CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/

Solving problems by search

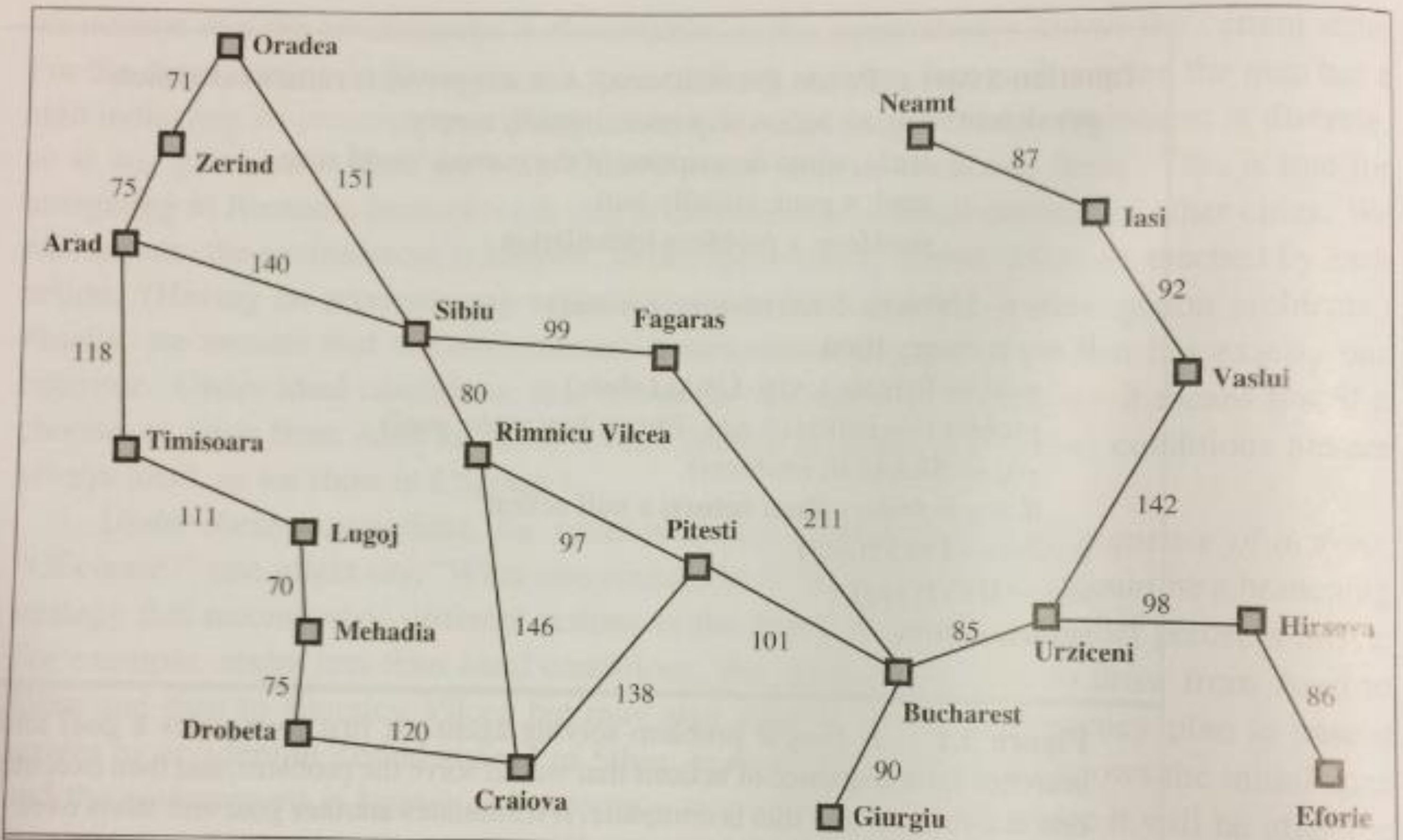
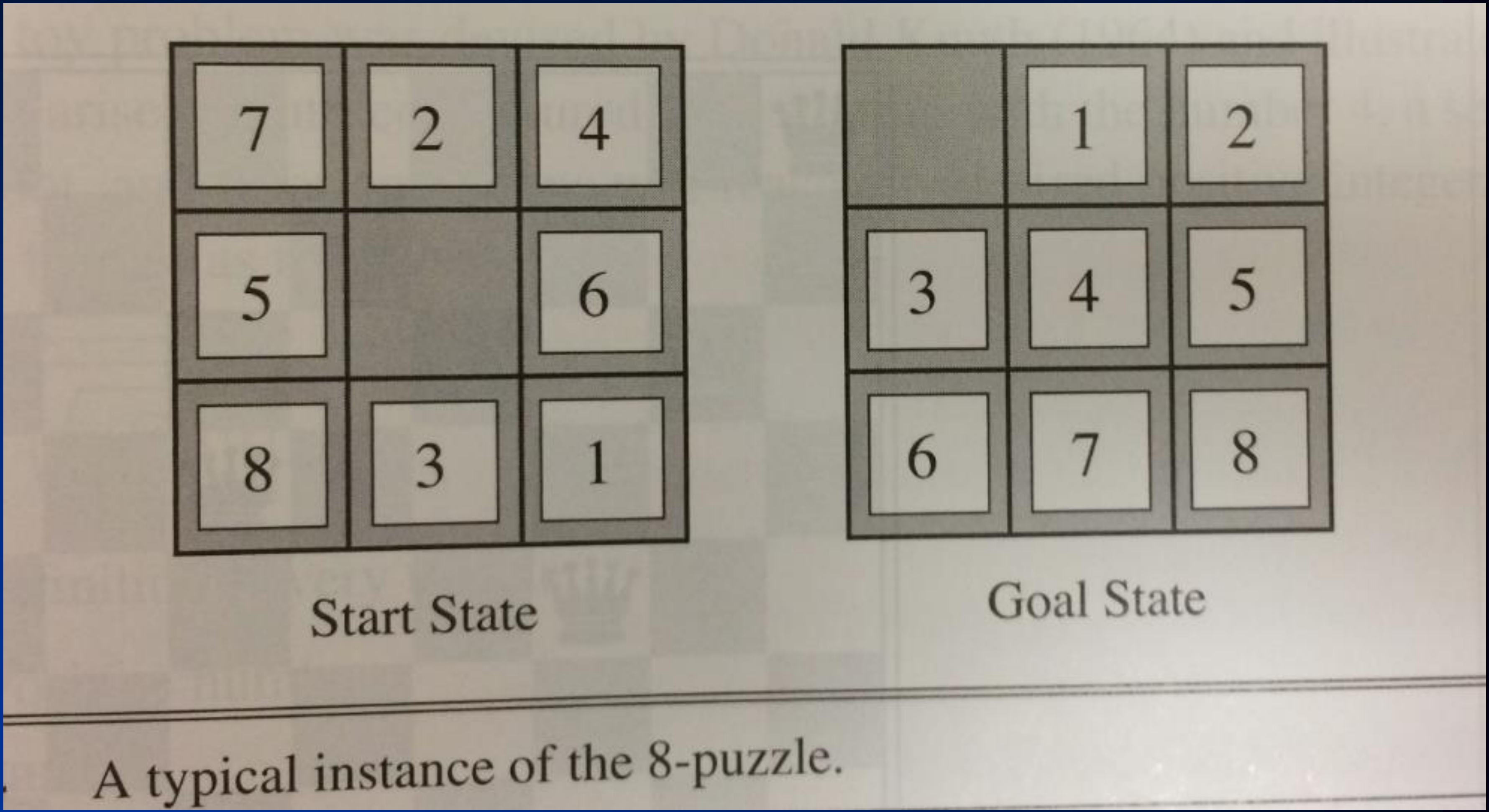


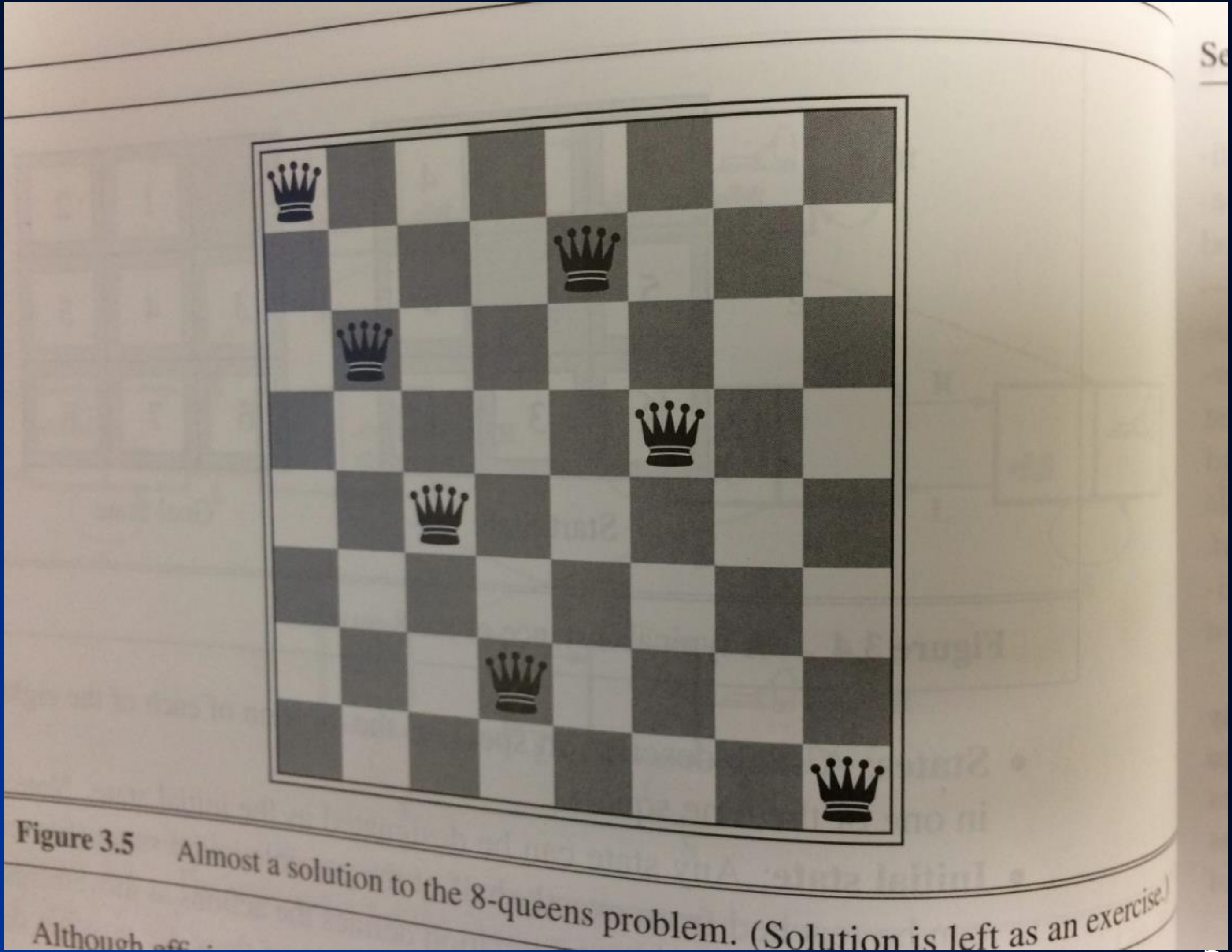
Figure 3.2 A simplified road map of part of Romania.

8-puzzle



A typical instance of the 8-puzzle.

8-queens



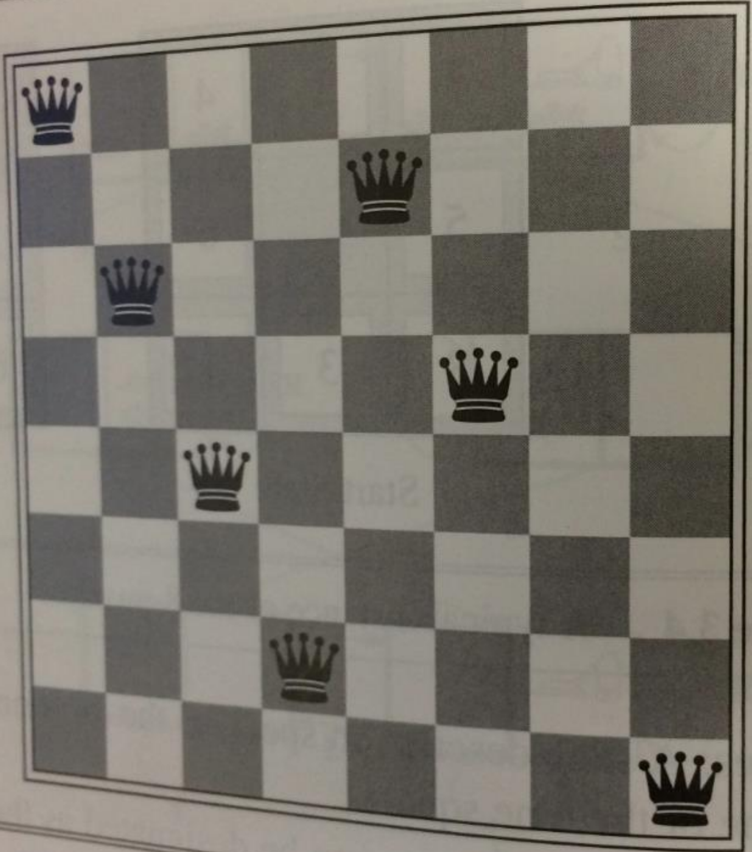


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Although eff

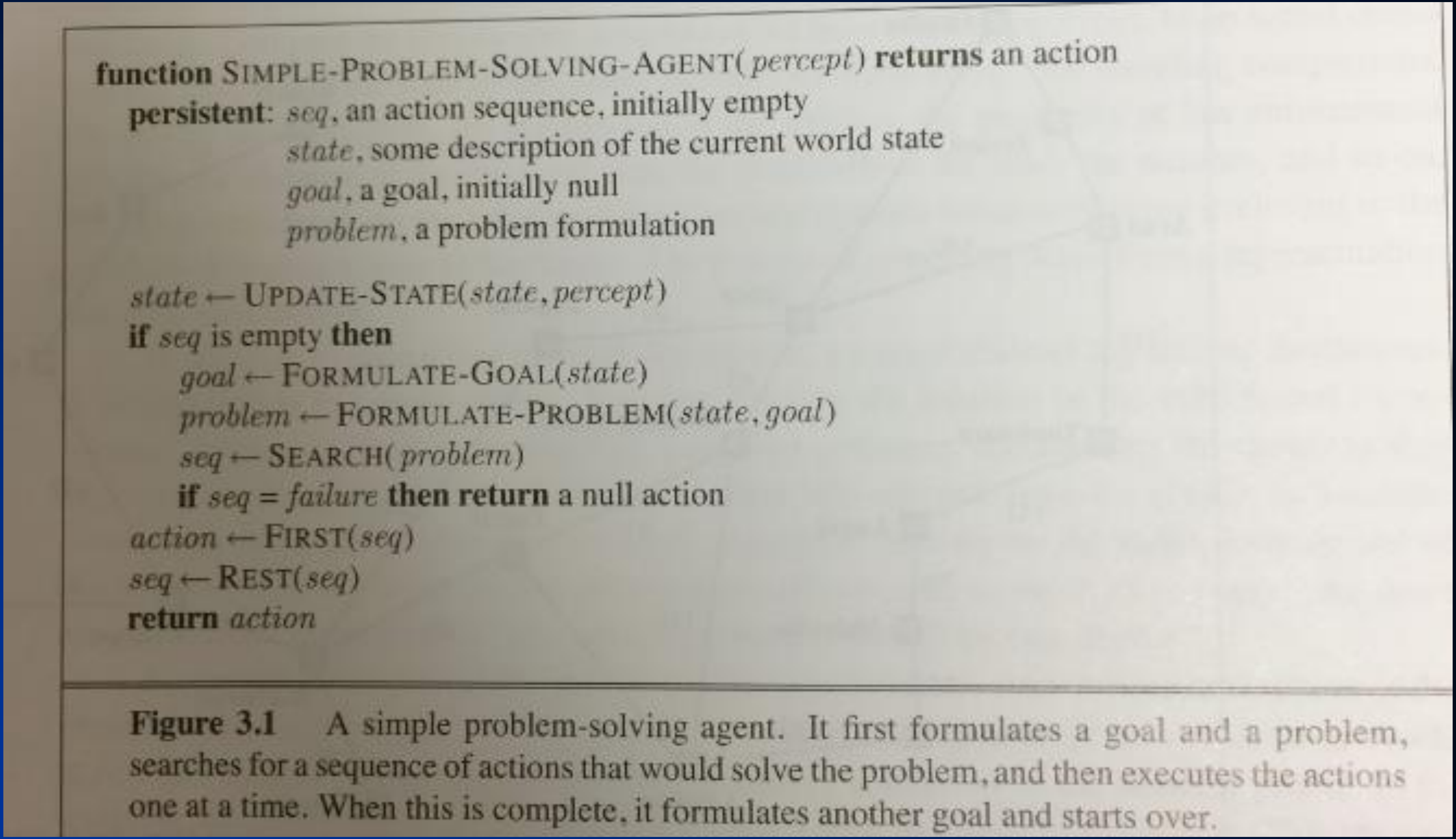
Search problem definition

- States
- Initial state
- Actions
- Transition model
- Goal test
- Path cost

Definition for 8-queens problem

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square
- **Goal test**: 8 queens are on the board, none attacked
- **Path cost**: (Not applicable)

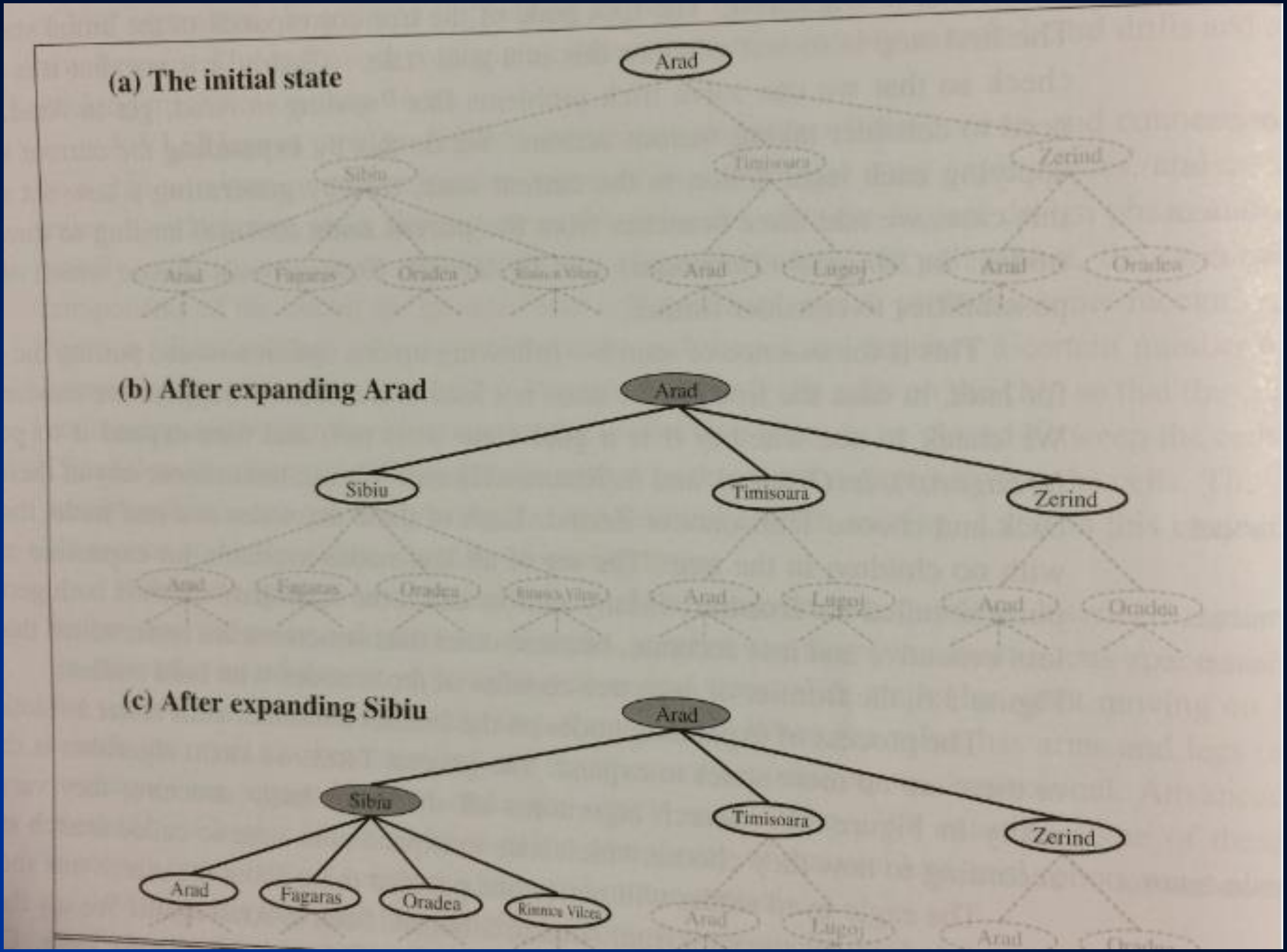
Problem-solving approach



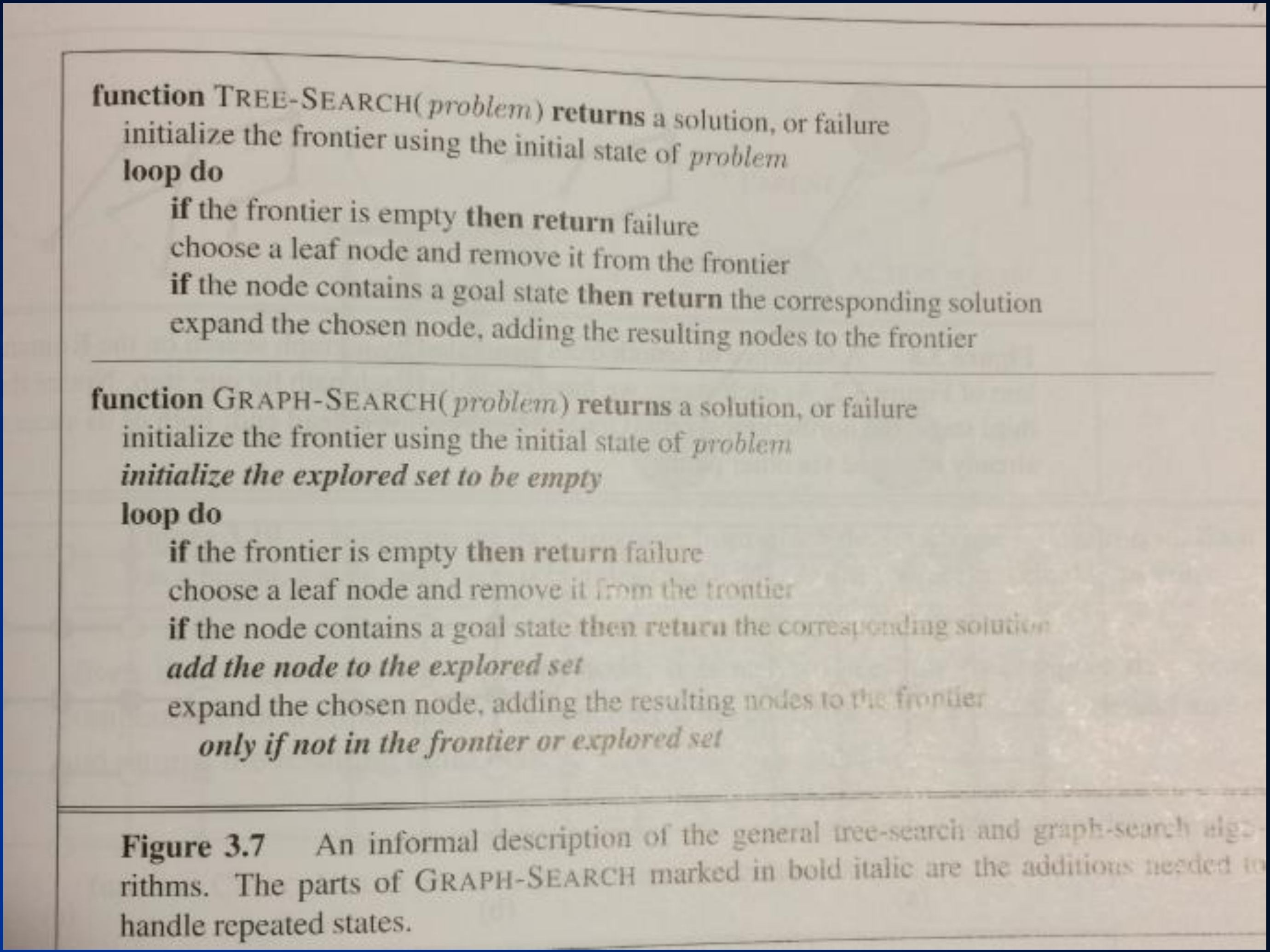
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  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
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Searching Romania



General tree and graph search algorithm



```
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
```

Figure 3.7 An informal description of the general tree-search and graph-search alg rithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed t handle repeated states.

Real-world search

- **Driving directions** (e.g., Google maps)
- **Airline travel problems**: Find “optimal” flight subject to conditions entered by user (e.g., for kayak.com)
- **Tourism problems**: E.g., “Visit every city in Romania map at least once, starting and ending in Bucharest.”
- **Traveling salesman problem**: Find shortest tour in which each city visited exactly once
- **VLSI layout**: Positioning millions of components and connections on a chip to minimize area, circuit delays, stray capacitances, and maximize manufacturing yield
- **Robot navigation**: generalization of route-finding problem to continuous space with potentially infinite set of actions and states.
- **Automatic assembly sequencing** of complex objects by a robot, e.g., electric motors and protein design

Evaluating performance

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution, as defined on page 68 (i.e., lowest path cost among all solutions)
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

Complexity

- Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, |V| + |E|, where V is the set of vertices (nodes) of the graph and E is the set of edges (links). This is appropriate when the graph is an explicit data structure that is input to the search program (e.g., map of Romania).

Complexity

- In AI, the graph is often represented *implicitly* by the initial state, actions, and transition model, and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:
  - b, the **branching factor** or maximum number of successors of any node;
  - d, the **depth** of the shallowest goal node (i.e., the number of steps along the path from the root);
  - m, the maximum length of any path in the state space
- Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

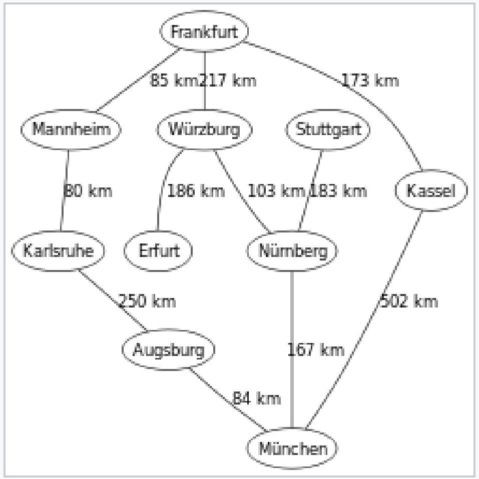
Search algorithm effectiveness

- To assess the effectiveness of a search algorithm we can consider just the **search cost**—which typically depends on the time complexity but can include a term for memory usage– or we can use **total cost**, which combines the search cost and the path cost of the solution found.
- For problem of finding route from Arad to Bucharest, the search cost is the amount of time taken by the search (milliseconds) and the solution cost is the total length of the path in kilometers.
  - To compute total cost, we can convert km to ms by using a car's average speed.

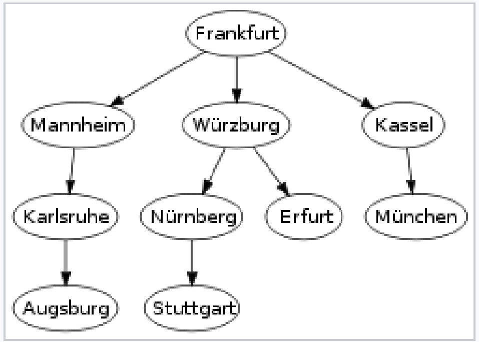
Uninformed and informed search

- For **uninformed search** (aka **blind search**), 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” that another are called **informed search** or **heuristic search** strategies.

Breadth-first search (BFS)



An example map of Germany with some connections between cities



The breadth-first tree obtained when running BFS on the given map and starting in Frankfurt

Breadth-first search (BFS)

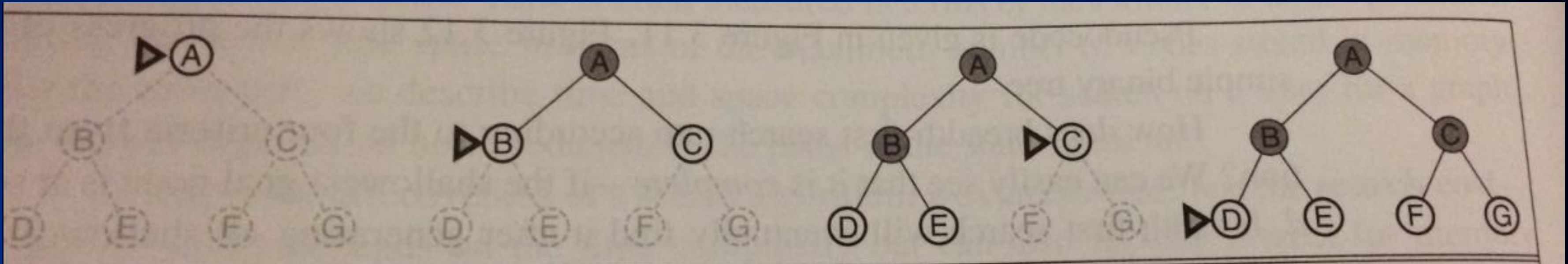
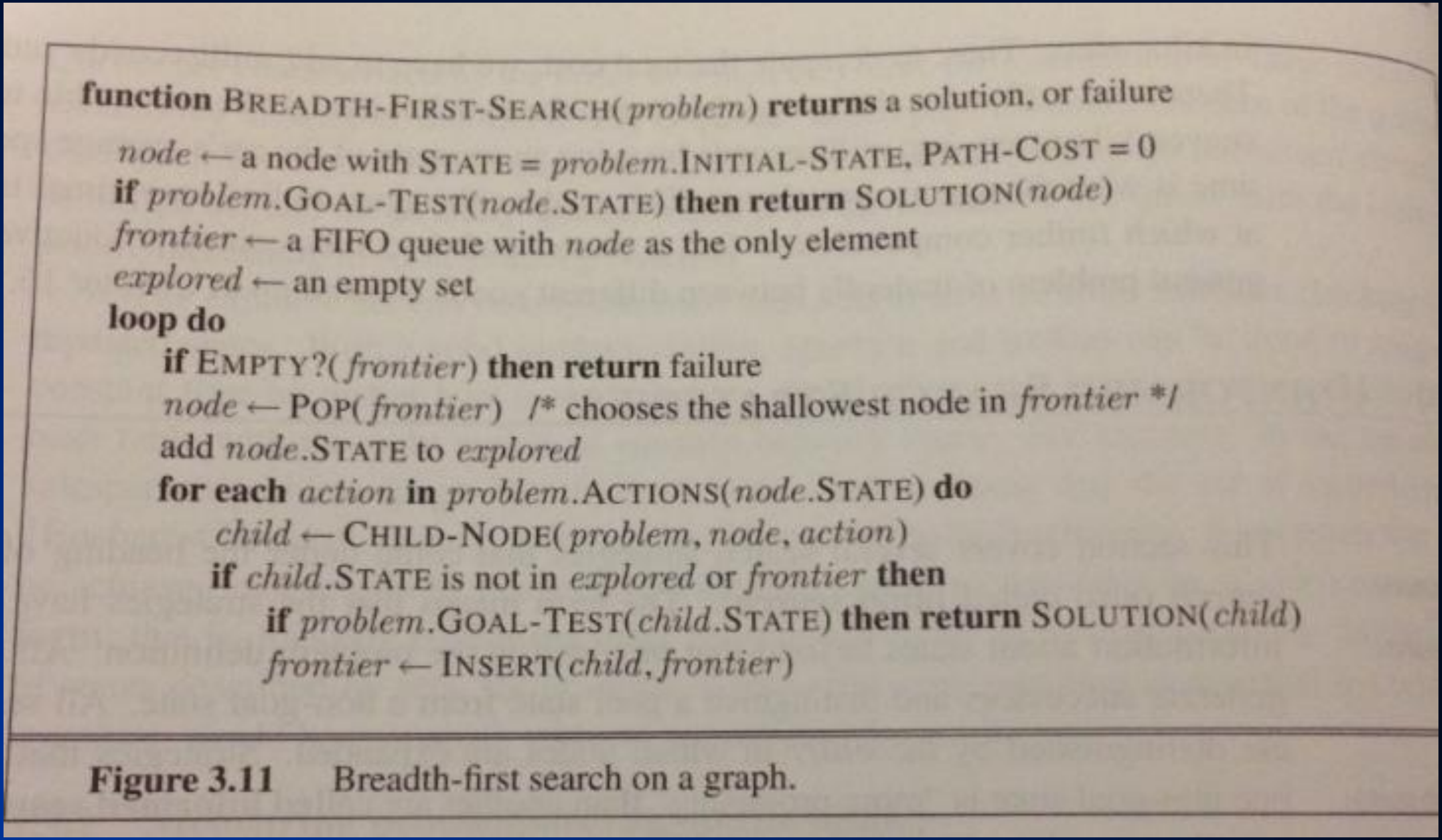


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

BFS

- *Shallowest* unexpanded node is chosen next for expansion
- Uses first-in-first-out (FIFO) queue
  - Queue: “pops” oldest element (first in)
  - Stack: pops newest element (LIFO queue)
  - Priority queue: pops element with highest “priority”
- One slight tweak on the general graph-search algorithm is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion

BFS



```
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.

BFS

- Is BFS "complete"?
  - Is BFS guaranteed to find a solution when one exists?

BFS

- Yes we can easily see that it is *complete* – if the shallowest goal node is at some finite depth d, BFS will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now the *shallowest* goal node is not necessarily the *optimal* one; technically BFS 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.

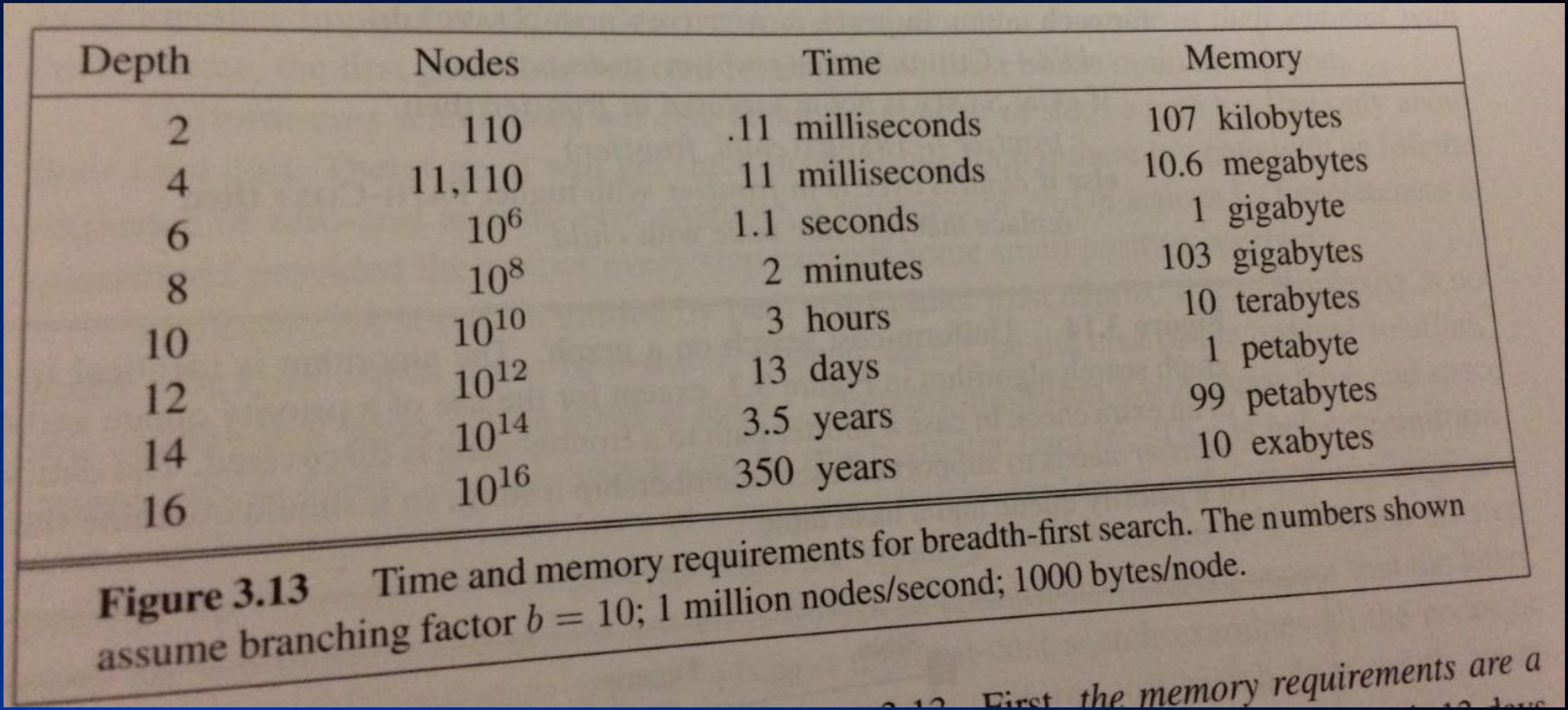
BFS running time

- b nodes at first level each of which generates b more nodes at second level, for a total of b^2 at the second level, b^3 at third level, etc. If we suppose that the solution is at depth d, then 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 + … + b^d = O(b^d)
- If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer at depth d would be expanded before the goal was detected and the time complexity would be O(b^(d+1))

BFS memory

- For any kind of graph search, which stores every expanded node in the *explored* set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search, 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), i.e., it is dominated by the size of the frontier.
- Switching to a tree search would not save much space, and in a state space with many redundant paths switching could cost a great deal of time.

BFS



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.

First, *the memory requirements are a*

BFS lessons

- *The memory requirements are a bigger problem for BFS 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. Fortunately, other strategies require less memory.
- The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for BFS (or indeed any uninformed search) to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

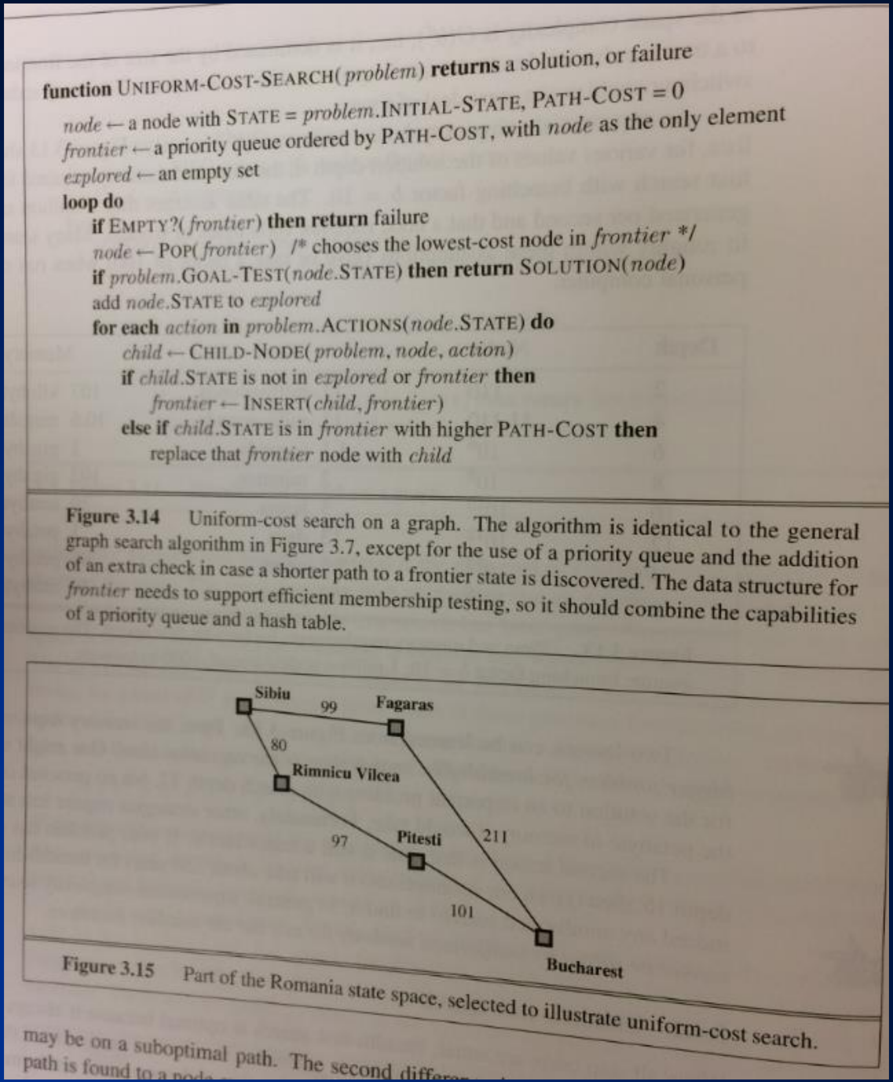
Uniform-cost search (UCS)

- When all step costs are equal, BFS is optimal because it always expands the *shallowest* unexpanded node. By 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 g(n)*. This is done by storing the frontier as a priority queue ordered by *g*.

Uniform-cost search

- In addition to the ordering of the queue by path cost, there are two other significant differences from BFS. The first is that the goal test is applied to a node when it is *selected for expansion* (as in the generic graph-search algorithm) 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.

UCS



```
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.

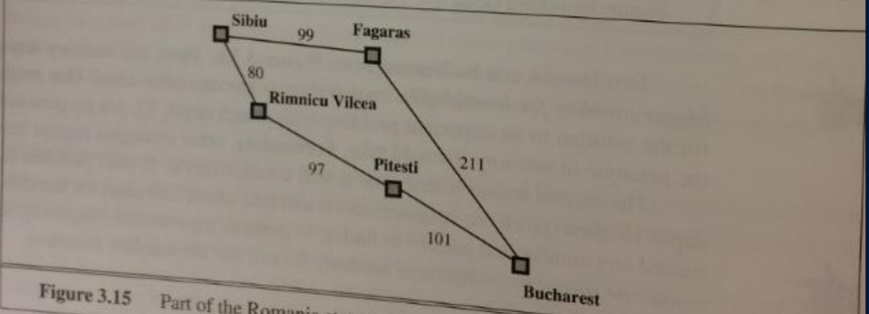


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

may be on a suboptimal path. The second differ
path is found to a node

UCS

- Problem: get from Sibiu to Bucharest
- 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 UCS 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.

UCS

- How does UCS shape up on the “Big 4”?
  - Optimality, completeness, time, space

UCS

- It is easy to see that UCS is optimal in general. First, we observe that whenever UCS selects a node n for expansion, the optimal path to that node has been found. (Were this not the case, there would have to be another frontier node n’ on the optimal path from the start node to b; by definition, n’ would have lower g-cost than n and would have been selected first.) Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that *UCS expands nodes in order of their optimal path cost*. Hence, the first goal node selected for expansion must be the optimal solution.

UCS

- UCS does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions. Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε.

UCS

- UCS is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d. Instead, let 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 O(b^(1+floor(C*/ ε)), which can be much greater than b^d. This is because UCS can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, it is O(b^(d+1)). When all step costs are the same, UCS is similar to BFS, except that the latter stops as soon as it generates a goal, whereas UCS examines all the nodes at the goal's depth to see if one has lower cost; thus, UCS does strictly more work by expanding nodes at depth d unnecessarily.

Depth-first search (DFS)

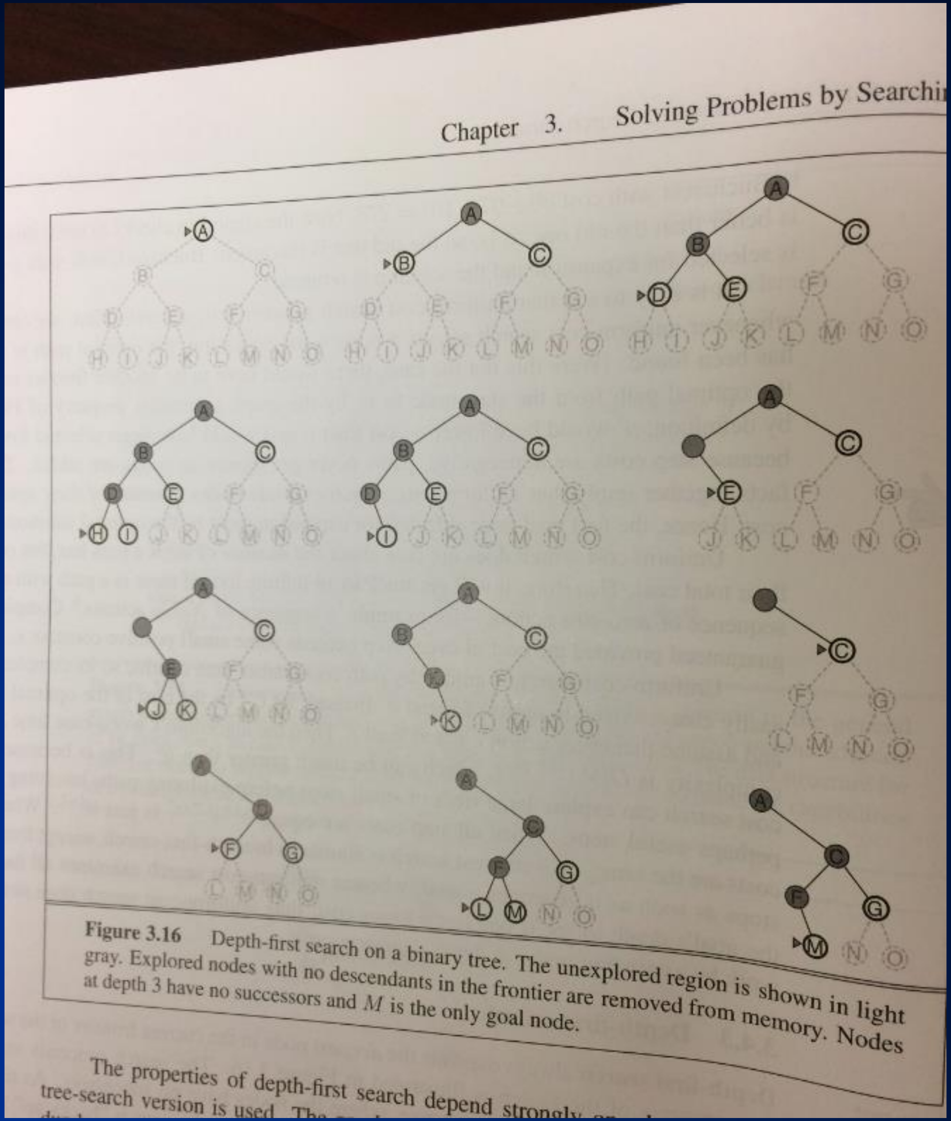


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

The properties of depth-first search depend strongly
tree-search version is used.

Depth-limited search (DLS)

- Imposes a depth limit L on DFS to prevent it from getting “stuck” along hopeless path.

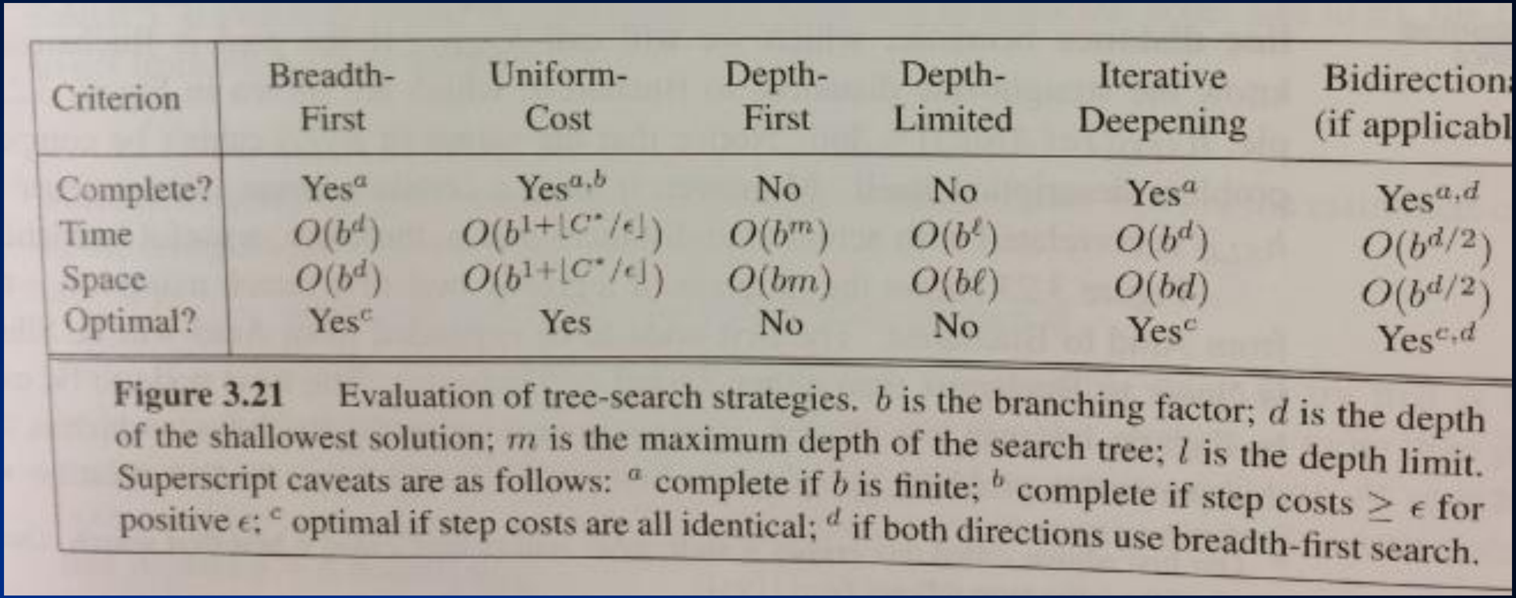
Iterated deepening depth-first search

- Runs DLS with depth limit 0, 1, 2, etc. Ends when depth limit reaches d, depth of the shallowest goal node.

Bidirectional search

- 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.

Comparing uninformed search strategies



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirection (if applicabl
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.

Informed search algorithms

- Greedy best-first search
- A* search
- Memory-bounded heuristic search

- These algorithms have a heuristic function to help guide the search.

Homework for next class

- Chapter 4 from Russell-Norvig textbook.