Unit 3
SEARCHING

- Searching is the process of finding the required states or nodes.
- Searching is to be performed through the state space.
- Search process is carried out by constructing a search tree.
- Search is a universal problem-solving technique.

Why is search important in Artificial Intelligence?

Modeling and solving problems on a computer in this manner dates back to Alan Turing and many AI problems can be easily modeled as state spaces.
Solving these problems can "simply" be reduced to exploring the state space, and identifying the correct answer.

SEARCH TERMINOLOGY

- **Problem Space**: Environment in which the search takes place.
- **Problem Instance:** It is Initial state + Goal state
- **Problem Space Graph:** It represents problem state. States are shown by nodes and operators are shown by edges.
- **Depth of a problem:** Length of a shortest path or shortest sequence of operators from Initial State to goal state.

- **Space Complexity:** The maximum number of nodes that are stored in memory.
- **Time Complexity:** The maximum number of nodes that are created.
- **Admissibility:** A property of an algorithm to always find an optimal solution.
- **Branching Factor:** The average number of child nodes in the problem space graph.

SEARCH STRATEGIES

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:

Completeness:

does it generate to find a solution if there is any?

Optimality:

does it always find the highest quality (least-cost) solution?

Time complexity:

How long does it take to find a solution?

Space complexity:

How much memory does it need to perform the search?

Time and space complexity are measured in terms of

- *b:* maximum branching factor of the search tree
- *d:* depth of the least-cost solution
- *m*: maximum depth of the state space (may be ∞)

CLASSIFICATION

Uninformed Search (Blind Search/Brute force search)

The search algorithms that do not use any extra information regarding the problem

- ✓ Depth First Search
- ✓ Breath First Search
- ✓ Depth Limit Search
- ✓ Iterative deepening
- ✓ Uniform Cost
- ✓ Bidirectional Search

Informed search or Heuristic search

Informed search have problem specific knowledge apart from problem definition

- ✓ Hill climbing Search
- ✓ Best first Search
  - Greedy Best First Search
  - A* Search
- ✓ Simulated Annealing

DEPTH FIRST SEARCH (DFS)

- Proceeds down a single branch of the tree at a time
- Expands the root node, then the leftmost child of the root node
- Always expands a node at the deepest level of the tree
- Only when the search hits a dead end (a partial solution which can't be extended), the search backtrack and expand nodes at higher levels.

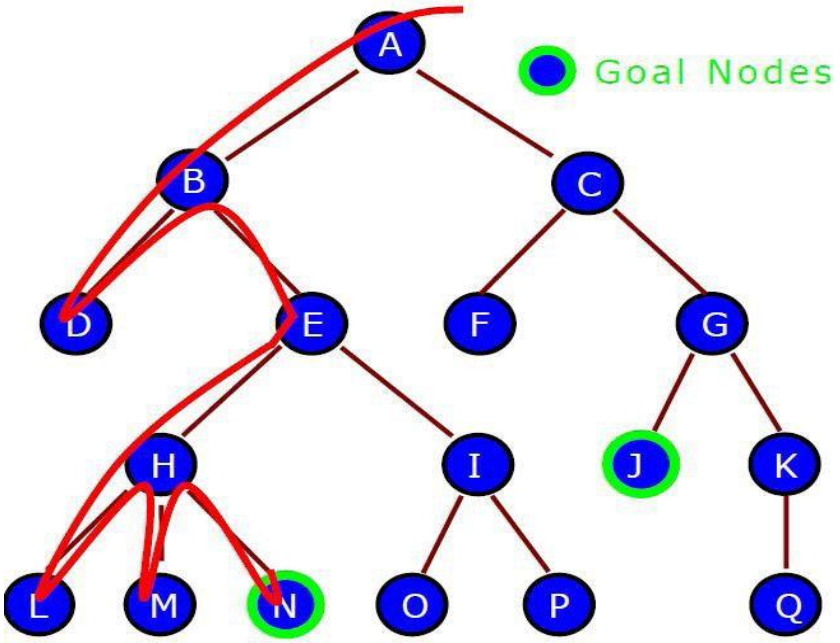


Fig. Depth-first search (DFS)

DEPTH FIRST SEARCH (DFS)

- ***Completeness***: Incomplete as it may get stuck going down an infinite branch that doesn't leads to solution.
- ***Optimality***: The first solution found by the DFS may not be shortest.
- ***Space complexity***: For m nodes down the path, we have to store b nodes extra for each of the m nodes. That's how you get a O(bm) space complexity.
- ***Time Complexity***:

If you can access each node in O(1) time, then with branching factor of b and max depth of m, the total number of nodes in this tree would be = b * b *b … m times = b^m, resulting in total time to visit each node proportional to bm. Hence the complexity = O(b^m)

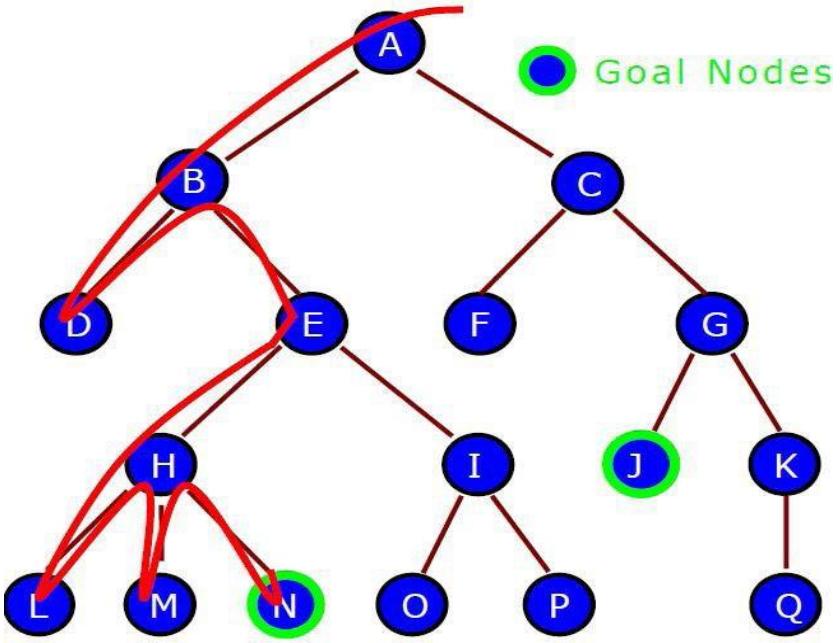
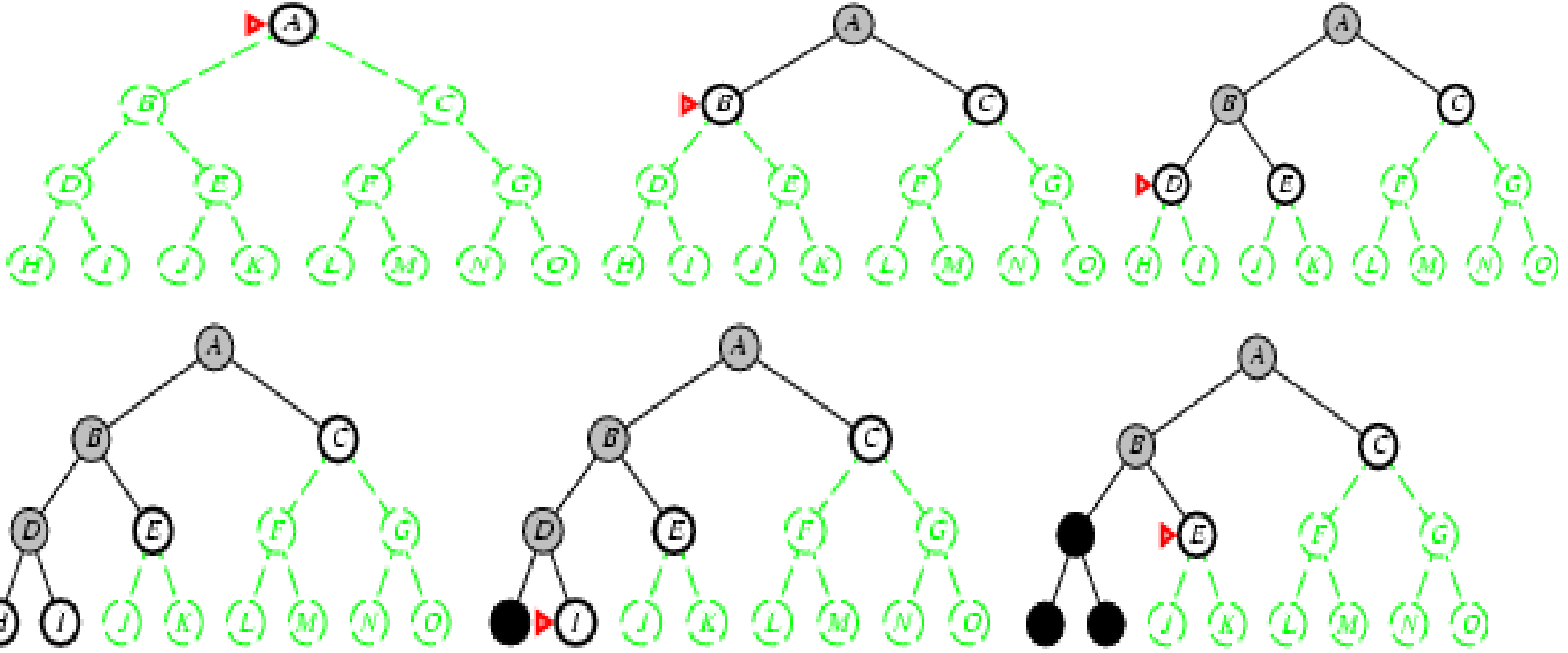


Fig. Depth-first search (DFS)

DFS example (find path from A to E)



Breadth-First Search (BFS)

- Proceeds level by level down the search tree
- Starting from the root node (initial state) explores all children of the root node, left to right
- If no solution is found, expandsthe first (leftmost) child of the root node, then expands the second node and so on

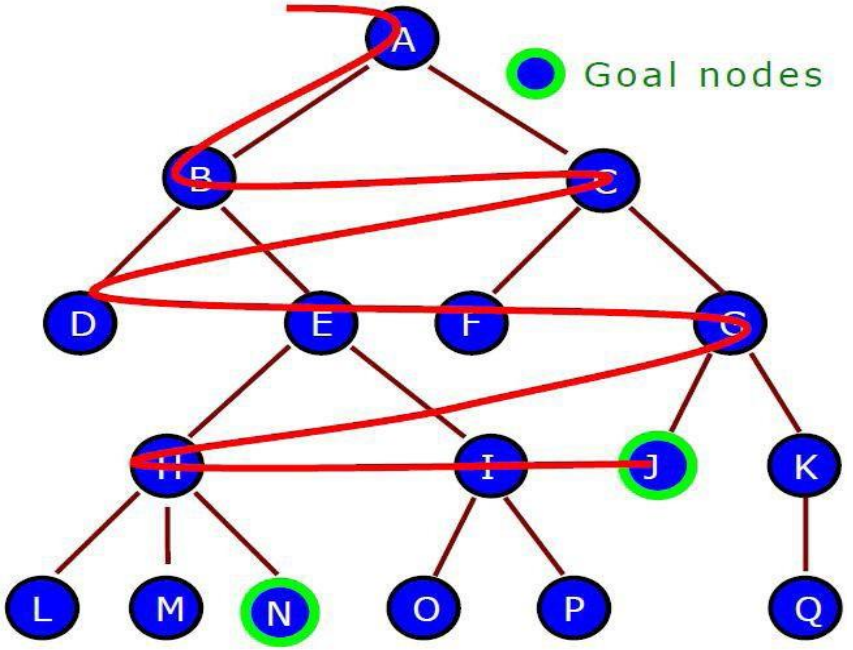


Fig. Breadth-first search (BFS)

Breadth-First Search (BFS)

- Completeness: Complete if the goal node is at finite depth
- Optimality: It is guaranteed to find the shortest path
- Time Complexity: $O(b^{d+1})$
- Space Complexity: $O(b^{d+1})$

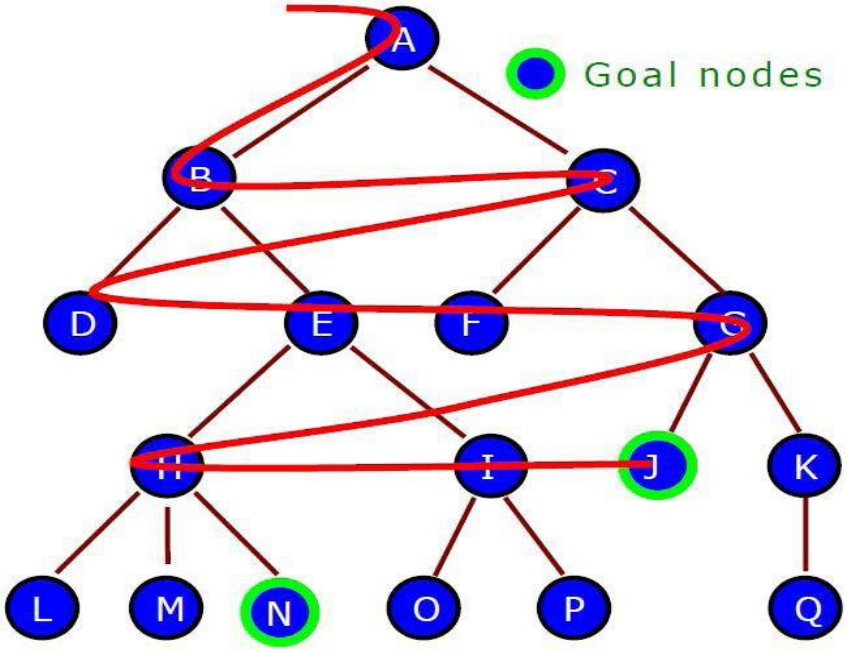
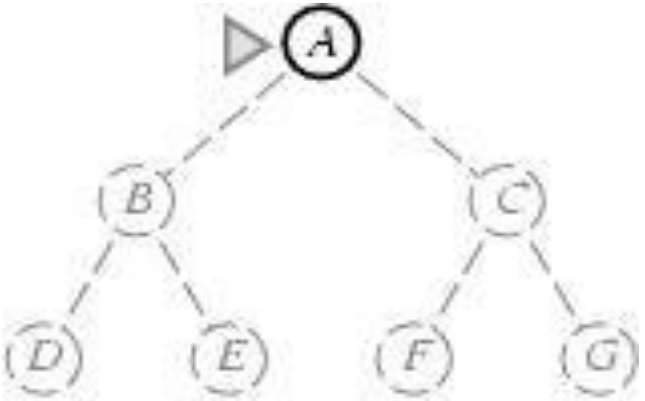
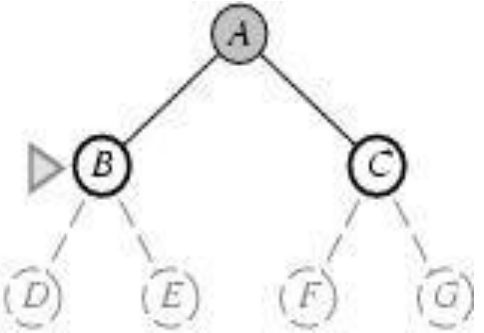
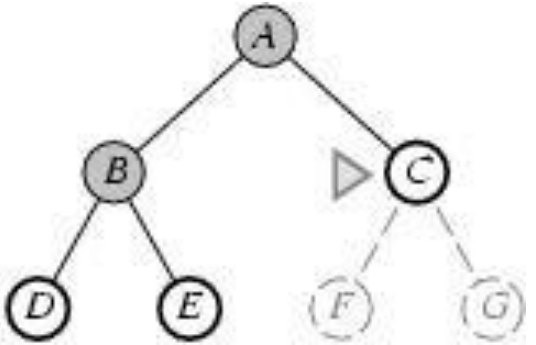


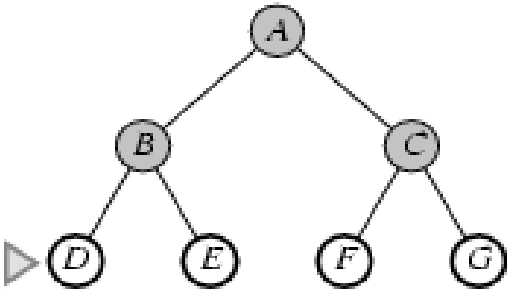
Fig. Breadth-first search (BFS)

BFS example (Find path from A to D)







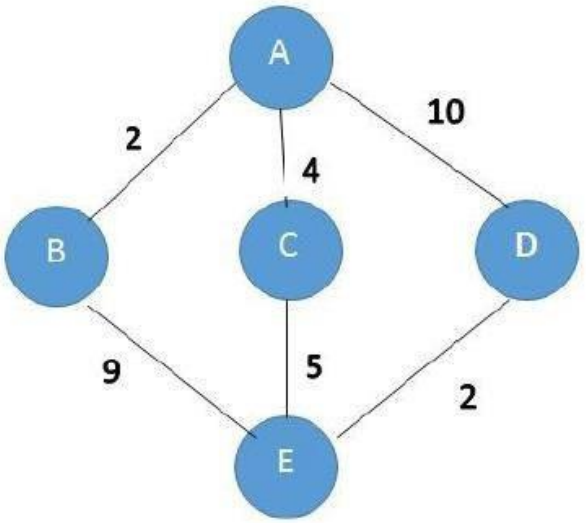


Uniform-Cost Search

Uniform-cost is guided by path cost rather than path length like in BFS, the algorithms starts by expanding the root, then expanding the nodewith the lowest cost from the root, the search continues in this mannerfor all nodes.

- **Completeness:**
  Complete if the cost of each step exceeds some small positive integer, this to prevent infinite loops.
- **Optimality:**
  Optimal in the sense that the node that it always expands is the node with the least path cost.
- **Time Complexity:** $O(b^{C/e})$.
- **Space Complexity:** $O(b^{C/e})$

UCS example (Find path from A to E)



- Expand A to B, C, and D.
- The path to B is the cheapest one with path cost 2.
- Expand B to E
- Total path cost = 2+9 =11
- This might not be the optimal solution since the path AC as path cost 4 (less than 11)

- Expand C to E
- Total path cost = 4+5 =9
- Path cost from A to D is 10 (greater than path cost, 9)
- Hence optimal path is ACE.

Depth Limit Search

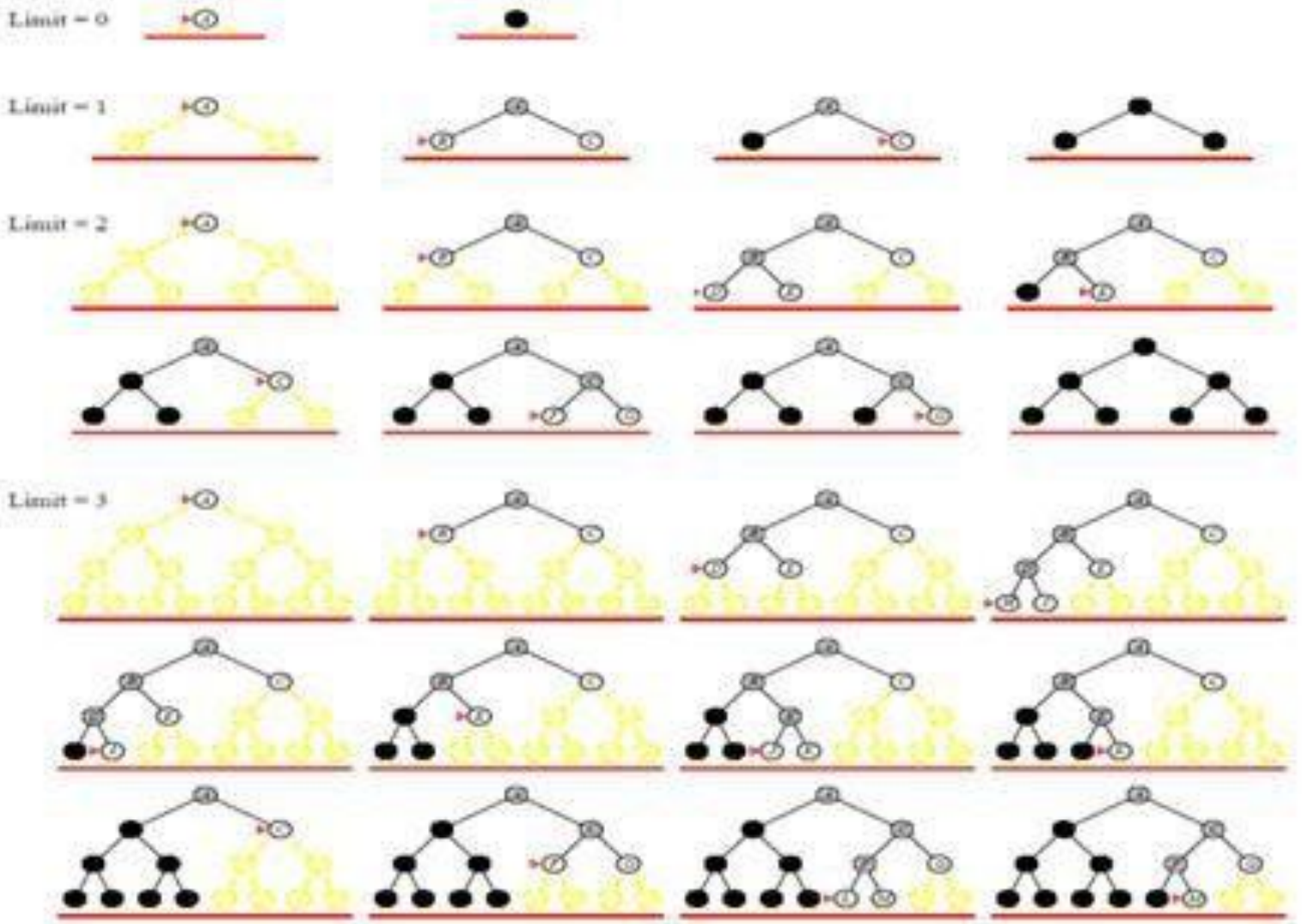
- Depth-first search will not find a goal if it searches down a path that has infinite length. So, in general, depth-first search is not guaranteed to find a solution, so it is not complete.
- This problem is eliminated by limiting the depth of the search to some value L. However, this introduces another way of preventing depth-first search from finding the goal: if the goal is deeper than L it will not be found.
- Perform depth first search but only to a pre-specified depth limit L.
- No node on a path that is more than L steps from the initial state

Depth Limit Search

- ***Completeness***: Incomplete as solution may be beyond specified depth level.
- ***Optimality***: not optimal
- ***Space complexity***: b as branching factor and L as tree depth level, O (b.L)
- ***Time Complexity***: O(bL)

Iterative Deepening Search (IDS)

- Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits.
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
- Search is helpful only if the solution is at given depth level.

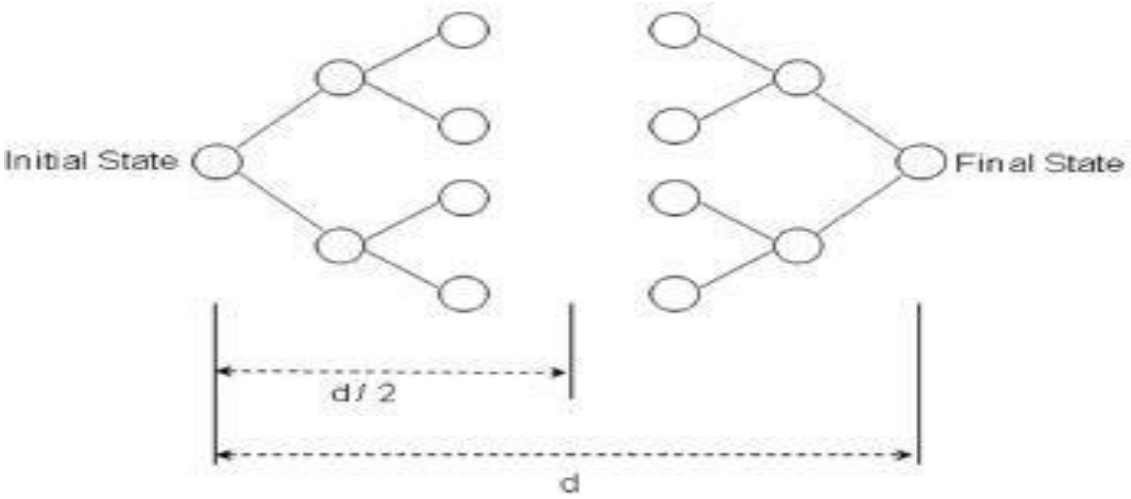

Limit = 0
Limit = 1
Limit = 2
Limit = 3

What is the difference between DLS and Iterative Deepening?

DLS is a search strategy resulting when we limit the depth of the Search and use the modest memory requirements of DFS.
IDS works by looking for the depth d, thus starting with depth limit 0 and make a DFS and if the search failed it increase the depth limit by 1 and try a DFS again with depth 1

Bidirectional Search

- As the name suggests, bidirectional search suggests to run 2 simultaneous searches
- One from the initial state and the other from the Final state
- Those 2 searches stop when they meet each other at some point in the middle of the graph.



Bidirectional Search

- Completeness:

  Bidirectional search is complete when we use BFS in both searches

- Optimality:

  Like the completeness, bidirectional search is optimal when BFS is used.

- Time/Space Complexity : $O(b^{d/2})$

Drawbacks of uniformed search

- Criterion to choose next node to expand is limited.
- Does not exploit the structure of the problem.
- One may not hunche about what can be a good move.

Which Search to Use

- If you have a good heuristic, obviously you want to use heuristic search but for some domains good heuristics are hard to produce
- If not, there are memory and time considerations
  - BFS and the like are guaranteed to find short paths, but use a lot of memory and are slow
  - DFS is much faster, but isn't guaranteed to find a solution
  - Even for heuristic search we sometimes just do the equivalent of DFS on the heuristic value. This is known as greedy search

Assignment:

Compare all the Search techniques

Informed Search (Heuristic Search)

- Informed search have problem specific knowledge apart from problem definition.
- They use experimental algorithm which improves efficiency of search process.
- The idea is to develop a domain specific heuristic function h(n) where h(n) guesses the cost of getting to the goal from node n.

Heuristic Function

The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal

The heuristic function is denoted by h(n)

Best-First Search (BFS)

- Best-First search is a graph-based heuristic search algorithm
- The name “best-first” refers to the method of exploring the node with the best “score” first.
- An evaluation function is used to assign a score to each candidate node. The evaluation function must represent some estimate of the cost of the path from state to the closest goal state

Algorithm

1. Put the initial node on a list START
2. If START = GOAL or START = EMPTY, then terminate search
3. Assign the next node as START and call this node-A
4. If A = GOAL, terminate the search with success
5. Else-if, node has successor and generate all of them. Find out how far they are from the GOAL node.
6. Sort all the children generated so far by remaining distance from the goal. Name the list as START-1. Replace START = START-1
7. Go to step-2

Types of BFS

- Greedy Best First Search
- A* Search

Greedy Best First Search

- It tries to get as close as it can to the goal.
- It expands the node that appears to be closest to the goal
- It evaluates the node by using heuristic function only.
- Evaluation function f(n) = h(n)

h(n) - is estimate of cost from n to goal

- is 0 for goal state

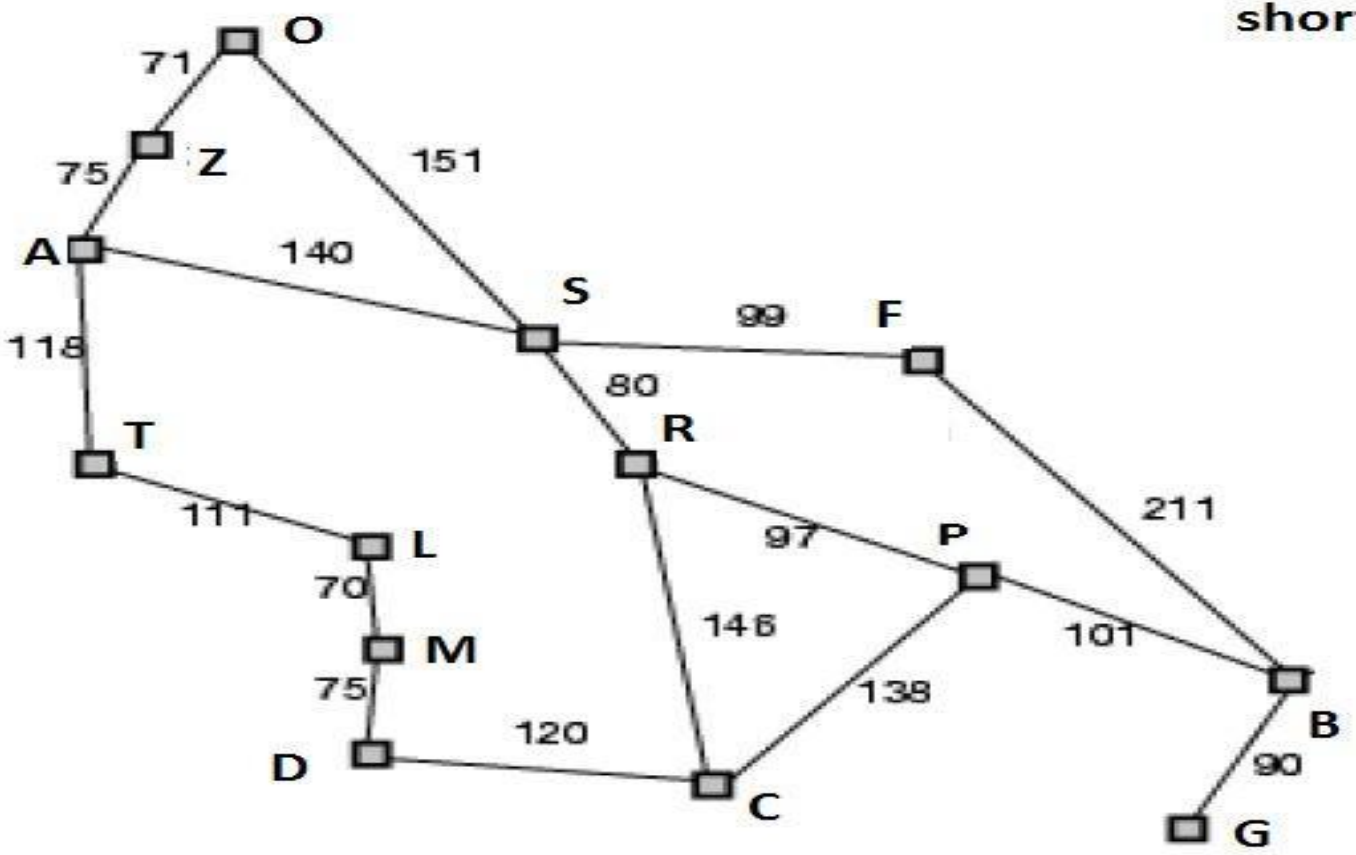
Properties

- ***Completeness:*** No – can get stuck in loops
- ***Time Complexity***: O (b^m), but a good heuristic can give dramatic improvement
- ***Space Complexity***: O (b^m), keeps all nodes in memory
- ***Optimality***: No

Applications:

This algorithm is used in Huffman encoding, minimum spanning tree, Dijkstra's algorithm etc.

Given following graph of cities, starting at City “A”, problem is to reach to the “B”

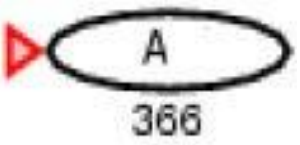


shortest line distance to B

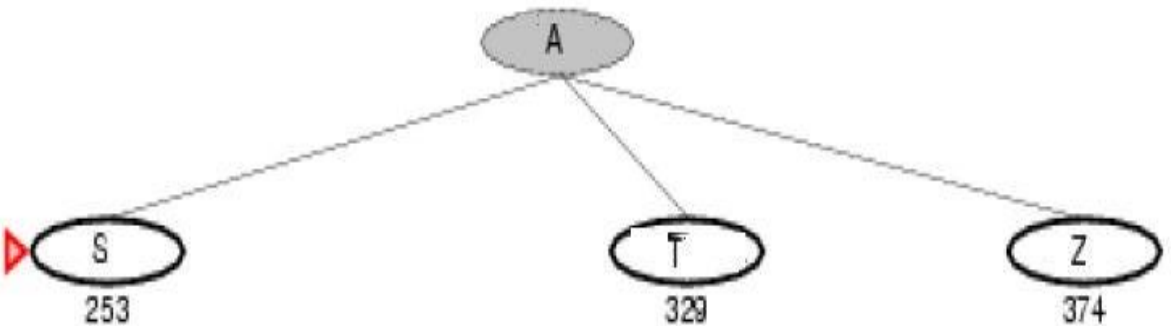
A	366
B	0
C	160
D	242
F	176
G	77
L	244
M	241
O	380
P	10
R	193
S	253
T	329
Z	374

Solution using greedy best first can be as below:

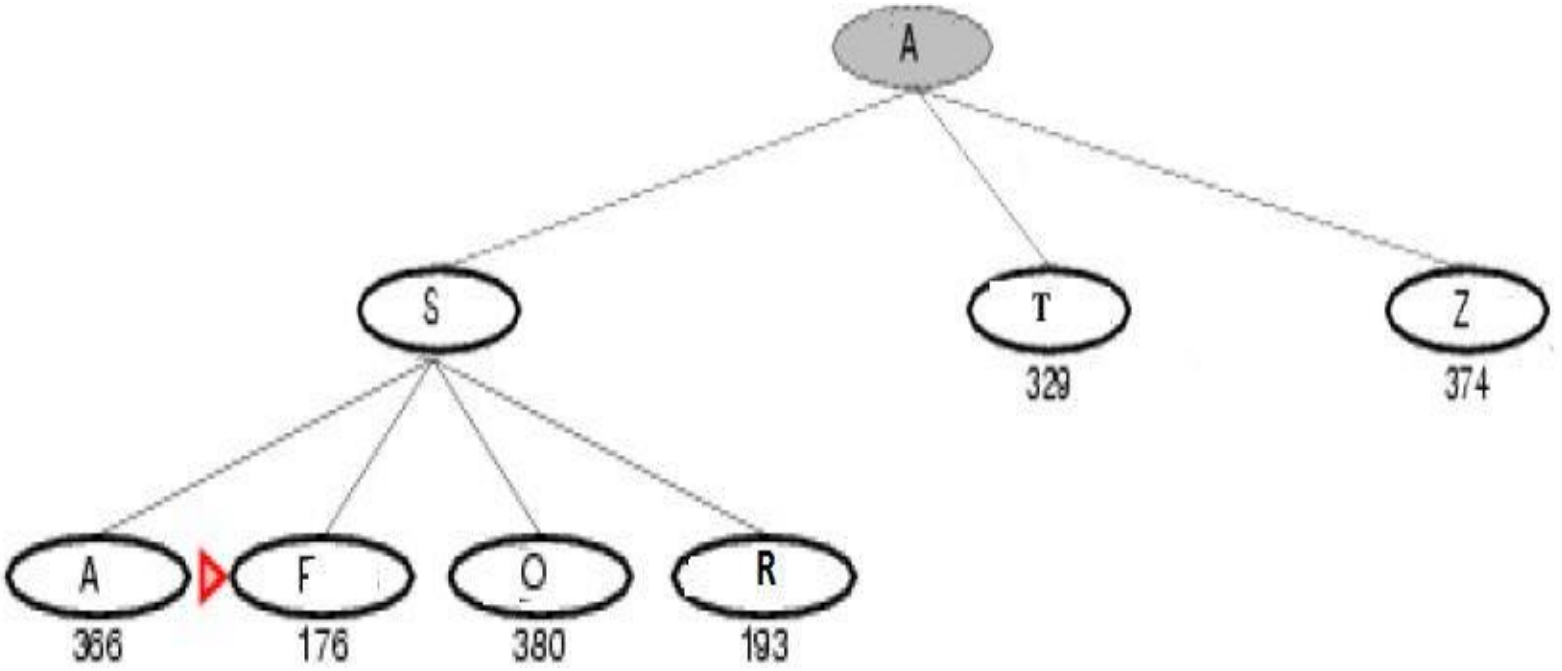
Step 1: Initial State



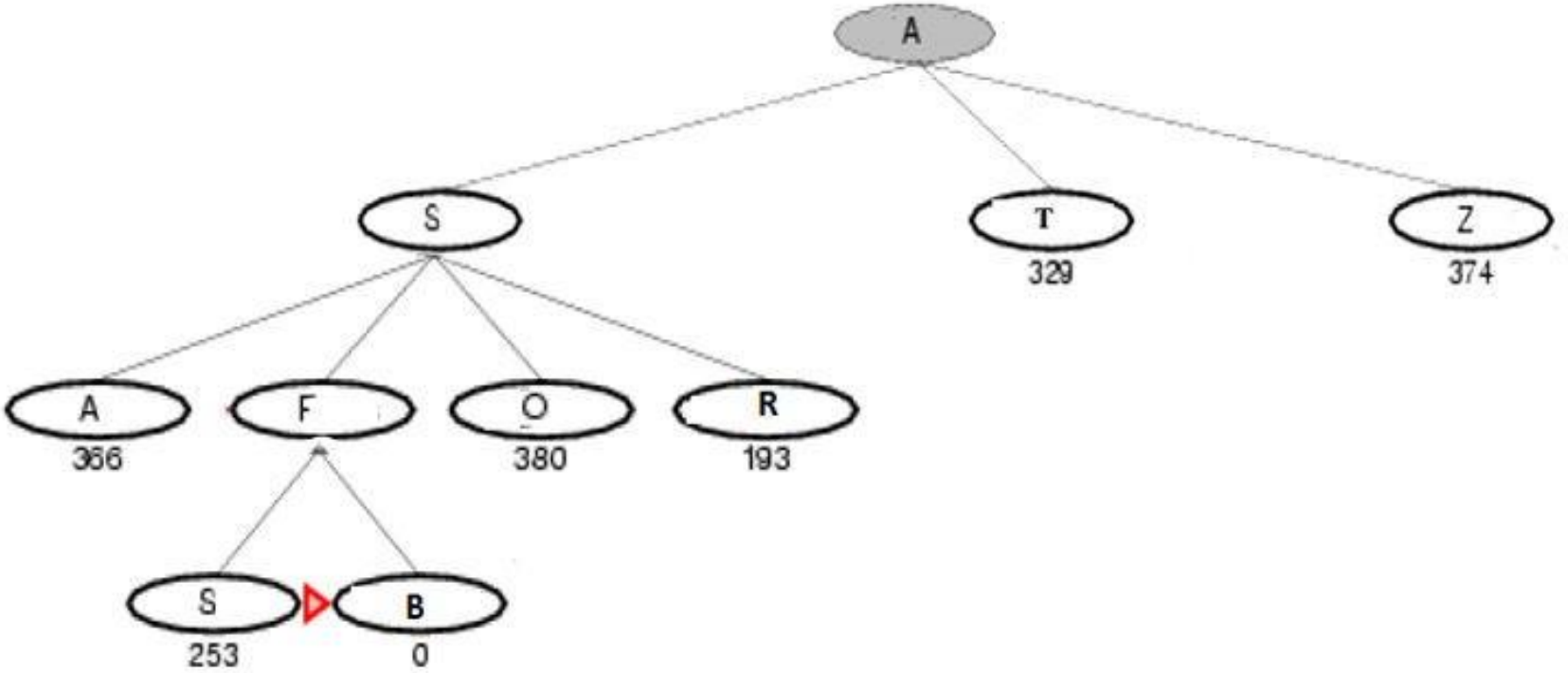
Step 2: After expanding A



Step 3: After expanding S



Step 3: After expanding F



A* Search

- It finds a minimal cost-path joining the start node and a goal node for node n.
- Evaluation function: f(n) = g(n) + h(n)

  Where,
  - g(n) = cost so far to reach n from root
  - h(n) = estimated cost to goal from n

  f(n) = estimated total cost of path through n to goal.
- Avoid expanding paths that are already expensive
- The main drawback of A* algorithm and indeed of any best-first search is its memory requirement.

Admissible heuristics

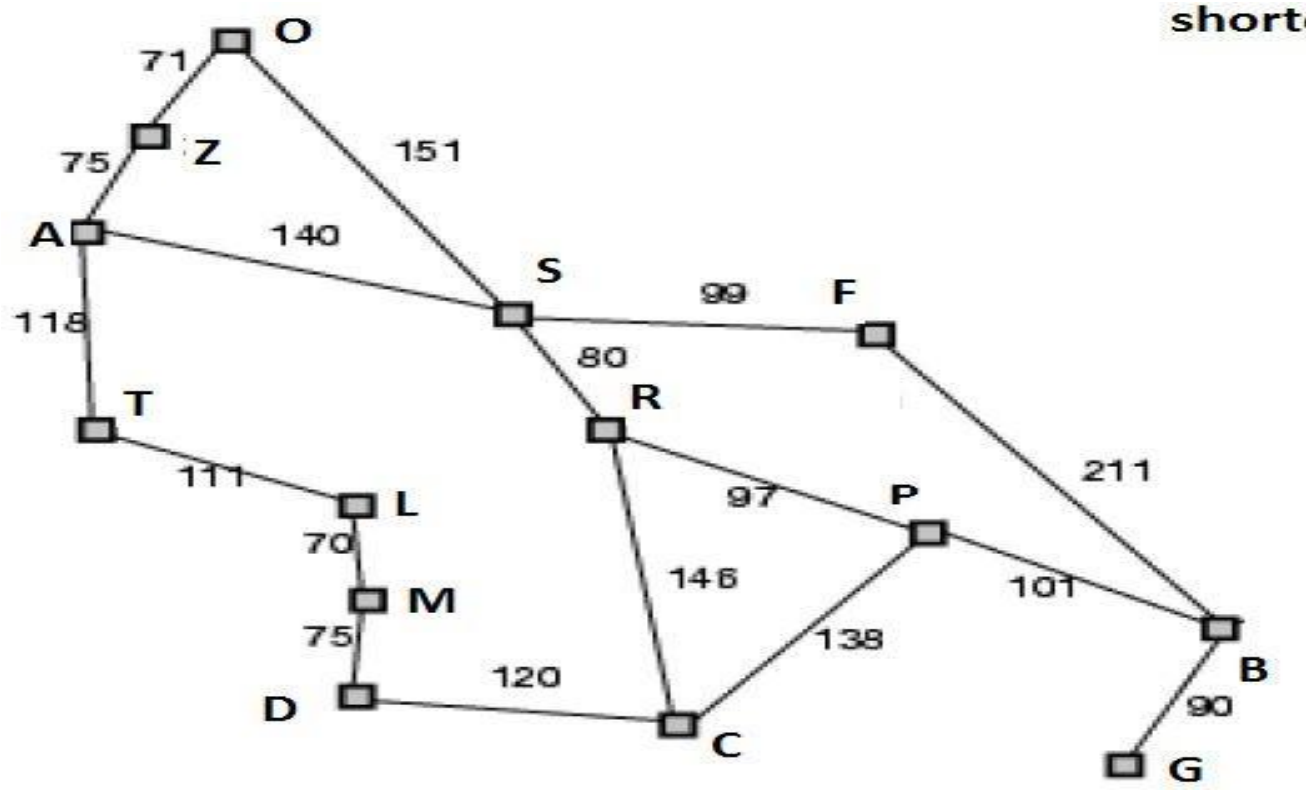
- A heuristic h(n) is admissible if for every node n,$h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Thus, f(n) = g(n) + h(n) never overestimates the true cost of a solution

A heuristic function is said to be consistent, or monotone, if its estimate is always less than or equal to the estimated distance from any neighboring vertex to the goal.

A heuristic is consistent if for every node n, every successor n' of n if

$$f(n') \geq f(n)$$

A* search example (Find path from A to B)

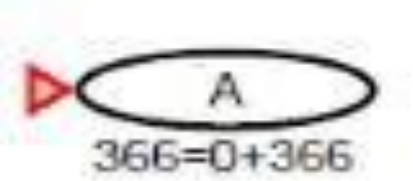


shortest line distance to B

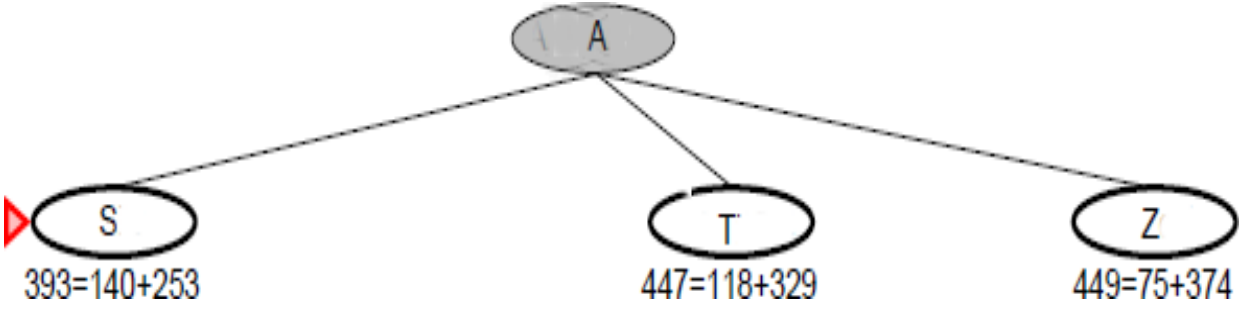
A	366
B	0
C	160
D	242
F	176
G	77
L	244
M	241
O	380
P	10
R	193
S	253
T	329
Z	374

Here, evaluate nodes connected to source. Evaluate f(n)=g(n)+h(n) for each node. Select node with lowest f(n) value.

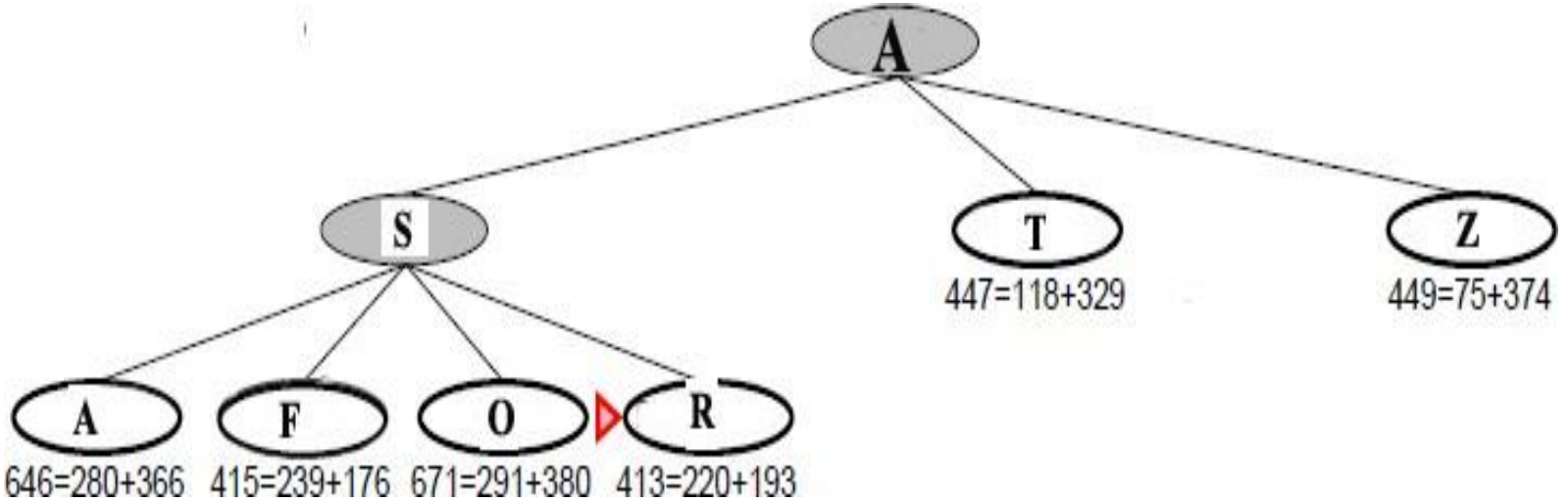
Step 1:



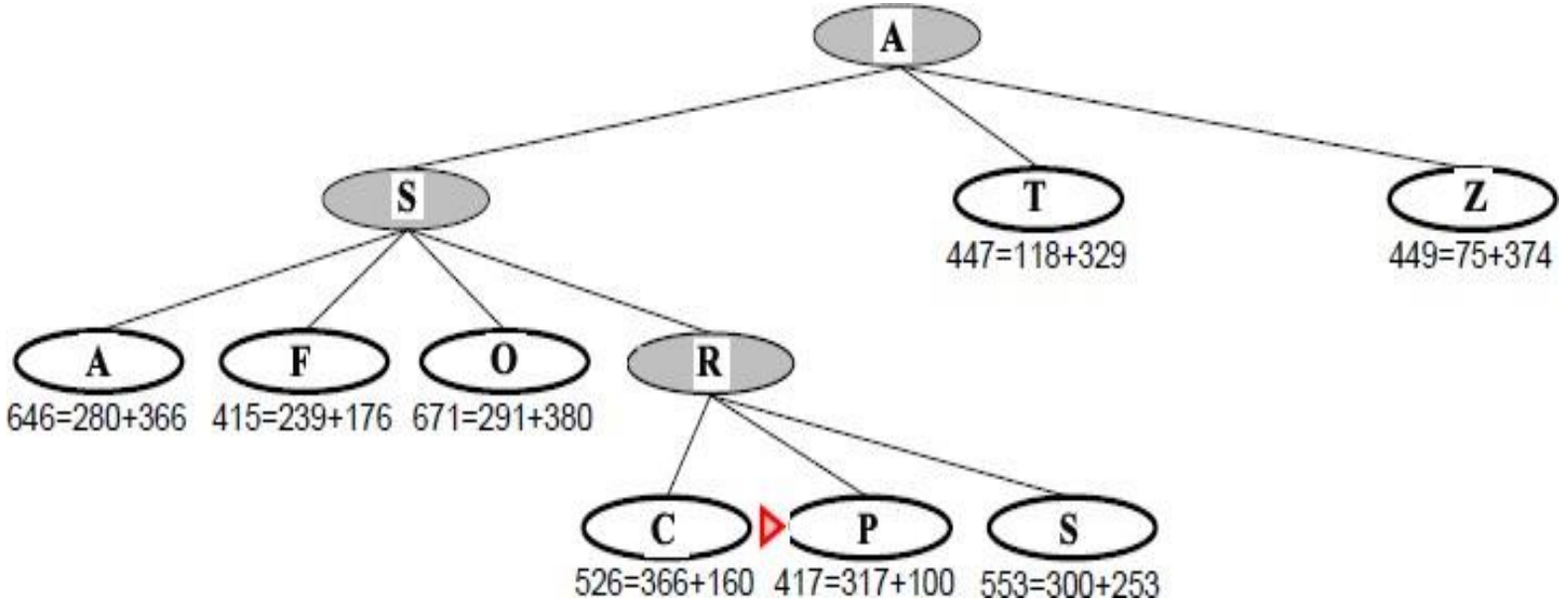
Step 2



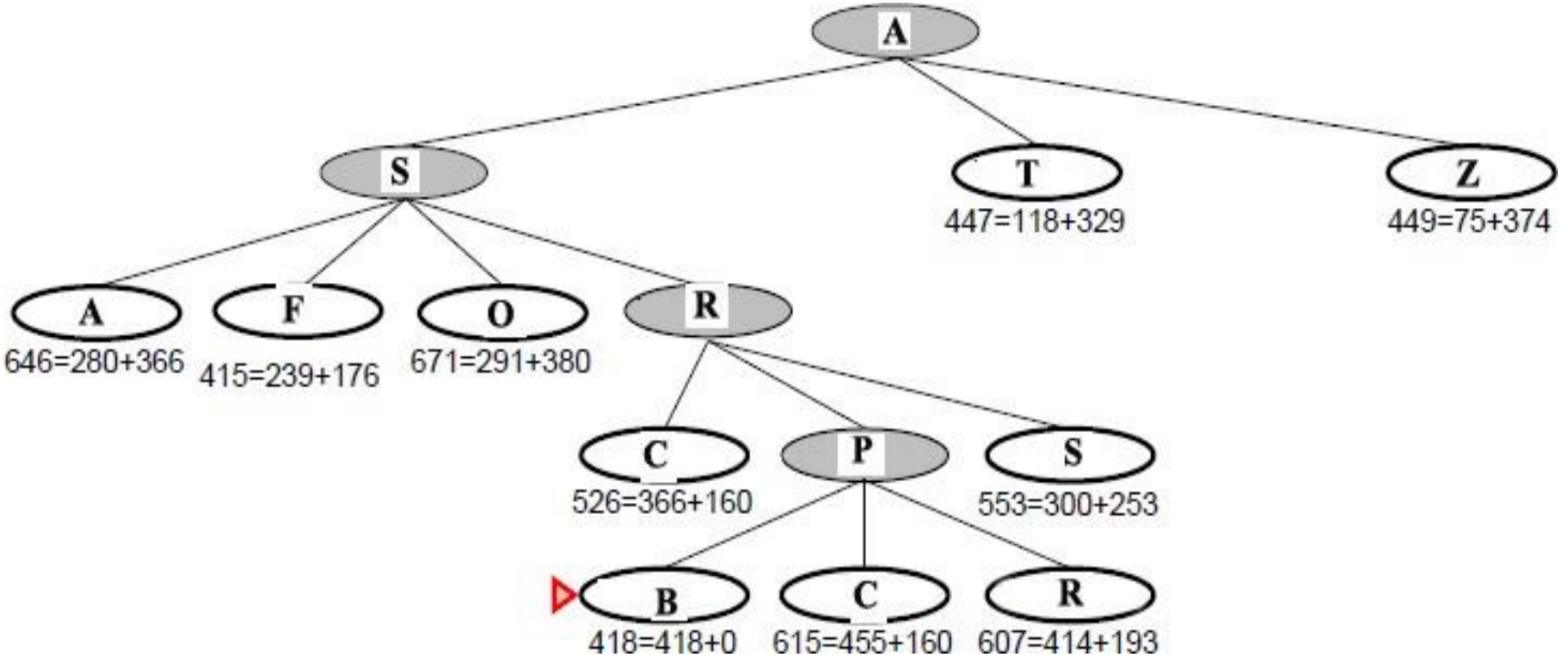
Step 3:


A
S
T
Z
447=118+329
449=75+374
A
F
O
R
646=280+366
415=239+176
671=291+380
413=220+193

Step 4:

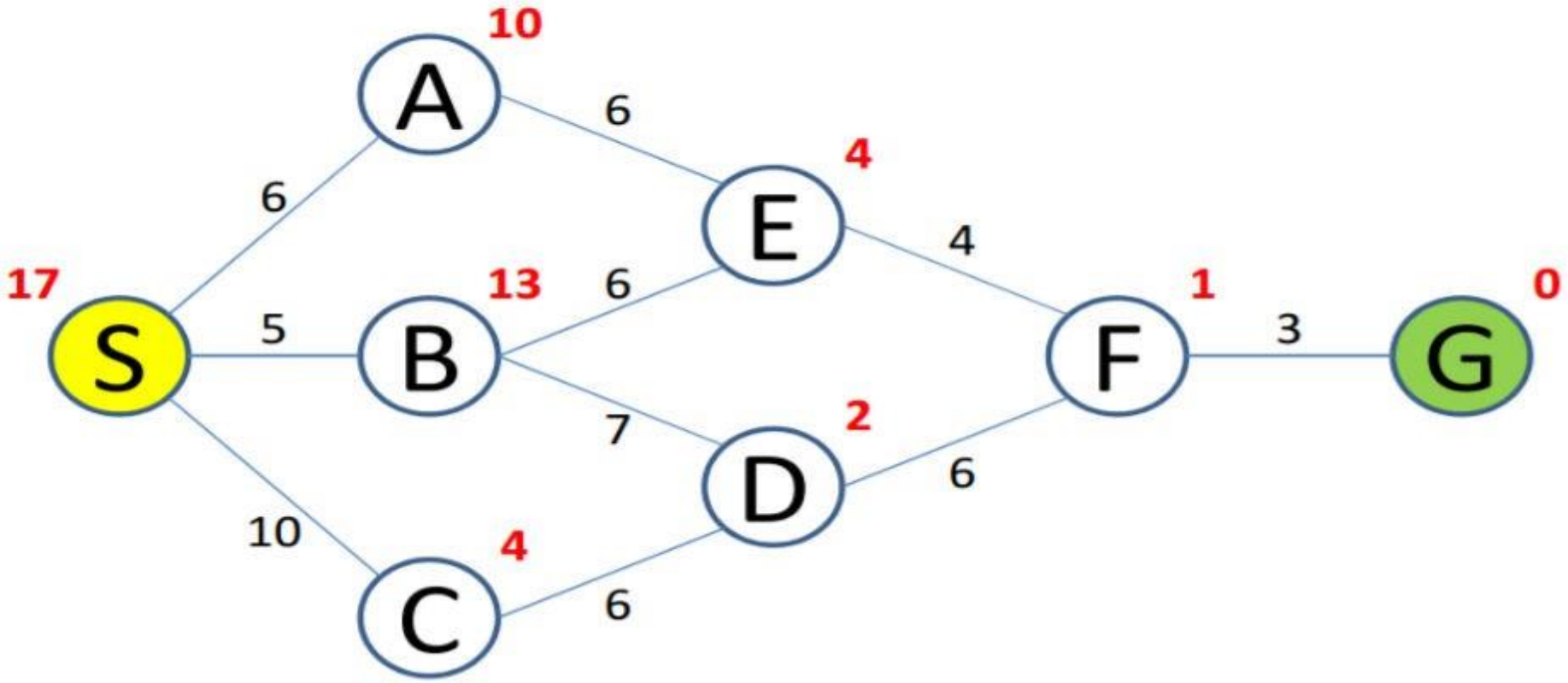

A
S
T
447=118+329
Z
449=75+374
A
646=280+366
F
415=239+176
O
671=291+380
R
C
526=366+160
P
417=317+100
S
553=300+253

Step 5:


A
S
T
447=118+329
Z
449=75+374
A
646=280+366
F
415=239+176
O
671=291+380
R
C
526=366+160
P
S
553=300+253
B
418=418+0
C
615=455+160
R
607=414+193

Assignment

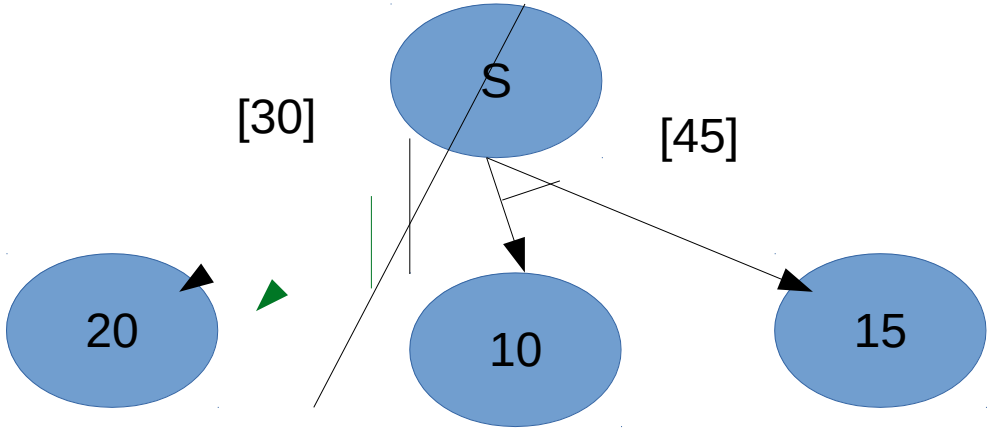
Perform Greedy and A* Search



AO* Searching

- AO* Search is a type of heuristic search algorithm .
- AO* Search is used when problems can be divided into sub parts and which can be combined
- AO* in artificial intelligence is represented using AND OR graph or AND OR tree
- AO* have one or more and arc in it

Edge = 10



Hill Climbing Search

- Hill climbing is an extension of depth-first search which uses some knowledge such as estimates of the distance of each node from the goal to improve the search
- It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value.
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next

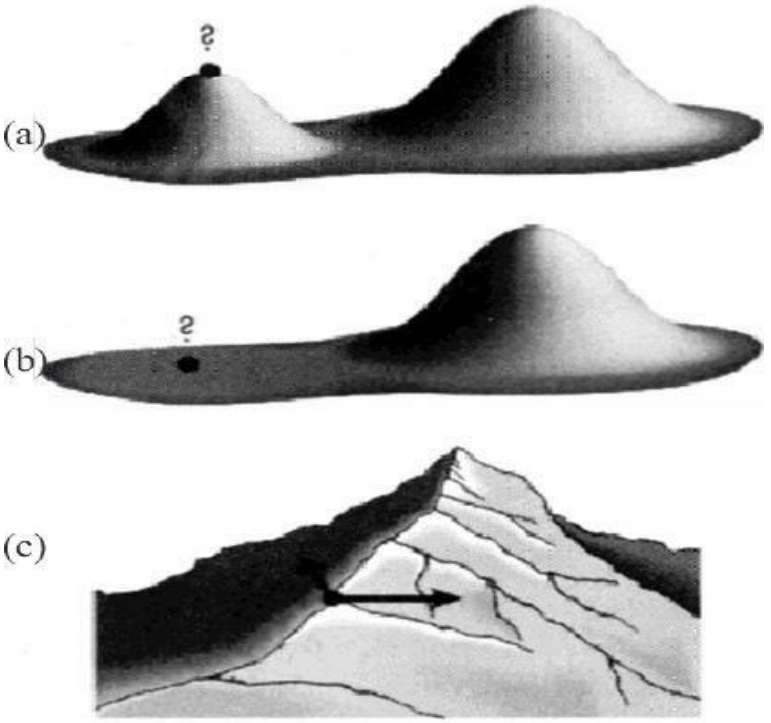


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Hill Climbing Search

- Drawbacks
  - Local Maxima

    A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  - Plateaus: A plateau is an area of the searchspace where evaluation function is flat, thusrequiring random walk
  - Ridges: Where there are steep slopes and the search direction is not towards the top but towards the side

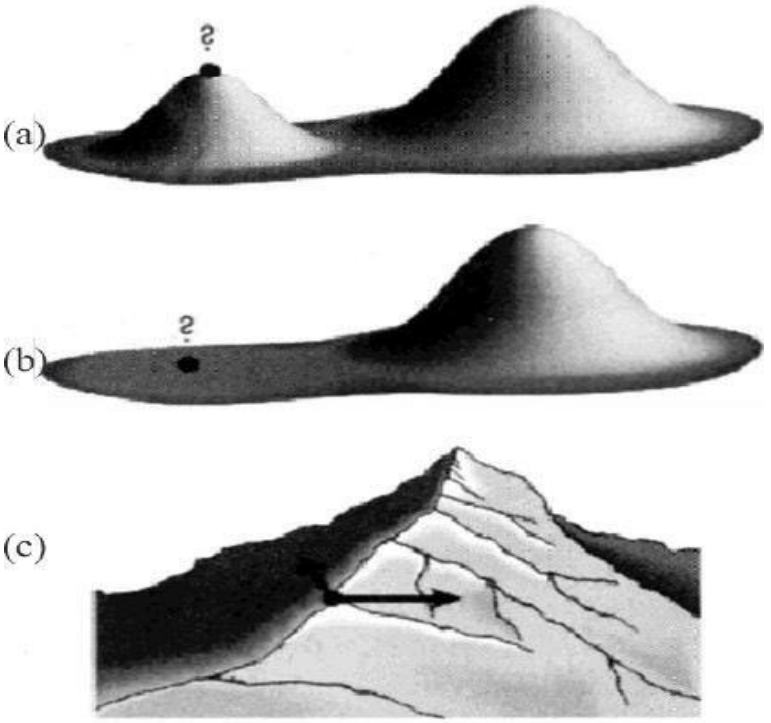


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Hill Climbing Search

- **Remedies**

  - **Back tracking for local maximum:**
    The back tracking help in undoing what is been done so far and permit to try totally different part to attain the global peak.
  - **Big Jump:**
    A big jump is the solution to escape from plateaus because all neighbors' points have same value using the greedy approach
  - **Random restart:**
    Keep restarting the search from random locations until a goal is found

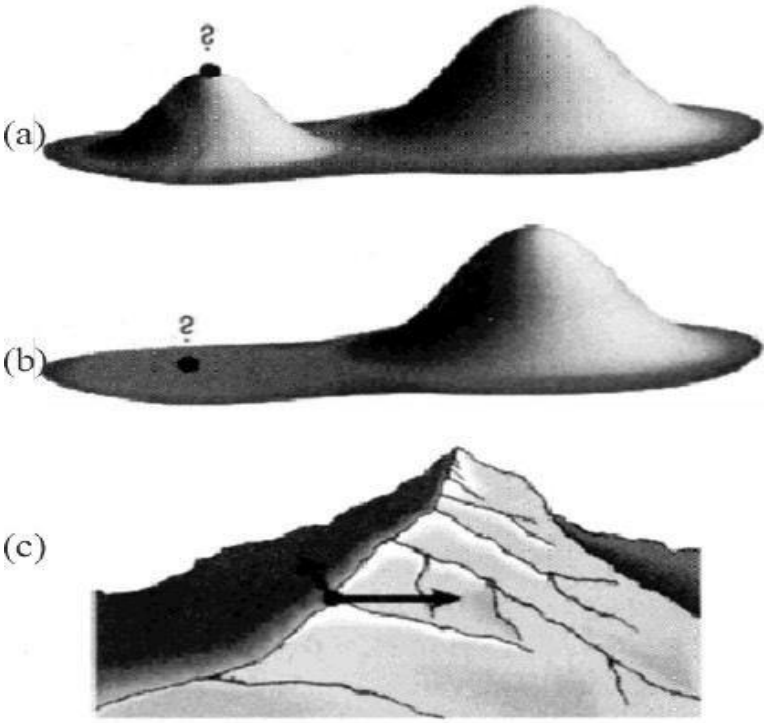
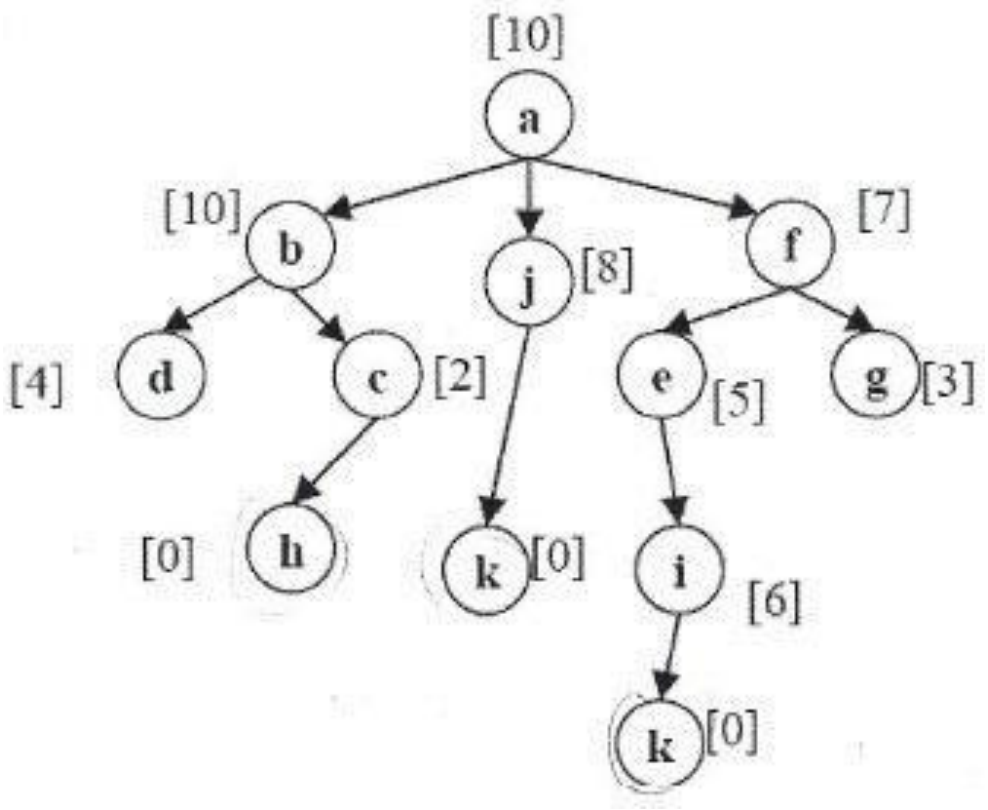


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Why Hill Climbing is not Complete?

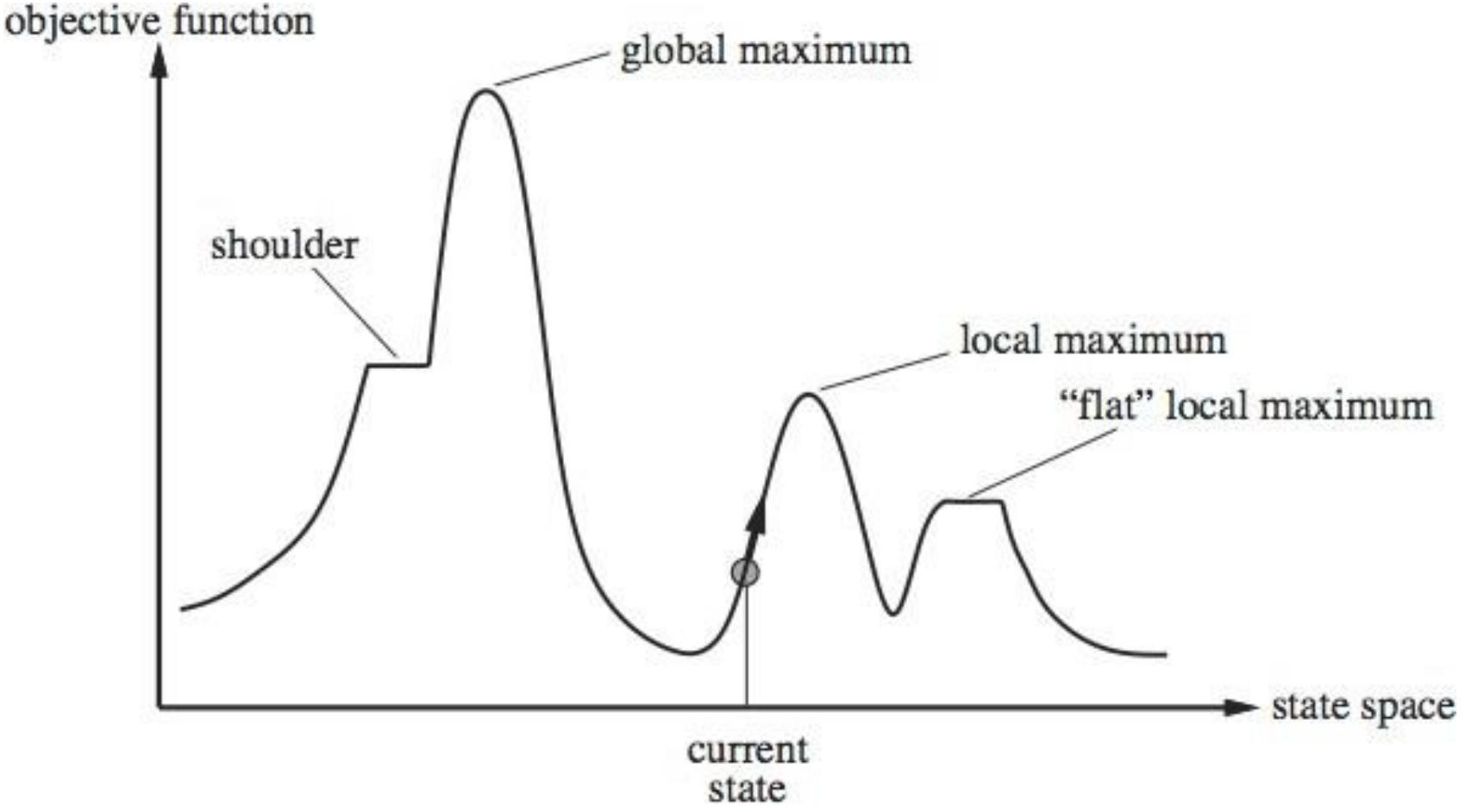
- Hill-climbing always attempts to make changes thatimprove the current state.
- The main problem that hill climbing can encounter is that of local maxima. This occurs when the algorithm stops making progress towards an optimal solution; mainly dueto the lack of immediate improvement in adjacent states.

Problems in Hill Climbing



Here, "a" is initial and h and k are final states

- We start a-> f-> g and then what ??finish(without result)
- A common way to avoid getting stuck in local maxima with Hill Climbing is to use random restarts. In your example if G is a local maxima, the algorithmwould stop there and then pick another random node to restartfrom. So if J or C were picked (orpossibly A, B, or D), this would find the global maxima in H or K


objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current
state

Simulated Annealing

- Simulated Annealing escapes local maxima by allowing some "bad" moves but gradually decrease their frequency.
- Instead of restarting from a random point, we allow the search to take some downhill steps to try to escape local maxima

Means Ends Analysis

- Means ends analysis is the problem solving techniques used commonly in Artificial intelligence for limiting search in AI program.
- The MEA technique as a problem-solving strategy was first introduced in 1961 by Allen Newell and Herbert A. Simon in their computer problem-solving program General Problem Solver (GPS).
- The MEA technique is a strategy to control search in problem-solving. Given a current state and a goal state, an action is chosen which will reduce the difference between the two. The action is performed on the current state to produce a new state, and the process is recursively applied to this new state and the goal state. Note that, in order for MEA to be effective, the goal-seeking system must have a means of associating to any kind of detectable difference those actions that are relevant to reducing that difference.
-

Genetic algorithm

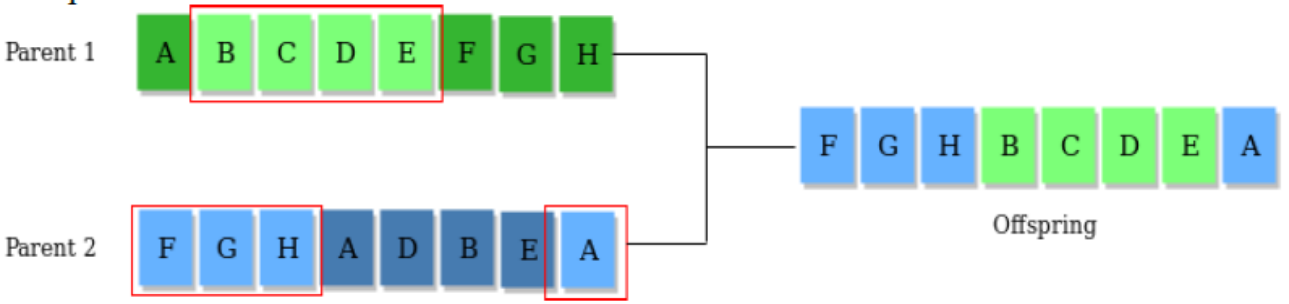
A Genetic Algorithm can be implemented using the following outline algorithm
1. Initialize a population of chromosomes
2. Evaluate each chromosome (individual) in the population
2.1. Create new chromosomes by mating chromosomes in the current population (using crossover and mutation)
2.2. Delete members of the existing population to make way for the new members
2.3. Evaluate the new members and insert them into the population
3. Repeat stage 2 until some termination condition is reached (normally based on time or number of populations produced)
4. Return the best chromosome as the solution

Operators of Genetic Algorithms

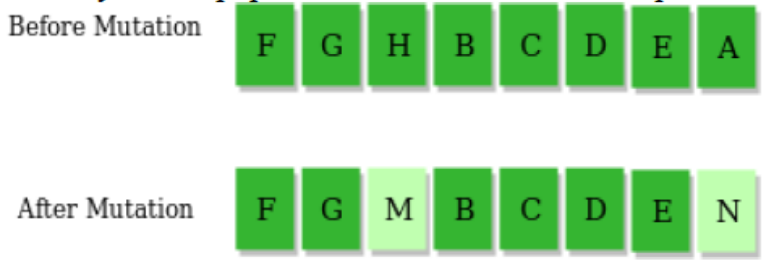
Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to the successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example –



3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence. For example –



Gradient Descent

Gradient Descent is a first-order optimization algorithm widely used in machine learning, optimization, and numerical analysis. It is particularly effective for solving **local optimization problems**, where the goal is to minimize (or maximize) a given function f(x)f(x)f(x). The algorithm works iteratively to adjust variables in the direction of the **steepest descent** of the function, as indicated by its gradient.

How Gradient Descent Works in Local Search

1. **Initialization**:
   - Start with a random initial point x_0 in the search space.
2. **Gradient Calculation**:
   - Compute the gradient $\nabla f(x_t)$ at the current point.
3. **Step Update**:
   - Update x_t by taking a step in the opposite direction of the gradient (downhill direction).
4. **Convergence Check**:
   - Repeat the process until convergence (e.g., when the gradient magnitude is close to zero or the change in $f(x)$ is negligible).

Example Problem: Minimize $f(x) = x^2 + 4x + 4$

Step-by-Step Gradient Descent

1. **Objective Function**: $f(x) = x^2 + 4x + 4$
    - Derivative (Gradient): $f'(x) = 2x + 4$
2. **Initialization**:
    - Start with an initial guess $x_0 = 5$.
    - Set the learning rate $\eta = 0.1$.

3. **Iterations**:
    - Compute the gradient $f'(x)$.
    - Update x_t using $x_{t+1} = x_t - \eta f'(x_t)$.

Iteration t	x_t	Gradient $f'(x_t)$	$x_{t+1} = x_t - \eta f'(x_t)$	$f(x_t)$
0	5.0	$2(5) + 4 = 14$	$5 - 0.1(14) = 3.6$	$5^2 + 4(5) + 4 = 49$
1	3.6	$2(3.6) + 4 = 11.2$	$3.6 - 0.1(11.2) = 2.48$	$3.6^2 + 4(3.6) + 4 = 29.16$
2	2.48	$2(2.48) + 4 = 8.96$	$2.48 - 0.1(8.96) = 1.584$	$2.48^2 + 4(2.48) + 4 = 17.71$
3	1.584	$2(1.584) + 4 = 7.168$	$1.584 - 0.1(7.168) = 0.867$	$1.584^2 + 4(1.584) + 4 = 11.52$

4. **Convergence**:
    - As t increases, x_t approaches the minimum point $x = -2$, and $f(x)$ approaches the minimum value $f(-2) = 0$.

Iteration t	x_t	Gradient $f'(x_t)$	$x_{t+1} = x_t - \eta f'(x_t)$	$f(x_t)$
0	5.0	$2(5) + 4 = 14$	$5 - 0.1(14) = 3.6$	$5^2 + 4(5) + 4 = 49$
1	3.6	$2(3.6) + 4 = 11.2$	$3.6 - 0.1(11.2) = 2.48$	$3.6^2 + 4(3.6) + 4 = 29.16$
2	2.48	$2(2.48) + 4 = 8.96$	$2.48 - 0.1(8.96) = 1.584$	$2.48^2 + 4(2.48) + 4 = 17.71$
3	1.584	$2(1.584) + 4 = 7.168$	$1.584 - 0.1(7.168) = 0.867$	$1.584^2 + 4(1.584) + 4 = 11.52$

4. **Convergence**:
    - As t increases, x_t approaches the minimum point $x = -2$, and $f(x)$ approaches the minimum value $f(-2) = 0$.

Adversarial Search

- Competitive environments in which the agents goals are in conflict, give rise to adversarial (oppositional) search, often known as games.
- In AI, games means fully observable environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite.

  E.g. If first player wins, the other player necessarily loses.

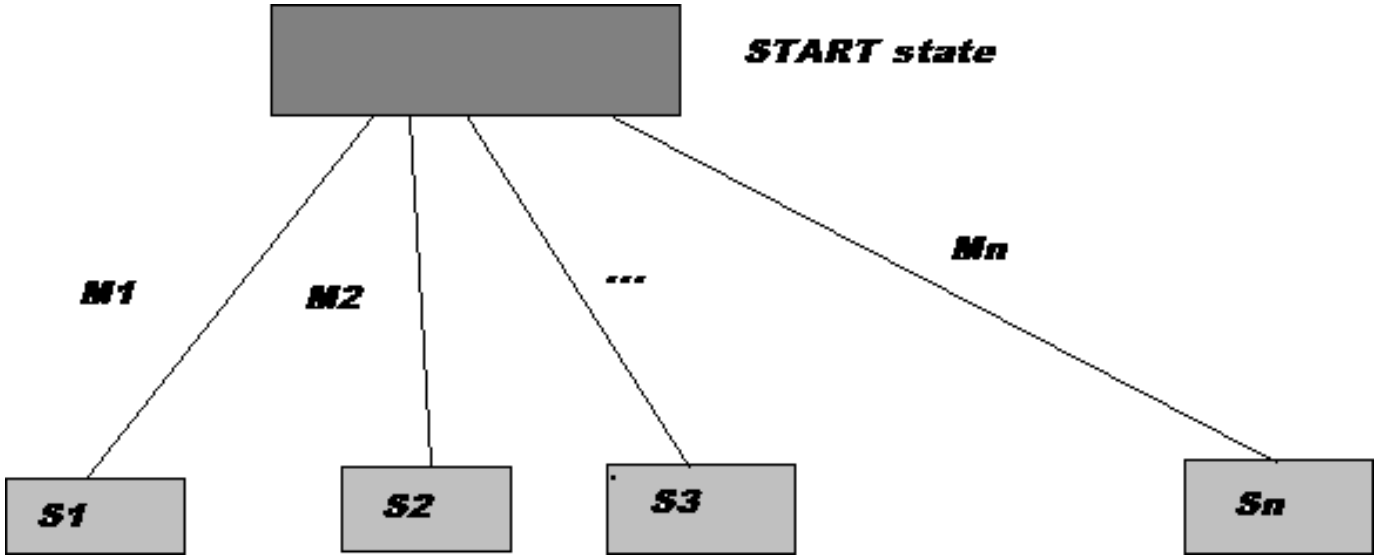
Game Playing

A game can be formally defined as a kind of search problem as below:

- **Initial state**: It includes the board position and identifies the player's to move.
- **Successor function**: It gives a list of (move, state) pairs each indicating a legal move and resulting state.
- **Terminal test**: This determines when the game is over. States where the game is ended are called terminal states.
- **Utility function**: It gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0).

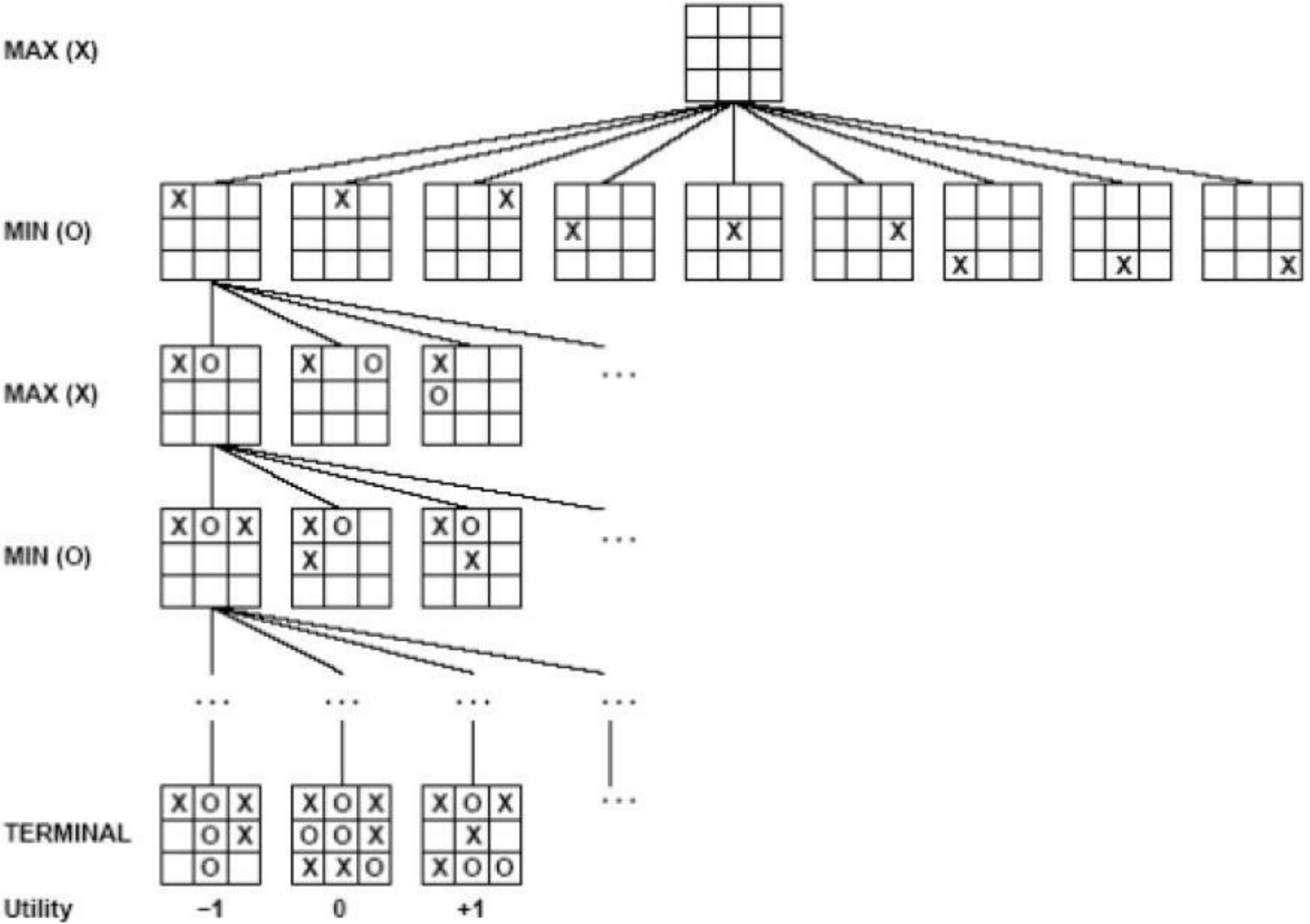
Game Trees

- We can represent all possible games(of a given type) by a directed graph often called a game tree.
- The nodes of the graph represent the states of the game. The arcs of the graph represent possible moves by the players (+ and -)



Example: Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line.
The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.
A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X).
The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
−1
0
+1

Mini max Algorithm

- It is a recursive algorithm for choosing the next move in a n-player game, usually a two player game
- The value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach the position.
- The player then makes the move that maximizes the minimum value of the position from the opponents possible moves called maximizing player and other player minimize the maximum value of the position called minimizing player.

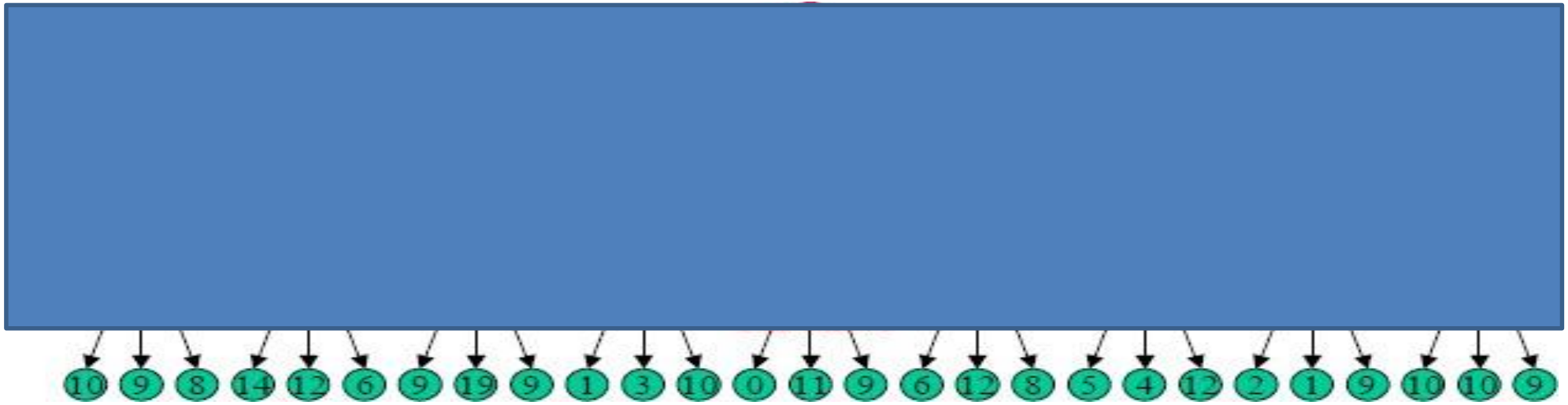
Mini Max Game Search

- It is a Depth-first search with limited depth.
- Assume the opponent will make the best move possible.
- Algorithm
  - *mini max (player, board)*
  - *if(game over in current board position)*
    return winner
  - *if(max's turn)*
    return maximal score of calling minimax
    else (min's turn)
    return minimal score of calling minimax

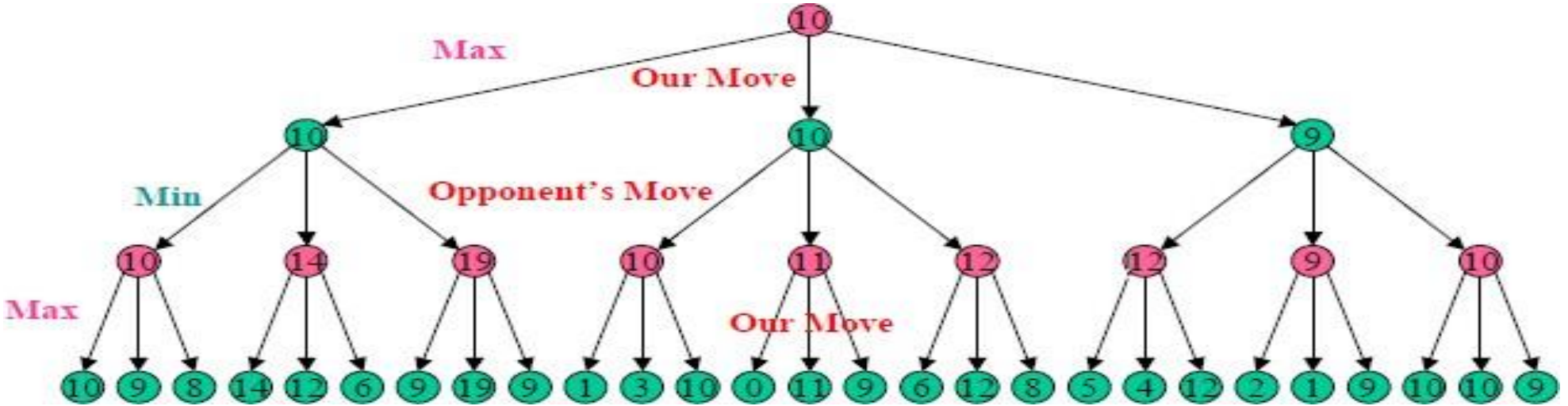
Properties of MiniMax

- ***Complete?*** Yes (if tree is finite)
- ***Optimal?*** Yes (against an optimal opponent)
- ***Time complexity?*** $O(b^m)$
- ***Space complexity?*** $O(b^m)$ (depth-first exploration)

We first consider games with two players; MAX and MIN. MAX moves first, and then they take turns moving until the game is over. Each level of the tree alternates, MAX is trying to maximize score, and MIN is trying to minimize MAX score in order to undermine success



We first consider games with two players; MAX and MIN. MAX moves first, and then they take turns moving until the game is over. Each level of the tree alternates, MAX is trying to maximize score, and MIN is trying to minimize MAX score in order to undermine success



The mini max algorithm returns the best move for MAX under the assumption that MIN play optimally. What happens when MIN plays sub optimally ?

Optimality is still well defined, even if the opponent isn't playing well. Moreover, if the game treeis small enough that the agent can fully explore it, then the optimal player really doesn't care what the other one does.

Let's say Max goes first. What will Max do? He will look at every possible game sequence. He will then take the action which guarantees that he will get a score of at X. No matter what Min does in subsequent moves, Min can never get a score less than X.

The **Min-Conflicts Heuristic** is a local search algorithm specifically designed for solving **Constraint Satisfaction Problems (CSPs)**. It works by minimizing the number of conflicts in a solution and is particularly effective for problems with large state spaces or a high number of constraints.

Steps in Min-Conflicts Heuristic

1. **Start with an initial (possibly random) assignment** of values to variables.
2. While a solution has not been found:
    - Select a variable that is currently **in conflict**.
    - Assign the variable a value that **minimizes the number of conflicts**.
3. **Stop** when a solution is found (i.e., no conflicts remain) or after a predefined number of iterations.

Example: N-Queens Problem

Problem:

Place 4 queens on a 4×4 chessboard such that no two queens are in the same row, column, or diagonal.

Step-by-Step Walkthrough

1. **Initial Random Assignment**:
    - Place queens randomly, one per column:

2. **Evaluate Conflicts**:
    - Conflicts arise when queens threaten each other:
        - Row conflicts: 0 (each row has only 1 queen).
        - Column conflicts: 0 (1 queen per column).
        - Diagonal conflicts: Count the number of queens on each diagonal. Example: Queen in column 2 threatens queens in column 1 and column 3.
3. **Select a Variable in Conflict**:
    - Pick one queen involved in a conflict (e.g., the queen in column 2).
4. **Minimize Conflicts**:
    - Move the selected queen to a row that minimizes the number of conflicts:

```
. Q . .
. . Q .
Q . . .
. . . Q
```

5. **Repeat**:
    - Continue selecting conflicting queens and reassigning their positions until no conflicts remain.