Hill Climbing (1)

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. E.g., Traveling-salesman problem in which we need to minimize the distance traveled by the salesman.

- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.

- It is mostly used when a good heuristic is available. A heuristic function is a function that will rank all the possible alternatives at any branching step in the search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Hill Climbing (2)

- Following are some main features of Hill Climbing Algorithm:
  - Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
  - Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
  - No backtracking: It does not backtrack the search space, as it does not remember the previous states.
- Advantages:
  - Hill Climbing is a simple and intuitive algorithm that is easy to understand and implement.
  - It can be used in a wide variety of optimization problems, including those with a large search space and complex constraints.
  - It is often very efficient in finding local optima, making it a good choice for problems where a good solution is needed quickly.
  - The algorithm can be easily modified and extended to include additional heuristics or constraints.
- Disadvantages:
  - Hill Climbing can get stuck in local optima, meaning that it may not find the global optimum of the problem.
  - It is sensitive to the choice of initial solution, and a poor initial solution may result in a poor final solution.
  - It does not explore the search space very thoroughly, which can limit its ability to find better solutions.
  - It may be less effective than other optimization algorithms, such as genetic algorithms or simulated annealing, for certain types of problems.

Hill Climbing (3)

- State-space Diagram for Hill Climbing:
  - The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
  - On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

- Different regions in the State-space Diagram for Hill Climbing:
  - Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
  - Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
  - Current state: It is a state in a landscape diagram where an agent is currently present.
  - Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.
  - Shoulder: It is a plateau region which has an uphill edge.

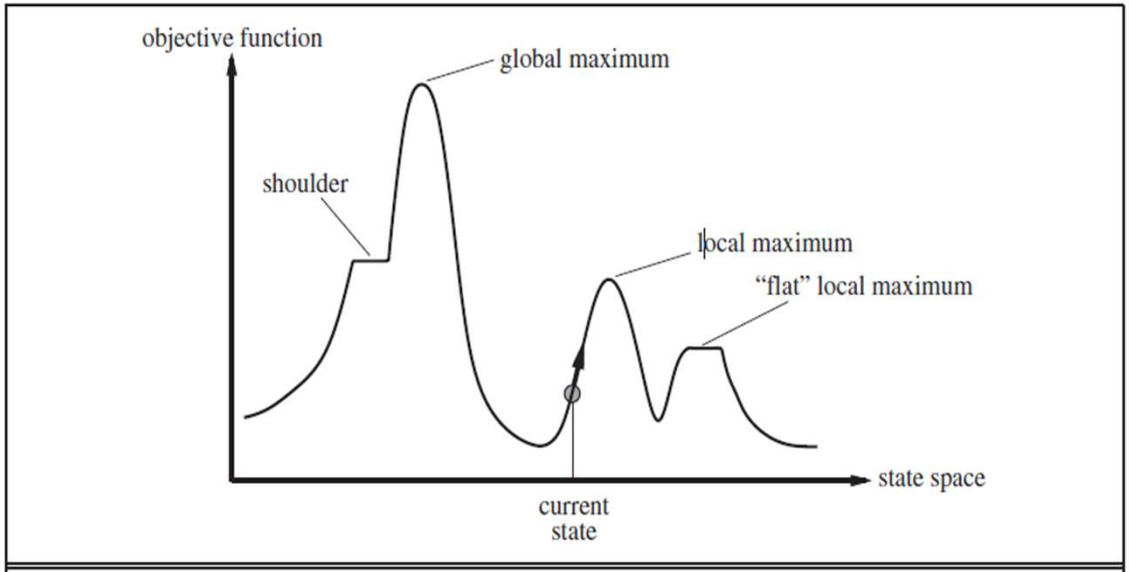


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill Climbing (4)

- Basic Steps of the hill climbing search:

```
Form a one-element queue consisting of a zero-length path that contains only the root node.
Repeat
    Remove the first path from the queue; create new paths by extending the first path to
    all the neighbors of the terminal node.
    Reject all new paths with loops.
    Sort the new paths, if any, by the estimated distances between their terminal nodes and
    the goal.
Until the first path in the queue terminates at the goal node or the queue is empty
If the goal node is found, announce success, otherwise announce failure
```

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

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

Figure 4.2 The hill-climbing search algorithm, 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.

Hill Climbing (5)

- **Types of Hill Climbing Algorithm:** 1) Simple, 2) Steepest-Ascent, and 3) Stochastic hill climbing algorithm

1. Simple hill climbing:

- It is the simplest hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
- This algorithm is less time consuming.
- It generates less optimal solution, and the solution is not guaranteed.
- Algorithm for simple hill climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

Step 4: Check new state:

If it is goal state, then return success and quit.

Else if it is better than the current state then assign new state as a current state.

Else if not better than the current state, then return to step2.

Step 5: Exit.

Hill Climbing (6)

2. Steepest-Ascent hill-climbing:

- It is a variation of simple hill climbing algorithm. It examines all the neighboring nodes of the current state and selects one neighbor node which is the closest to the goal state.
- This algorithm consumes more time as it searches for multiple neighbors
- Algorithm for steepest-ascent hill climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and stop,

else make current state as initial state.

Step 2: Loop until a solution is found or the current state does not change.

1. Let SUCC be a state such that any successor of the current state will be better than it.
2. For each operator that applies to the current state:
   a) Apply the new operator and generate a new state.
   b) Evaluate the new state.
   c) If it is goal state, then return it and quit, else compare it to the SUCC.
   d) If it is better than SUCC, then set new state as SUCC.
   e) If the SUCC is better than the current state, then set current state to SUCC.

Step 3: Exit.

Hill Climbing (7)

3. Stochastic hill climbing:

- It does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides (based on the amount of improvement in that neighbor) whether to choose it as a current state or examine another state.
- Algorithm for stochastic hill climbing:

Step 1: Evaluate the initial state. If it is a goal state, then stop and return success.

Otherwise, make the initial state the current state.

Step 2: Repeat these steps until a solution is found or the current state does not change.

1. Select a state that has not been yet applied to the current state.
2. Apply the successor function to the current state and generate all the neighbor states.
3. Among the generated neighbor states which are better than the current state choose a state randomly (or based on some probability function).
4. If the chosen state is the goal state, then return success, else make it the current state and repeat step 2 of the second point.

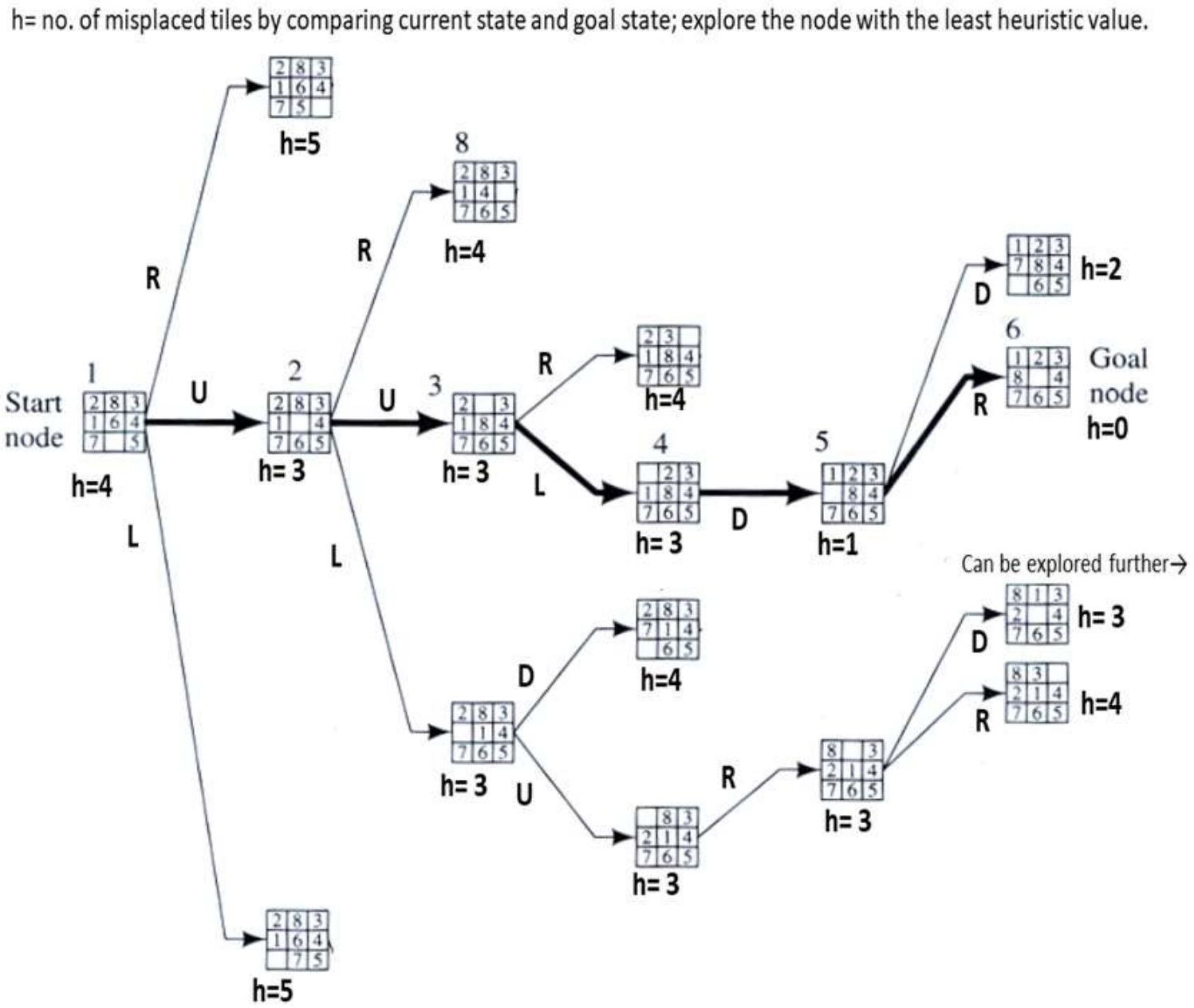
Step 5: Exit.

Hill Climbing (8)

- Problems of hill climbing: Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :
- Local maximum: At a local maximum all neighboring states have a value that is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
  - To overcome the local maximum problem: Utilize the backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
- Plateau: On the plateau, all neighbors have the same value. Hence, it is not possible to select the best direction.
  - To overcome plateaus: Make a big jump. Randomly select a state far away from the current state. Chances are that we will land in a non-plateau region.
- Ridge: Any point on a ridge can look like a peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
  - To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Hill Climbing (3)

- **Steepest-Ascent Hill Climbing for 8-Puzzle Problem:**
- Heuristics h= no. of misplaced tiles by comparing current state and goal state;
- Explore the node with the least heuristic value.



Best-First Search

Source:

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.
2. Elaine Rich and Kevin Knight, "Artificial Intelligence" Tata McGraw Hill.

Best-First Search (1)

- Best-first search is Greedy algorithm. It always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms.
- With the help of best-first search, at each step, we can choose the most promising node.
- In the best-first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e., f(n)= h(n) where, h(n)= estimated cost of the cheapest path from the state at node n to a goal state.
- The Greedy best-first algorithm is implemented by the priority queue. We use a priority queue to store costs of nodes. So, the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.
- Advantages:
  - Best-first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
  - This algorithm is more efficient than BFS and DFS algorithms.
- Disadvantages:
  - It can behave as an unguided depth-first search in the worst-case scenario.
  - It can get stuck in a loop as DFS.
  - This algorithm is not optimal.

Best-First Search (2)

- Best-First Search Algorithm:
  - **Step 1:** Place the starting node into the OPEN list.
  - **Step 2:** If the OPEN list is empty, Stop and return failure.
  - **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
  - **Step 4:** Expand the node n, and generate the successors of node n.
  - **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
  - **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
  - **Step 7:** Return to Step 2.

- Pseudocode for Best-First Search Algorithm:

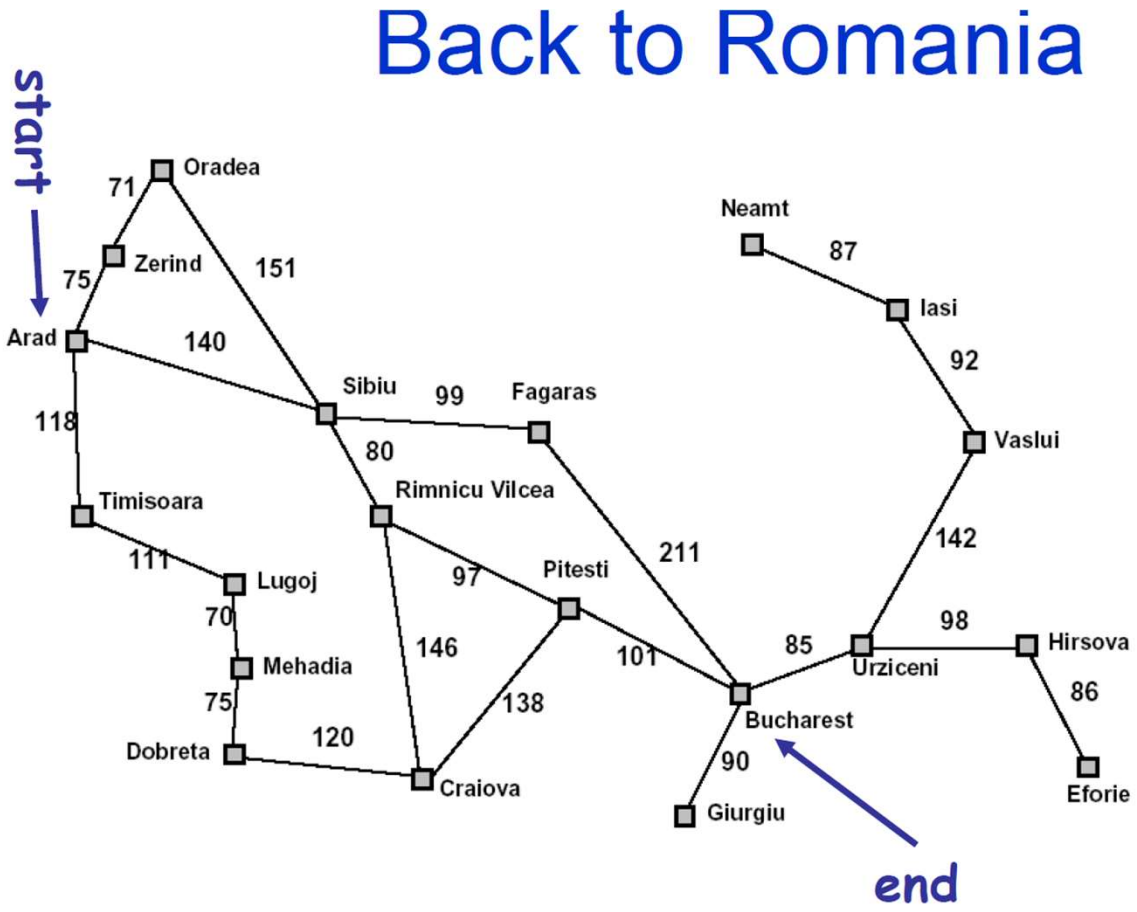
```
Best-First-Search(Graph g, Node start)
  1) Create an empty PriorityQueue
    PriorityQueue pq;
  2) Insert "start" in pq.
    pq.insert(start)
  3) Until PriorityQueue is empty
      u = PriorityQueue.DeleteMin
      If u is the goal
        Exit
      Else
        Foreach neighbor v of u
          If v "Unvisited"
            Mark v "Visited"
            pq.insert(v)
        Mark u "Examined"
End procedure
```

Best-First Search (3)

- **Best-fit Search Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - m is the maximum depth of any node, and m >> d (Shallowest solution depth)
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + \ldots. b^m = O(b^m)$
- Time Complexity:
  - In the worst case, if the shallowest solution is the last node generated at depth m, then the total number of nodes generated is $O(b^m)$ and hence time complexity of best-fit search will be $O(b^m)$.
- Space Complexity:
  - The worst-case space complexity of Greedy best-first search is $O(b^m)$ where, m is the maximum depth of the search space.
- Completeness: Greedy best-first search is also incomplete, even if the given state space is finite.
- Optimal: This algorithm is not optimal.

Best-First Search (4)

- **Example: Route Finding-Romania City Map**
  - What's the real shortest path from Arad to Bucharest?
  - What's the distance on that path?



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-First Search (5)

- **Example: Route Finding-Romania City Map (cont..)**
- **Greedy Best-First solution:**

Arad→ Sibiu → Fagaras → Bucharest

Distance= 140+99+211=450

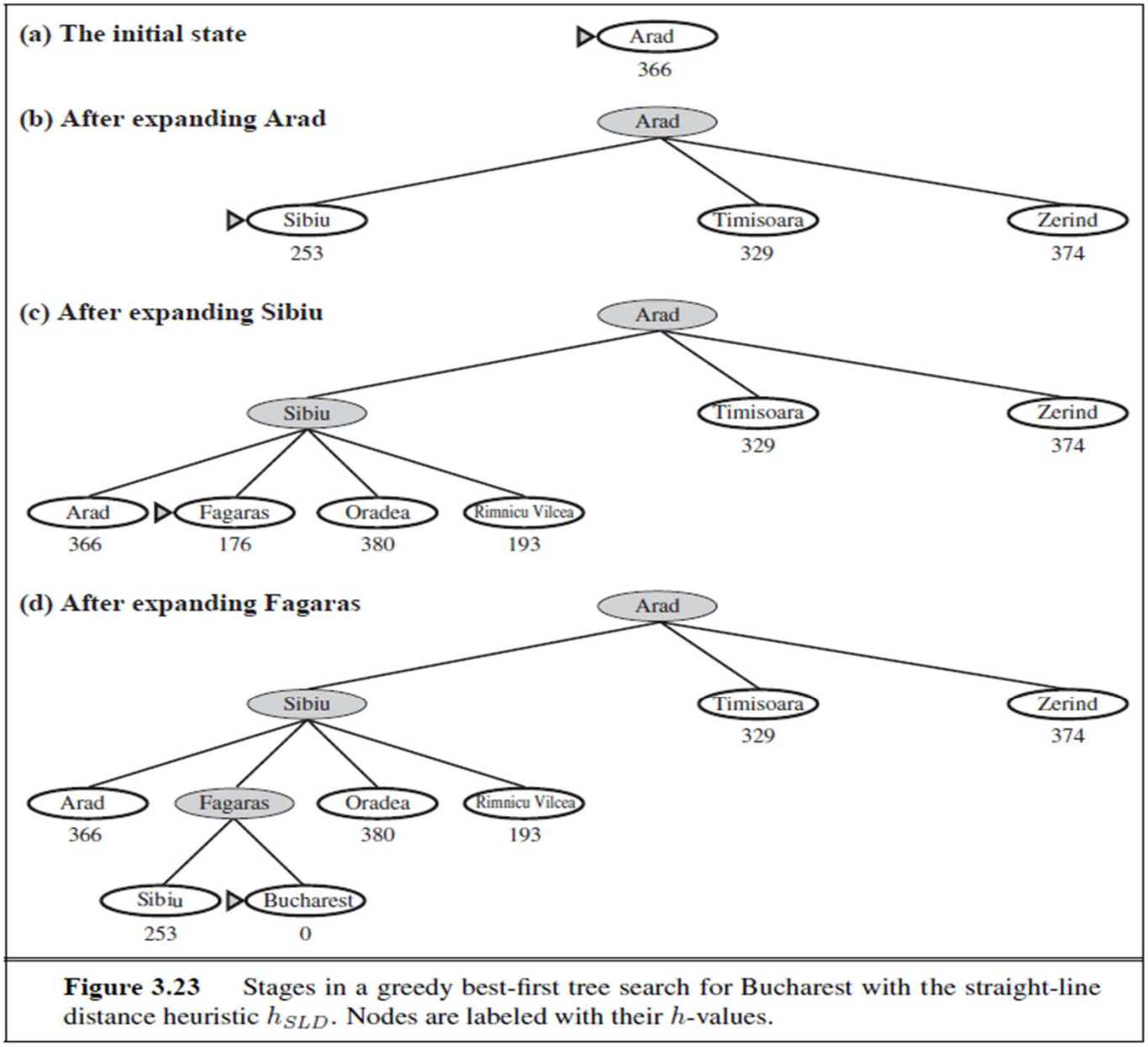


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* Algorithm

Source:

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.

A* Algorithm (1)

- The most widely known form of best-first search is called A* search (pronounced "A-star search"). It evaluates nodes by combining g(n), the cost to reach the node, 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, 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.
- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n).

- Advantages:
  - A* search algorithm is the best algorithm than other search algorithms.
  - A* search algorithm is optimal and complete.
  - This algorithm can solve very complex problems.

A* Algorithm (2)

- Disadvantages:
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - A* search algorithm has some complexity issues.
  - The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

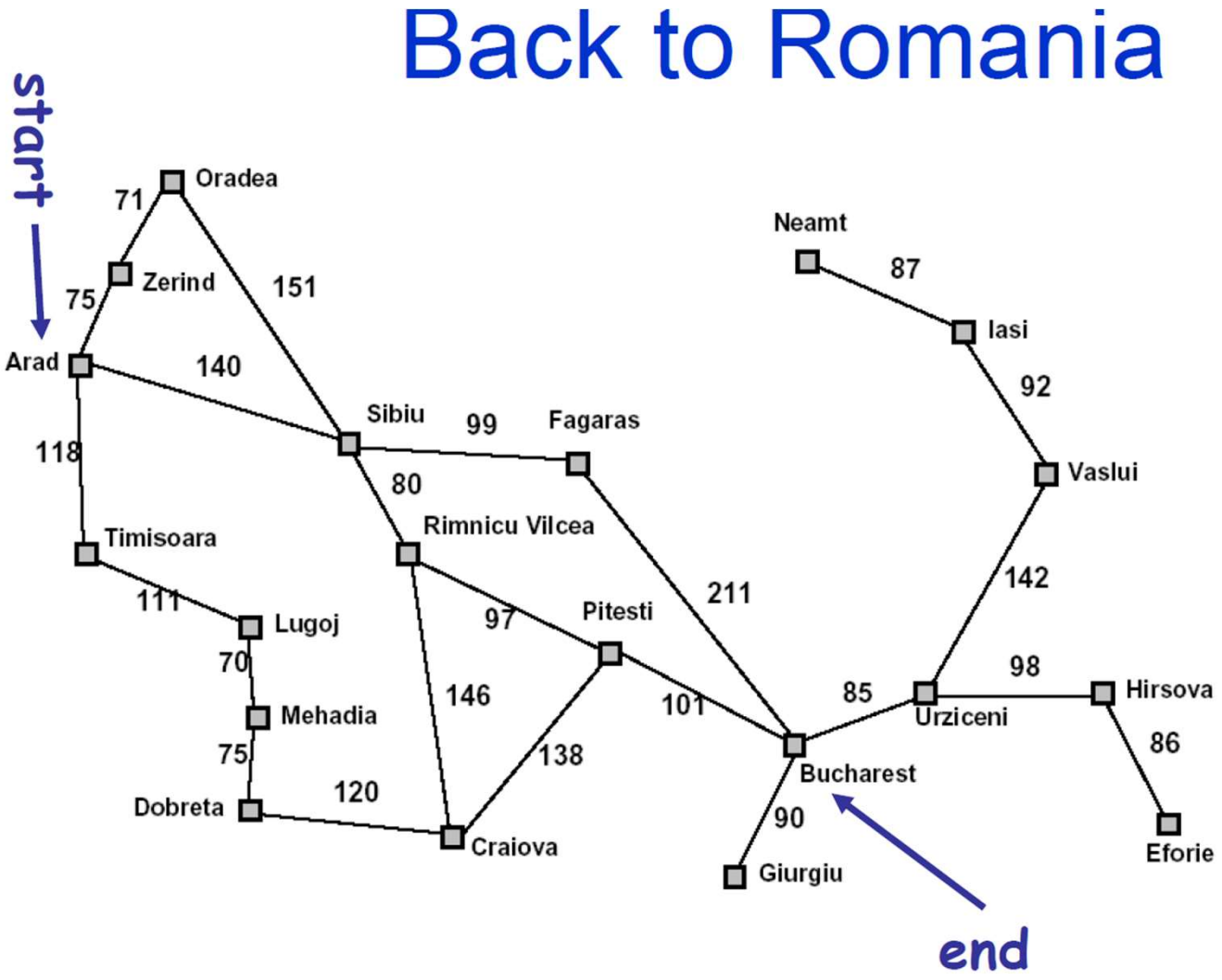
- Algorithm of A* search:
- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node, then return success and stop, otherwise
- **Step 4:** Expand node n and generate all of its successors and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to Step 2.

A* Algorithm (3)

- **A* Algorithm Properties:**
- Cartoon of search tree has:
  - b is the branching factor
  - m is the maximum depth of any node, and m >> d (Shallowest solution depth)
  - Number of nodes in entire tree = $b^0 + b^1 + b^2 + \ldots. b^m = O(b^m)$
- Time Complexity:
  - The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So, in the worst case, if the shallowest solution is the last node generated at depth m, then the total number of nodes generated is $O(b^m)$ and hence time complexity of A* algorithm will be $O(b^m)$.
- Space Complexity:
  - The worst-case space complexity of A* algorithm is $O(b^m)$ where, m is the maximum depth of the search space.
- Completeness: A* algorithm is complete as long as: 1) Branching factor is finite, and 2)Cost at every action is fixed.
- Optimal: A* search algorithm is optimal if it follows below two conditions:
  - **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
  - **Consistency:** Second required condition is consistency for only A* graph-search. If the heuristic function is admissible, then A* tree search will always find the least cost path.

A* Algorithm (4)

- **Example: Route Finding-Romania City Map**



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Algorithm (5)

- **Example: Route Finding-Romania City Map (cont..)**

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* solution:**

Arad→ Sibiu → Rimnicu Vilcea → Pitesti → Bucharest

Distance= 140+80+97+101=418

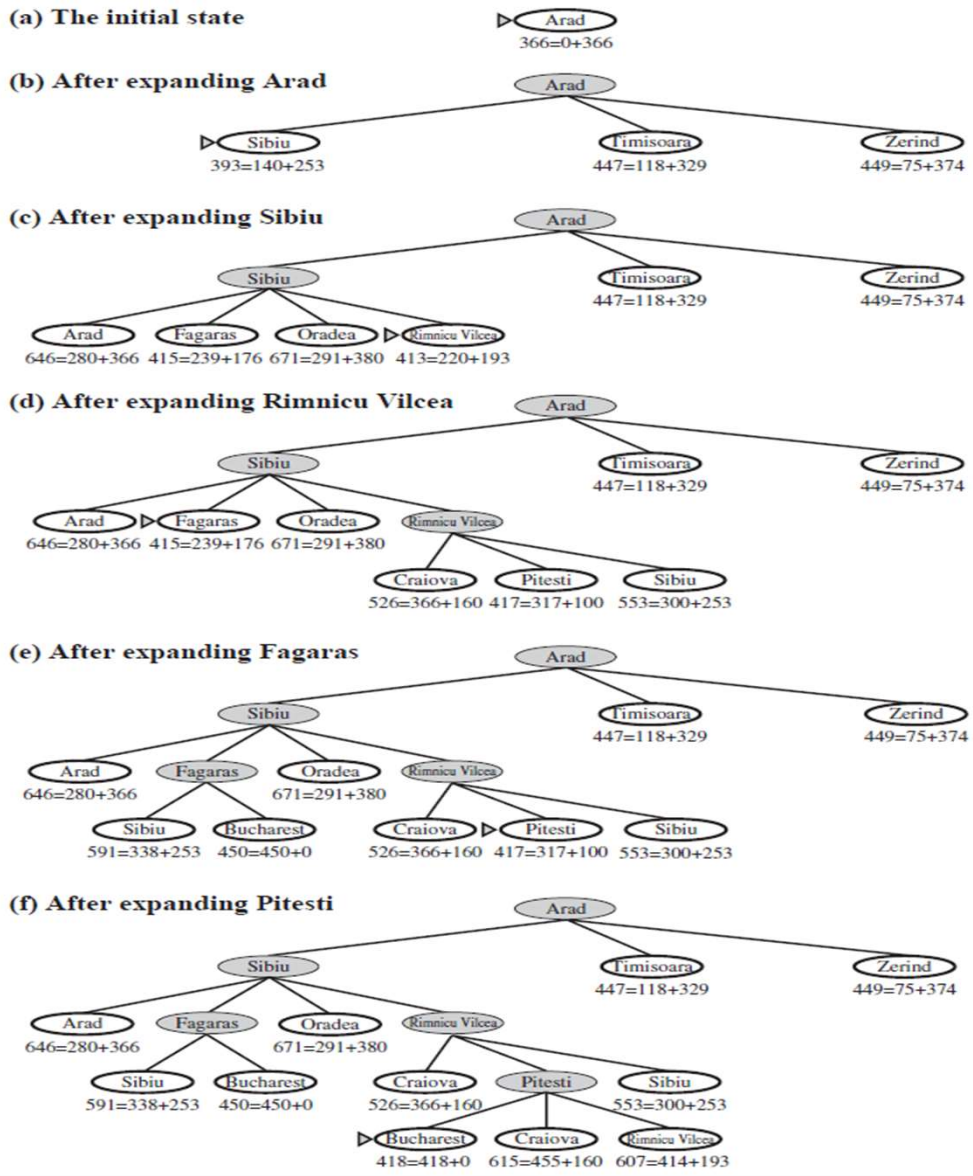


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

A* Algorithm (6)

- **Example: 8-Puzzle Problem**

Initial State

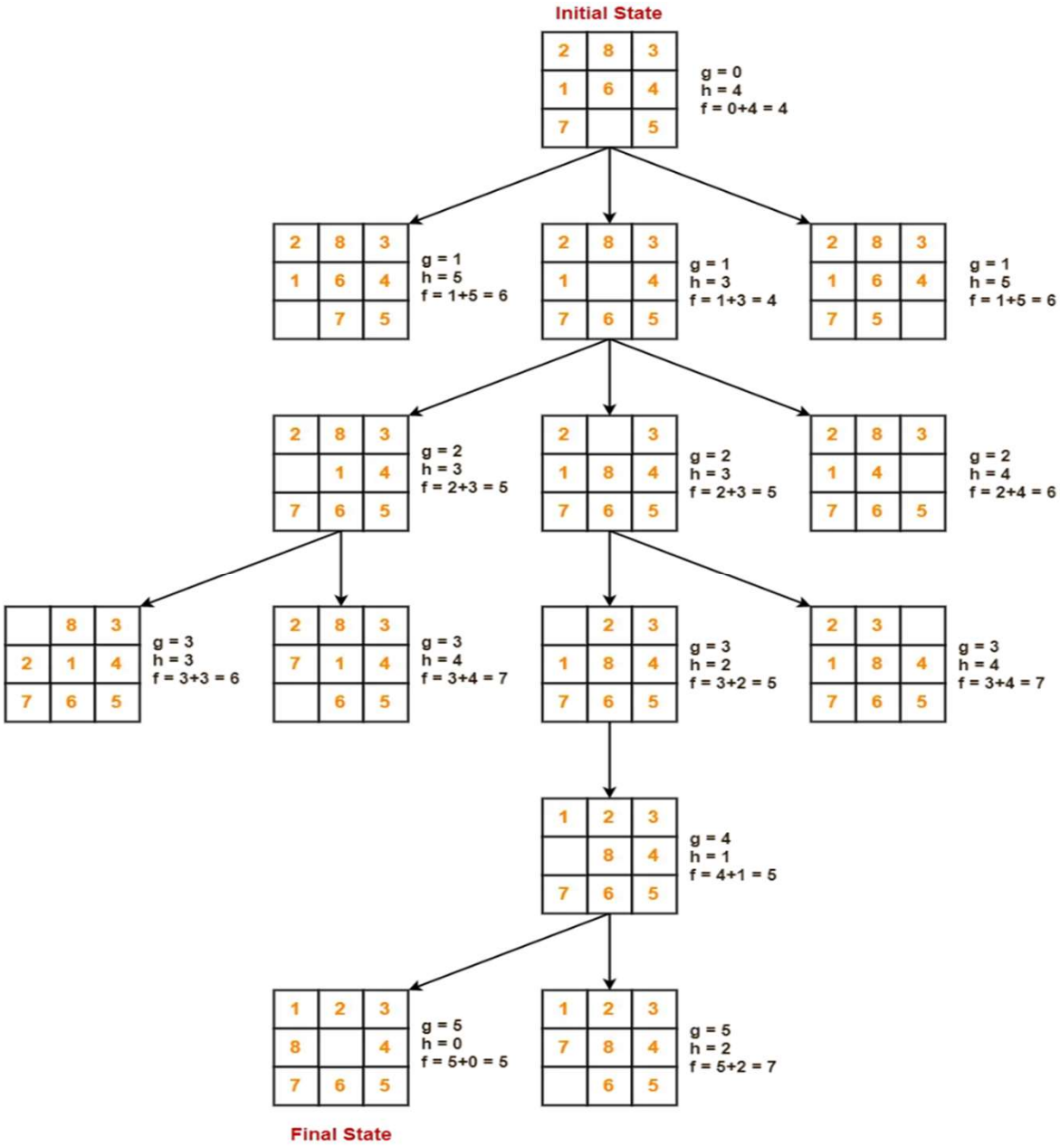
Final State

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

where

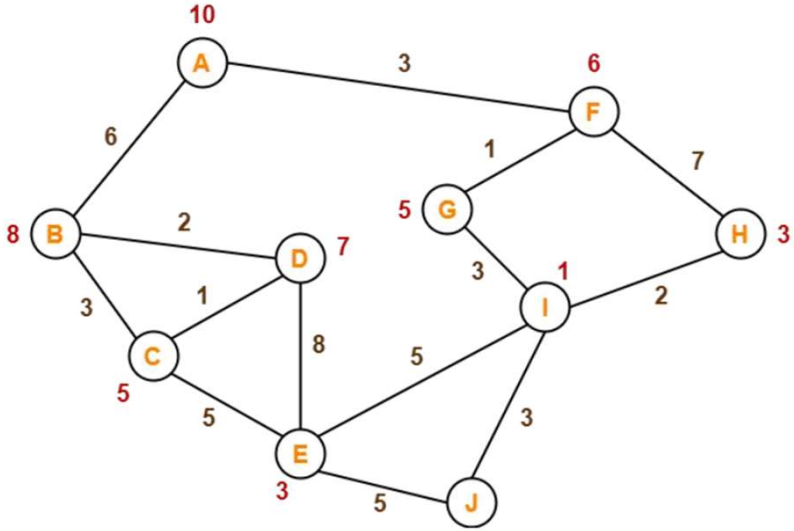
g(n) = Depth of node and

h(n) = Number of misplaced tiles.



A* Algorithm (7)

- **Example: Route Finding**
- Consider the following graph below. Find the most cost-effective path to reach from start state A to final state J using A* Algorithm. The numbers written on edges represent the distance between the nodes. The numbers written on nodes represent the heuristic value.



- **Solution:**