Module 2

- Problem‑solving: Informed Search Strategies, Heuristic functions

INFORMED (HEURISTIC) SEARCH STRATEGIES

- informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself.

—can find solutions more efficiently than can an uninformed strategy.

- The general approach we consider is called best-first search.

Best-first search

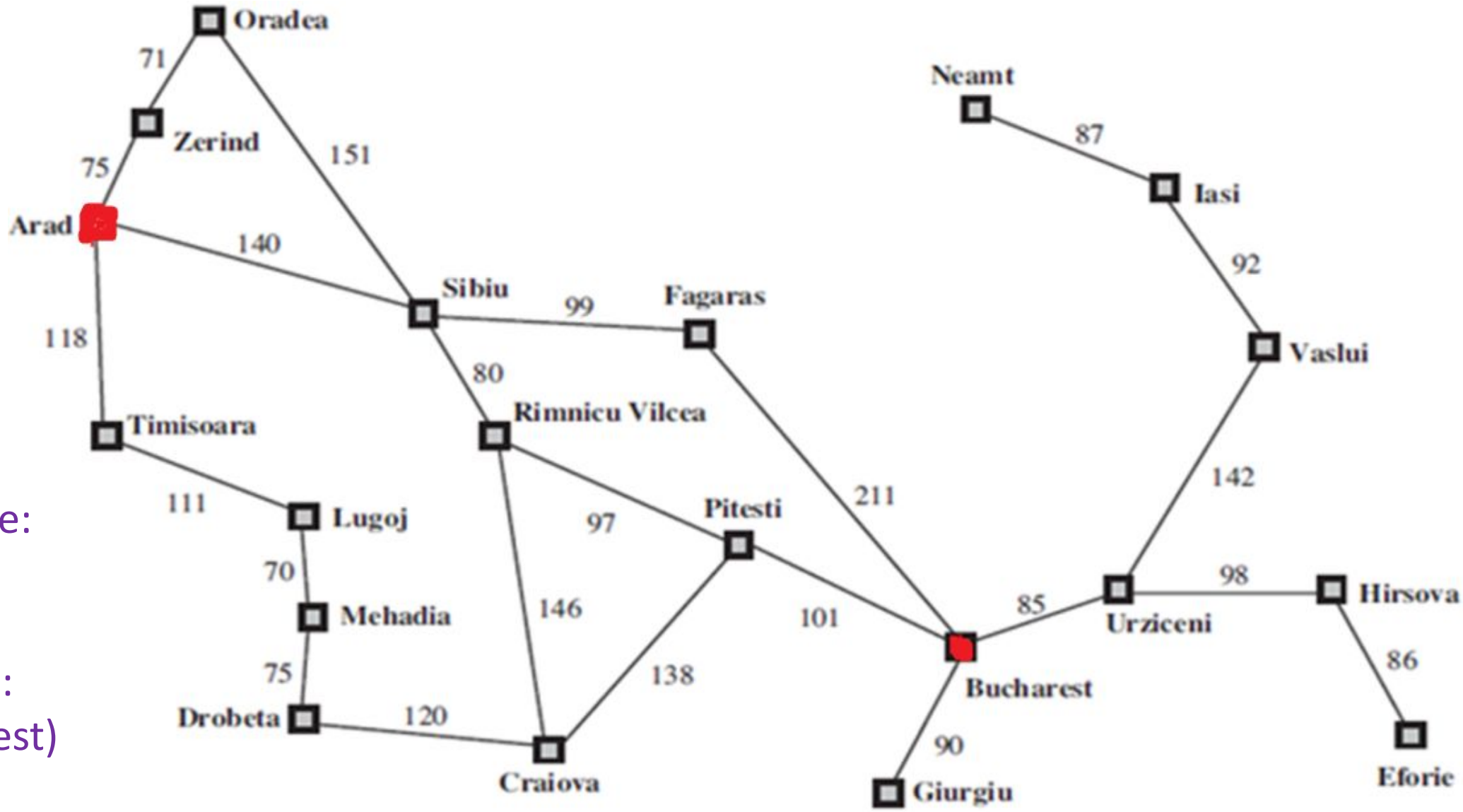
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f(n).
- The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue.

- The choice of f determines the search strategy.
- Most best-first 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.
- if n is a goal node, then h(n)=0.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

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).

- Let us see how this works for route-finding problems in Romania; we use the straight line distance heuristic, which we will call h_{SLD} .
- If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure.
- For example, h_{SLD} (In(Arad)) = 366.
- Notice that the values of h_{SLD} cannot be computed from the problem description itself.
- Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

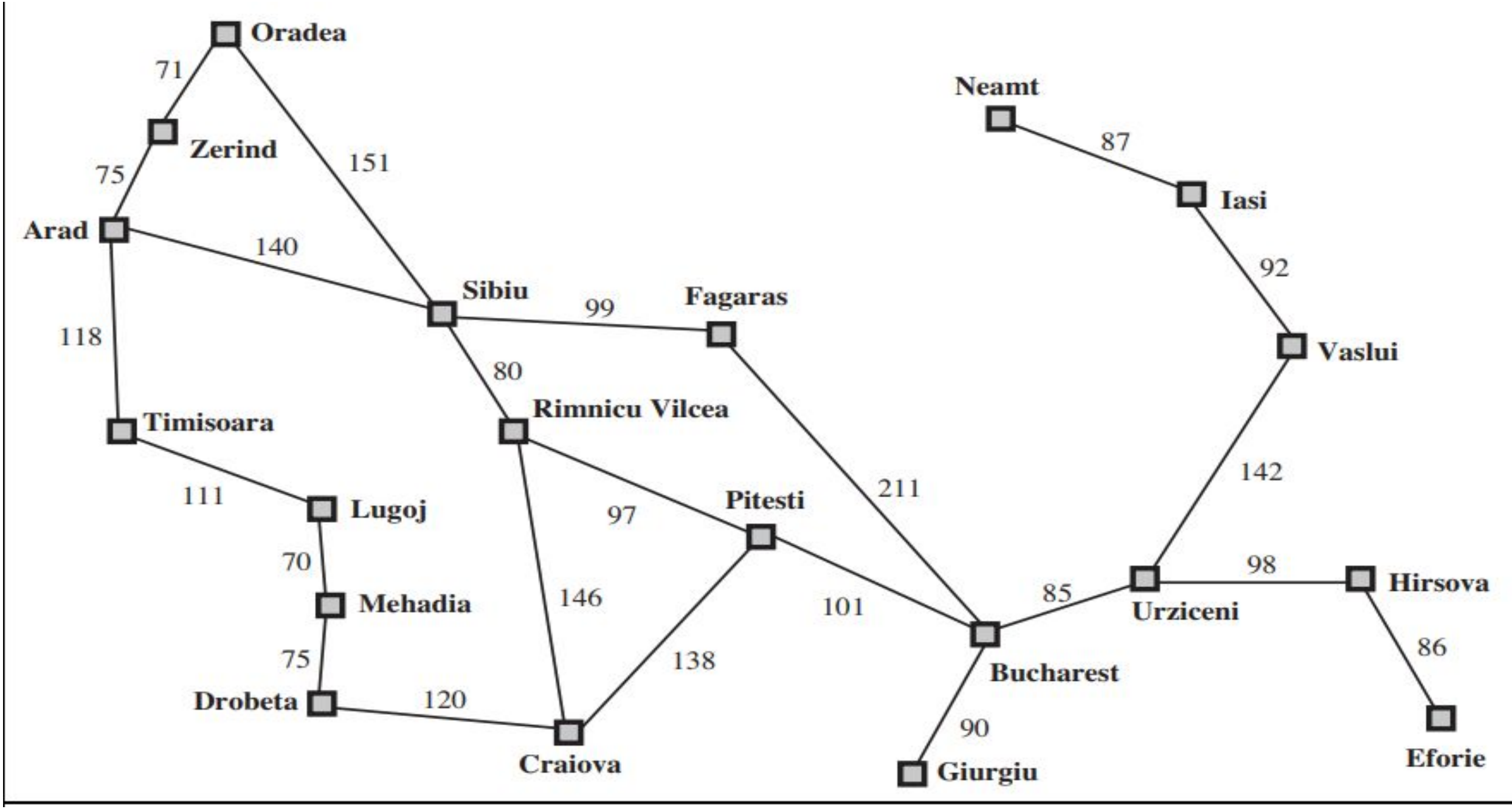


Initial State:
In(Arad)

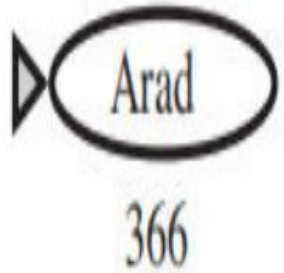
Goal State:
In(Bucharest)

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.

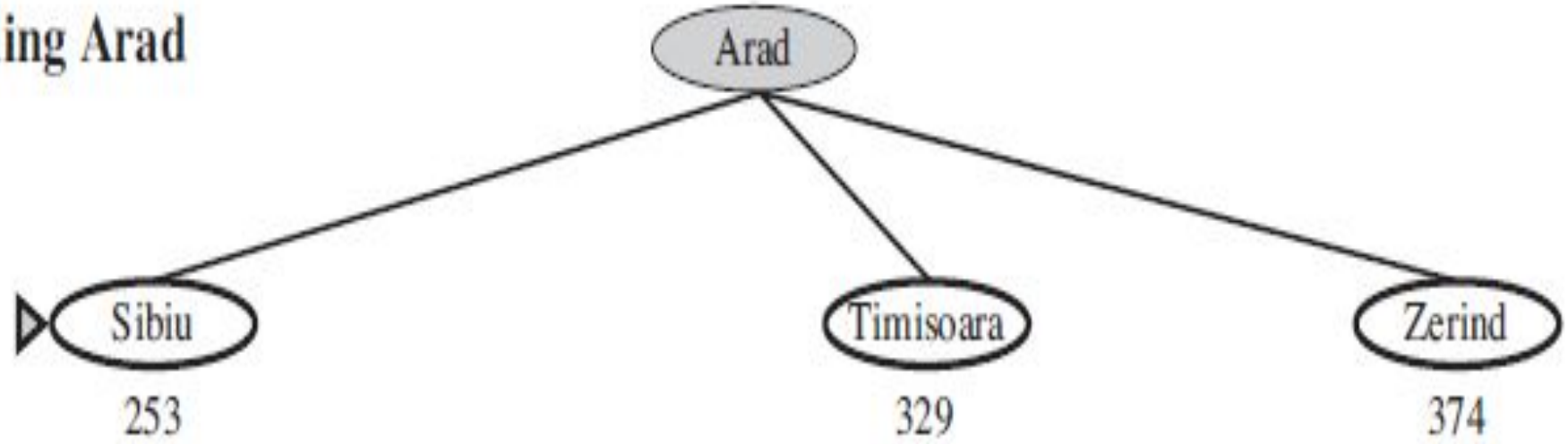

Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Pitesti
211
Lugoj
97
98
70
Hirsova
Mehadia
146
85
Urziceni
101
75
86
Bucharest
Drobeta
120
138
90
Craiova
Eforie
Giurgiu

(a) The initial state

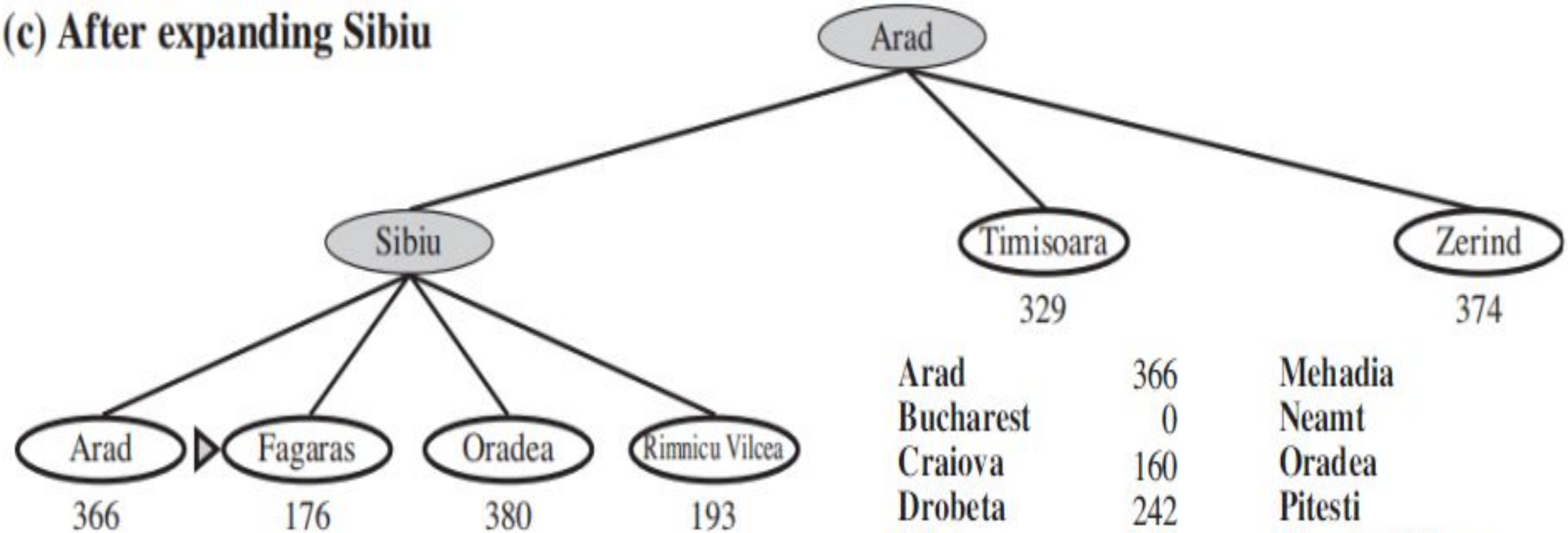


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

(b) After expanding Arad



(c) After expanding Sibiu



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

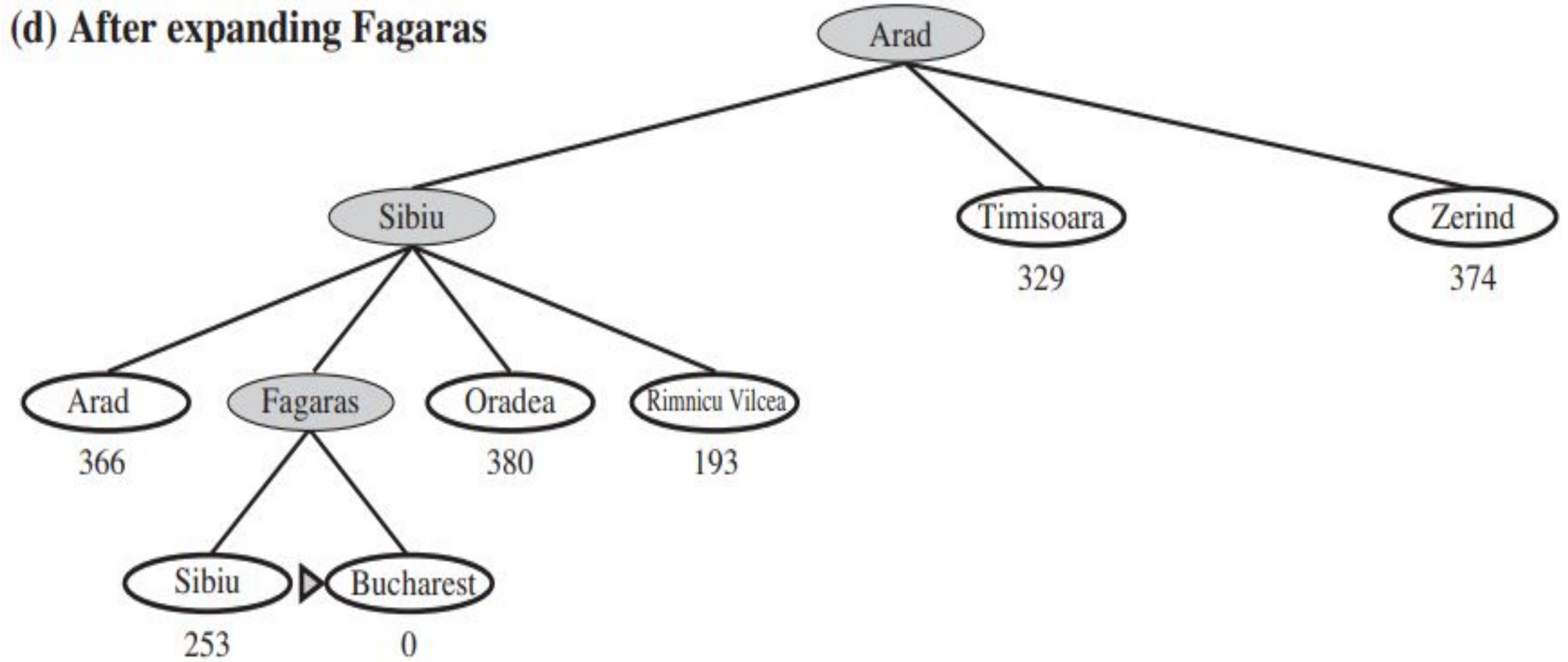
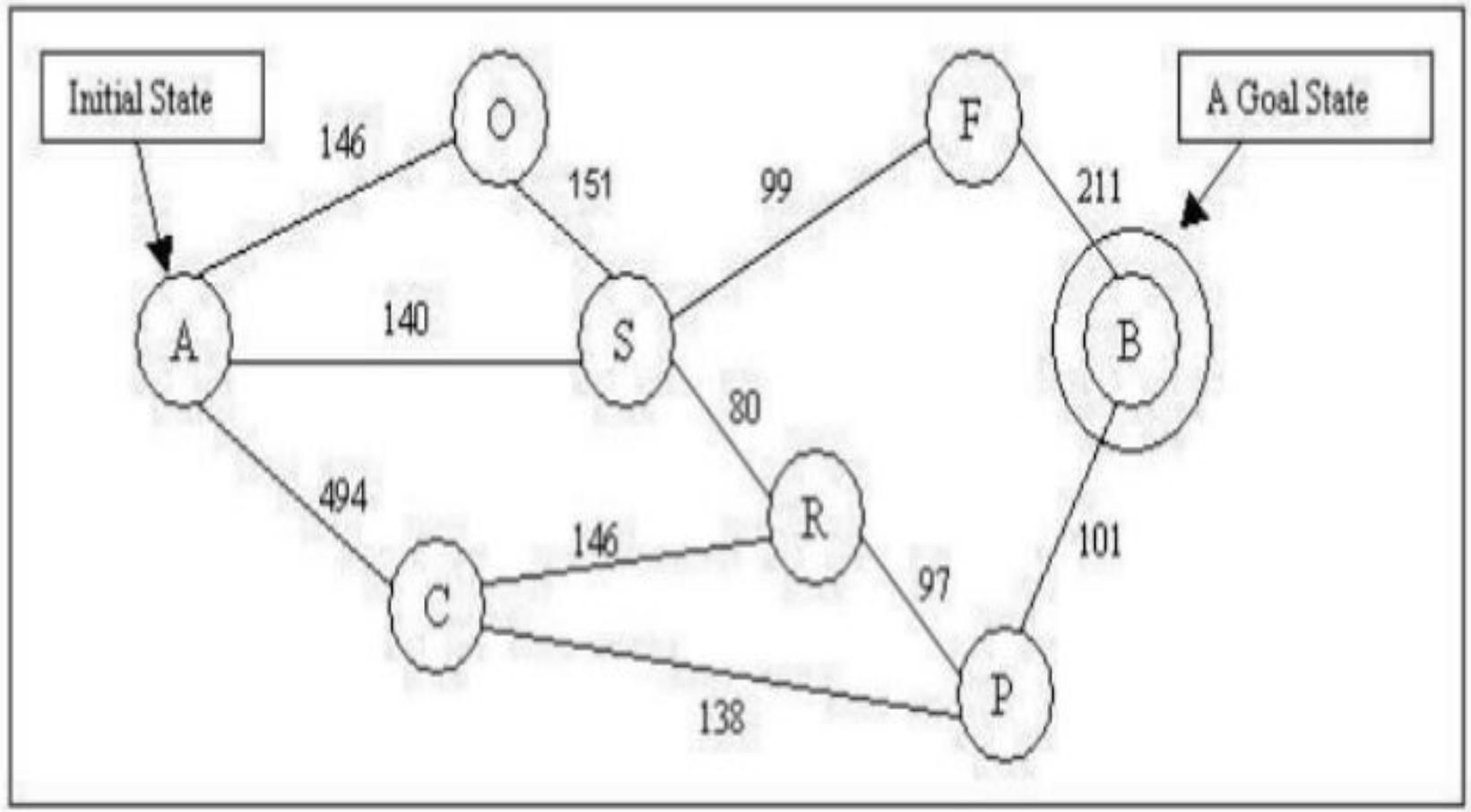


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

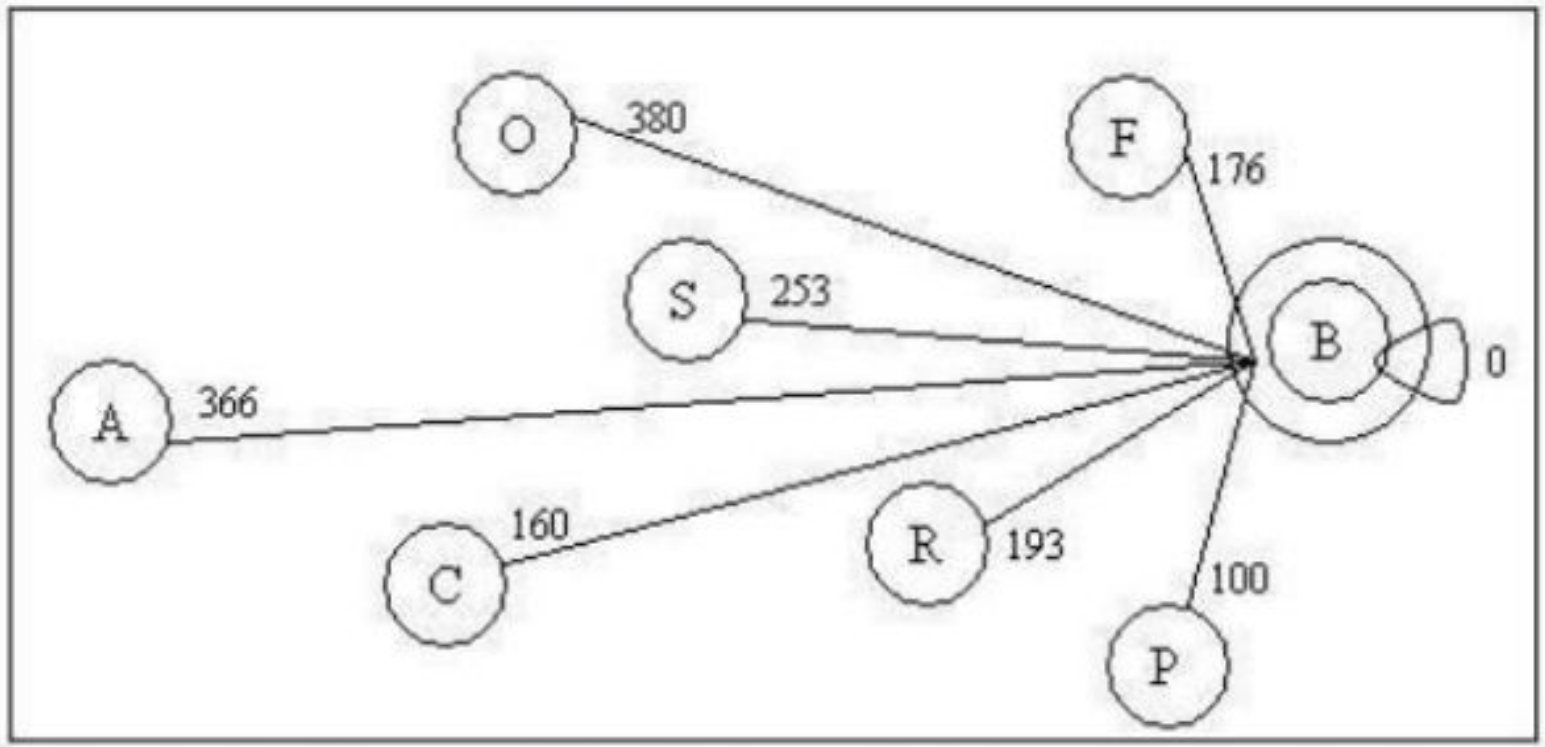
- greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.
- It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
- This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.
- Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search.

- The worst-case time and space complexity for the tree version is O(b^m), where m is the maximum depth of the search space.
- With a good heuristic function, however, the complexity can be reduced substantially.
- The amount of the reduction depends on the particular problem and on the quality of the heuristic.

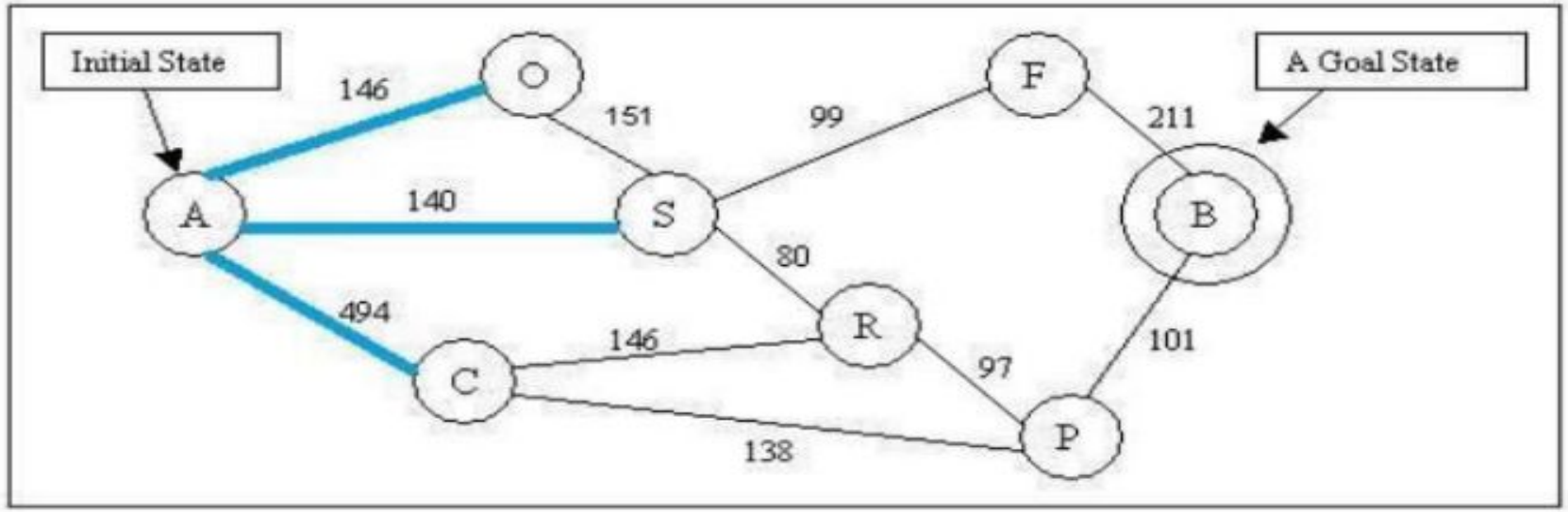
Now let's use an example to see how greedy best-first search works Below is a map that we are going to search the path on. For this example, let the valuation function f(n) simply be equal to the heuristic function h(n).

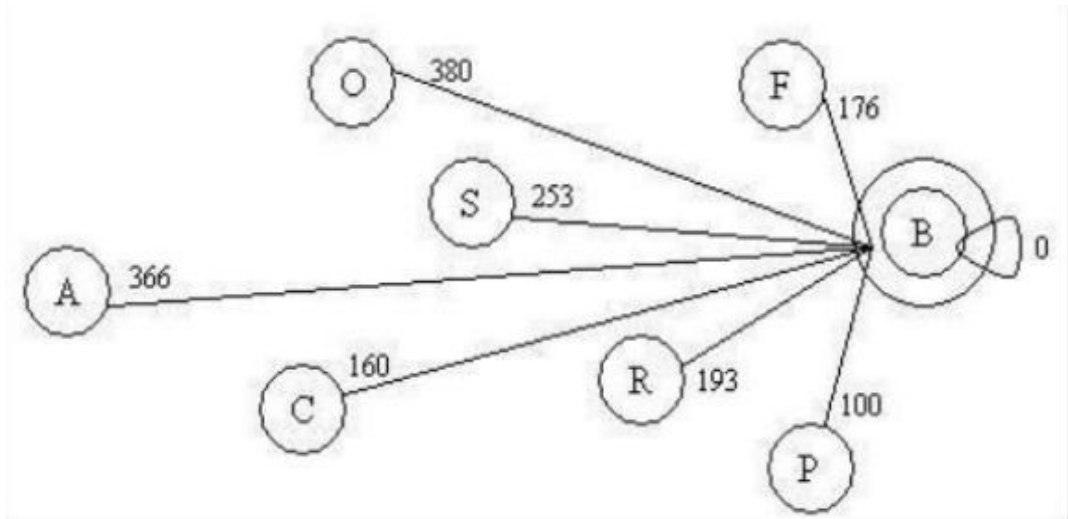


This is a map of a city. We are going to find out a path from city A to city B.

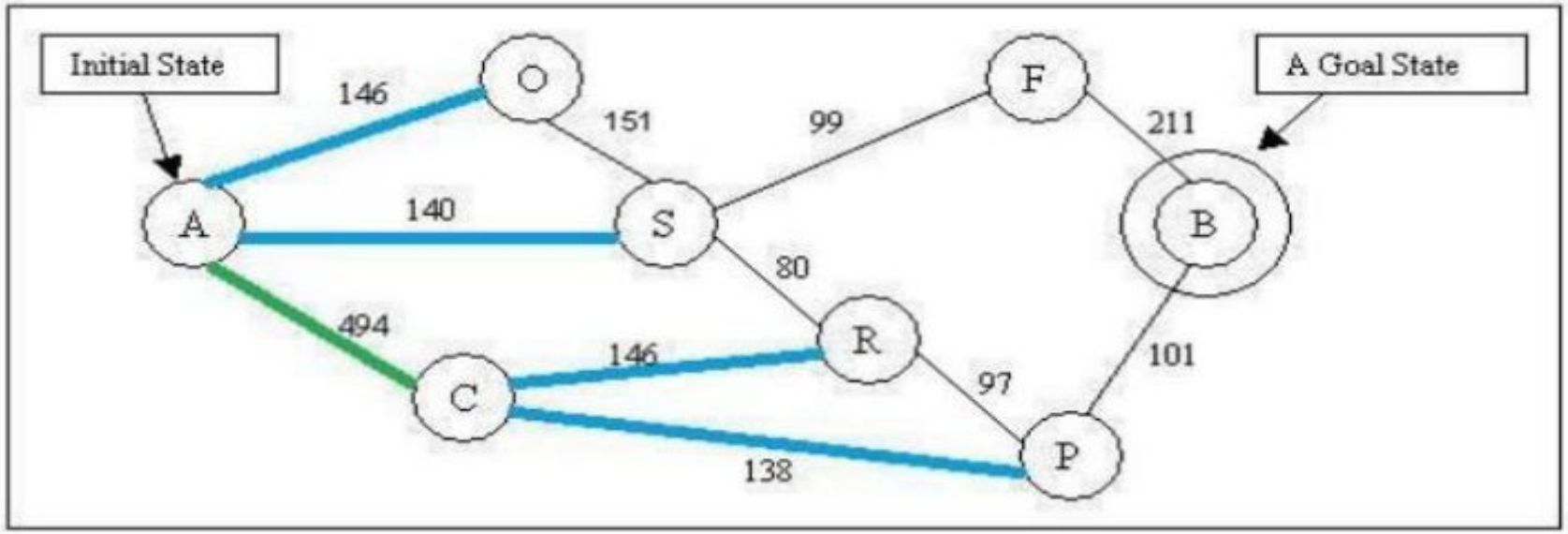


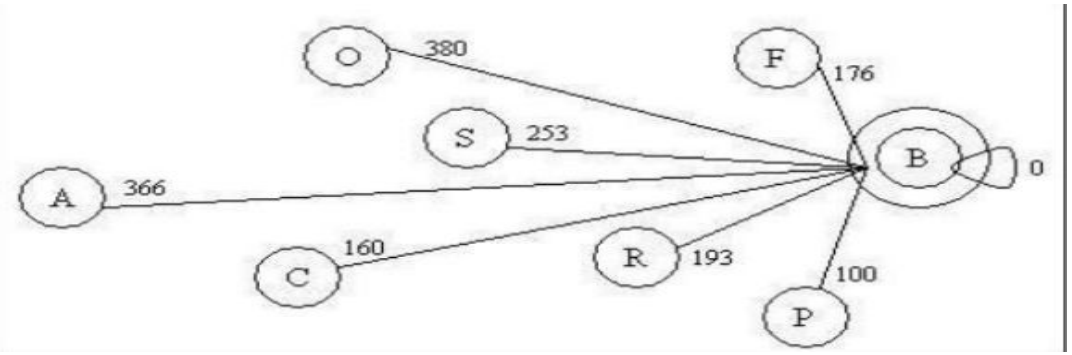
This graph shows the straight distance from each city to city B. The straight distances serve as each cities' h(n) in this example. Because $f(n) = h(n)$ our algorithm will at every stage choose to explore the node that we know is closest to our goal. Let's assume cities on the map as nodes and path between cities as edges. If you start exploring at node A, we have node O, S and C reachable now.



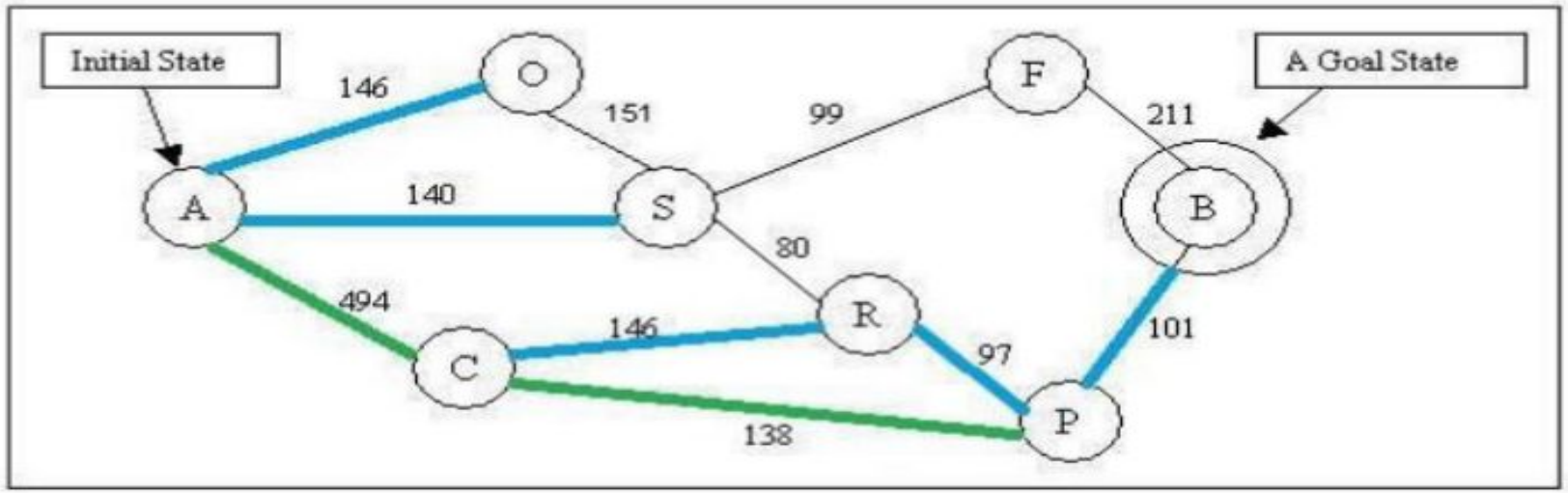


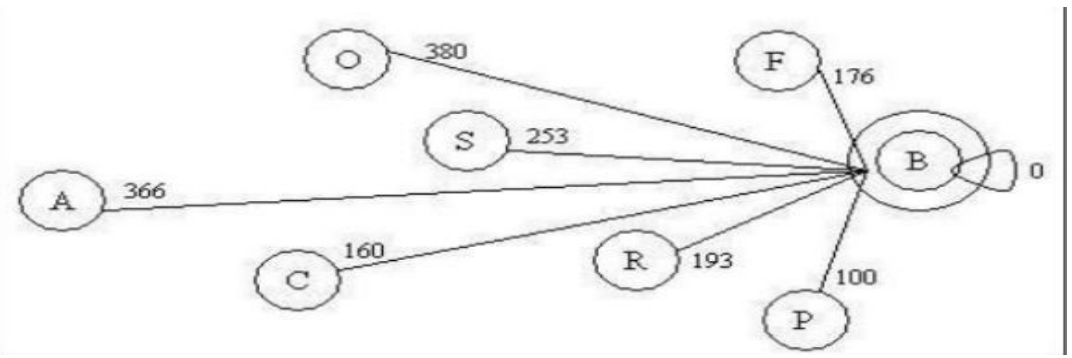
According to the estimate function $f(n) = h(n)$, and given by the graph of h(n) values above: Node O's f(n) = 380, Node S' f(n) = 253, Node C's f(n) = 160. The algorithm will choose to explore node C. After explored C, we have new node R and P reachable now.



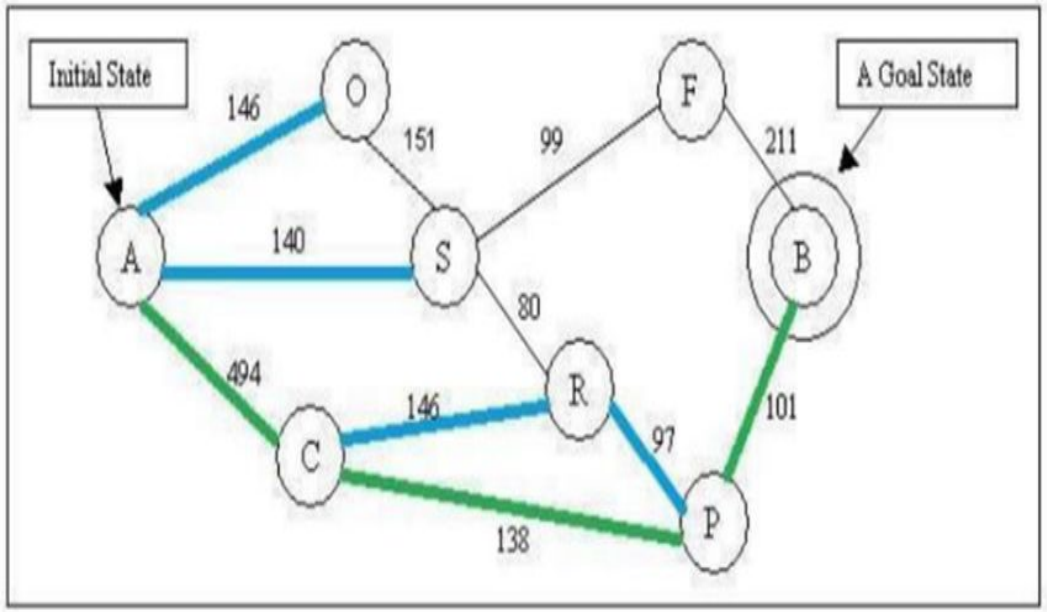


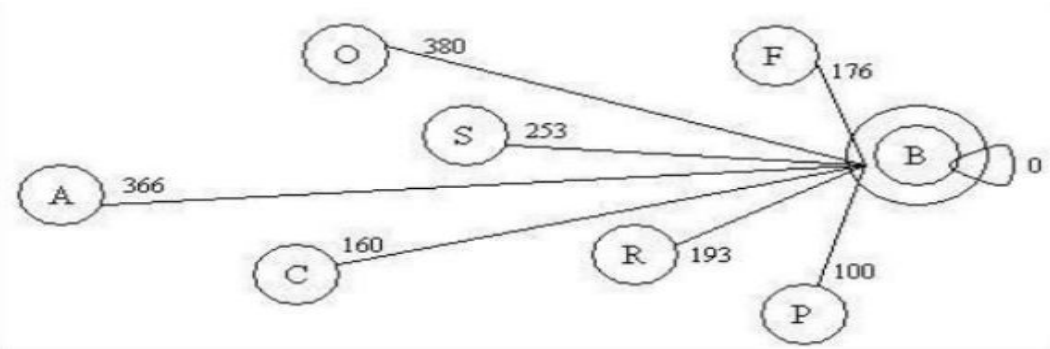
According to the estimate function f(n): Node O's f(n) = 380, Node S' f(n) = 253, Node R's f(n) = 193, Node P's f(n) = 100. The algorithm will choose node P to explore. After explored C, we have new node B reachable now.





According to the estimate function f(n): Node O's f(n) = 380, Node S' f(n) = 253, Node R's f(n) = 193, Node B's f(n) = 0. The algorithm will choose node B to explore. After explored B, we have reached our goal state. The algorithm will be stopped and the path is found.





Note that in this example, the distance between the current node and next node does NOT inform our next step, only the heuristic given by the distance between potential nodes to the goal.

One can generalize the evaluation function of a target node to be a weighted sum of the heuristic function and the distance from the current node to that target, which could produce a different result.

A* search: Minimizing the total estimated solution cost

- The most widely known form of best-first search is called A*.
- It evaluates nodes by combining g(n), the cost to reach the node (Start node to goal state), and h(n), the cost to get from the node to the goal:

$f(n) = g(n) + h(n)$.

- Since g(n) gives the path cost from the start node to node n (actual cost), and h(n) is the estimated cost of the cheapest path from n to the goal, we have

f(n) = estimated cost of the cheapest solution through n .

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n).

- It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete and optimal.

- The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses g + h instead of g.

Conditions for optimality: Admissibility and consistency

- The first condition we require for optimality is that h(n) be an admissible heuristic.
- An admissible heuristic is one that never overestimates the cost to reach the goal.
- Because g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.

- An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest.
- Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

- A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search.
- 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' :

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

Progress of an A* tree search

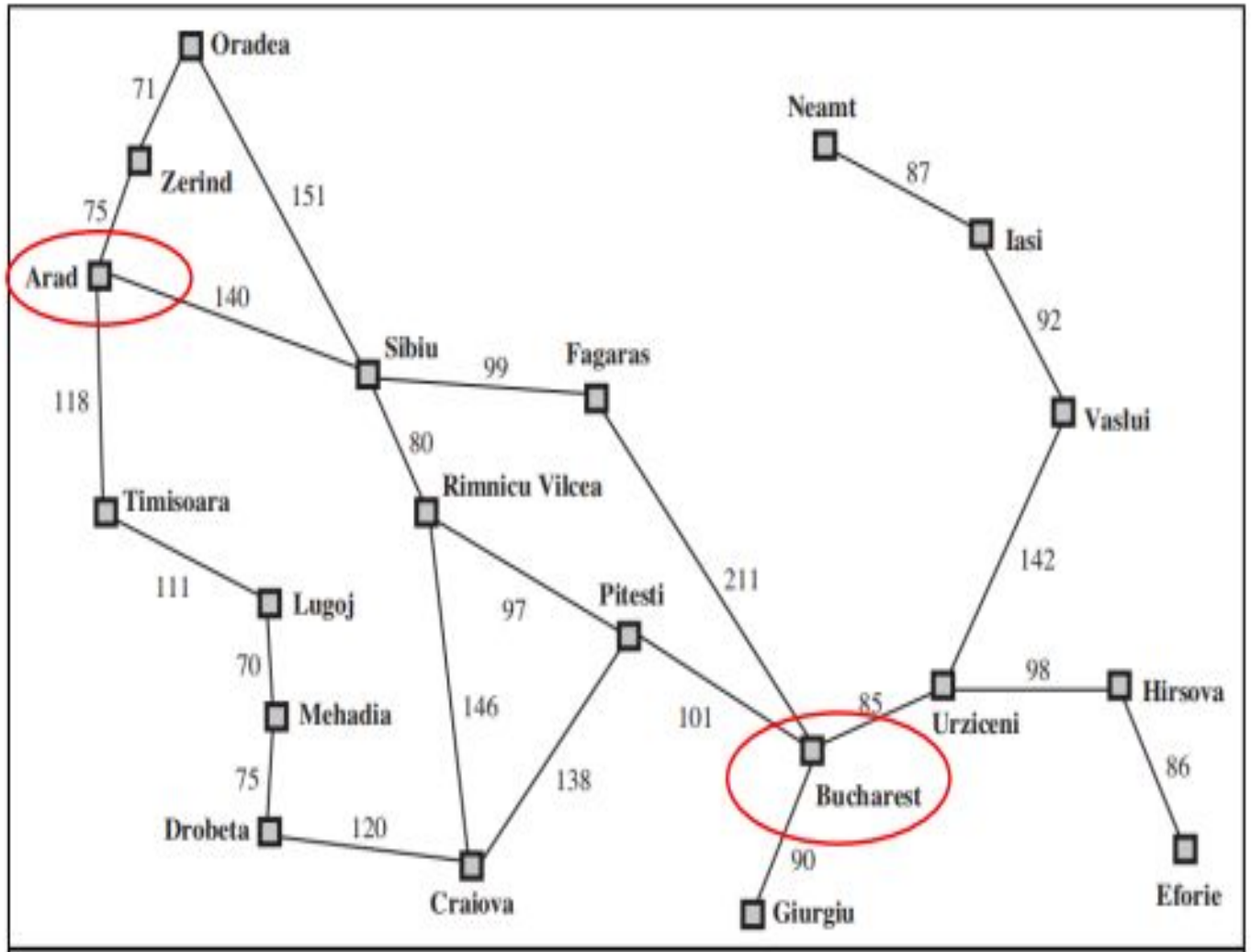
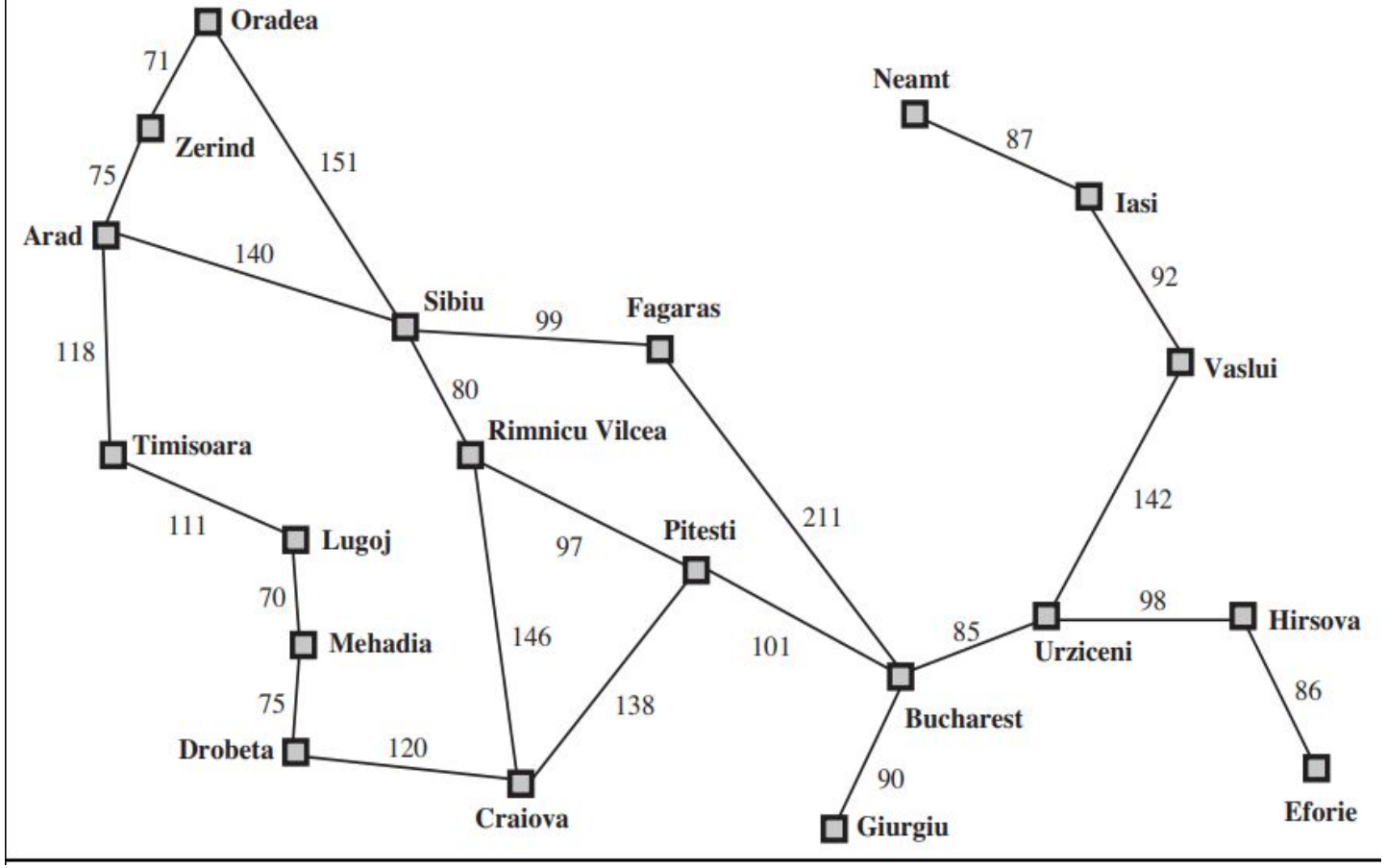


Figure 3.2 A simplified road map of part of Romania.

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

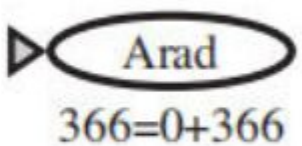
Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.



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

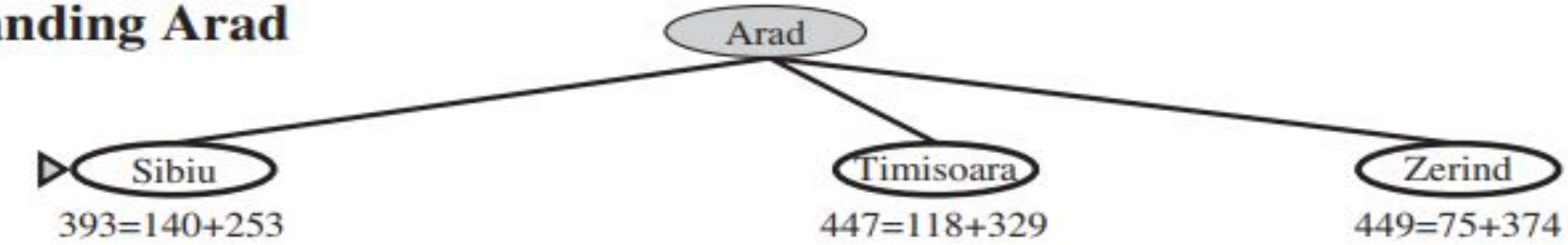
Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

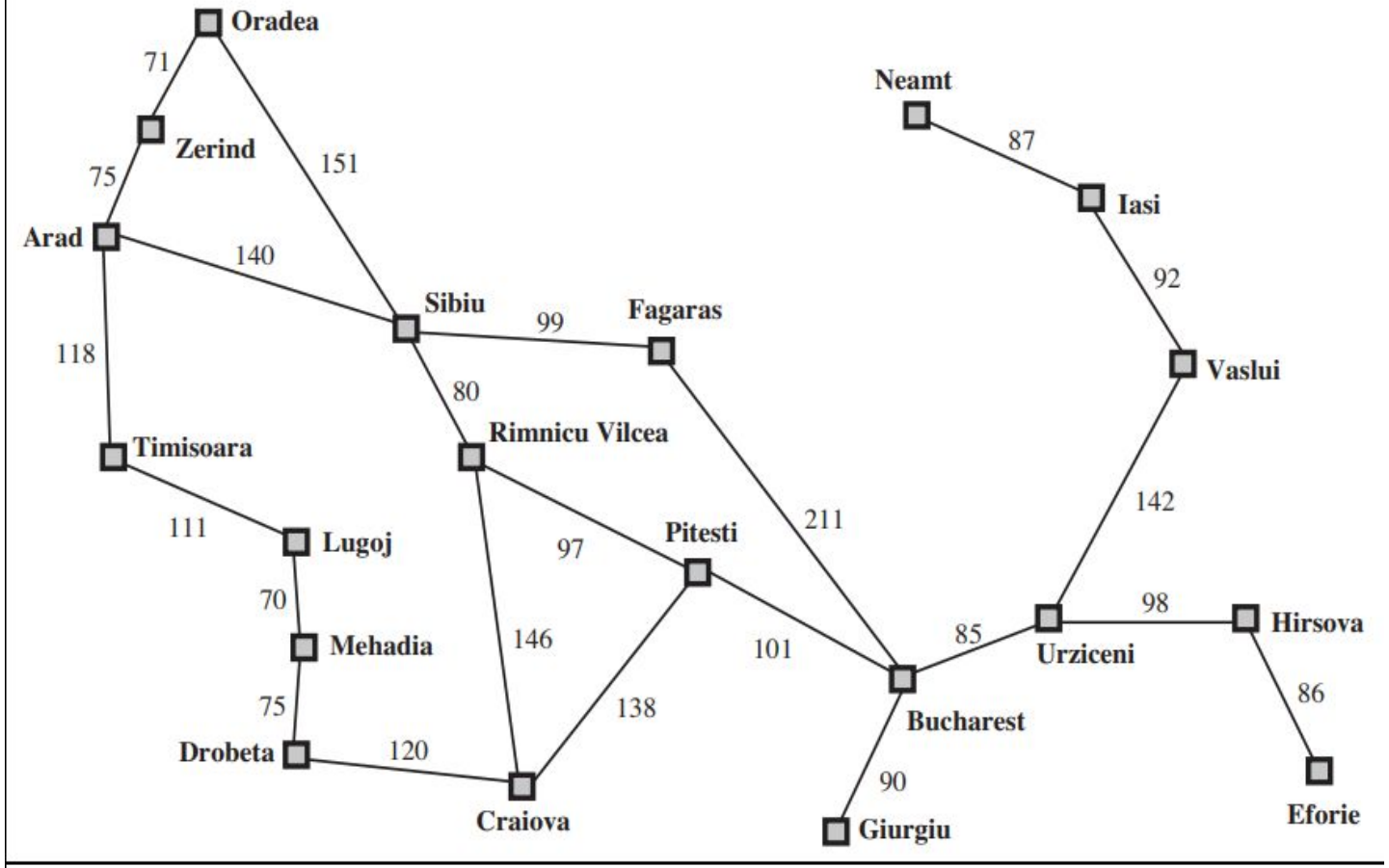
(a) The initial state



366=0+366

(b) After expanding Arad

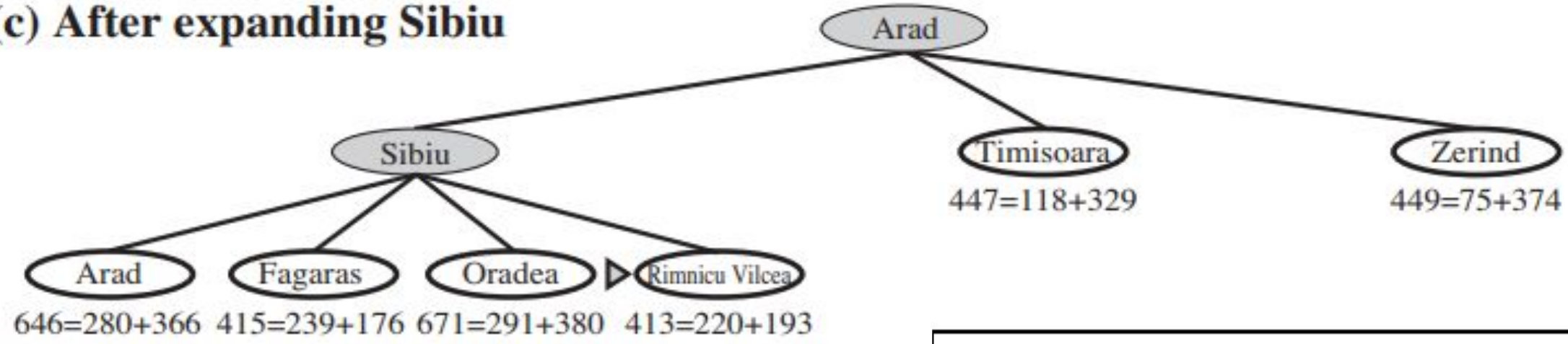


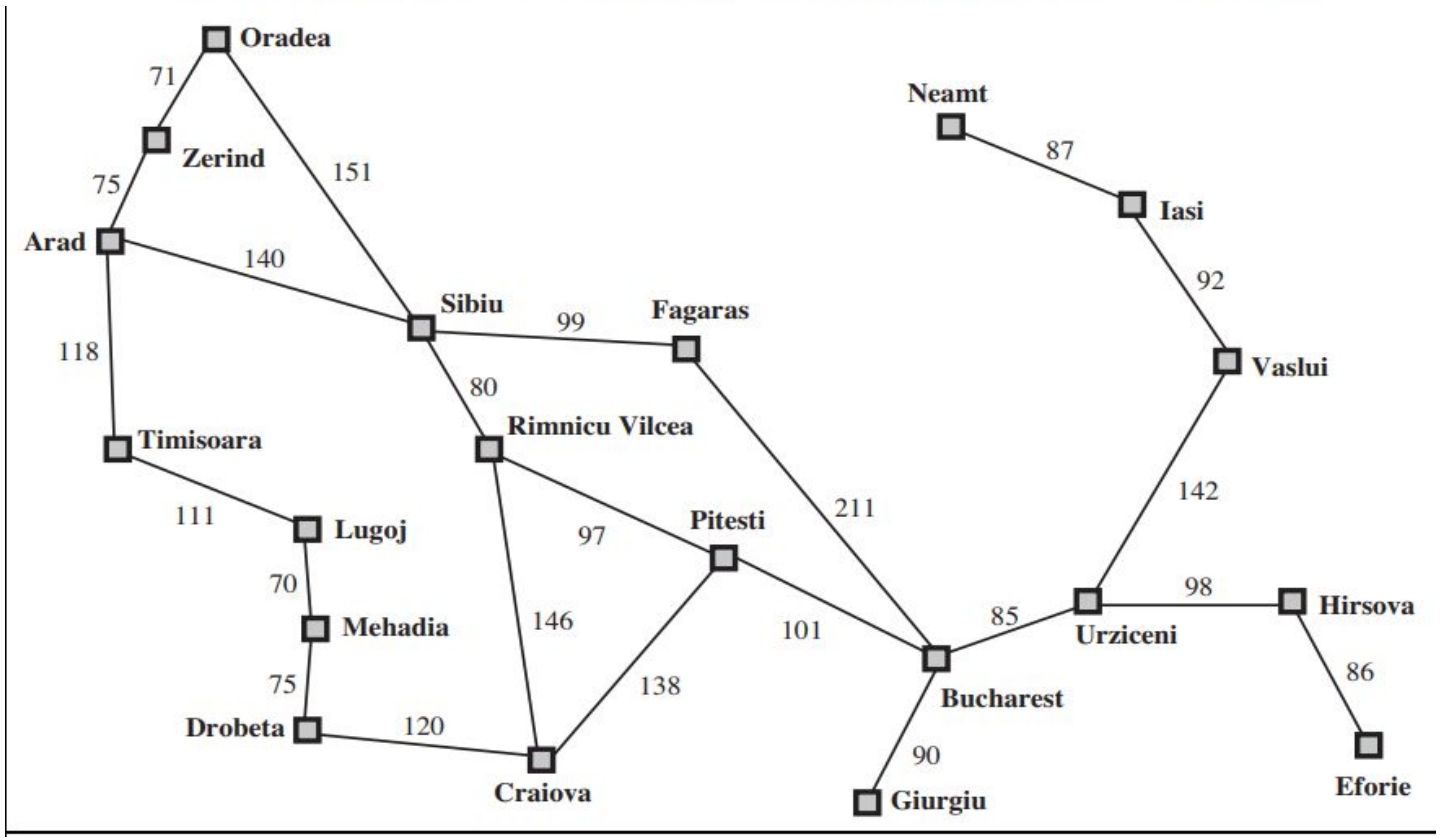


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

(c) After expanding Sibiu

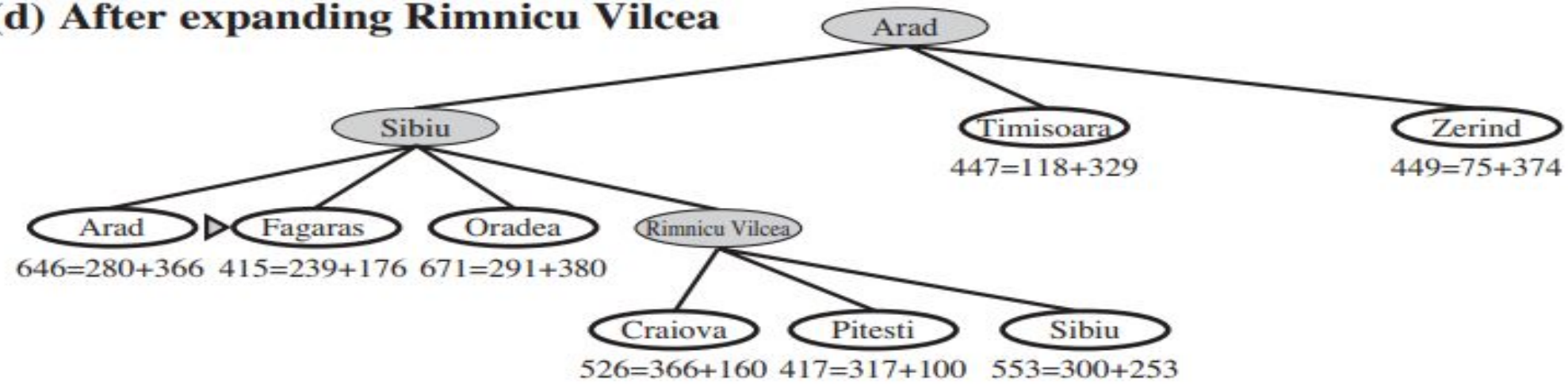


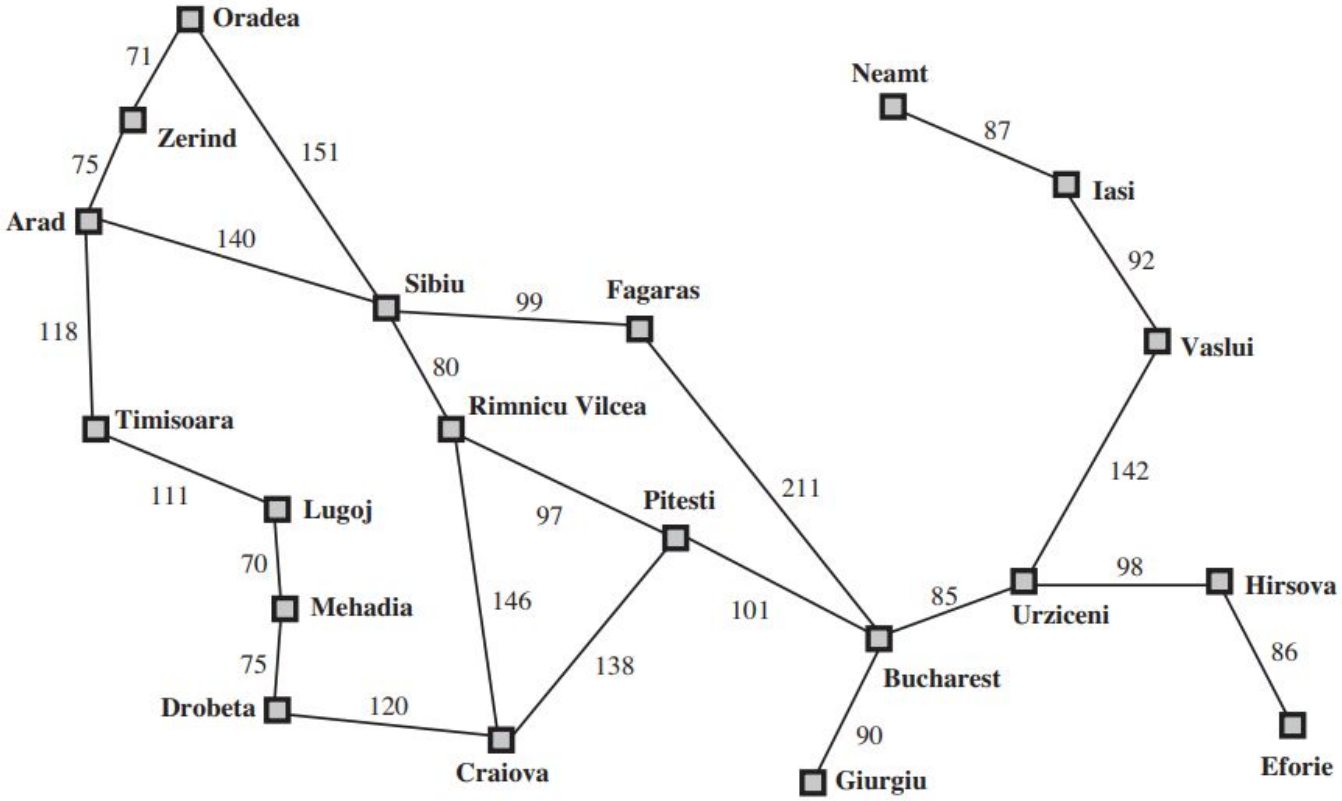


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

(d) After expanding Rimnicu Vilcea

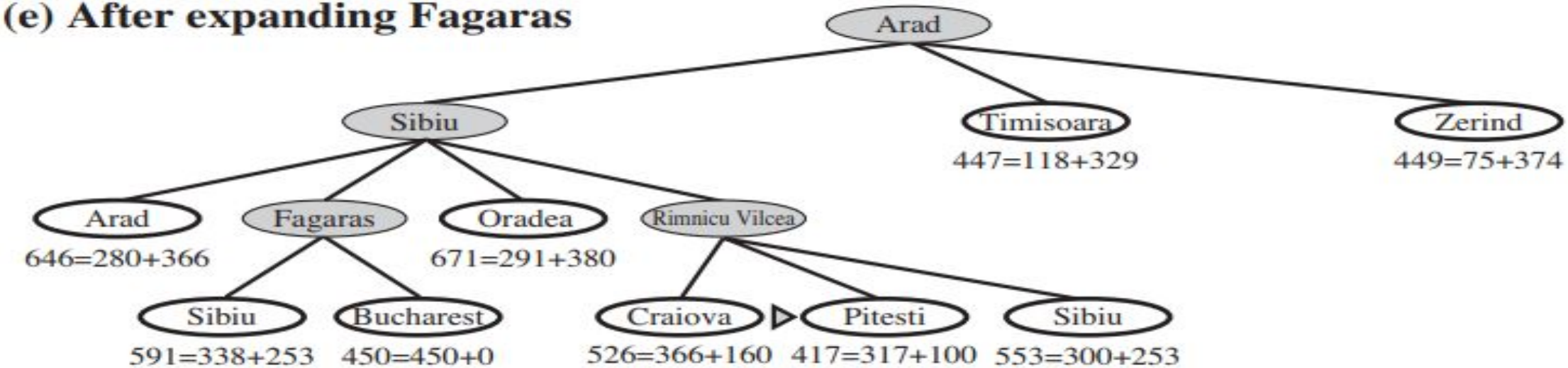


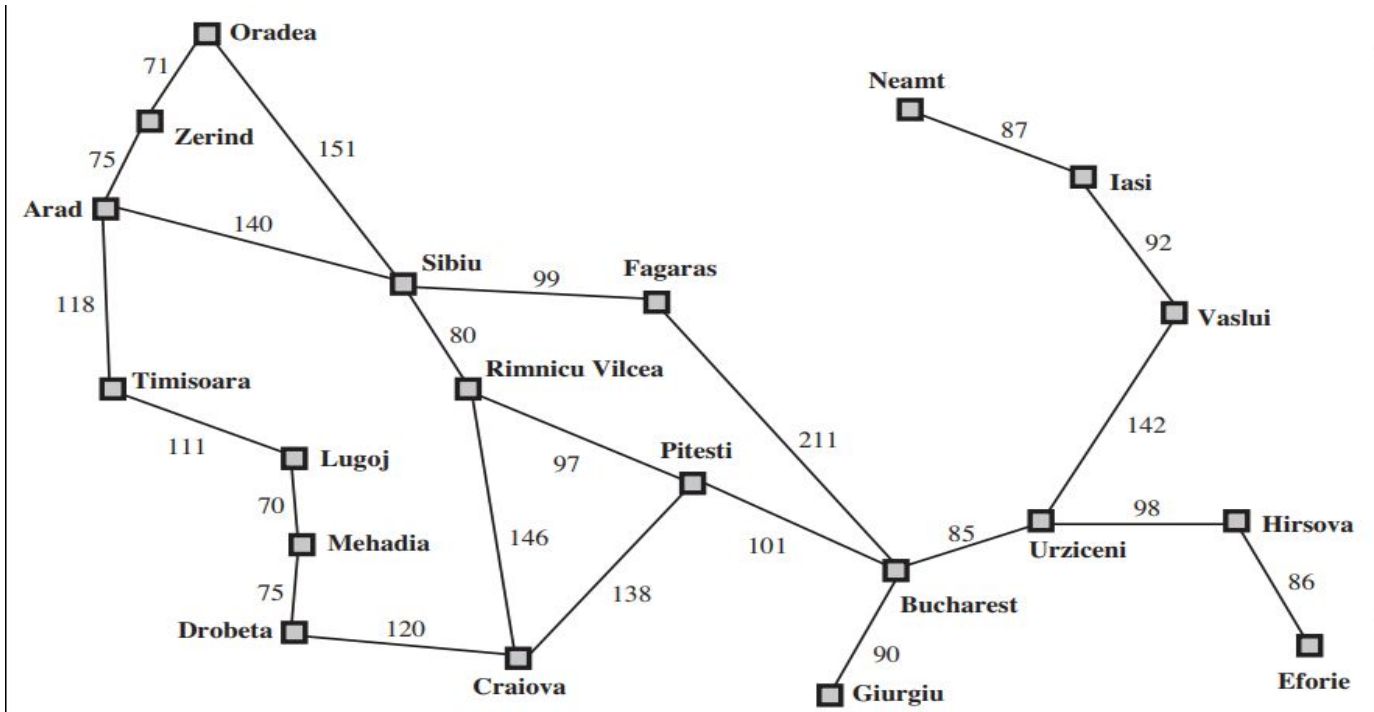


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

(e) After expanding Fagaras

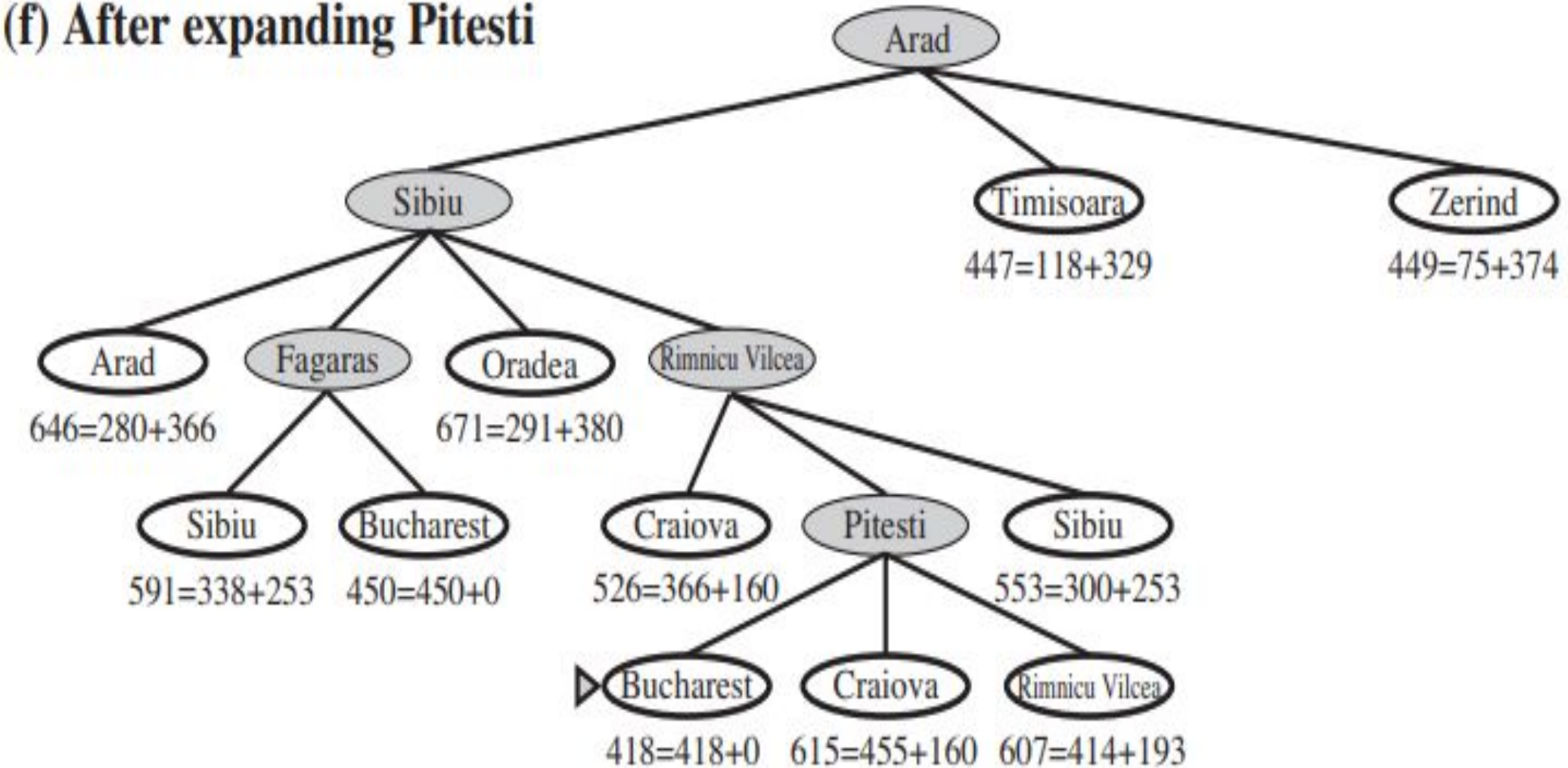




Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

(f) After expanding Pitesti



Optimality of A*

the tree-search version of A∗ *is optimal if* h(n) *is admissible, while the graph-search version is optimal if* h(n) *is consistent.*

A∗ expands no nodes with f(n) > C∗—for example, Timisoara is not expanded in even though it is a child of the root

the subtree below Timisoara is **pruned**; because hSLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality

Pruning eliminates possibilities from consideration without having to examine them

A∗ is **optimally efficient** for any given consistent heuristic.

- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A∗
- This is because any algorithm that *does not* expand all nodes with f(n) < C∗ runs the risk of missing the optimal solution.

if h(n) is consistent, then the values of f(n) along any path are nondecreasing.

The proof follows directly from the definition of consistency.

Suppose n is a successor of n; then $g(n') = g(n) + c(n, a, n')$ for some action a, and we have

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

whenever A* 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 n

because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first

the sequence of nodes expanded by A∗ using GRAPH-SEARCH is in nondecreasing order of f(n).

Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h = 0) and all later goal nodes will be at least as expensive.

Contours in the state space

Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on.

Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

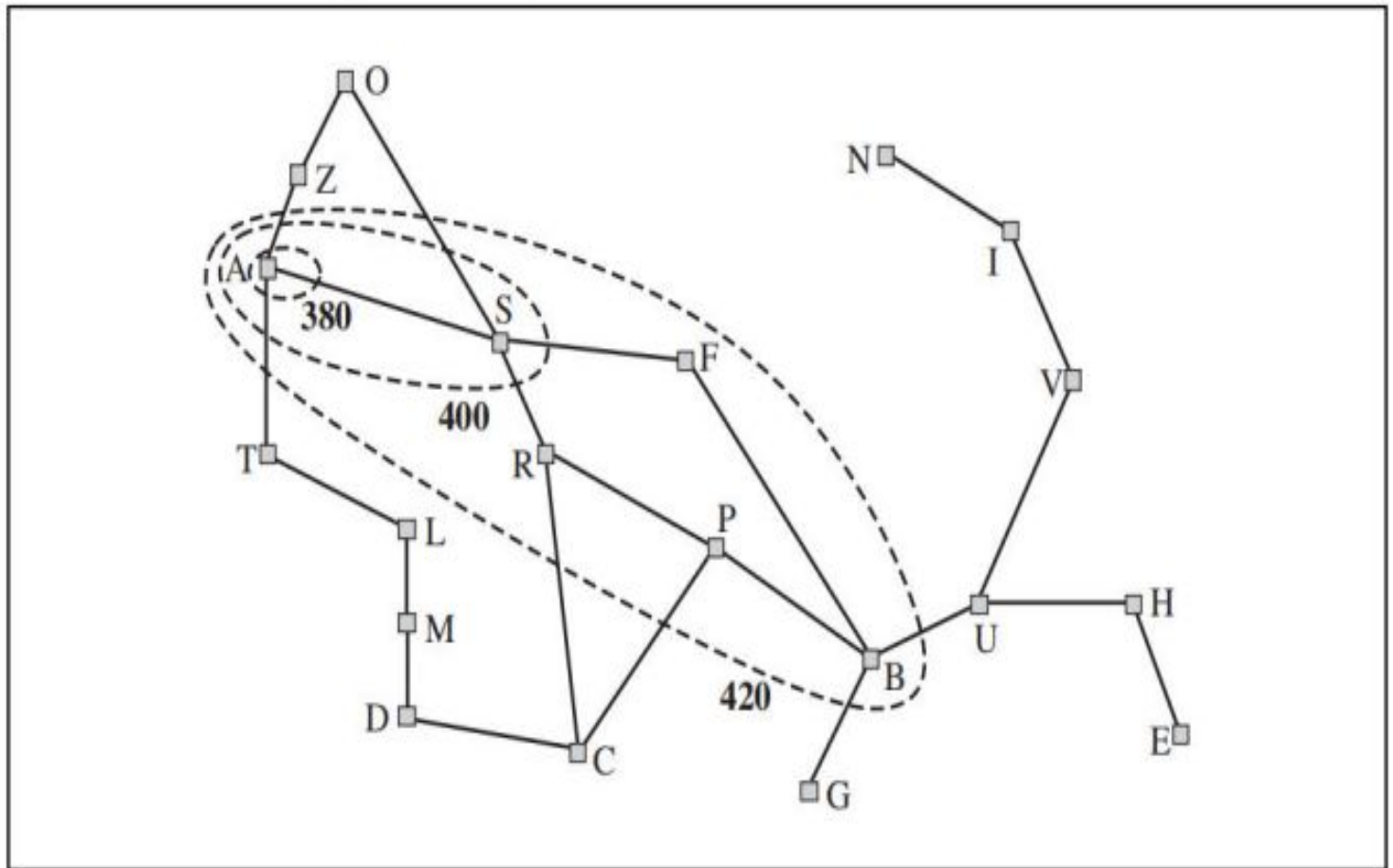


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

If $C*$ is the cost of the

optimal solution path, then we can say the following:

- $A*$ expands all nodes with $f(n) < C*$.
- $A*$ might then expand some of the nodes right on the "goal contour" (where $f(n) = C*$) before selecting a goal node.

Disadvantages of A* Algorithm

the number of states within the goal contour search space is still exponential in the length of the solution.

Memory-bounded heuristic search

- The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A*- (IDA*) A * algorithm.
- The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost (g +h) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration.
- IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

IDA* Algorithm

1. Initialization

- Set the root as the current node and find the f-score.

2. Set the threshold

- Set the cost limit as a threshold for a node ie, maximum f-score allowed for that node for further explorations.

3. Node Expansion

-Expand the current node to its children and find f-scores

4. Pruning

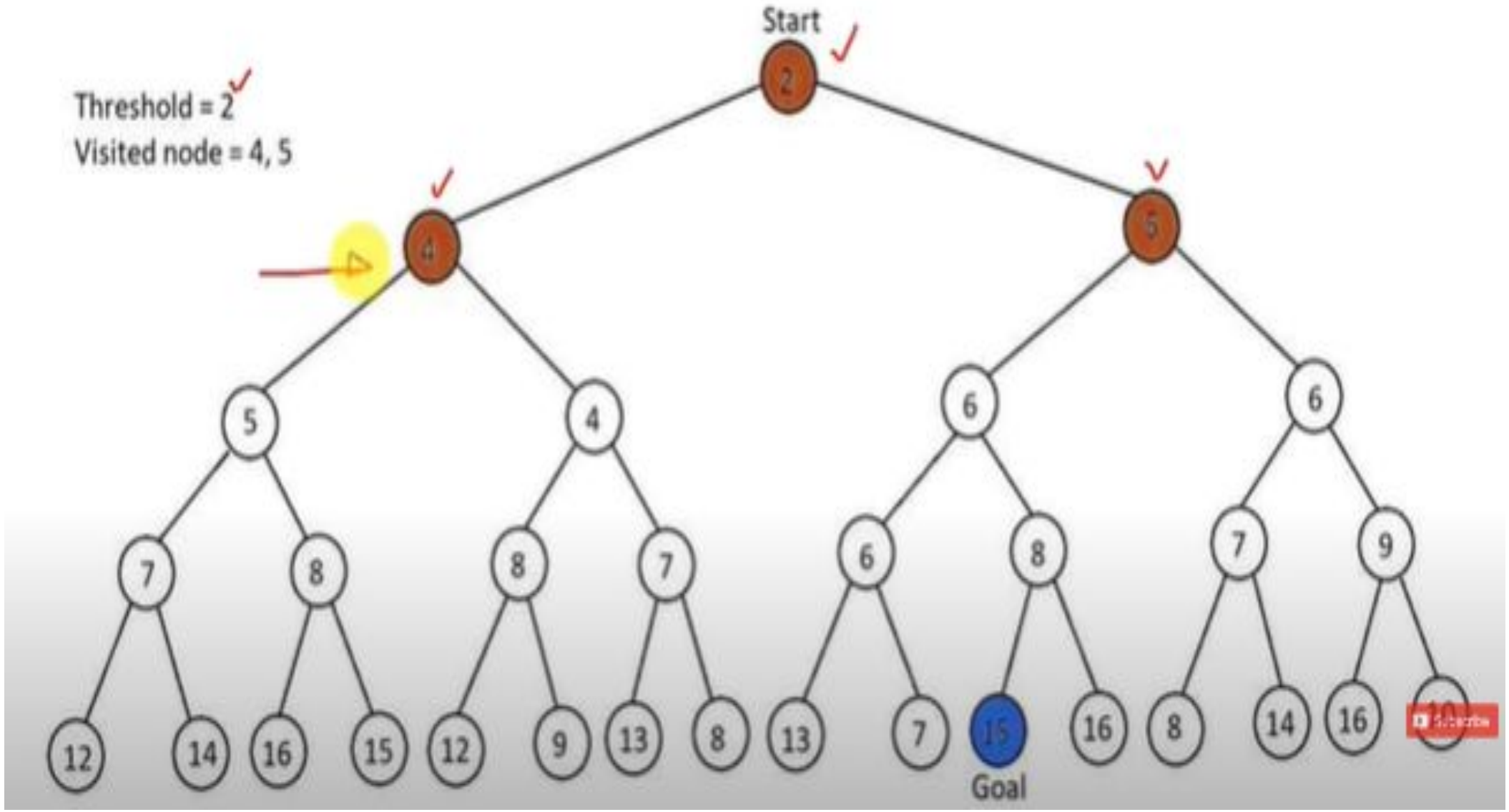
-If for any node the f-score>threshold, prune that node because it’s considered too expensive for that node and store it in the visited node list.

5. Return Path

-If the goal node is found, then return the path from the start node –goal node

6. Update the threshold

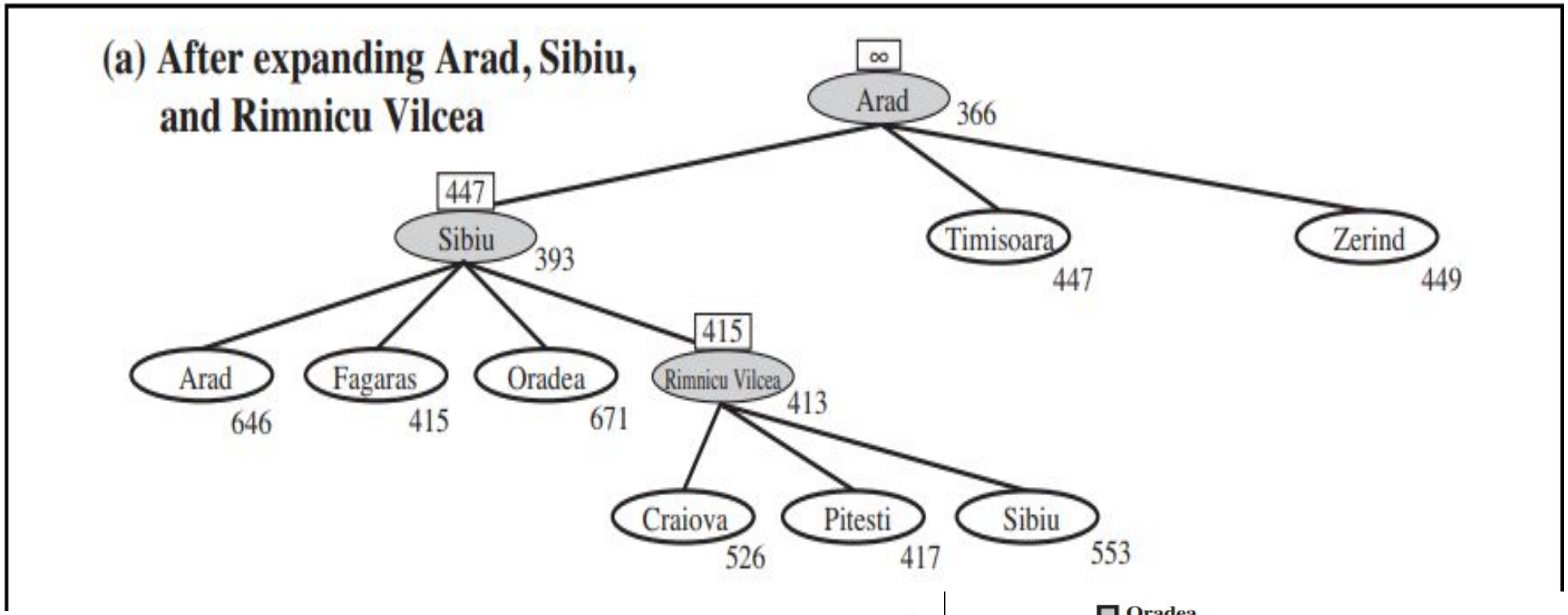
- If goal node is not found, the repeat from step 2 by changing the threshold with the minimum pruned value from the visited node list. And continue it until you reach the goal node.


Start
Threshold = 2
Visited node = 4, 5
2
4
5
5
4
6
6
7
8
8
7
6
8
7
9
12
14
16
15
12
9
13
8
13
7
15
16
8
14
16
10
Goal

RECURSIVE BEST-FIRST SEARCH

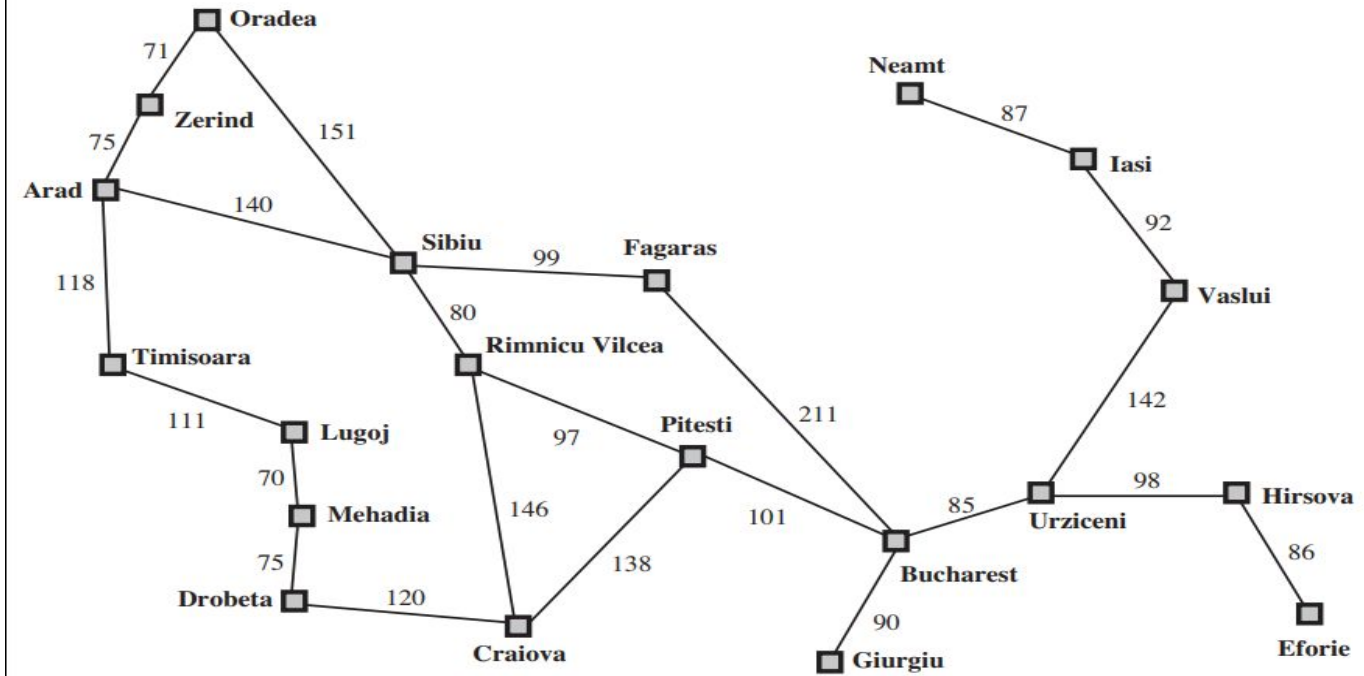
- Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.
- Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.

- As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children.
- In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it’s worth reexpanding the subtree at some later time.

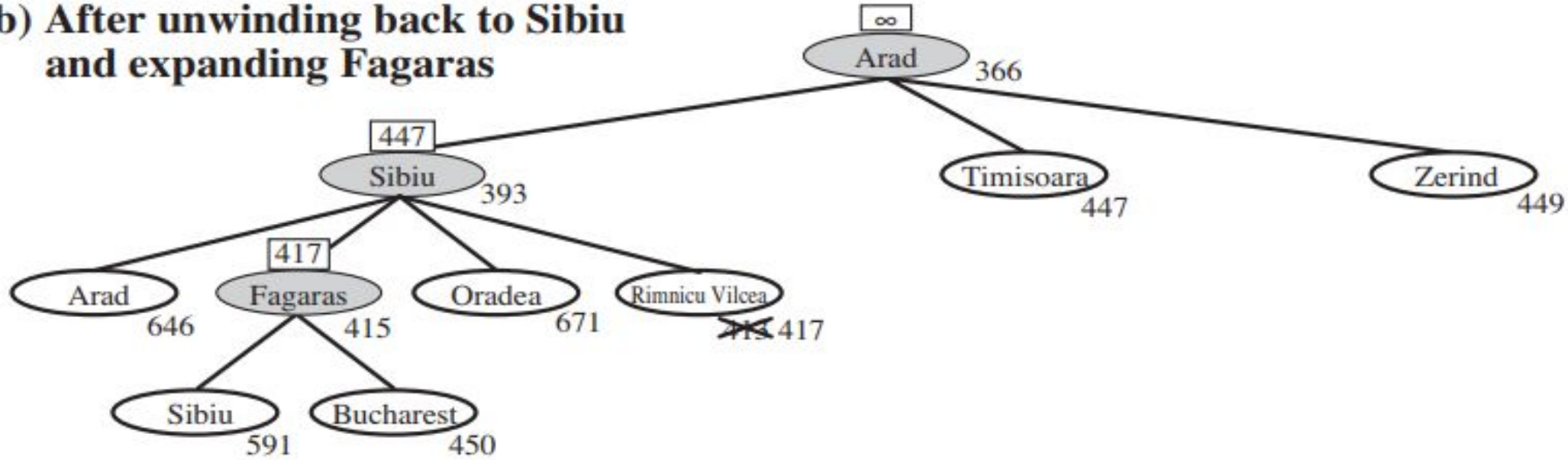


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

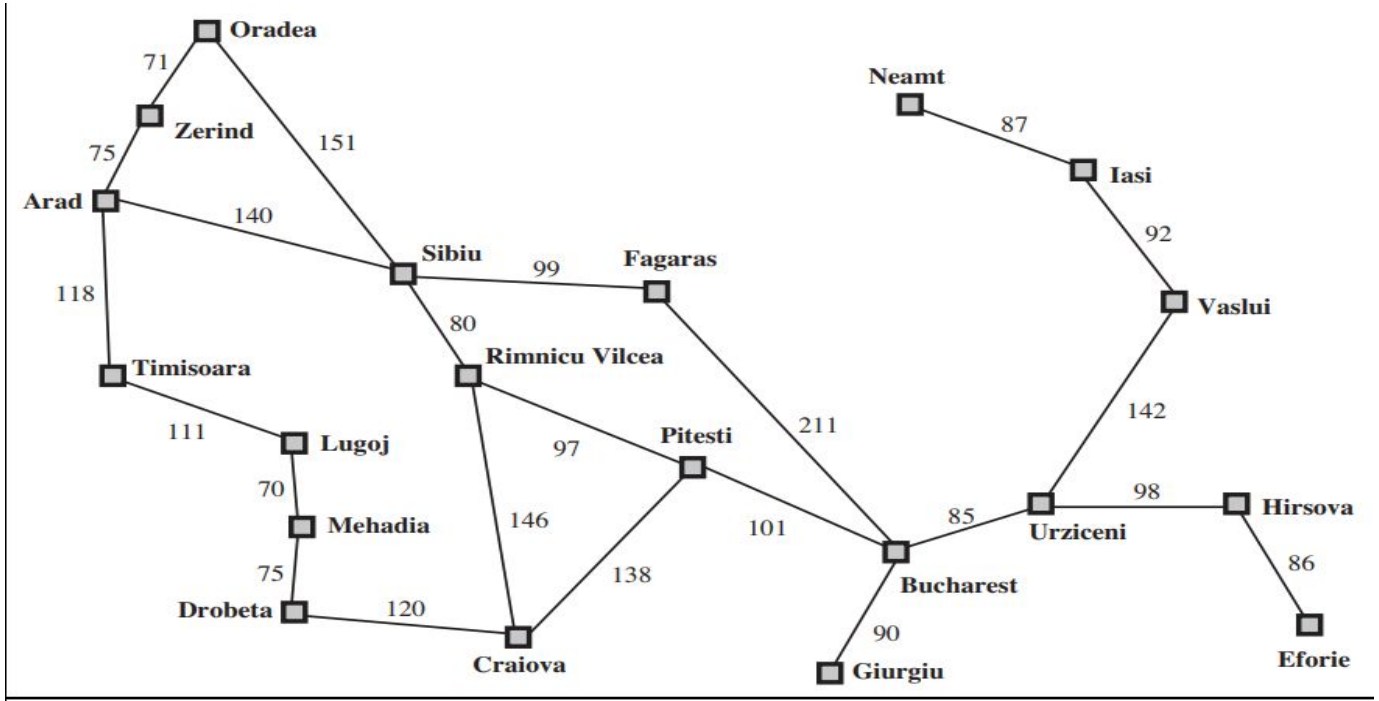


(b) After unwinding back to Sibiu and expanding Fagaras

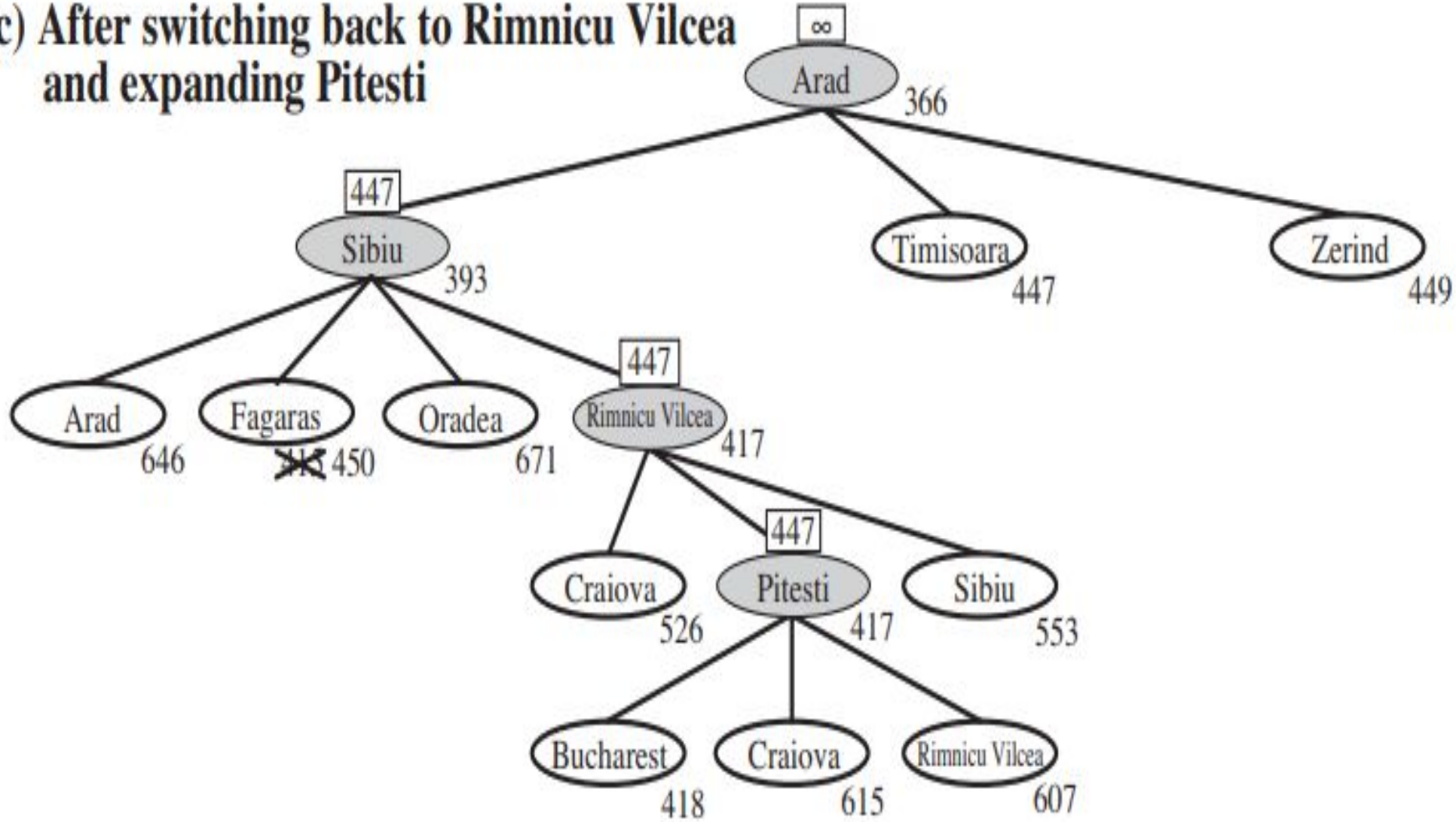


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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



(c) After switching back to Rimnicu Vilcea and expanding Pitesti



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

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

- RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration.
- Like A* tree search, RBFS is an optimal algorithm if the heuristic function h(n) is admissible.
- Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.

SMA*

- SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node—the one with the highest f-value.
- To avoid selecting the same node for deletion and expansion, SMA* expands the newest best leaf and deletes the oldest worst leaf.
- These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory

- If the leaf is not a goal node, then even if it is on an optimal solution path, that solution is not reachable with the available memory.
- Therefore, the node can be discarded exactly as if it had no successors.
- On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of thrashing in disk paging systems.)
- Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*.

4. Learning to search better

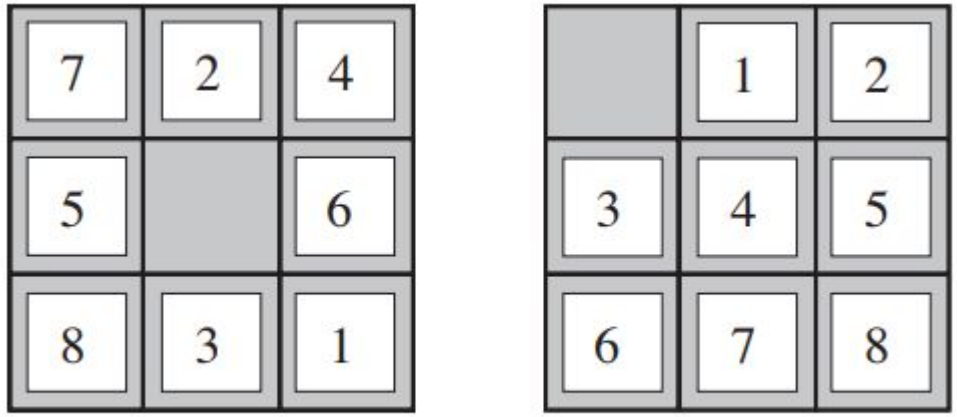
- We have presented several fixed strategies—breadth-first, greedy best-first, and so on—that have been designed by computer scientists.
- Could an agent learn how to search better? The answer is yes, and the method rests on an important concept called the metalevel state space.
- Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an object-level state space such as Romania.
- For example, the internal state of the A* algorithm consists of the current search tree.
- Each action in the metalevel state space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree.
- The goal of learning is to minimize the total cost of problem solving, trading off computational expense and path cost.

HEURISTIC FUNCTIONS

- The 8-puzzle was one of the earliest heuristic search problems.
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.)
- This means that an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states.
- A graph search would cut this down by a factor of about 170,000 because only $9!/2 = 181,440$ distinct states are reachable.

- This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^13, so the next order of business is to find a good heuristic function.
- If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal.
- here are two commonly used candidates:

- h1 = the number of misplaced tiles. In Figure , all of the eight tiles are out of position, so the start state would have h1 = 8.
- h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.



- h2 = the sum of the distances of the tiles from their goal positions.
- Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances.
- This is sometimes called the city block distance or Manhattan distance.
- h2 is also admissible because all any move can do is move one tile one step closer to the goal.
- Tiles 1 to 8 in the start state give a Manhattan distance of

$h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.

As expected, neither of these overestimates the true solution cost, which is 26.

The effect of heuristic accuracy on performance

Heuristic Performance

Experiments on sample problems can determine the number of nodes searched and CPU time for different strategies.

One other useful measure is effective branching factor: If a method expands N nodes to find solution of depth d, and a uniform tree of depth d would require a branching factor of b* to contain N nodes, the effective branching factor is b*

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

Experimental Results on 8-puzzle problems

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

Generating admissible heuristics from relaxed problems

- We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better.
- *How might one have come up with h2?*
- ***Is it possible for a computer to invent such a heuristic mechanically?***
- h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle.

- If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h1 would give the exact number of steps in the shortest solution.
- Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution.
- A problem with fewer restrictions on the actions is called a relaxed problem.
- The state-space graph of the relaxed problem is a supergraph of the original state space *because the removal of restrictions creates added edges in the graph.*

- Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts.
- Hence, **the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.**

- Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent

- Relaxed problems for 8 puzzle:

(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

- From (a), we can derive h2 (Manhattan distance). The reasoning is that h2 would be the proper score if we moved each tile in turn to its destination.
- We can derive h1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step.

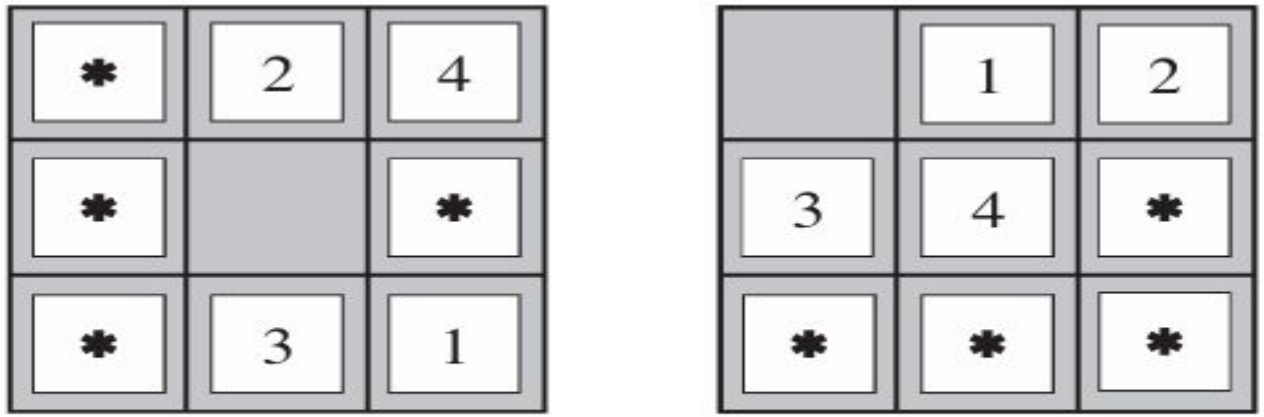
- A program called ABSOLVER can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques.
- ABSOLVER generated a new heuristic for the 8-puzzle that was better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's Cube puzzle.

One problem with generating new heuristic functions is that one often fails to get a single "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\} .$$

Generating admissible heuristics from subproblems: Pattern databases

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.



- The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases.

- The idea behind pattern databases is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank.

- The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

- The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on.
- Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value.
- A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000.
- One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be added, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves.

- This is the idea behind disjoint pattern databases.
- With such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance.
- For 24-puzzles, a speedup of roughly a factor of a million can be obtained.

Learning heuristics from experience

- A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n.

How could an agent construct such a function?

- Solution:

1. Relaxed problems
2. learn from experience-Techniques using neural nets, decision trees etc

- Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state's value, rather than with just the raw state description.

- A common approach is to use a linear combination:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) \ .$$