ARTIFICAL INTELLIGENCE

MODULE -3
SEARCHING STRATERGIES, LOCAL SEARCH ALGORITHMS

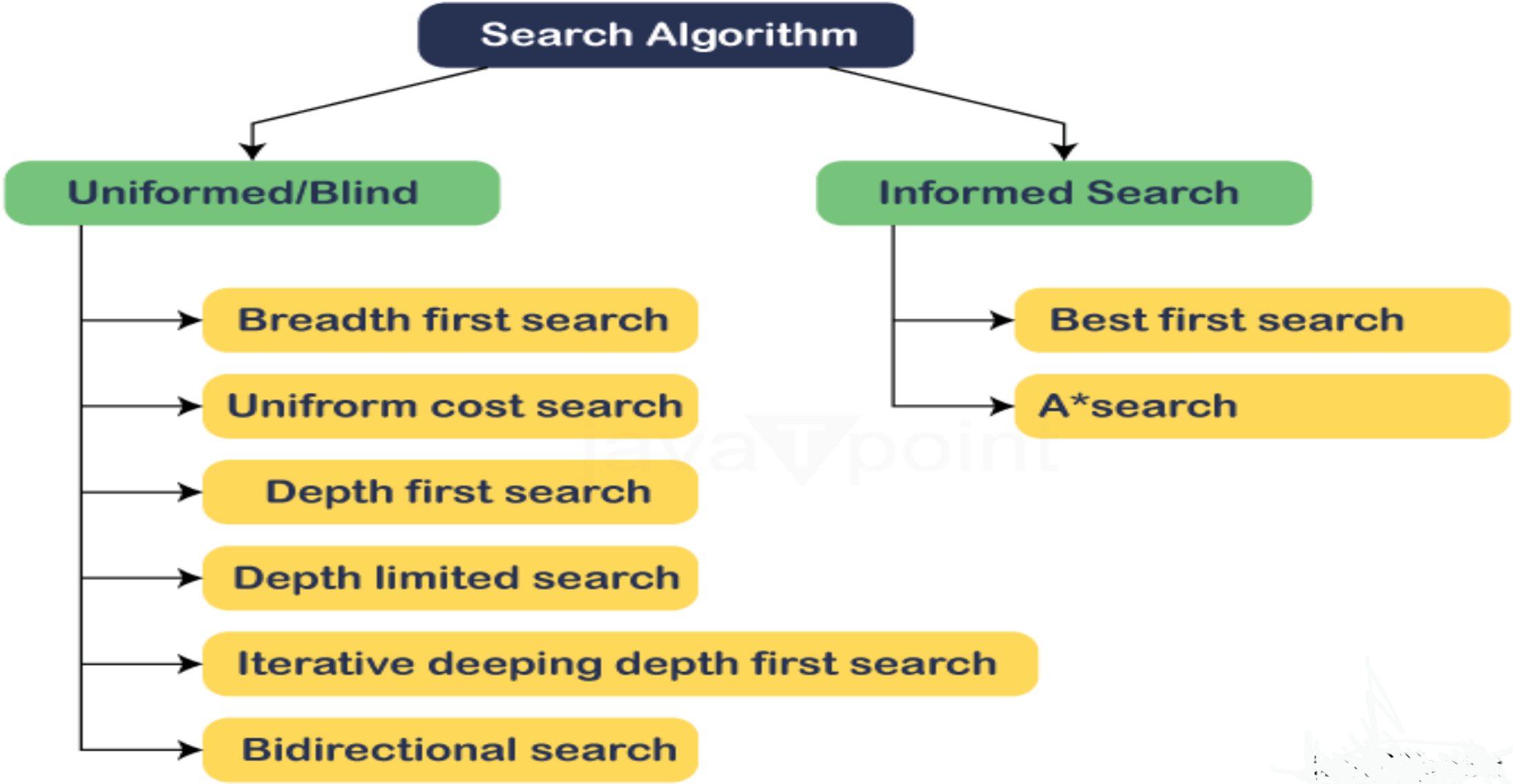
Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

A search problem consists of:

- **A State Space -** Set of all possible states where you can be.
- **A Start State -** The state from where the search begins.
- **A Goal State -** A function that looks at the current state returns whether or not it is the goal state.

The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.

This plan is achieved through search algorithms.


Search Algorithm
Uniformed/Blind
Informed Search
Breadth first search
Unifrorm cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Best first search
A*search

Uninformed Search Algorithms:

The search algorithms in this section have **no additional information on the goal node** other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions

Uninformed search is also called "**Blind search".** These algorithms can only generate the **successors and differentiate between the goal state and non goal state.**

Informed search algorithms

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can **find a solution more efficiently** than an uninformed search strategy. Informed search is also called a **Heuristic search.**

A heuristic is a way which might not always be guaranteed for ***best solutions but guaranteed to find a good solution in reasonable time.***

Informed search can solve much **complex problem which could not be solved in another way.**

An example of informed search algorithms is a traveling salesman problem.

- Greedy Search
- A* Search

Uninformed Search Algorithms

Introduction:

Uninformed search is one in which the search systems do not use any clues about the suitable area but it depend on the random nature of search. Nevertheless, they begins the exploration of search space (all possible solutions) synchronously,.

The search operation begins from the initial state and providing all possible next steps arrangement until goal is reached. These are mostly the simplest search strategies, but they may not be suitable for complex paths which involve in irrelevant or even irrelevant components.

These algorithms are necessary for **solving basic tasks or providing simple processing before passing on the data to more advanced search algorithms that incorporate prioritized information.**

Following are the various types of uninformed search algorithms:

- Breadth-first Search
- Depth-first Search
- Depth-limited Search
- Iterative deepening depth-first search
- Uniform cost search
- Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for **traversing a tree or graph**. This algorithm searches ***breadthwise in a tree or graph, so it is called breadth-first search.***
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using **FIFO queue data structure.**

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- It also helps in finding the shortest path in goal state, since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
- It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

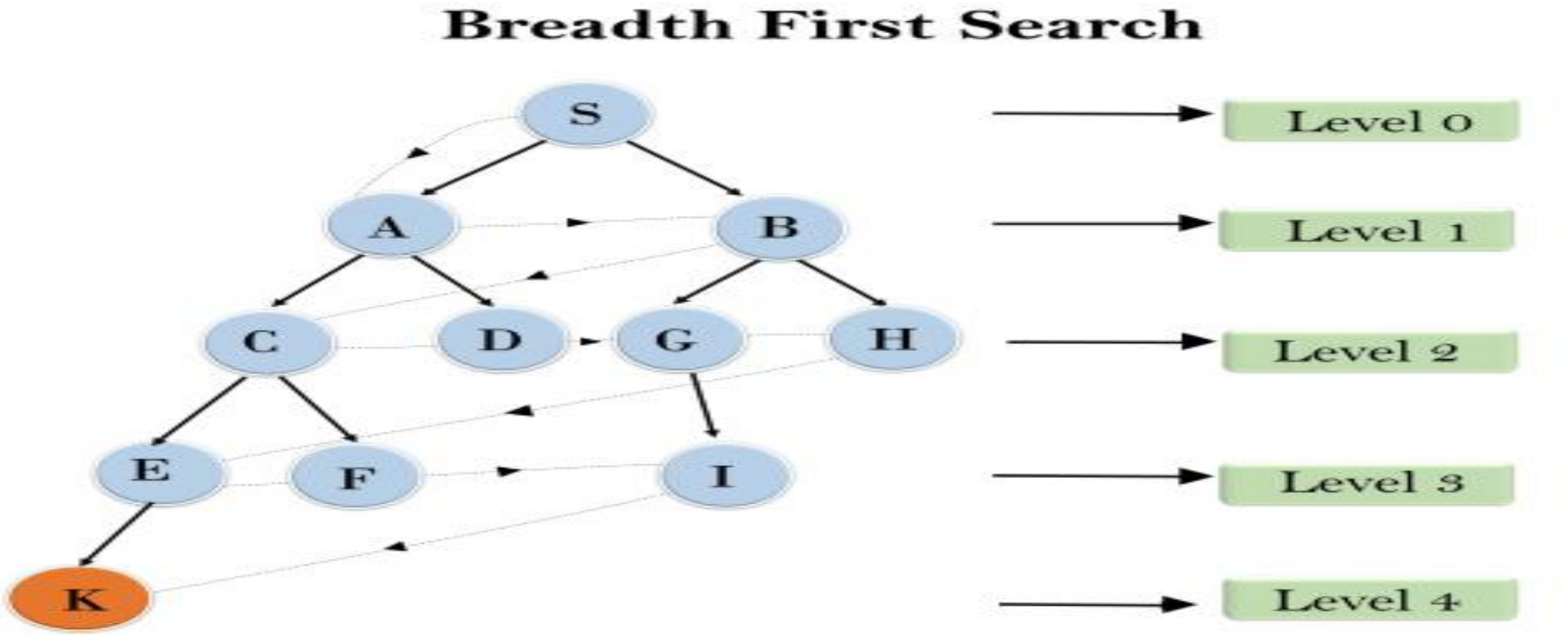
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.
- It can be very inefficient approach for searching through deeply layered spaces, as it needs to thoroughly explore all nodes at each level before moving on to the next

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b2+b3+.....+ bd= O(bd)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

Depth-first search isa recursive algorithm for traversing a tree or graph data structure.

It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.

DFS uses a stack data structure for its implementation.

The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- With the help of this we can stores the route which is being tracked in memory to save time as it only needs to keep one at a particular time.

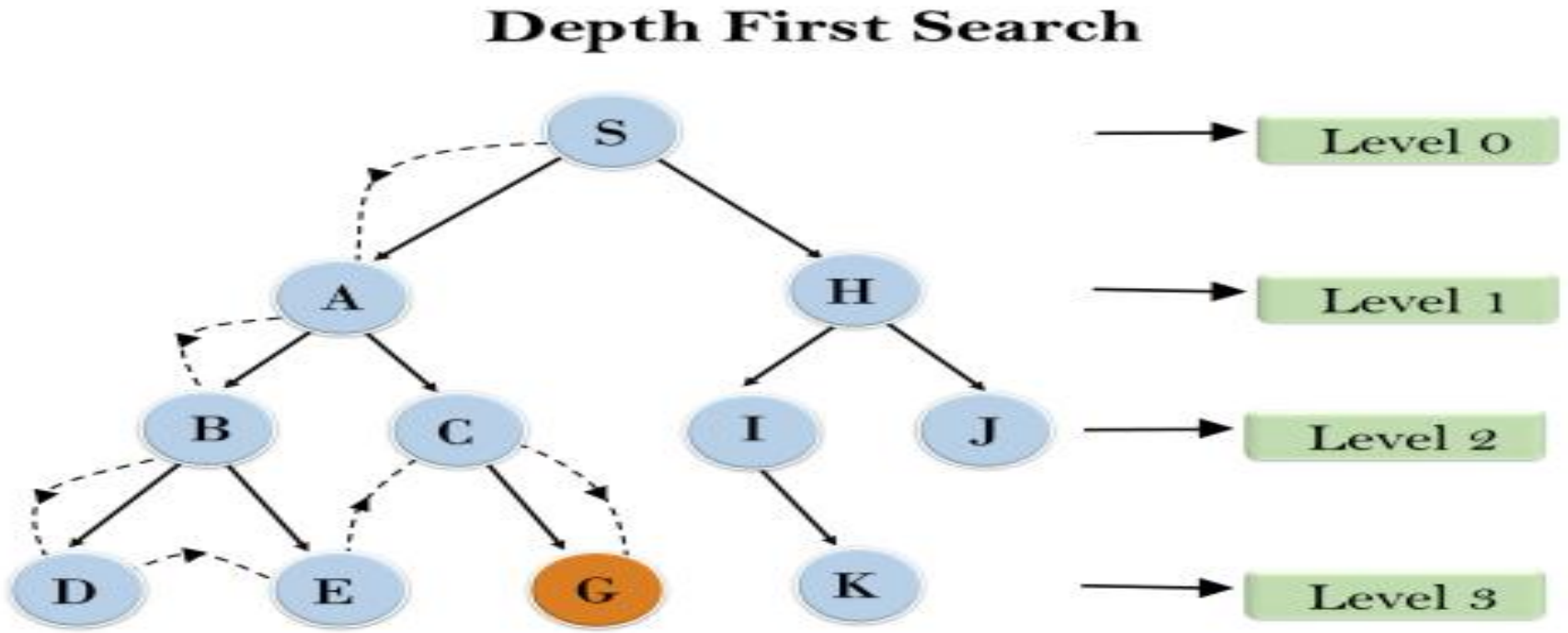
Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.
- The depth-first search (DFS) algorithm does not always find the shortest path to a solution.

Example: In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing **E, it will backtrack the tree as E has no other successor and still goal node is not found**. After **backtracking it will traverse node C and then G, and here it will terminate as it found goal node.**



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+ n2+ n3 +.....+ nm=O(nm)$$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is - **O(bm)**

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a **predetermined limit**. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the **depth limit will treat as it has no successor nodes further.**

Depth-limited search can be terminated with two Conditions of failure:

- **Standard failure value:** It indicates that problem does not have any solution.
- **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

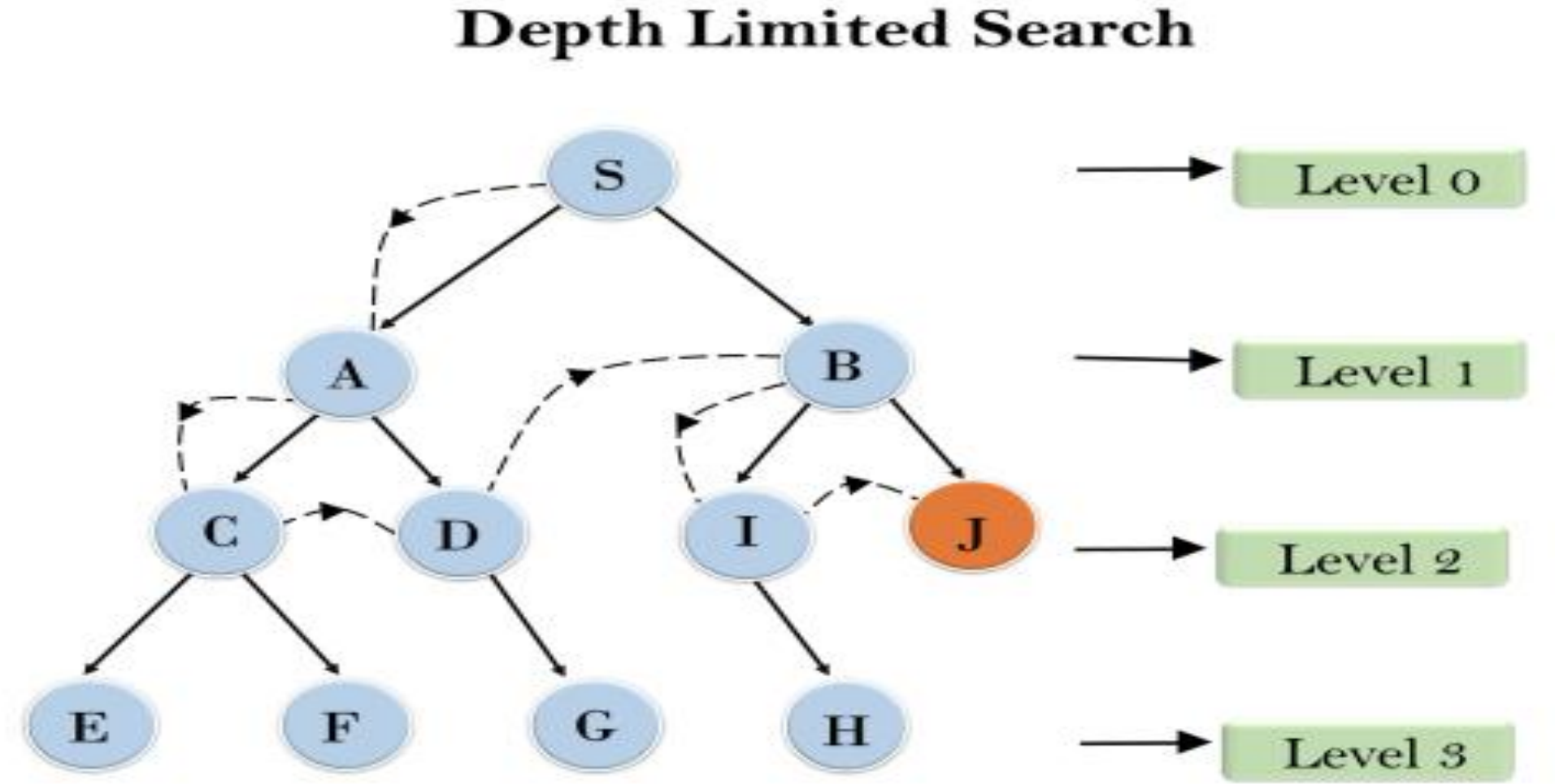
Advantages:

- Depth-Limited Search will restrict the search depth of the tree, thus, the algorithm will require fewer memory resources than the straight BFS (Breadth-First Search) and IDDFS (Iterative Deepening Depth-First Search). After all, this implies automatic selection of more segments of the search space and the consequent why consumption of the resources.
- When there is a leaf node depth which is as large as the highest level allowed, do not describe its children, and then discard it from the stack.
- Depth-Limited Search does not explain the infinite loops which can arise in classical when there are cycles in graph of cities.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.
- The effectiveness of the Depth-Limited Search (DLS) algorithm is largely dependent on the depth limit specified. If the depth limit is set too low, the algorithm may fail to find the solution altogether.

Example:


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b\ell)$ where b is the branching factor of the search tree, and l is the depth limit.

Space Complexity: Space complexity of DLS algorithm is $O(b\times\ell)$ where b is the branching factor of the search tree, and l is the depth limit.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell>d$.

4. Iterative deepening depth-first Search:

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain **"depth limit",** and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Here are the steps for Iterative deepening depth first search algorithm:

- Set the depth limit to 0.
- Perform DFS to the depth limit.
- If the goal state is found, return it.
- If the goal state is not found and the maximum depth has not been reached, increment the depth limit and repeat steps 2-4.
- If the goal state is not found and the maximum depth has been reached, terminate the search and return failure.

Advantages:

It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

- It is a type of straightforward which is used to put into practice since it builds upon the conventional depth-first search algorithm.
- It is a type ofsearch algorithm which provides guarantees to find the optimal solution, as long as the cost of each edge in the search space is the same.
- It is a type ofcomplete algorithm, and the meaning of this is it will always find a solution if one exists.

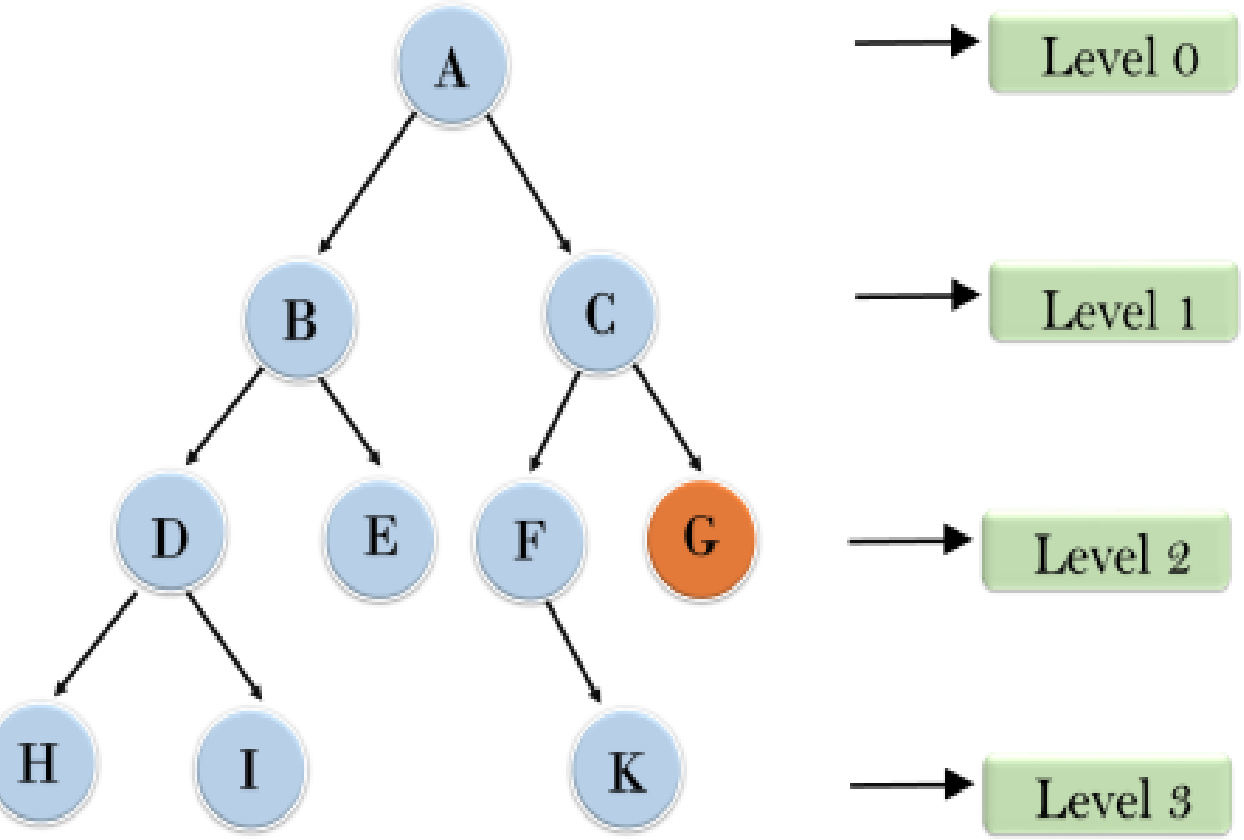
Disadvantage

The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(bd).**

Space Complexity: The space complexity of IDDFS will be **O(bd).**

5. Bidirectional Search Algorithm:

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as **forward-search and other from goal node called as backward-search,** to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other. Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

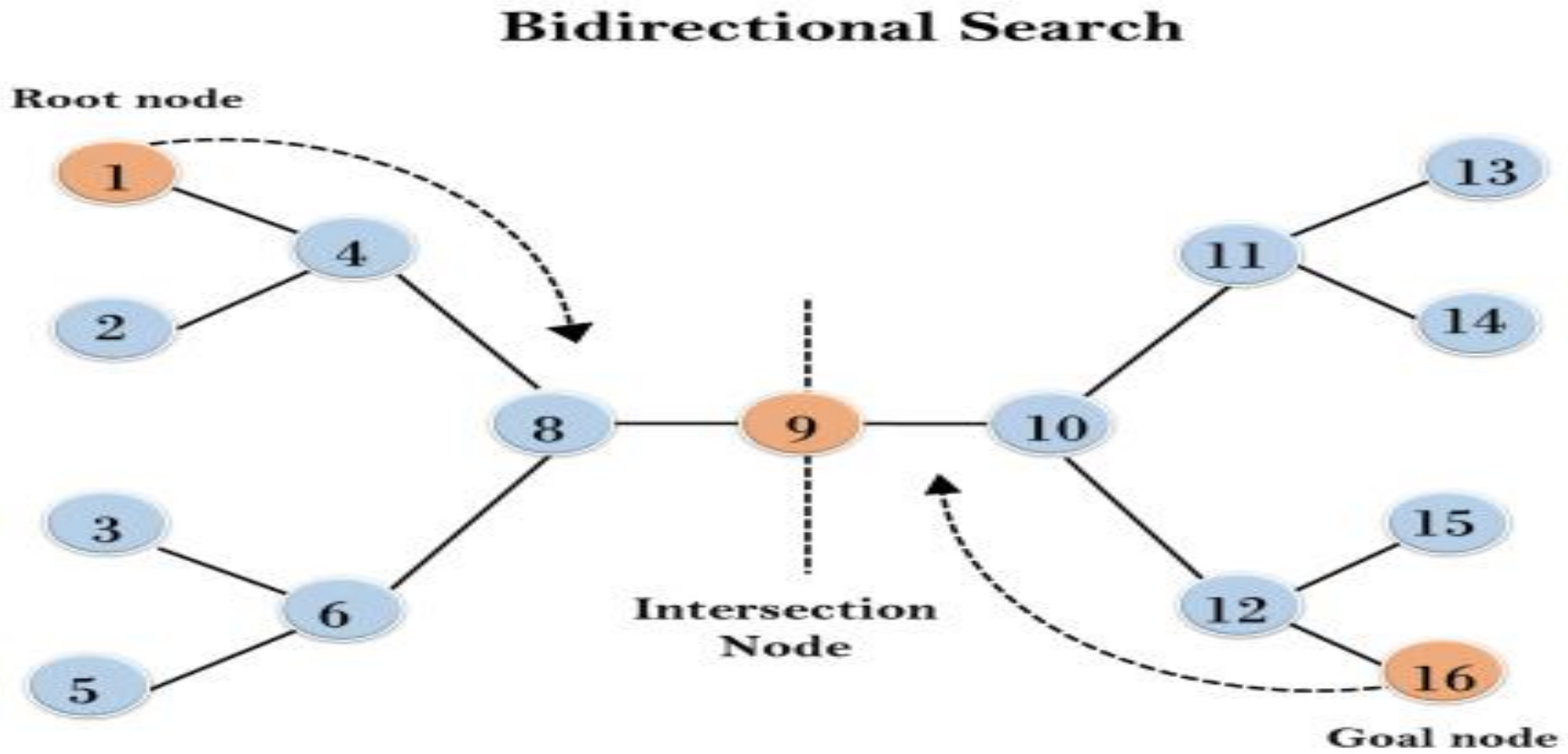
- Bidirectional search is fast.
- Bidirectional search requires less memory
- The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
The algorithm terminates at node 9 where two searches meet.



Completeness:

Bidirectional Search is complete if we use BFS in both searches.

Time Complexity:

Time complexity of bidirectional search using BFS is **O(bd).**

Space Complexity:

Space complexity of bidirectional search is O(bd).

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

- So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any **additional knowledge about search space.**
- But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1.) Best-first Search Algorithm (Greedy Search):

- Greedy best-first search algorithm 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.

$$\mathbf{f(n)= g(n).}$$

where, h(n)= estimated cost from node n to the goal.

- The greedy best first algorithm is implemented by the **priority queue.**

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.

Advantages:

•Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.

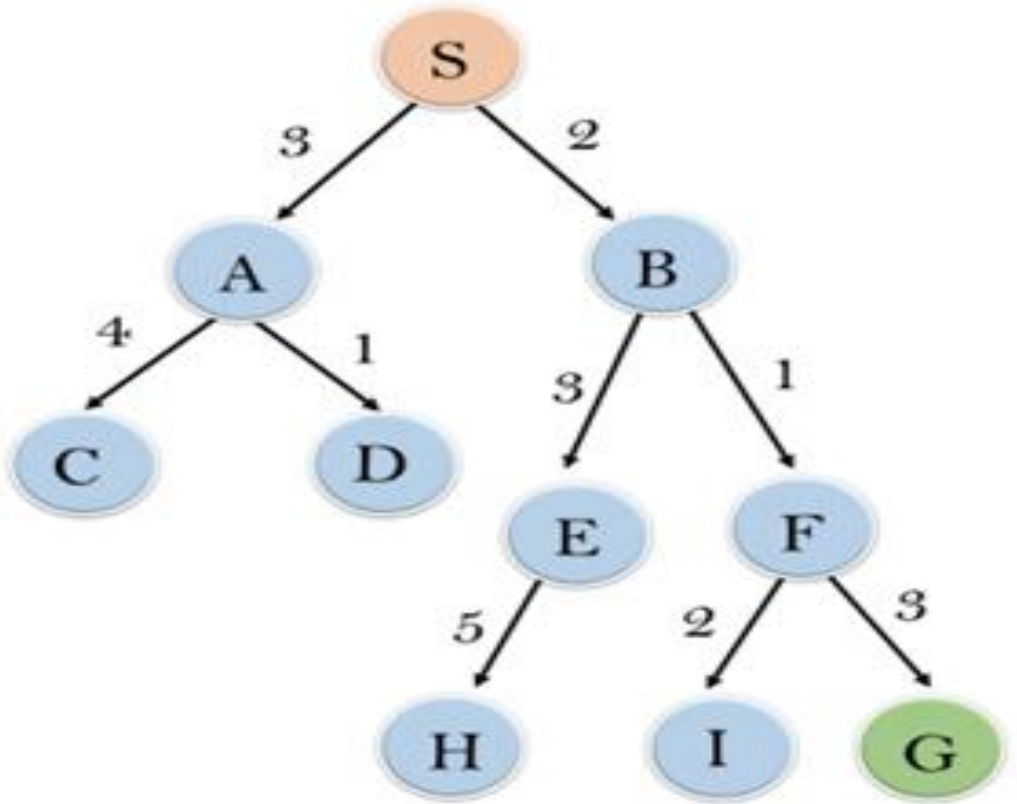
•This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

•It can behave as an unguided depth-first search in the worst case scenario.

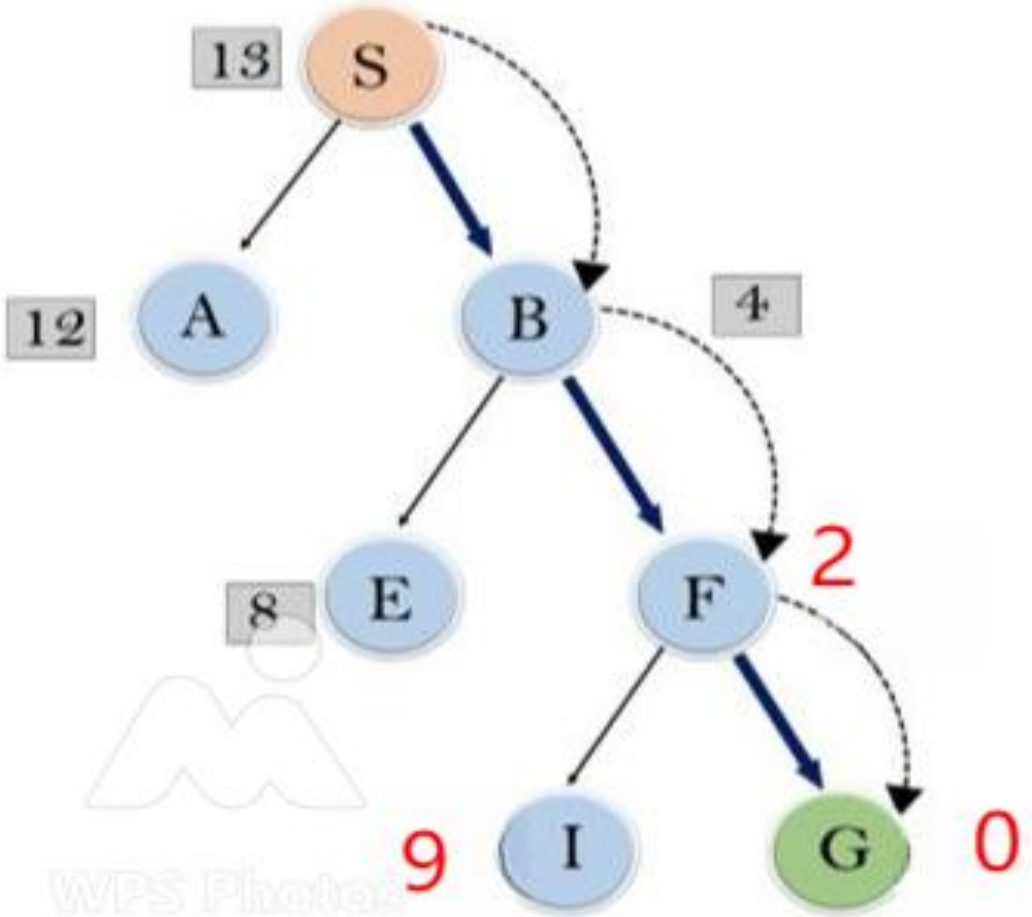
•It can get stuck in a loop as DFS.

•This algorithm is not optimal.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

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.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has features of greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.

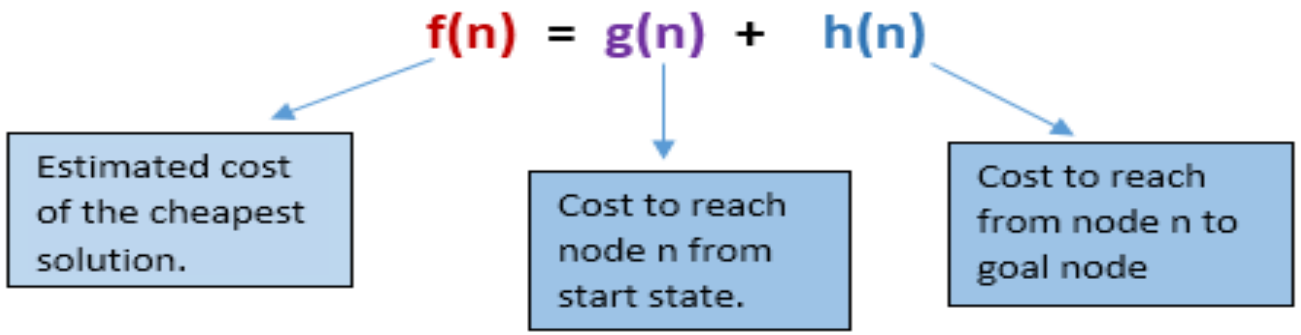
Algorithm of A* search:

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

- This search algorithm expands less search tree and provides optimal result faster. A* algorithm uses g(n)+h(n) instead of g(n).
- In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



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**.

Advantages:

A* search algorithm is the best algorithm than other search algorithms.

A* search algorithm is optimal and complete.

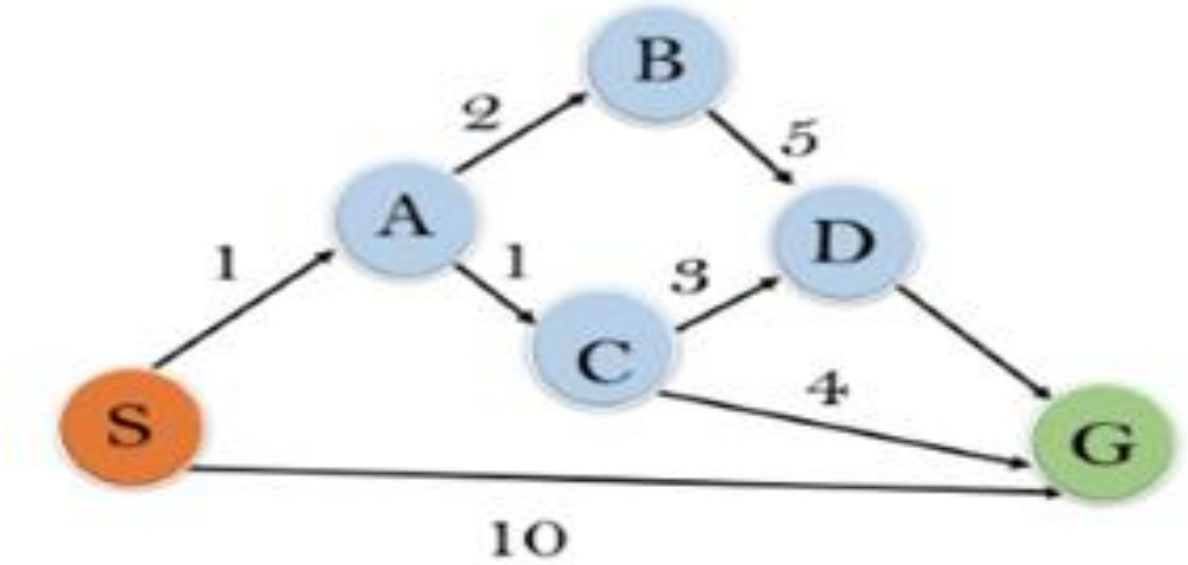
This algorithm can solve every complex problems.

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.



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

Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

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)

Complete: A* algorithm is complete as long as:

Branching factor is finite.

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.
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 the time complexity is O(b^d), where b is the branching factor.
Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Problem Reduction with AO* Algorithm

- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution.
- The decomposition of the problem or problem reduction generates AND arcs.
- Marelli and Montanani [1973],[1978] and Nilson[1980] proposed.
- Nilson calls it AO* Algorithm.

- One AND arc may point to any number of successor nodes.
- All these must be solved so that the arc will rise to many arcs, indicating several possible solutions.
- Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.
- OR node represents a choice between possible decomposition
- AND node represents a given decomposition

Memory-bounded search (Memory Bounded Heuristic Search) in AI

Search algorithms are fundamental techniques in the field of artificial intelligence (AI) that let agents or systems solve challenging issues. Memory-bounded search strategies are necessary because AI systems often encounter constrained memory resources in real-world circumstances. The notion of memory-bound search, often referred to as memory-bounded heuristic search, is examined in this article along with its importance in AI applications. We will review how AI effectively manages search jobs when memory resources are limited and provide a useful how-to manual for putting memory-bound search algorithms into practice.

Conventional search methods, such as the A* or Dijkstra's algorithms, sometimes require infinite memory, which may not be realistic in many circumstances.

Memory-bound search algorithms, on the other hand, are created with the limitation of finite memory in mind. The goal of these algorithms is to effectively use the memory that is available while finding optimum or nearly optimal solutions. They do this by deciding which information to keep and retrieve strategically, as well as by using heuristic functions to direct the search process.

Benefits of Memory-Bound Search

1. Memory-bound search algorithms perform well when memory is limited. They don't need a lot of memory to hold the whole search space or exploration history to locate solutions.

2. Memory-bound search algorithms are useful for a variety of AI applications, particularly those integrated into hardware with constrained memory. IoT devices, robots, autonomous cars, and real-time systems fall under this category.

3. Memory-bound search looks for the optimal answer given the memory restrictions. These algorithms may often effectively provide optimum or almost ideal answers by using well-informed heuristics.

4. The memory allocation and deallocation techniques used by these algorithms are dynamic. They make decisions about what data to keep and when to remove or replace it, so memory is used effectively during the search process.

Heuristic Function In AI

Heuristic functions are strategies or methods that guide the search process in AI algorithms by providing estimates of the most promising path to a solution. They are often used in scenarios where finding an exact solution is computationally infeasible. Instead, heuristics provide a practical approach by narrowing down the search space, leading to faster and more efficient problem-solving.

Heuristic functions transform complex problems into more manageable subproblems by providing estimates that guide the search process. This approach is particularly effective in AI planning, where the goal is to sequence actions that lead to a desired outcome.

Heuristic Search Algorithm in AI

Heuristic search algorithms leverage heuristic functions to make more intelligent decisions during the search process. Some common heuristic search algorithms include:

A* Algorithm

The A* algorithm is one of the most widely used heuristic search algorithms. It uses both the actual cost from the start node to the current node (g(n)) and the estimated cost from the current node to the goal (h(n)). The total estimated cost (f(n)) is the sum of these two values:

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

Greedy Best-First Search

The Greedy Best-First Search algorithm selects the path that appears to be the most promising based on the heuristic function alone. It prioritizes nodes with the lowest heuristic cost (h(n)), but it does not necessarily guarantee the shortest path to the goal.

Hill-Climbing Algorithm

The Hill-Climbing algorithm is a local search algorithm that continuously moves towards the neighbor with the lowest heuristic cost. It resembles climbing uphill towards the goal but can get stuck in local optima.

Role of Heuristic Functions in AI

Heuristic functions are essential in AI for several reasons:

- They reduce the search space, leading to faster solution times.
- They provide a sense of direction in large problem spaces, avoiding unnecessary exploration.
- They offer practical solutions in situations where exact methods are computationally prohibitive.

Common Problem Types for Heuristic Functions

Heuristic functions are particularly useful in various problem types, including:

1. Pathfinding problems, such as navigating a maze or finding the shortest route on a map, benefit greatly from heuristic functions that estimate the distance to the goal.

2. In constraint satisfaction problems, such as scheduling and puzzle-solving, heuristics help in selecting the most promising variables and values to explore.

3. Optimization problems, like the traveling salesman problem, use heuristics to find near-optimal solutions within a reasonable time frame.

Local search Algorithms

Hill Climbing Algorithm in Artificial Intelligence

- 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**. One of the widely discussed examples of Hill climbing algorithm is "**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.**
- Hill Climbing is mostly used **when a good heuristic is available.**
- In this algorithm, we don't need to maintain and handle the "*search tree or graph as it only keeps a single current state*"

Features of Hill Climbing:

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.

- **Deterministic Nature:**

Hill Climbing is a deterministic optimization algorithm, which means that given the same initial conditions and the same problem, it will always produce the same result. There is no randomness or uncertainty in its operation.

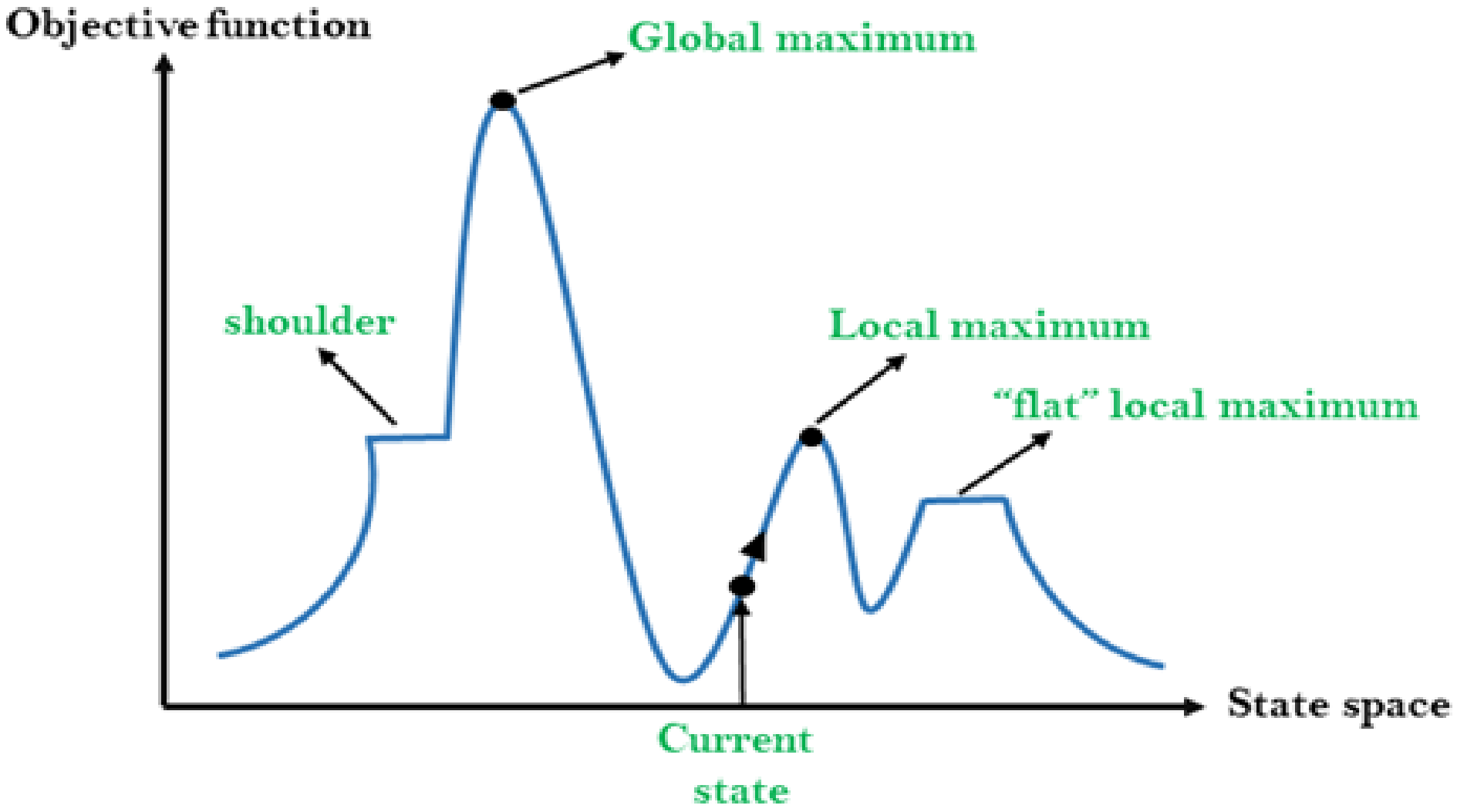
- **Local Neighborhood:**

Hill Climbing is a technique that operates within a small area around the current solution. It explores solutions that are closely related to the current state by making small, gradual changes. This approach allows it to find a solution that is better than the current one although it may not be the global optimum.

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.


Objective function
Global maximum
shoulder
Local maximum
"flat" local maximum
State space
Current
state

Different regions in the state space landscape:

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.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

•Simple hill climbing is the "simplest way to implement a 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 has the following features:

- Less time consuming
- 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.

2. Steepest-Ascent hill climbing:

- The steepest-Ascent algorithm is a variation of simple hill climbing algorithm.
- This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is 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.

Let SUCC be a state such that any successor of the current state will be better than it.

For each operator that applies to the current state:

Apply the new operator and generate a new state.

Evaluate the new state.

If it is goal state, then return it and quit, else compare it to the SUCC.

If it is better than SUCC, then set new state as SUCC.

If the SUCC is better than the current state, then set current state to SUCC.

Step 5: Exit.

3. Stochastic hill climbing:

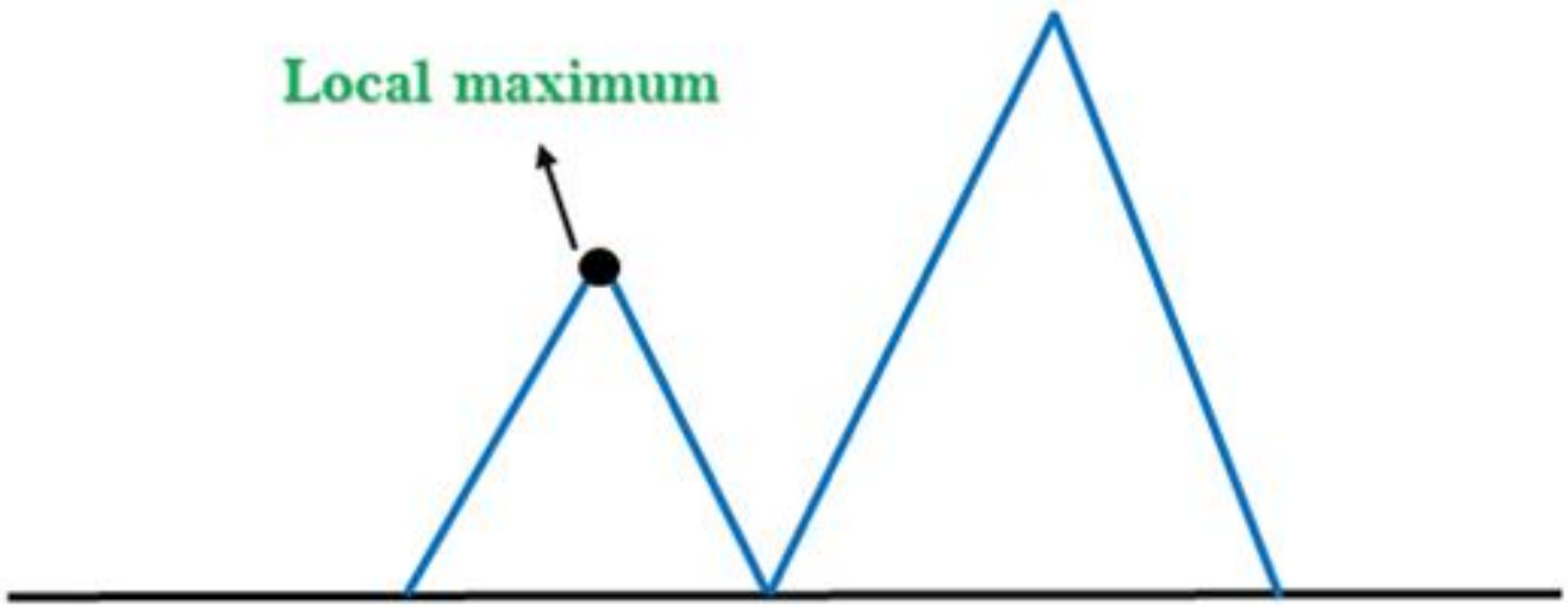
•Stochastic hill climbing does not examine for all its neighbor before moving.

•Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

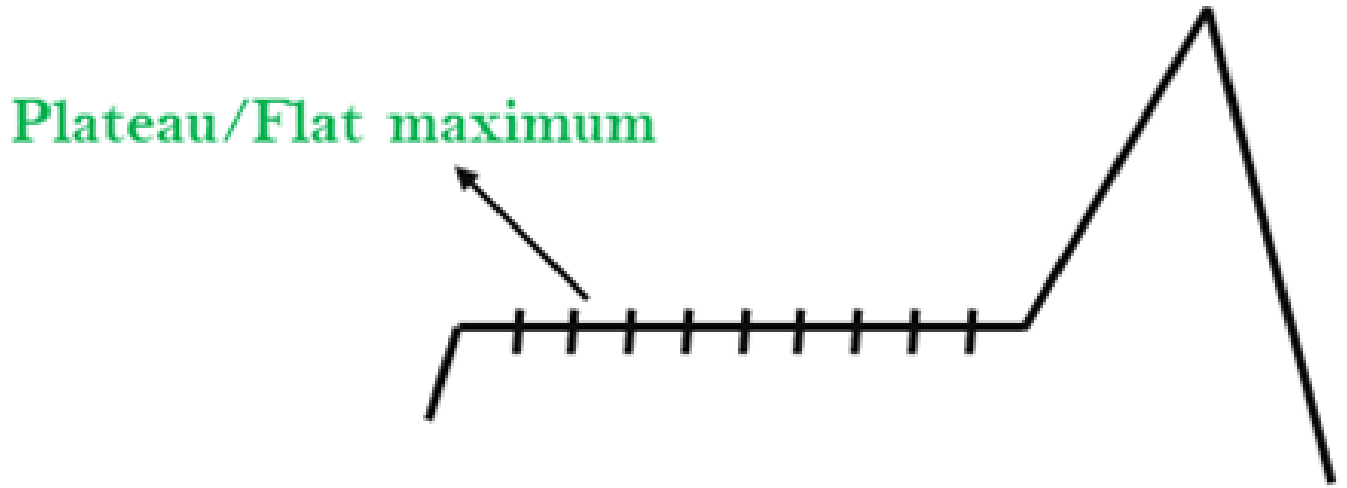
1. **Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. **Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.**


Local maximum

2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.


Plateau/Flat maximum

3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Ridge

Simulated Annealing:

•A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum.

•And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

•In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state.

•The same process is used in simulated annealing **in which the algorithm picks a random move, instead of picking the best move.** If the random move improves the state, then it follows the same path.

•Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

Applications of Hill Climbing Algorithm

The hill climbing technique has seen wide-spread usage in artificial intelligence and optimization respectively. It methodically solves those problems via coupled research activities by systematically testing options and picking out the most appropriate one. Some of the application are as follows:

Some of the application are as follows:

1. Machine Learning:

Fine tuning of machine learning models frequently is doing the hyper parameter optimization that provides the model with guidance on how it learns and behaves. Another exercise which serves the same purpose is hill training. Gradual adjustment of hyperparameters and their evaluation according to the respectively reached the essence of the hill climbing method.

2. Robotics:

In robotics, hill climbing technique turns out to be useful for an artificial agent roaming through a physical environment where its path is adjusted before arriving at the destination.

3. Network Design:

The tool may be employed for improvement of network forms, processes, and topologies in the telecommunications industry and computer networks. This approach erases the redundancy thus the efficiency of the networks are increased by studying and adjusting their configurations.

Local Beam Search Algorithm

- Beam search is a heuristic search algorithm that **explores a graph by expanding the most optimistic node in a limited set.** Beam search is an optimization of best-first search that reduces its memory requirements.

- Best-first search is a graph search that orders all partial solutions according to some heuristic. But in beam search, only a predetermined number of best partial solutions are kept as candidates. Therefore, it is a greedy algorithm.

- Beam search uses **breadth-first search to build its search tree**. At each level of the tree, it generates all **successors of the states at the current level, sorting them in increasing order of heuristic cost. However, it only stores a predetermined number (β),** of **best states at each level called the beamwidth**. Only those states are expanded next.

- The greater the beam width, the fewer states are pruned. No states are pruned with infinite beam width, and beam search is identical to breadth-first search.
- The beamwidth bounds the memory required to perform the search. Since a goal state could potentially be pruned, beam search sacrifices completeness (the guarantee that an algorithm will terminate with a solution if one exists).
- Beam search is not optimal, which means there is no guarantee that it will find the best solution.
- The beam width can either be **fixed or variable**. One approach that uses a variable beam width starts with the width at a minimum. If no solution is found, the beam is widened, and the procedure is repeated.

Components of Beam Search

A beam search takes **three components** as its input:

- **The problem** is the problem to be solved, usually represented as a graph, and contains a set of nodes in which one or more of the nodes represents a goal.
- **The set of heuristic rules** are rules specific to the problem domain and prune unfavorable nodes from memory regarding the problem domain.
- **The memory** is where the "beam" is stored, memory is full, and a node is to be added to the beam, the most costly node will be deleted, such that the memory limit is not exceeded.

Beam Search Algorithm :

```
beamSearch(problemSet, ruleSet, memorySize)
    openMemory = new memory of size memorySize
    nodeList = problemSet.listOfNodes
    node = root or initial search node
    Add node to openMemory;
    while (node is not a goal node)
        Delete node from openMemory;
        Expand node and obtain its children, evaluate those children;
        If a child node is pruned according to a rule in ruleSet, delete it;
        Place remaining, non-pruned children into openMemory;
        If memory is full and has no room for new nodes, remove the worst
            node, determined by ruleSet, in openMemory;
        node = the least costly node in openMemory;
```

Uses of Beam Search

A beam search is most often used to maintain tractability in large systems with insufficient memory to store the entire search tree. For example,

- It has been used in many machine translation systems.
- Each part is processed to select the best translation, and many different ways of translating the words appear.
- According to their sentence structures, the top best translations are kept, and the rest are discarded. The translator then evaluates the translations according to a given criterion, choosing the translation which best keeps the goals.
- The first use of a beam search was in the Harpy Speech Recognition System, CMU 1976.

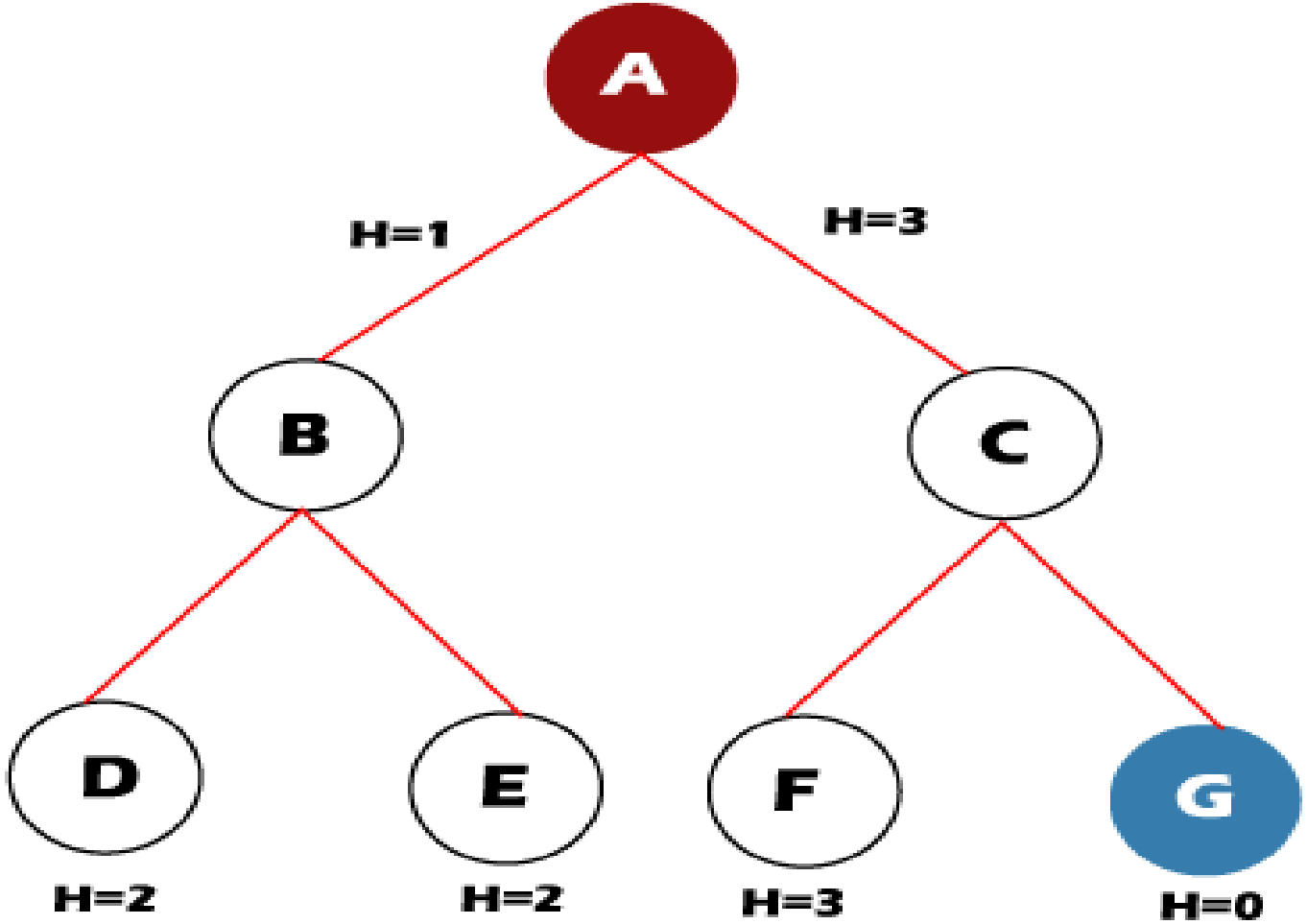
Drawbacks of Beam Search

Here is a drawback of the Beam Search with an example:

- In general, the Beam Search Algorithm is not complete. Despite these disadvantages, beam search has found success in the practical areas of speech recognition, vision, planning, and machine learning.
- The main disadvantages of a beam search are that the search may not result in an optimal goal and may not even reach a goal at all after given unlimited time and memory when there is a path from the start node to the goal node.
- The beam search algorithm terminates for two cases: a required goal node is reached, or a goal node is not reached, and there are no nodes left to be explored.
- A more accurate heuristic function and a larger beam width can improve Beam Search's chances of finding the goal.

For example, let's take the value of ß = 2 for the tree shown below. So, follow the following steps to find the goal node.

Step 1: OPEN= {A}
Step 2: OPEN= {B, C}
Step 3: OPEN= {D, E}
Step 4: OPEN= {E}
Step 5: OPEN= { }
The open set becomes empty without finding the goal node.



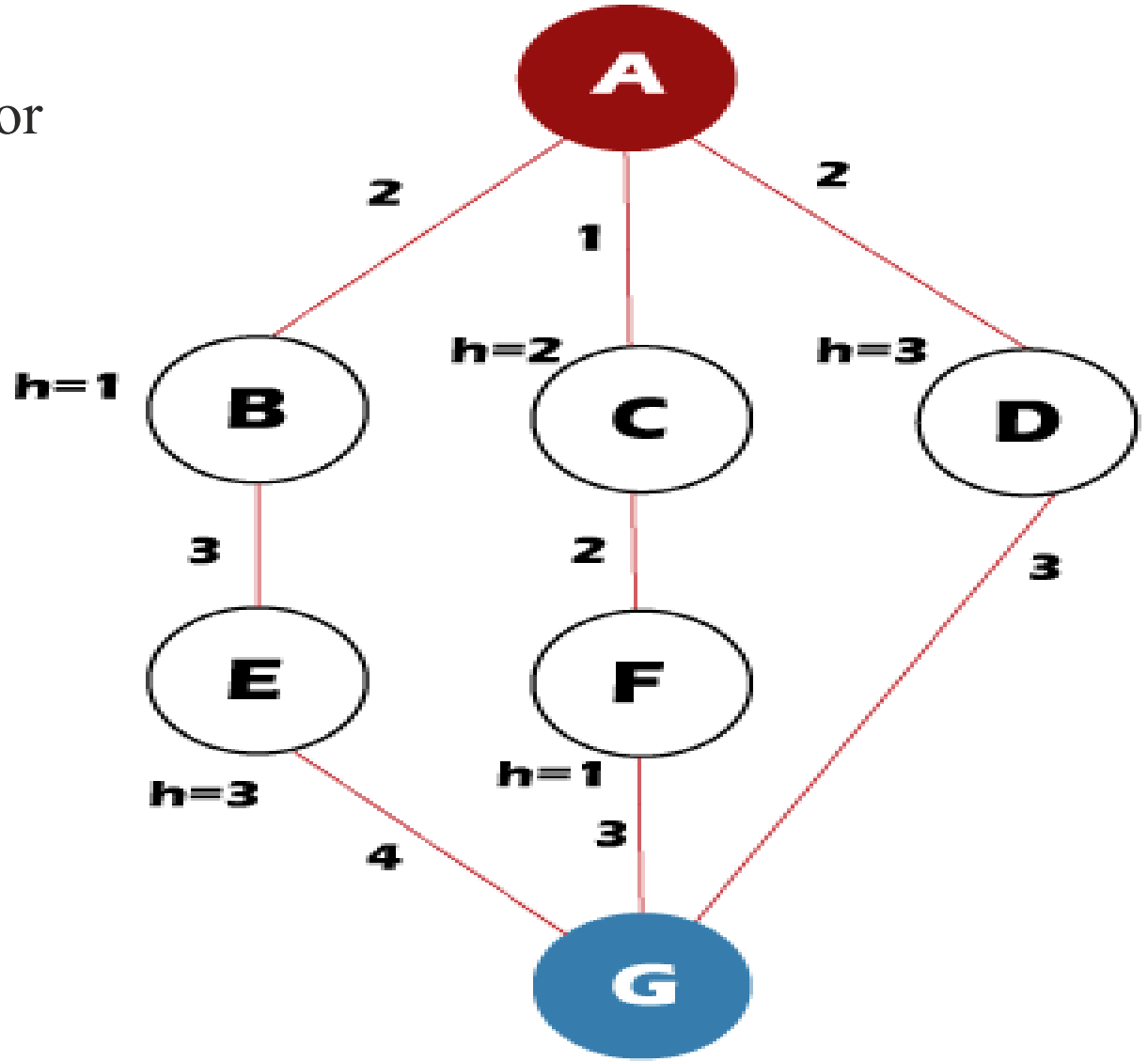
Beam Search Optimality

- The Beam Search algorithm is not complete in some cases. Therefore it is also not guaranteed to be optimal. It can happen because of these reasons:
- The beam width and an inaccurate heuristic function may cause the algorithm to miss expanding the shortest path.
- A more precise heuristic function and a larger beam width can make Beam Search more likely to find the optimal path to the goal.
- For example, we have a tree with heuristic values shown below:

Follow the following steps to find the path for the goal node.

Step 1: OPEN= {A}
Step 2: OPEN= {B, C}
Step 3: OPEN= {C, E}
Step 4: OPEN= {F, E}
Step 5: OPEN= {G, E}
Step 6: Found the goal node {G}, now stop.
Path: A-> C-> F-> G

But the Optimal Path is **A-> D-> G**



Time Complexity of Beam Search

The time complexity of the Beam Search algorithm depends on the following things, such as:

In the worst case, the heuristic function leads Beam Search to the deepest level in the search tree.

- **The worst-case time = O(B*m)**
- B is the beam width, and m is the maximum depth of any path in the search tree.

Space Complexity of Beam Search

The space complexity of the Beam Search algorithm depends on the following things, such as:

The worst-case space complexity = O(B*m)

B is the beam width, and m is the maximum depth of any path in the search tree.

Genetic algorithms

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics.

They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species that can adapt to changes in their environment can survive and reproduce and go to the next generation.

In simple words, they simulate **"survival of the fittest"** among individuals of consecutive generations to solve a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution.

Foundation of Genetic Algorithms

Genetic algorithms are based on an analogy with the genetic structure and behavior of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1.Individuals in the population compete for resources and mate

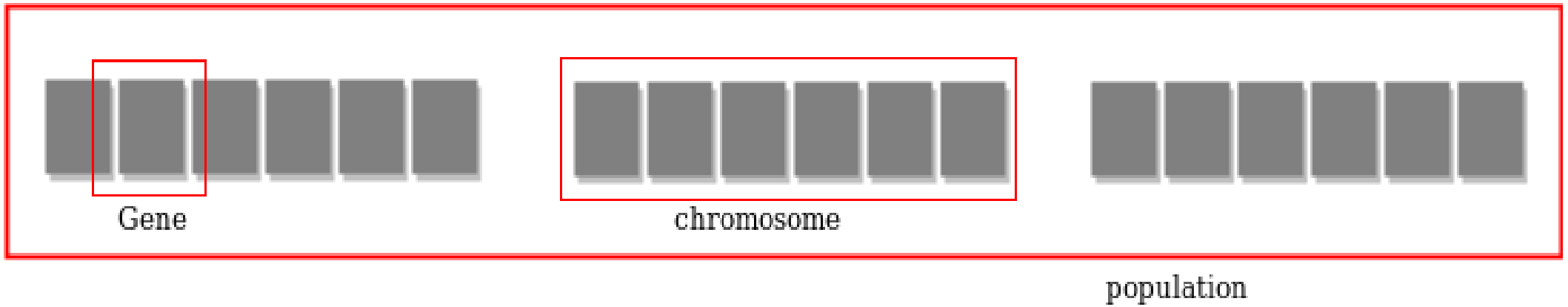
2.Those individuals who are successful (fittest) then mate to create more offspring than others

3.Genes from the "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.

4.Thus each successive generation is more suited for their environment.

Search space

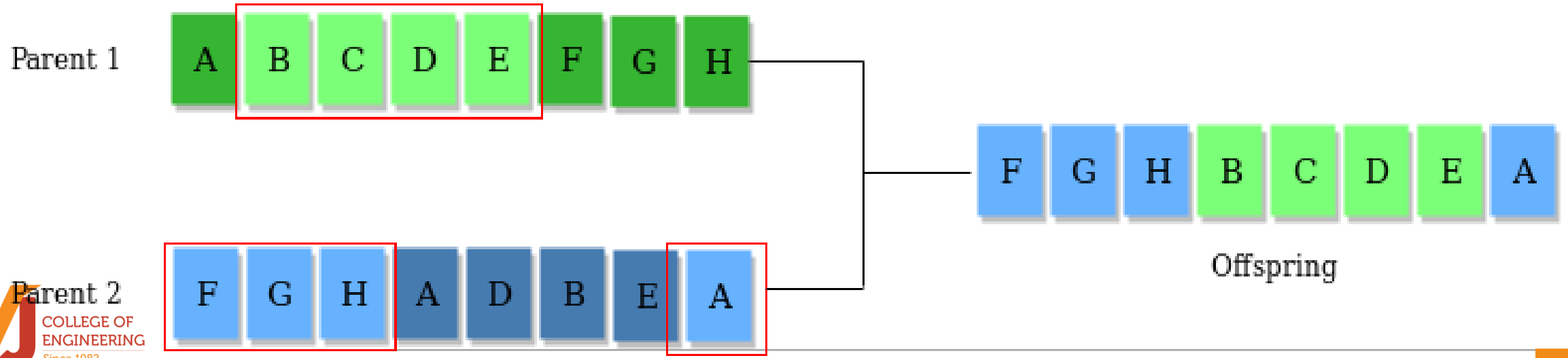
- The population of individuals are maintained within search space.
- Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components.
- These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



Operators of Genetic Algorithms :

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

- **Population:** A population represents a group of potential solutions to a problem. These solutions are often referred to as "individuals" or "chromosomes."
- **Selection Operator:** The selection process determines which individuals from the population will be chosen as parents for reproduction based on their fitness (how well they solve the problem).
- **Crossover Operator:** Crossover involves combining genetic information from two parent individuals to create one or more offspring.



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

- **Fitness Function:** The fitness function quantifies how well an individual solves the problem. It assigns a fitness score to each individual based on their performance, and this score guides the selection process.

Illustrating Exploration of Solution Spaces:

To understand how GAs explore solution spaces through evolution, consider an example: **Traveling Salesman Problem (TSP).** In TSP, the goal is to find the ***shortest route*** that visits a set of cities and returns to the starting city.

Initially, a population of possible routes (solutions) is generated randomly. Each route represents a different way to visit the cities.

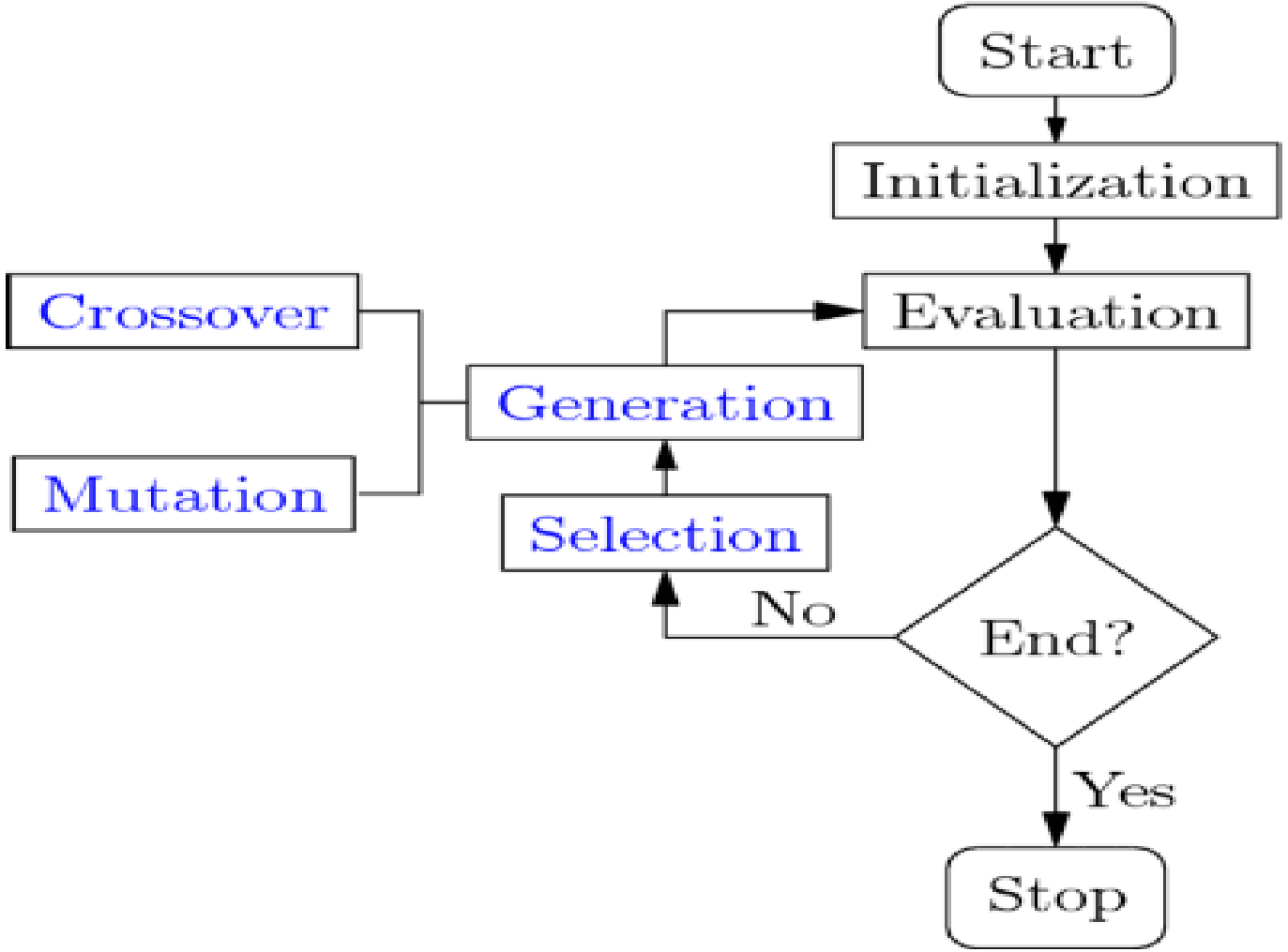
During the evolution process:

- Selection favors **routes with shorter distances**, as determined by the fitness function.
- **Crossover combines genetic information from two parent routes to create new routes.**
- **Mutation introduces slight variations in routes, potentially leading to improved solutions.**

Over several generations, the population evolves. Routes with shorter distances become more prevalent, and the algorithm converges toward an optimal or near-optimal solution.

This evolutionary process of selection, crossover, and mutation continues until a termination condition is met, such as a maximum number of generations or achieving a satisfactory solution.

Genetic Algorithm in AI Workflow



Genetic Algorithm Workflow

Step-by-step workflow of a Genetic Algorithm (GA). This process mirrors the principles of biological evolution and problem-solving in AI.

1. Initialization:

The process begins by creating an initial population of potential solutions to the problem. These solutions, often represented as strings or vectors, are referred to as individuals or chromosomes.

2. Selection:

Selection is a critical step in GAs, where individuals are chosen for reproduction based on their fitness. The more fit an individual is, the more likely it is to be selected as a parent.

3. Crossover:

Crossover, also known as recombination, involves taking genetic information from two parent individuals and combining it to create one or more offspring.

4. Mutation: Mutation introduces small random changes into the genetic information of offspring. This element of randomness helps maintain diversity within the population and allows for exploration of new solutions

5. Evaluation: The fitness function evaluates the performance of each individual in the population. It quantifies how well each individual solves the problem.

6. Termination:

Termination criteria determine when the GA should stop running. Common termination conditions include

- Reaching a maximum number of generations.
- Achieving a satisfactory solution or fitness level.
- Running out of computation time or resources.

Why use Genetic Algorithms :

- They are Robust
- Provide optimisation over large space state.
- Unlike traditional AI, they do not break on slight change in input or presence of noise

Application of Genetic Algorithms :

Genetic algorithms have many applications, some of them are –

- Recurrent Neural Network
- Mutation testing
- Code breaking
- Filtering and signal processing
- Learning fuzzy rule base etc

END OF MODULE 3