UNIT – II

CHAPTER – 3

Solving Problems by Searching

Uninformed/Blind Search Strategies

- The term means that the strategies have no additional information about states beyond that provided in the problem definition.
- They can only generate successors and distinguish a goal state from a non-goal state.
- There are various search strategies based on the order in which the nodes are expanded.

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening Depth-first search
6. Bidirectional search

Breadth-first Search

- Is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.

- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

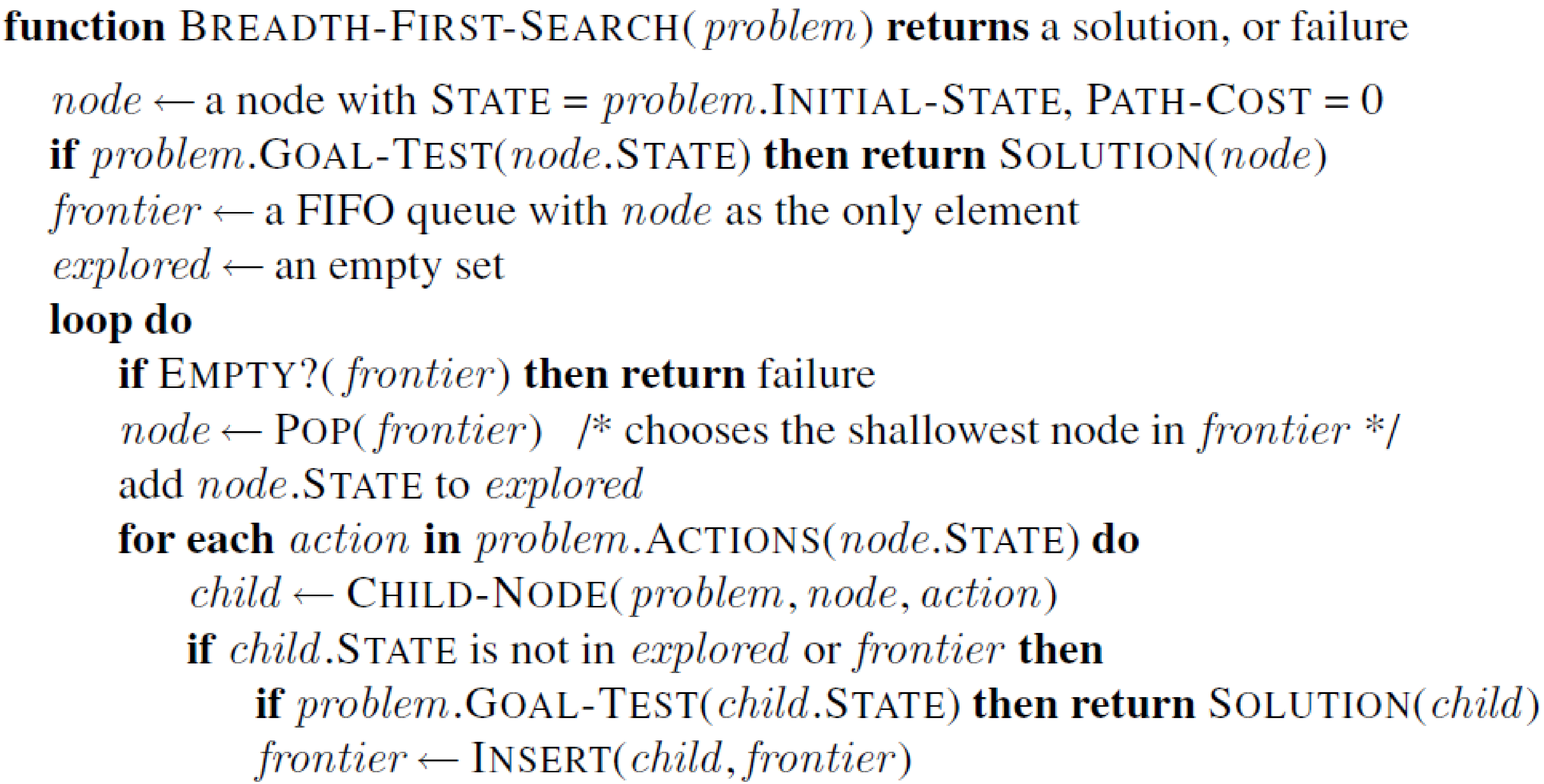
- Breadth-first search is an instance of the general graph-search algorithm in which the ***shallowest*** **unexpanded node is chosen for expansion**.

- This is achieved by using a FIFO queue for the frontier where new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are *shallower* than the new nodes, get expanded first.

Breadth-first Search : Example and Complexity

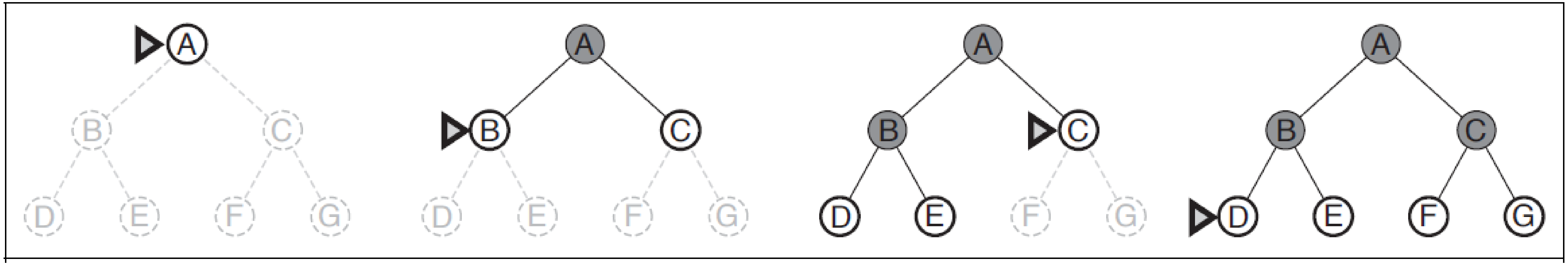
- The goal test is applied to each node when it is generated rather than when it is selected for expansion.
- Thus, breadth-first search always has the shallowest path to every node on the frontier.
- By Assuming uniform breadth factor ‘*b*’, a tree generates b^d nodes at level ‘*d*’. thus total nodes generated is: $b + b^2 + b^3 + \ldots\ldots + b^d$. Hence, time complexity b is : $O(b^d)$
- If the algorithm applies the goal test to nodes when selected for expansion, then the time complexity is: $O(b^{d+1})$
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier

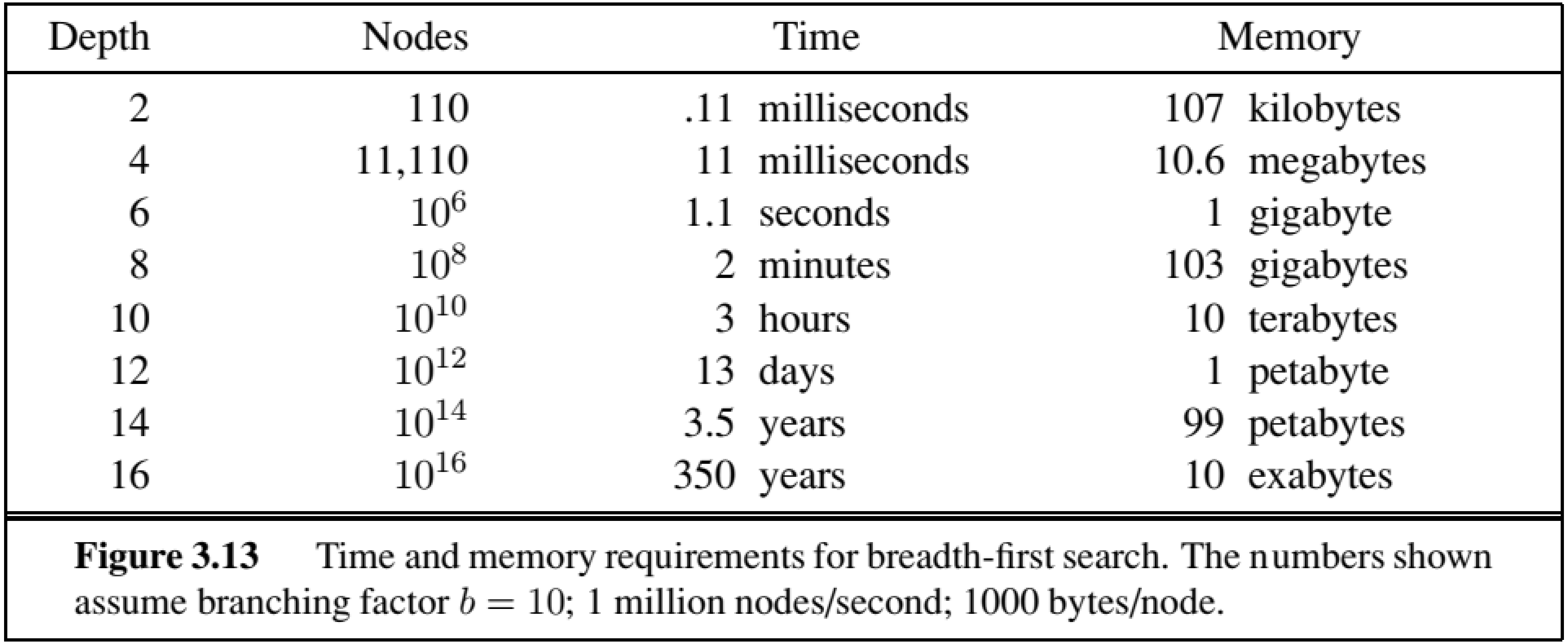
Breadth-first Search : Pseudocode



```
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)
```

BFS on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.





Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Breadth-first Search : Lessons-Learned

- The memory requirements are a bigger problem for breadth-first search than is the execution time.
  - One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take.
- Moreover time is still a major factor. If your problem has a solution at depth 16, then will take about 350 years to find it is not feasible.
- In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Uniform-cost Search

- It is a type of uninformed search that performs a search based on the lowest path cost.
- When all step costs are equal, BFS is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- Instead of expanding the **shallowest** node, **uniform-cost search** expands the node ‘*n*’ with the *lowest path cost denoted as function ‘g(n)’*.
- This is done by storing the frontier as a priority queue ordered by ‘*g*’.

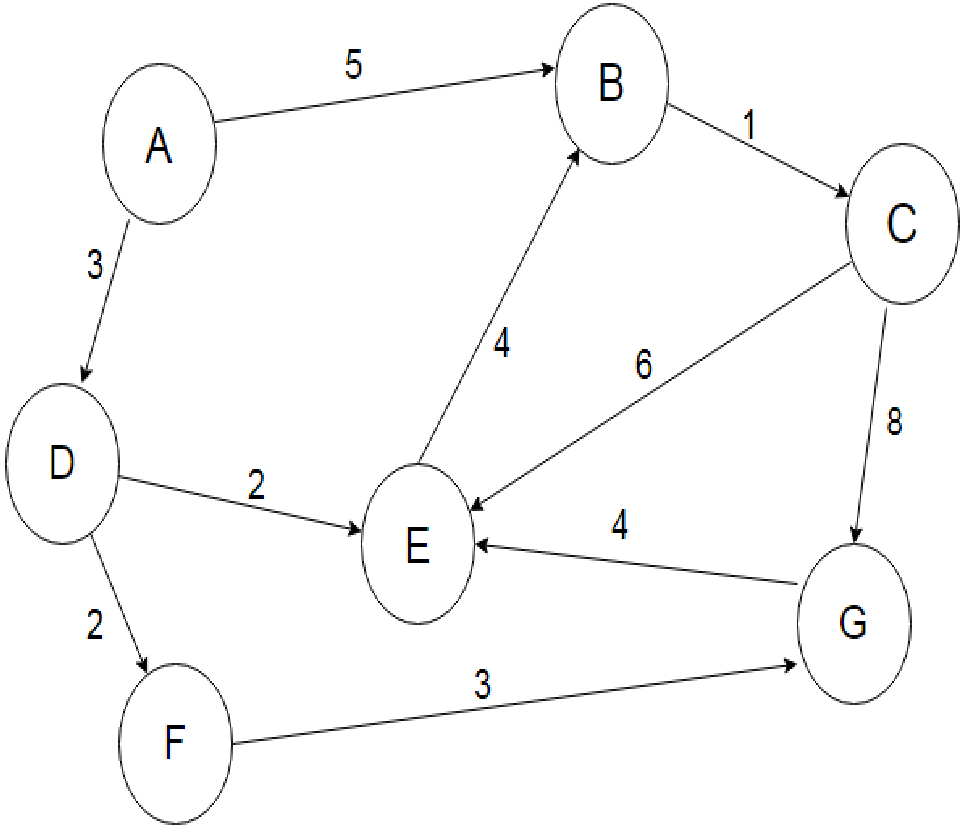
Uniform-cost Search: contd…

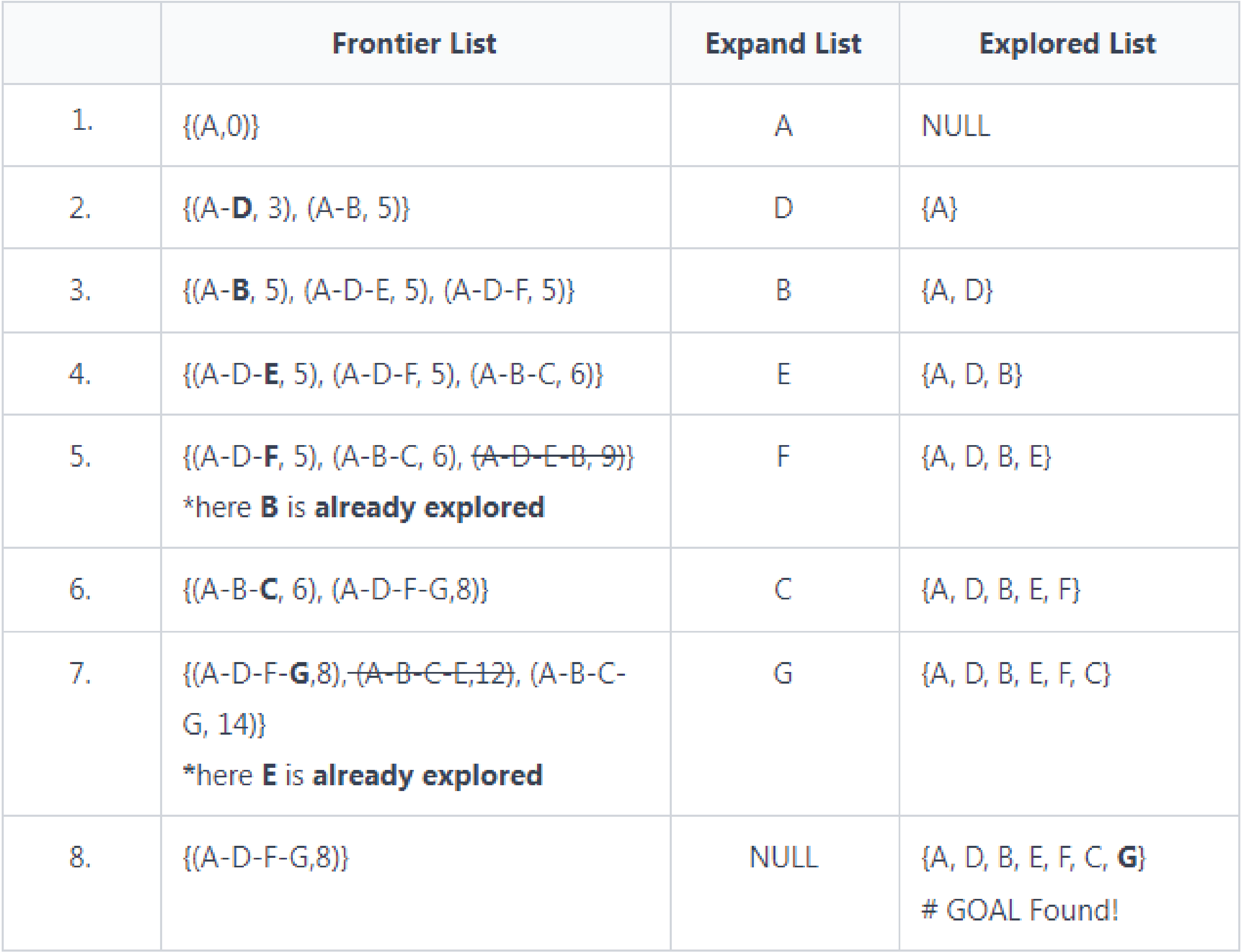
- Frontier list will be based on the priority queue. Every new node will be added at the end of the list and the list will give priority to the least cost path.
- The node at the top of the frontier list will be added to the expand list, which shows that this node is going to be explored in the next step. It will not repeat any node. If the node has already been explored, you can discard it.
- Explored list will be having the nodes list, which will be completely explored.

Uniform-cost Search: contd…

- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.
- The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated.
- The reason is that the first goal node that is generated may be on a suboptimal path.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier.

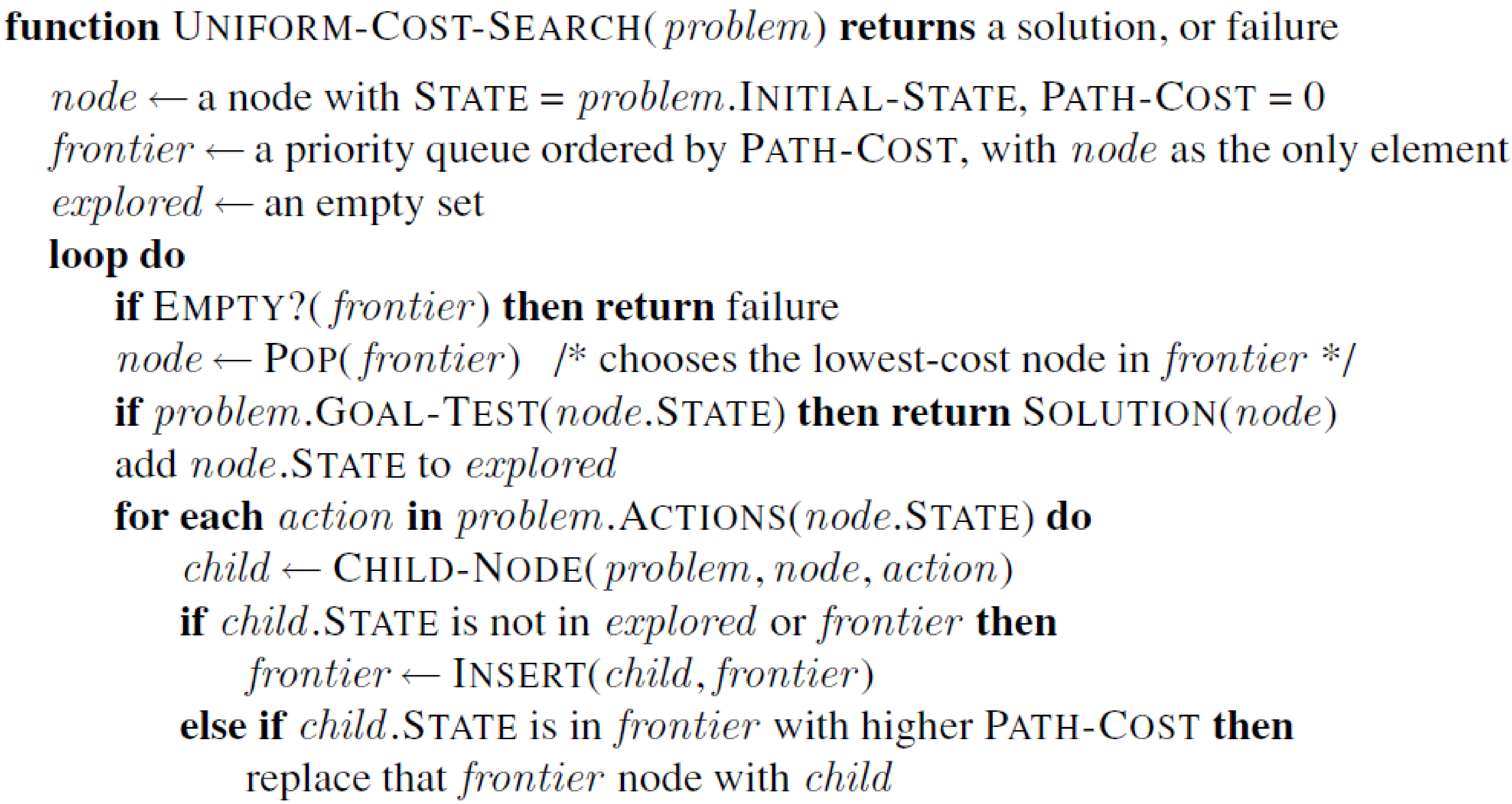
Example





	Frontier List	Expand List	Explored List
1.	{(A,0)}	A	NULL
2.	{(A-**D**, 3), (A-B, 5)}	D	{A}
3.	{(A-**B**, 5), (A-D-E, 5), (A-D-F, 5)}	B	{A, D}
4.	{(A-D-**E**, 5), (A-D-F, 5), (A-B-C, 6)}	E	{A, D, B}
5.	{(A-D-**F**, 5), (A-B-C, 6), ~~(A-D-E-B, 9)~~} *here **B** is **already explored**	F	{A, D, B, E}
6.	{(A-B-**C**, 6), (A-D-F-G,8)}	C	{A, D, B, E, F}
7.	{(A-D-F-**G**,8)~~, (A-B-C-E,12)~~, (A-B-C-G, 14)} *here **E** is **already explored**	G	{A, D, B, E, F, C}
8.	{(A-D-F-G,8)}	NULL	{A, D, B, E, F, C, **G**} # GOAL Found!

Uniform-cost Search : Pseudocode



```
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 : Example and Complexity

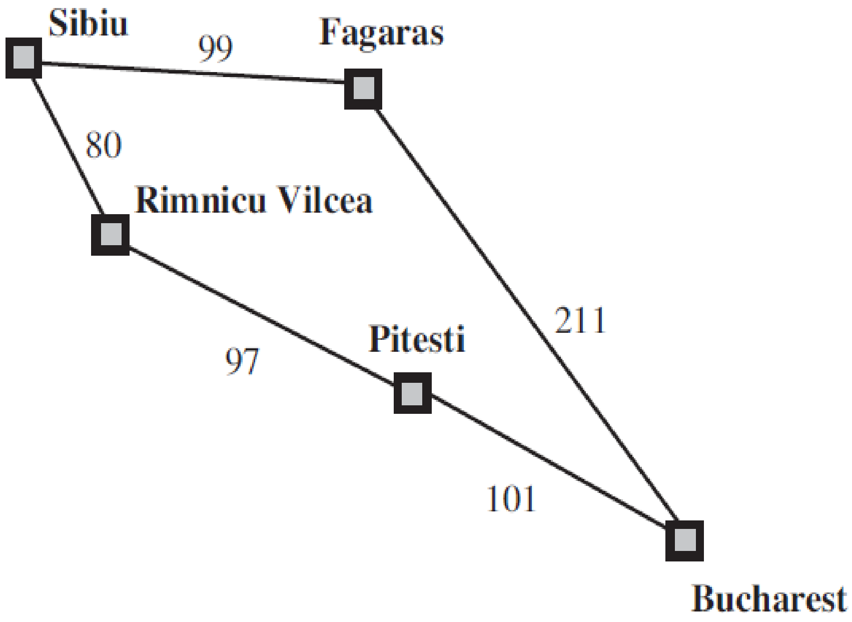
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.

- Instead, let $\mathbf{C^*}$ be the cost of the optimal solution, and assume that every action costs at least ε.

- Then the algorithm's worst-case time and space complexity is $\mathbf{O(b^{1+ C*/\varepsilon})}$ which can be much greater than b^d

- When all step costs are the same, uniform-cost search is similar to breadth-first search

- The latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost



- Thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily

Evaluating Uniform Cost Search

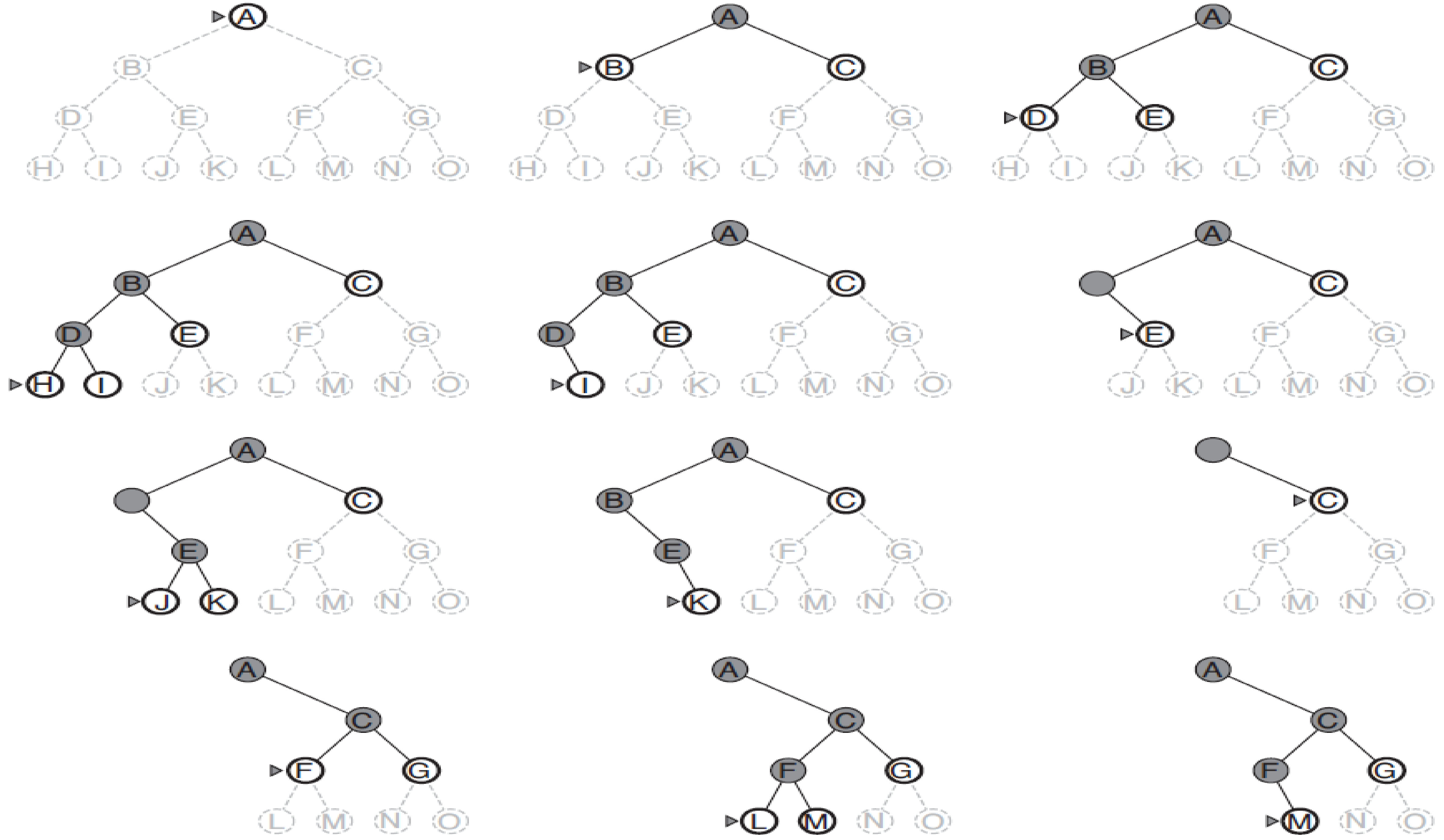
- The algorithm can be evaluated by four of the following factors:
- **Completeness**: Uniform Cost Search is complete if the branching factor **b** is finite.
- **Time complexity:** The time complexity of Uniform Cost Search is exponential $O(\mathbf{b}^{(1+C/\varepsilon)})$, because every node is checked.
- **Space completeness**: The space complexity is also exponential $O(\mathbf{b}^{(1+C/\varepsilon)})$,because all the nodes are added to the list for comparisons of priority.
- **Optimality**: Uniform Cost Search gives the optimal solution or path to the goal node.

Depth-first Search

- **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- It uses a LIFO queue for its implementation

Depth-first Search : Example & Facts

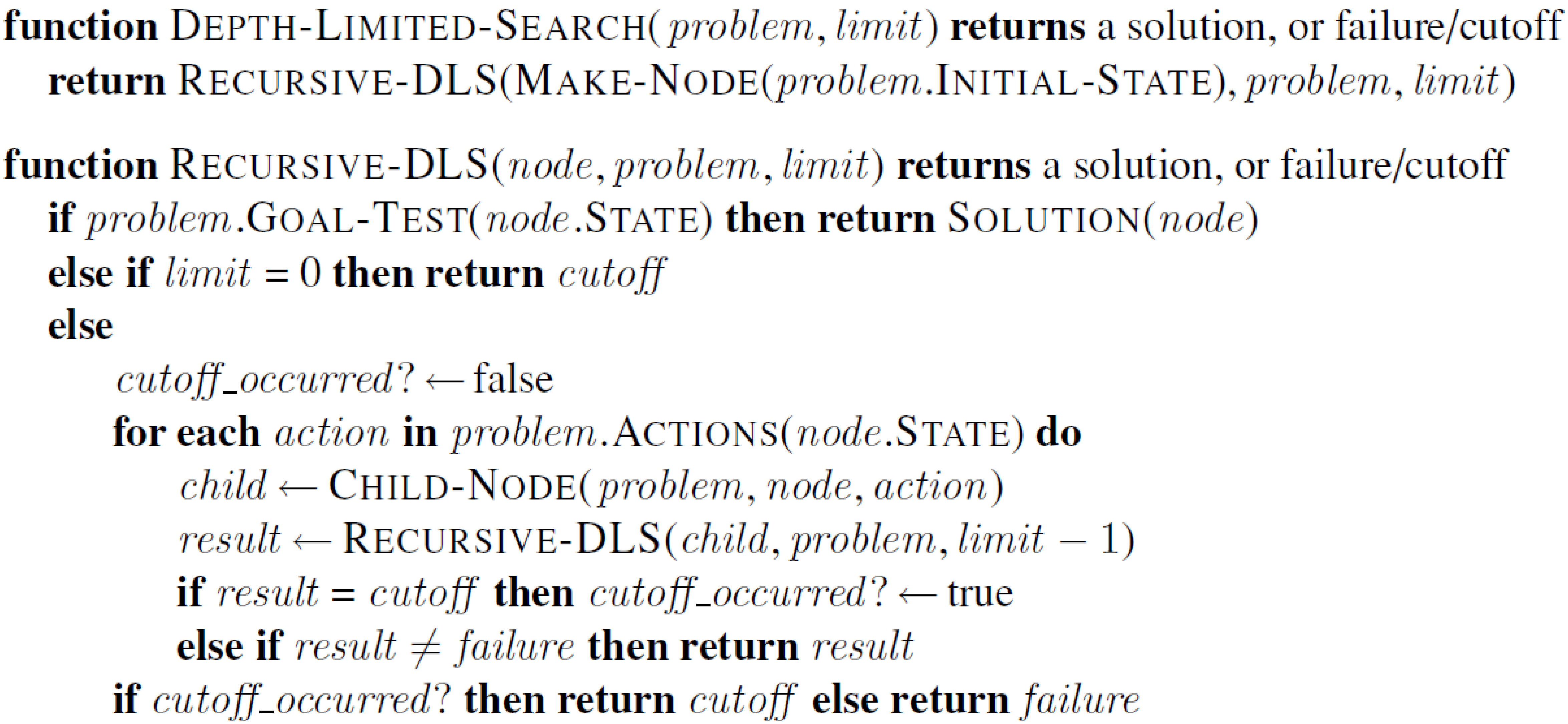
- Fig : Depth-first search on a binary tree.
- The unexplored region is shown in light Gray.
- Explored nodes with no descendants in the frontier are removed from memory.
- Nodes at depth 3 have no successors and **M** is the only goal node.
- Time Complexity is O(b^m) and Space Complexity is O(bm) where *m* is the max depth level of tree and b is the breadth factor
- DFS will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.
- Depth-first Search would require 156 kilobytes instead of 10 exabytes at depth d = 16, a factor of 7 trillion times less space.
- Hence it is better than Breadth First Search in terms of space and so used in several AI methods.


A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G I J K L M N O
A C E F G J K L M N O
A C E F G J K L M N O
A B C E F G K L M N O
C F G L M N O
A C F G L M N O
A C F G L M N O
A C F G M N O

Depth-limited Search

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ***'l'***.
- That is, nodes at depth ***'l'*** are treated as if they have no successors. This approach is called depth-limited search.
- The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit.
- Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$
- Time Complexity is $O(b^l)$ and Space Complexity is $O(bl)$ where *l* is the max depth limit imposed on tree and b is the breadth factor.

Depth-limited Search : Pseudocode for recursive implementation of depth-limited tree search



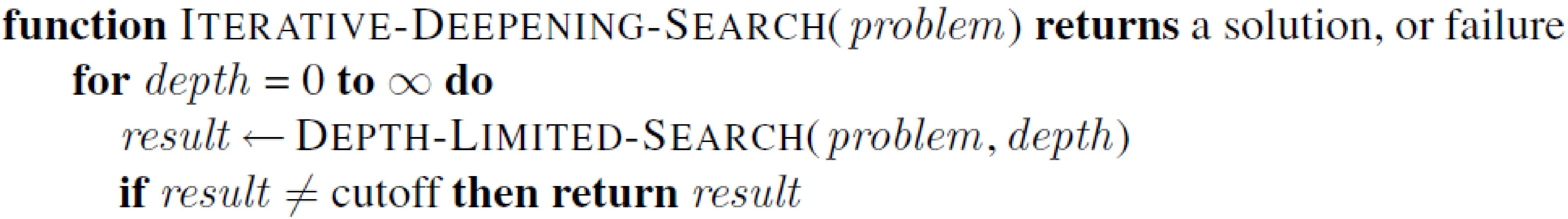
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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
```

Iterative Deepening Depth-First Search

- **Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Iterative deepening combines the benefits of depth-first and breadth-first search.
    - Like depth-first search, its memory requirements are modest: O(bd) to be precise.
    - Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

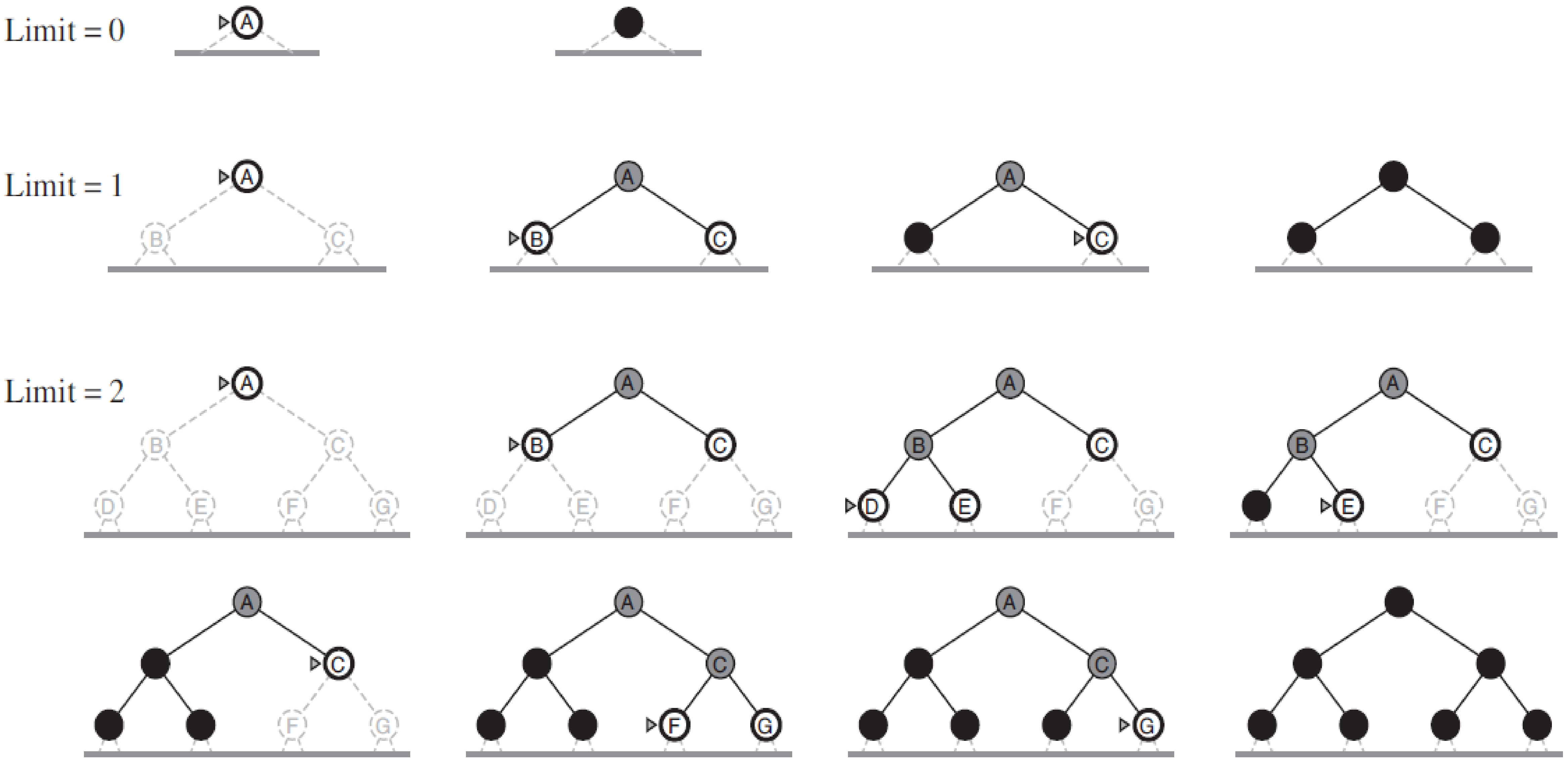
Iterative Deepening Depth-First Search: Pseudocode

Pseudocode for the iterative deepening search algorithm, which 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.

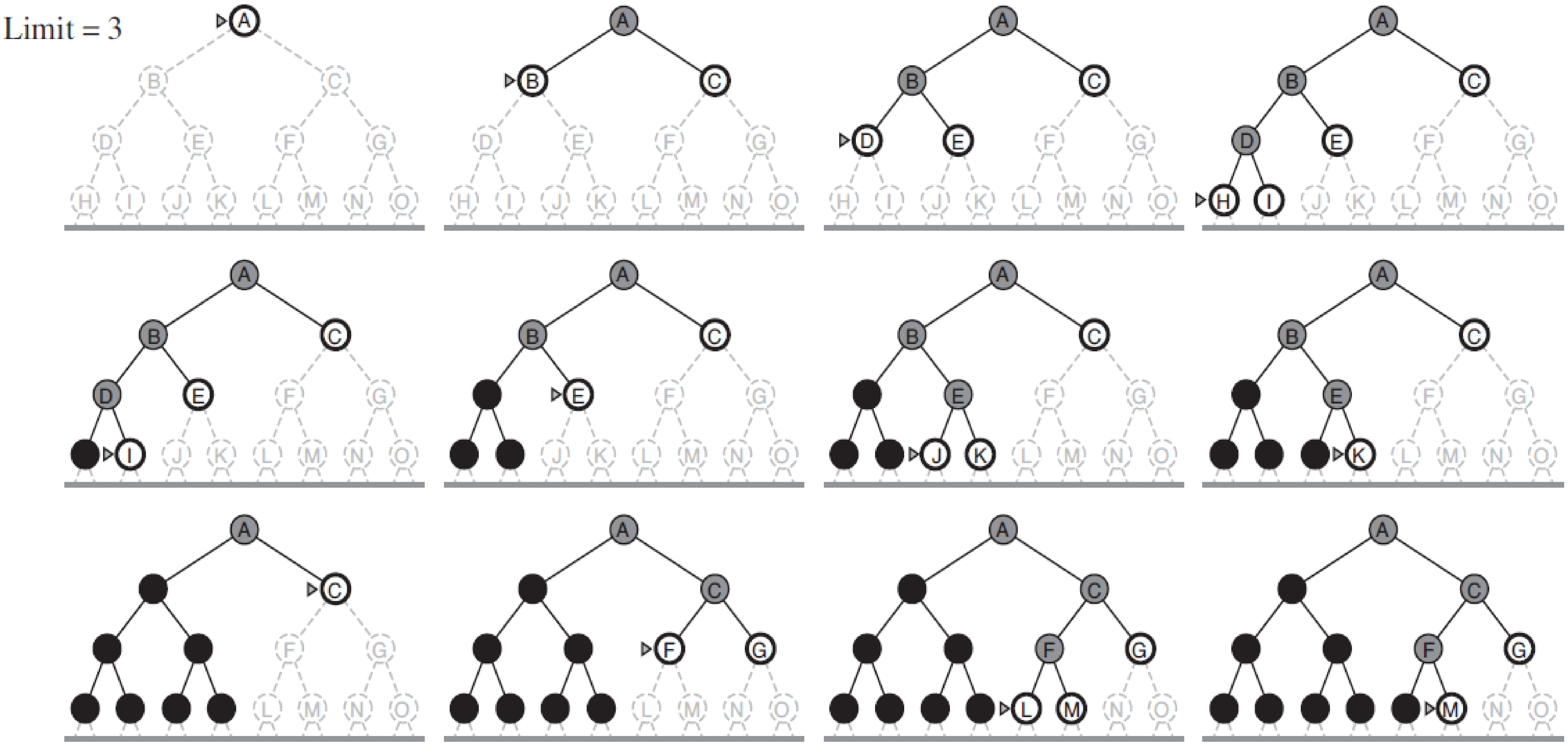


```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to $\infty$ do
        result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
        if result $\neq$ cutoff then return result
```

Four iterations of iterative deepening search on a binary tree.



Four iterations of iterative deepening search on a binary tree Contd..



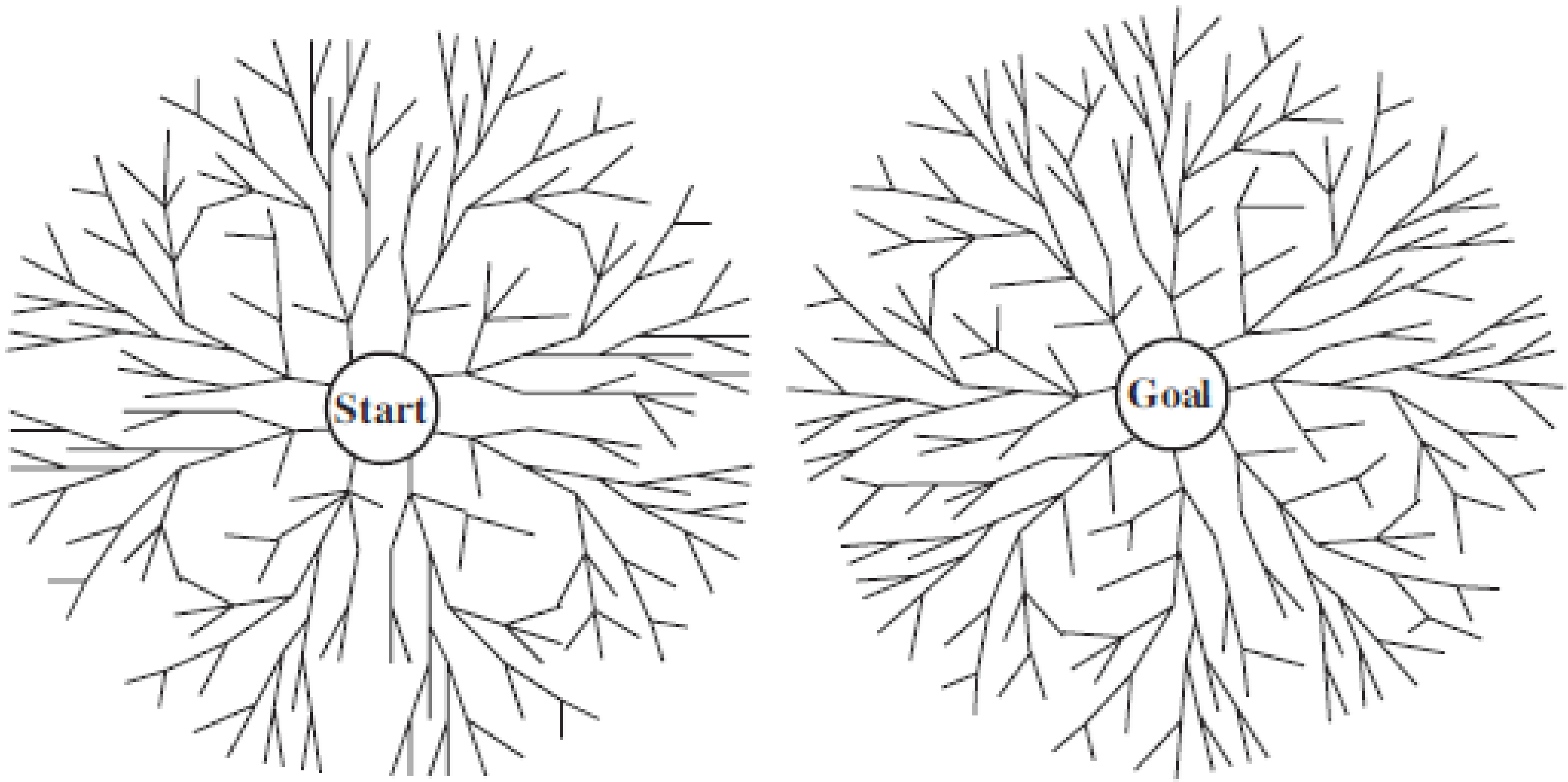
Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches one forward from the initial state the other backward from the goal hoping that the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
- Or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

The reduction in time complexity makes bidirectional search attractive, but how do we search backward? : This is not as easy as it sounds.

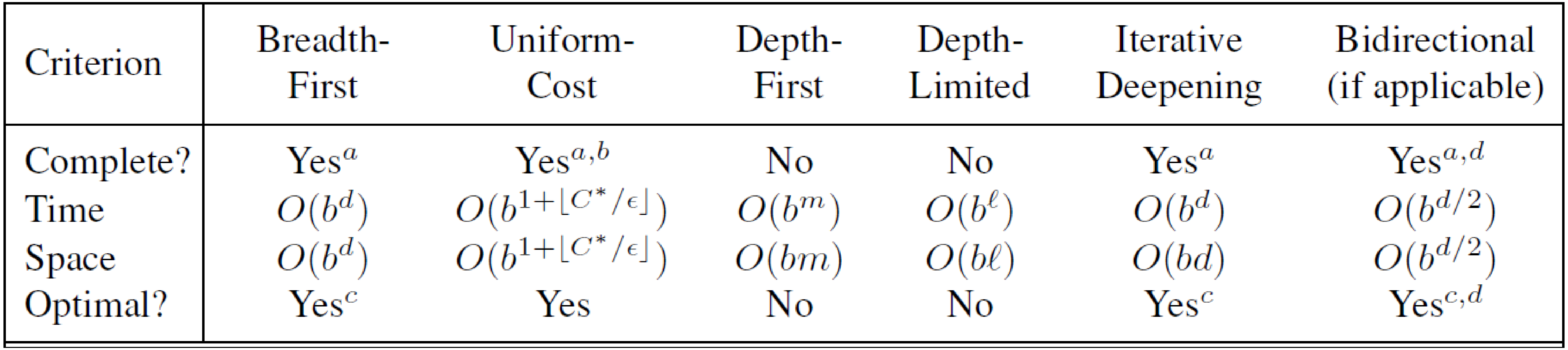
Let the **predecessors** of a state '*x*' be all those states that have '*x*' as a successor. Bidirectional search requires a method for computing predecessors. When all the actions in the state space are reversible, the predecessors of '*x*' are just its successors. Other cases may require substantial ingenuity.

Bidirectional Search



A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

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.

Superscript caveats are as follows:
a complete if b is finite;
b complete if step costs ≥ for positive ;
c optimal if step costs are all identical;
d if both directions use breadth-first search.

Informed/Heuristics Search

- Literal meaning: Heuristics are mental shortcuts that can facilitate problem-solving It is also called **rules-of-thumb**, that reduce cognitive load. They can be effective for making immediate judgments, may not be optimal.
- In the context of artificial intelligence, a heuristic is a strategy or method that guides the search for solutions in a more **efficient way** than **exhaustive search** methods, by making educated guesses or applying practical rules of thumb.

Informed/Heuristics Search contd…

- An **informed search** strategy is one that uses **problem-specific knowledge** beyond the definition of the problem itself can find solutions more efficiently than can an uninformed strategy.
- Additional information of the domain is known as heuristics and is denoted as h(n)
- Generally *h(n)* = estimated cost of the cheapest path from the state at node n to a goal state.
- We are studying two strategies under this category
  - ✓ Greedy Best-First Search
  - ✓ A* Search

Greedy Best-first Search

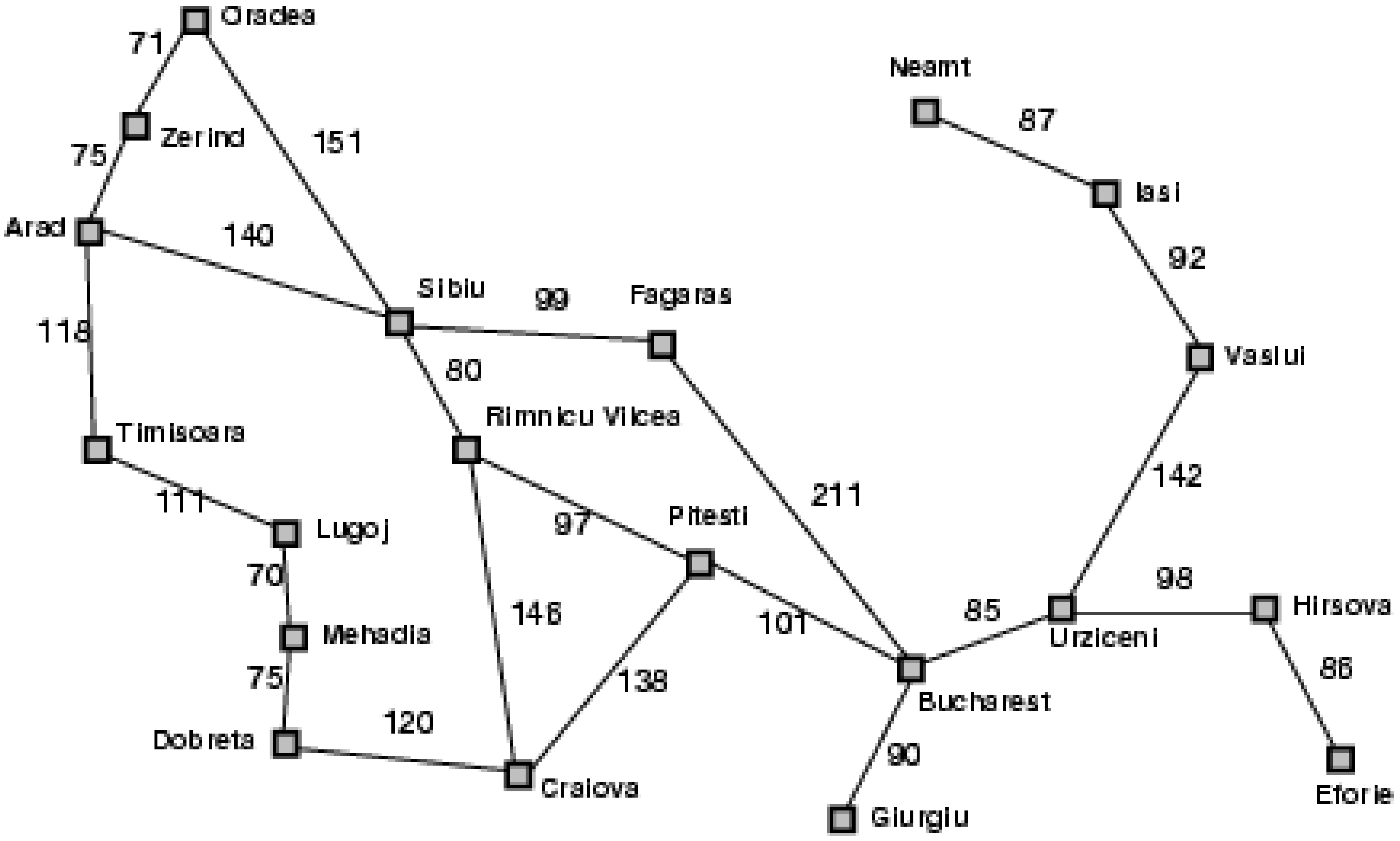
- **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$

- Problem solving strategy is similar to the uniform-cost search, but only the difference is that the greedy best search uses the *h(n) [heuristics of n]* as the criteria for selecting the node for further expansion while uniform-cost search uses the *g(n)[actual cost from source to n]* as the criteria.

- Worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space.

Greedy Best-first Search:Example 1

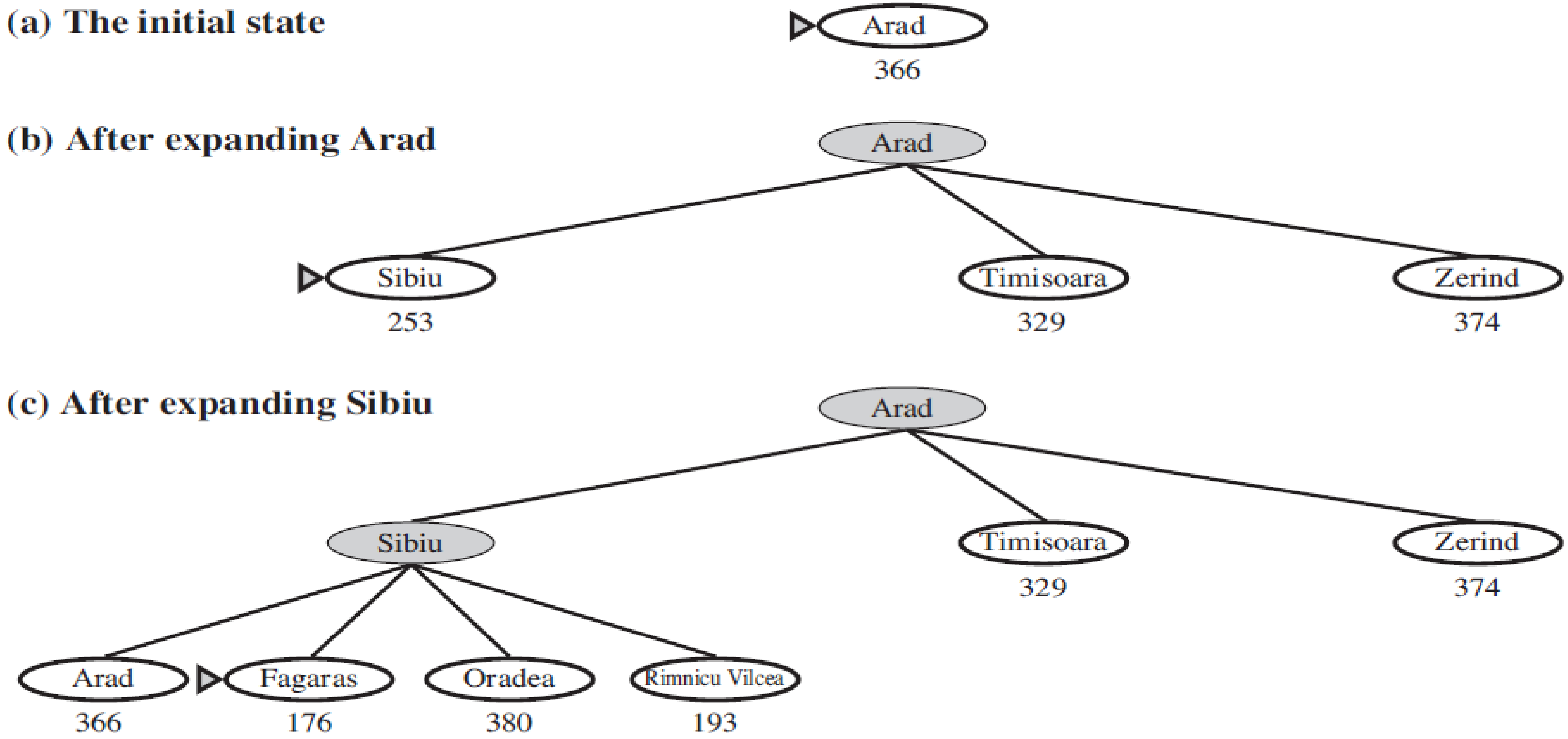
Heuristic Values: Straight line distance between every city to Bucharest



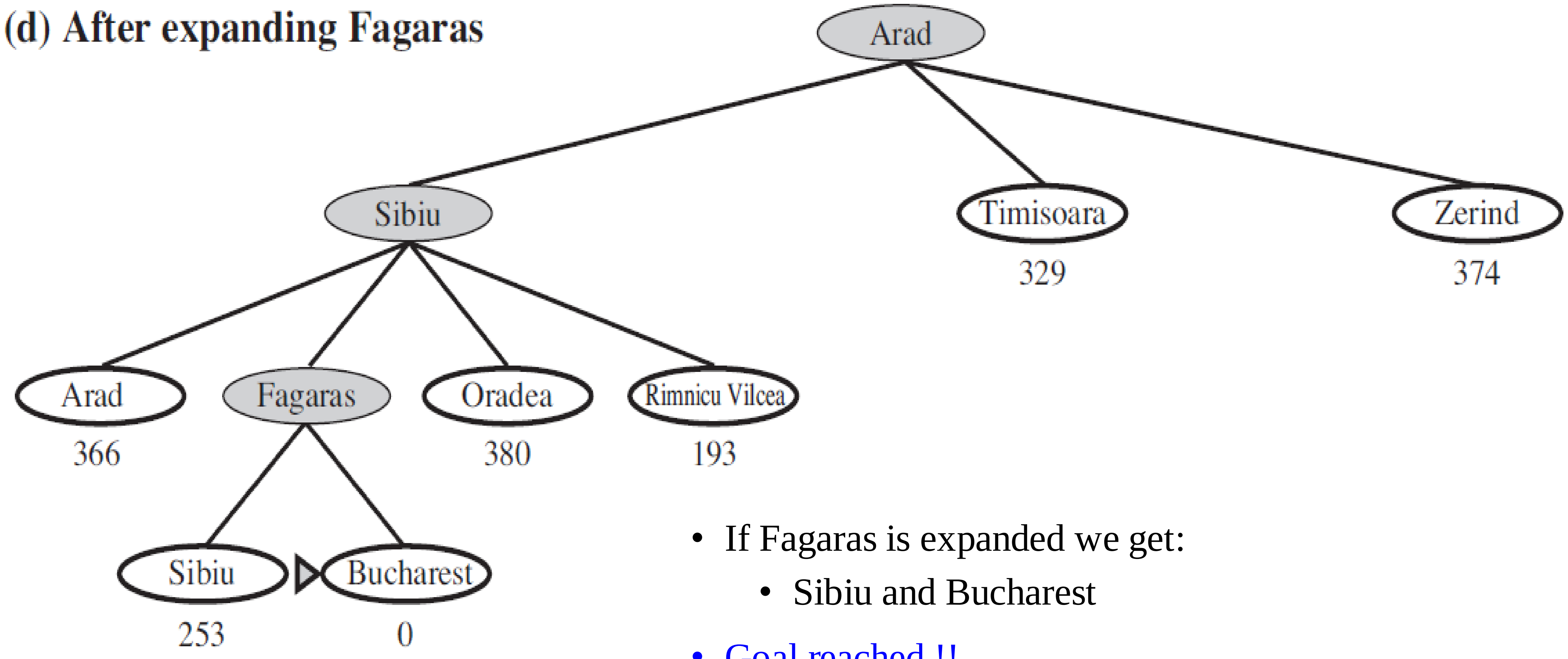
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search:Example

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

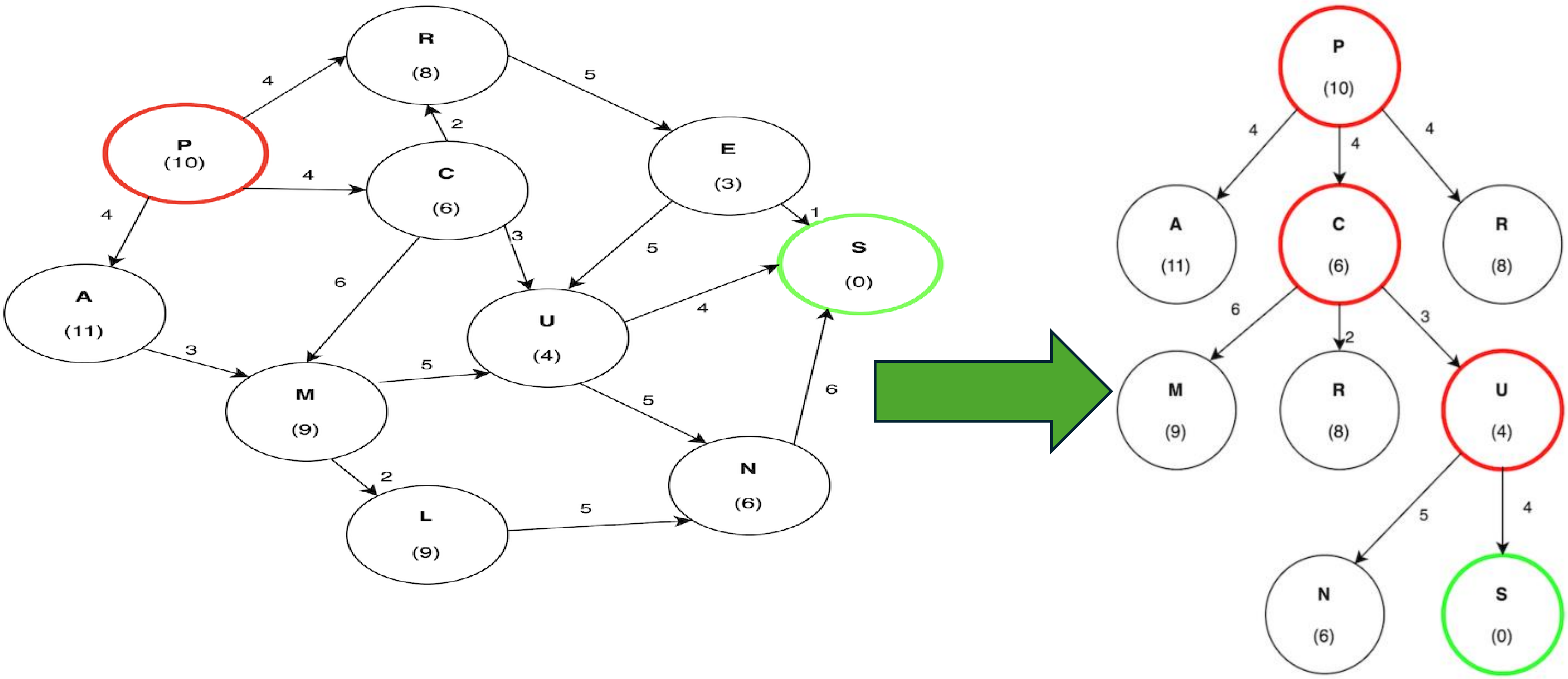


Greedy Best-first Search:Example

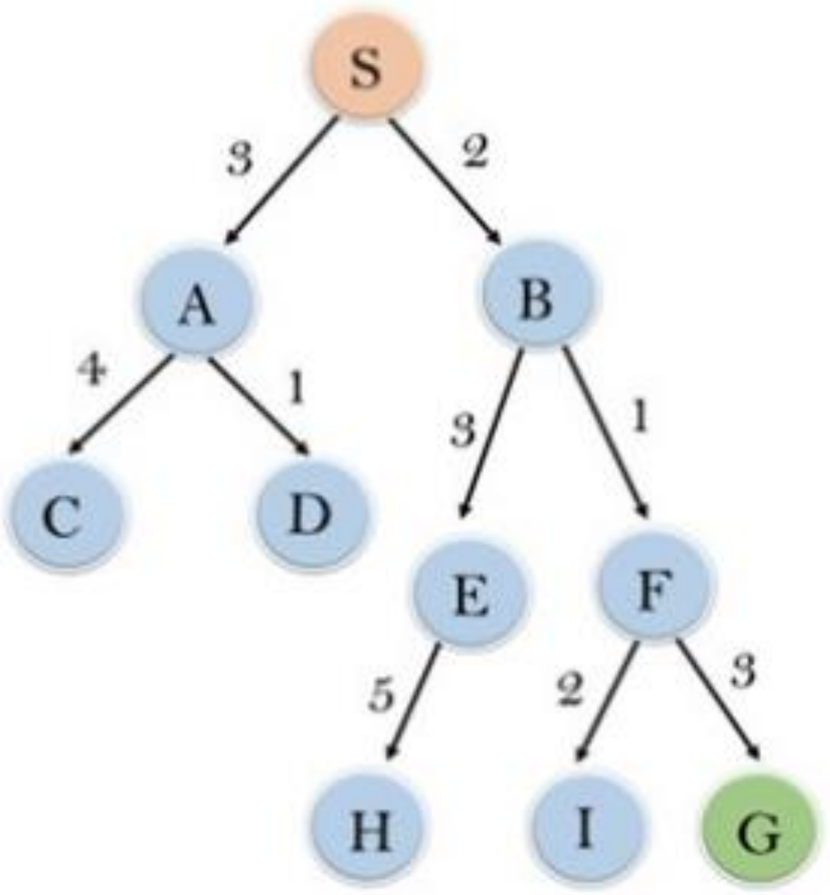


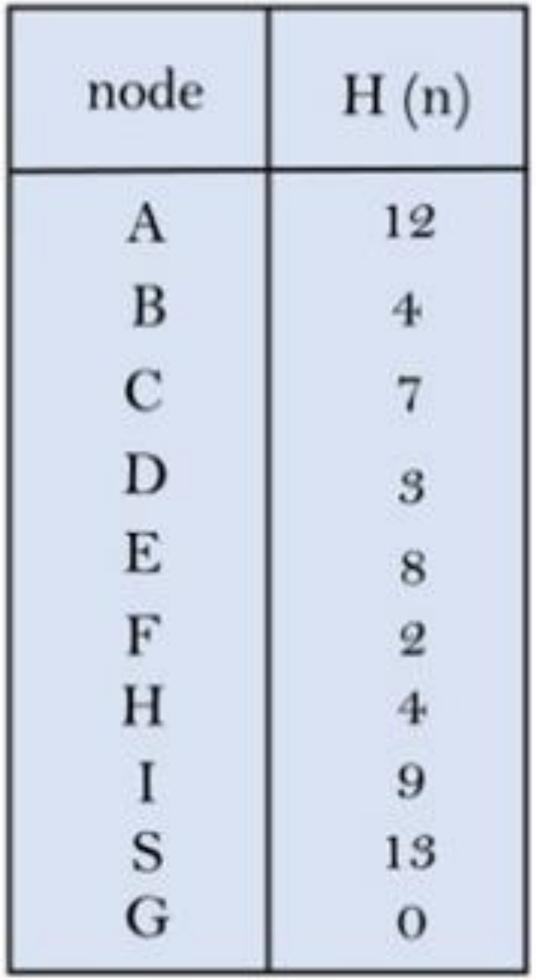
- If Fagaras is expanded we get:
  - Sibiu and Bucharest
- Goal reached !!
  - Yet not optimal (see Arad, Sibiu, Rimnicu Vilcea, Pitesti)

Greedy Best-first Search:Example 2



Greedy Best-first Search:Example 3





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

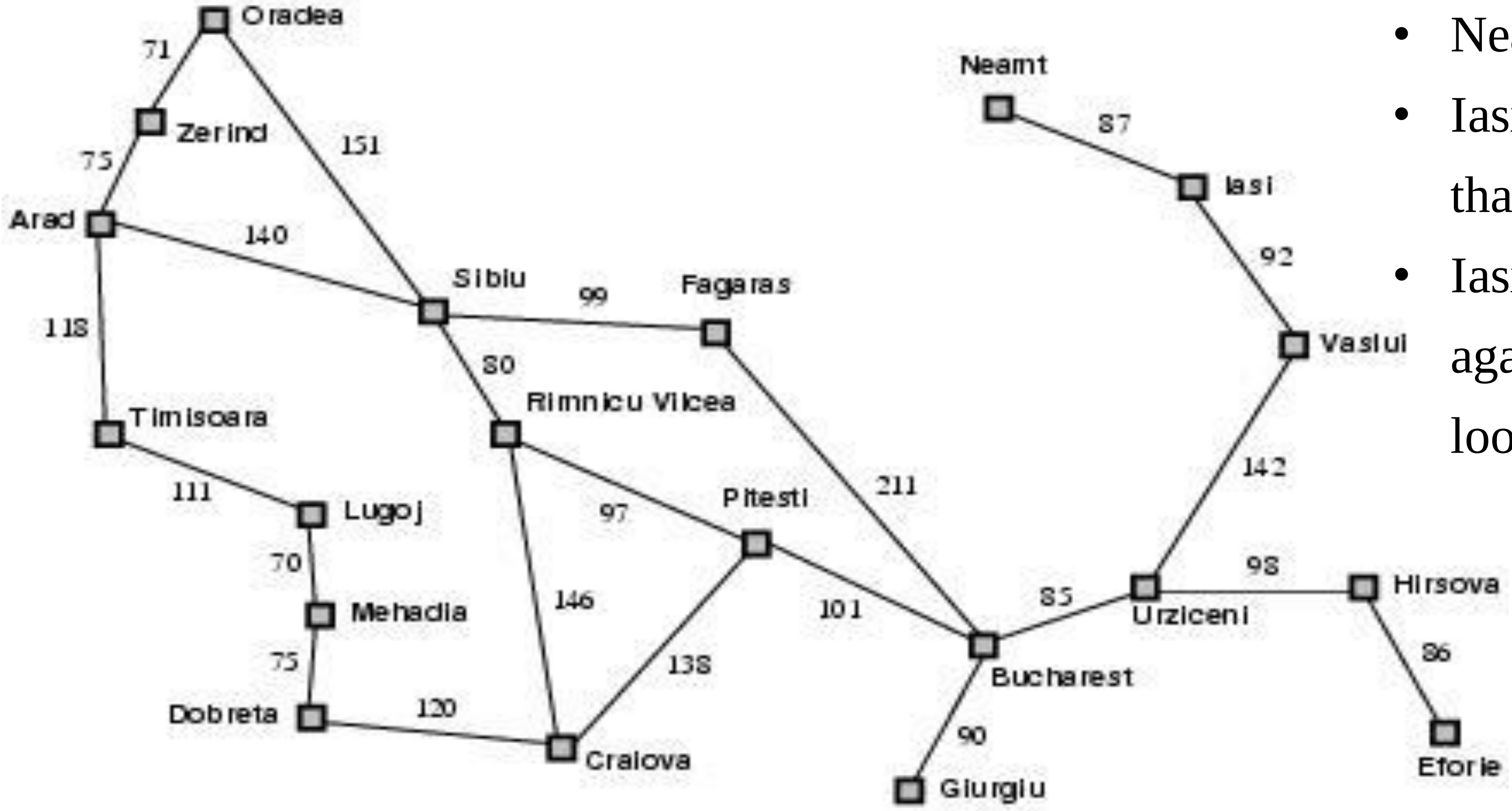
Greedy Best-first Search: Algorithm

1. Initialize a tree with the root node being the start node in the open list.
2. If the open list is empty, return a failure, otherwise, add the current node to the closed list.
    1. Add the child nodes of the current node to the open list.
    2. Remove the node(child) with the lowest **h(x)** value from the open list for exploration, and add it to the closed list.
    3. If a child node is the target, return a success. Otherwise, if the node has not been in either the open or closed list, add it to the open list for exploration.

Properties of Greedy Search

- Completeness: NO
- Optimality: No
- Time and Space complexity
  - ➢ Worst-case : same as DF-search [O(b^d)] (with m maximum depth of search space)
  - ➢ Good heuristic can give dramatic improvement.
  - ➢ Keeps all the nodes in the memory

Properties of Greedy Search: Completeness



- Iasi to Fagaras
- First Neamt will be expanded
- Neamt puts Iasi back
- Iasi is closer to Fagaras than Vaslui
- Iasi will be expanded again, leading to an infinite loop

A* search : Minimizing the total estimated solution cost

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n)=g(n) + h(n)$
  - $g(n)$ the cost **(so far)** to reach the node
  - $h(n)$ **estimated cost** to get from the node to the goal
  - $f(n)$ estimated total cost of path through n to goal

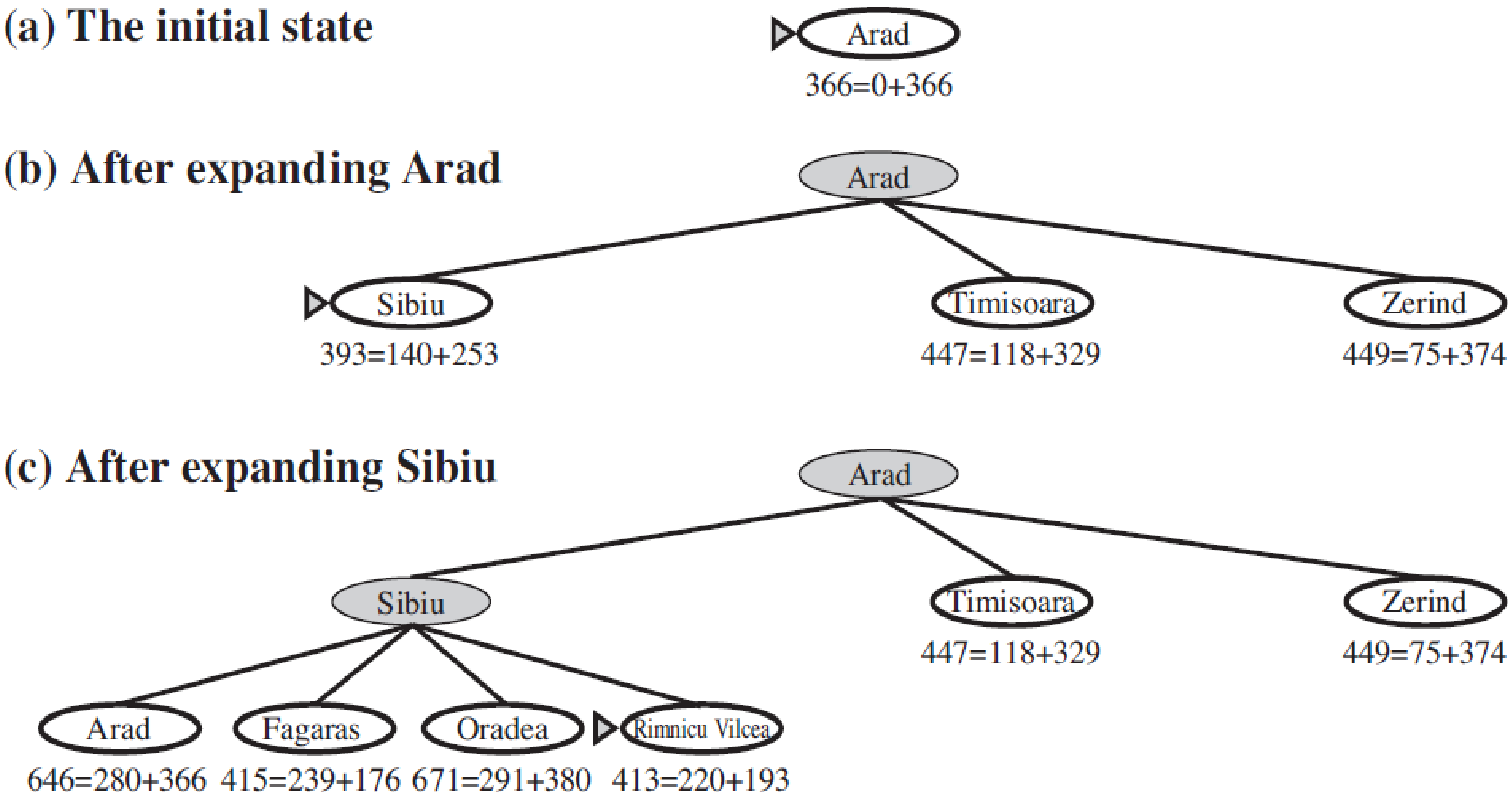
A* search procedure

g'(n'):Actual cost to reach n'
h(n'):Heuristic at n'

1. Put the start node s in *Frontier*.
2. If *Frontier* is empty, exit with failure.
3. Remove from *Frontier* and place in *Explore* a node n for which f(n) is minimum.
4. If n is a goal node, exit with the solution obtained by tracing back pointers from n to s.
5. Expand n, generating all of its successors. For each successor n' of n:
    1. Compute g'(n'); compute f'(n')=g'(n')+h(n')
    2. if n' is already on *Frontier* or *Explore* and g'(n')<g(n'), let g(n')=g'(n'), let f(n')=f'(n'), redirect the pointer from n' to n and, if n' is on *Explore*, move it to *Frontier*.
    3. if n' is neither on *Frontier* nor on *Explore*, let f(n')=f'(n'), attach a pointer from n' to n, and place n' on *Frontier*.
6. Go to 2.

A* Search Example-1

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



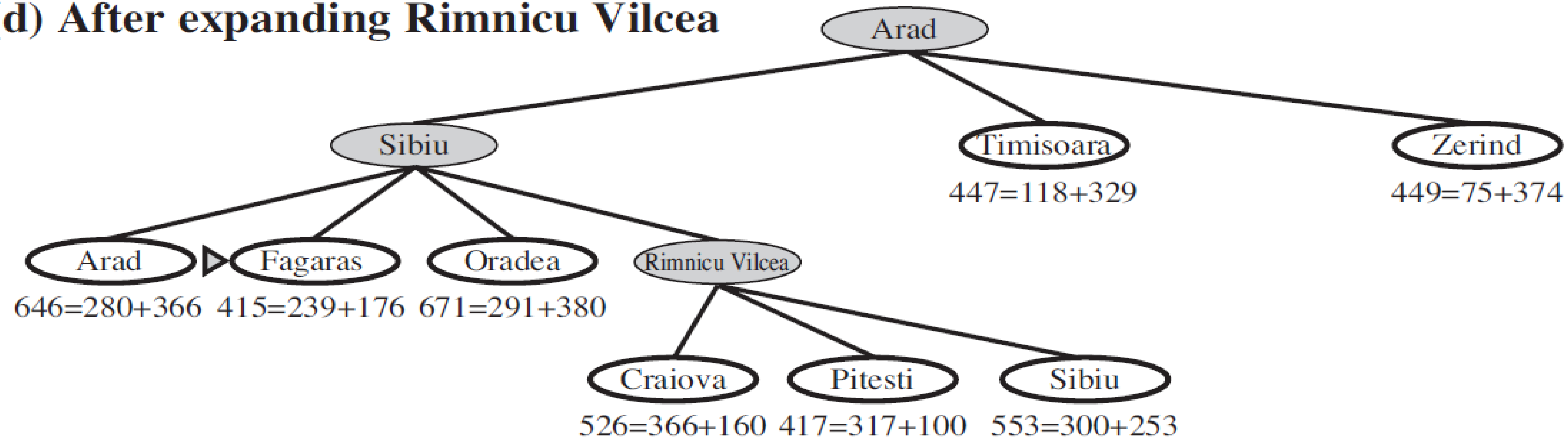
(a) The initial state

(b) After expanding Arad

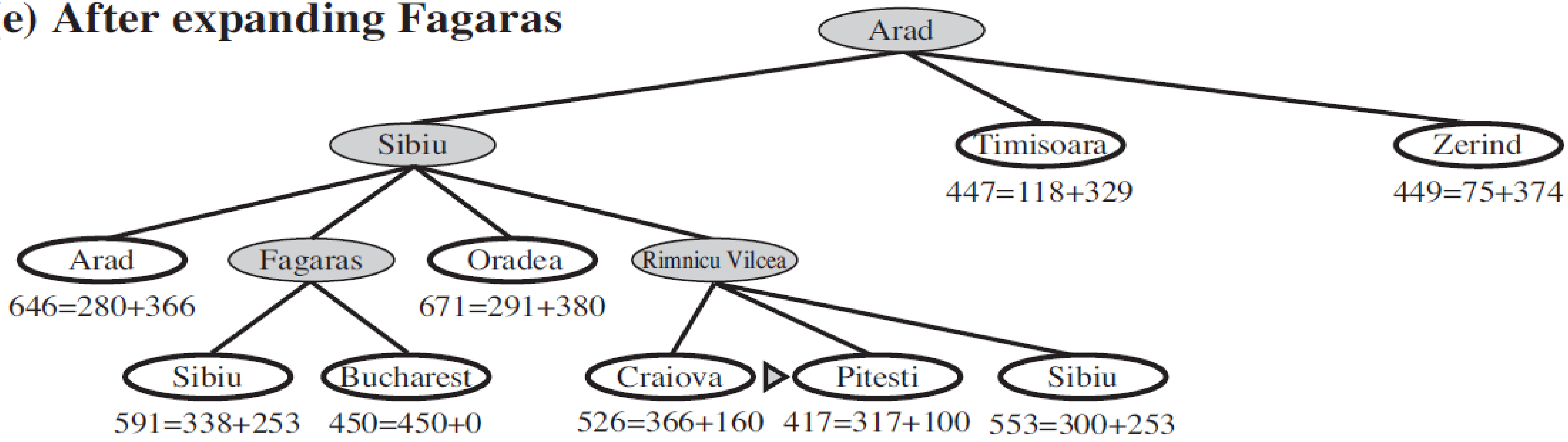
(c) After expanding Sibiu

A* Search Example Contd…

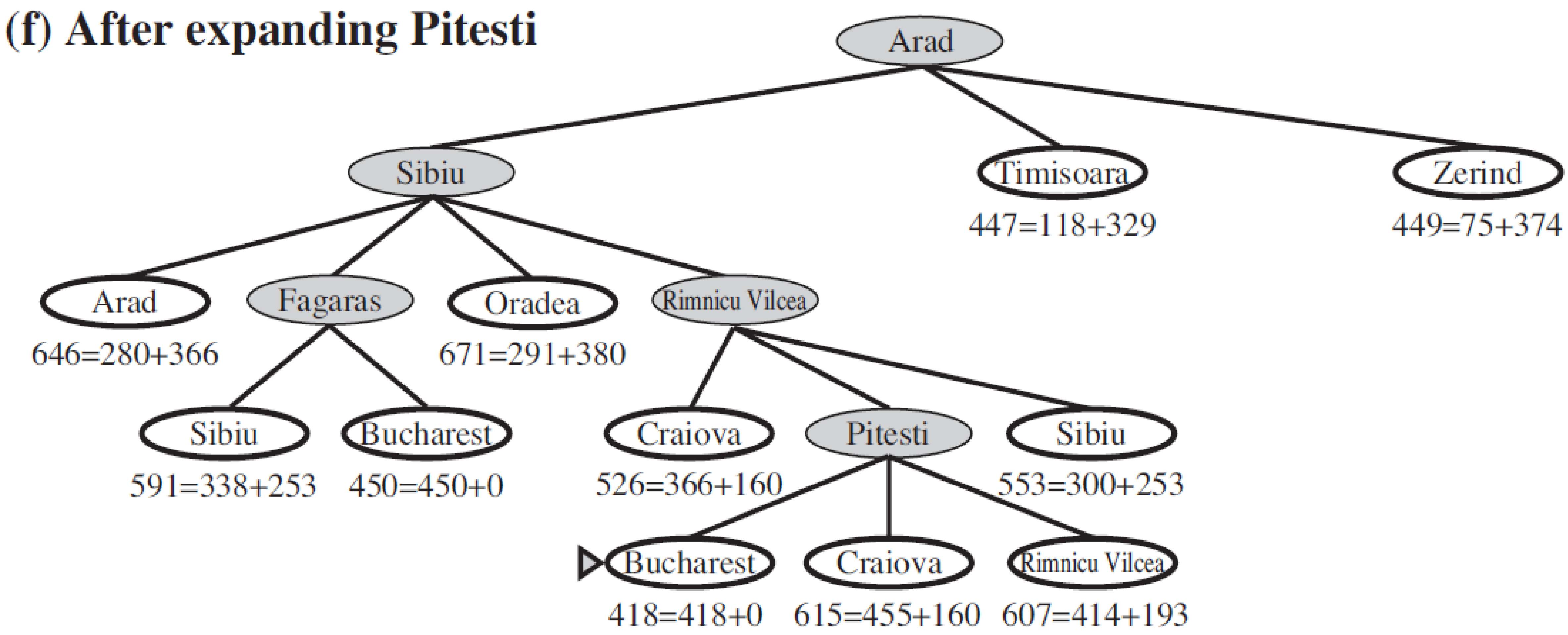
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras

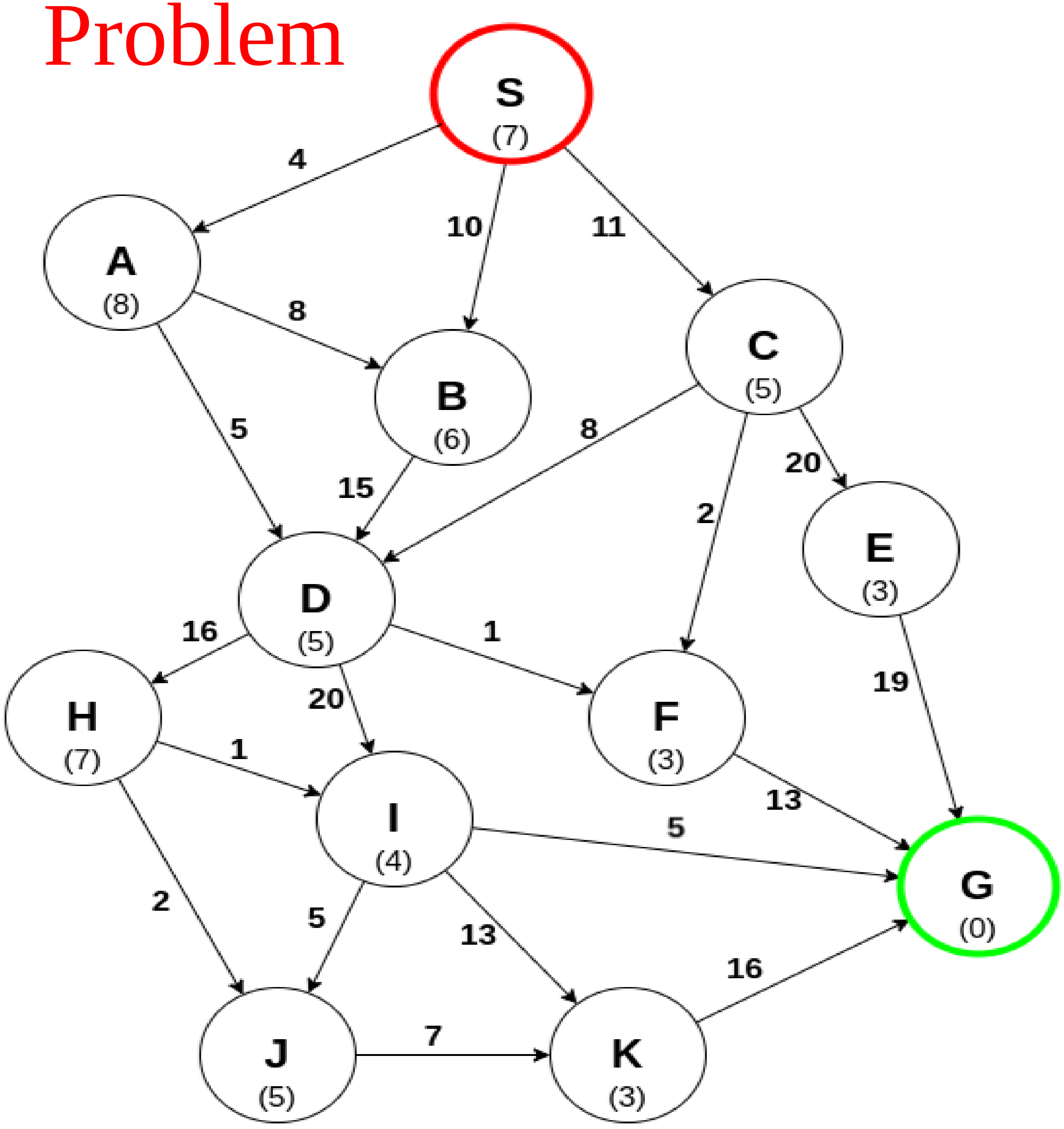


A* Search Example Contd…

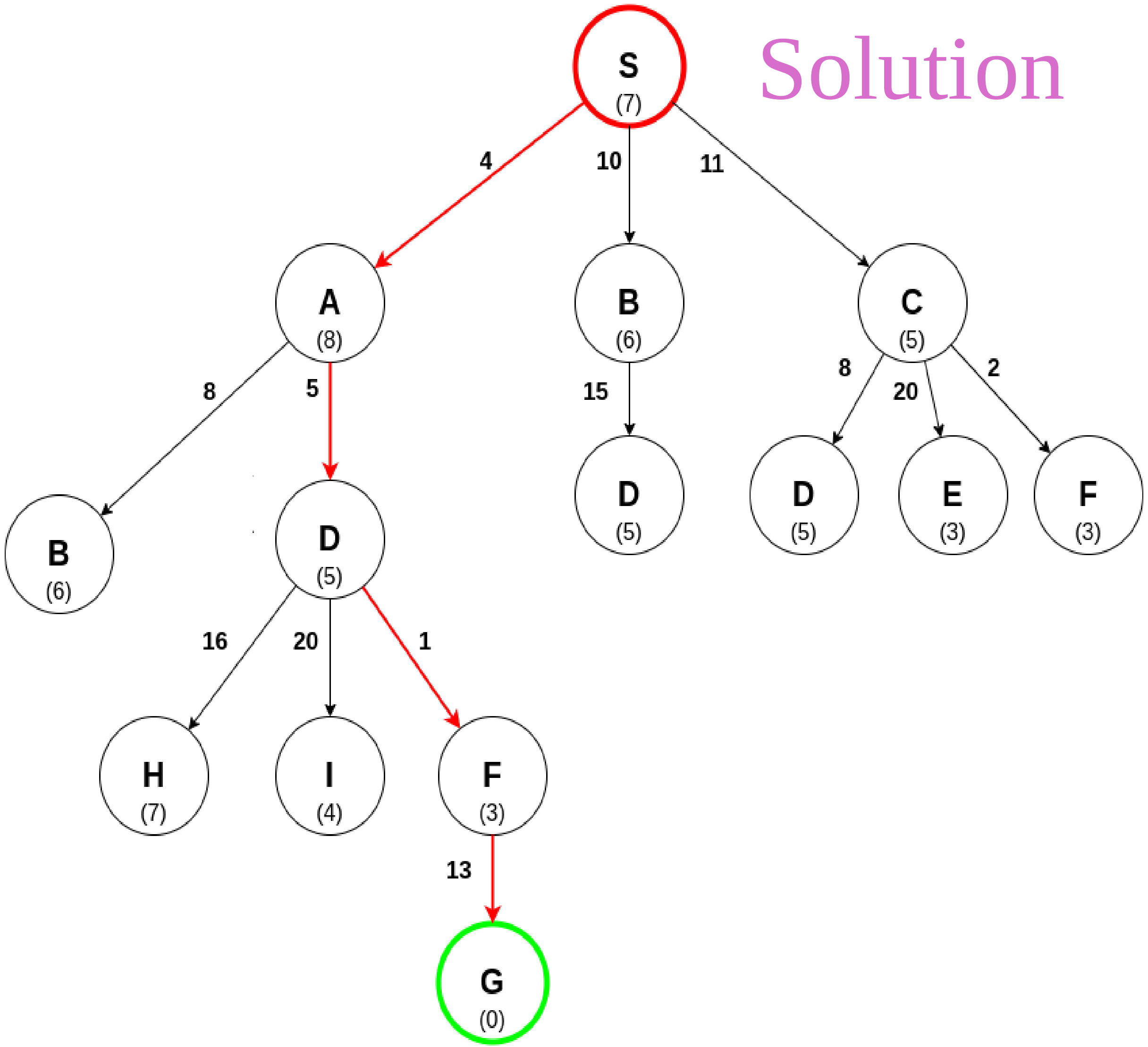


A* Search Example-2

Problem

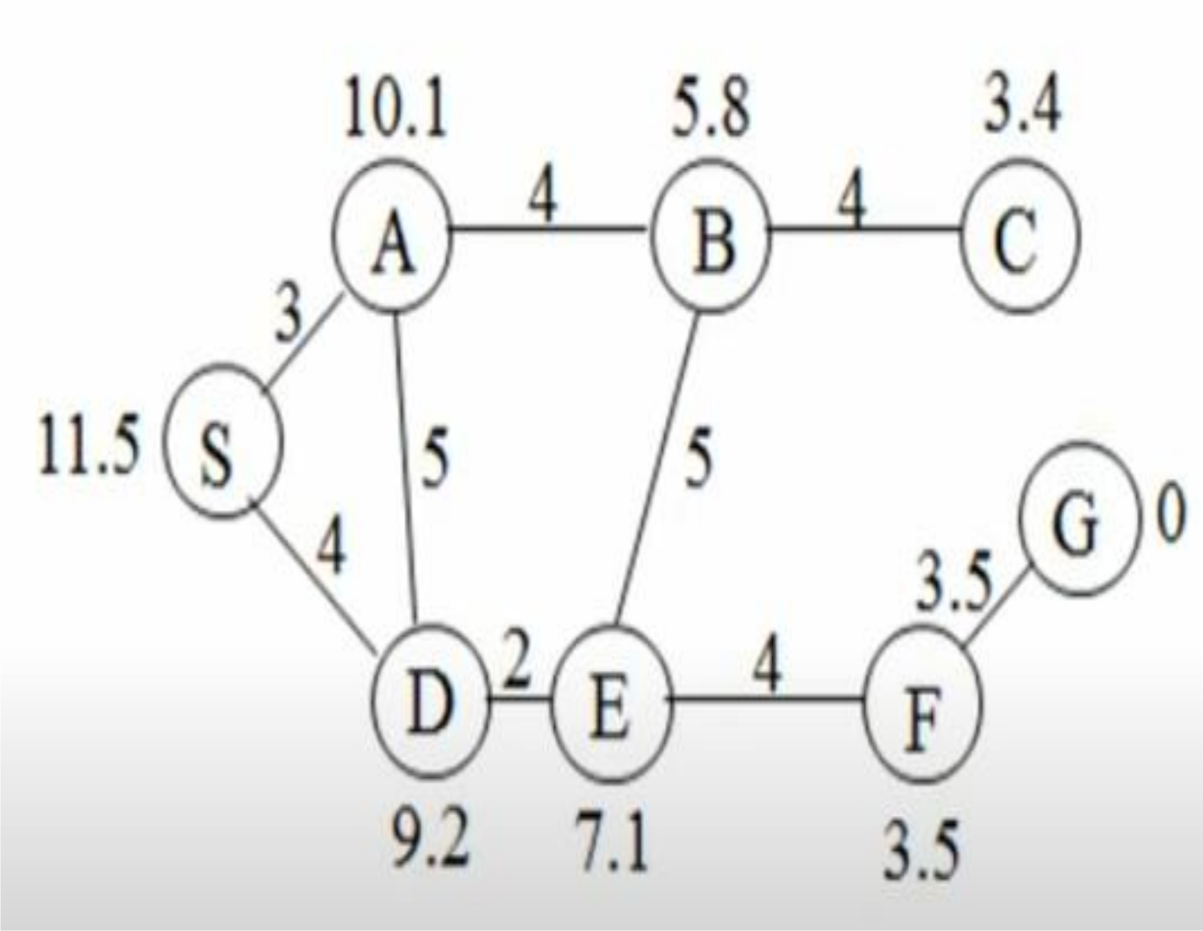


Solution

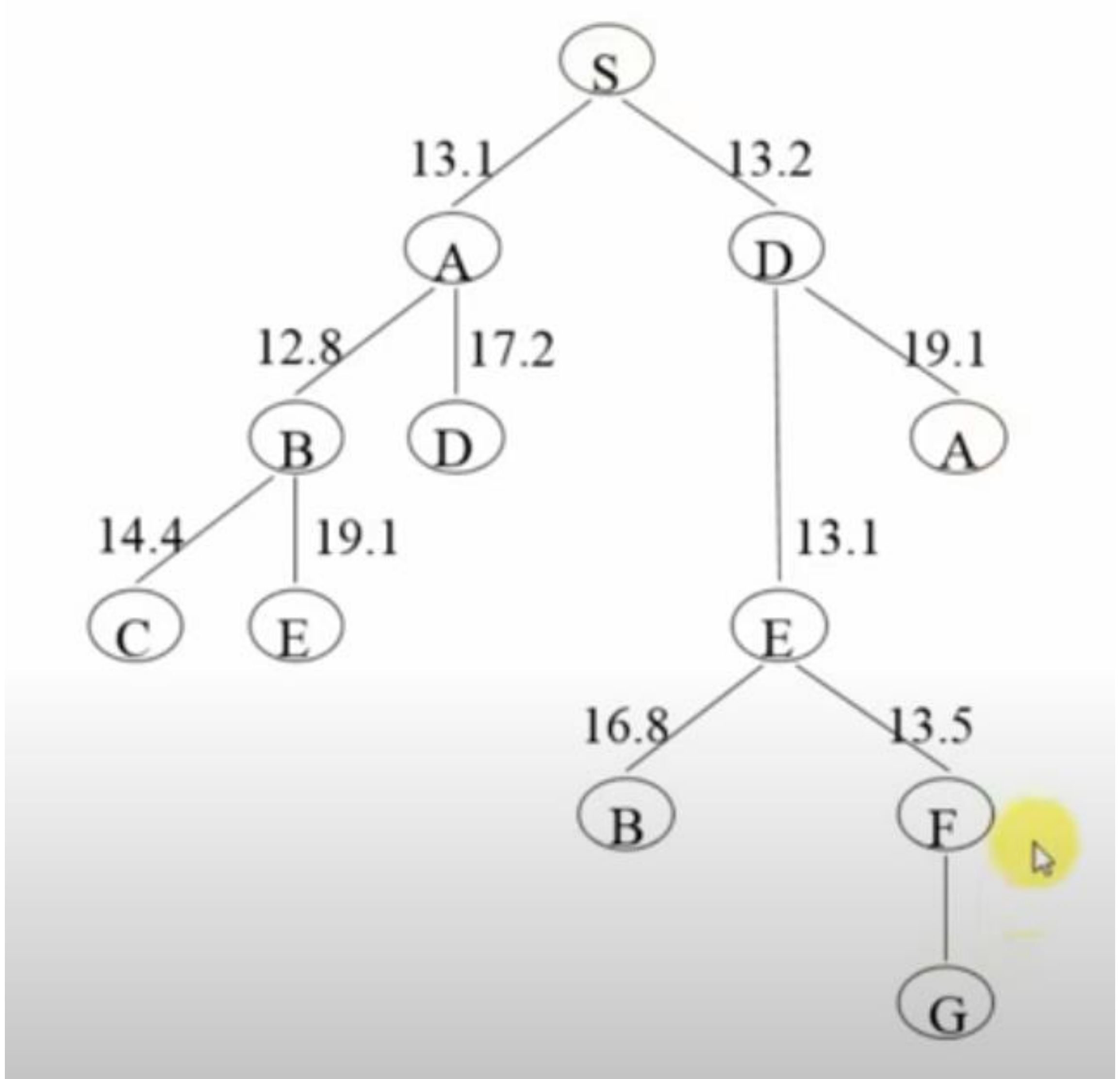


A* Search Example-3

Problem



Solution



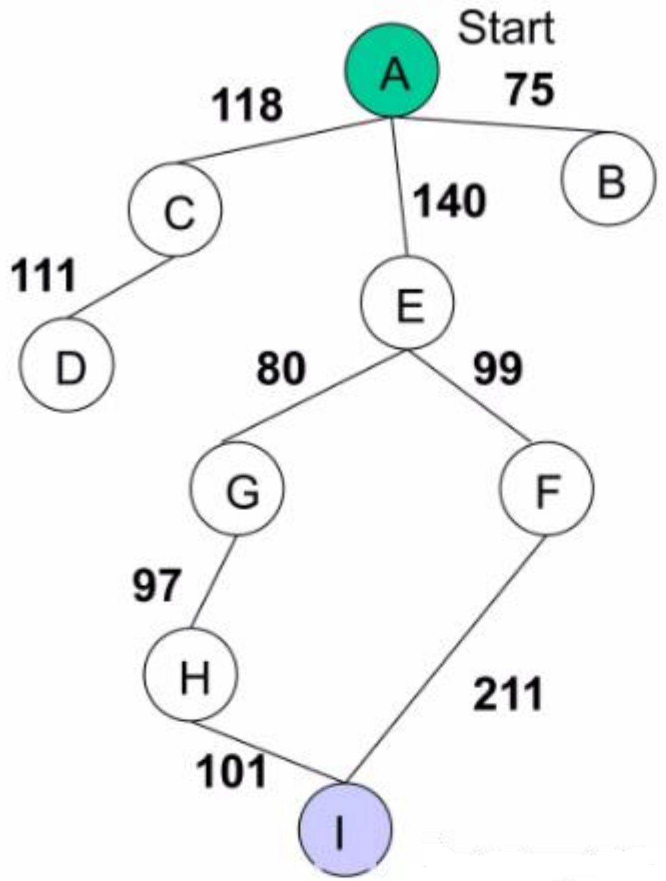
Conditions for optimality in A* search

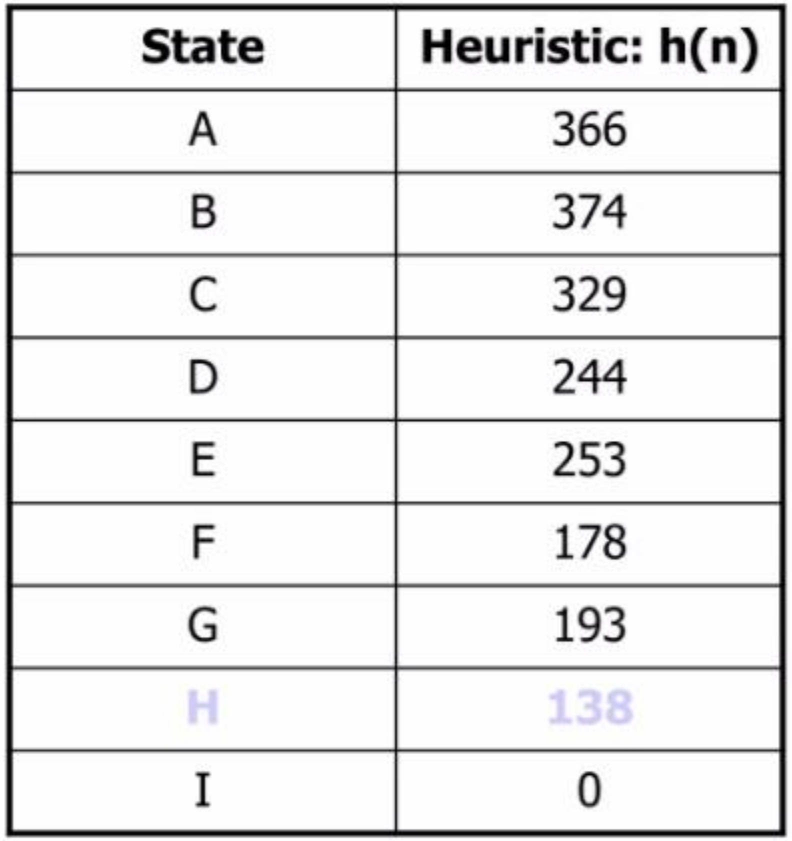
- A* search uses an **admissible** heuristic: A heuristic is admissible if it *never overestimates* the cost to reach the goal, as g(n) is the actual cost to reach n.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.
- It can be formally defined as follows:

  1. $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n
  2. $h(n) >= 0$ so $h(G)=0$ for any goal G.

- Example: . $h_{SLD}(n)$ being a straight line cannot be an overestimate.
- A heuristic h(n) is **consistent** if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n' :
  - $h(n) \leq c(n, a, n') + h(n')$

Example: Not admissible





State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	138
I	0

Optimality of A*

A* has the following properties:

Case 1: The tree-search version of A* is optimal if *h(n)* is admissible,

Case 2: The graph-search version is optimal if *h(n)* is consistent.

Admissible:

- A heuristic *h(n)* is admissible if for every node *n*,

  $h(n) \leq h^{*}(n)$, where $h^{*}(n)$ is the true cost to reach the goal state from *n*

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

Consistent:

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

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

- If *h* is consistent, we have

  $$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$$

  i.e., *f(n)* is non-decreasing along any path

A* search Evaluation

Completeness: YES

Time complexity:

Exponential: Number of nodes expanded is still exponential in the length of the solution.

(exponential with path length)

It is $O(b^{\Delta})$ where $\Delta \equiv h^* - h,$ (absolute error)

Space complexity:

More Space: It keeps all generated nodes in memory

Optimality: YES

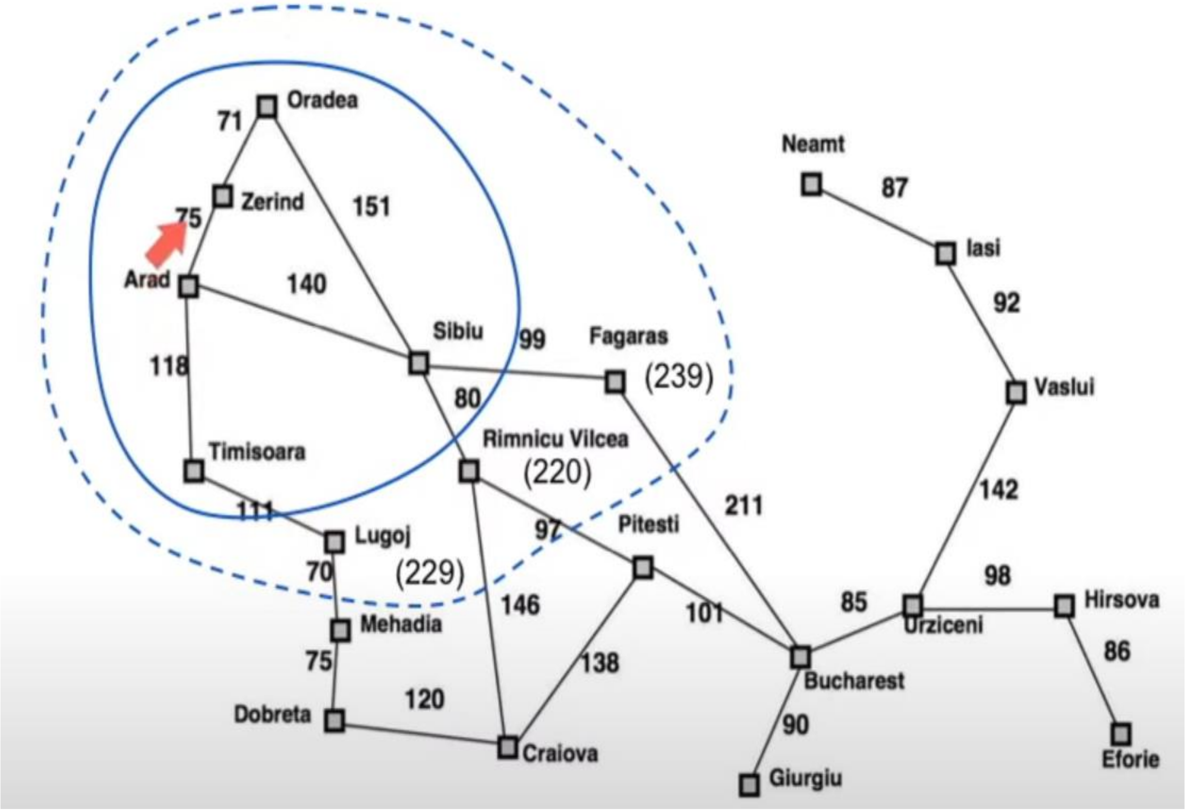
If C∗ is the cost of the optimal solution path, then we can say the following:

A* expands all nodes with *f(n)*< *C**

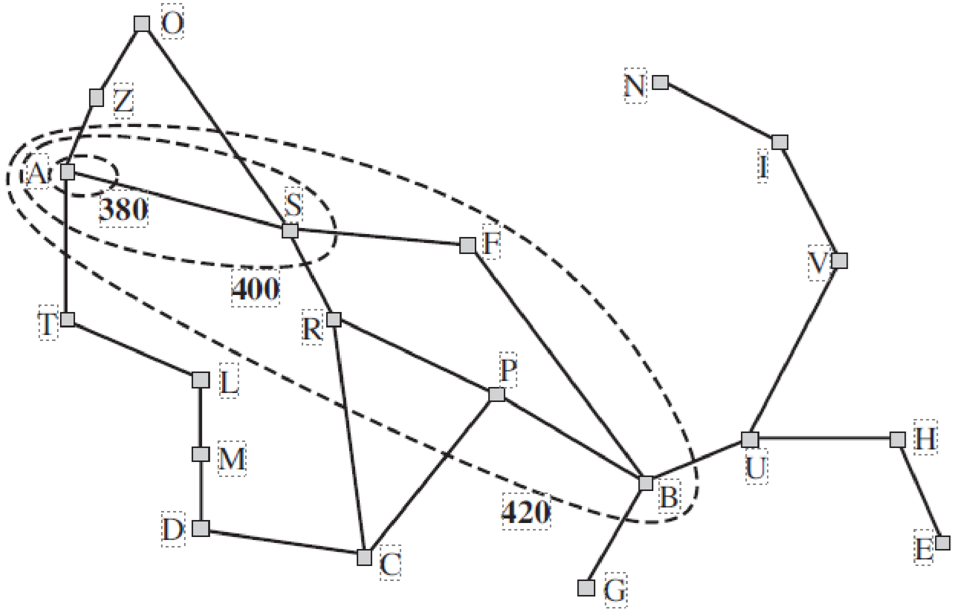
A* expands some nodes with *f(n)*=*C**

A* expands no nodes with *f(n)*>*C**

Uniform Cost Search vs A* Search



USC: the bands will be “circular” around the start state



A*: the bands will stretch toward the goal state

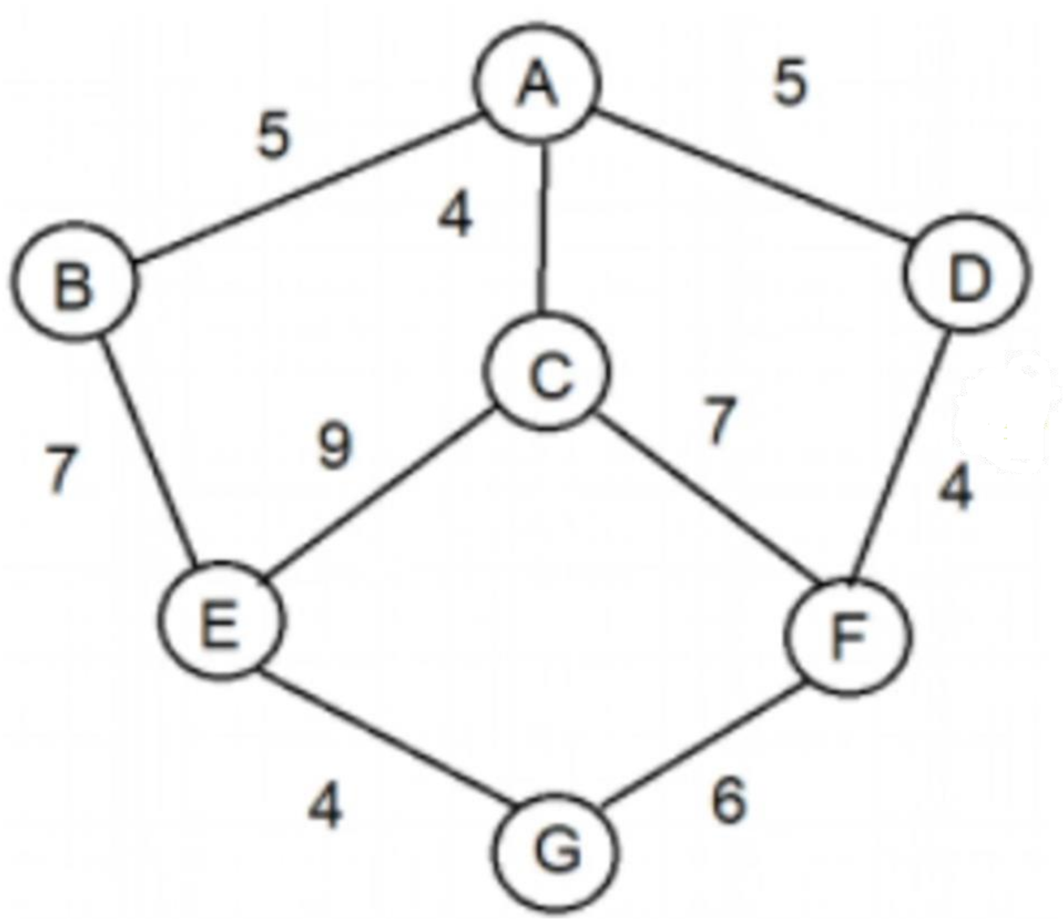
Memory-bounded heuristic search

- Some solutions to A* space problems (maintain completeness and optimality)
    - Iterative-deepening A* (IDA*)
        - Here cutoff information is the *f*-cost (*g*+*h*) instead of depth
    - Recursive Best-First Search(RBFS)
        - Recursive algorithm that attempts to mimic standard best-first search with linear space.
    - (Simple) Memory-bounded A* ((S)MA*)
        - Drop the worst-leaf node when memory is full

Recursive best-first search

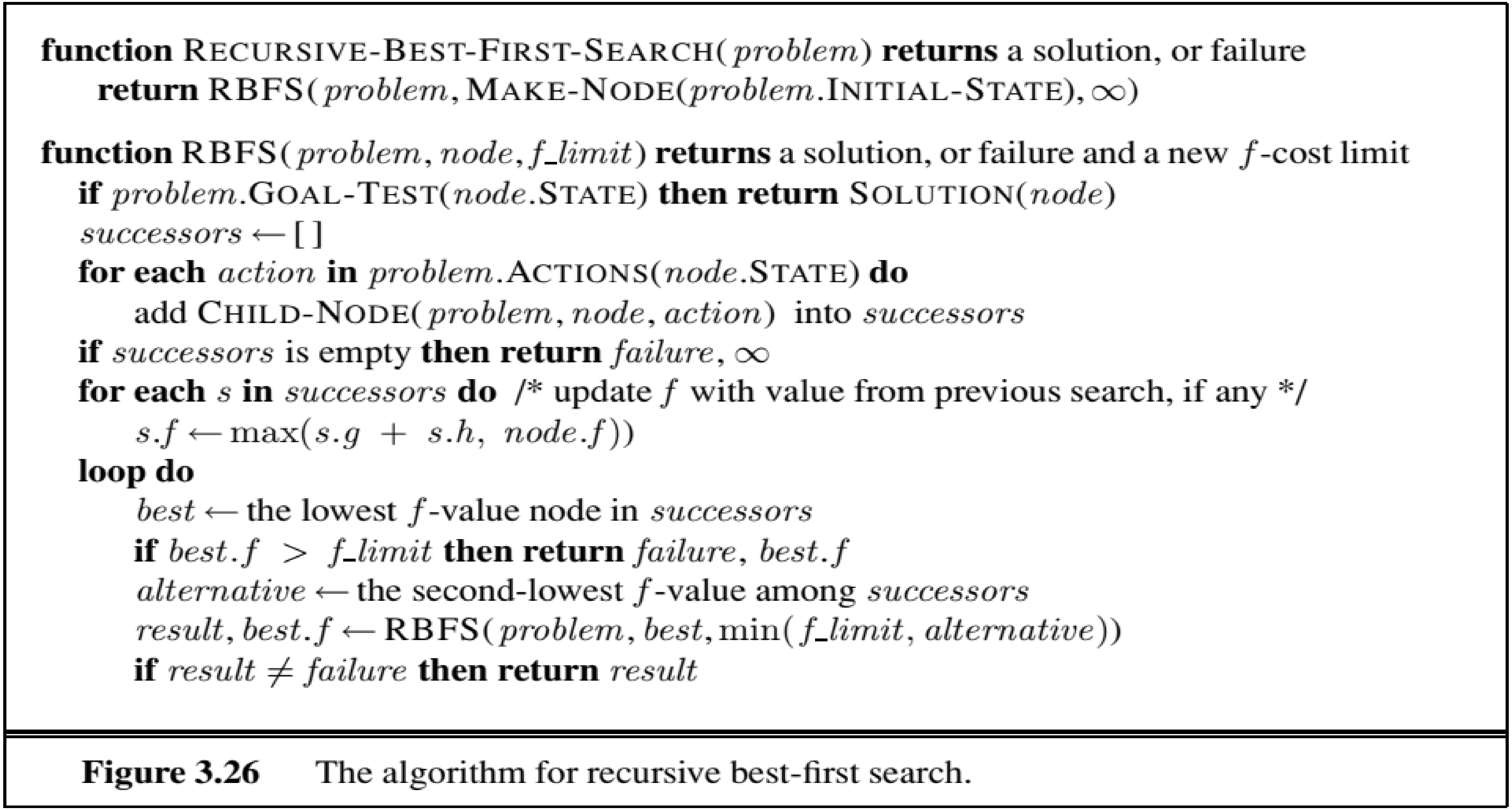
- Keeps track of the f-value of the best-alternative path available.
  - If current f-values exceeds this alternative f-value then backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
  - Re-expansion of this result is thus still possible.

Recursive best-first search: Example



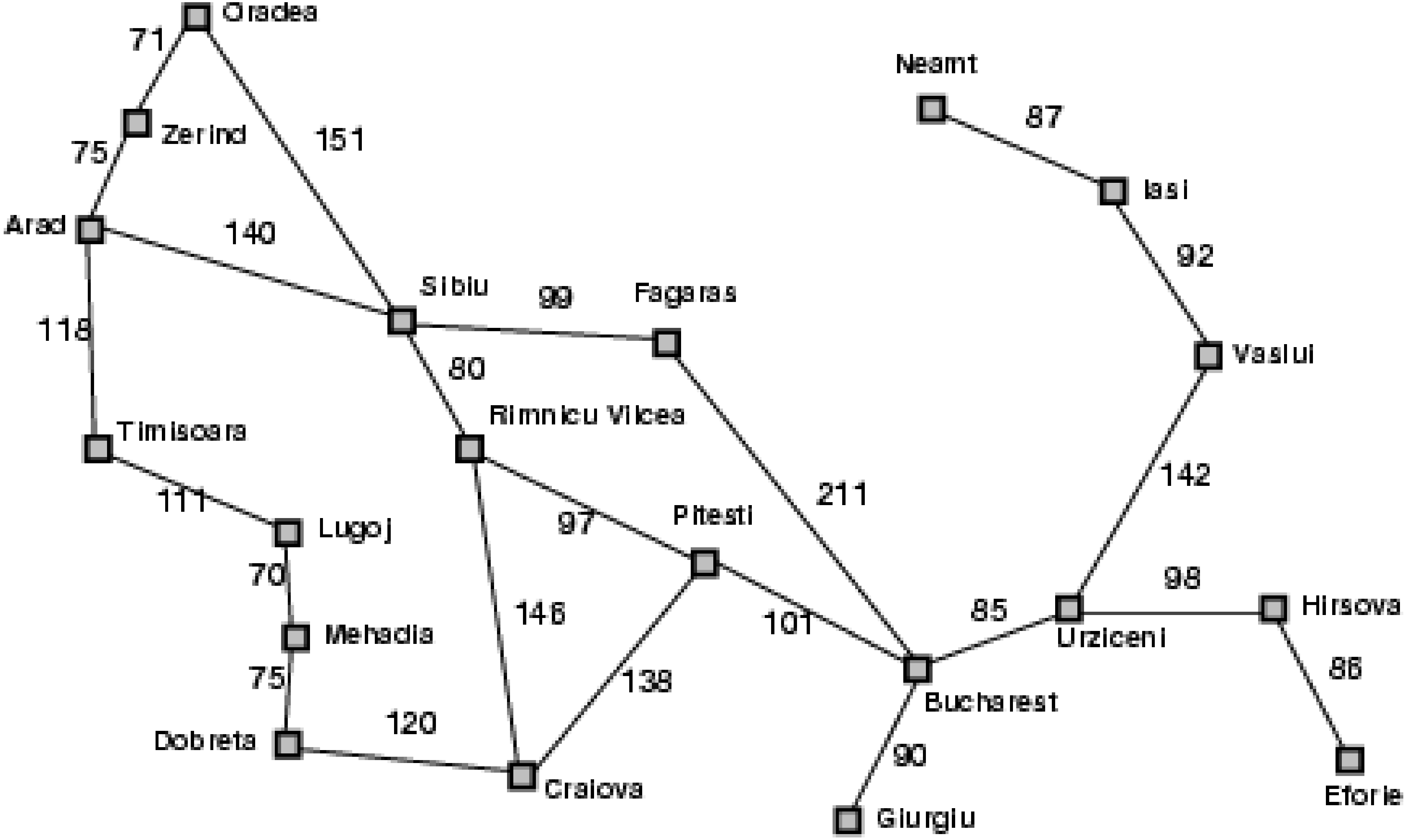
A	12
B	12
C	10
D	7
E	2
F	6
G	0

Recursive best-first search Pseudocode



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

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 $\leftarrow$ [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action)  into successors
  if successors is empty then return failure, $\infty$
  for each s in successors do  /* update $f$ with value from previous search, if any */
      $s.f \leftarrow \max(s.g + s.h,\ node.f)$
  loop do
      best $\leftarrow$ the lowest $f$-value node in successors
      if $best.f > f\_limit$ then return failure, best.f
      alternative $\leftarrow$ the second-lowest $f$-value among successors
      result, best.f $\leftarrow$ RBFS(problem, best, min(f_limit, alternative))
      if result $\neq$ failure then return result
```

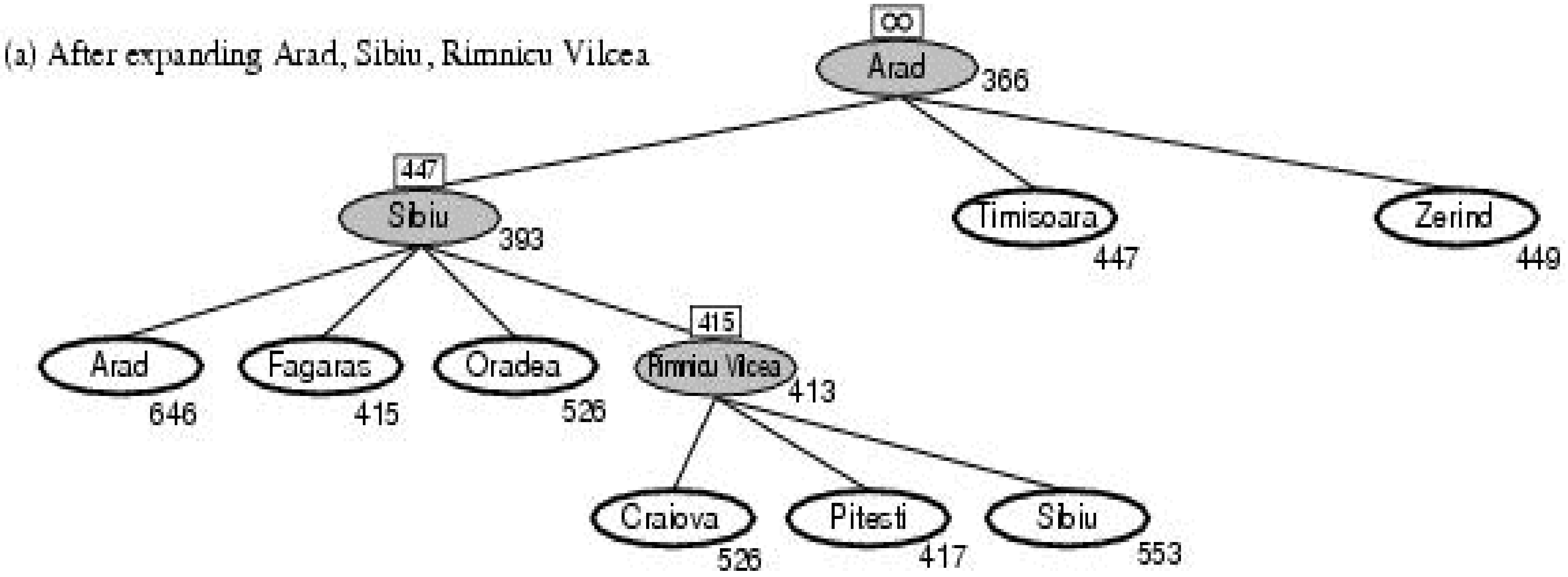
Figure 3.26 The algorithm for recursive best-first search.



Straight–line distance
to Bucharest

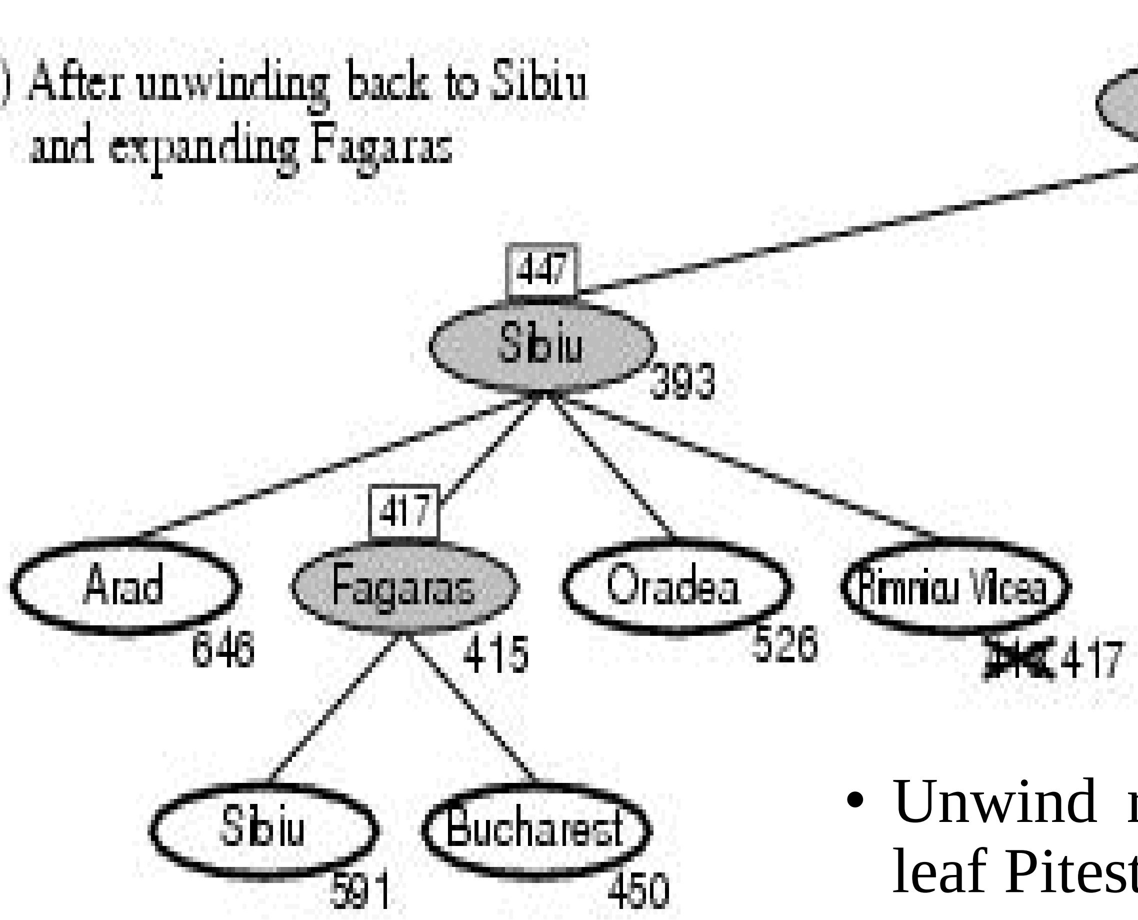
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recursive best-first search, example.



- Path until Rumnicu Vilcea is already expanded
- Above node; *f*-limit for every recursive call is shown on top.
- Below node: *f(n)*
- The path is followed until Pitesti which has a *f*-value worse than the *f-limit*.

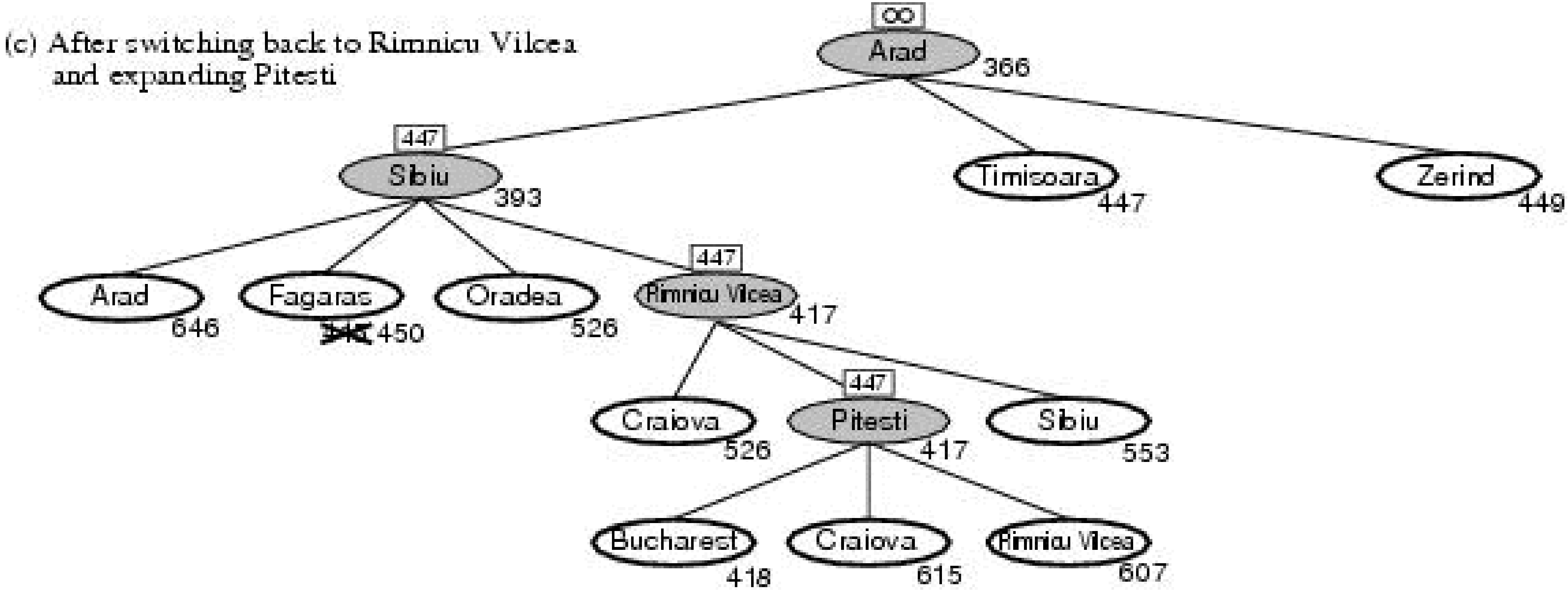
(b) After unwinding back to Sibiu and expanding Fagaras



- Unwind recursion and store best *f*-value for current best leaf Pitesti

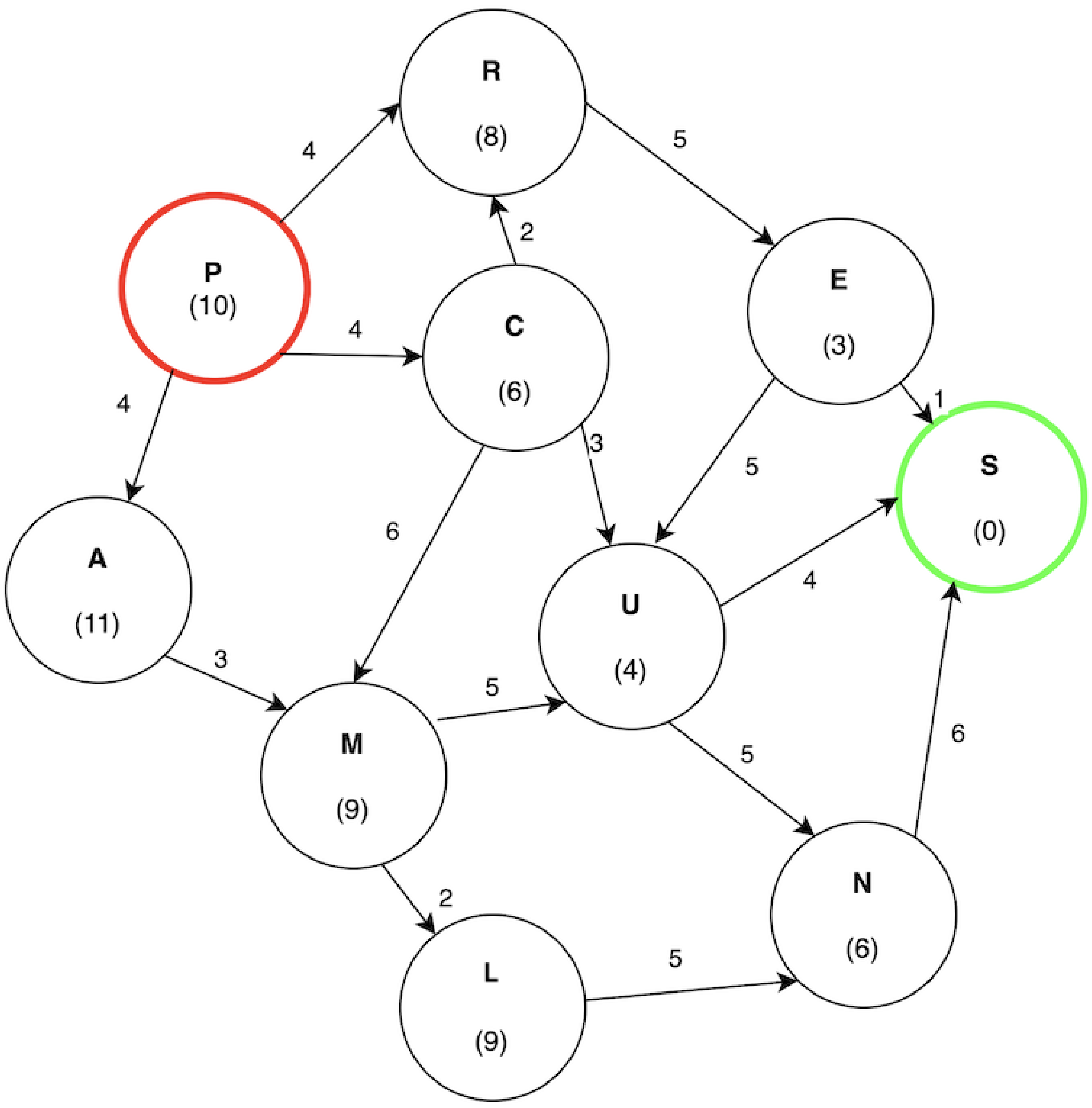
  result, *f* [*best*] ← RBFS(*problem*, *best*, min(*f_limit*, *alternative*))

- *best* is now Fagaras. Call RBFS for new *best*
  - *best* value is now 450

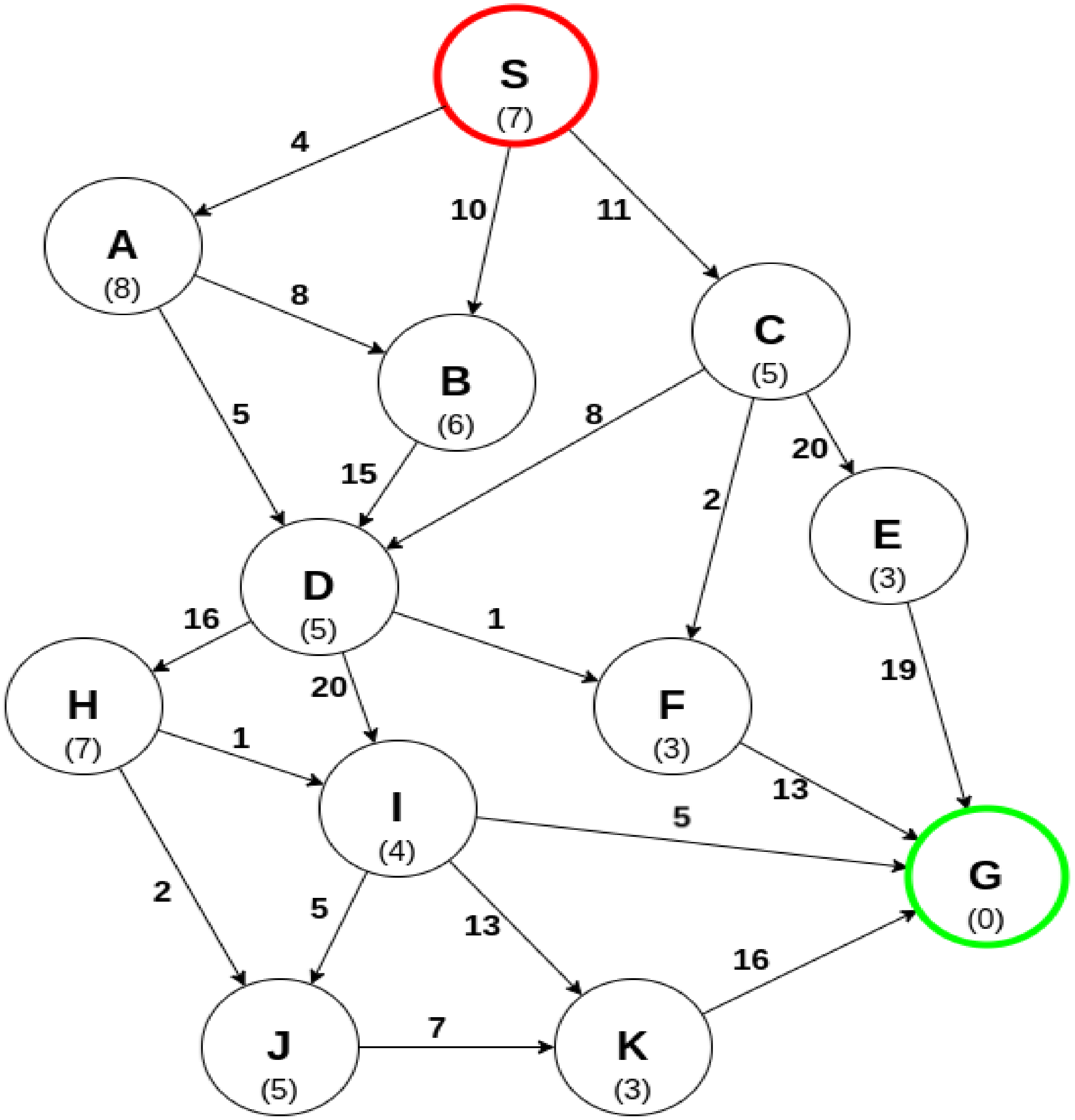


- Unwind recursion and store best f-value for current best leaf Fagaras
  result, f [best] ← RBFS(problem, best, min(f_limit, alternative))
- best is now Rimnicu Viclea (again). Call RBFS for new best
  - Subtree is again expanded.
  - Best alternative subtree is now through Timisoara.
- Solution is found since because 447 > 417.

Example-3



Example-4



Recursive Best First Search evaluation

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, optimal if *h(n)* is admissible
- Space complexity is *O(bd)*.
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
- IDA* and RBFS suffer from ***too little*** memory.

(Simplified) Memory-bounded A*

- Use all available memory.
  - i.e. expand best leaf nodes until available memory is full
  - When full, SMA* drops worst leaf node (highest *f*-value)
  - Like RFBS backup forgotten node to its parent

- What if all leaves have the same *f*-value?
  - Same node could be selected for expansion and deletion.
  - SMA* solves this by expanding *newest* best leaf and deleting *oldest* worst leaf.
- SMA* is complete if solution is reachable, optimal if optimal solution is reachable.

Learning to search better

- All previous algorithms use *fixed strategies*.
- Agents can learn to improve their search by exploiting the *meta-level state space*.
  - Each meta-level state is a internal (computational) state of a program that is searching in *the object-level state space*.
  - In A* such a state consists of the current search tree
- A meta-level learning algorithm from experiences at the meta-level.