every node n and every successor n' of n generated by action a,

$$h(n) \leq c(n, a, n') + h(n')$$



- This is a form of the triangle inequality.
- Every consistent heuristic is also admissible.

Optimality of A* (Graph Search)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- The graph-search version of A* is optimal if $h(n)$ is consistent. We show this in 2 steps:

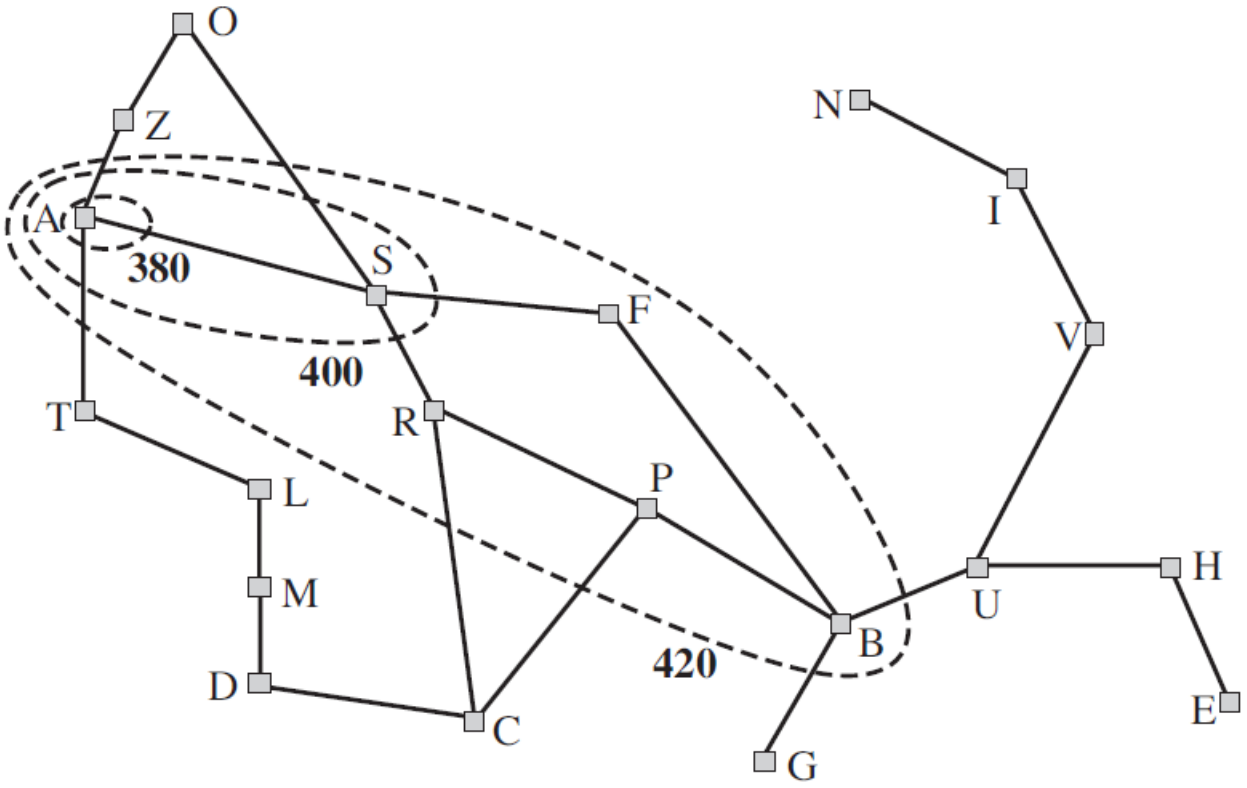
1. If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.
   - Suppose n' is a succ. of n, then $g(n') = g(n) + c(n, a, n')$
   - Therefore
$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n')$$
$$\geq g(n) + h(n) = f(n)$$
2. Whenever A* selects a node n for expansion, the optimal path to that node has been found.
   - otherwise there must be another frontier node n' on the optimal path from start node to n; as f is nondecreasing along any path, n' would have lower f-cost than n and would have been selected first.

Contours in State Space

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- As *f*-costs are nondecreasing along any path we can draw contours in the state space.
- See figure 3.25 on the following page
- With uniform-cost search (A* with $h(n) = 0$), contours are circular around the start state.
- With more accurate heuristics, the bands will stretch towards the goal state and become more narrow
- If C^* is the cost of the optimal solution path, then
  - A* expands all nodes with $f(n) < C^*$
  - A* may expand some nodes with $f(n) = C^*$ before finding the goal state.
- Completeness requires that there are only finitely many nodes with cost less than or equal to C^*, which is true if all step costs exceed some finite ε and if b is finite.

Contours in State Space



Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

A* is Optimally Efficient

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- A* is optimally efficient for any given consistent heuristic, i.e. no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly for nodes with $f(n) = C^*$).
  - This is because any algorithm that does not expand all nodes with $f(n) < C^*$ may miss the optimal solution.
- So A* is complete, optimal and optimally efficient, but it still is not the solution to all search problems:
- As A* keeps all generated solutions in memory it runs out of space long before it runs out of time.

3.5.3 Memory-bounded Heuristic Search

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- How to reduce memory requirements of A*?
- Use iterative deepening A* (IDA*)
- Cutoff used is $f(n) = g(n) + h(n)$ rather than the depth.
- At each iteration, the cutoff value is the smallest *f*-cost of any node that exceeded the cutoff on the previous iteration.
- This works well with unit step costs.
- It suffers from severe problems with real-valued costs.

Recursive Best-First Search (RBFS)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

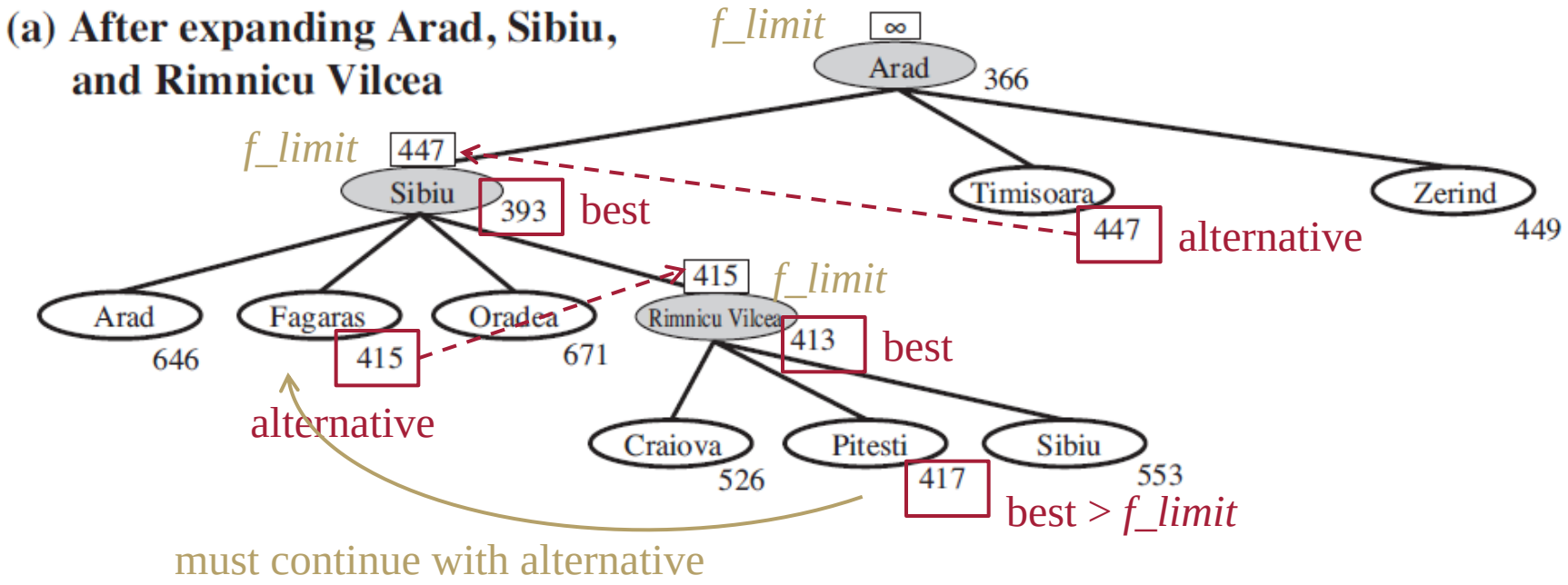
- Simple recursive algorithm that tries to mimic the operation of standard best-first search in linear space (see algorithm on next page)
- It is similar to recursive DFS, but uses a variable *f_limit* to keep track of the best alternative path available from any ancestor of the current node.
- If the current node exceeds *f_limit*, the recursion unwinds back to the alternative path. Then, RBFS replaces the *f*-value of each node on the path with a backed-up value, the best value of its children.
- In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and may decide to re-expand it later.

Recursive Best-First Search

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

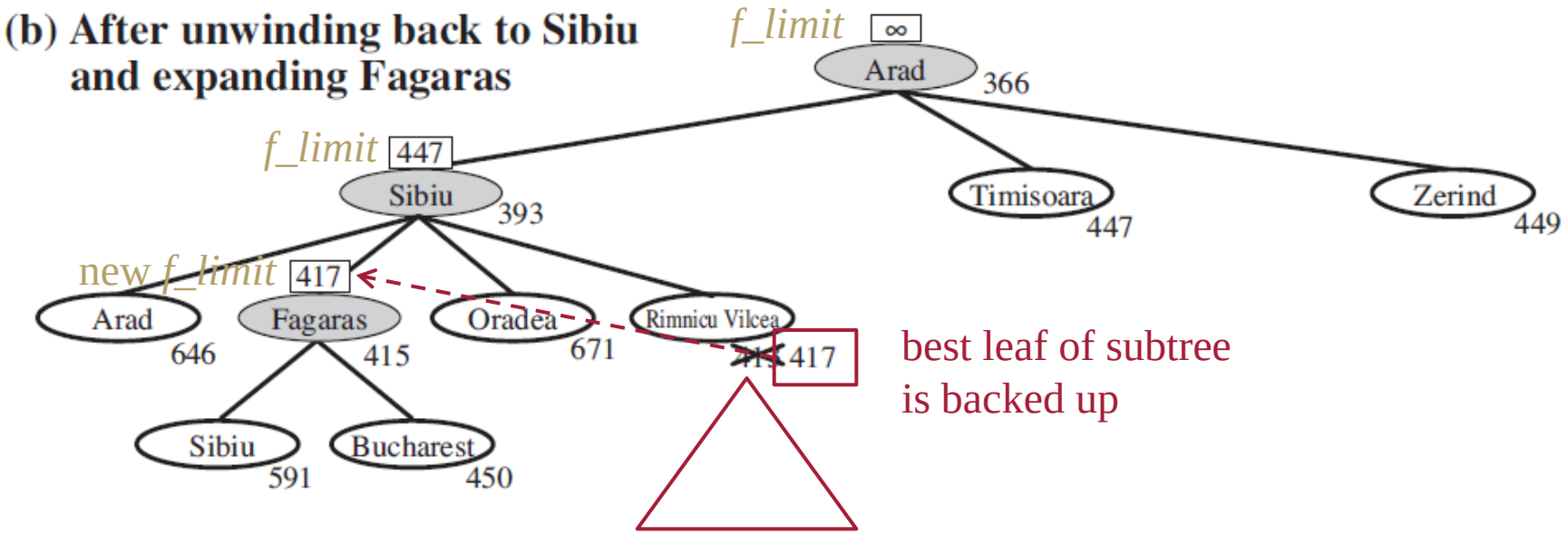
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

RBFS Search AIMA example



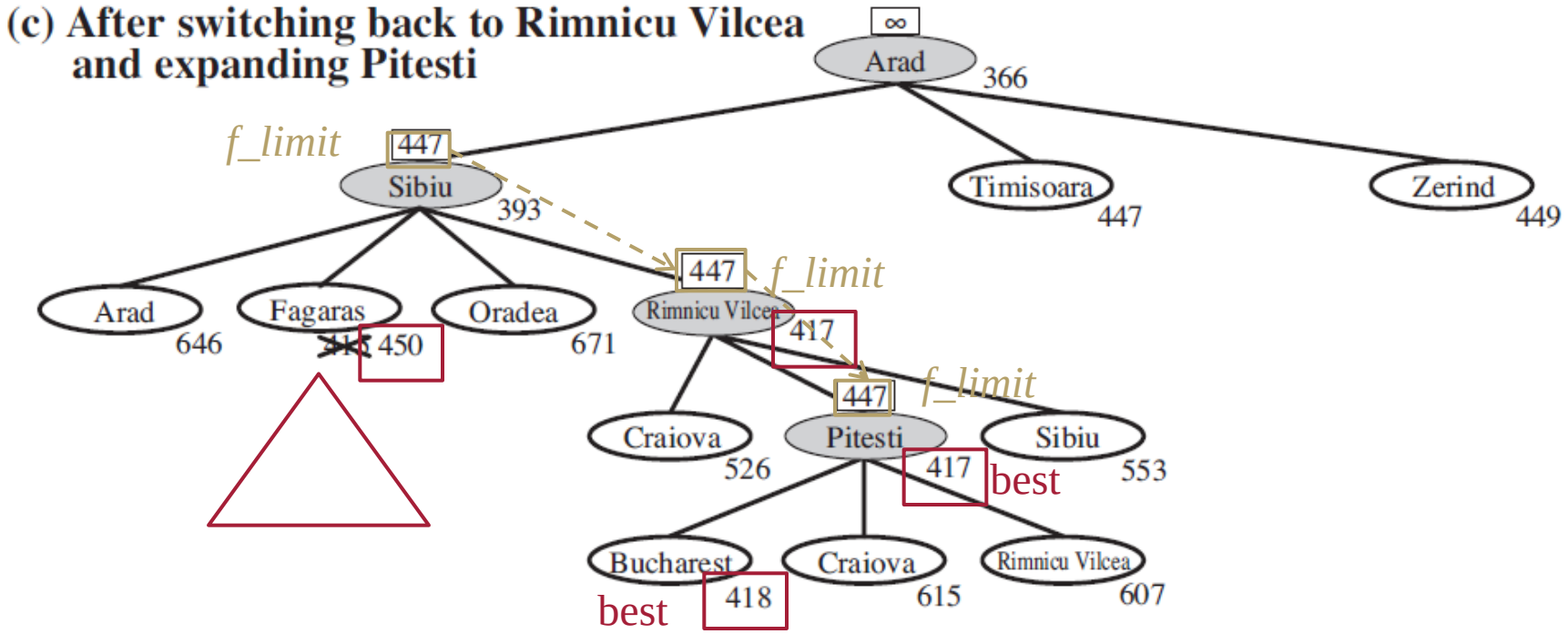
Stages in an RBFS search for the shortest route to Bucharest. The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

RBFS Search AIMA example



(b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450.

RBFS Search AIMA example



(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Simplified Memory-bounded A* (SMA*)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- IDA* and RBFS use too little memory
  - IDA* retains only 1 number between iterations (*f*-cost limit)
  - RBFS retains more information, but uses only linear space
- SMA* proceeds just like A*, expanding the best leaf until memory is full. Then it must drop an old node.
- SMA* always drops the worst leaf node (highest *f*-value). (In case of ties with same *f*-value, SMA* expands the newest leaf and deletes the oldest leaf)
- Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way the ancestor of a forgotten subtree knows the quality of the best path in that subtree.

3.6 Heuristic Functions

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- The quality of any heuristic search algorithm depends on its heuristic
- Two admissible heuristics for the 8-puzzle:
  - h_1 = # misplaced tiles
  - h_2 = sum of Manhattan dist. of all tiles to goal position
  - h_2 is better for search than h_1. See Fig. 3.29 in book.
- The effective branching factor *b** may characterize the quality of a heuristics:
  - If *N* is the # nodes generated by a search algorithm and *d* is the solution depth, then *b** ist the branching factor, which a uniform tree of depth *d* would need to have to contain *N*+1 nodes. Thus

$$N+1=1+b^*+(b^*)^2+\ldots+(b^*)^d$$

Chapter 3 Summary 1

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Search is used in environments which are deterministic, observable, static and completely known.
- First a goal must be identified and a well-defined problem must be formulated.
- A problem consists of five parts: the initial state, a set of actions, a transition function describing the results of actions, a goal test function and a path cost function.
- The environment of the problem is represented by a state space. A path from initial state to a goal state is a solution.
- Search algorithms treat states and actions as atomic, they do not consider their internal structure.
- The Tree-Search algorithm considers all possible paths to find a solution, whereas Graph-Search avoids redundant paths.

Chapter 3 Summary 2

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Search algorithms are judged on the basis of completeness, optimality, time and space complexity.
- Uninformed search methods have access to only the problem definition (no heuristics).
  - Breadth-first search expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.
  - Uniform-cost search expands the node with the longest path cost, g(n), and is optimal for general step costs.
  - Depth-first search expands the deepest unexpanded node first. It is neither complete nor optimal but has linear space complexity.
  - Iterative deepening search calls DFS with increasing depth limits. It is complete, optimal for unit step costs, has time complexity comparable to DFS and linear space complexity.
  - Bidirectional search may reduce time complexity enormously, but is not always applicable and may require too much space.

Chapter 3 Summary 3

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Informed search methods have access to a heuristic function $h(n)$ that estimates the cost of a solution from n.
  - The generic best-first search selects a node for expansion according to an evaluation function.
  - Greedy BFS expands nodes with minimal $h(n)$. It is not optimal.
  - A* search expands nodes with minimal $f(n) = g(n) + h(n)$. It is complete and optimal, if $h(n)$ is admissible (for tree-search) or consistent (for graph-search). Space complexity is exponential
  - RBFS (recursive best-first search) and SMA* (simplified memory-bounded A*) are robust, optimal search algorithms that use limited amounts of memory
- The performance of heuristic search algorithms depends on the quality of their heuristic function.
- Methods to construct good heuristics are: relaxing the problem definition, using a pattern DB with precomputed solution costs, or automated learning from experience.