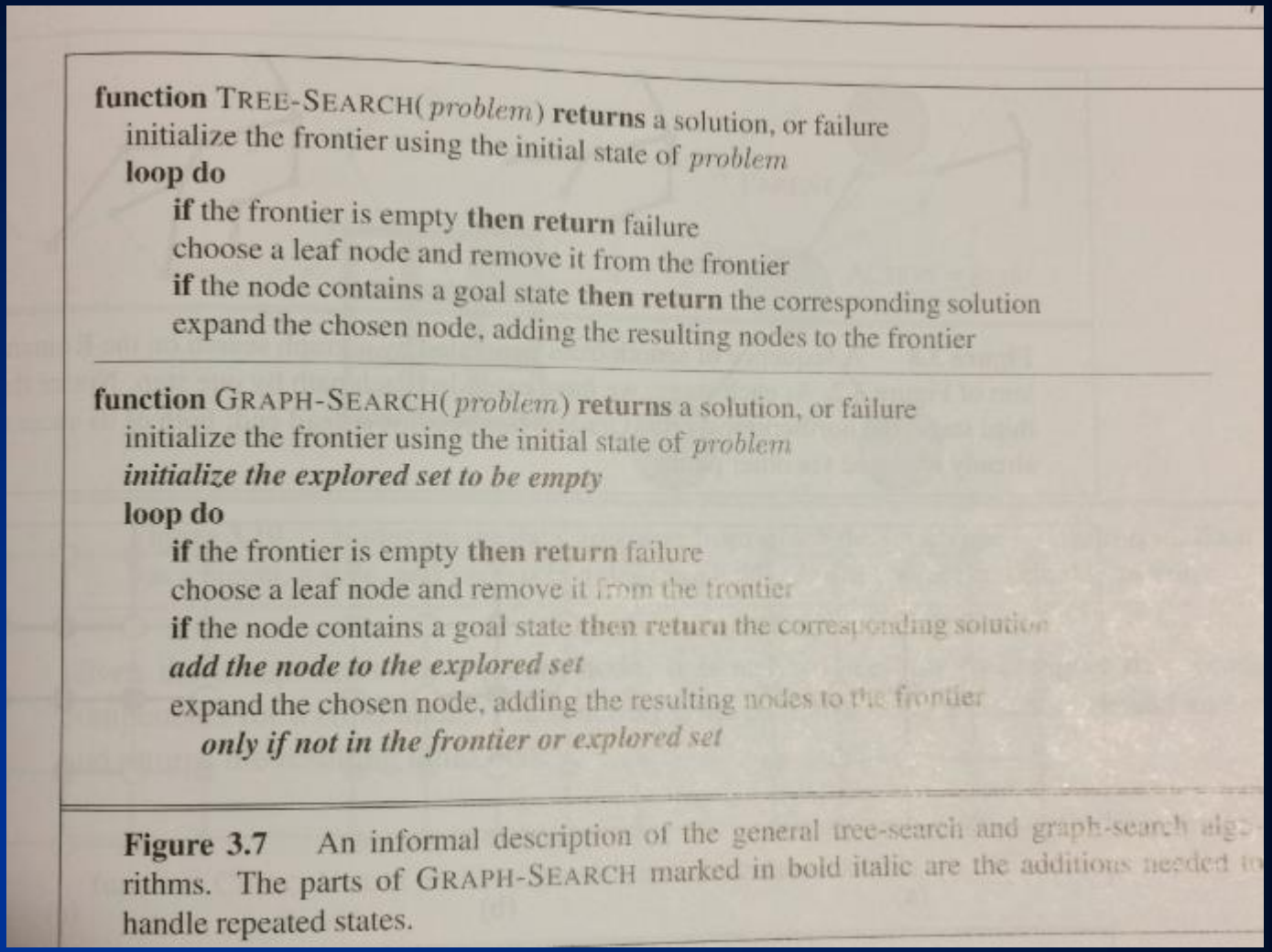
CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 out Tuesday
  - Programming problem in Python and theory problems

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 algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

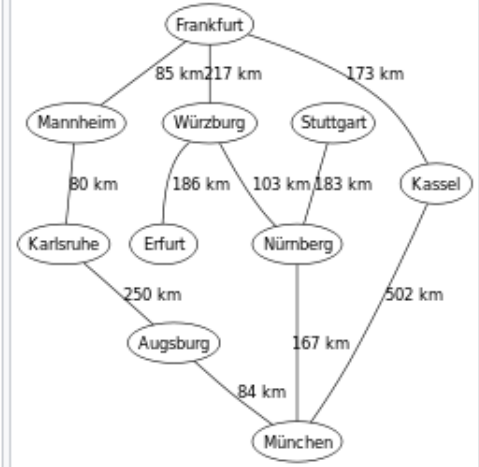
Real-world search

- **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

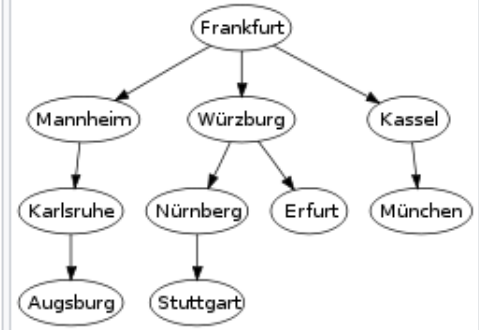
Evaluating performance

- **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?

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)

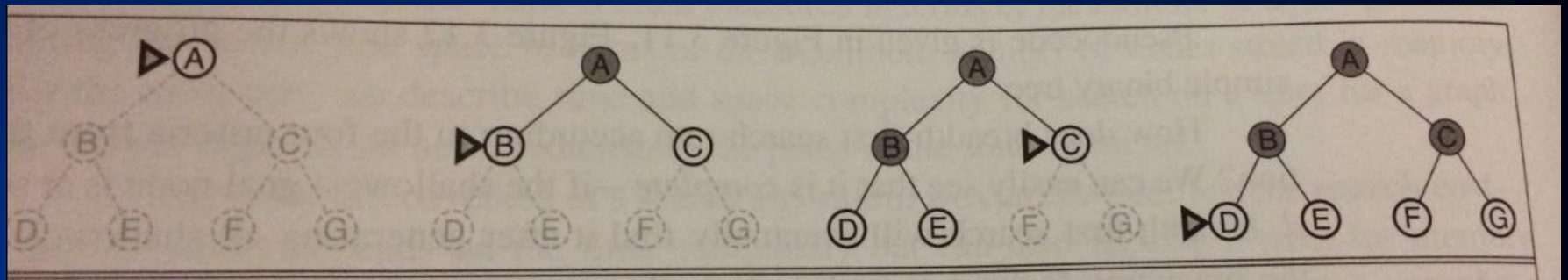


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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.

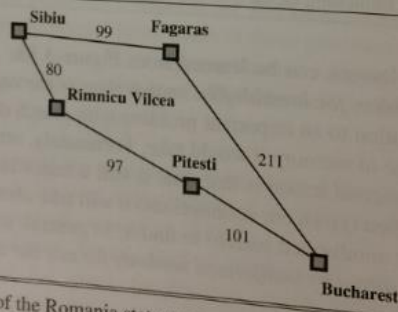


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

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.

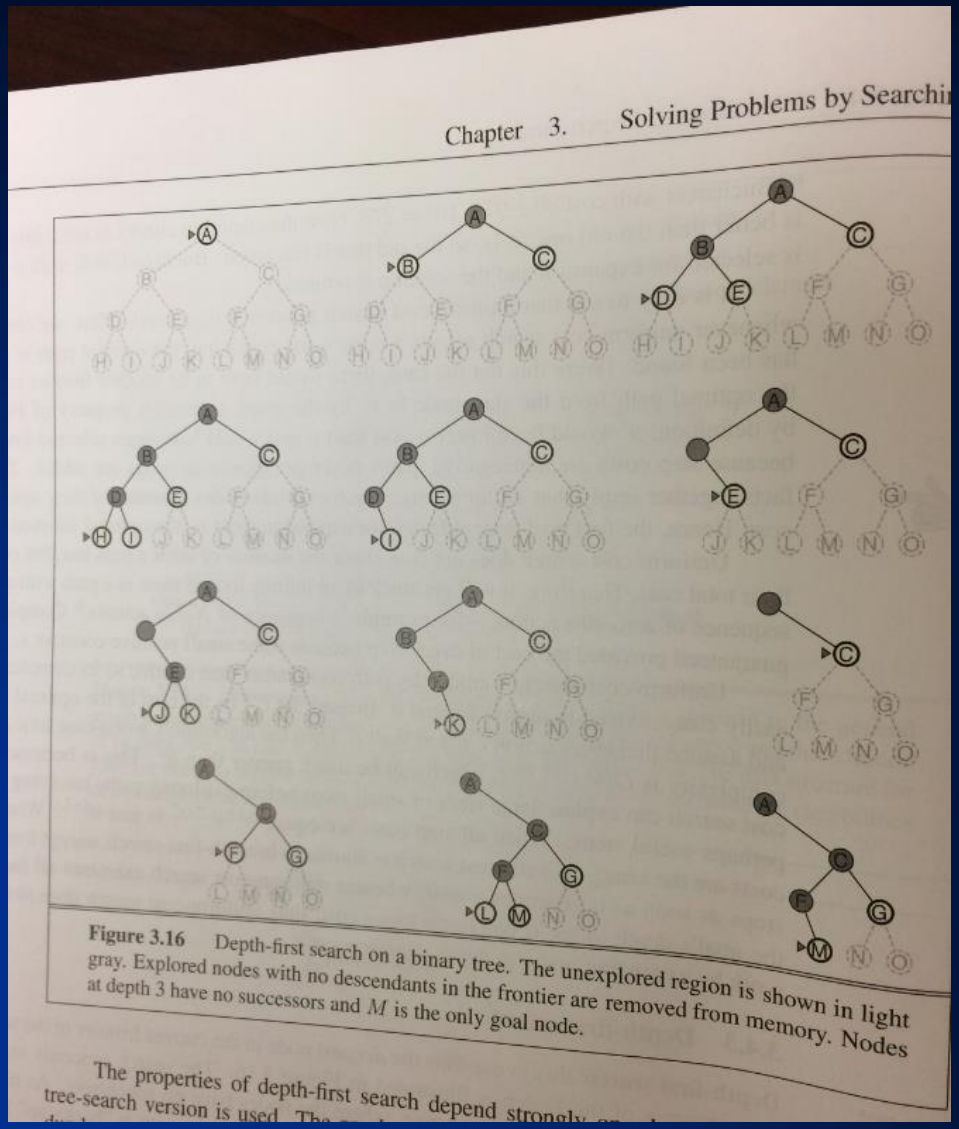
UCS

- 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)



DFS

- BFS: FIFO queue, DFS: LIFO queue
- Commonly implemented recursively

```
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.

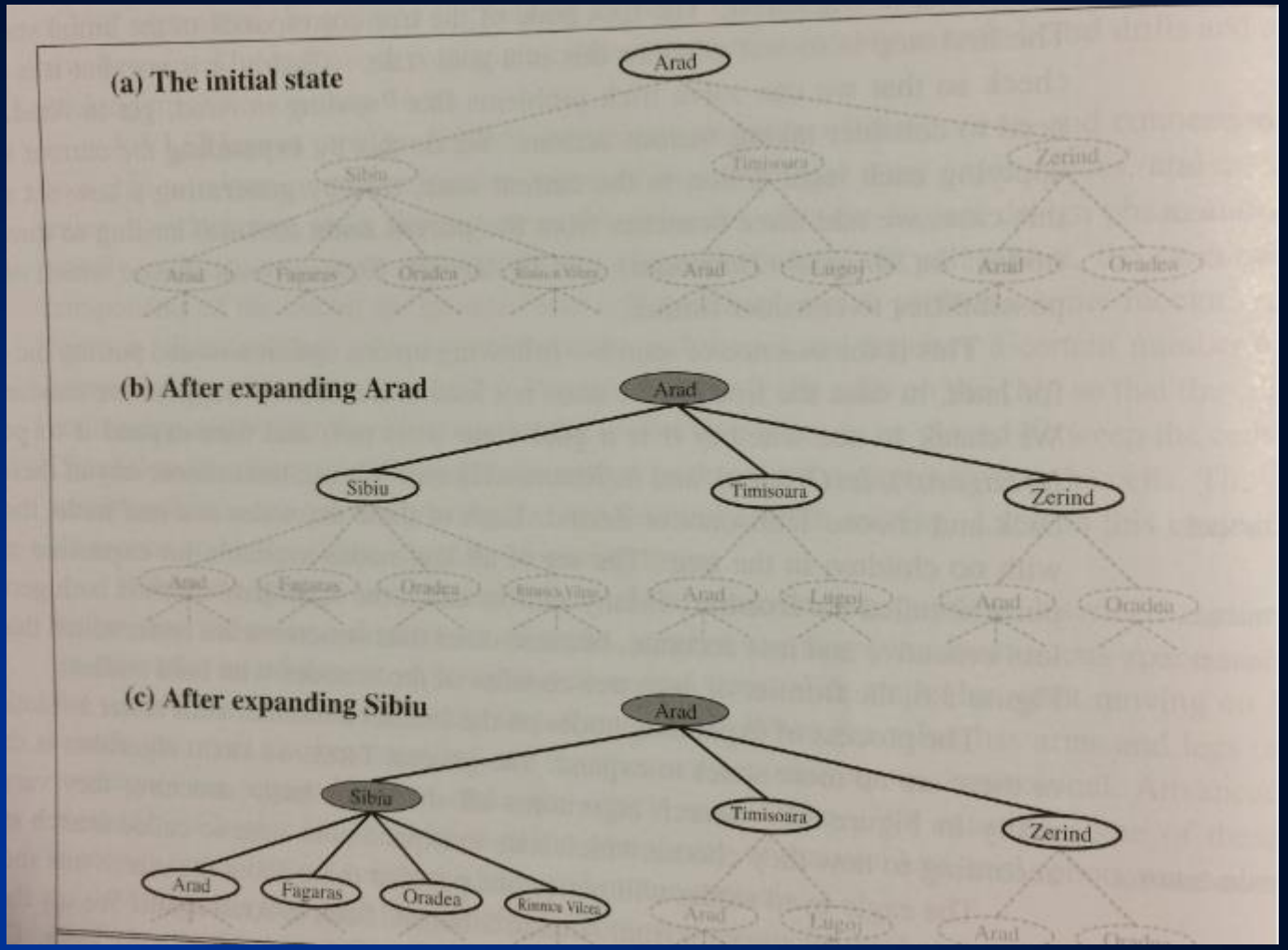
Big 4 for DFS

- Completeness
- Optimality
- Time
- Space

DFS completeness

- The graph-search version, which avoids repeated states and redundant paths, is complete in finite spaces because it will eventually expand every node.
- The tree-search version, on the other hand, is *not* complete—for example, it will follow the Arad-Sibiu-Arad-Sibiu loop forever.

Searching Romania



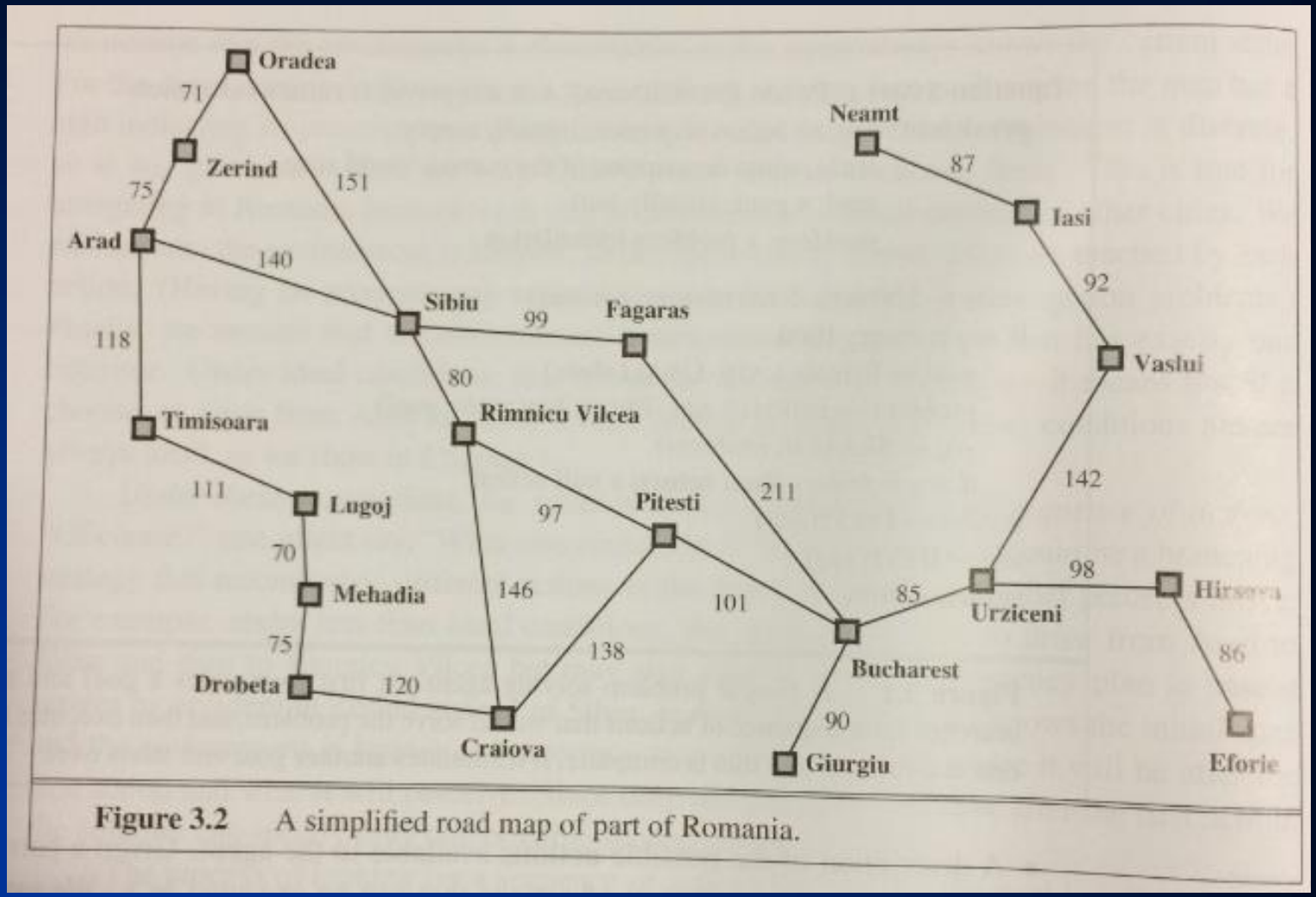
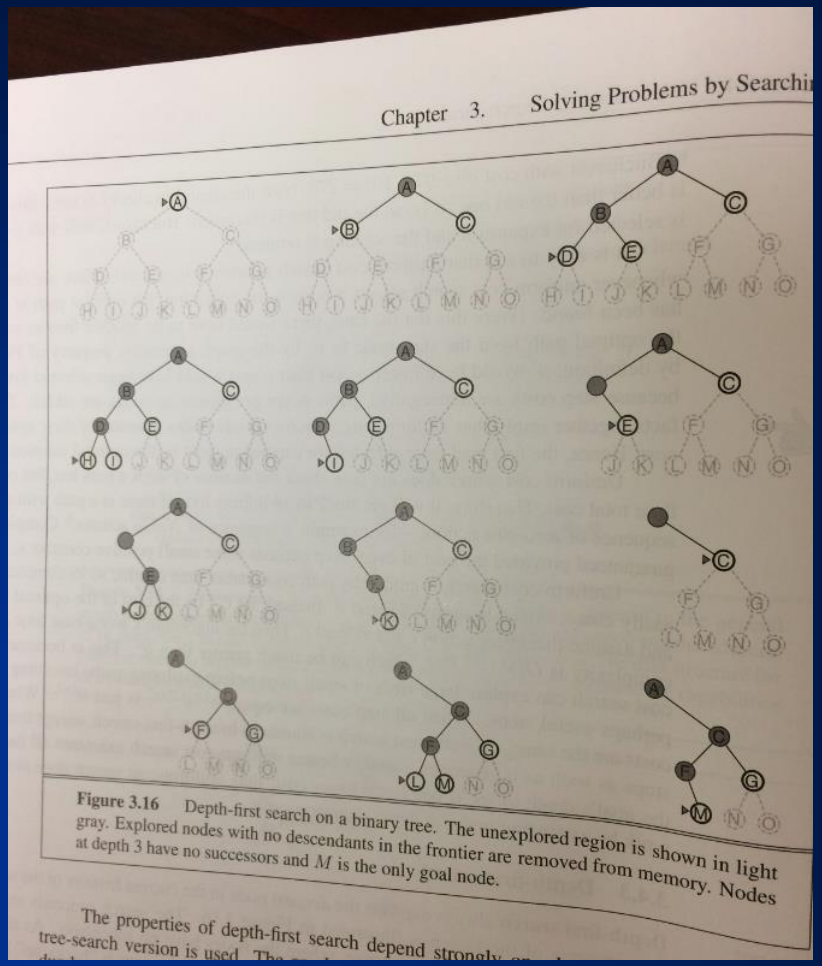


Figure 3.2 A simplified road map of part of Romania.

DFS optimality

- For similar reasons, both versions are nonoptimal.



Chapter 3. Solving Problems by Searchi

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

DFS time complexity

- For graph search the time complexity is bounded by the size of the state space (which may be infinite).
- For tree search, on the other hand, we may generate all of the O(b^m) nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space. Note that m can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

DFS space complexity

- So far, DFS has no clear advantage over BFS!
- What about space complexity? Recall it was O(b^d) for BFS.
- For DFS graph search there is no advantage. But DFS tree search only needs to store 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, DFS requires storage of only O(bm) nodes.
- Eg for d = 10 it requires only 156 kilobytes instead of 10 exabytes, which is a factor of 7 trillion times less space.
- This has led to adoption of DFS as basic workhorse in many areas of AI: e.g., constraint satisfaction, propositional satisfiability, and logic programming.

Depth-limited search (DLS)

- Imposes a depth limit L on DFS to prevent it from getting "stuck" along hopeless path. This solves the "infinite path problem."
- Problem: if we choose L < d, then DLS is incomplete, because the shallowest goal is beyond the depth limit.
- Like DFS, DLS will also be nonoptimal if L > d.
- Time complexity is O(b^L) and space is O(bL).
- DFS can be viewed as special case of DLS with L = ∞.
- In Romania there are 20 cities. What might be value for L?

DLS

- If there is a solution, it must be of length 19 at the longest, so L = 19 is a possible choice.
- But in fact if we studied the map carefully, we could discover that any city can be reached from any other city in at most 9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.

Iterated deepening depth-first search

- Runs DLS with depth limit 0, 1, 2, etc. to find the best depth limit. Ends when depth limit reaches d, depth of the shallowest goal node.
- IDS combines the benefits of DFS and BFS.
  - Like DFS, its memory requirements are modest: O(bd)
  - Like BFS, 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.
- Figure on next page shows 4 iterations of IDS. Black circles indicate nodes that have been removed from memory.

IDS

```
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.

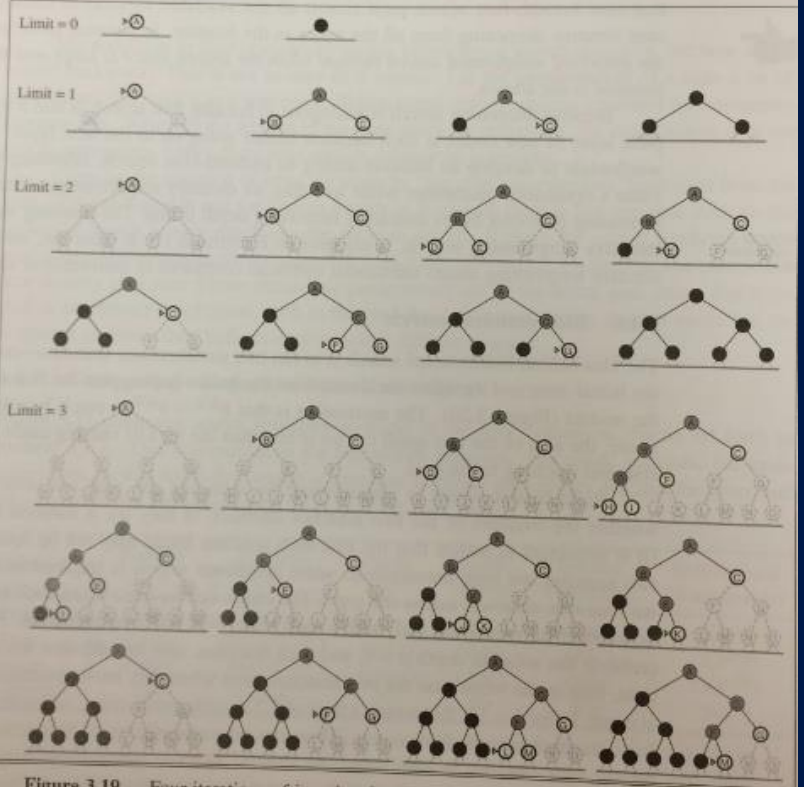


Figure 3.19 Four iterations of iterative deepening search on a binary tree. Black circles indicate nodes that have been removed from memory.

IDS

- IDS may seem wasteful because states are generated multiple times. It turns out this is not too costly; most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.
- N(IDS) = (d)b + (d-1)b^2 + … + (1)b^d
- Same asymptotic time complexity O(b^d) as BFS
- If b = 10 and d = 5:
  - N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,000

IDS

- *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

- 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. The idea is that b^(d/2) + b^(d/2) is much less than b^d (the area of the two small circles is less than the area of one big circle centered on the start and reaching the goal).

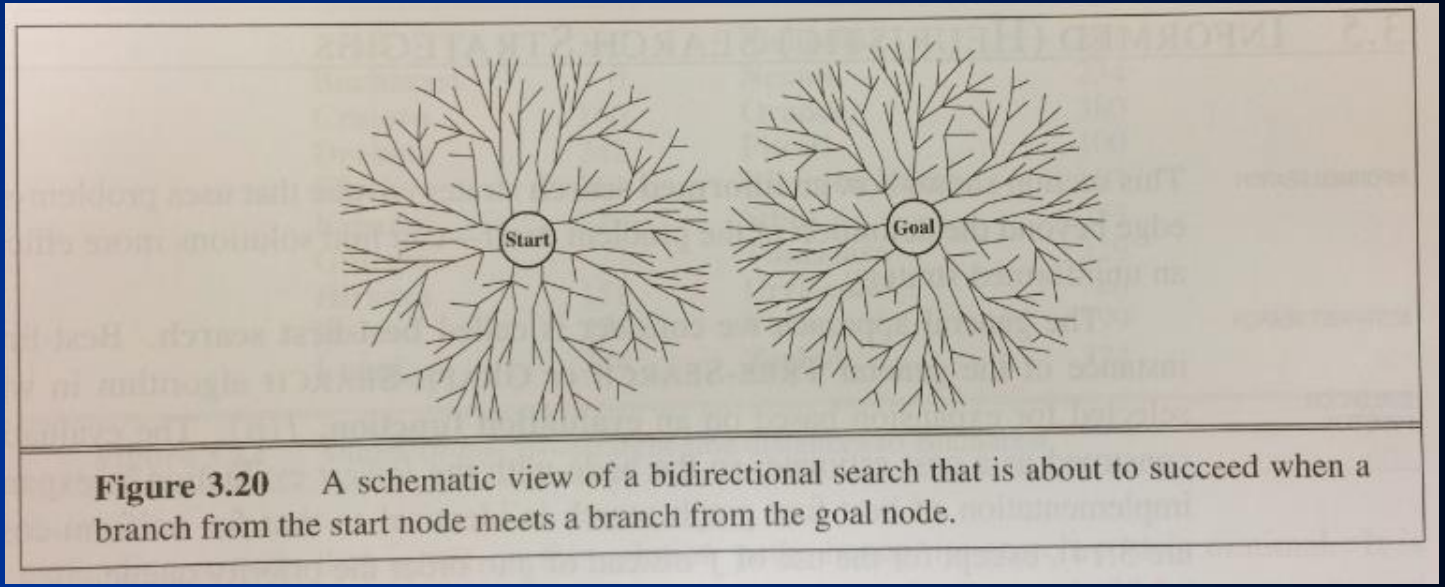


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Bidirectional search

- BDS 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 check can be done when search node is generated or selected for expansion and, with a hash table, will take constant time.
- For example, if a problem has solution depth d=6, and each direction runs BFS one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. For b = 10, this means a total of 2,220 node generations, compared to 1,111,110 for standard BFS. Thus time complexity is O(d^(b/2)). But space complexity is also O(d^(b/2)).

BDS

- Not always that easy to “search backwards.” Let the **predecessors** of a state x be all those states that have x as a successor. BDS requires a method for computing predecessors. When all actions in the state space are reversible, the predecessors of x are just its successors. Other cases may require substantial ingenuity.

Comparing uninformed search strategies

- Comparison is for tree-search versions. For graph searches, the main differences are that DFS is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.

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.

Informed (heuristic) search strategies

- General approach we consider is called **best-first search**.
- For the uninformed graph and tree search, the order in which a node is selected for evaluation was based on being first or last in.
- For informed best-first search, we assume we have an **evaluation function** f(n). This can be construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first search is the same as for UCS, except for the use of f instead of g to order the priority queue.

Best-first search

- The choice of f determines the search strategy.
  - Exercise 3.21 shows that best-first tree search includes DFS as a special case.
- Most best-first search algorithms include as a component of f a **heuristic function** denoted h(n):
  - h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

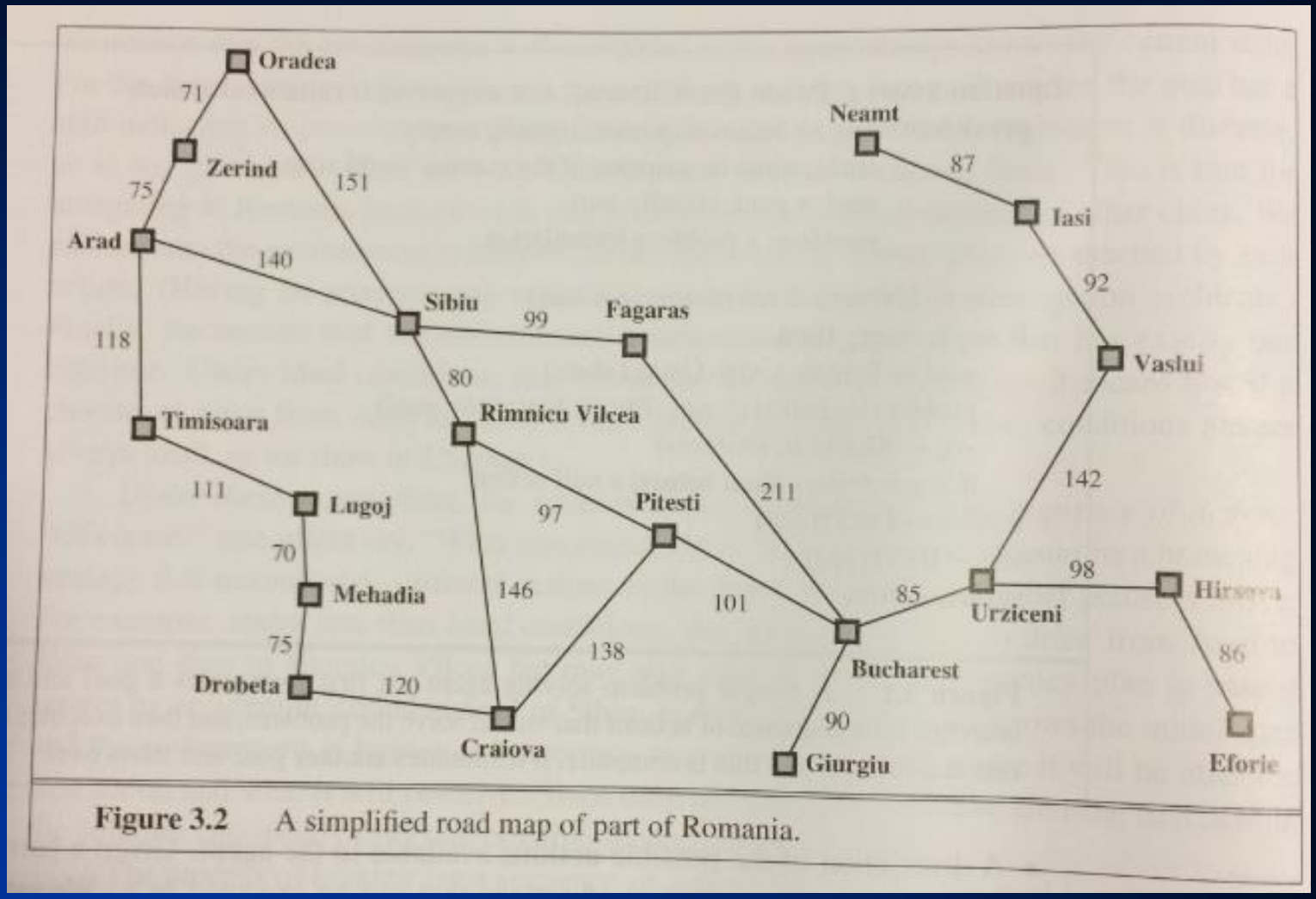


Figure 3.2 A simplified road map of part of Romania.

Best-first search

- What are some possible heuristic functions for the Romania map problem?

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

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Best-first search

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n) = 0.

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).
- Romania using SLD with goal Bucharest.
- E.g., H_SLD(In(Arad)) = 366.
- Search cost is minimal, but it is not optimal; the path via Sibiu and Faragas to Bucharest is 32 km longer than the path through RV and Pitesti. This shows why the algorithm is “greedy” – at each step it tries to get as close to the goal as it can.

Greedy best-first search

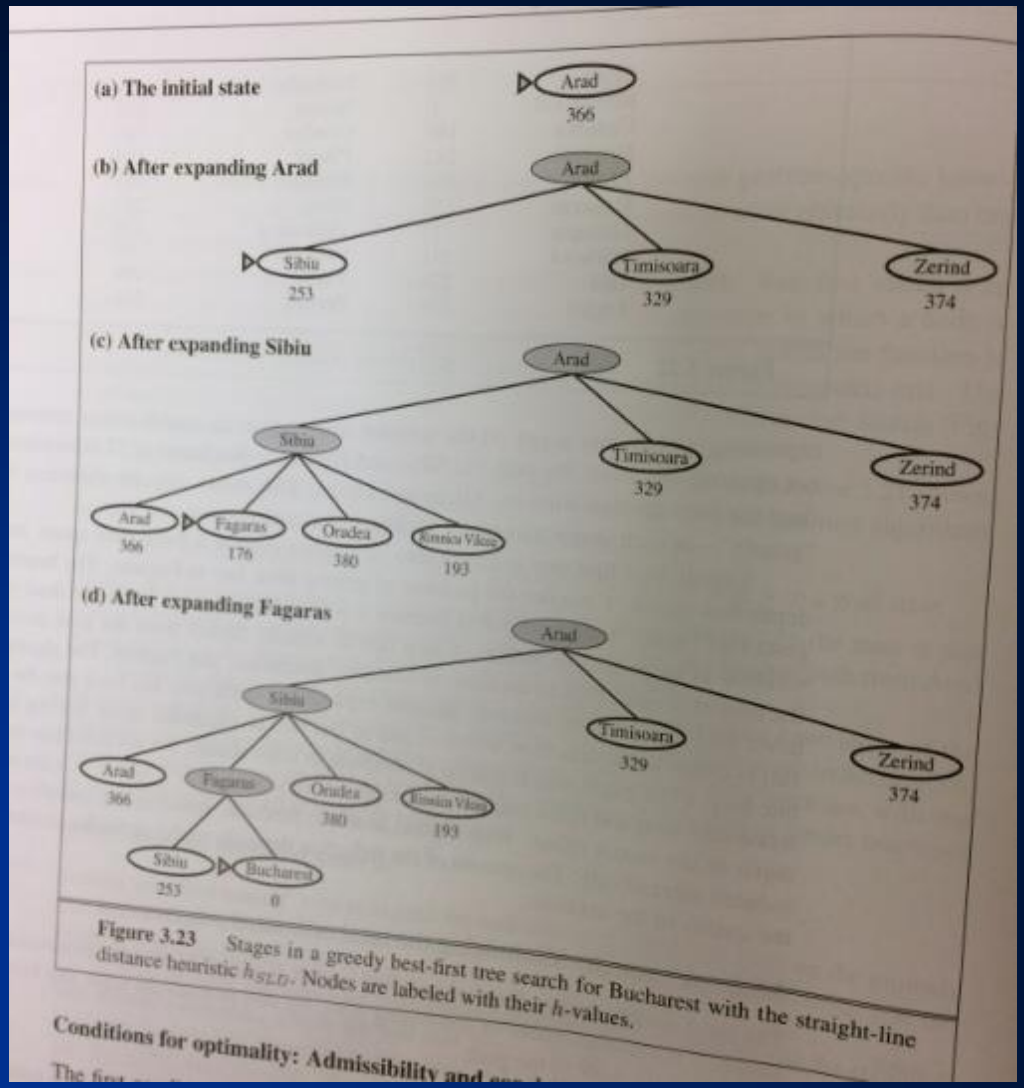


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

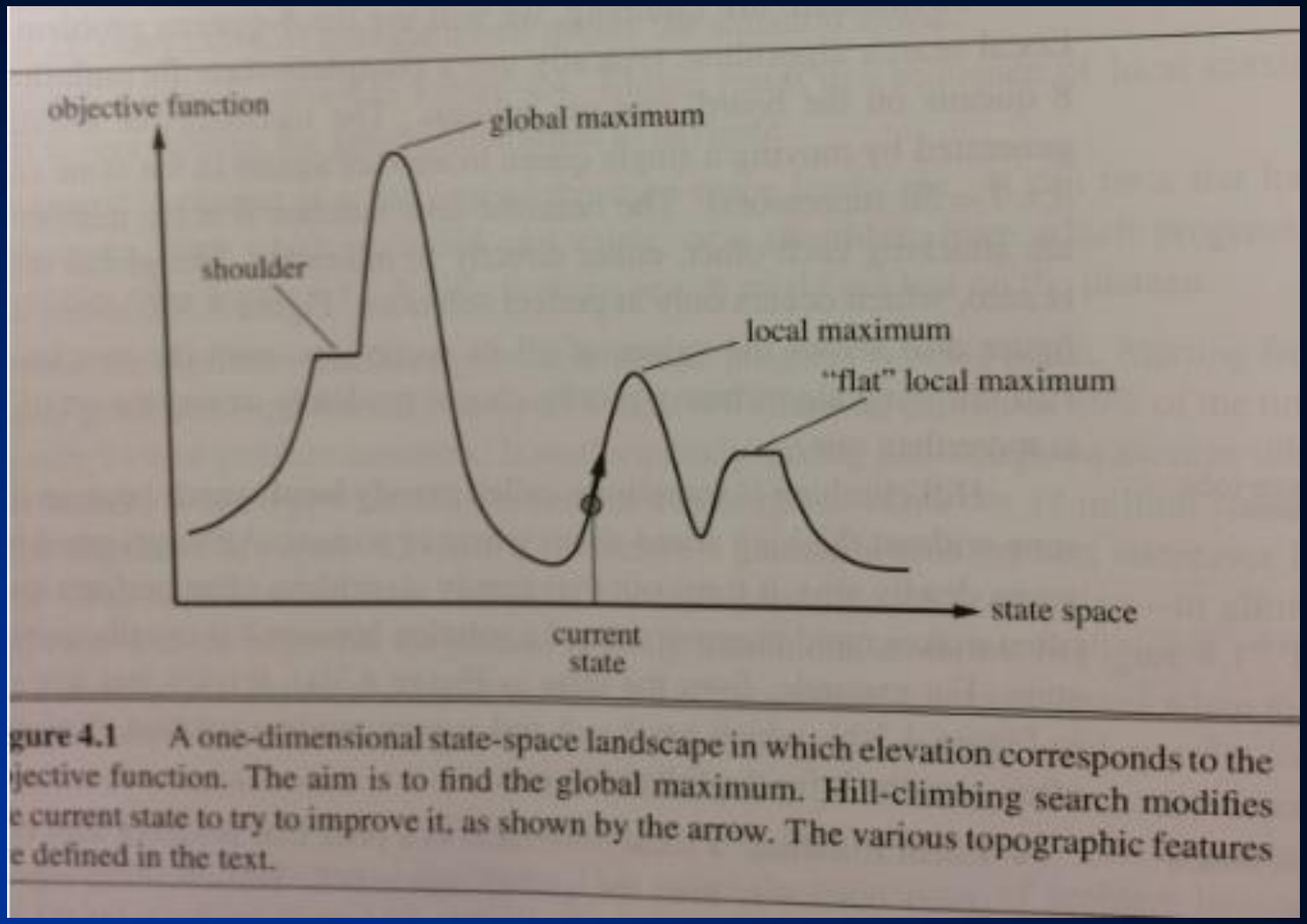
Conditions for optimality: Admissibility and

Informed search algorithms

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

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

Local search



gure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the jective function. The aim is to find the global maximum. Hill-climbing search modifies e current state to try to improve it, as shown by the arrow. The various topographic features e defined in the text.

Homework for next class

- Chapter 5 from Russell-Norvig textbook.
- HW1: out on Tuesday