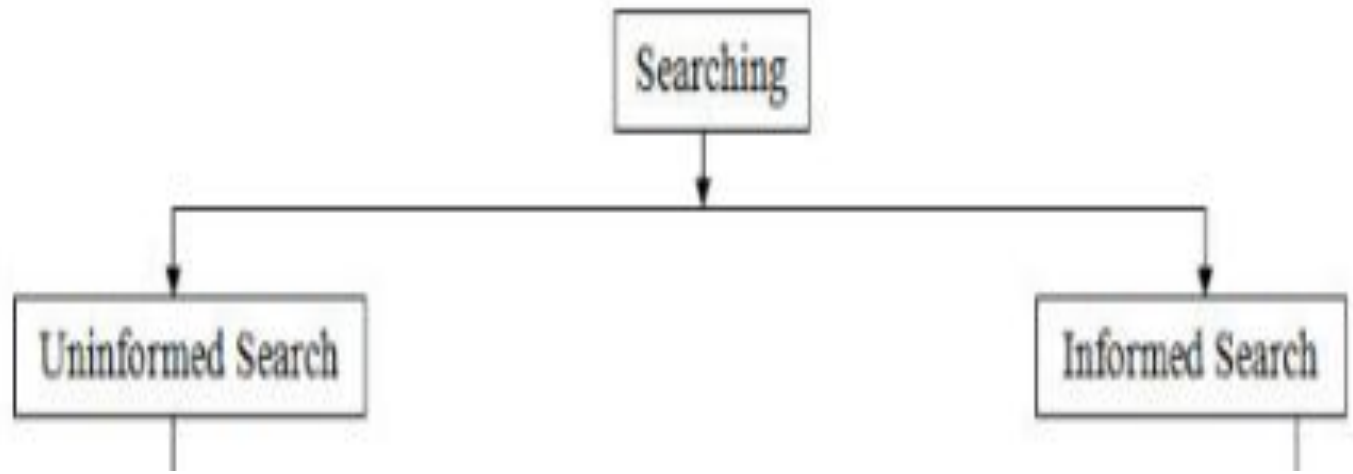
Module 2

PART 3

DIFFERENT TYPES OF SEARCHING



UNINFORMED SEARCH STRATEGIES

- **Uninformed search -** also called **blind search**.
- The term means that:
- the strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the order in which nodes are expanded.
- Strategies that know whether one non-goal state is “more promising” than another are called informed search or heuristic search strategies

Types of Blind Searches

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-first 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 very simply by using a FIFO queue for the frontier. Thus, 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.

- 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.

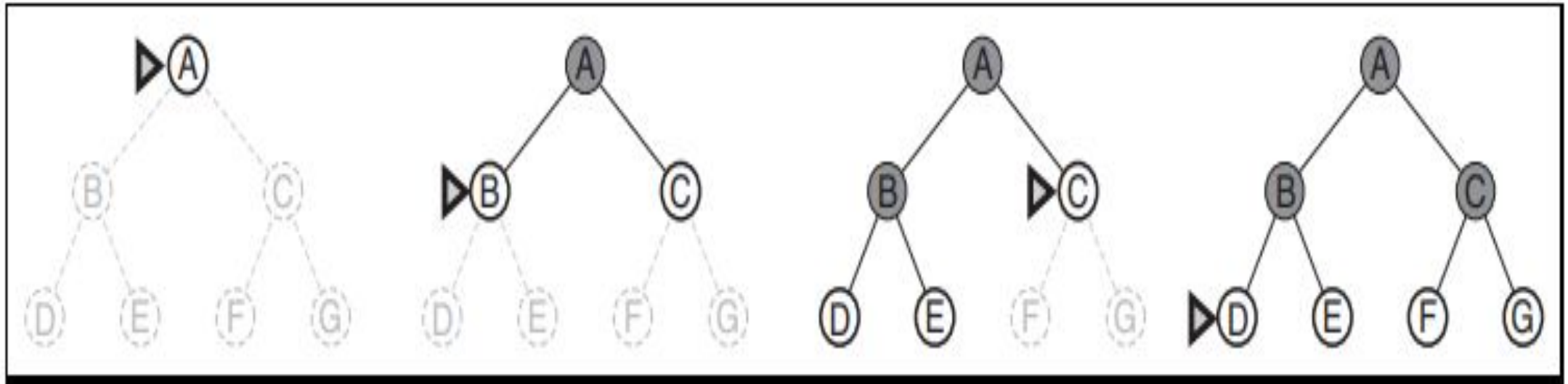


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

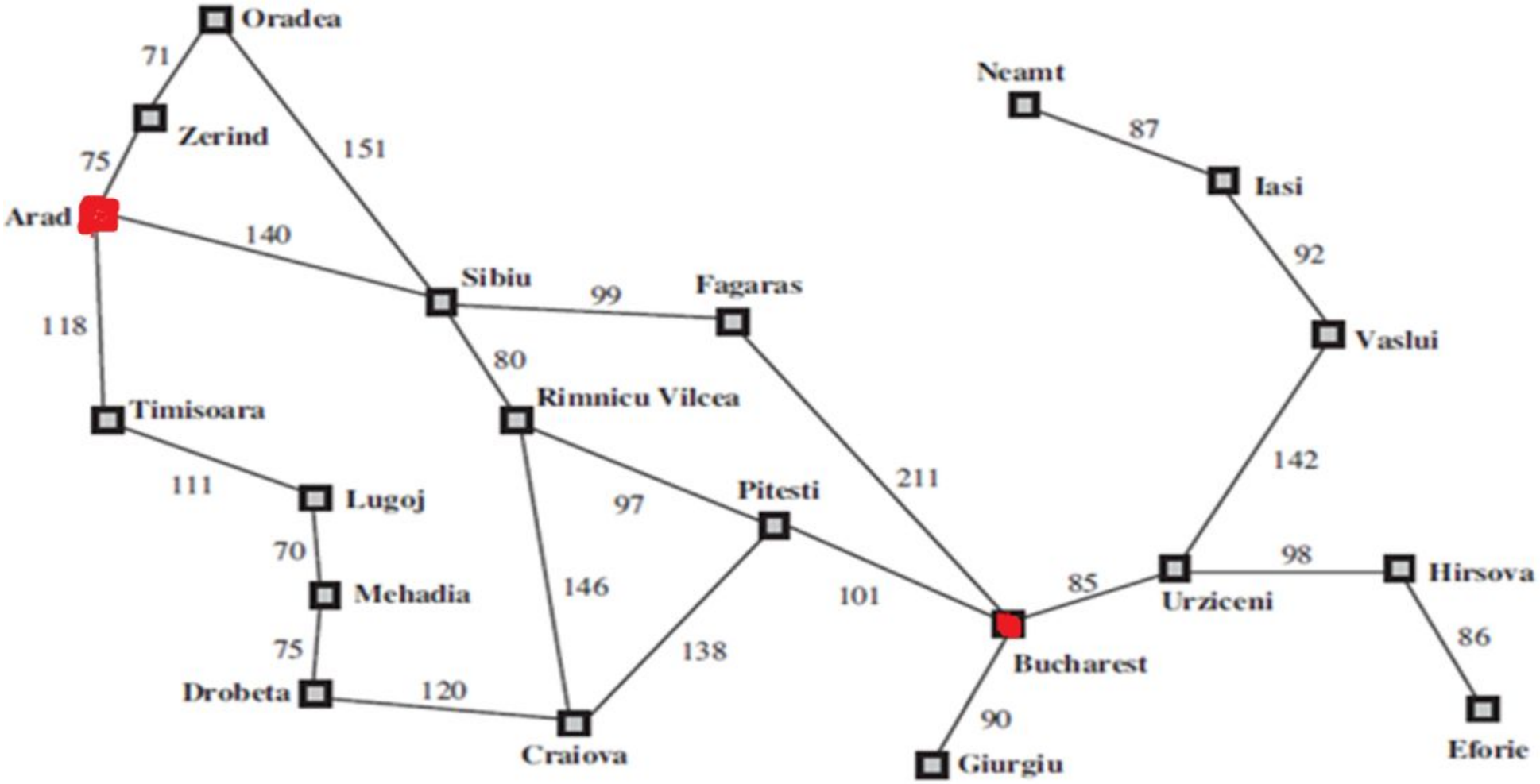
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop 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)
```

Figure 3.11 Breadth-first search on a graph.

Breadth-first search - Example

Initial State:
In(Arad)

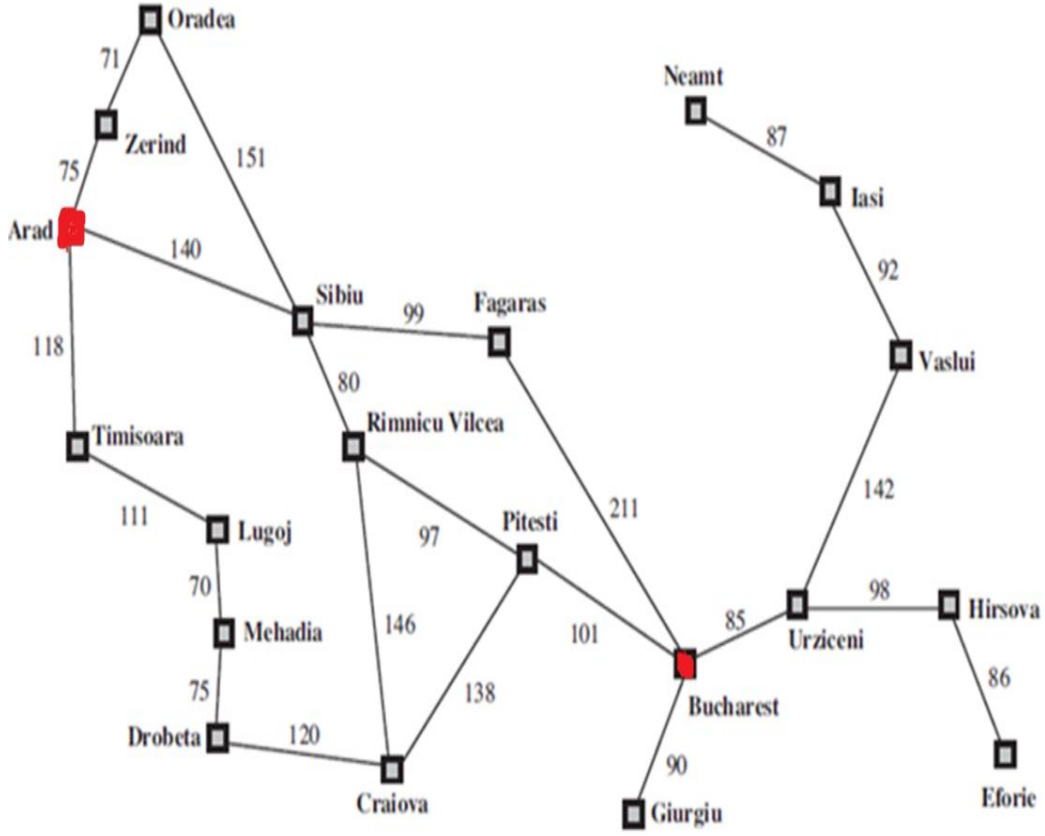
Goal State:
In(Bucharest)



Breadth-first search - Example

Valid Actions
Go(Zerind)
Go(Timisoara)
Go(Sibiu)

Arad

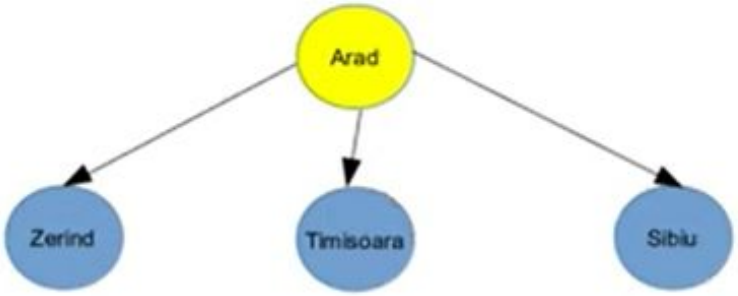


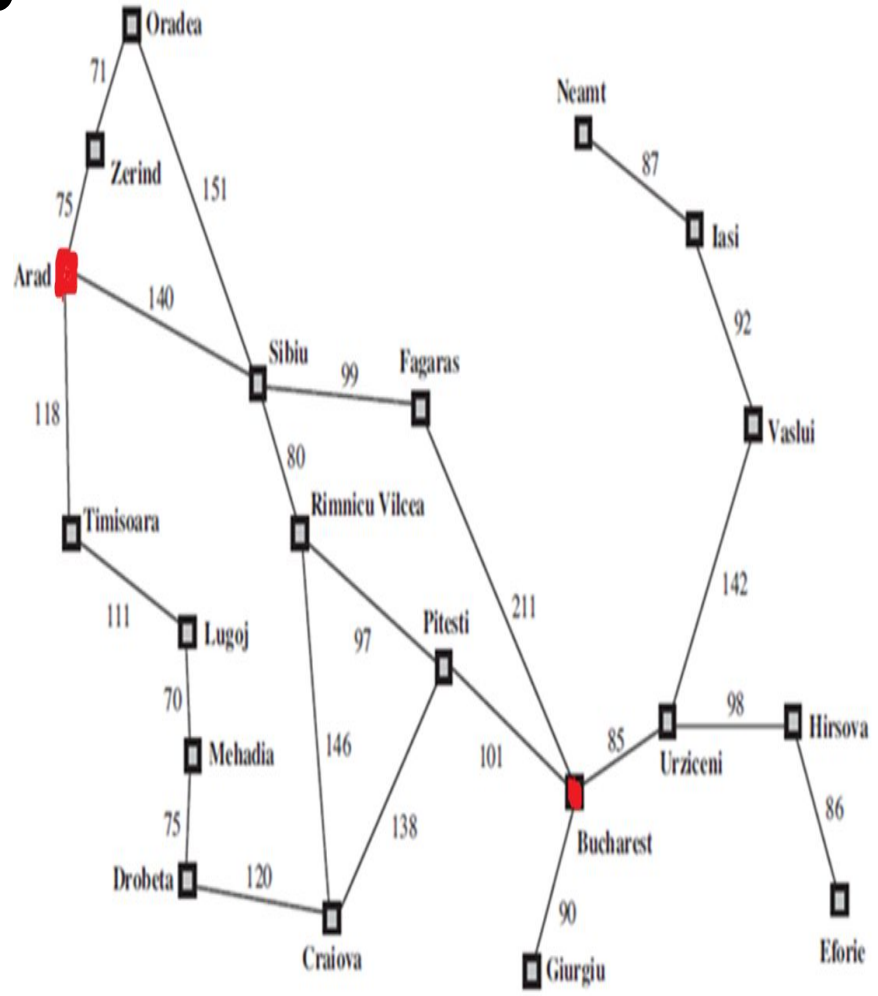
Frontier = []

Explored Set = [Arad]

Breadth-first search - Example

Valid Actions
Go(Zerind)
Go(Timisoara)
Go(Sibiu)



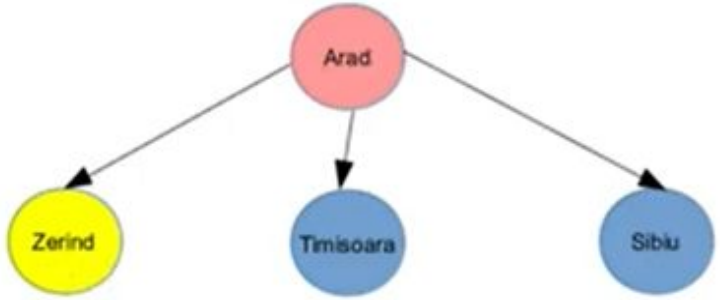


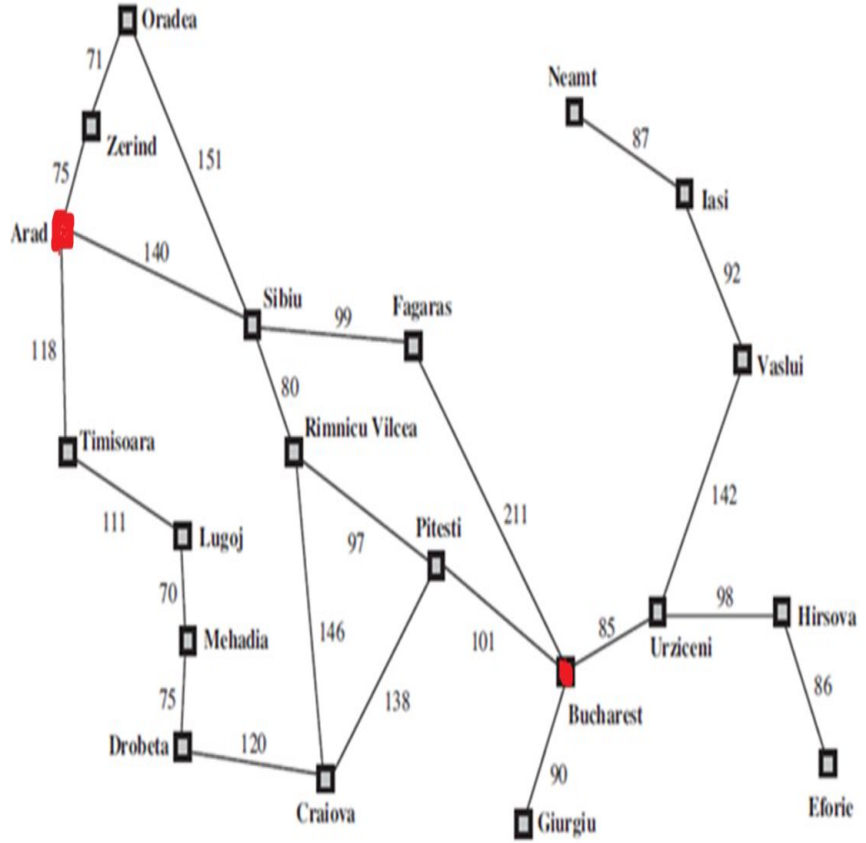
Frontier = [Zerind, Timisoara, Sibiu]

Explored Set = [Arad]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Oradea)



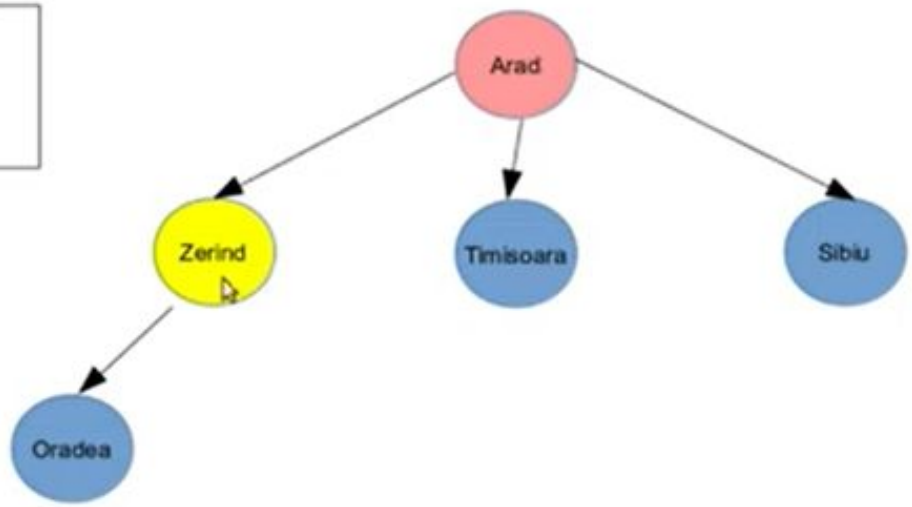


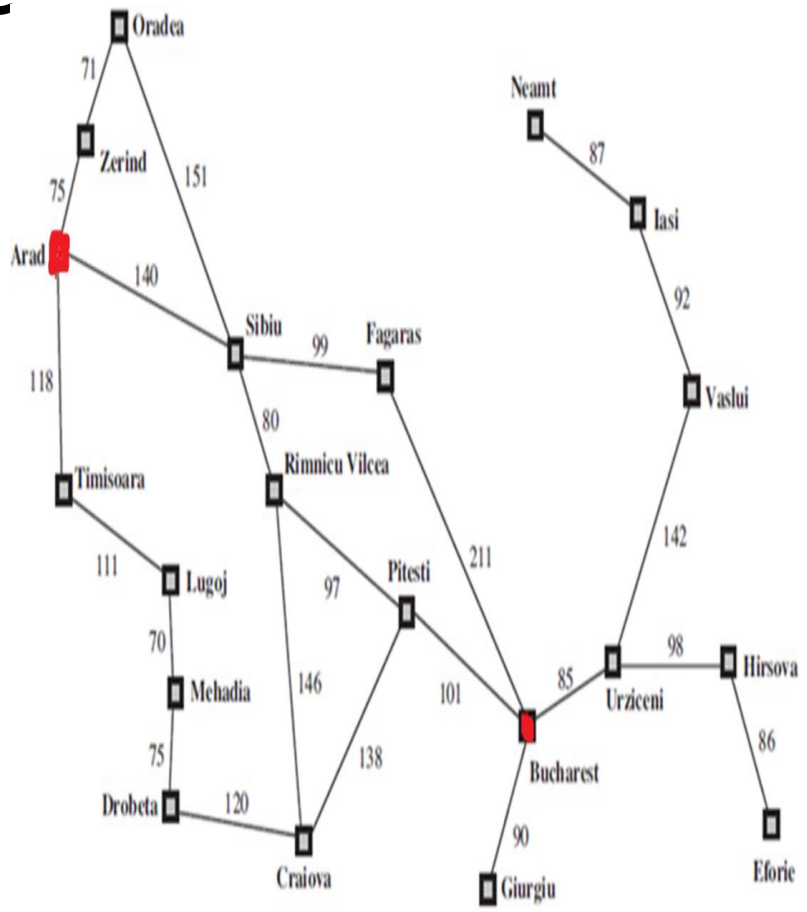
Frontier = [Timisoara, Sibiu]

Explored Set = [Arad, Zerind]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Oradea)



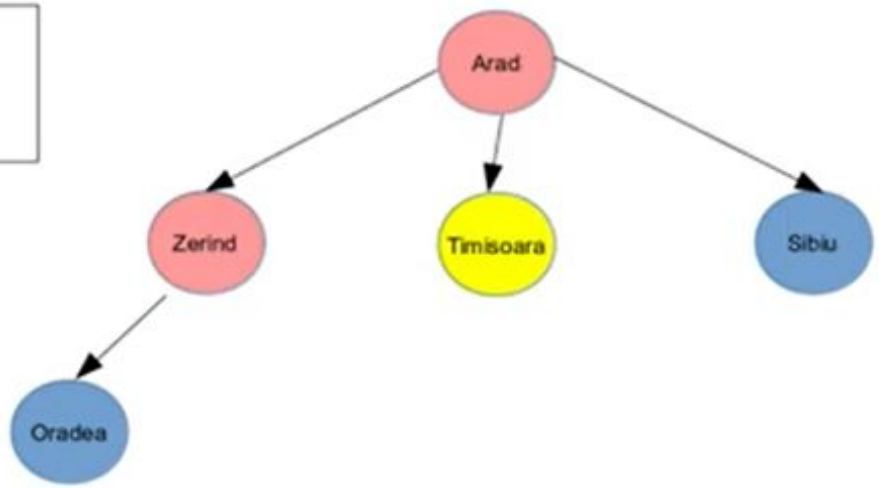


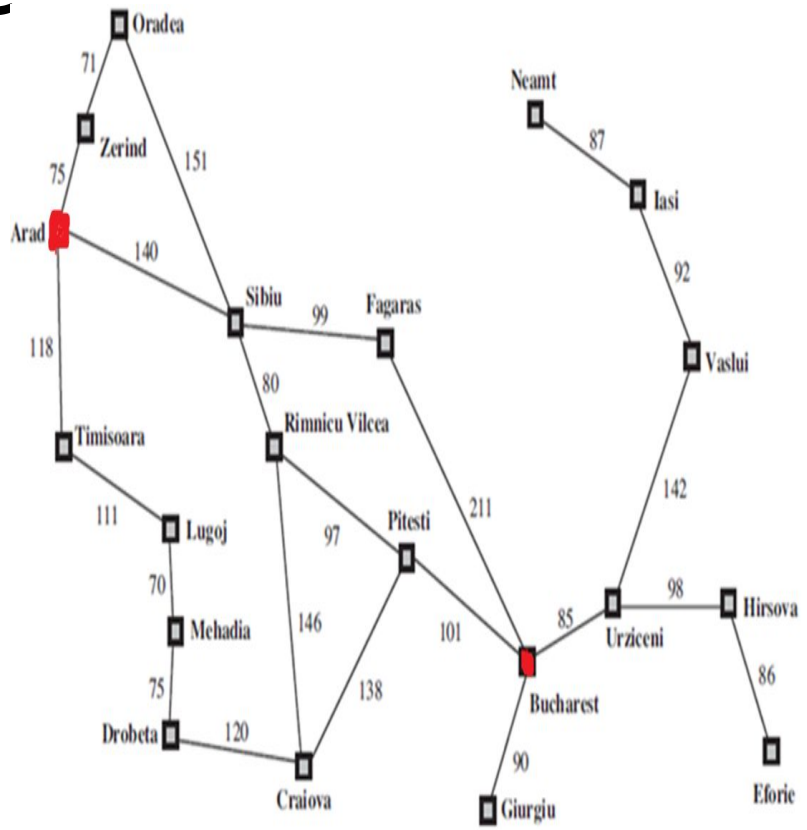
Frontier = [Timisoara, Sibiu, Oradea]

Explored Set = [Arad, Zerind]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Lugoj)



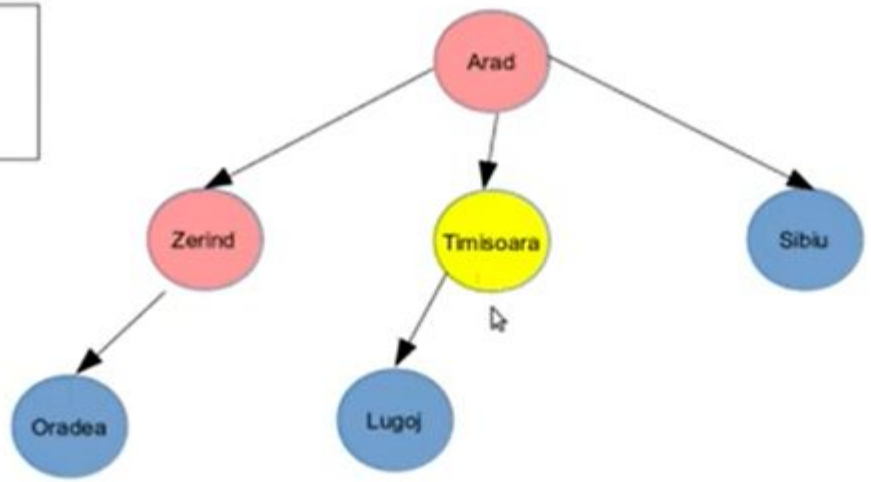


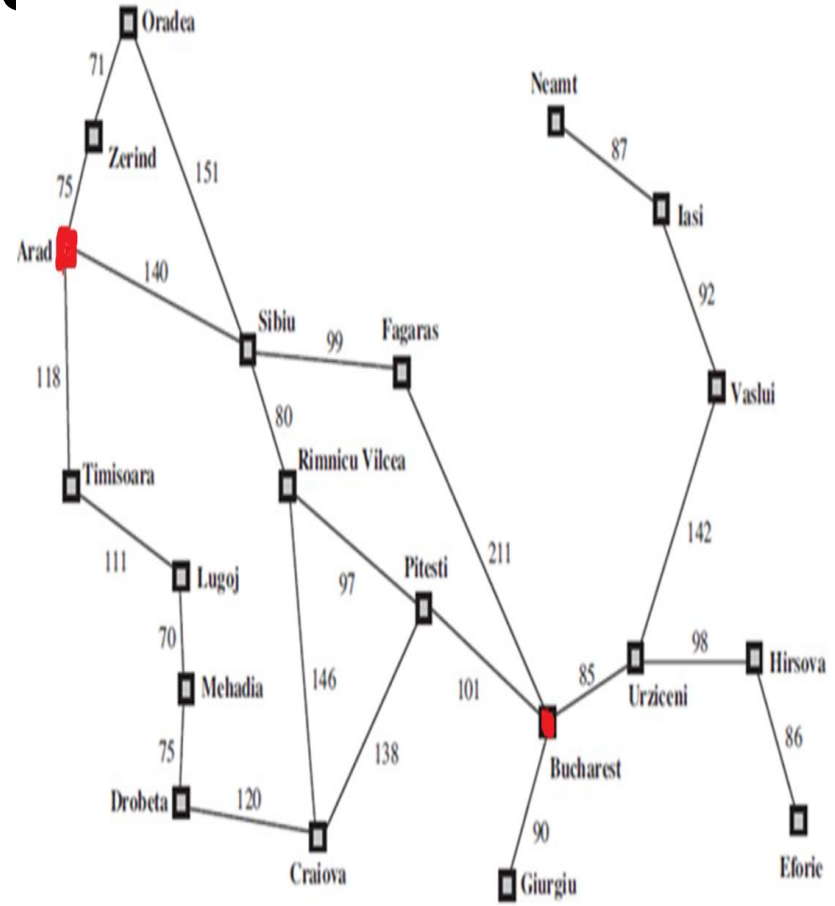
Frontier = [Sibiu, Oradea]

Explored Set = [Arad, Zerind, Timisoara]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Lugoj)



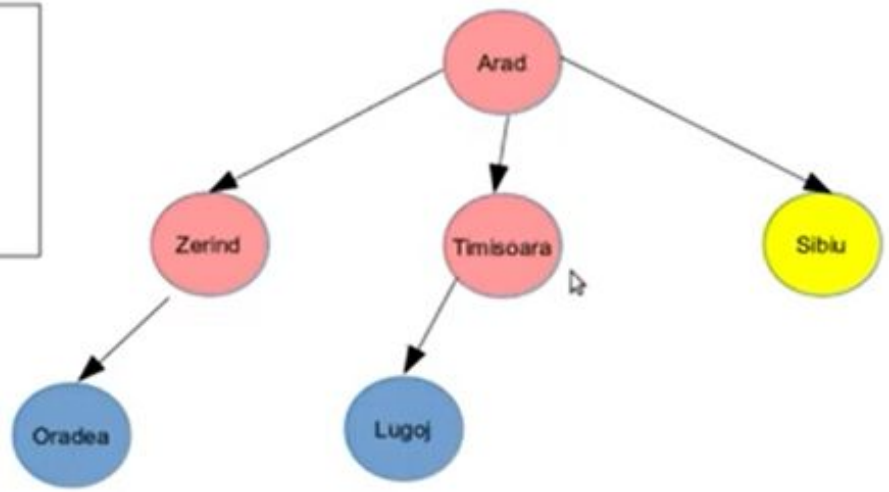


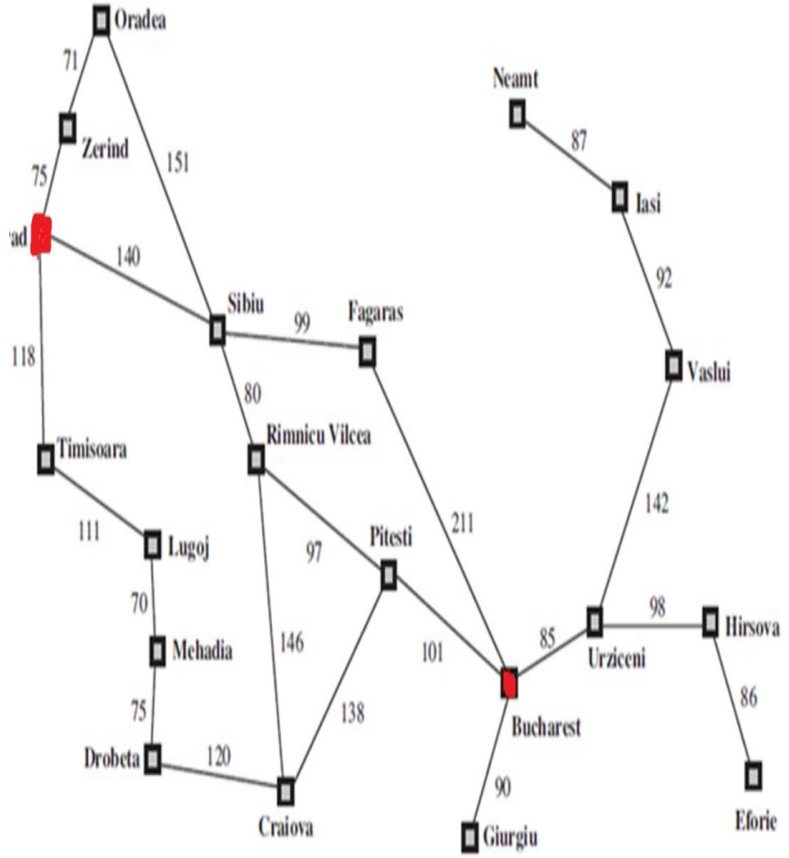
Frontier = [Sibiu, Oradea, Lugoj]

Explored Set = [Arad, Zerind, Timisoara]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Oradea)
Go(Rimnicu)
Go(Fagaras)



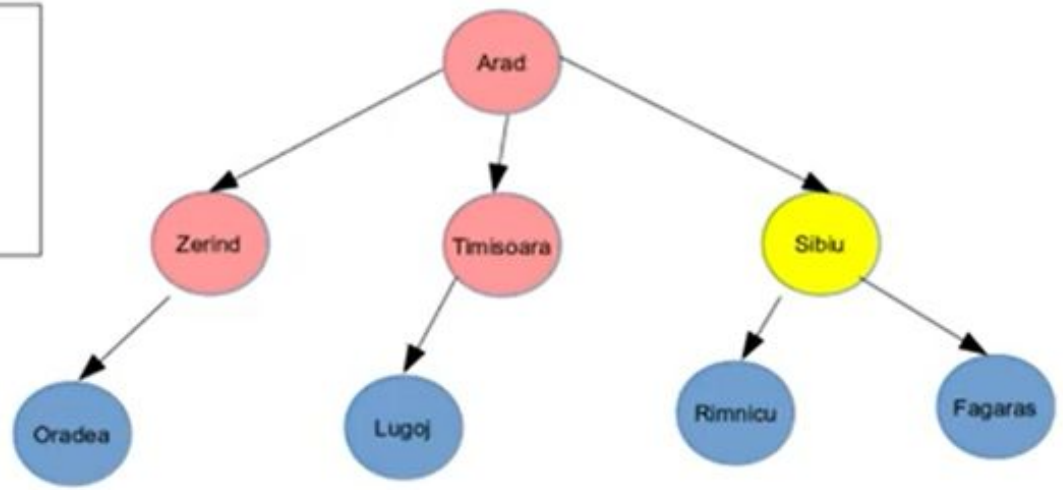


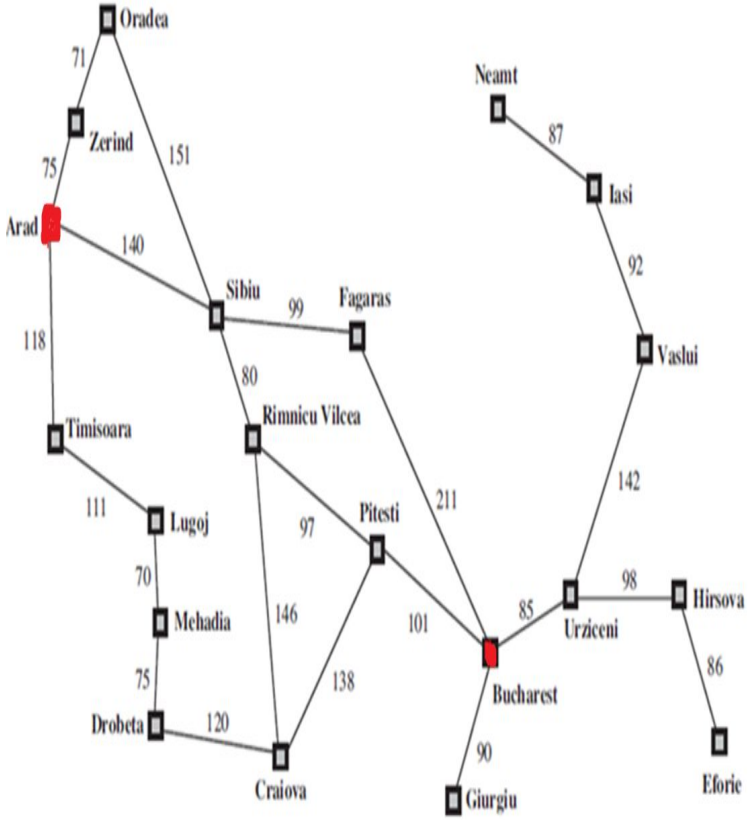
Frontier = [Oradea, Lugoj]

Explored Set = [Arad, Zerind, Timisoara, Sibiu]

Breadth-first search - Example

Valid Actions
Go(Arad)
Go(Oradea)
Go(Rimnicu)
Go(Fagaras)



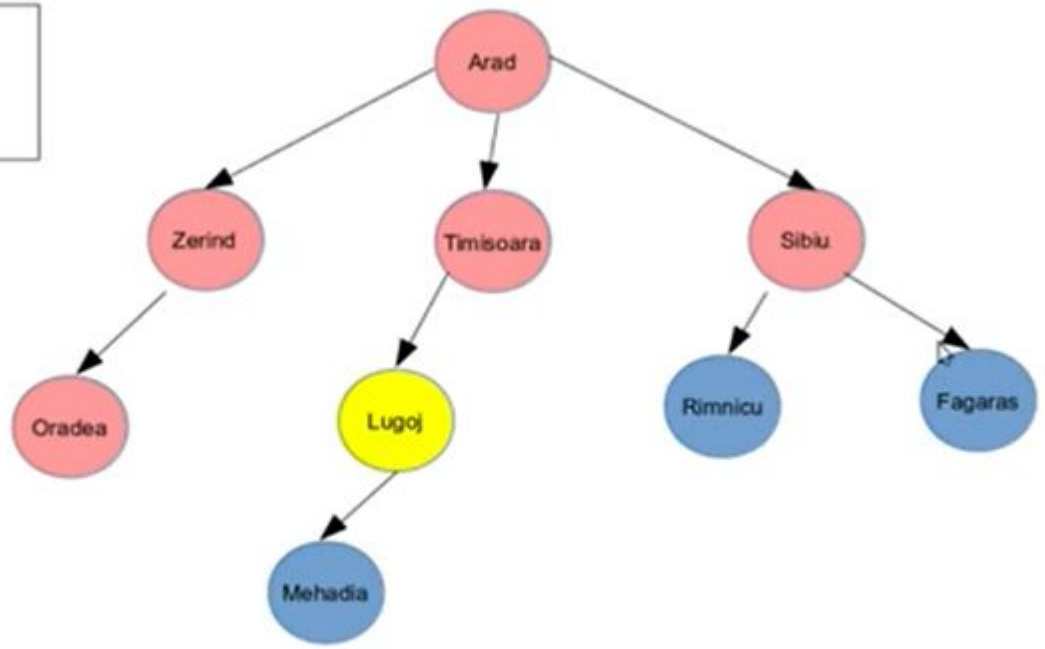


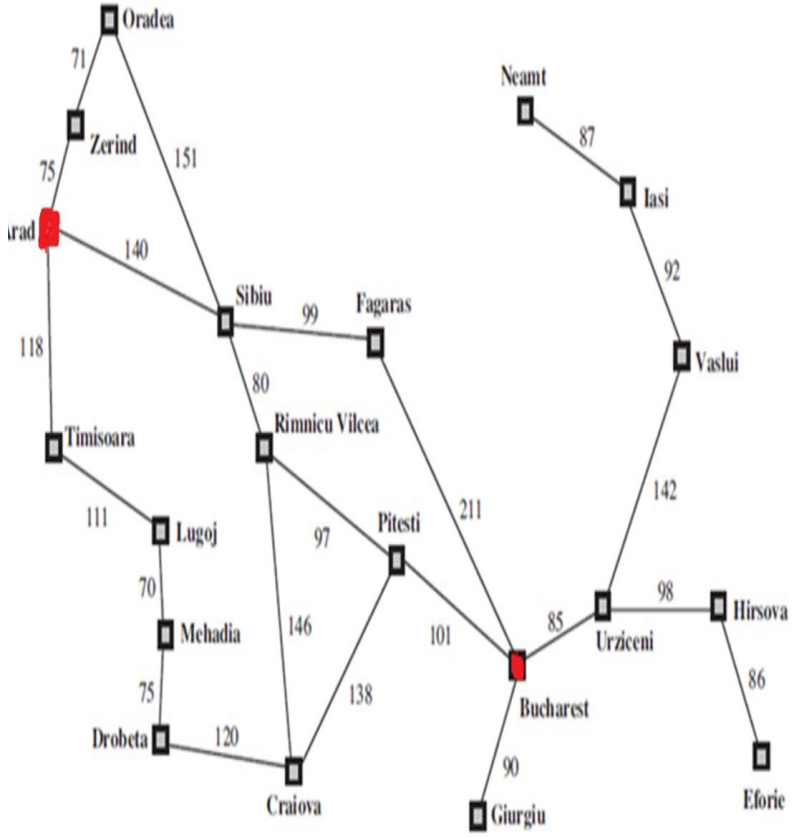
Frontier = [Oradea, Lugoj, Rimnicu, Fagaras]

Explored Set = [Arad, Zerind, Timisoara, Sibiu]

Breadth-first search - Example

Valid Actions
Go(Timisoara)
Go(Mehadia)



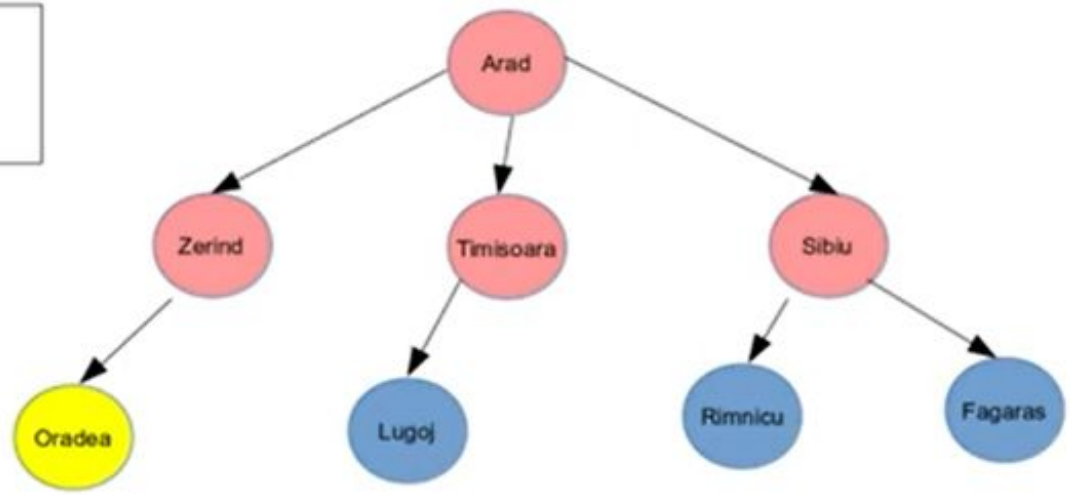


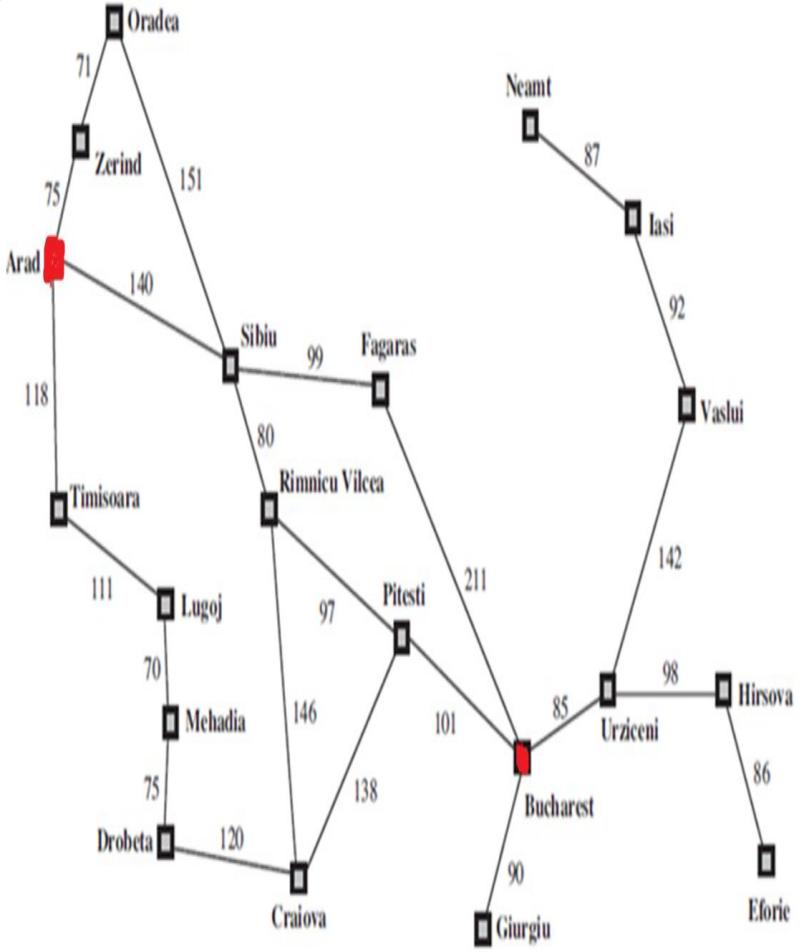
Frontier = [Rimnicu, Fagaras, Mehadia]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj]

Breadth-first search - Example

Valid Actions
Go(Zerind)
Go(Sibiu)



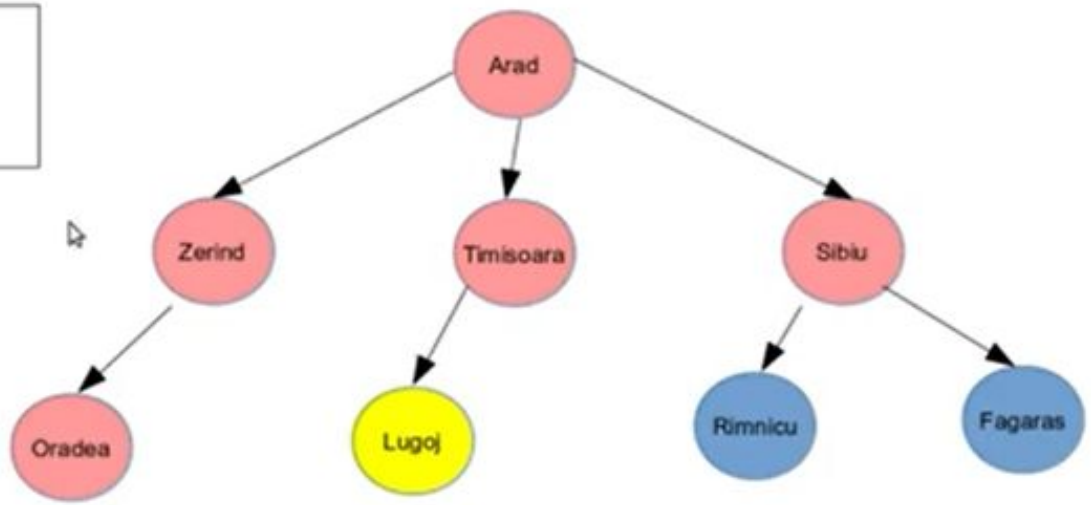


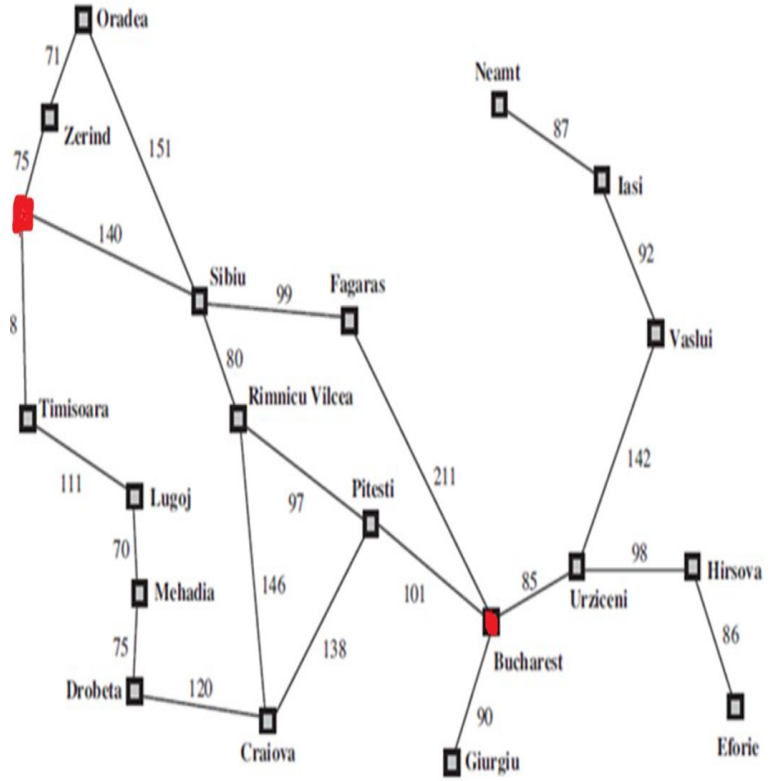
Frontier = [Lugoj, Rimnicu, Fagaras]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea]

Breadth-first search - Example

Valid Actions
Go(Timisoara)
Go(Mehadia)



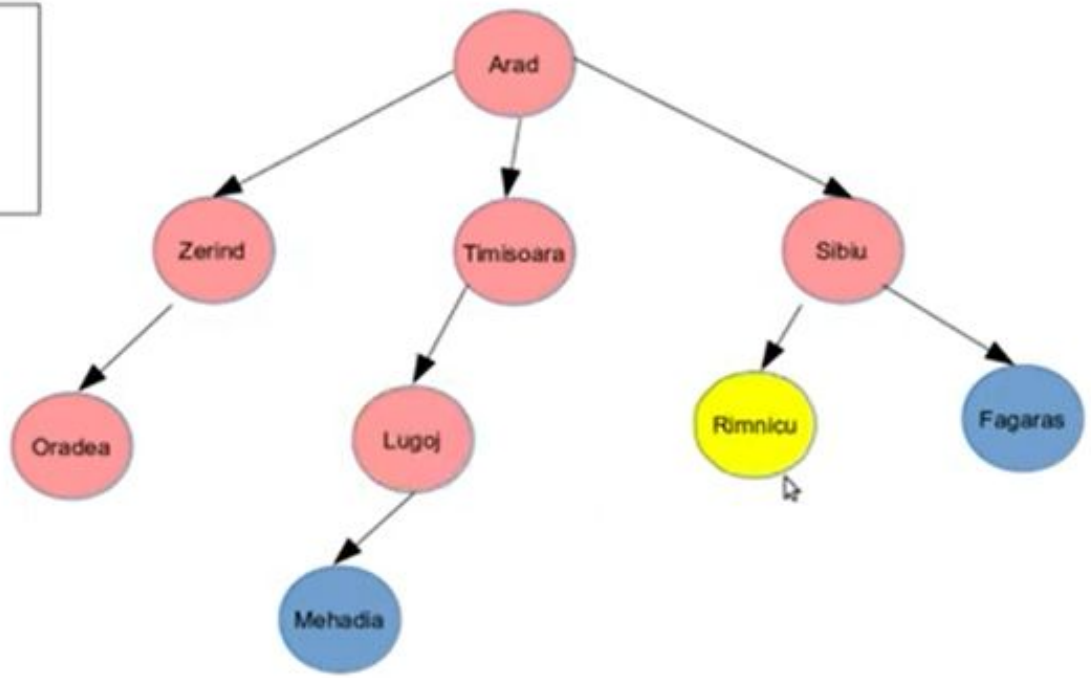


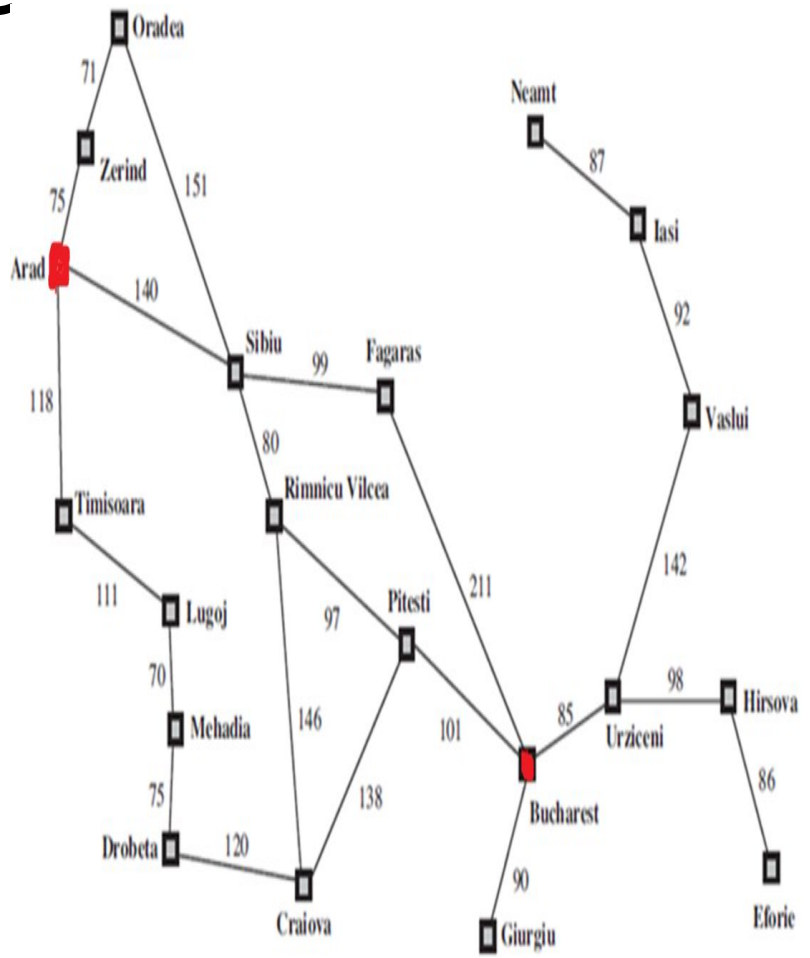
Frontier = [Rimnicu, Fagaras]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj]

Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Pitesti)
Go(Craiova)



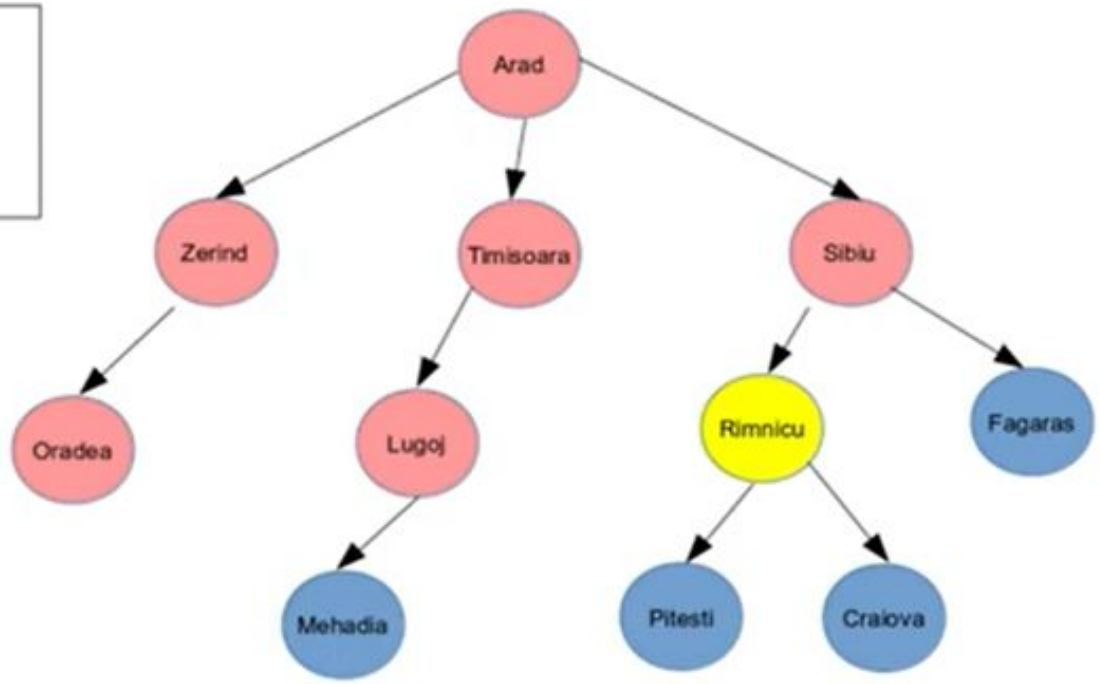


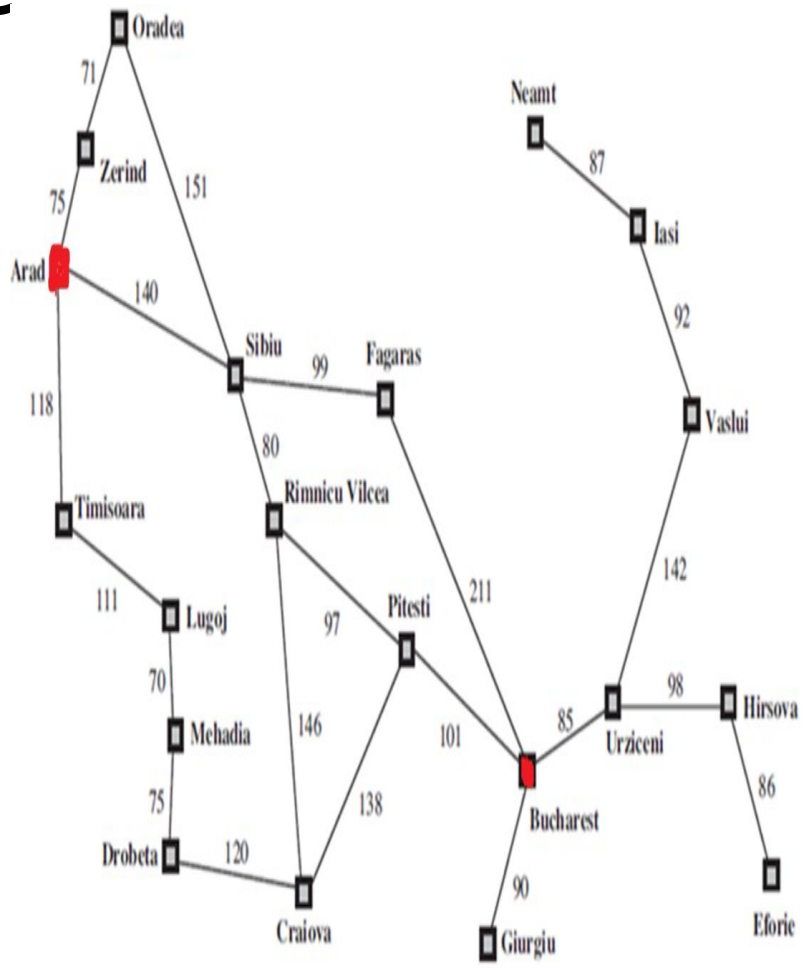
Frontier = [Fagaras, Mehadia]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu]

Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Pitesti)
Go(Craiova)



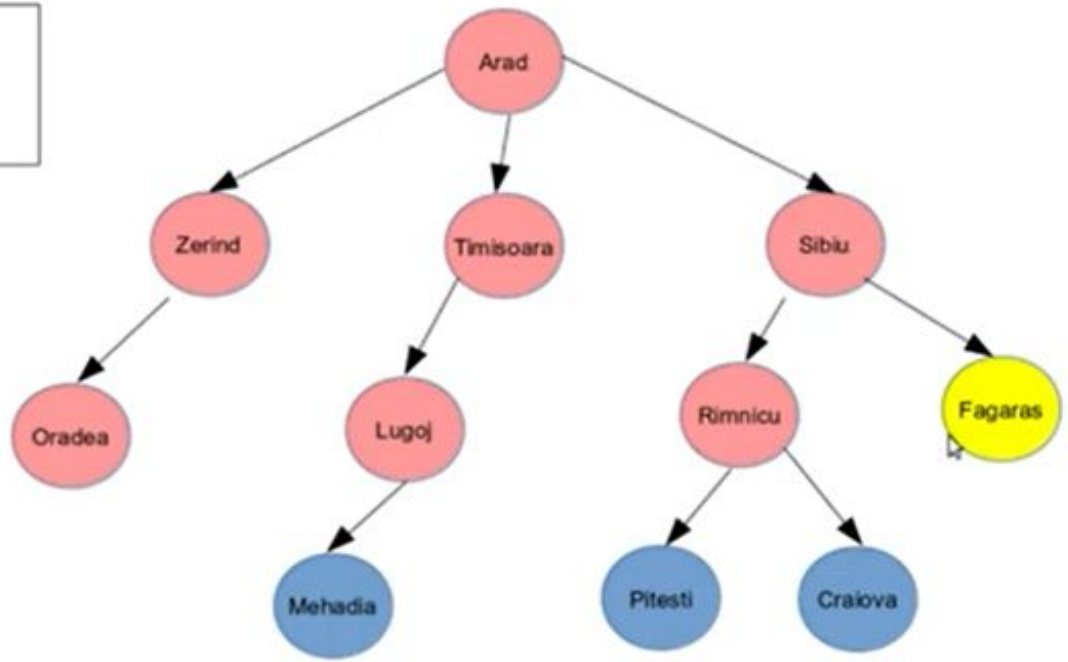


Frontier = [Fagaras, Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu]

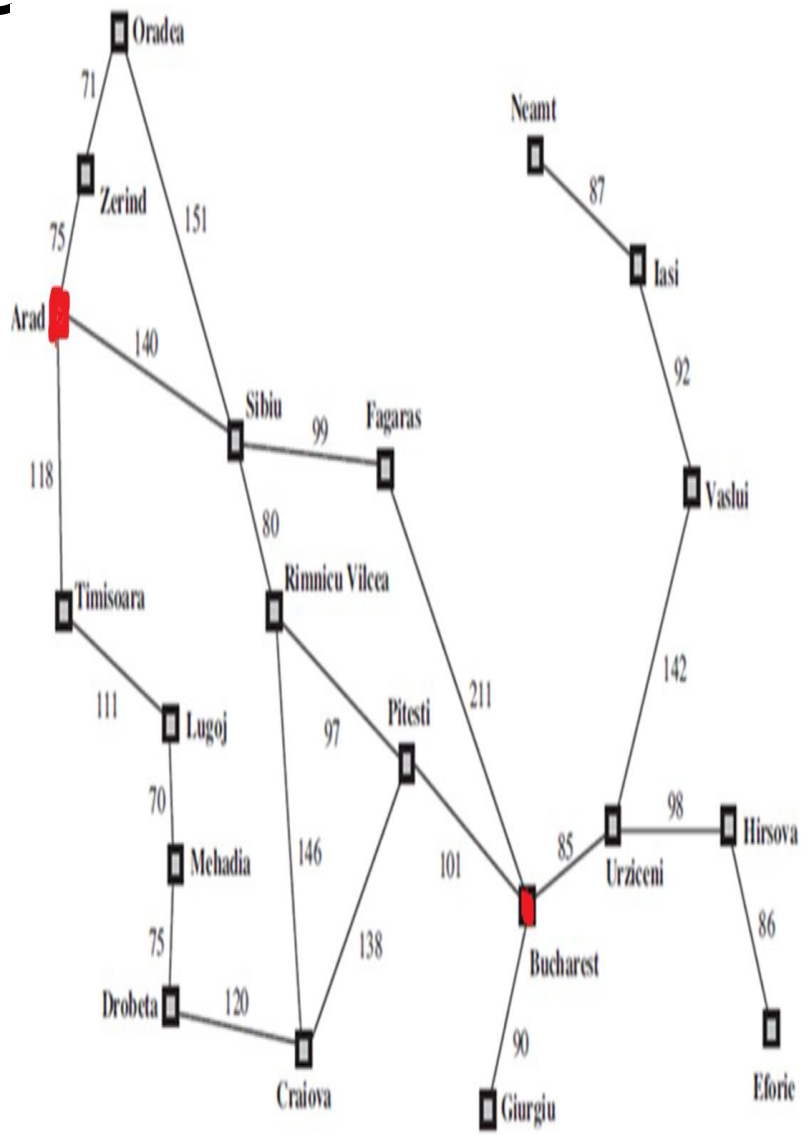
Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Bucharest)



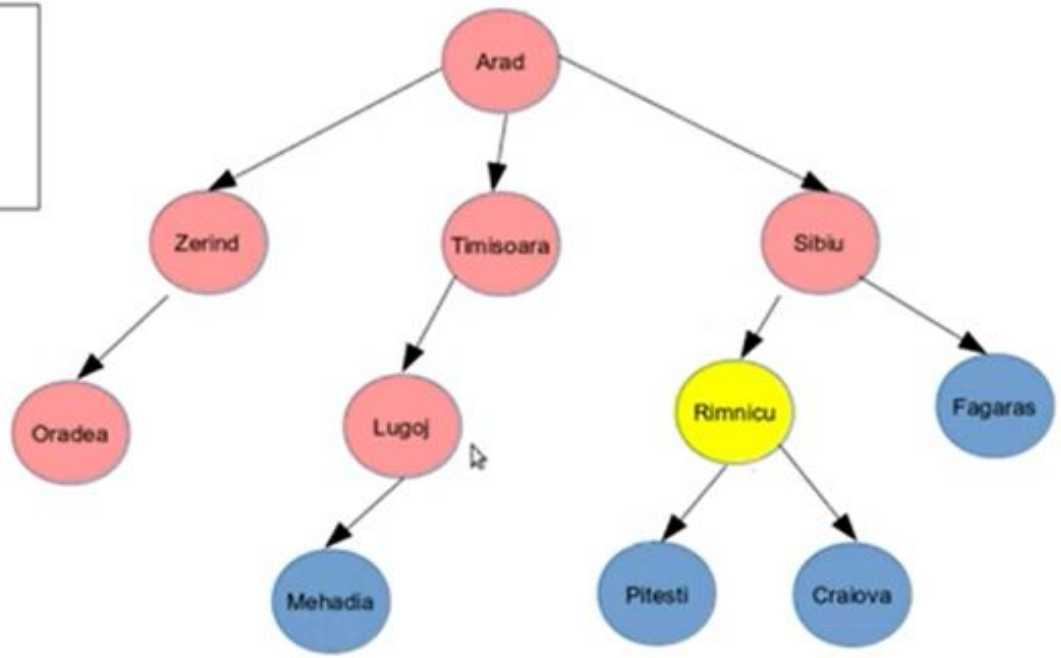
Frontier = [Mehadia, Pitesti, Craiova]

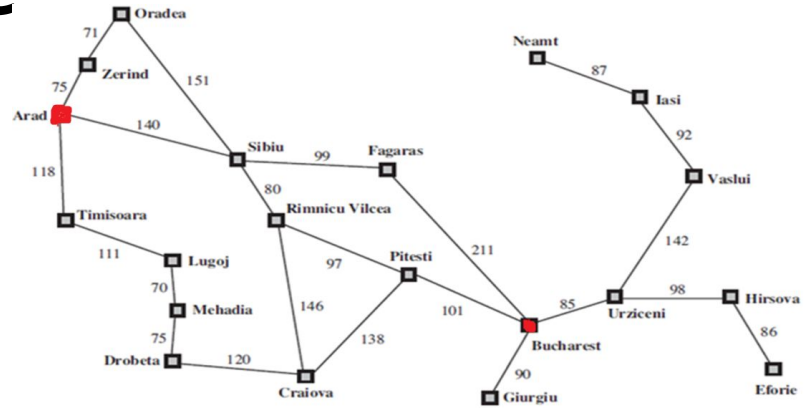
Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]



Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Pitesti)
Go(Craiova)



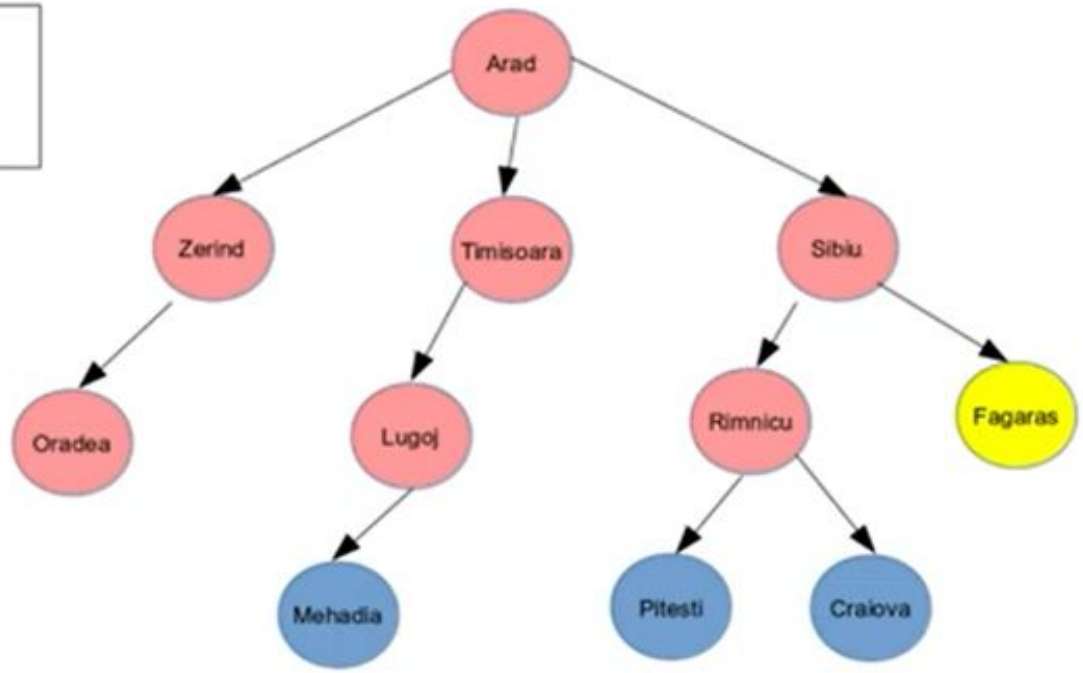


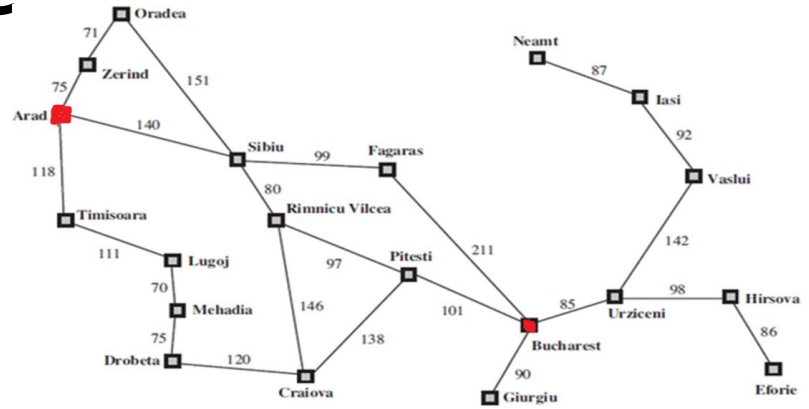
Frontier = [Fagaras, Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu]

Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Bucharest)



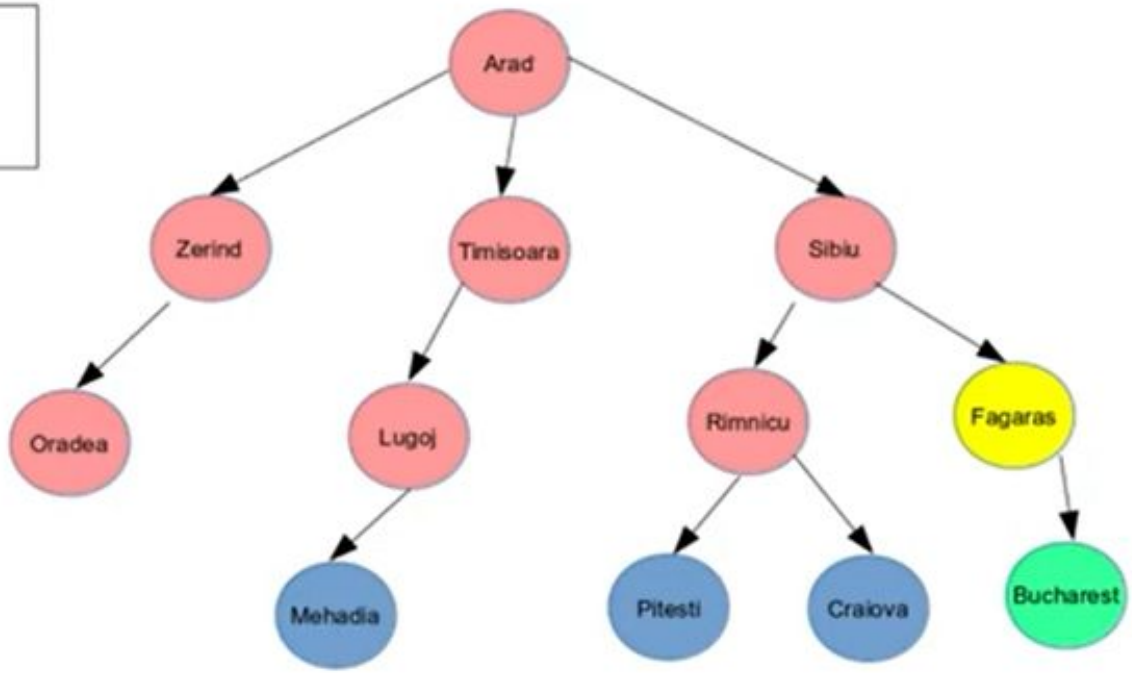


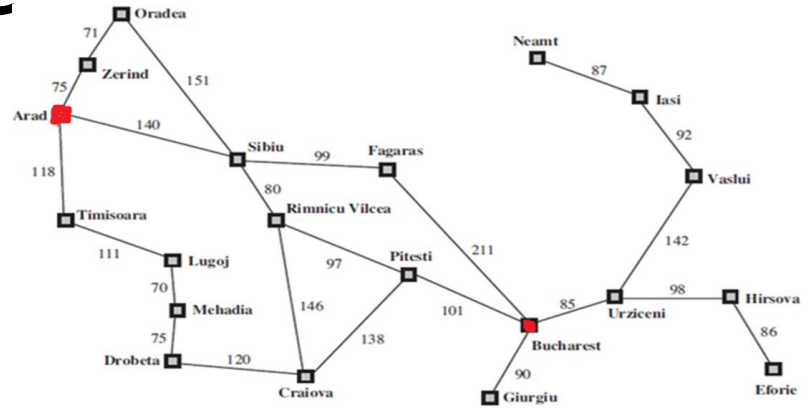
Frontier = [Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]

Breadth-first search - Example

Valid Actions
Go(Sibiu)
Go(Bucharest)

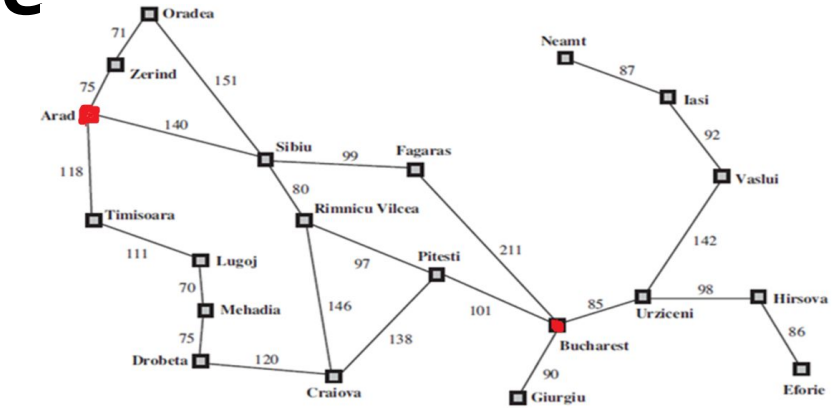




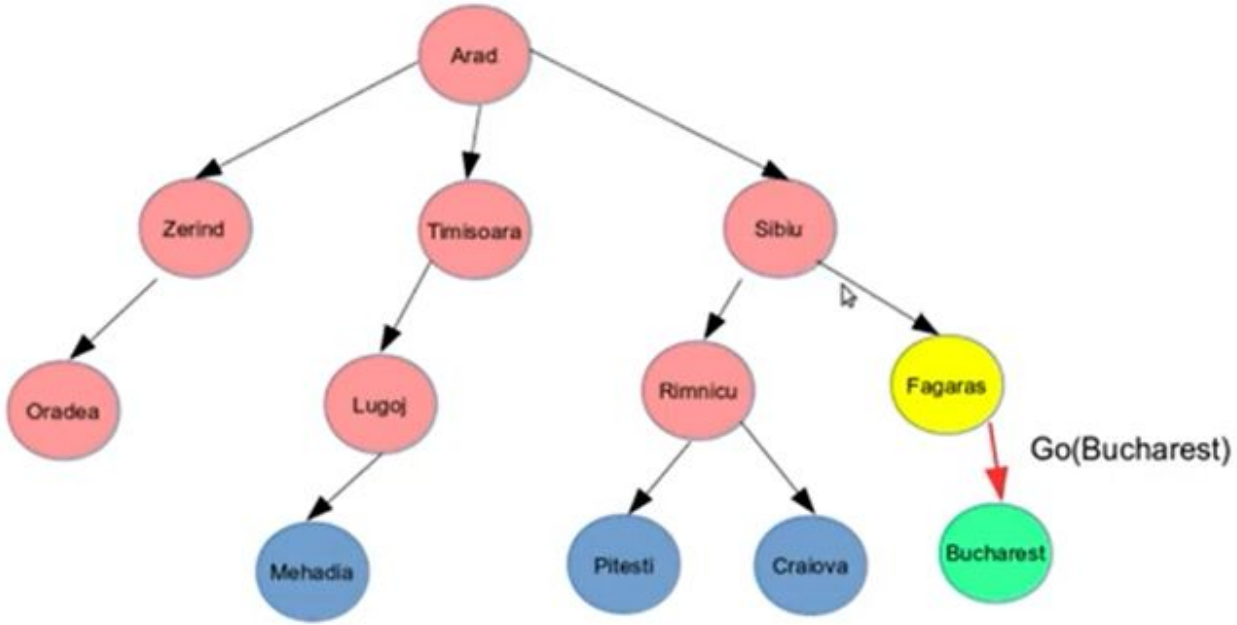
Frontier = [Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]

Breadth-first search - Example



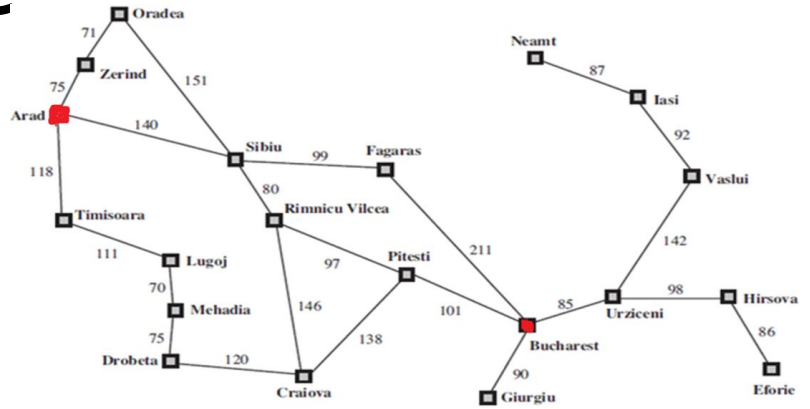
Solution = [Go(Bucharest)]



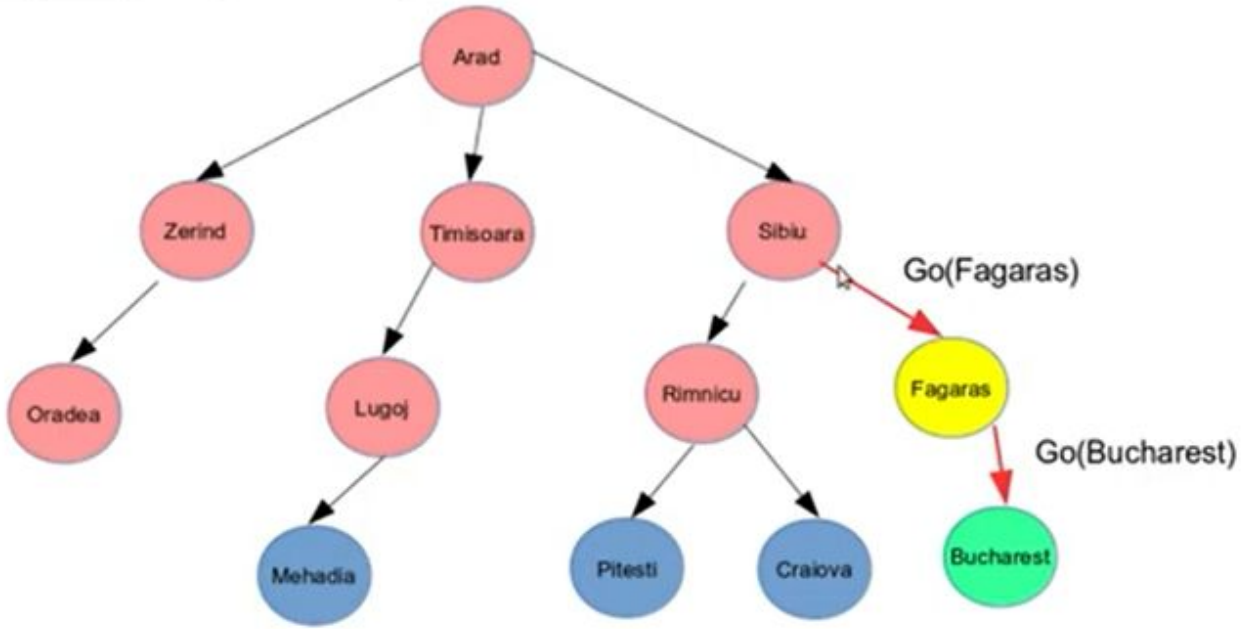
Frontier = [Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]

Breadth-first search - Example



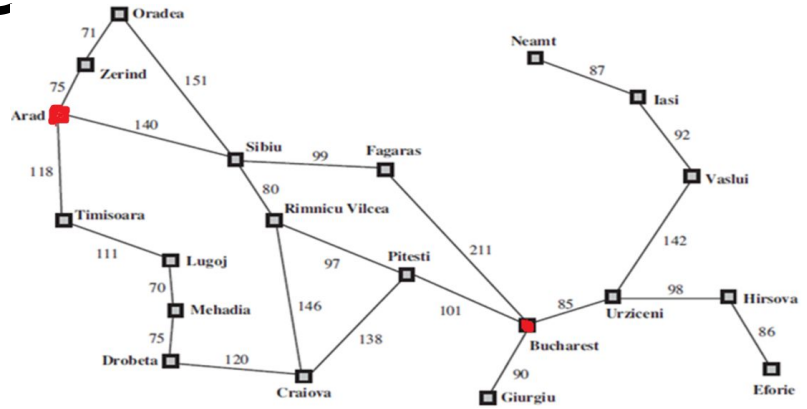
Solution = [Go(Fagaras), Go(Bucharest)]



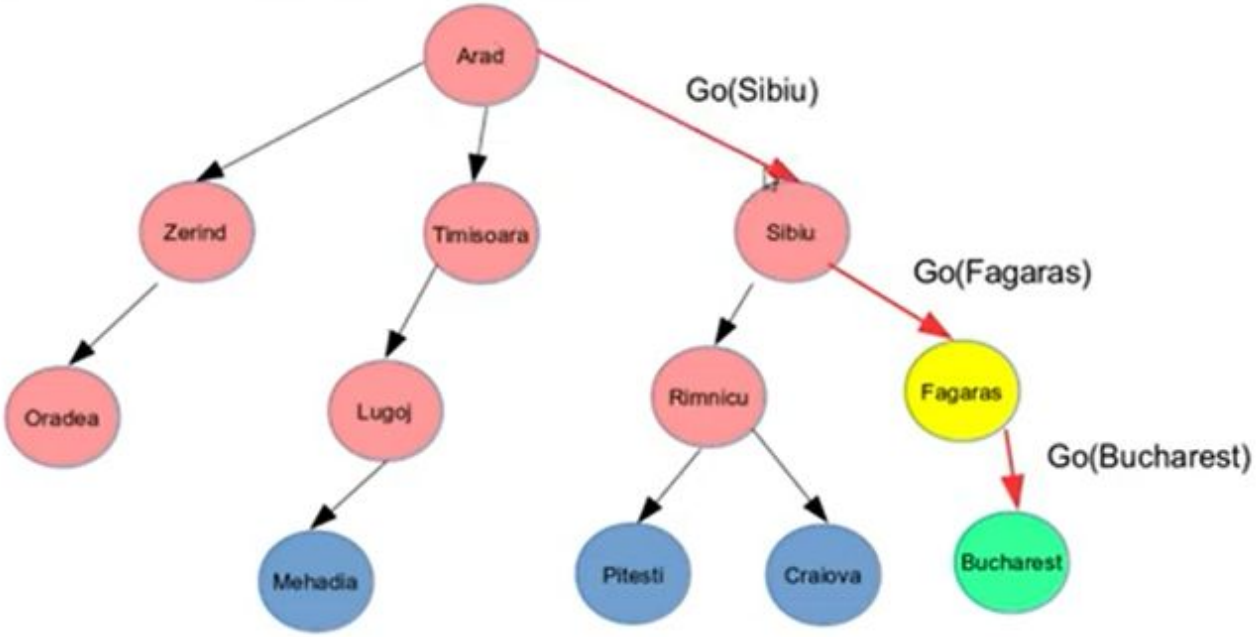
Frontier = [Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]

Breadth-first search - Example



Solution = [Go(Sibiu), Go(Fagaras), Go(Bucharest)]



Frontier = [Mehadia, Pitesti, Craiova]

Explored Set = [Arad, Zerind, Timisoara, Sibiu, Oradea, Lugoj, Rimnicu, Fagaras]

Problem solving performance-BFS

complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes

not optimal one: breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

Time Complexity: The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of *these* generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Space Complexity: every node generated remains in memory. 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 $O(b^d)$, exponential complexity

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.

Summary

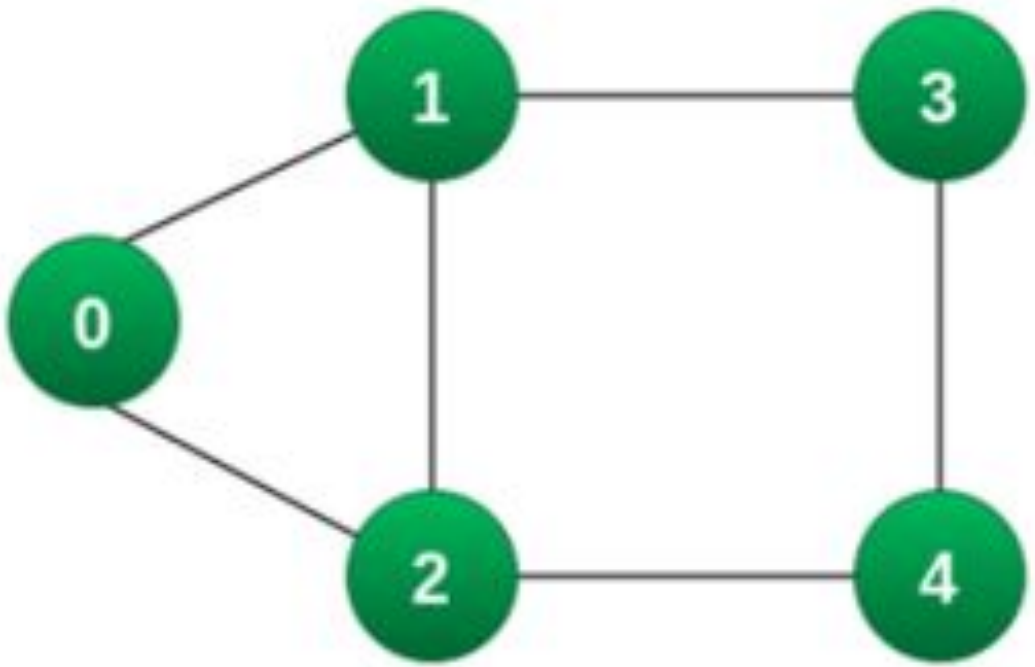
the memory requirements are a bigger problem for breadth-first search than is the execution time

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.

If your problem has a solution at depth 16, then it will take about 350 years for breadth-first search to find it.

In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

Explain the working of Breadth first algorithm for the following graph with initial state 0 and goal state 4.



Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- Uniform-cost search algorithm 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 g(n).
- This is done by storing the frontier as a priority queue ordered by g.

```
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
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

- 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.

problem is to get from Sibiu to Bucharest

The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.

The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177.

The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310.

Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278.

Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

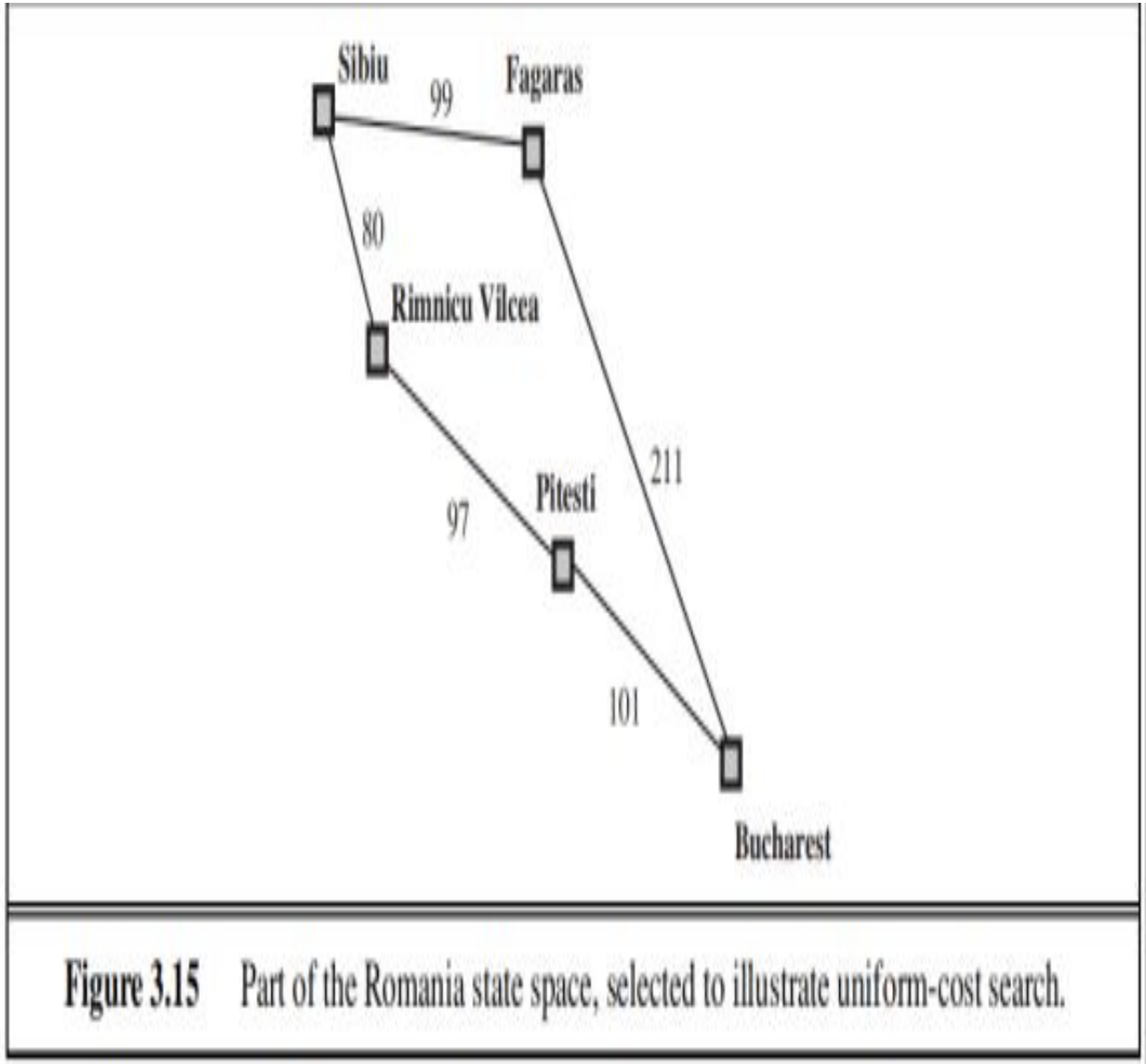


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Problem solving performance-UCS

Complete: guaranteed provided the cost of every step exceeds some small positive constant ε

Optimal: optimal path to that node has been found. because step costs are nonnegative, paths never get shorter as nodes are added. uniform-cost search expands nodes in order of their optimal path cost.

Time Complexity: # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution

Space Complexity: # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$

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

let C^* be the cost of the optimal solution and that every action costs at least ε

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

This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps.

When all step costs are equal $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}.

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

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.

- The depth-first search algorithm is an instance of the graph-search algorithm, whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- A LIFO queue means that the most recently generated node is chosen for expansion.
- This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

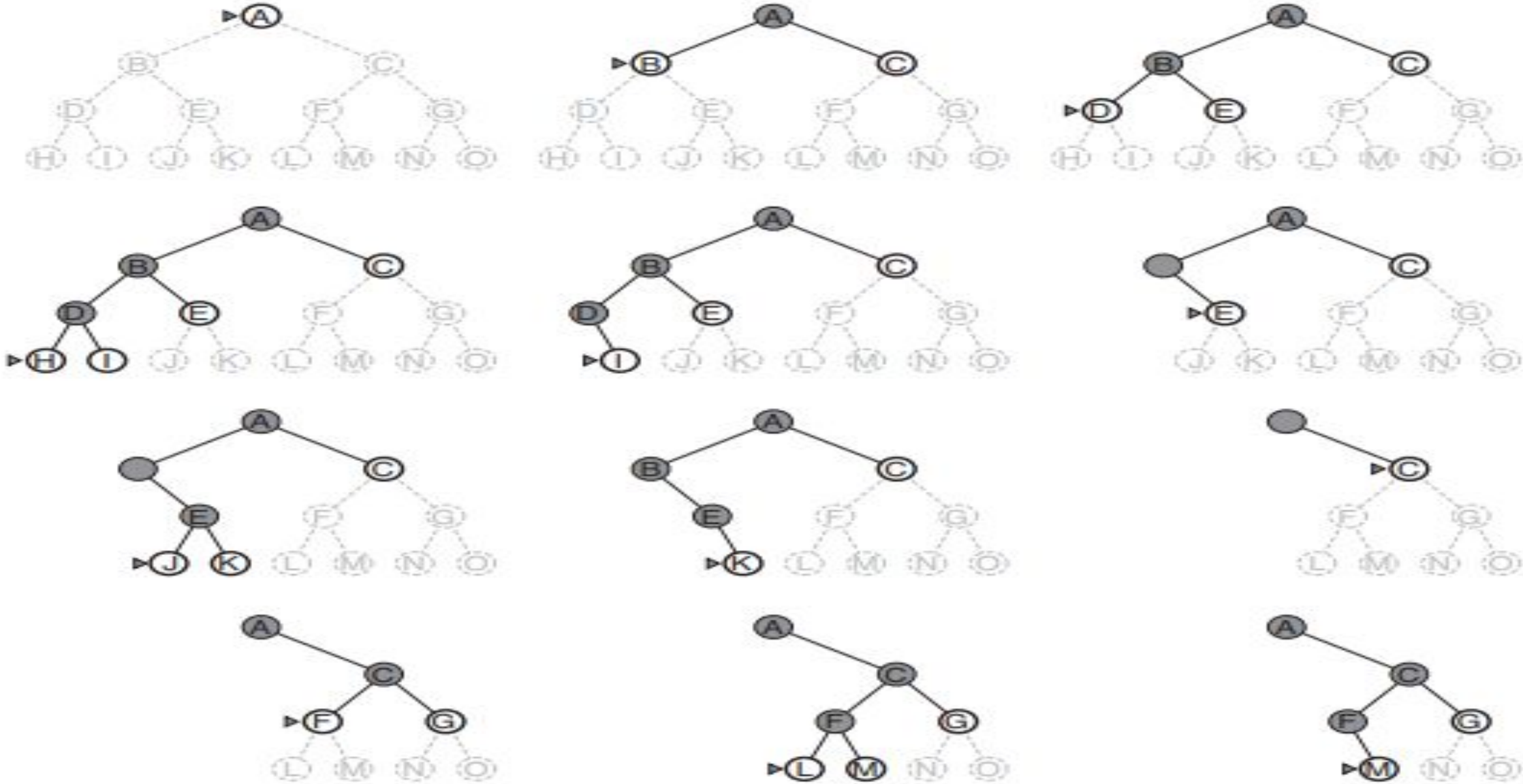


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

- As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.
- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
- The tree-search version, on the other hand, is not complete.
- Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths.

The general Graph-Search Algorithm (Depth First Search Algorithm)

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~~ remove the last node inserted from frontier

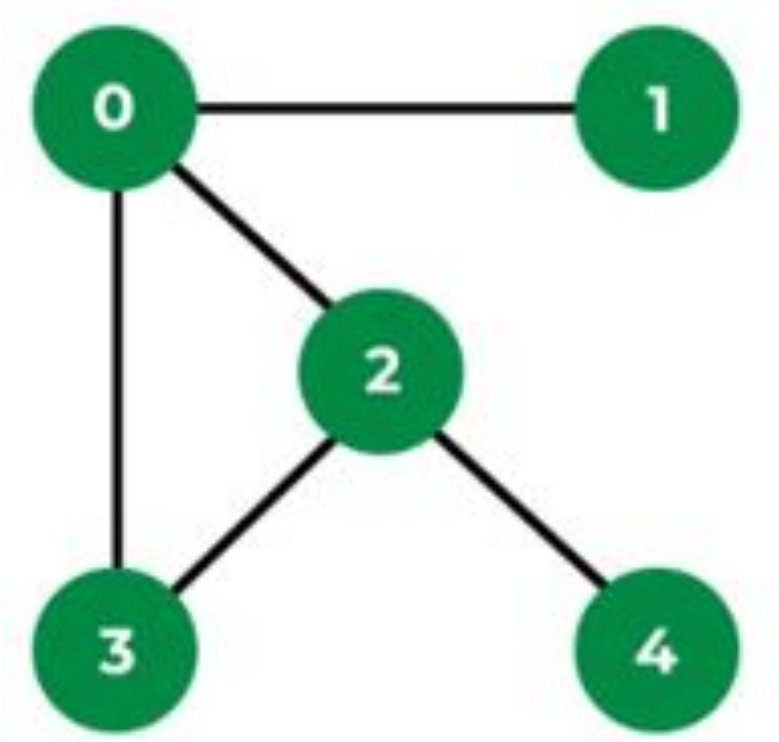
if the node contains a goal state **then return** the corresponding solution

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***

Depth-first search on a graph

- Explain the working of Depth first algorithm for the following graph with initial state 0 and goal state 4.



- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
- The tree-search version is *not* complete.
- Both versions are nonoptimal.
- The time complexity of depth-first graph search is bounded by the size of the state space.
- A depth-first tree search may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node.

  Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

- Depth-first search seems to have no clear advantage over breadth-first search.
- But, DFS is better for the space complexity.
- For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.

  Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(bm) nodes.

Backtracking Search

- **Backtracking search** is a variant of depth-first search.
- It uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.

  In this way, only O(m) memory is needed rather than O(bm). Backtracking search
- Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and O(m) actions.
- For this to work, we must be able to undo each modification when we go back to generate the next successor.
- For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

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. (This is likely when d is unknown.)

Depth-limited search will also be non optimal if we choose $l > d$.

Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$

Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.

- Sometimes, depth limits can be based on knowledge of the problem.

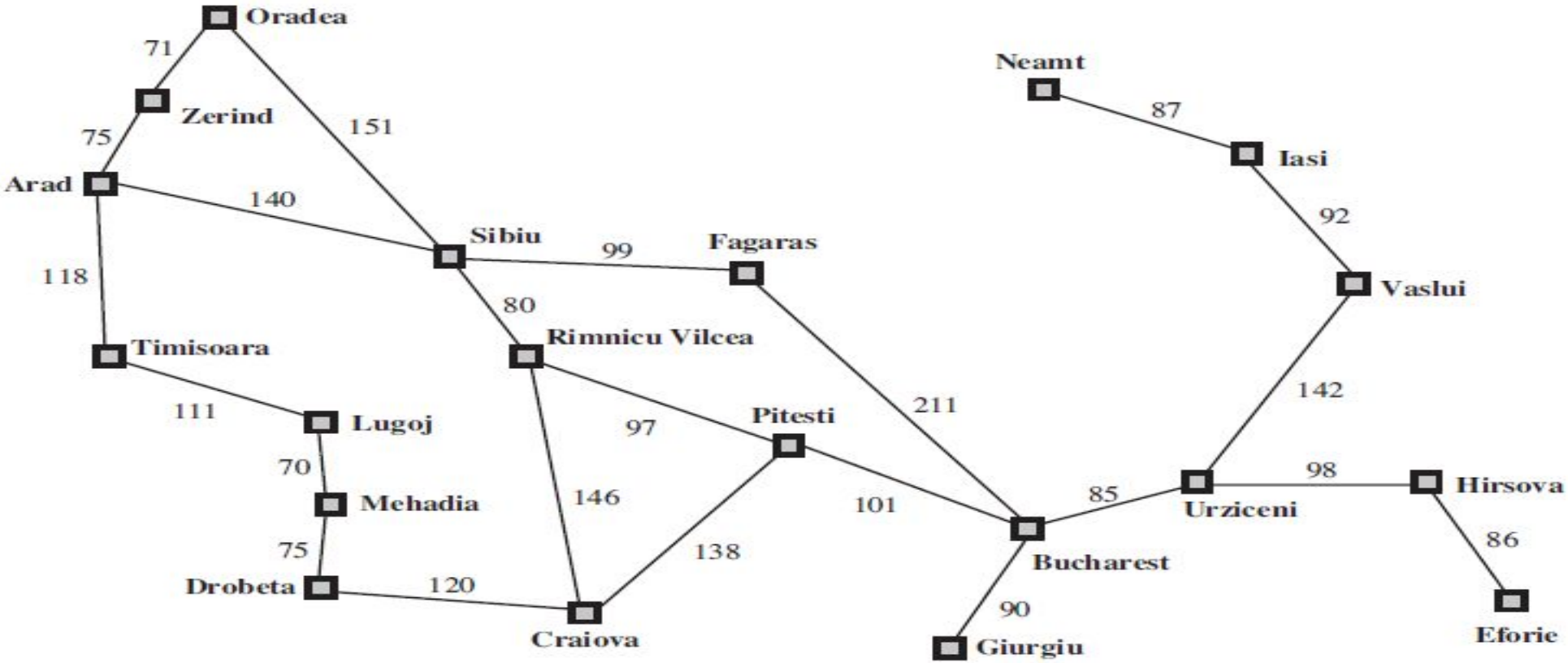
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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 3.17 A recursive implementation of depth-limited tree search.

Depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

A simplified road map of Romania



- Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities.
- Therefore, we know that if there is a solution, it must be of length 19 at the longest, so ‘l’ = 19 is a possible choice.
- The diameter of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.
- For most problems, however, we will not know a good depth limit until we have solved the problem.

- Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm.
- Alternatively, it can be implemented as a simple recursive algorithm.
- Notice that depth-limited search can terminate with two kinds of failure:

  -the standard failure value indicates no solution;

  -the cutoff value indicates no solution within the depth limit.

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.
- 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.

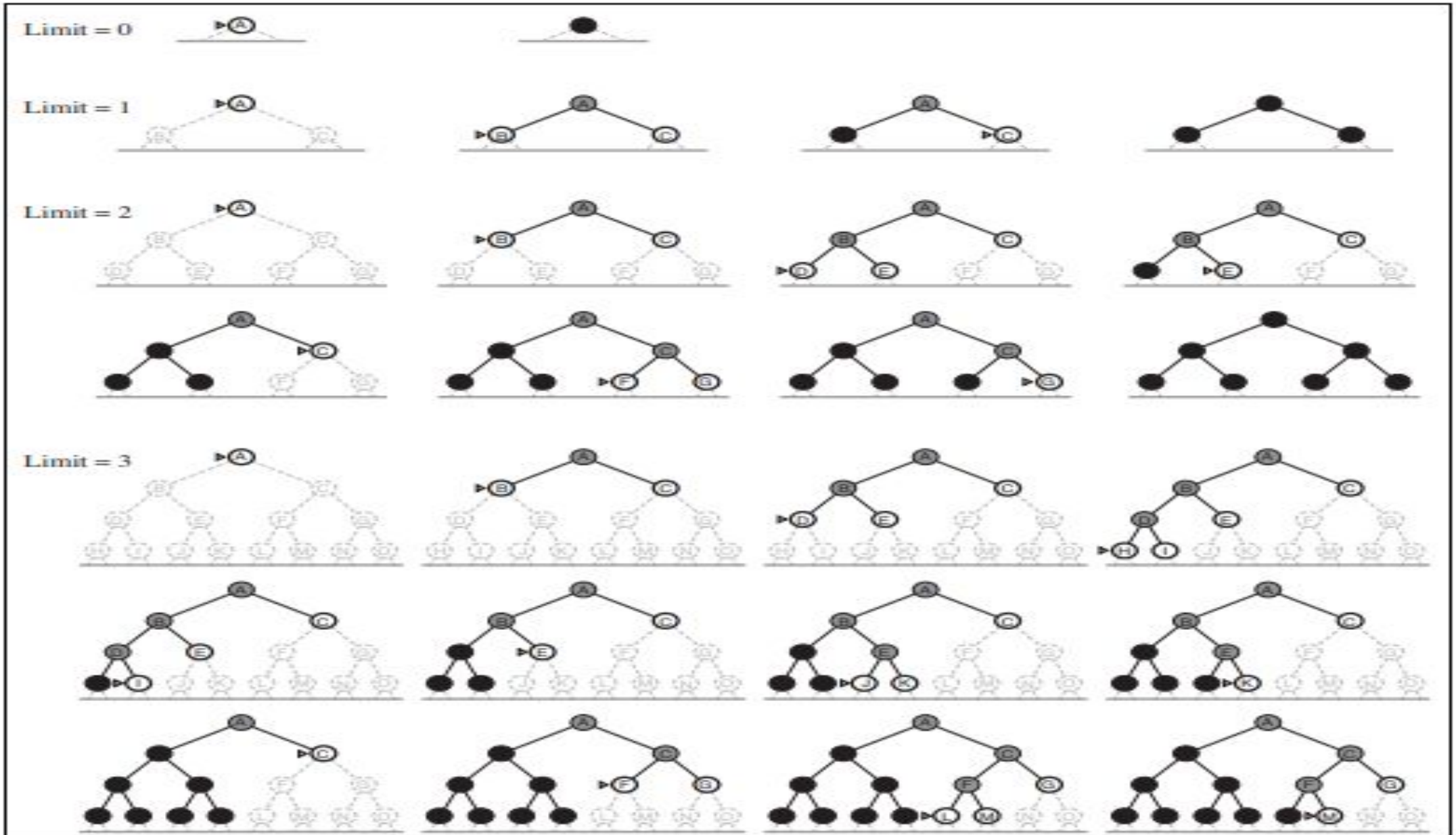


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

```
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
```

Figure 3.18 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.

- 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 search may seem wasteful because states are generated multiple times. It turns out this is not too costly.

- The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.
- In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times.
- So the total number of nodes generated in the worst case is

$$N(\mathrm{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d ,$$

which gives a time complexity of $O(b^d)$—asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 .$$

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.

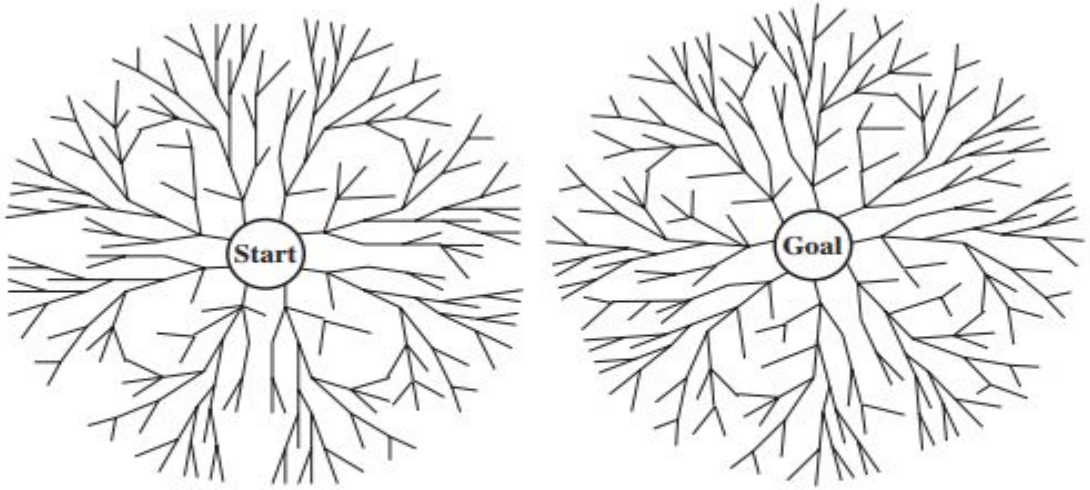


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

- 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 first such solution found may not be optimal, even if the two searches are both breadth-first; some additional search is required to make sure there isn’t another short-cut across the gap.
- The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time.

- The time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$.
- The space complexity is also $O(b^{d/2})$.
- We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done.
- This space requirement is the most significant weakness of bidirectional search.

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]

Figure 3.21 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. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.