INFORMED (HEURISTIC) SEARCH STRATEGIES

- uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently
- **best-first search: evaluation function, f(n).**
- **f(n)->**construed as a cost estimate-*lowest evaluation is expanded first*

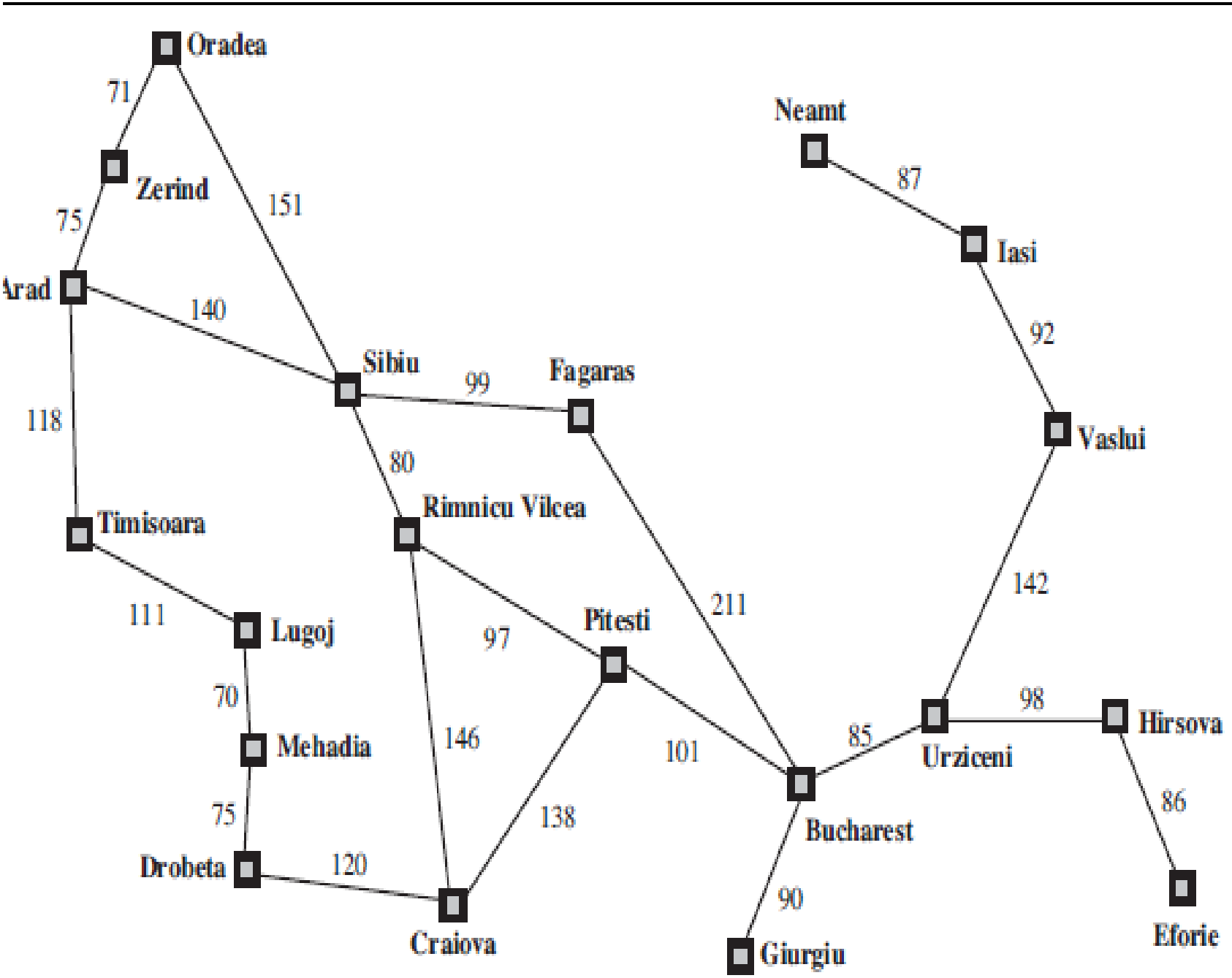
- Most best-first algorithms - **heuristic function, h(n):** estimated cost of the cheapest path from the state at node *n to a goal state.*
- additional knowledge of the problem is imparted to the search algorithm
- For example-Romania map: straight-line distance (h_{SLD})
- It takes a certain amount of experience to know that h_{SLD} is correlated with actual road distance (and as such is a useful heuristic)

Best FS

- f(n) = h(n)

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

- The initial state

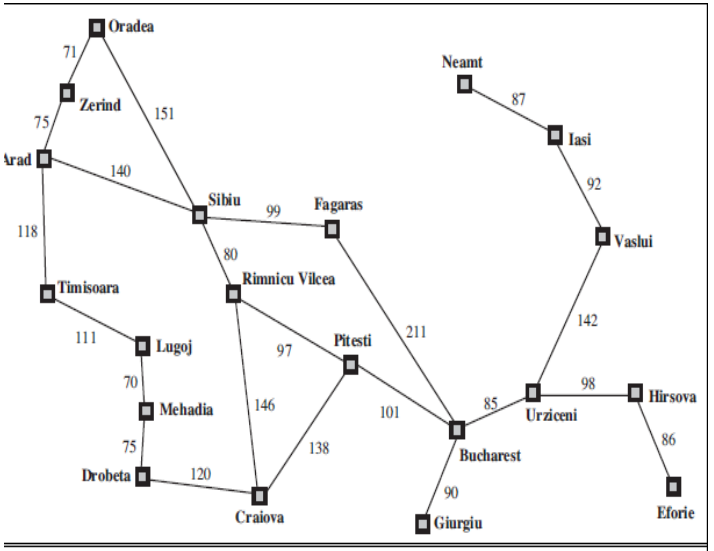
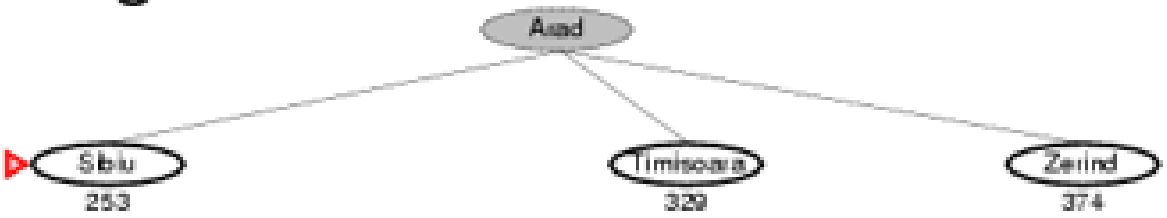


Figure 3.2 A simplified road map of part of Romania.

- After expanding Arad



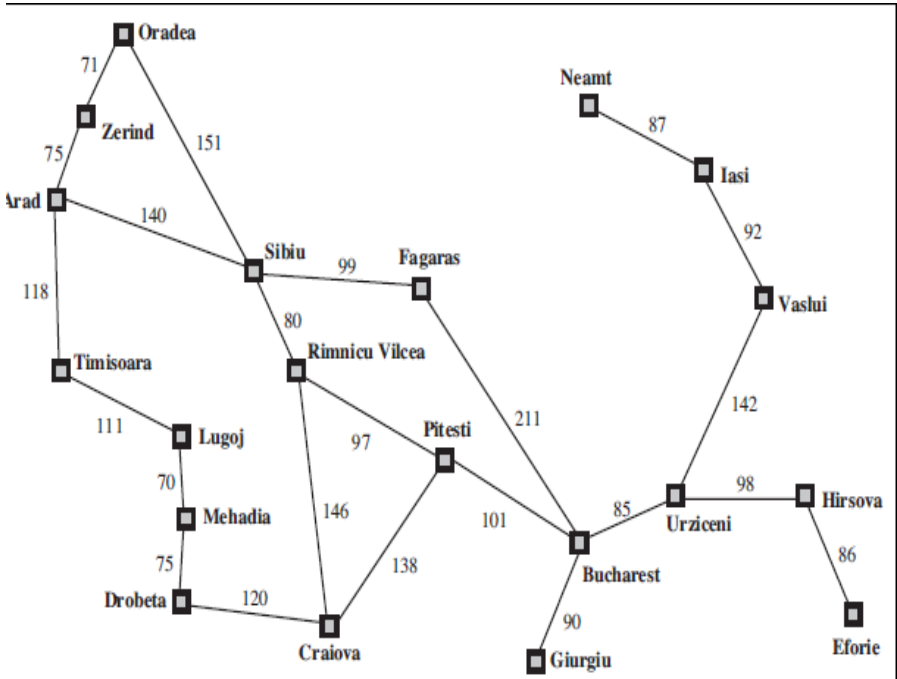
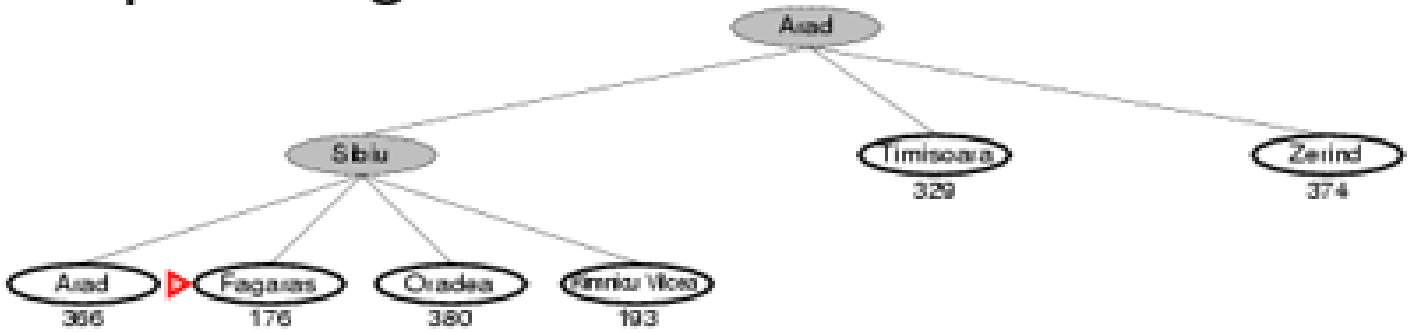


Figure 3.2 A simplified road map of part of Romania.

- After expanding Sibiu



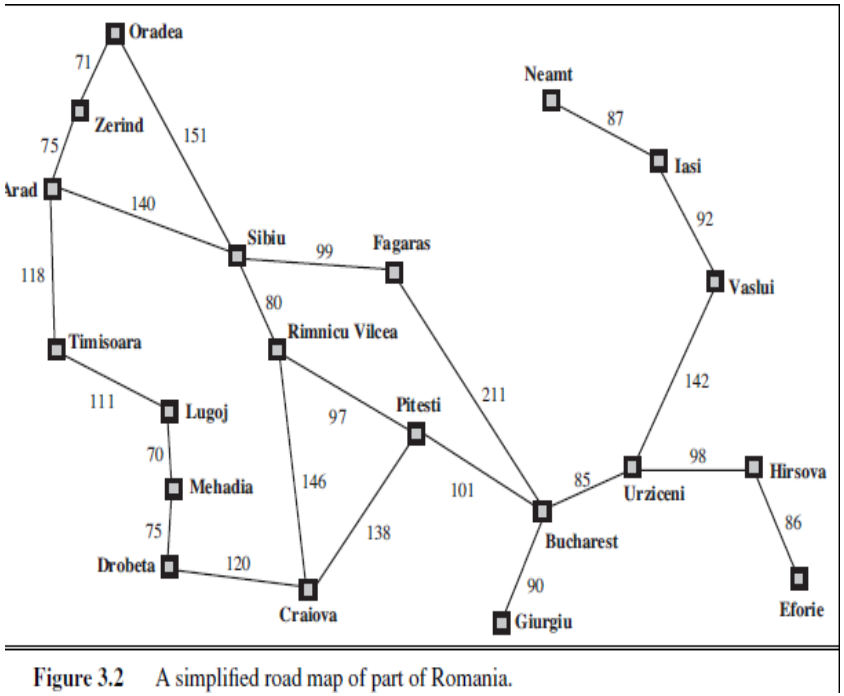
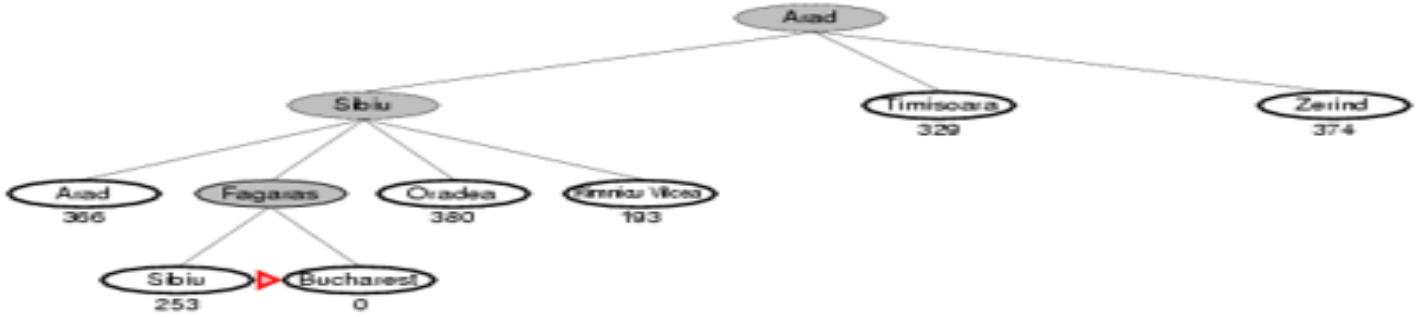


Figure 3.2 A simplified road map of part of Romania.

- After expanding Fagaras



Attention: path [Arad, Sibiu, Fagaras] is not optimal!

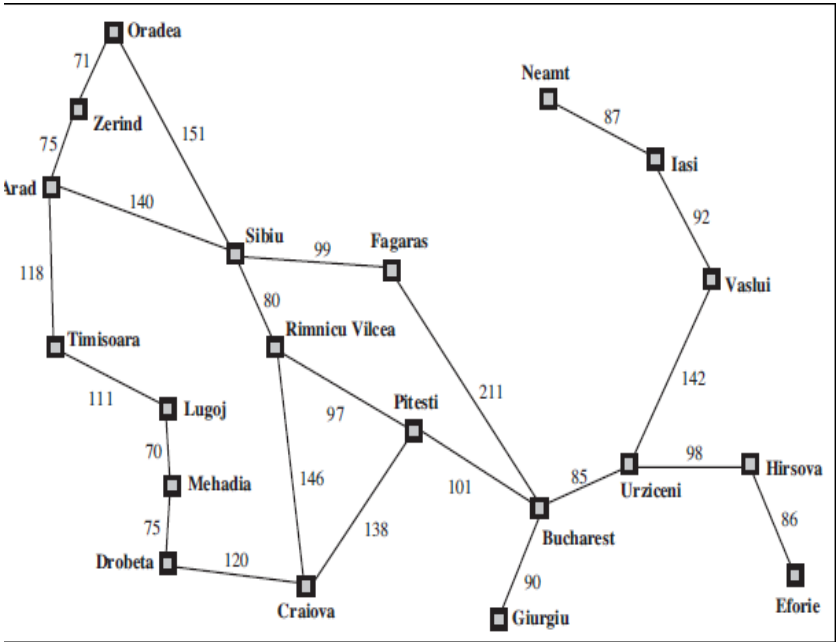


Figure 3.2 A simplified road map of part of Romania.

Greedy Best First Search

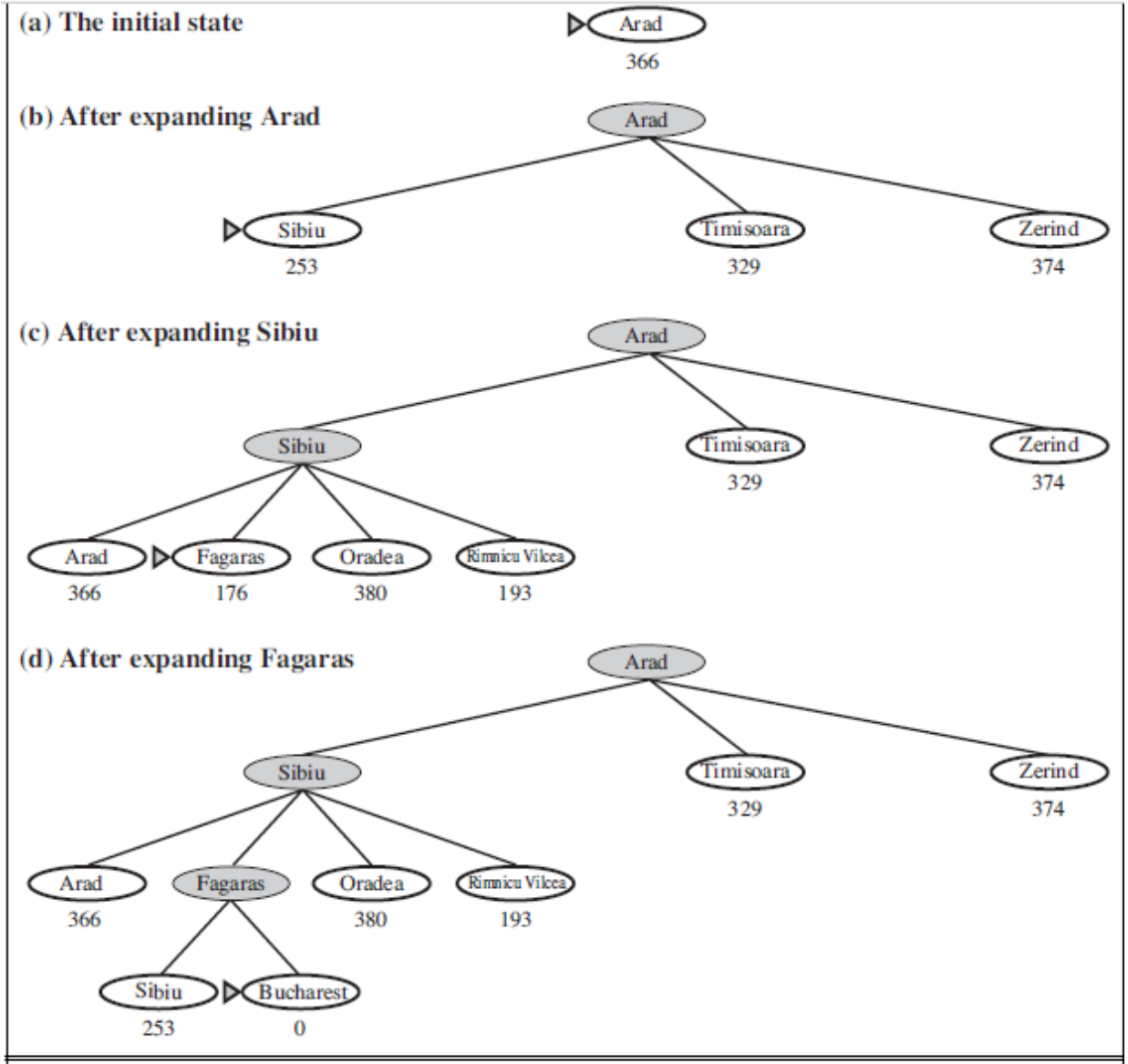
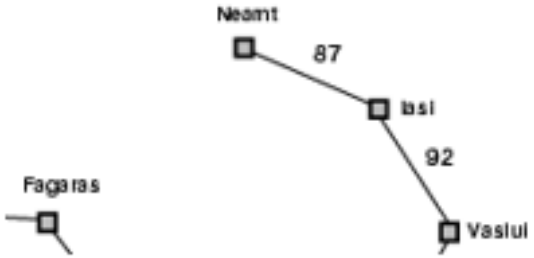


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Properties of GBFS

- Problem: getting from Iasi to Fagaras
- Heuristic suggest that Neamt to be extended first

- Complete? No – can get stuck in loops, e.g., from Iasi to Fagaras:



Iasi → Neamt → Iasi → Neamt → ...

- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

- Be careful: detect repeated steps otherwise solution will be never found.

- Search cost is minimal but It is not optimal: the path via **S** - **F** to **B** is 32 kilometers > the path through **R** - **P**.

  -> algorithm is called “greedy”—at each step it tries to get as close to the goal as it can, instead of?

A* search: Minimizing the total estimated solution cost

- The most widely-known form of best-first search is **A*** search.
- Evaluation function:
  - f(n)=g(n)+h(n)
- It evaluates nodes by combining
  - *g(n)*- cost so far to reach *n*
  - *h(n.)*- is the estimated cost of the cheapest path from n to the goal
  - we have f (n) = estimated total cost of path through *n* to goal
- 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.

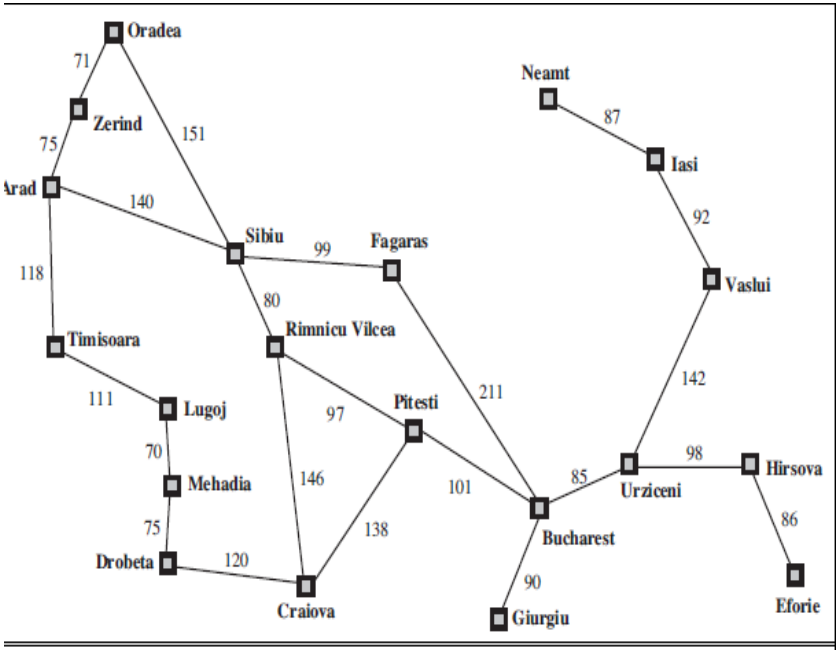
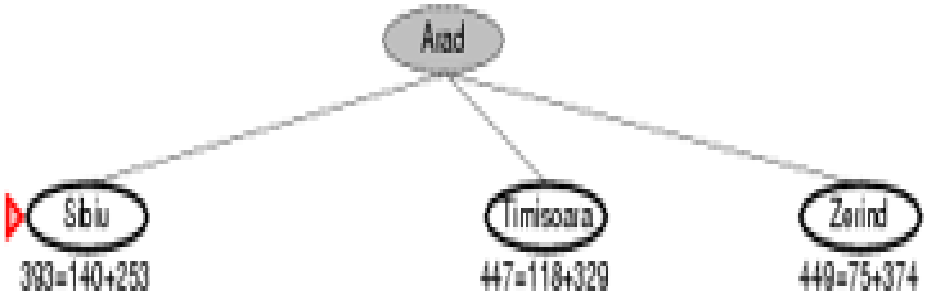


Figure 3.2 A simplified road map of part of Romania.

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



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

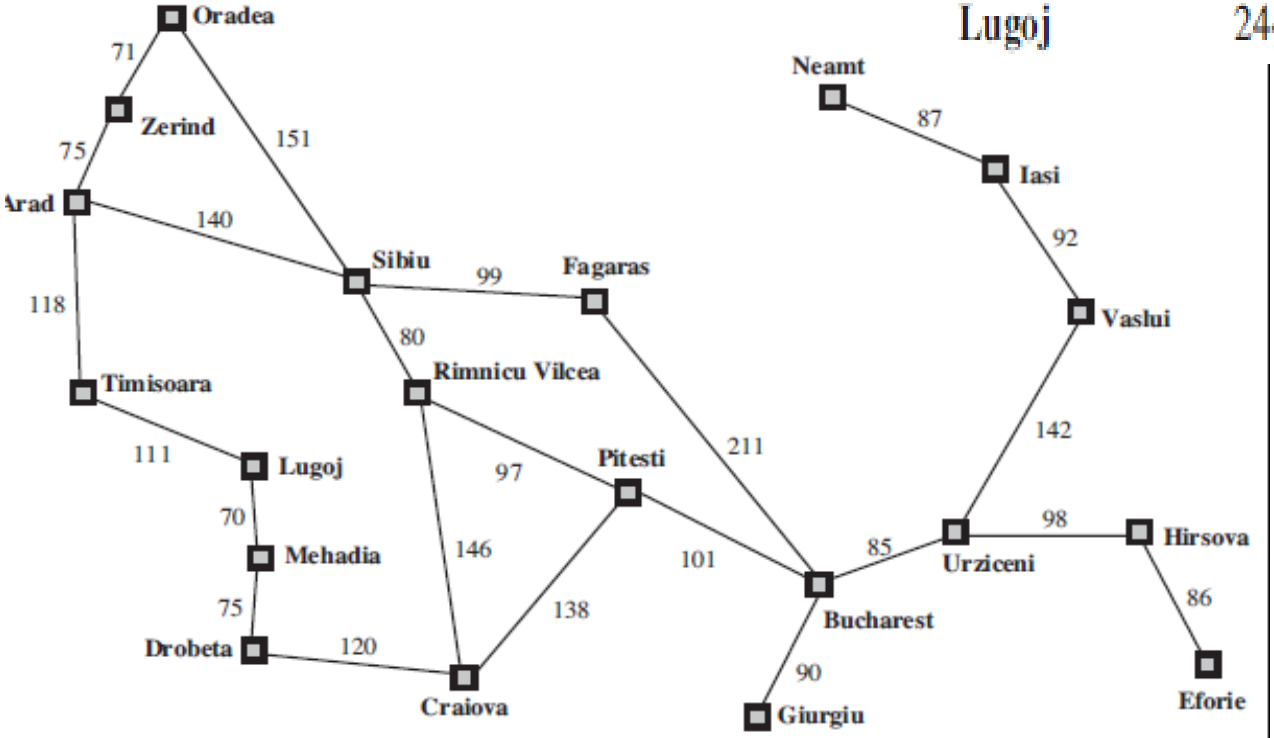
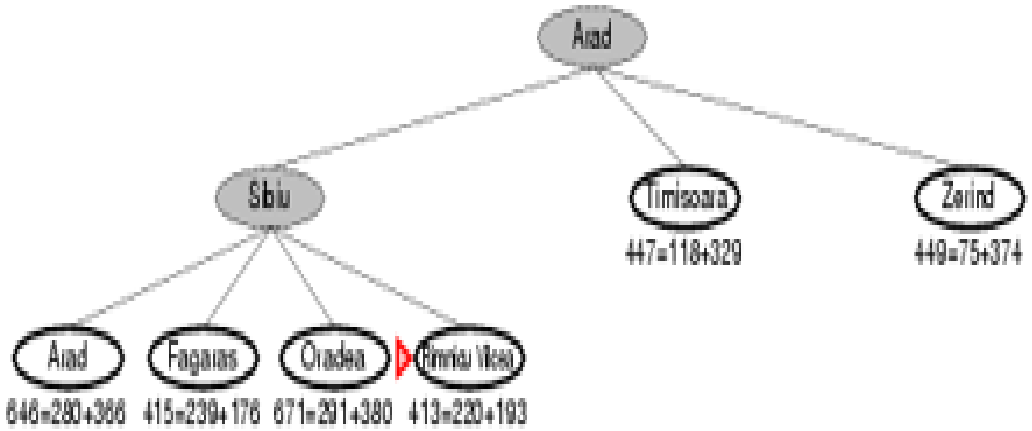


Figure 3.2 A simplified road map of part of Romania.



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

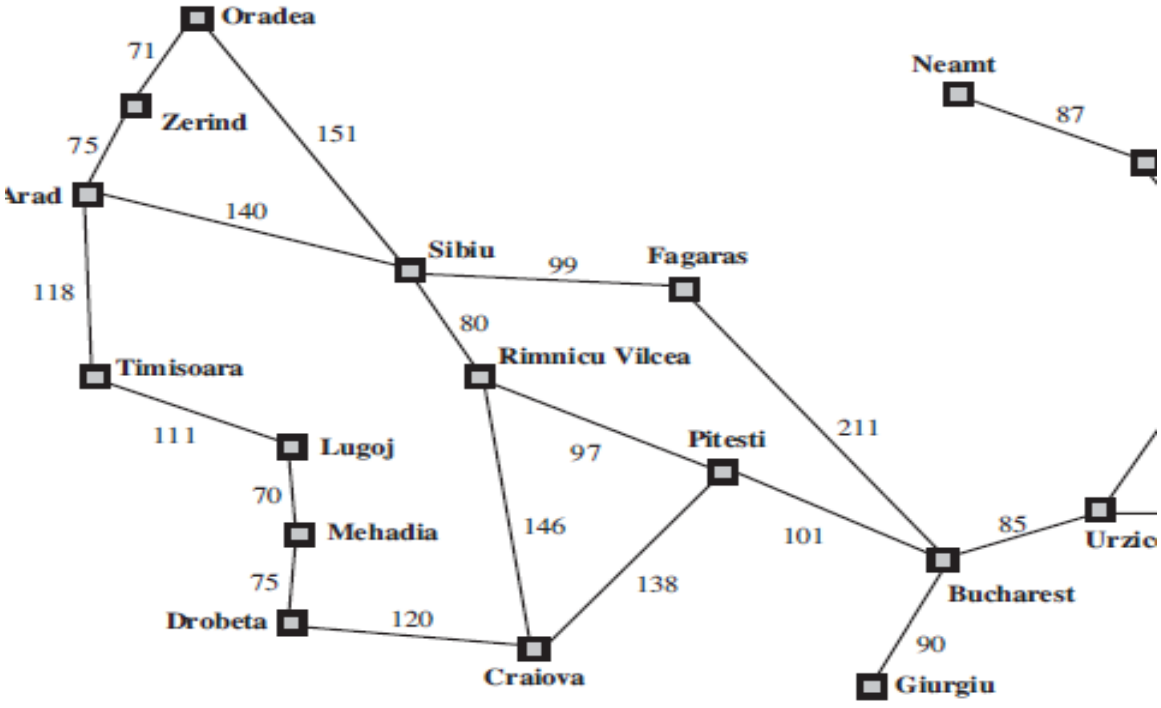
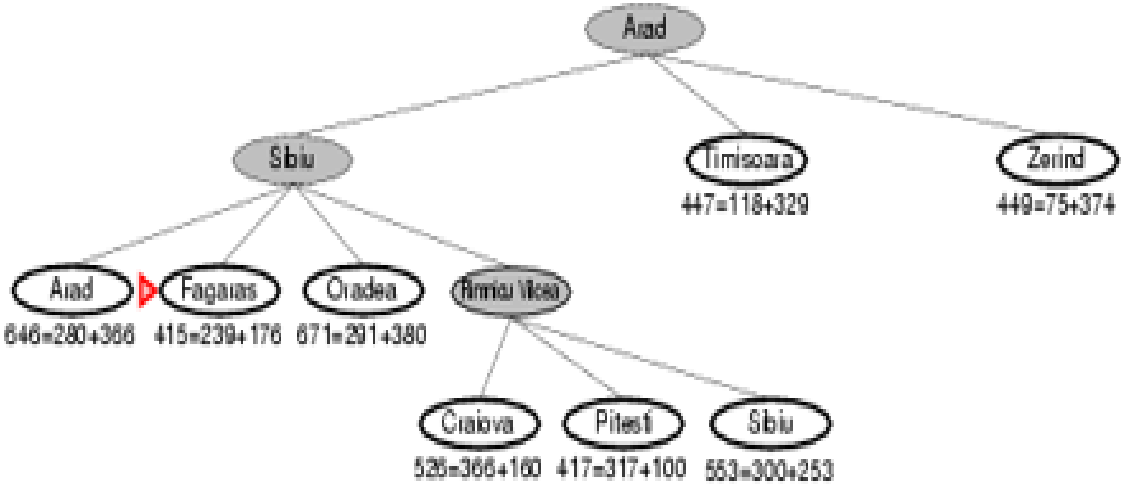


Figure 3.2 A simplified road map of part of Romania.



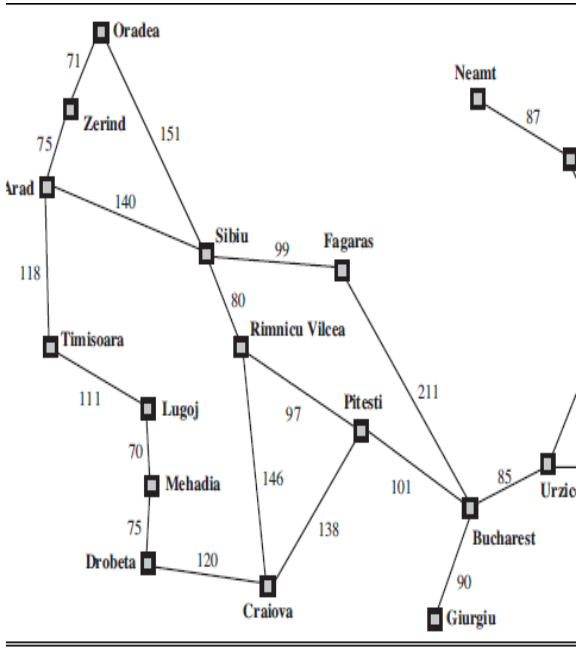
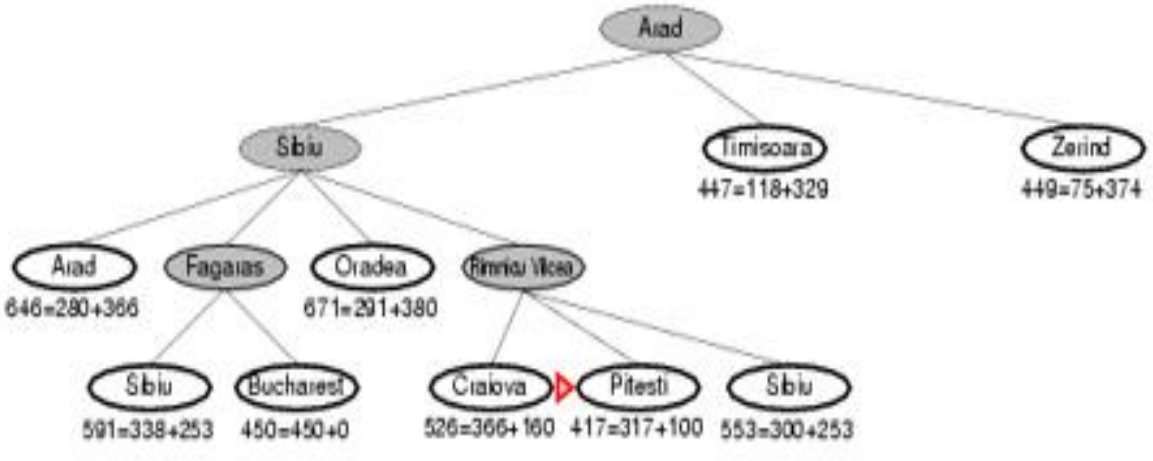


Figure 3.2 A simplified road map of part of Romania.

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



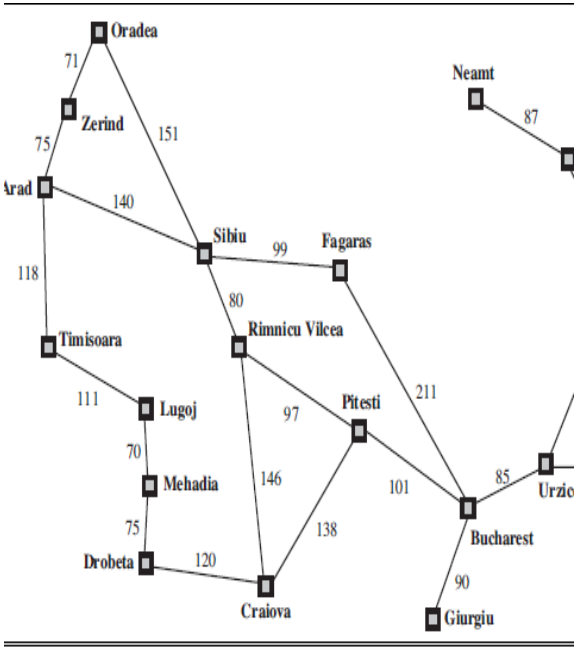
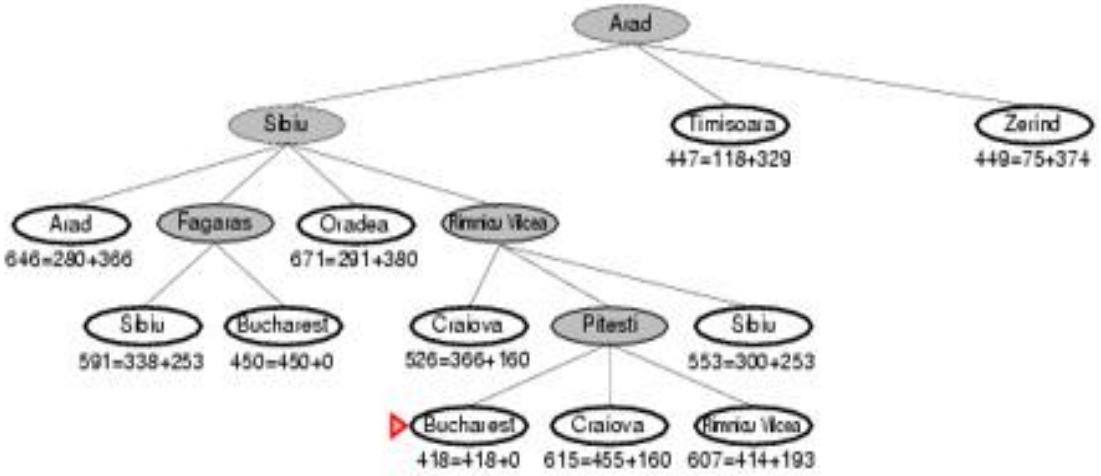


Figure 3.2 A simplified road map of part of Romania.

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



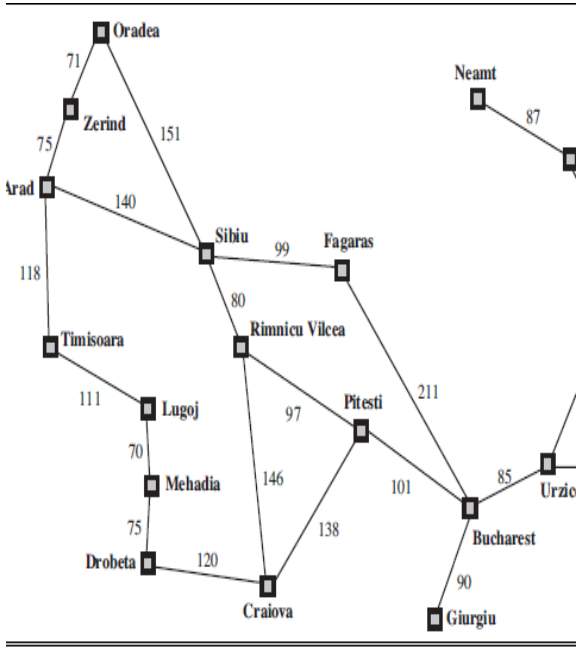
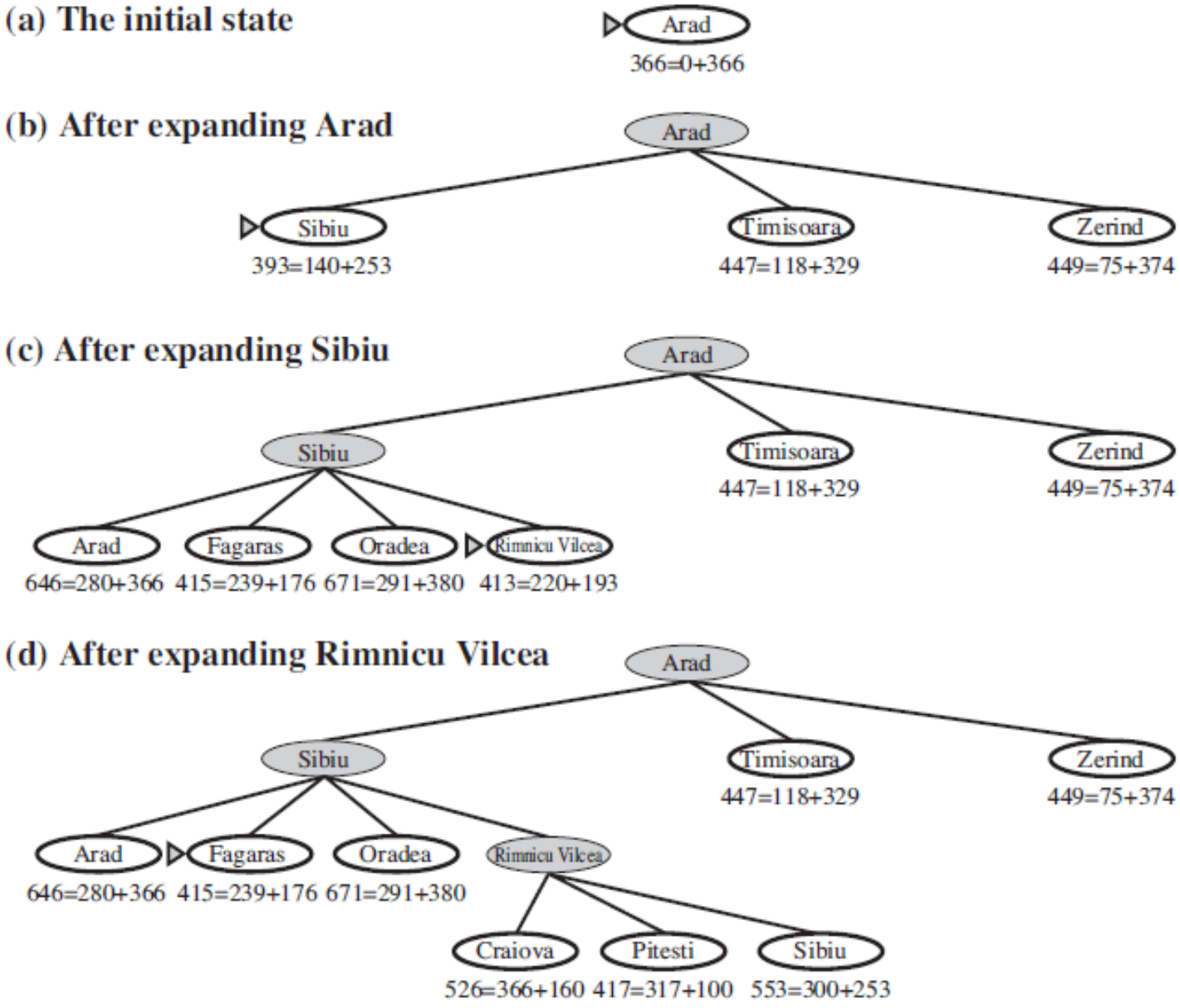
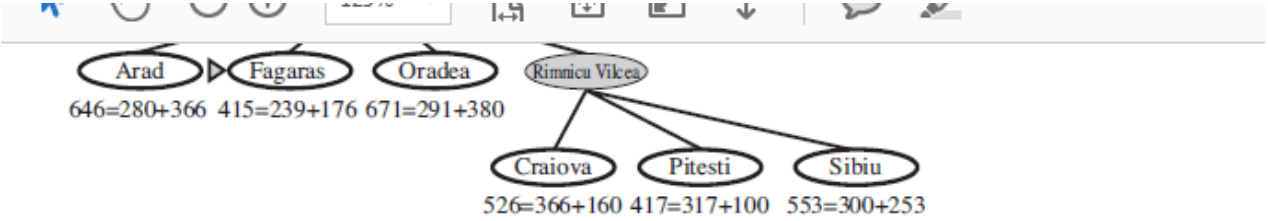


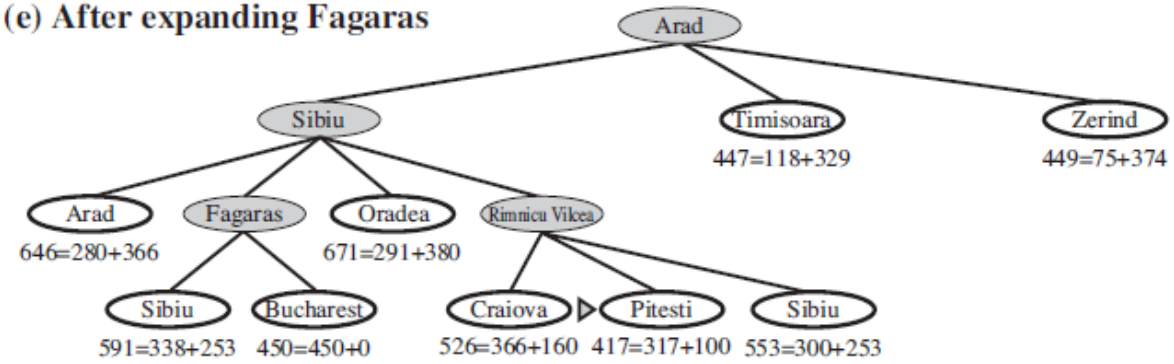
Figure 3.2 A simplified road map of part of Romania.

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

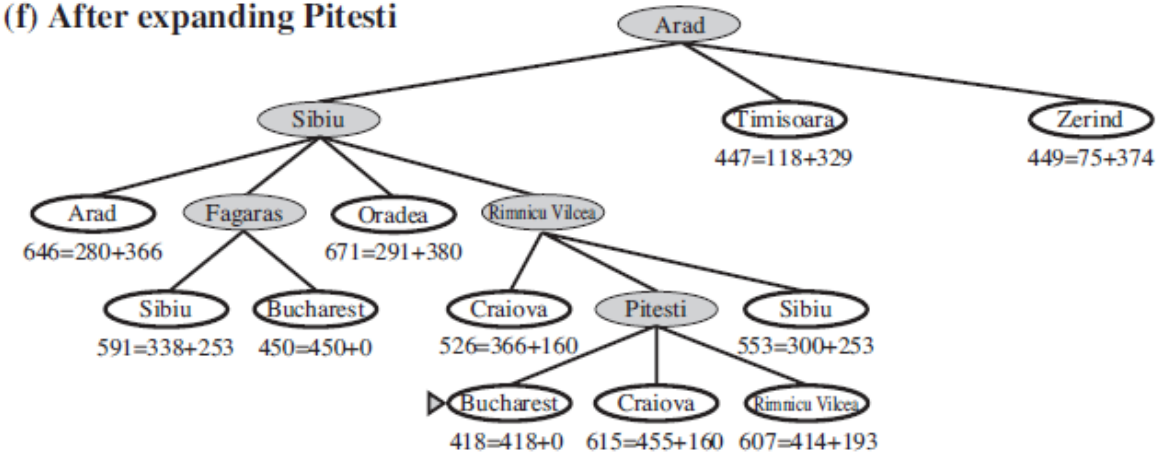

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
393=140+253
447=118+329
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253



(e) After expanding Fagaras



(f) After expanding Pitesti



Optimality and admissible: A*

- optimality : straightforward to analyze if it is used with TREE-SEARCH.
- In this case, A* is optimal if *h(n)* is an admissible heuristic-that is, provided that *h(n)* ***never overestimates*** the cost to reach the goal.
- Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is.
- Since *g(n)* is the exact cost to reach *n,* we have as immediate consequence that *f (n)* never overestimates the true cost of a solution through n.
- An obvious example of an admissible heuristic is the straight-line distance h_{sLD} that we used in getting to goal node.

- we can extract a general proof that A* ***using*** TREE-SEARCH ***is optimal*** *if h(n)* ***is admissible.***
- Suppose a Then, because *G2* is suboptimal and because $h(G_2)$= 0 (true for any goal node), vie know
- $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$.

- Now consider a fringe node *n* that is on an optimal solution path-for example, Pitesti in the example of the preceding paragraph. (There must always be such a node if a solution exists.)
- If *h(n)* does not overestimate the cost of completing the solution path, then we know that
- $f(n) = g(n) + h(n) <= C^*$.

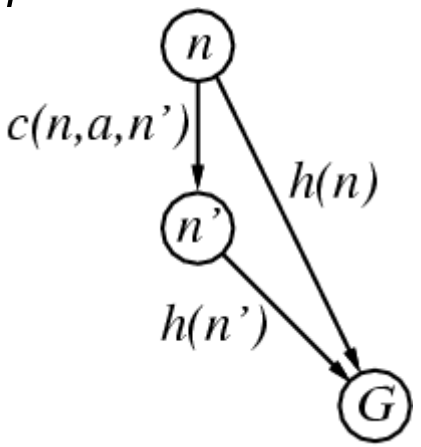
- Now we have shown that f $(n) <= C^* <$ f (G_2), so G_2 will not be expanded anti A* must return an optimal solution.
- If we use the GRAPH-SEARCH algorithm of Figure 3.19 instead of TREE-SEARCH, then this proof breaks down

Consistency:

- The second solution is to ensure that the optimal path to any repeated state is always the first one followed- **consistency** (also called **monotonicity).**
- **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':*
- $h(n) <= c(n,a , n') + h(n')$.
- This is a form of the general **triangle inequality,** which stipulates that each side of a triangle cannot be longer than the sum of the other two sides.
- Here, the triangle is formed by *n, n',* and the goal closest to *n*
- If *h* is consistent, we have
-

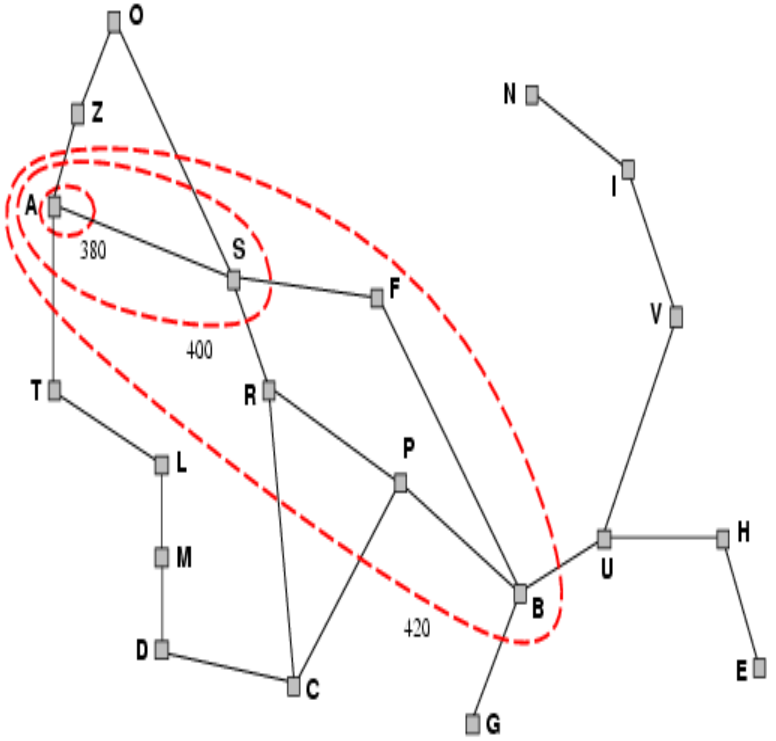
$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n,a,n') + h(n') \\
&\geq g(n) + h(n) \\
&= f(n)
\end{aligned}
$$

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



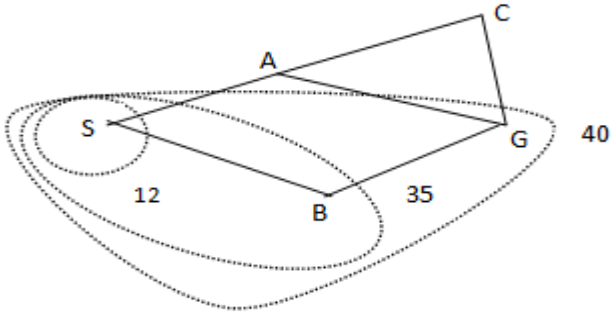
Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



The fact that *f* -costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map.

Figure 4.4 shows an example. Inside the contour labeled 12 (35,40), all nodes have f *(n)* less than or equal to (12 (35,40), and so on.



- Then, because A* expands the fringe 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.
- With uniform-cost search (A* search using *h(n)* = O), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C* is the cost of the optimal solution path, then we can say the following:
- A* expands all nodes with *f* (n) < C*.

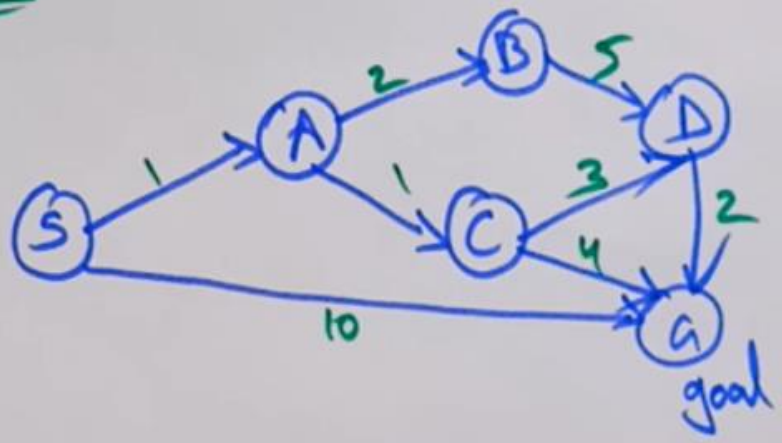
Performance measures

- Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have higher *f* -cost, and thus higher g-cost (because all goal nodes have *h(n)* = 0).
- One final observation is that among optimal algorithms of this type-algorithms that extend search paths from the root-A* is **optimally efficient** for any given heuristic function.
- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with f *(n)* = C*).
- This is because any algorithm that *does not* expand all nodes with f *(n)* < C* runs the risk of missing the optimal solution.
- That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying.

A* Search Algorithm :

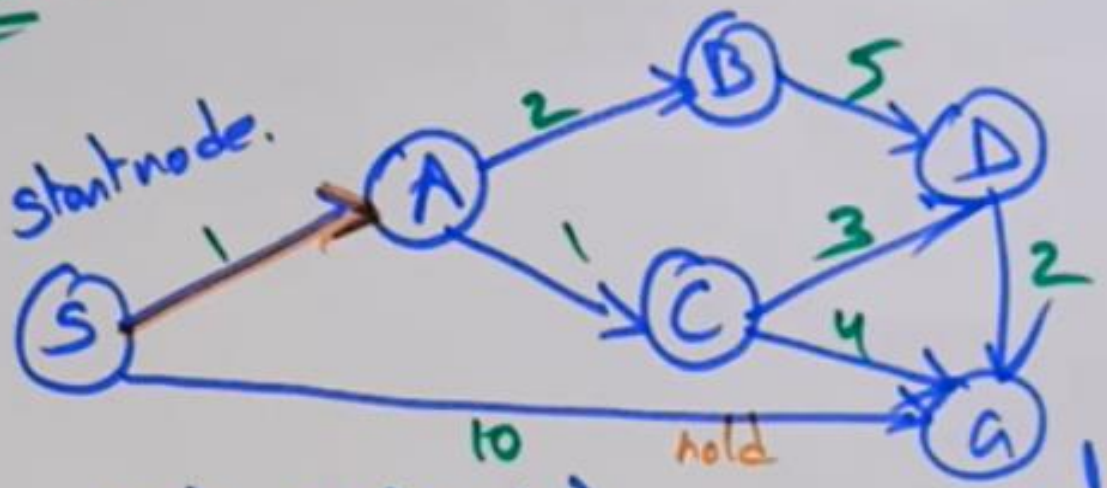
Example :

State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



ch Algorithm :

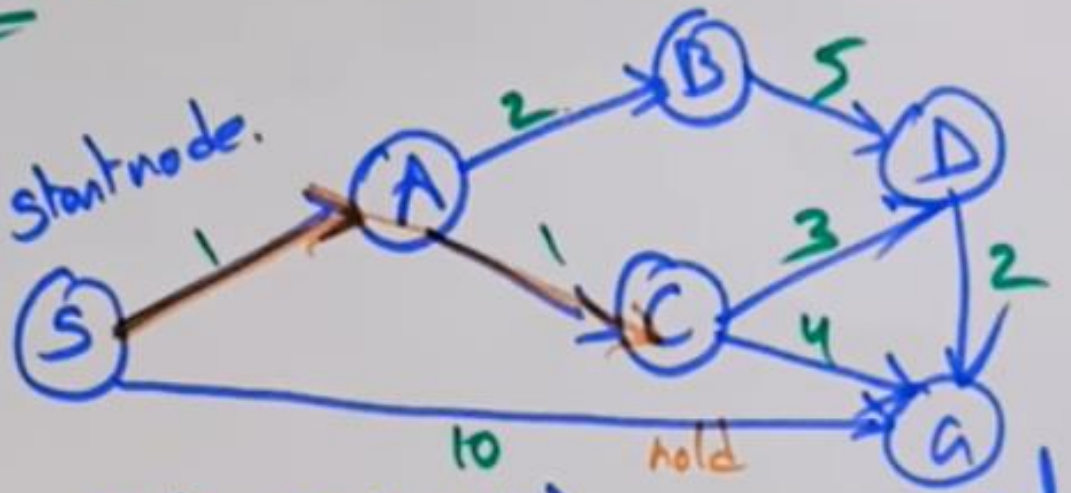
le :



① $S \rightarrow A \Rightarrow f(n) = g(n) + h(n)$

$= 1 + 3 = 4$ ✓

$S \rightarrow G = f(n) = 10 + 0 = 10$ hold



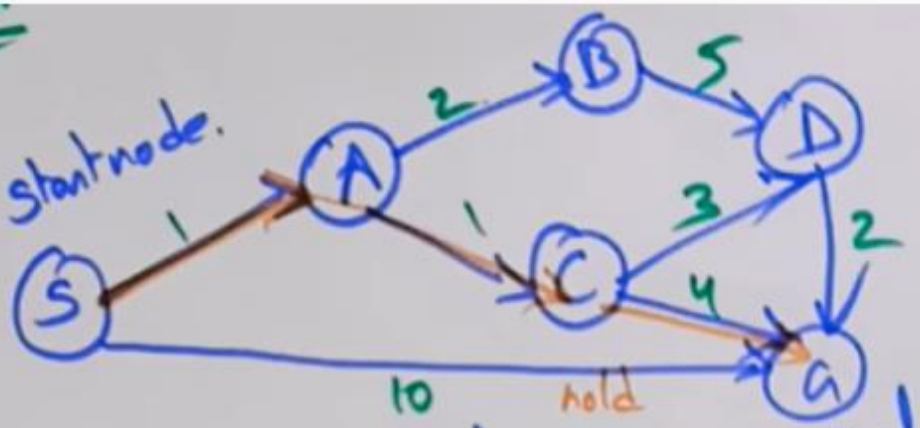
① $S \rightarrow A \Rightarrow f(n) = g(n) + h(n)$

$= 1 + 3 = 4$ ✓

$S \rightarrow G = f(n) = 10 + 0 = 10$ hold

② $S \rightarrow A \rightarrow B \Rightarrow f(n) = 3 + 4 = 7$ hold

$S \rightarrow A \rightarrow C \Rightarrow f(n) = 2 + 2 = 4$ ✓



① $S \to A \Rightarrow f(n) = g(n) + h(n)$
$= 1 + 3 = 4$ ✓

$S \to G = f(n) = 10 + 0 = 10$ hold ✗

② $S \to A \to B \Rightarrow f(n) = 3 + 4 = 7$ hold ✗

$S \to A \to C \Rightarrow f(n) = 2 + 2 = 4$ ✓

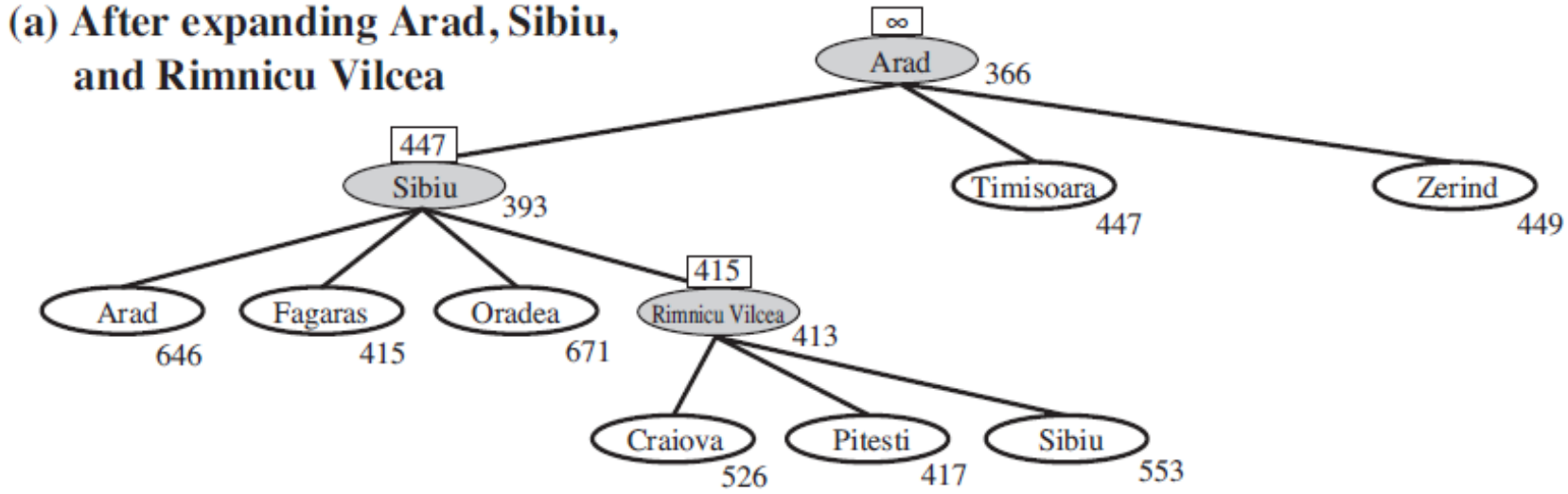
③ $S \to A \to C \to D \Rightarrow f(n) = 5 + 6 = 11$ hold ✗

$S \to A \to C \to G \Rightarrow f(n) = 6 + 0 = 6$ ✓

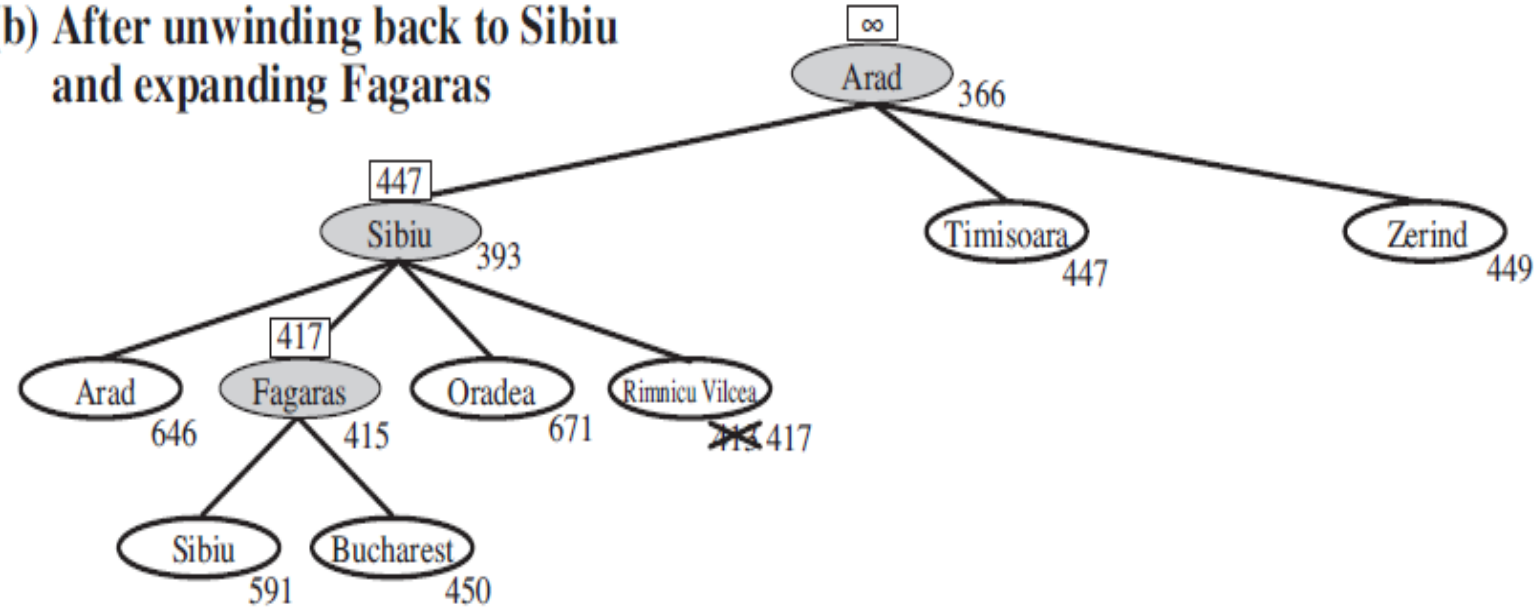
$S \to A \to C \to$ Cost = 6

Recursive best-first search (RBFS)

- Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it keeps 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 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 re-expanding the subtree at some later time.


(a) After expanding Arad, Sibiu, and Rimnicu Vilcea
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
Fagaras
415
Oradea
671
415
Rimnicu Vilcea
413
Craiova
526
Pitesti
417
Sibiu
553

(b) After unwinding back to Sibiu and expanding Fagaras



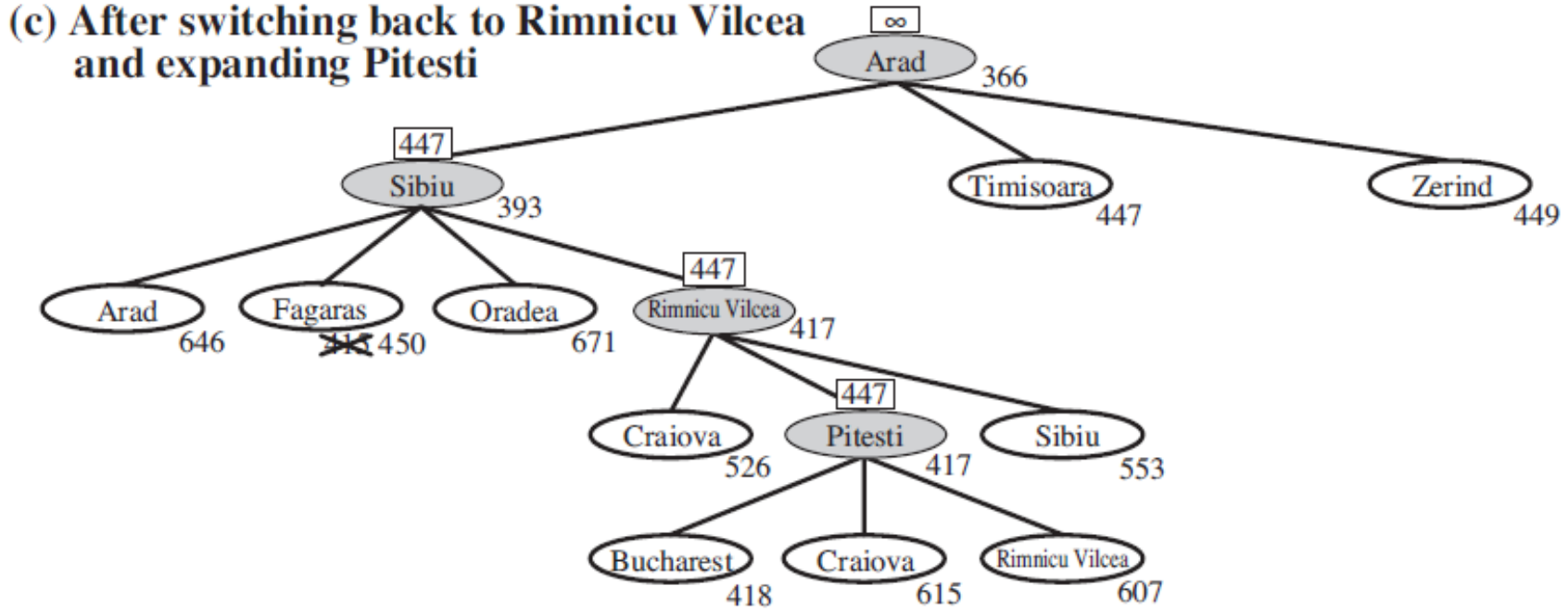


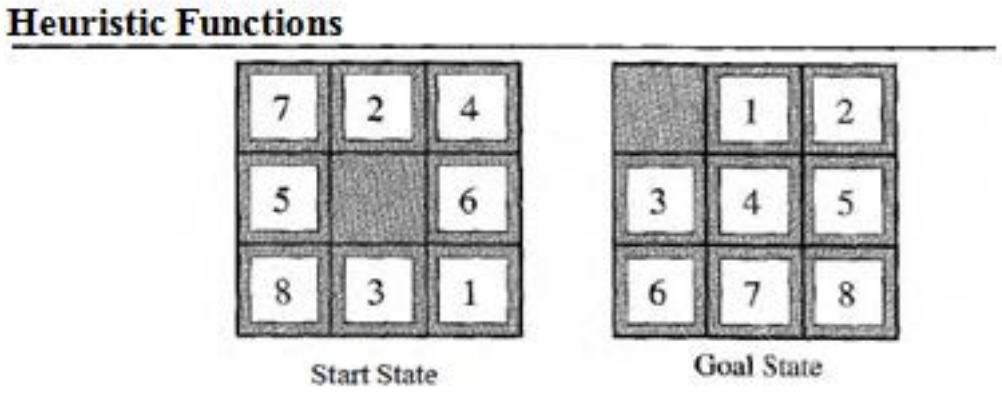
Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

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

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        iff [best] > f-limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f-limit, alternative))
        if result ≠ failure then return result
```

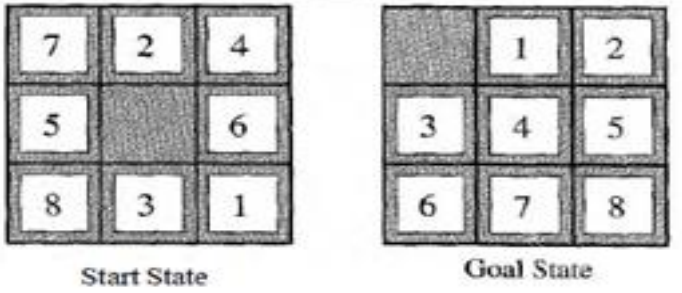
Figure 4.5 The algorithm for recursive best-first search.

Heuristic Functions



- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3. This means that an exhaustive search to depth 22 would look at about $\mathbf{3^{22}}$ = 3.1 x 10^{10} states.
- By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only 9!/2 = 181,440 distinct states that are reachable.

Heuristic Functions



- *hl* = the number of misplaced tiles.
- so the start state would have *hl* = 8.
- *hl* 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.
- Manhattan distance of *h2=3+l+2+2+2+3+3+2=18.*
- As we would hope, neither of these overestimates the true solution cost, which is 26.

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- The search algorithms that we have seen so far are designed to explore search spaces systematically.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not.
- When a goal is found, the *path* to that goal also constitutes a solution to the problem.

- In many problems, however, the path to the goal is irrelevant.
- For example, in the 8-queens problem what matters is the final configuration of queens, not the order in which they are added.
- This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.
- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state.

- Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages:

(1) they use very little memory-usually a constant amount;

(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

- In addition to finding goals, local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.**

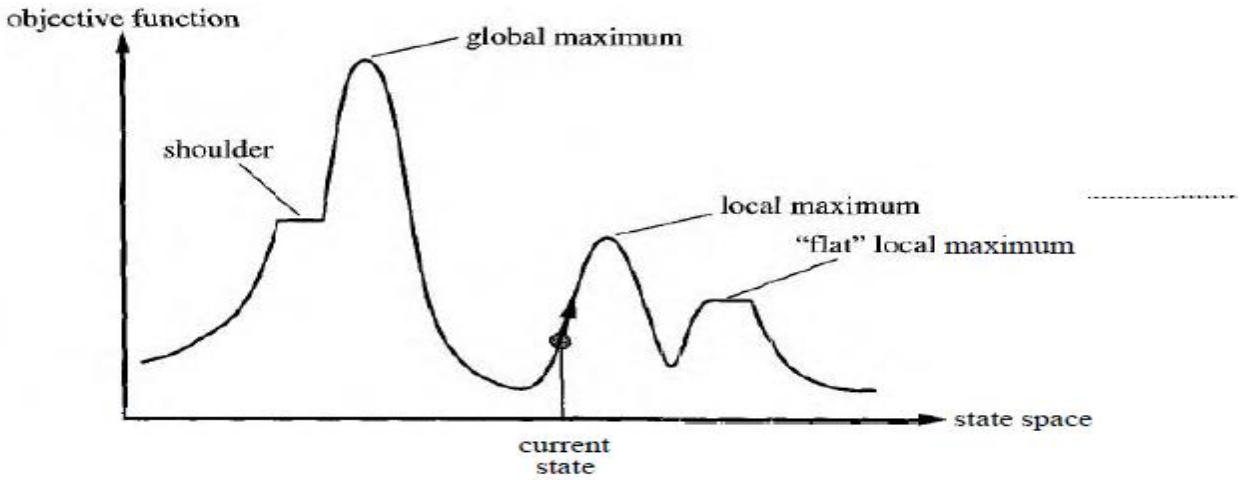


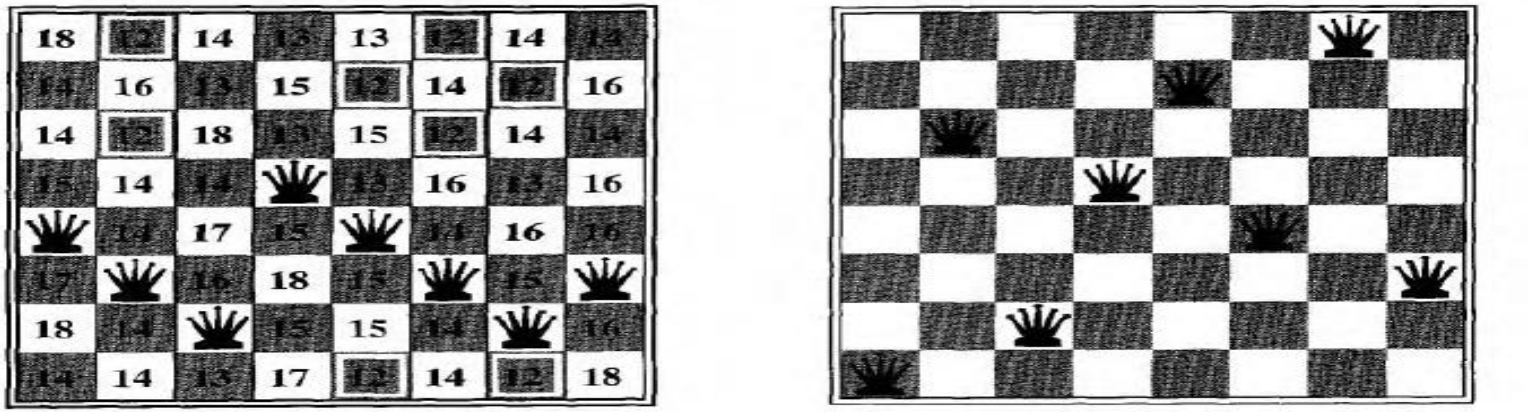
Figure 4.10 A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

- consider the **state space land scape**
- If elevation corresponds to cost, then the aim is to find the lowest valley-a **global minimum;**
- if elevation corresponds to an objective function, then the aim is to find the highest peak-a **global maximum,.**
- Local search algorithms explore this landscape. A **complete,** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a, global minimum/maximum.

Hill-climbing search

- The **hill-climbing** search algorithm -simply a loop that continually moves in the direction of increasing value-that is, uphill. It terminates when it reaches a "top" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value.
- Hill-climbing does not look ahead beyond the immediate neighbors of the current state.
- To illustrate hill-climbing, we will use the **8-queens problem**. Local-search algorithms typically use a **complete-state formulation,** where each state has 8 queens on the board, one per column.
- The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.

- The global minimum of this function is zero, which occurs only at perfect solutions.
- Figure 4.12(a) shows a state with h = 17.
- The figure also shows the values of all its successors, with the best successors having h = 12.
- Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.



(a) (b)

Figure 4.12 (a) An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     neighbor ← a highest-valued successor of current
     if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
     current ← neighbor
```

Figure 4.11 The hill-climbing search algorithm (**steepest ascent** version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Figure 4.13 Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

- In each case, the algorithm reaches a point at which no progress is being made. Starting from
- a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck-not bad for a state space with 8^8 = 17 million states.
- The algorithm in Figure 4.11 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going-to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4. 15?
- The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur when ever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sidleways moves allowed.

- For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill-climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.
- Variants :
  - **Stochastic hill climbing** chooses at random from among the uphill moves;
  - the probability of selection can vary with the steepness of the uphill move.
  - This usually converges more slowly than steepest ascent, but in some state landscapes it finds better solutions.
  - **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
  - This is a good strategy when a state has many (e.g., thousands) of successors.
  - **Random-restart hill climbing** adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial state, stopping when a goal is found.
  - It is complete with probability approaching 1, for the trivial reason that it will eventually generate a goal state as the initial state
  - if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.
  - a reasonably good local maximum can often be found after a small number of restarts.

- hill-climbing that *never makes "downhill" moves to lower value* (or higher cost) -incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—(moving to a successor chosen uniformly at random from the set of successors)—is complete but extremely inefficient.
- Therefore, reasonable to try to combine hill climbing + random walk =both efficiency and completeness.

Local beam search

- (problem of memory limitations)
- The **local beam search algorithm keeps track of k states rather than** just one.
- It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts.
- Otherwise, it selects the k best successors from the complete list and repeats.
- may be looks like the same in parallel, but diffr
- generate the best successors say to the others, “Come over here, the grass is greener!”

- they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing
- **STOCHASTIC BEAM analogous to stochastic hill climbing, helps alleviate this problem**
- Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic algorithms

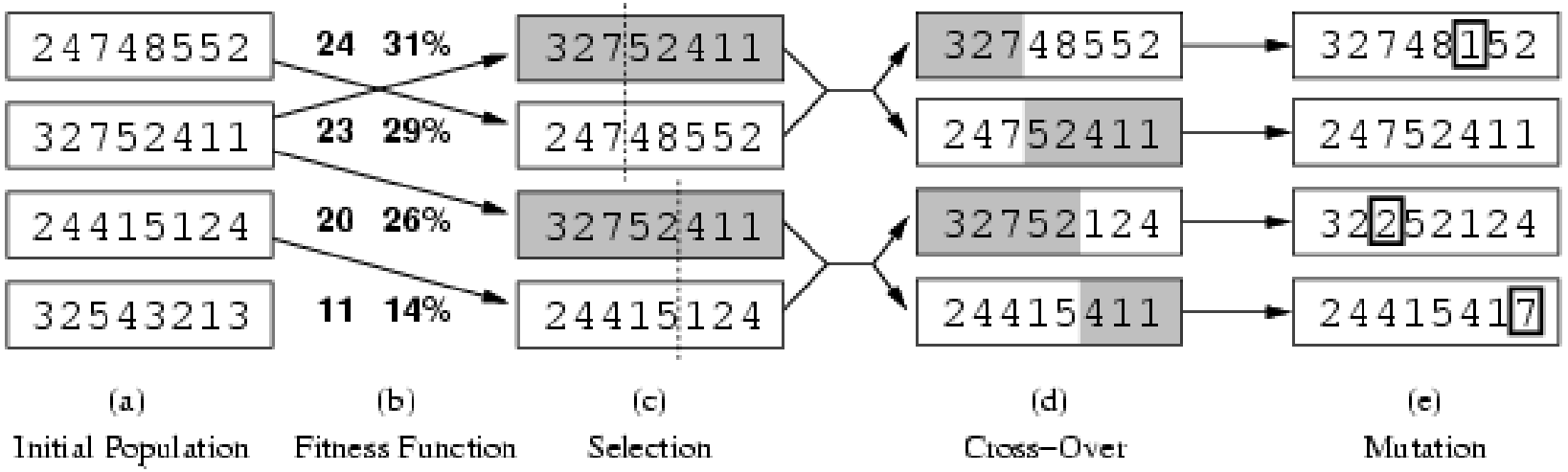
- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)

- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)

- Evaluation function (fitness function). Higher values for better states.

- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

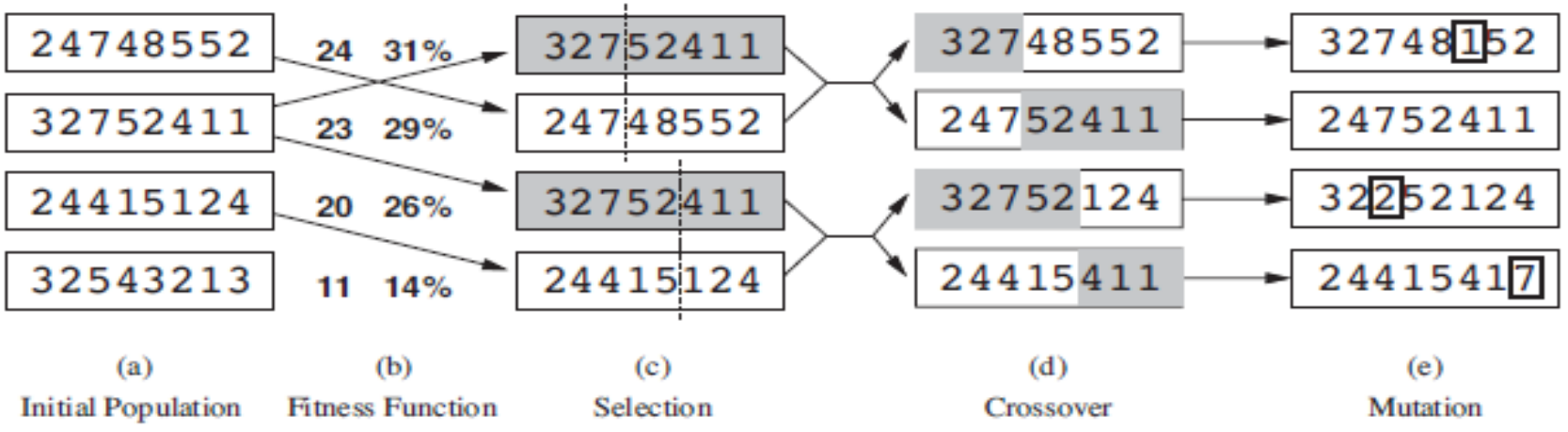


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

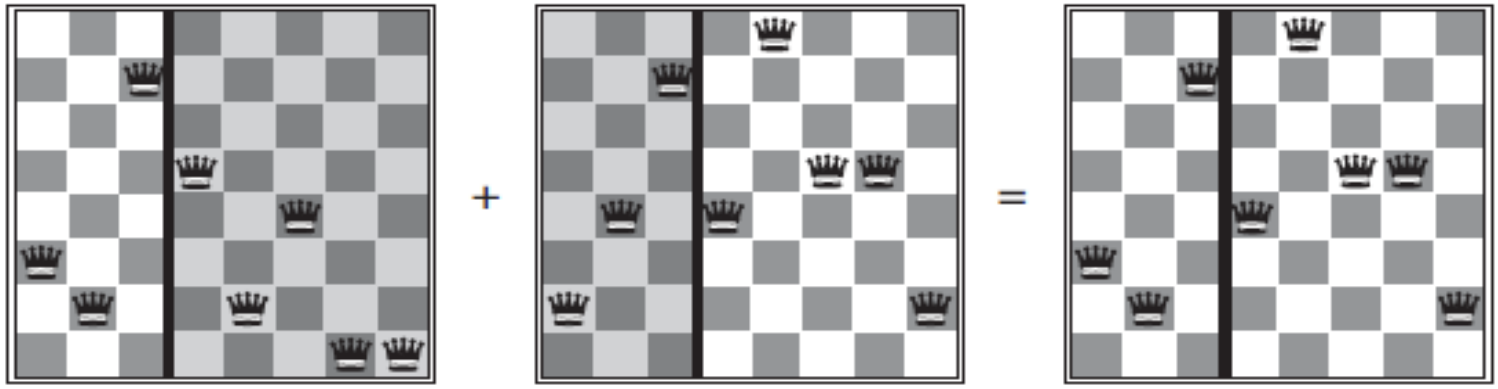


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.