Unit-III
Informed Search Methods

-Dr. Radhika V. Kulkarni

Associate Professor, Dept. of Computer Engineering,

Vishwakarma Institute of Technology, Pune.

Sources:

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill
4. Saroj Kaushik, “Artificial Intelligence”, Cengage Publication.
5. https://www.javatpoint.com/digital-image-processing-tutorial

DISCLAIMER

This presentation is created as a reference material for the students of TY-CS, VIT (AY 2023-24 Sem-1).

It is restricted only for the internal use and any circulation is strictly prohibited.

Syllabus

Unit-III Informed Search Methods

Generate & test, Hill Climbing, Best First Search, A* and AO* Algorithm, Constraint satisfaction, Means Ends Analysis, Game playing: Minimax Search, Alpha-Beta Cut offs, Waiting for Quiescence.

Informed Search Algorithms

- These algorithms apply the strategies that know whether one non-goal state is “more promising” than another. They are also called heuristic search strategies.
- They can do quite well given some guidance on where to look for solutions.
- E.g., Hill Climbing, Best First Search, A* and AO* Algorithm, Constraint satisfaction, Means Ends Analysis, Minimax Search, Alpha-Beta Cut offs etc.
- Key features of informed search algorithms in AI:
  - Use of Heuristics – informed search algorithms use heuristics, or additional information, to guide the search process and prioritize which nodes to expand.
  - More efficient – informed search algorithms are designed to be more efficient than uninformed search algorithms, such as breadth-first search or depth-first search, by avoiding the exploration of unlikely paths and focusing on more promising ones.
  - Goal-directed – informed search algorithms are goal-directed, meaning that they are designed to find a solution to a specific problem.
  - Cost-based – informed search algorithms often use cost-based estimates to evaluate nodes, such as the estimated cost to reach the goal or the cost of a particular path.
  - Prioritization – informed search algorithms prioritize which nodes to expand based on the additional information available, often leading to more efficient problem-solving.
  - Optimality – informed search algorithms may guarantee an optimal solution if the heuristics used are admissible (never overestimating the actual cost) and consistent (the estimated cost is a lower bound on the actual cost).

Generate and Test Search

Sources:

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.

Generate and Test Search(1)

- Generate and Test Search or Exhaustive Search proposes all possible solutions and then tests them for their feasibility (test whether the solutions satisfy all of the constraints).

- It is a heuristic search technique based on Depth First Search with Backtracking which guarantees to find a solution if done systematically and there exists a solution.

- It ensures that the best solution is checked against all possible generated solutions. The evaluation is carried out by the heuristic function.

- If there are some paths which are most unlikely to lead us to the goal state, then they are not considered.

- The heuristic does this by ranking all the alternatives and is often effective in doing so.

Generate and Test Search (2)

- Systematic generate and test may prove to be ineffective while solving complex problems.
- There is a technique to improve in complex cases as well by combining generate and test search with other techniques so as to reduce the search space. E.g., Constrain satisfaction techniques/ Bounding function.
- The exhaustive generate and test search algorithms are typically used to solve the problems with very small input size. But for certain finite size combinatorial problems, exhaustive search or its variant is the only way to find out the correct solution. E.g. 8-Queens problem.
- If each of the *n* variable domains has size *d*, then the solution space has d^n elements. If there are *c* number of constraints, the total number of constraints tested is $O(c\ d^n)$. As *n* becomes large, generate and test search very quickly becomes intractable.

Hill Climbing

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill

Local Search Algorithms

- Local search algorithm is a heuristic method for solving computationally hard optimization problems.
- Local search can be used on problems that can be formulated as finding a solution minimizing/maximizing a criterion among a number of candidate solutions.
- Local search algorithms move from solution to solution in the space of candidate solutions (the *search space*) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.
- They operate using a single current node (rather than multiple paths) and generally move only to neighbours of that node.
- Typically, the path followed by local search are not retained.
- Key advantages of local searches:
  - They use a very less amount of memory- a constant amount of time.
  - They can often find a reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.
- A complete local search algorithm always finds a goal if one exists.
- A local search algorithms are useful for solving optimization problems. An optimal algorithm always finds a global minimum or maximum.

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.

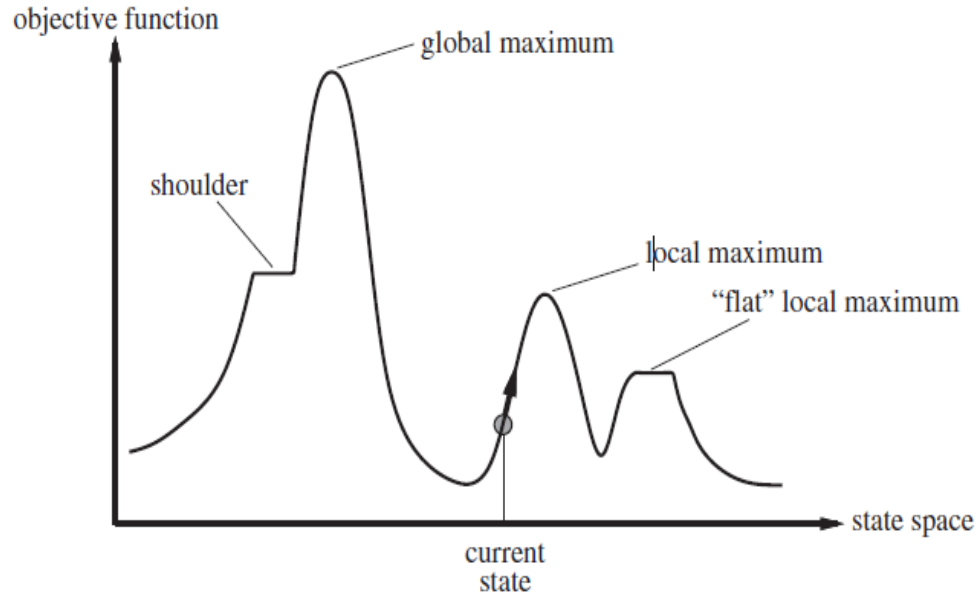


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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.

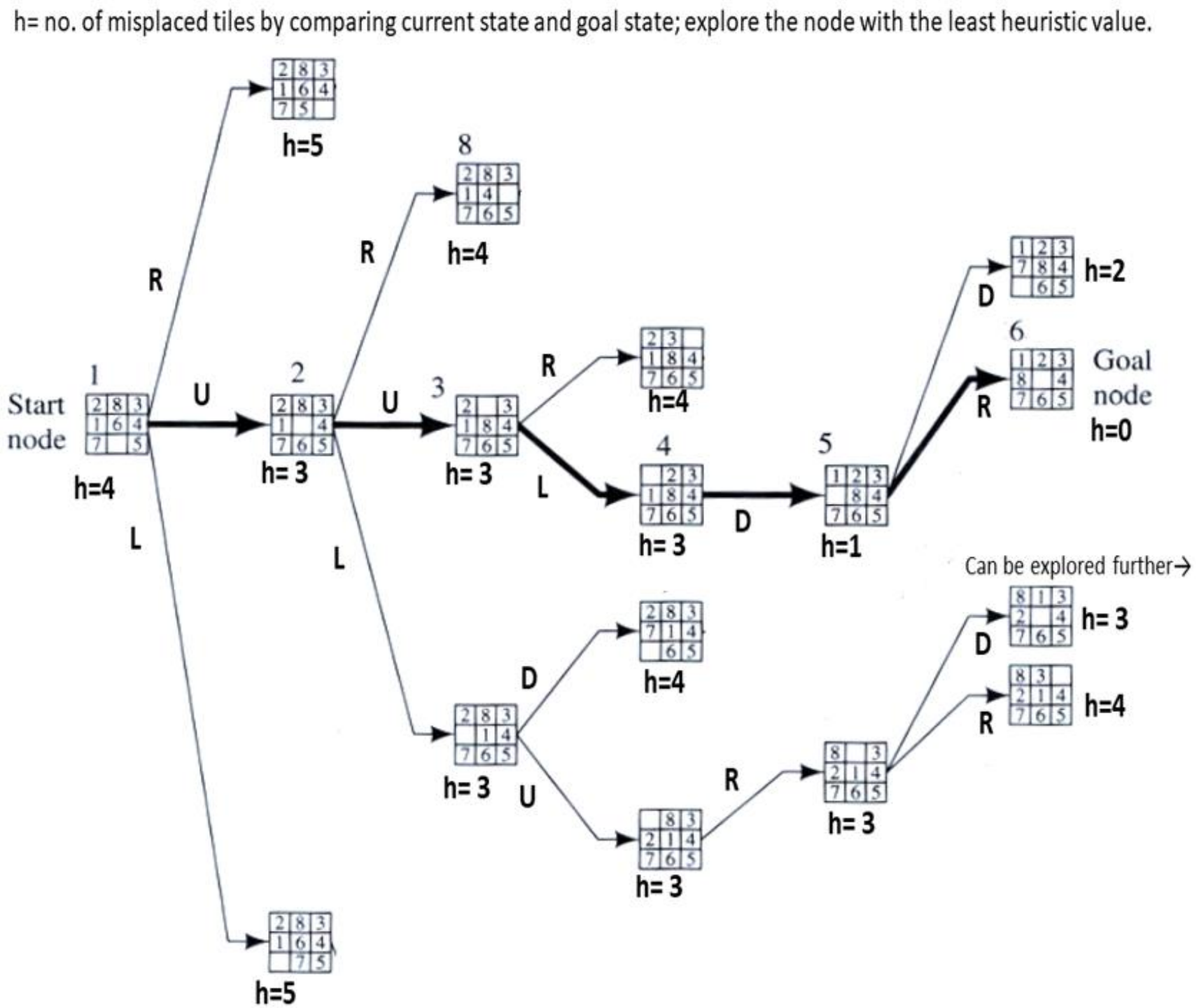
Step 5: Exit.

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, 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:

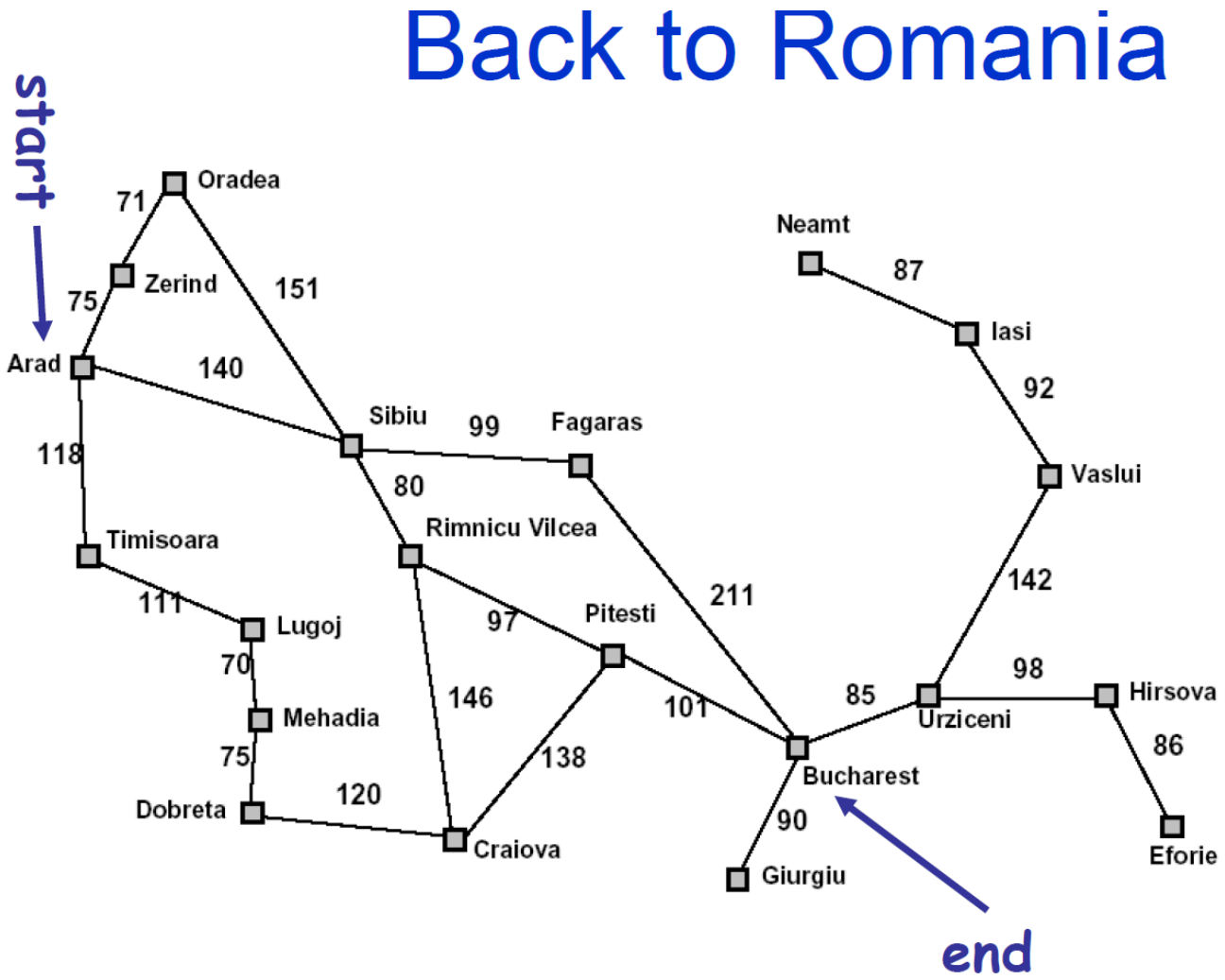
```
Best-First-Search(Graph g, Node start)
  1) Create an empty PriorityQueue
    PriorityQueue pq;
  2) Insert "start" in pq.
    pq.insert(start)
  3) Until PriorityQueue is empty
       u = PriorityQueue.DeleteMin
       If u is the goal
         Exit
       Else
         Foreach neighbor v of u
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

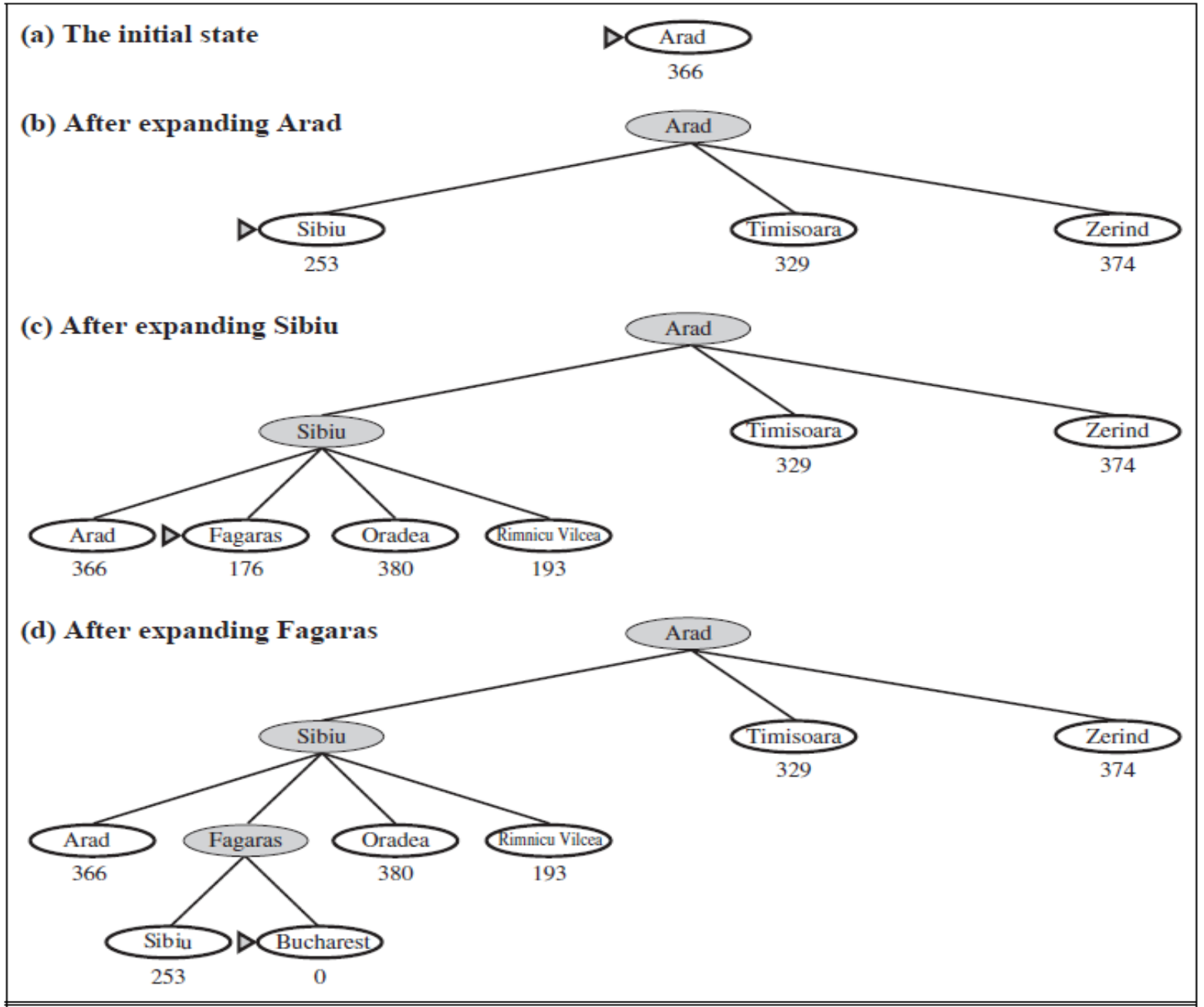


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* Algorithm

Source:

1. Stuart Russell & Peter Norvig, "Artificial Intelligence : A Modern Approach", Pearson Education, 2nd Edition.
2. Elaine Rich and Kevin Knight, "Artificial Intelligence" Tata McGraw Hill
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill

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.

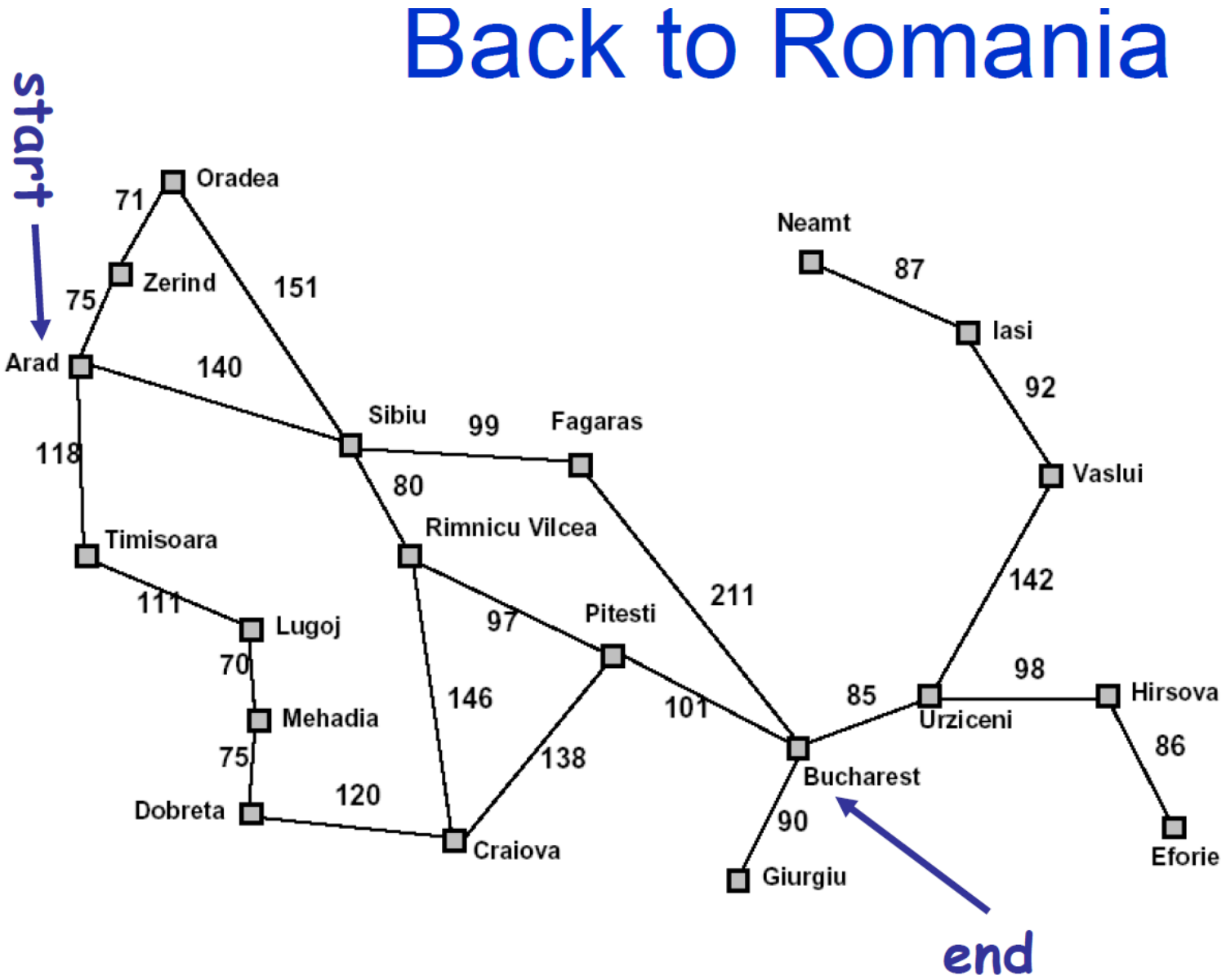
- Algorithm of A* search:
- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node 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

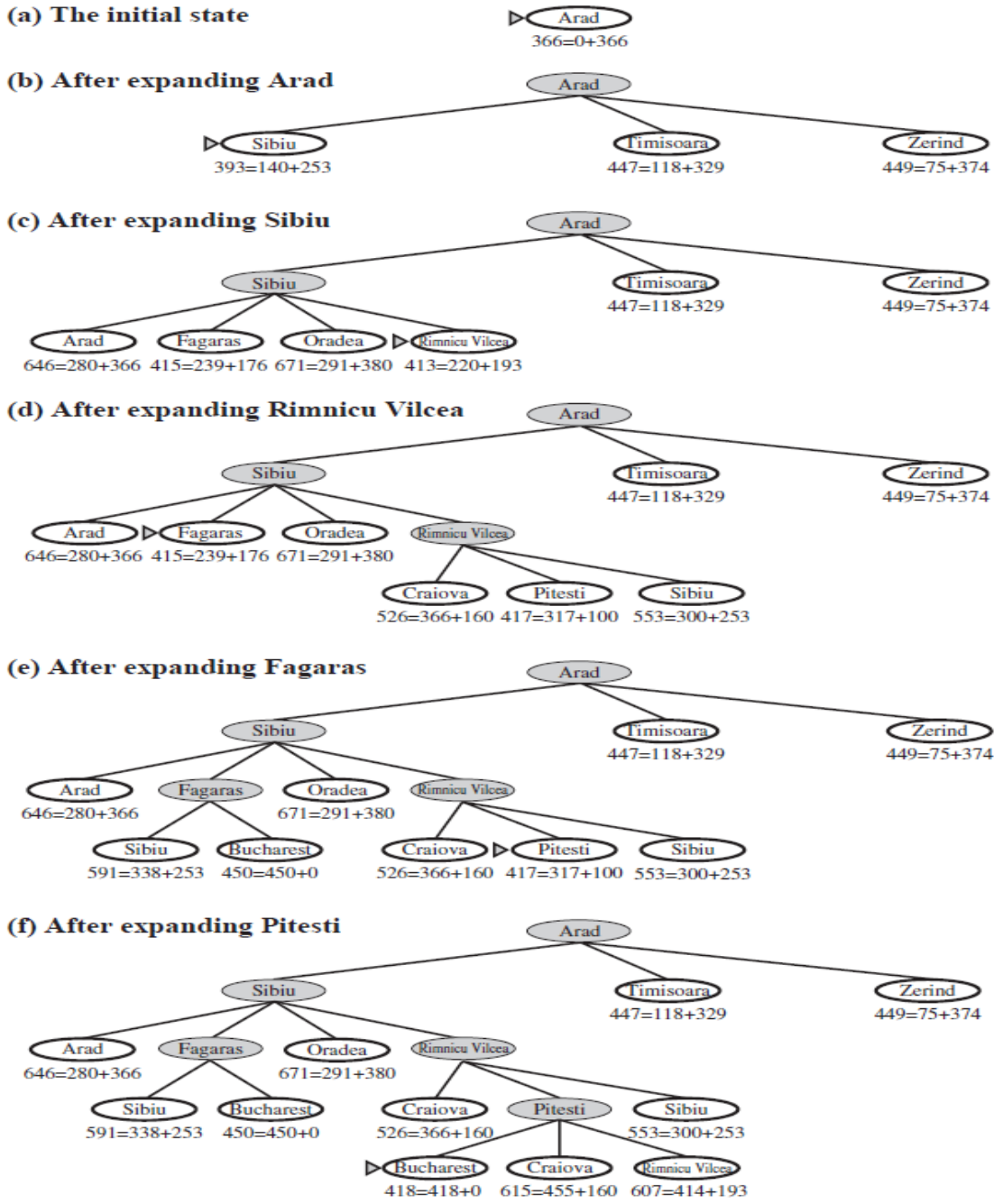


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

A* Algorithm (6)

- **Example: 8-Puzzle Problem**

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

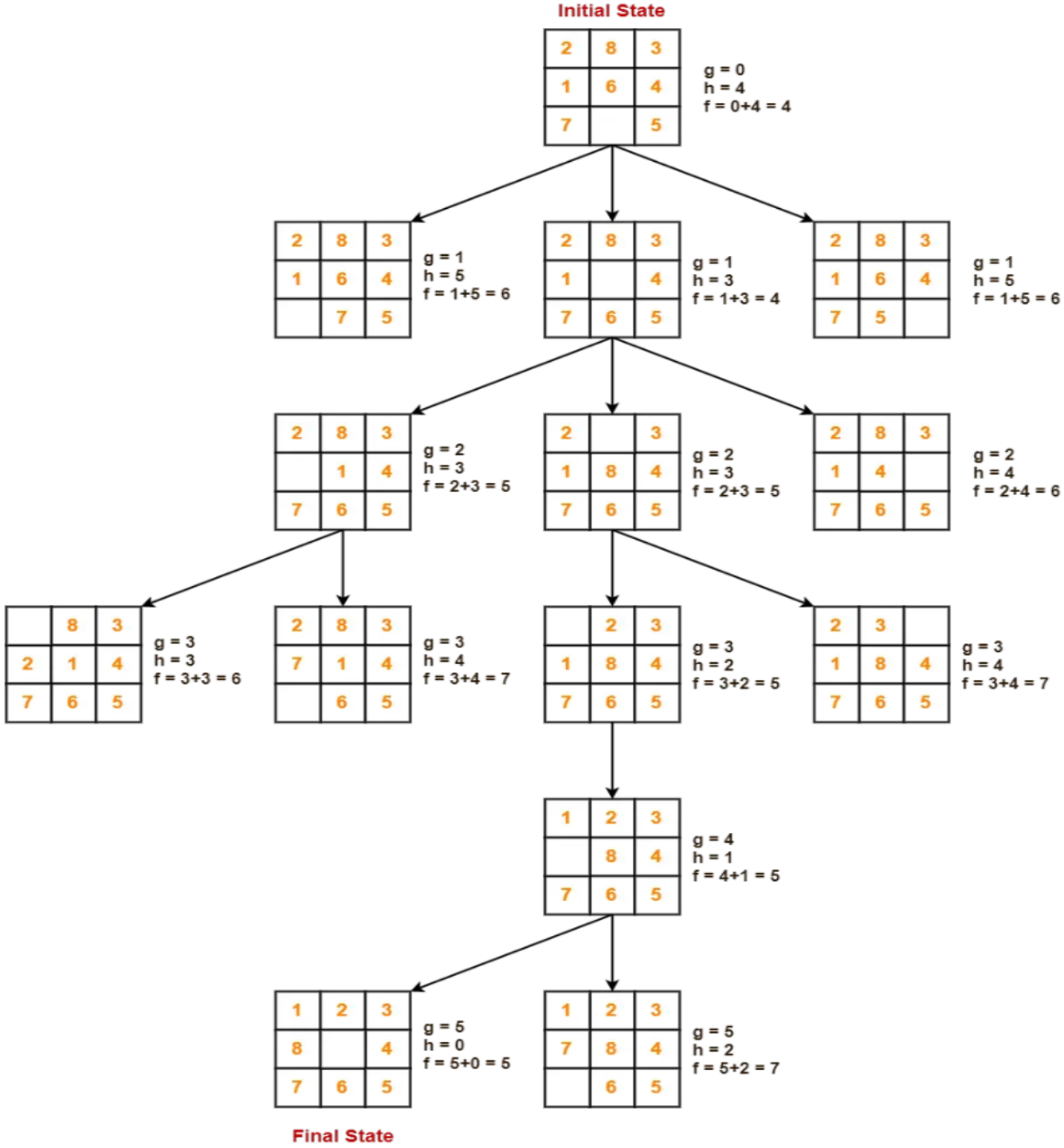
Final State

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

where

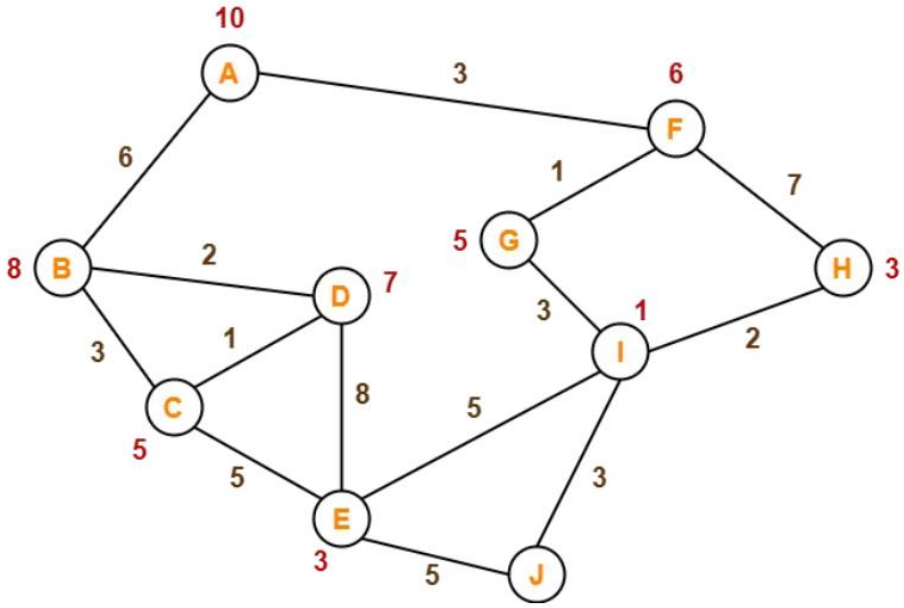
g(n) = Depth of node and

h(n) = Number of misplaced tiles.

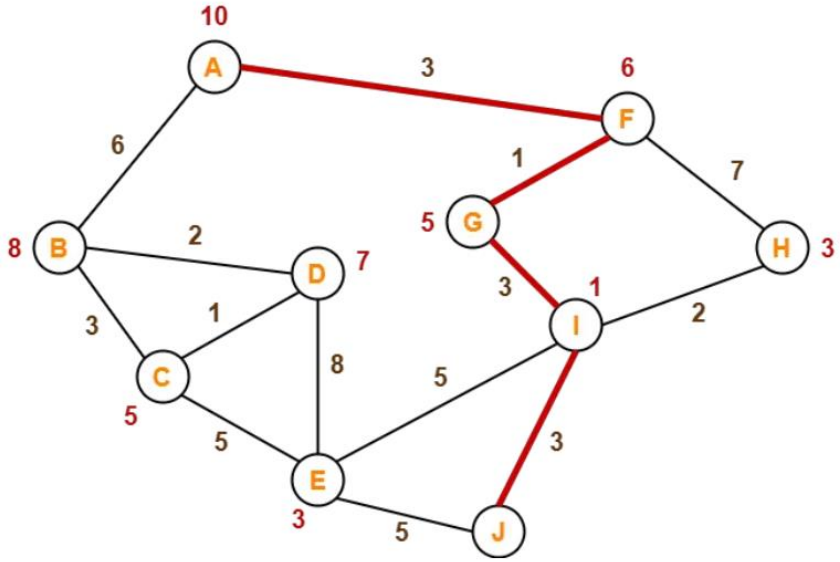


A* Algorithm (7)

- **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:**



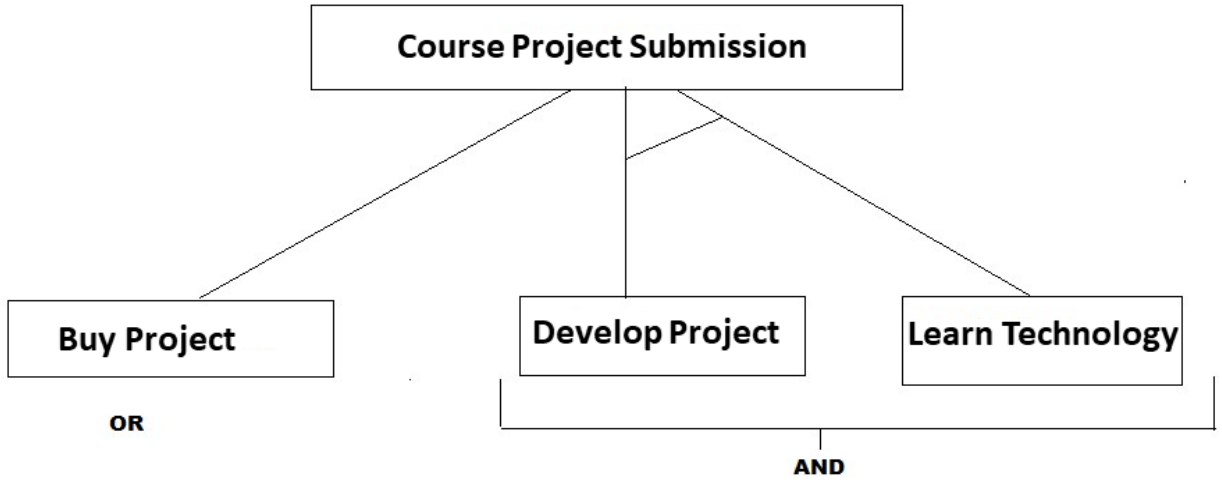
AO* Algorithm

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill

AO* Algorithm (1)

- AO* algorithm is a best-first search algorithm. AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.
- AND-OR graphs are specialized graphs that are used in problems that can be broken down into sub problems where AND side of the graph represent a set of task that need to be done to achieve the main goal , whereas the or side of the graph represent the different ways of performing task to achieve the same main goal.
  - The AND part of the graphs are represented by the AND-ARCS, referring that all the sub problems with the AND-ARCS need to be solved for the predecessor node or problem to be completed.
  - The edges without AND-ARCS are OR sub problems that can be done instead of the sub problems with And-arcs. It is to be noted that several edges can come from a single node as well as the presence of multiple AND arcs and multiple OR sub problems are possible.



AO* Algorithm (2)

- Difference between the A* Algorithm and AO* algorithm:
- A* algorithm and AO* algorithm both work on the best first search.
- They are both informed search and work on given heuristics values.
- A* always gives the optimal solution but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution doesn't explore all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses less memory.
- Opposite to the A* algorithm, the AO* algorithm cannot go into an endless loop.
- Comparing to A* algorithm, AO* algorithm is very efficient in searching the AND-OR trees.

- Working of AO* algorithm:
- The AO* algorithm works on the formula given below: $f(n) = g(n) + h(n)$

  where, g(n): The actual cost of traversal from initial state to the current state.

  h(n): The estimated cost of traversal from the current state to the goal state.

  f(n): The actual cost of traversal from the initial state to the goal state.

AO* Algorithm (3)

Algorithm 1: Pseudocode of AO* Algorithm

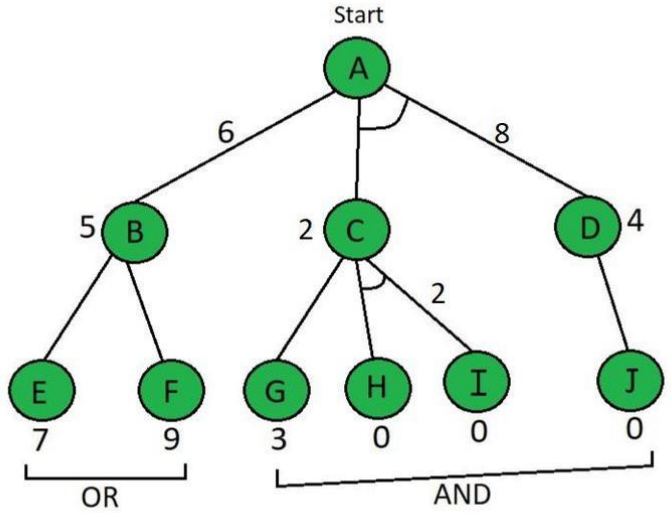
```
Data: Graph, StartNode
Result: The minimum cost path from StartNode to GoalNode
CurrentNode ← StartNode
while There is a new path with lower cost from StartNode to the
 GoalNode do
    calculate the cost of path from the current node to the goal
     node through each of its successor nodes;
    if the successor node is connected to other successor nodes by
     AND-ARCS then
        sum up the cost of all paths in the AND-ARC;
        return the total cost;
    else
        calculate the cost of the single path in the OR side;
        return the single cost;
    end
    find the minimum cost path
    CurrentNode ← SuccessorNodeOfMinimumCostPath
    if CurrentNode has no successor node then
        do the backpropagation and correct the estimated costs;
        CurrentNode ← StartNode
        return CurrentNode, New estimated costs;
    else
        return null;
    end
    return The minimum cost path;
end
```

AO* Algorithm (4)

- **AO* Algorithm Properties:**
- Time Complexity:
  - The time complexity of AO* 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 AO* algorithm will be $O(b^m)$.
- Space Complexity:
  - The worst-case space complexity of AO* algorithm is $O(b^m)$ where, m is the maximum depth of the search space.
- Completeness: AO* algorithm is complete meaning it finds a solution, if there is any, and does not fall into an infinite loop. Moreover, the AND feature in this algorithm reduces the demand for memory.
- Optimal: AO* search algorithm is not optimal because it stops as soon as it finds a solution and does not explore all the paths.

- The strength of AO* algorithm comes from a divide and conquer strategy. The AND feature brings all the tasks of the same goal under one umbrella and reduces the complexities. However, such a benefit comes at the cost of losing optimality.

AO* Algorithm (5)

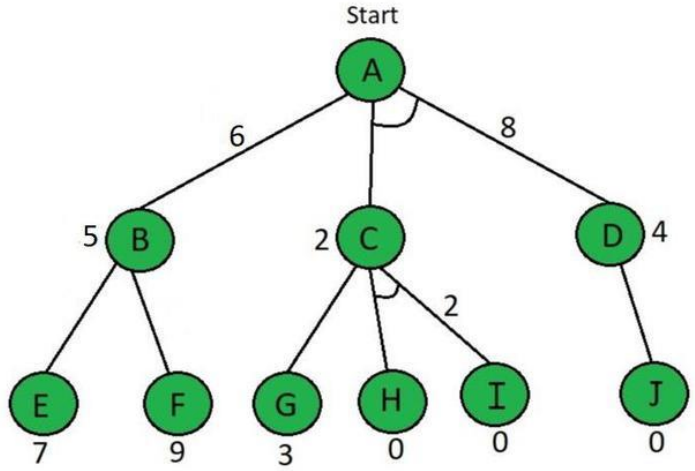


- Example: Here in this example below/besides the Node which is given is the heuristic value i.e h(n). Edge length is considered as 1.

- **Solution:** With help of f(n) = g(n) + h(n) evaluation function,

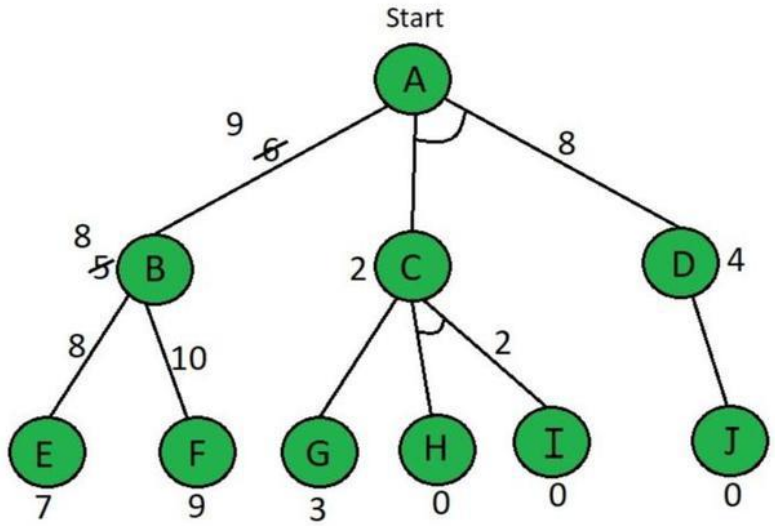
- **Step1:** Start from node A,
  - **f(A⇢B)** = g(B) + h(B)
    = 1 + 5 = 6 ……here g(n)=1 is taken by default for path cost
  - **f(A⇢C+D)** = g(C) + h(C) + g(D) + h(D)
    = 1 + 2 + 1 + 4 = 8 ……here we have added C & D because they are in AND

So, by calculation A⇢B path is chosen which is the minimum path, i.e f(A⇢B)



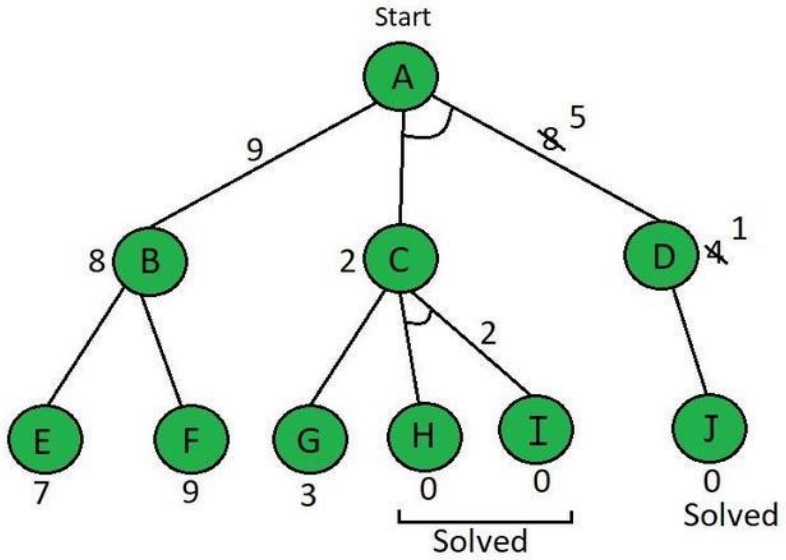
AO* Algorithm (6)

- **Solution:** (cont..)
- **Step2:** According to the answer of step 1, explore node B Here the value of E & F are calculated as follows,
  - **f(B⇢E)** = g(E) + h(E)
    = 1 + 7 = 8
  - **f(B⇢F)** = g(F) + h(F)
    = 1 + 9 = 10
  - So, by above calculation B⇢E path is chosen which is minimum path, i.e f(B⇢E) because B's heuristic value is different from its actual value. The heuristic is updated, and the minimum cost path is selected. The minimum value in our situation is 8.
  - Therefore, the heuristic for A must be updated due to the change in B's heuristic. So, we need to calculate it again.
  - **f(A⇢B)** = g(B) + updated h(B)
    = 1 + 8 = 9
  - We have updated all values in the above tree.



AO* Algorithm (7)

- **Solution:** (cont..)
- **Step3:** By comparing f(A⇢B) & f(A⇢C+D) now f(A⇢C+D) is shown to be smaller. i.e 8 < 9. So, explore f(A⇢C+D). The current node is C.
  - **f(C⇢G)** = g(G) + h(G) = 1 + 3 = 4
  - **f(C⇢H+I)** = g(H) + h(H) + g(I) + h(I) = 1 + 0 + 1 + 0 = 2 ……here we have added H & I because they are in AND
  - **f(C⇢H+I)** is selected as the path with the lowest cost and the heuristic is also left unchanged because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is 0, but Path A⇢D needs to be calculated because it has an AND.
  - **f(D⇢J)** = g(J) + h(J) = 1 + 0 = 1
  - The heuristic of node D needs to be updated to 1.
  - **f(A⇢C+D)** = g(C) + h(C) + g(D) + h(D) = 1 + 2 + 1 + 1 = 5
  - As we can see that path f(A⇢C+D) is get solved and this tree has become a solved tree now.
  - In simple words, the main flow of this algorithm is that we have to find firstly level 1st heuristic value and then level 2nd and after that update the values with going upward means towards the root node.
  - In this tree diagram, we have updated all the values.



Constraint Satisfaction

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill
4. http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf

Constraint Satisfaction Problem (1)

- Finding a solution that meets a set of constraints is the goal of constraint satisfaction problems (CSPs).

- Constraint Satisfaction Problems (CSP) representation:
  - The finite set of variables V1, V2, V3 ……..Vn.
  - Non-empty domain for every single variable D1, D2, D3 ……..Dn.
  - The finite set of constraints C1, C2 ……., Cm ; where each constraint Ci restricts the possible values for variables, e.g., V1 ≠ V2
  - Each constraint Ci is a pair <scope, relation> e.g.: <(V1, V2), V1 not equal to V2>
  - Scope = set of variables that participate in constraint.
  - Relation = list of valid variable value combinations.
    - There might be a clear list of permitted combinations. Perhaps a relation that is abstract and that allows for membership testing and listing.

- Examples of CSPs: Sudoku, 8-Queens Problem, Graph Coloring, Job scheduling, Crypt-Arithmetic Problem etc.

Constraint Satisfaction Problem (2)

- CSP can be solved by a backtracking search.
- It incrementally constructs a solution to a given problem by considering one candidate solution at a time.
- It discards the candidate solution if it violates the constraints mentioned in the problem. The constraints define a "bounding function" which helps to reject the candidate solutions not leading to the desired solution to the problem.
- Thus, it is the improved or smart exhaustive search. However, in the worst case, it also reaches to exponential complexity.
- Backtracking follows a depth first search.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

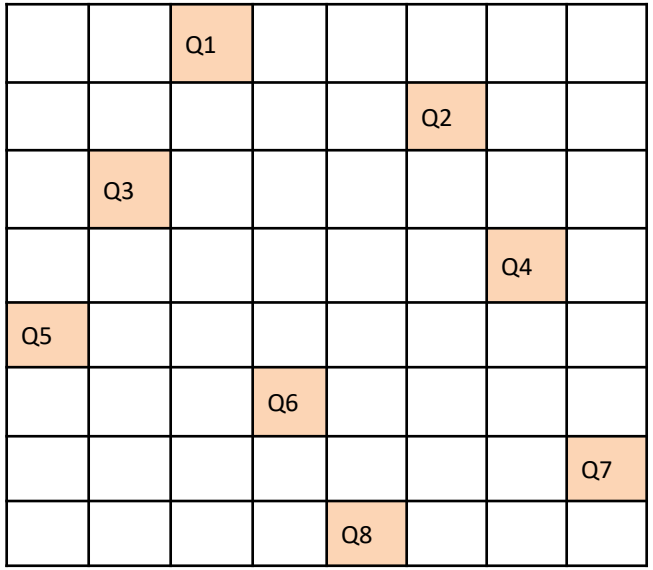
CSP Example1: N-Queens Problem(1)

- It is a puzzle of placing N queens on an N × N chessboard such that none of them can attack any other using the standard chess queen's moves.
- This implies that no two queens should be placed in the same row, column or diagonal.
- Mathematically it can be expressed as below :

  If 2 queens are placed at position (i, j) and (k, l) where i and k are row indices and j and l are column indices then

  - $i \neq k$ ....(No same row),
  - $j \neq l$ ...(No same column) and
  - $| i - k | \neq | j - l |$ ...(No same diagonal)

  … where $i, j, k, l \in \{1, 2, ..., 8\}$.



One of the solutions to 8-Queens problem

CSP Example1: N-Queens Problem(2)

- Basic steps of Recursive Backtracking Solution to N-Queens Problem:
  1) Start assigning a position to the 1st queen on the 1st row and 1st column.
  2) Place the next queen on a particular column of the next row by checking the valid non-attacking position.
  3) Similarly, assign the valid non-attacking column position of each row to each queen.
  4) If such valid column position is infeasible, discard that path without further exploration.
  5) Backtrack to previous queen's column position and assign alternative valid column position to that queen.
  6) Repeat step (4) and (5) until the complete solution is found. If the desired solution is not found then report the same.

CSP Example1: N-Queens Problem(3)

- Recursive Backtracking Algorithm for N-Queens Problem:

Algorithm Nqueens(k,n)

/*A recursive backtracking algorithm to find all solutions to the n-queens problem.

Input: n is the number of queens to be placed on an n x n chessboard. k is the queen number that is to be placed on the chessboard currently.

Output: An n-tuple solution X[1:n] representing the valid column positions of n queens.*/

```
{       k:=1;  //Initialization
        for(i:=1;i≤ n; i++) {
                if Place(k , i) then     //if a bounding function is satisfied
                {
                        X[k] :=i;
                        if (k = n) then write (X[1:n]);
                        else NQueens (k + 1, n);
                }
        }
}
```

CSP Example1: N-Queens Problem(4)

- Recursive Backtracking Algorithm for N-Queens Problem:

Algorithm Place(k,i)

/* It is a bounding function for the n-queens problem.

Input: k is the queen number that is to be placed on the k^{th} row. i is the column number of the k^{th} queen. A solution vector X[1:n] is a global array whose first (k-1) values are decided. X[1:n] is initialized with all 0 values. The function ABS(y) returns the absolute value of y.

Output: The algorithm returns TRUE if the kth queen is placed in the k^{th} row and the i^{th} column on an n x n chessboard, otherwise returns FALSE.*/

```
{
        for(j:=1; j≤ k-1; j++)
         {
                if ((X[j]=i) ||(ABS(X[j]–i) = ABS(j-k)))then
                /*Tests whether 2 queens are placed on the same column or on the
same diagonal*/
                return(FALSE);
         }
        return(TRUE);
}
```

CSP Example2: Crypt-Arithmetic Problem(1)

- Crypt-Arithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In cryptarithmetic problem, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.
- We can perform all the arithmetic operations on a given cryptarithmetic problem.

- The rules or constraints on a cryptarithmetic problem are as follows:
  - There should be a unique digit to be replaced with a unique alphabet.
  - The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4, nothing else.
  - Digits should be from 0-9 only.
  - There should be only one carry forward, while performing the addition operation on a problem.
  - The problem can be solved from both sides, i.e., lefthand side (L.H.S), or righthand side (R.H.S)

CSP Example2: Crypt-Arithmetic Problem(2)

- Example: Given a cryptarithmetic problem, i.e., S E N D + M O R E = M O N E Y
- In this example, add both terms S E N D and M O R E to bring M O N E Y as a result.

```
  SEND
+MORE
-----
MONEY
-----
```

- **Solution:**
- Starting from the left hand side (L.H.S) , the terms are **S** and **M**. Assign a digit which could give a satisfactory result. Let's assign **S->9** and **M->1**. Hence, we get a satisfactory result by adding up the terms and got an assignment for O as **O->0** as well.

```
  S          9
+ M        + 1
-----      ----
 MO         10
-----      ----
```

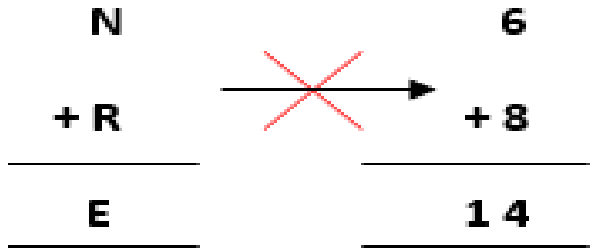
- Now, move ahead to the next terms **E** and **O** to get **N** as its output. Adding **E** and **O**, which means 5+0=0, which is not possible because according to cryptarithmetic constraints, **we cannot assign the same digit to two letters.** So, we need to think more and assign some other value.

```
  E          5
+ O        + 0
-----      ----
  N          5
-----      ----
```

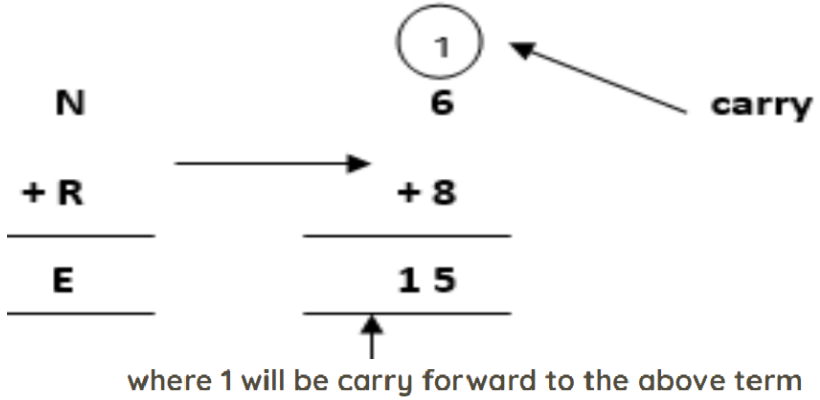
```
    1  <-- carry
  E          5
+ O        + 0
-----      ----
  N          6
-----      ----
```

CSP Example2: Crypt-Arithmetic Problem(3)

- Example: (cont..)
- **Note:** When we will solve further, we will get one carry, so after applying it, the answer will be satisfied.
- Further, adding the next two terms **N** and **R** we get,

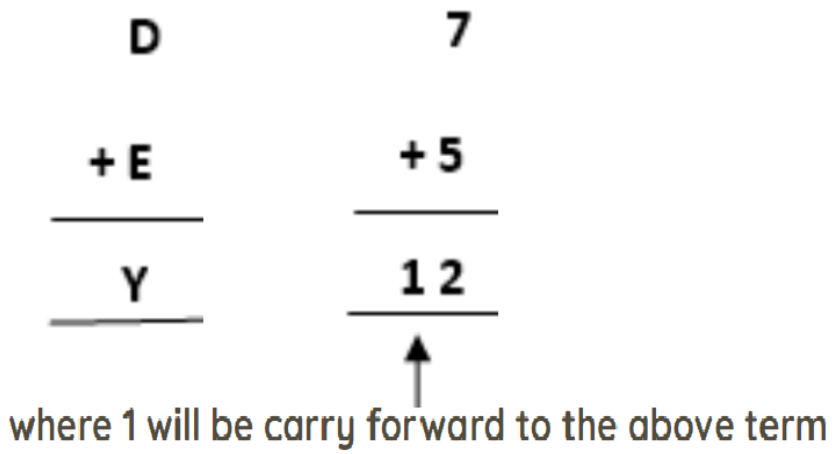


- But, we have already assigned **E->5**. Thus, the above result does not satisfy the values because we are getting a different value for E. So, we need to think more.
- Again, after solving the whole problem, we will get a carryover on this term, so our answer will be satisfied.



CSP Example2: Crypt-Arithmetic Problem(4)

- Example: (cont..)
- Again, on adding the last two terms, i.e., the rightmost terms **D** and **E**, we get **Y** as its result.



- Keeping all the constraints in mind, the final resultant is as follows:

S	9
E	5
N	6
D	7
M	1
O	0
R	8
y	2

CSP Example2: Crypt-Arithmetic Problem(5)

- Steps to solve Crypt-Arithmetic Problem:
- Start by examining the rightmost digit of the topmost row, with a carry of 0
- If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
- If we are currently trying to assign a char in one of the addends If char already assigned, just recur on the row beneath this one, adding value into the sum If not assigned, then
  - for (every possible choice among the digits not in use) make that choice and then on row beneath this one, if successful, return true if !successful, unmake assignment and try another digit
  - return false if no assignment worked to trigger backtracking
- Else if trying to assign a char in the sum
- If char assigned & matches correct, recur on next column to the left with carry, if success return true,
- If char assigned & doesn't match, return false
- If char unassigned & correct digit already used, return false
- If char unassigned & correct digit unused, assign it and recur on next column to left with carry, if success return true
- return false to trigger backtracking

Means Ends Analysis

Source:

1. https://www.javatpoint.com/digital-image-processing-tutorial

Means Ends Analysis(1)

- Means-Ends Analysis (MEA) is problem-solving techniques used in AI for limiting search in AI programs.
- It is a mixture of Backward and forward search technique.
- It was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current and goal state.
- It can be applied recursively for a problem. It is a strategy to control search in problem-solving.

- Following are the main Steps which describes the working of MEA technique for solving a problem.
  - First, evaluate the difference between Initial State and final State.
  - Select the various operators which can be applied for each difference.
  - Apply the operator at each difference, which reduces the difference between the current state and goal state.

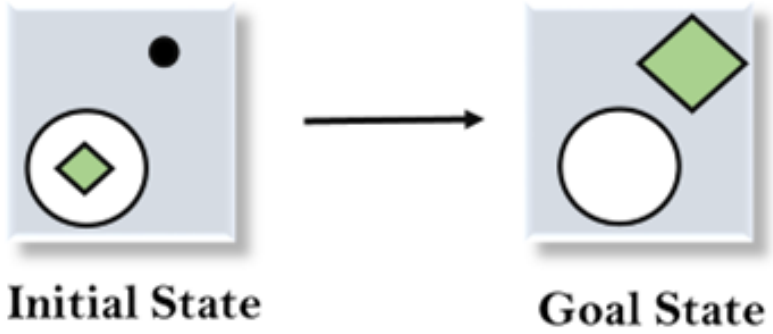
Means Ends Analysis(2)

- MEA Algorithm:
- Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.
- **Step 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.

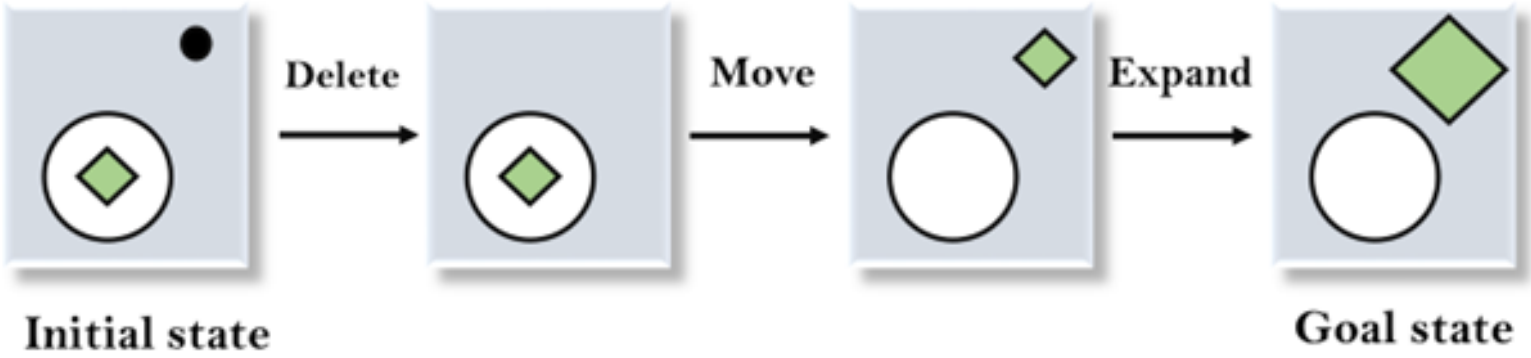
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.
  a) Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
  b) Attempt to apply operator O to CURRENT. Make a description of two states.
     i) O-Start, a state in which O?s preconditions are satisfied.
     ii) O-Result, the state that would result if O were applied In O-start.
  c) If (First-Part <------ MEA (CURRENT, O-START) And (LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

Means Ends Analysis(3)

- Example: In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.



- **Solution:**
- First find the differences between initial states and goal states, and for each difference, generate a new state and apply the operators. The operators we have for this problem are: 1) Move, 2) Delete, 3) Expand



Game Playing

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
3. Deepak Khemani, “A First Course in Artificial Intelligence”, McGraw Hill
4. https://www.javatpoint.com/digital-image-processing-tutorial

Game Playing

- Game playing is a popular application of artificial intelligence that involves the development of computer programs to play games, such as Chess, Checkers, tic-tac-toe, Go etc.
- It's a good reasoning problem, formal and nontrivial.
- It does direct comparison with humans and other computer programs is easy.
- In game playing it is challenging to handle unpredictable opponent's moves.
- The goal of game playing in AI is to develop algorithms that can learn how to play games and make decisions that will lead to winning outcomes.
- Two main approaches to game playing in AI:
  - Rule-based systems use a set of fixed rules to play the game.
  - Machine learning-based systems use algorithms to learn from experience and make decisions based on that experience.
- Game playing in AI is an active area of research and has many practical applications, including game development, education, and military training.
- By simulating game playing scenarios, AI algorithms can be used to develop more effective decision-making systems for real-world applications.
- The most common search technique in game playing is Minimax search.

Minimax Search (1)

- Mini-max Search is a recursive or backtracking algorithm which is used in decision-making and game theory.
- It provides an optimal move for the player assuming that opponent is also playing optimally.
- It uses recursion to search through the game-tree. It computes the minimax decision for the current state.
- In this algorithm two players play the game: one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Minimax Search (2)

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

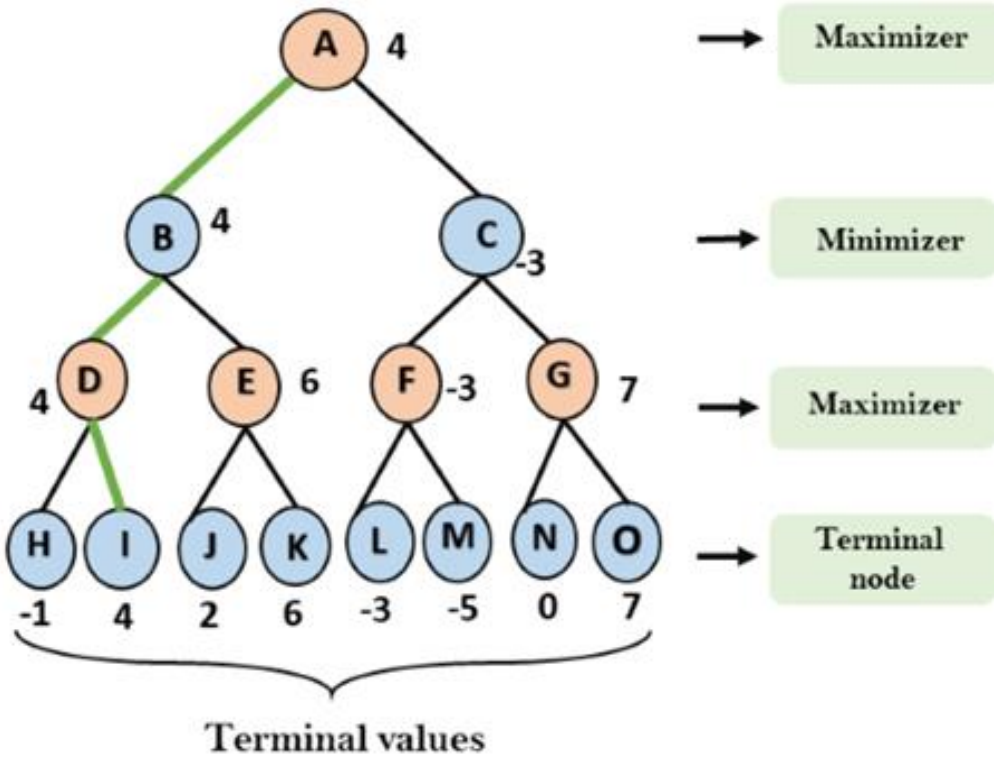
Minimax Search (3)

- Properties of Mini-Max algorithm:
- Complete- Mini-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- Optimal- Mini-Max algorithm is optimal if both opponents are playing optimally.
- Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is $O(b^m)$, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning.

Minimax Search (4)

- Example:
- In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.
- Maximizer determines the higher nodes values and finds the maximum among the all. However, Minimizer determines the lower nodes values and finds the minimum among the all.



Alpha-Beta Cutoffs (1)

- Alpha-beta cutoff/pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half.
- Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning.
- This involves two threshold parameter alpha and beta for future expansion, so it is called alpha-beta pruning or alpha-beta cutoff algorithm.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  - Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Alpha-Beta Cutoffs (2)

alpha= the highest value for MAX along the path
beta= the lowest value for MIN along the path

```
MinVal(state, alpha, beta){
  if (terminal(state))
          return utility(state);
  for (s in children(state)){
     child = MaxVal(s,alpha,beta);
     beta = min(beta,child);
     if (alpha>=beta) return child;
  }
  return beta; }
```

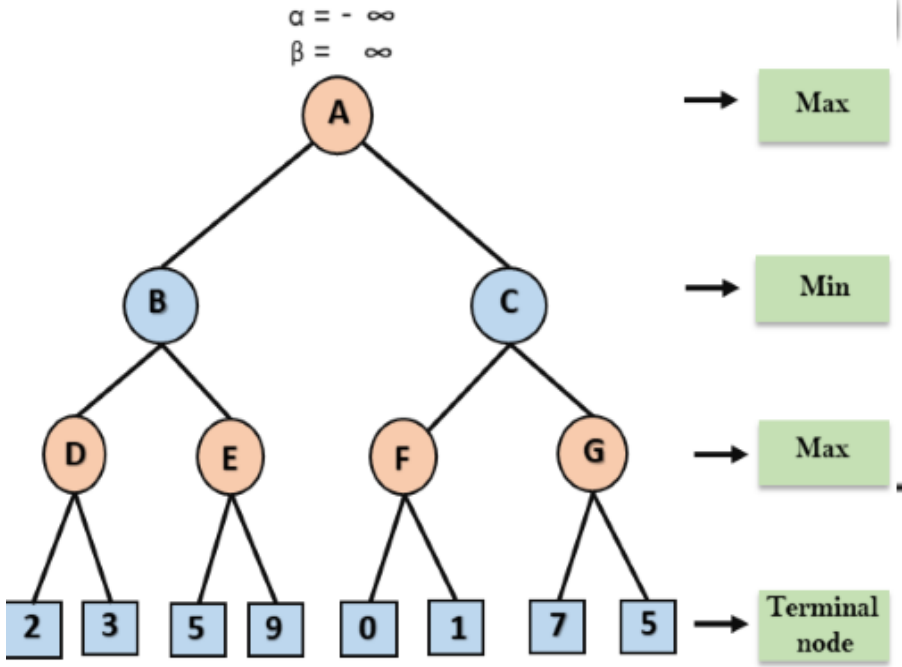
```
MaxVal(state, alpha, beta){
  if (terminal(state))
          return utility(state);
  for (s in children(state)){
     child = MinVal(s,alpha,beta);
     alpha = max(alpha,child);
     if (alpha>=beta) return child;
  }
  return beta; }
```

Alpha-Beta Cutoffs (3)

- Key points about alpha-beta pruning:
- The condition for Alpha-beta Pruning is that $\alpha >= \beta$.
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

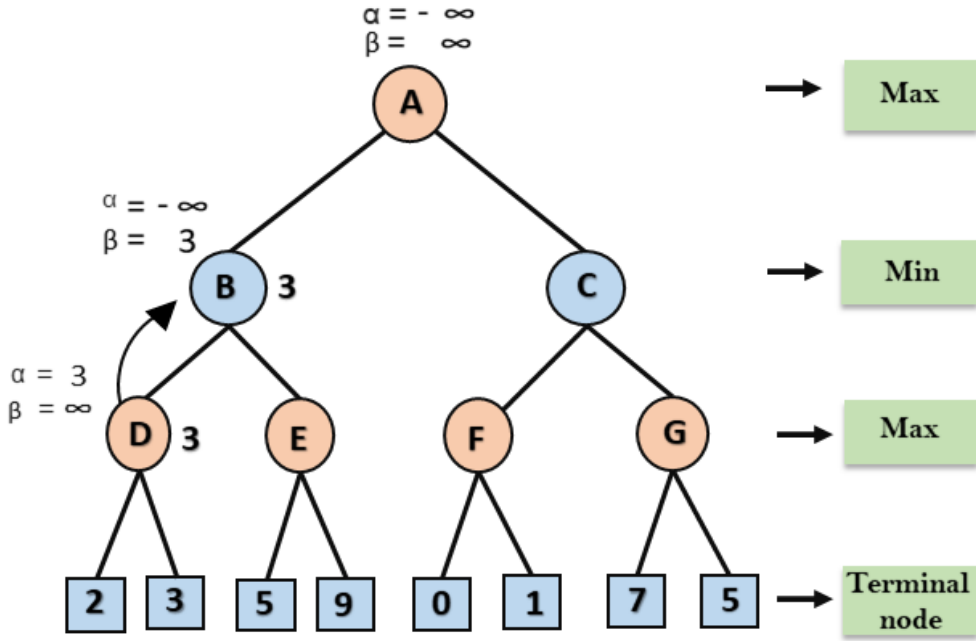
- Example:
- **Step 1:** We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. $\alpha = -\infty$ and $\beta = +\infty$. We will prune the node only when $\alpha >= \beta$.

- **Step 2:** Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

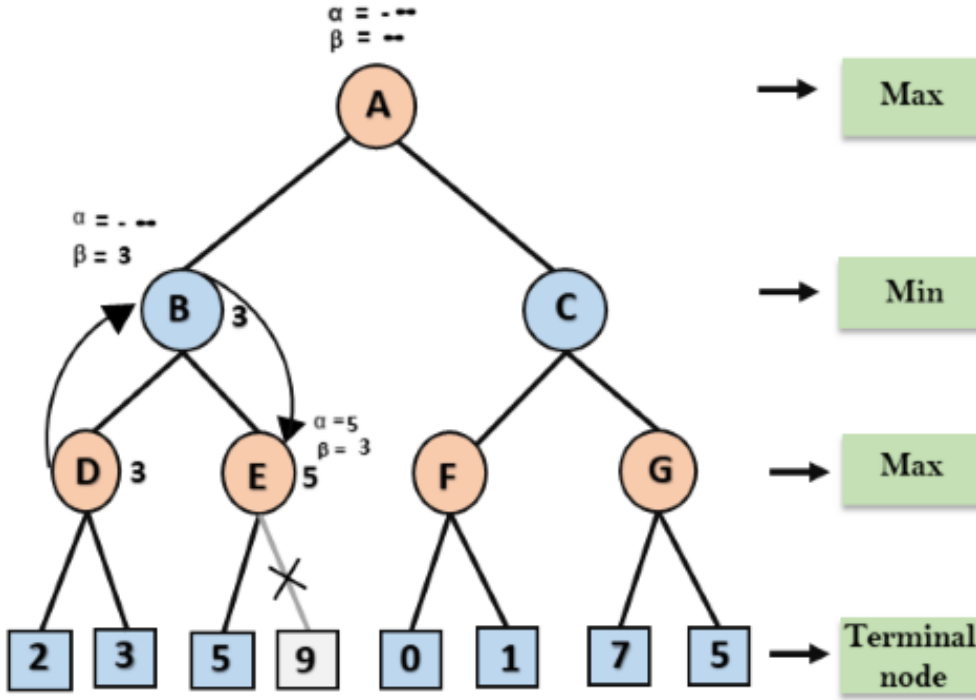


Alpha-Beta Cutoffs (4)

- Example: (cont..)
- **Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3. In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.



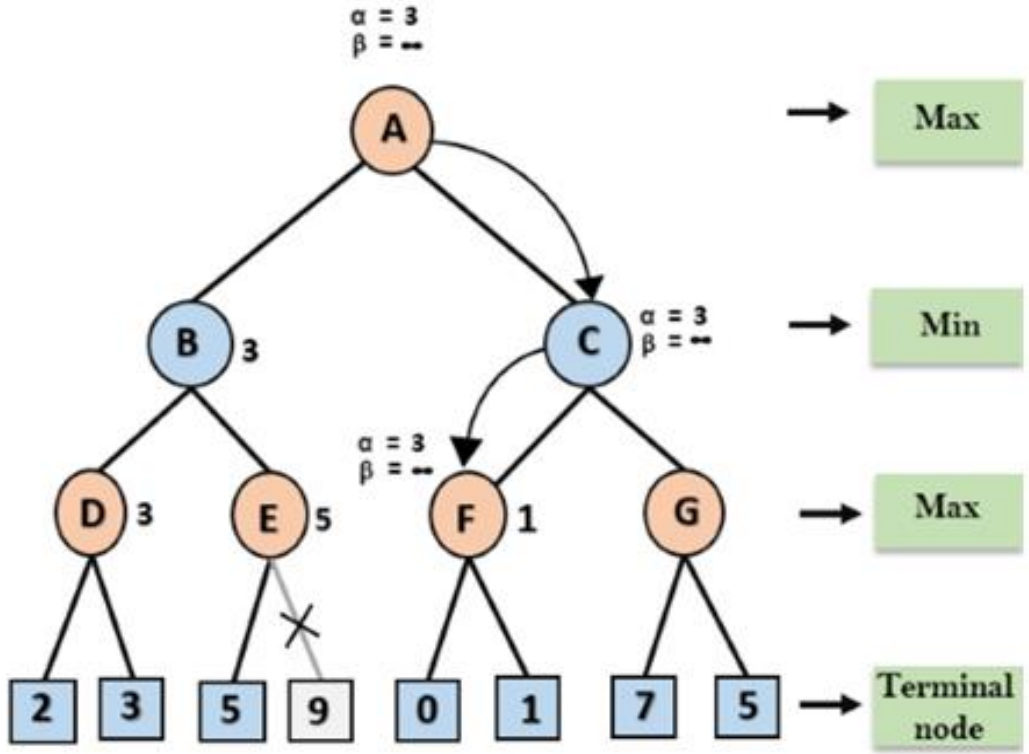
- **Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



Alpha-Beta Cutoffs (5)

- Example: (cont..)
- **Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C. At node C, α=3 and β= +∞, and the same values will be passed on to node F.

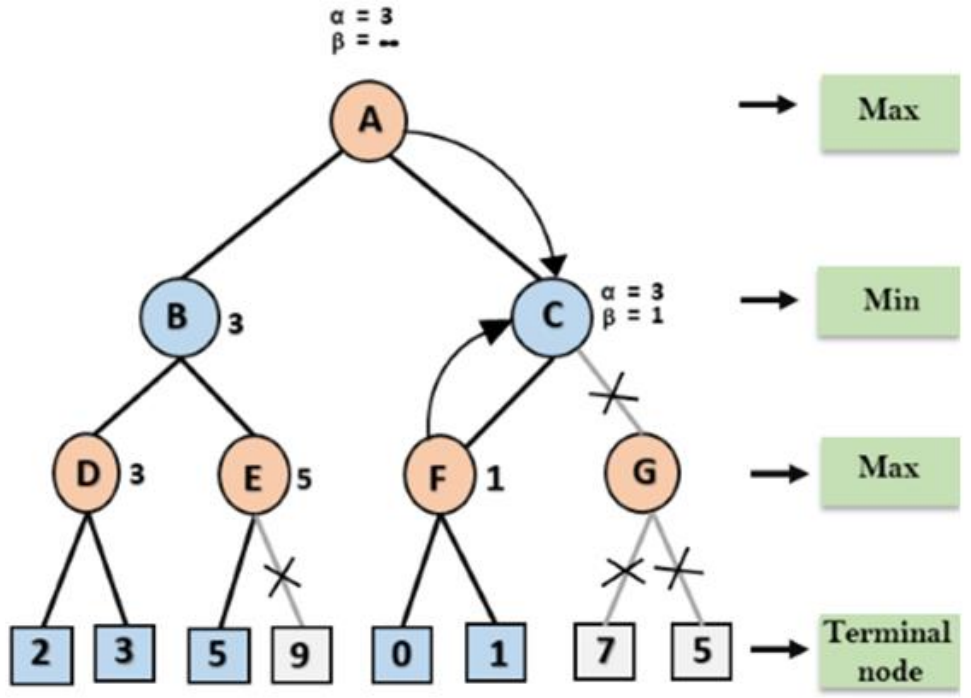
- **Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

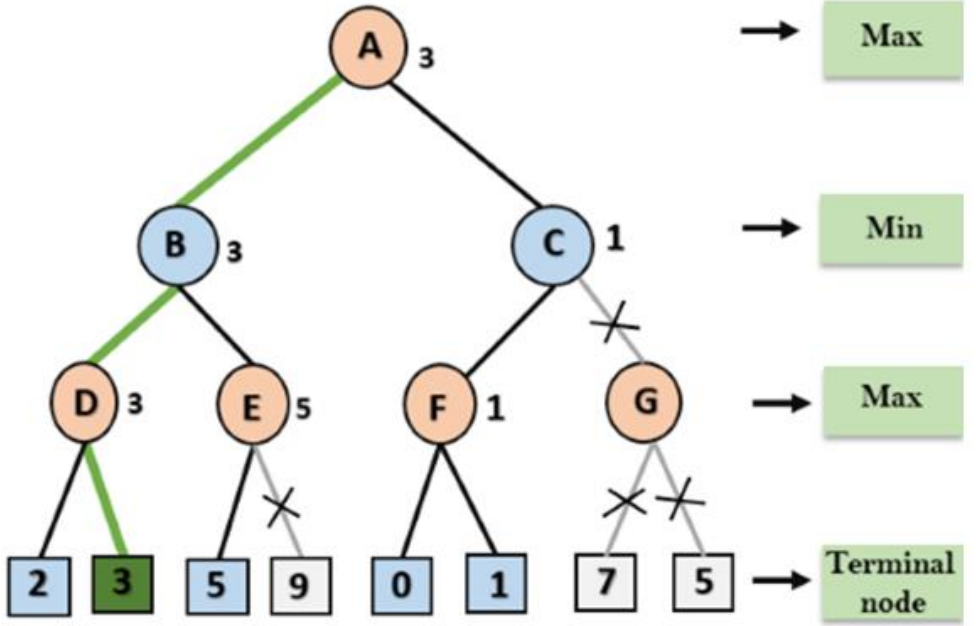


Alpha-Beta Cutoffs (6)

- Example: (cont..)
- **Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.

- **Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.





Alpha-Beta Cutoffs (7)

- Move Ordering in Alpha-Beta pruning:
- The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.
- It can be of two types:
  - Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
  - Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Time complexity in ideal ordering is $O(b^{m/2})$.

- Rules to find good ordering:
  - Occur the best move from the shallowest node.
  - Order the nodes in the tree such that the best nodes are checked first.
  - Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
  - We can bookkeep the states, as there is a possibility that states may repeat.

Waiting for Quiescence (1)

- A number of refinements are possible to alpha-beta. One is waiting for quiescence.
- Waiting for Quiescence search is an algorithm typically used to extend search at unstable nodes in minimax game trees in game-playing computer programs.
- It continues the search until no drastic change occurs from one level to the next.
- The evaluation function should be applied only to positions that are **quiescent**—that is, unlikely to exhibit wild swings in value in the near future.
- Nonquiescent positions can be expanded further until quiescent positions are reached. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material.
- Sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.
- It mitigates the effect of the horizon problem faced by AI engines for various games like chess and Go.
- The horizon problem: The problem with abruptly stopping a search at a fixed depth is called the 'horizon effect'.
- This involves searching past the terminal search nodes (depth of 0) and testing all the non-quiescent or 'violent' moves until the situation becomes calm, and only then apply the evaluator.
- It enables programs to detect long capture sequences and calculate whether or not they are worth initiating.
- It expands searches to avoid evaluating a position where tactical disruption is in progress.

Waiting for Quiescence (2)

- Pseudocode:

```
function quiescence_search(node, depth) is
    if node appears quiet or node is a terminal node or depth = 0 then
        return estimated value of node
    else
        (recursively search node children with quiescence_search)
        return estimated value of children

function normal_search(node, depth) is
    if node is a terminal node then
        return estimated value of node
    else if depth = 0 then
        if node appears quiet then
            return estimated value of node
        else
            return estimated value from quiescence_search(node, reasonable_depth_value)
    else
        (recursively search node children with normal_search)
        return estimated value of children
```

Thank you!