UNIT 2

GPS

- The **General Problem Solver** (**GPS**) was an AI program proposed by *Herbert Simon*, *J.C. Shaw*, and *Allen Newell*.
- It was the first useful computer program that came into existence in the AI world. The goal was to make it work as a universal problem-solving machine. Of course there were many software programs that existed before, but these programs performed specific tasks.
- GPS was the first program that was intended to solve any general problem. GPS was supposed to solve all the problems using the same base algorithm for every problem.

Examples of Problems in Artificial Intelligence

- Travelling Salesman Problem
- Tower of Hanoi Problem
- Chess
- Sudoku
- Logical Puzzles and so on.

Problem-Solving Agents

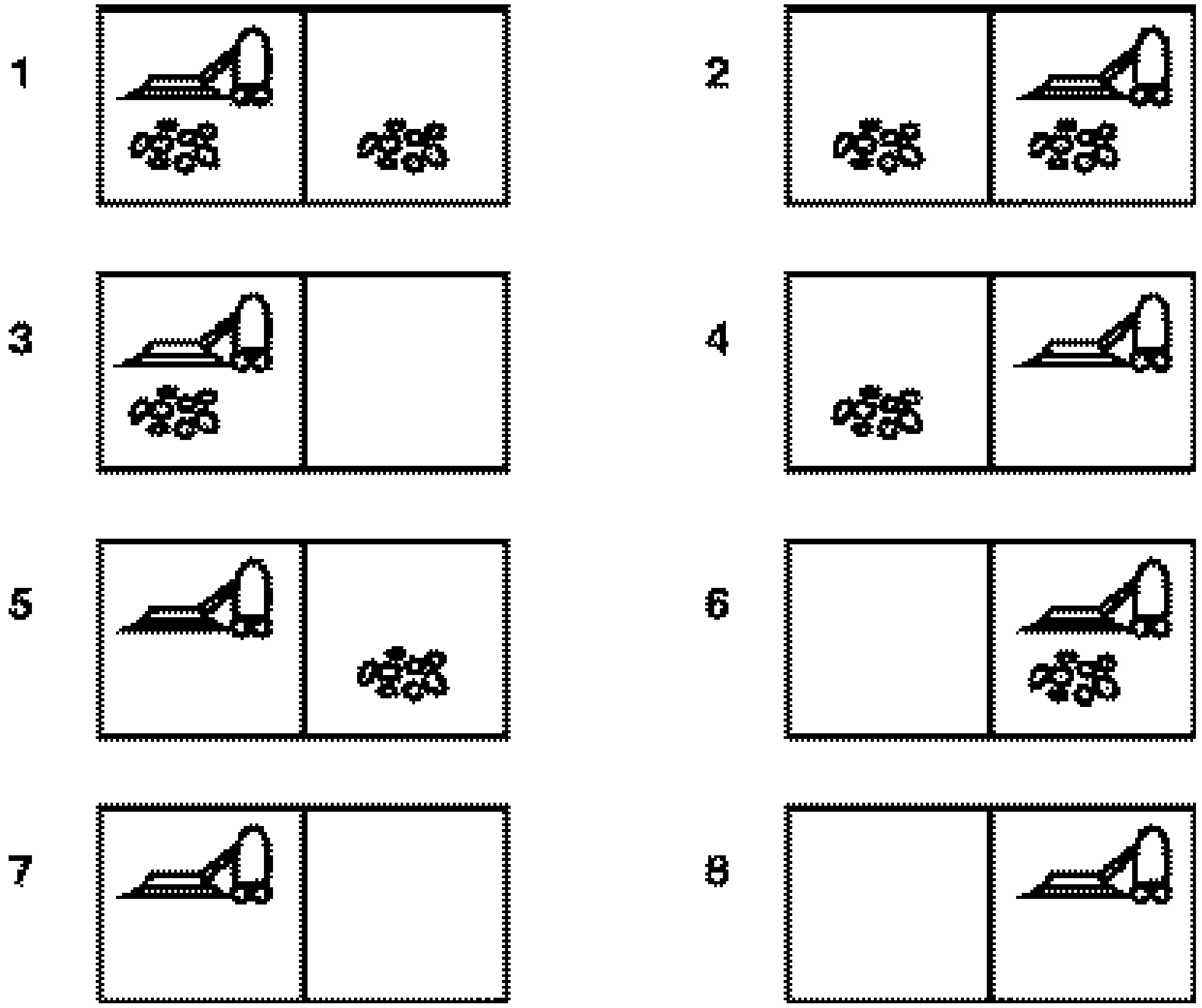
- Intelligent agents are supposed to act in such a way that the environment goes through a sequence of states that maximizes the performance measure.
  Unfortunately, this specification is difficult to translate into a successful agent design. The task is simplified if the agent can adopt a goal and aim to satisfy it.
- Intelligence requires knowledge and knowledge holds some less desirable properties like:
- It is enormous.
- It is tough to characterized precisely.
- It is dynamic.
- It is structured - a way that matches to the way it will be used.

Formulating problems

- **Problem formulation** is the process of deciding what actions and states to consider.
- In general, an agent with several intermediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value and then choosing the best one.
- This process is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase. Hence, we have a simple "formulate, search, execute" design for the agent.

Formulating problems

- Formulating problems is an art. First, we look at the different amounts of knowledge that an agent can have concerning its actions and the state that it is in. This depends on how the agent is connected to its environment.
- There are four essentially different types of problems.
- single state
- multiple state
- contingency
- exploration


1
2
3
4
5
6
7
8

- First, suppose that the agent's sensors give it enough information to tell exactly which state it is in (i.e., the world is completely accessible) and suppose it knows exactly what each of its actions does.
- Then it can calculate exactly what state it will be in after any sequence of actions. For example, if the initial state is 5, then it can calculate the result of the actions sequence {right, suck}.
- This simplest case is called a **single-state problem.**

- Now suppose the agent knows all of the effects of its actions, but world access is limited.
- For example, in the extreme case, the agent has no sensors so it knows only that its initial state is one of the set {1,2,3,4,5,6,7,8}.
- In this simple world, the agent can succeed even though it has no sensors. It knows that the action {right} will cause it to be in one of the states {2,4,6,8}. In fact, it can discover that the sequence {right, suck, left, suck} is guaranteed to reach a goal state no matter what the initial state is.
- In this case, when the world is not fully accessible, the agent must reason about sets of states that it might get to, rather than single states. This is called the **multiple-state problem.**

- The agent can solve the problem if it can perform sensing actions during execution.
- For example, starting from one of {1,3}: first suck dirt, then move right, then suck *only if there is dirt there*. In this case the agent must calculate a whole tree of actions rather than a single sequence, i.e., plans now have conditionals in them that are based on the results of sensing actions.
- For this reason, we call this a **contingency problem**. Many problems in the real world are contingency problems. This is why most people keep their eyes open when walking and driving around.
- Single-state and multiple-state problems can be handled by similar search techniques. Contingency problems require more complex algorithms.
- They also lend them selves to an agent design in which planning and execution are **interleaved.**

- The last type of problem is the **exploration problem**.
- In this type of problem, the agent has no information about the effects of its actions. The agent must experiment, gradually discovering what its actions do and what sorts of states exist.
- This is search in the real world rather than in a model.

Formulating problems

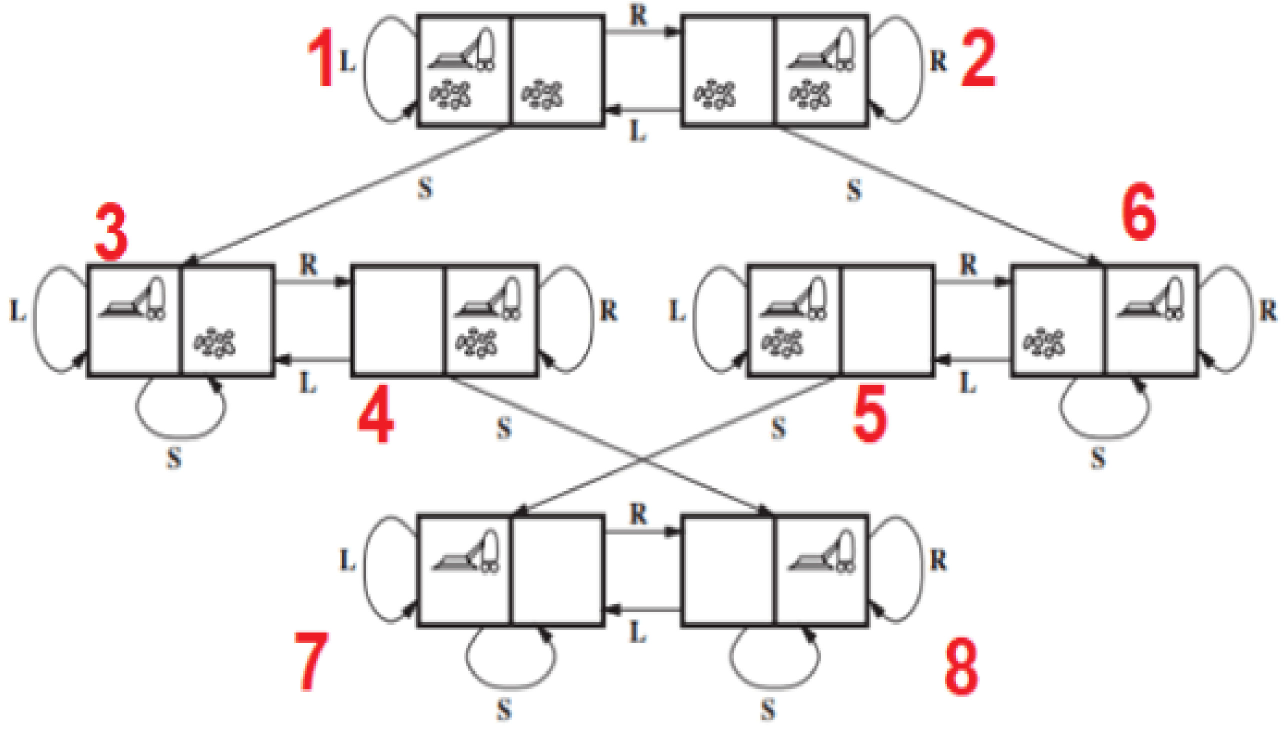
- **State space search** is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the intention of finding a *goal state* with the desired property.
- Problems are often modelled as a state space, a set of *states* that a problem can be in.
- The state space of a problem is the set of all states reachable from the initial state by executing any sequence of actions. States is representation of all possible outcomes.

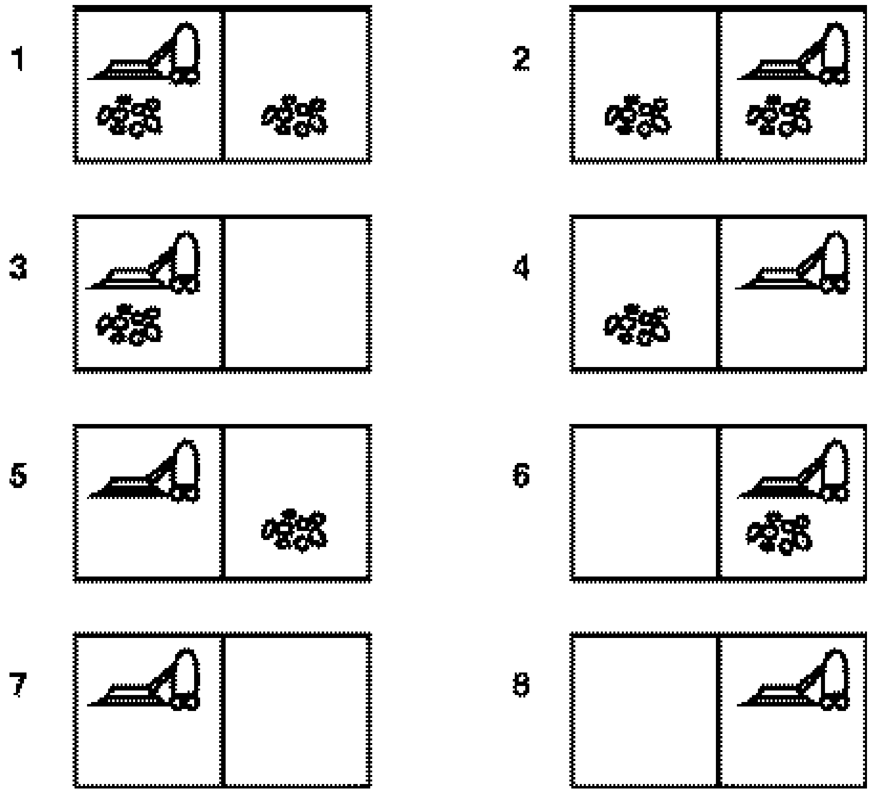
Components of problems formulation

- Initial state
- Actions
- Successor function
- Goal test
- Path cost

- Problem solution – well defined problem
- A solution to a problem is a sequence of actions chosen by the problem solving agent that leads from the initial state to a goal state.
- Optimal solution – least path cost

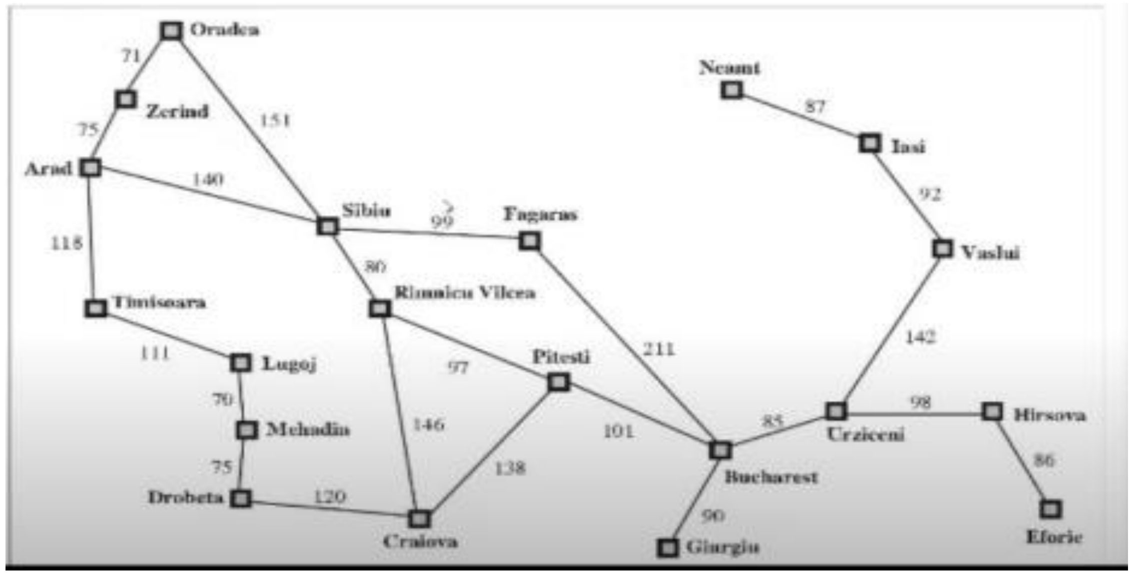
Vacuum cleaner problem





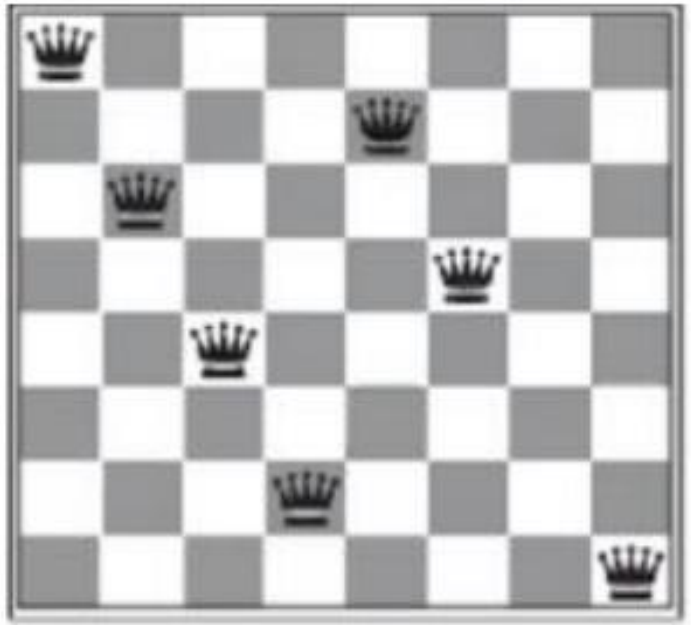
1) State : The state is determined by both the agent location and the dirt locations. There are 8 possible world states.
2) Initial state : Any state can be designated as the initial state.
3) Actions : Each state has just three actions : 1. Left 2. Right 3. Suck.
4) Transition model : Action left takes the agent to Leftmost square, Action Right takes the agent to Rightmost square and the Suck action cleans a dirty square and performs no action on clean square.
5) Goal test : This checks whether all the squares are clean.
6) Path cost : Each step costs 1, so the path cost is the number of steps in the path.

Map of Romania



1) The initial state : The initial state for our agent in this case (Romania) might be described as In(Arad).
2) Actions : It gives the possible actions from the current state, in this case the possible actions are (Go(Sibiu), Go(Timisoara), Go(Zerind)}.
3) Transition Model : This is Specified by using a function RESULT(s,a), that returns the state that results from doing action a in state s. RESULT(In(Arad),Go(Zerind)) = In(Zerind).
4) Path Cost : Path cost function assigns a numeric cost to each path, In the present case it is the distance in kilometers.
5) Goal Test : It determines whether the given state is Goal State

8 Queens Problem



1) States : Any arrangement of 0 to 8 queens on the boards is a state.
2) Initial state : Empty board i.e. No queens on the board.
3) Actions : Adding a queen to any empty square.
4) Transition model : Returns the board with a queen added to the specified square.
5) Goal test : 8 queen are on the board, in such a way that none is in attacking position.

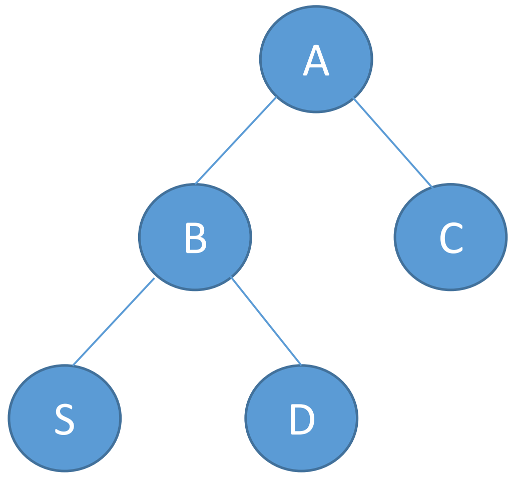
The Travelling Salesperson Problem (TSP)

1) States : All the cities which are to be visited by the salesman.
2) Initial State : Any city can be the initial state.
3) Actions : The agent can visit any city which is not yet visited from the current city.
4) Transition model : The next city becomes the current city
5) Goal test : Check if all the cities have been visited with minimum cost & also the final state is the initial state.
6) Path cost : This depends on the actions the agent has taken throughout the journey.

Measuring Problem Solving Performance

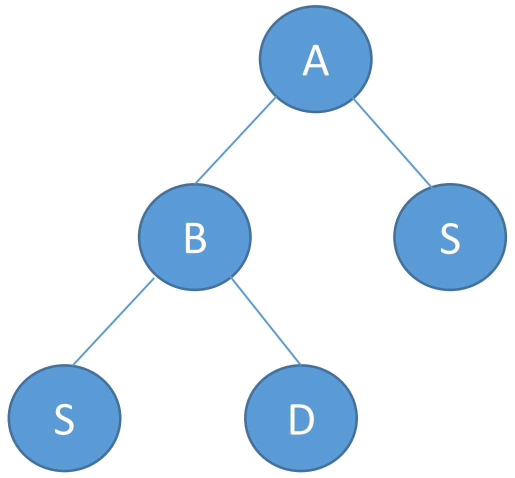
- Completeness:
- An algorithm is said to be complete if it definitely finds solution to the problem, if exists:

- Problem – move from A to S
- A-B-S(Solution found)
- A-C (Solution not found)



- Time complexity:
- How long does it take to find a solution? Usually measured in terms of the number of nodes expanded

- Problem – move from A to S
- A-B-S (Solution more time)
- A-S (Solution not found)

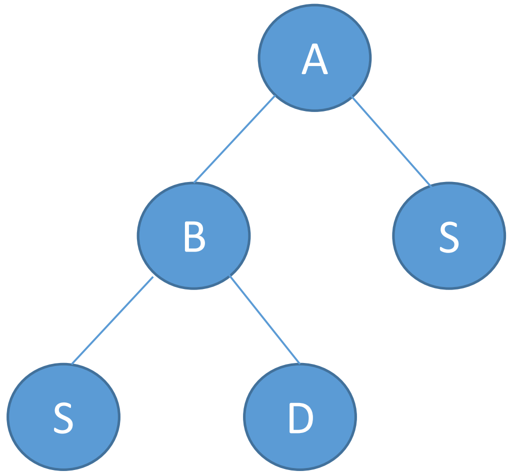


- Space complexity:
- How much space is used by algorithm? Measured in terms of maximum number of nodes in memory at a time

- if it has to save just previous step or two

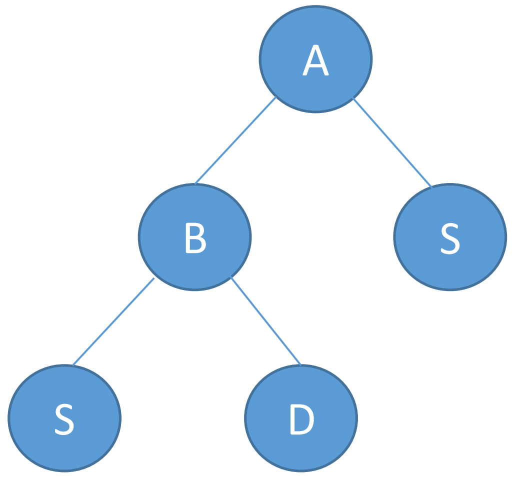
steps its space efficient compare to one has to

keep all nodes in memory



- Optimality:
- If a solution is found, is it guaranteed to be an optimal one? For example one with minimum cost

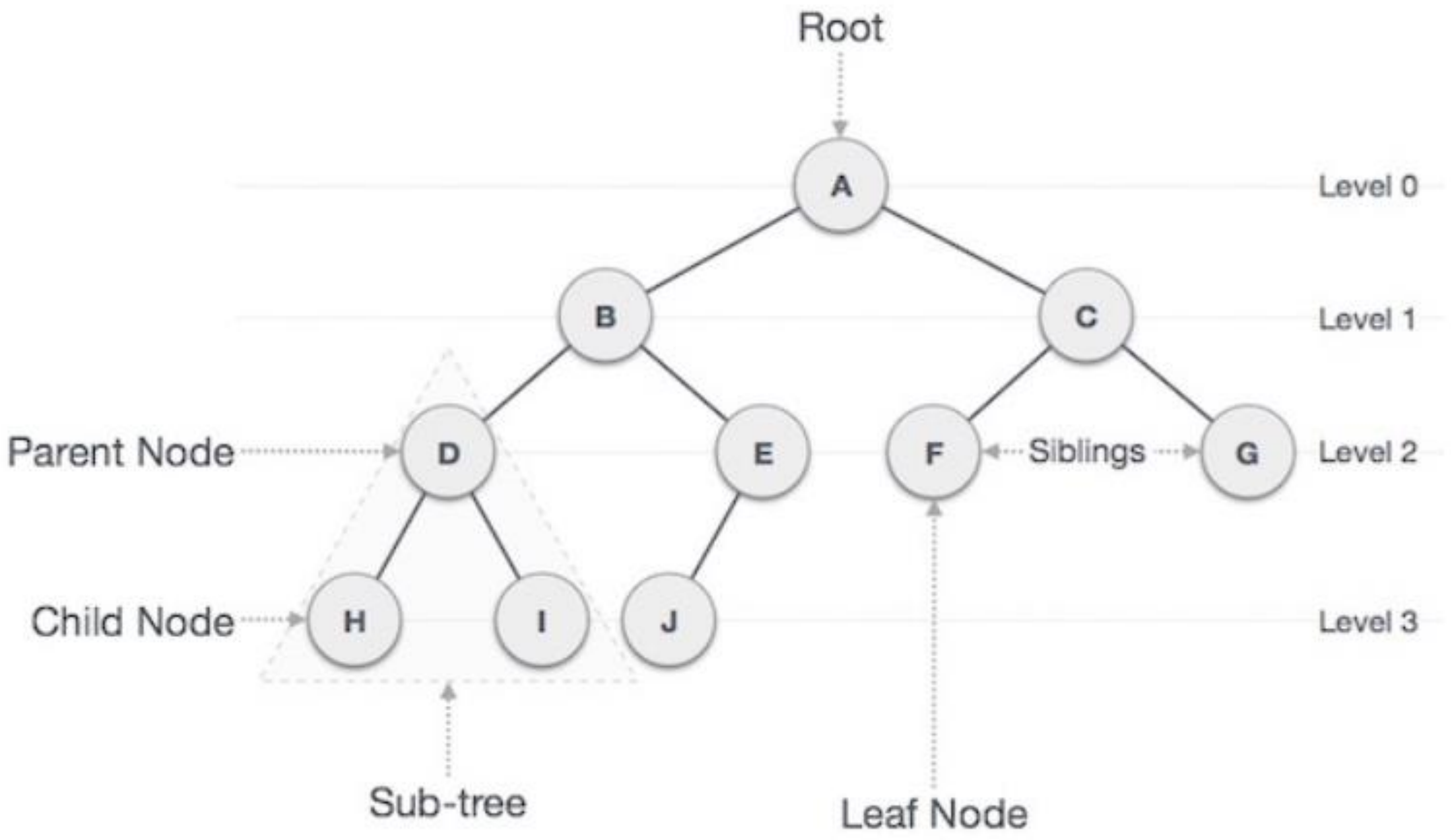
- A-B-S (Solution1)
- A-S (Solution2)
- Which one is optimal?



- Time and space complexity measurement combined:
- b: maximum branching factor of the search tree (max children of any node)
- d: depth of the least-cost solution
- m: maximum depth of state space (max length of any path in search space)

Node representation in search tree

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.


Root
A
Level 0
B
C
Level 1
Parent Node
D
E
F
Siblings
G
Level 2
Child Node
H
I
J
Level 3
Sub-tree
Leaf Node

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.

Uninformed Search Algorithms:

- 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

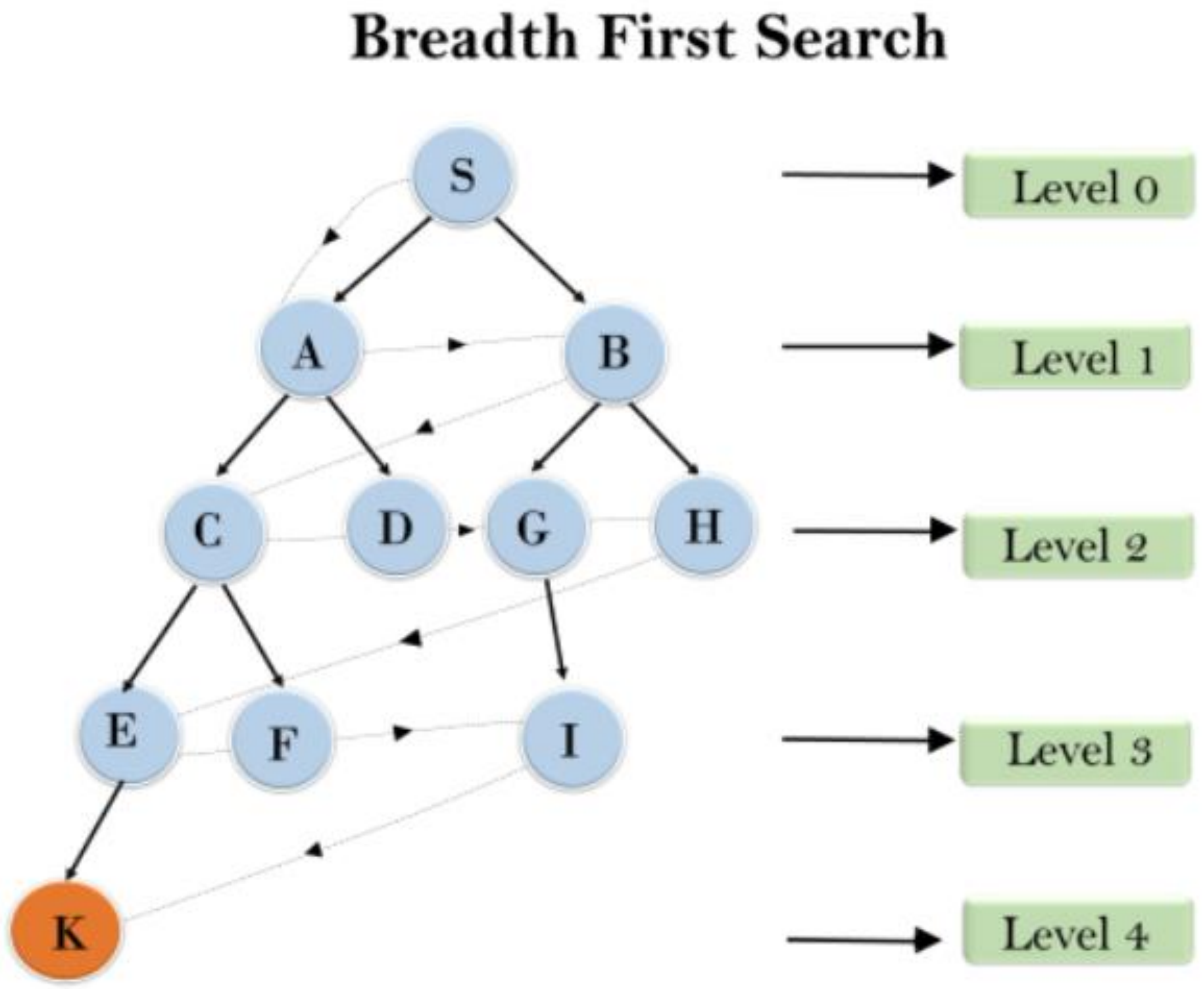
Uninformed Search Algorithms:

- Each of these algorithms will have:
- A problem **graph,** containing the start node S and the goal node G.
- A **strategy,** describing the manner in which the graph will be traversed to get to G.
- A **fringe,** which is a data structure used to store all the possible states (nodes) that you can go from the current states.
- A **tree,** that results while traversing to the goal node.
- A solution **plan,** which the sequence of nodes from S to G.

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.

- 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.
- **$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$**
- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is **$O(b^d)$**.
- **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.

- **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.
- **Disadvantages:**

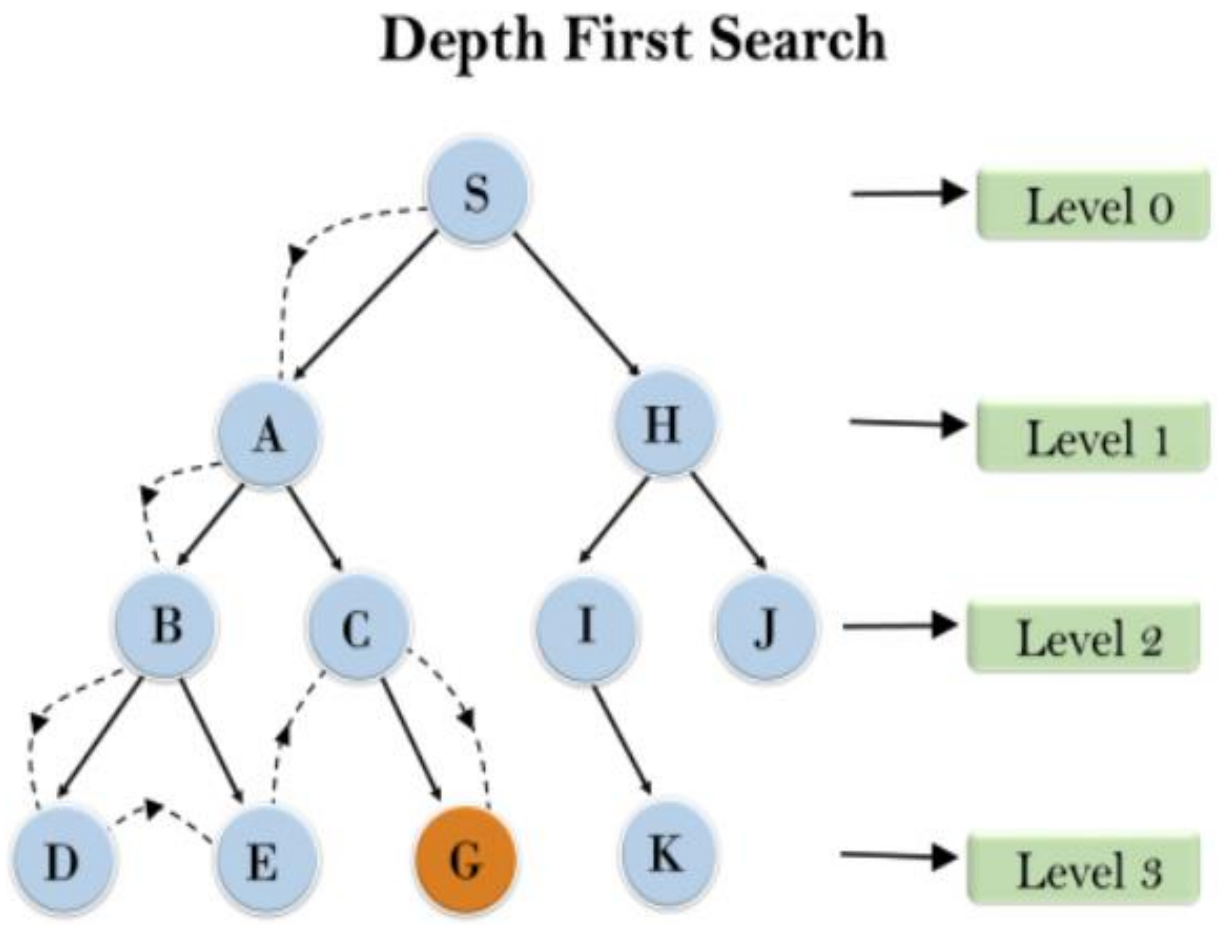
•It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

•BFS needs lots of time if the solution is far away from the root node.

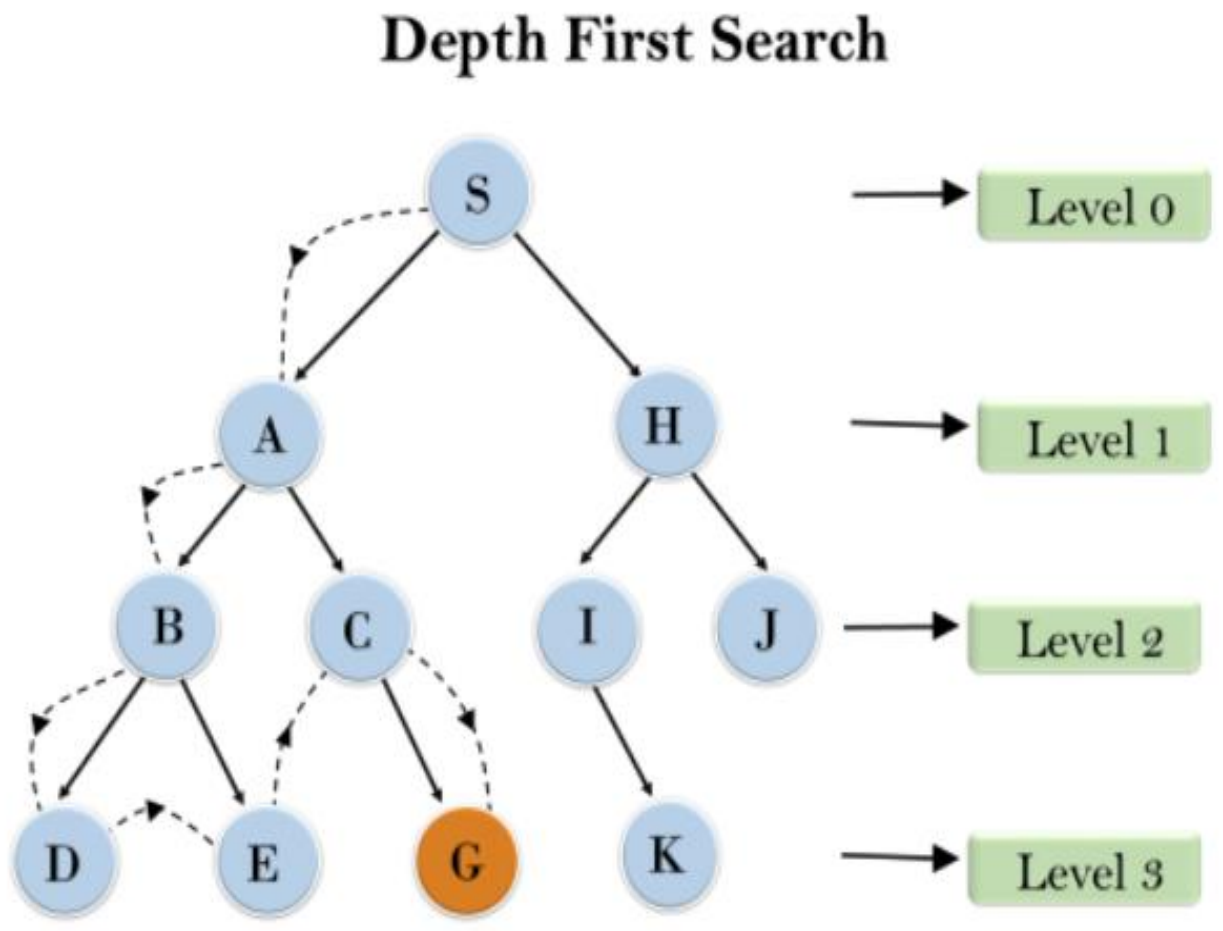
Depth-first Search

- Depth-first search is a 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.
- Note: **Backtracking** is an algorithm technique for finding all possible solutions using recursion.

- 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.



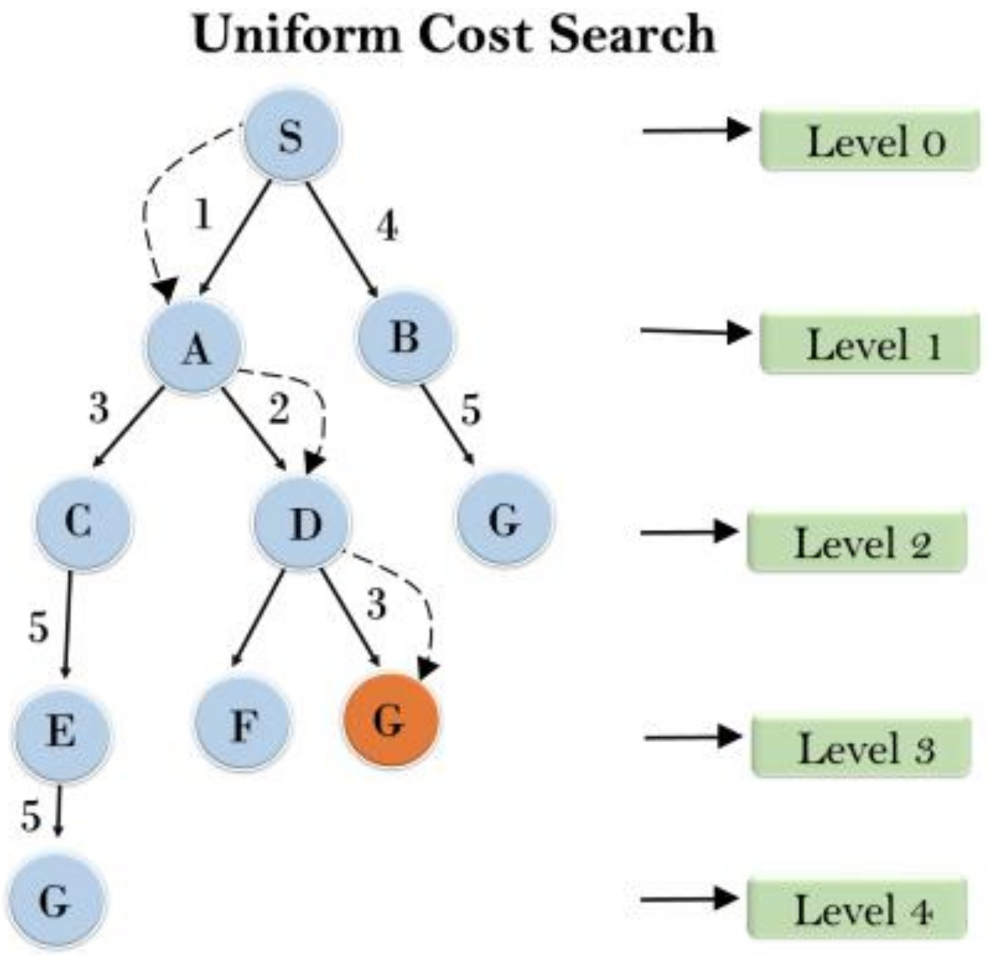
- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
- **$T(n)= 1+ b^2+ b^3 +.........+ b^m=O(b^m)$**
- **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(b^m)**.
- **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.

- **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).
- **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.

Uniform-cost Search Algorithm:

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph.
- This algorithm comes into play when a **different cost** is available for each edge.
- The primary goal of the uniform-cost search is to find a path to the goal node which has the **lowest cumulative cost**.
- Uniform-cost search expands nodes according to their path costs form the root node.
- It can be used to solve any graph/tree where the optimal cost is in demand.
- A uniform-cost search algorithm is implemented by the **priority queue**.
- It gives maximum priority to the lowest cumulative cost.
- Uniform cost search is **equivalent to BFS algorithm** if the path cost of all edges is the same.

Example:


Uniform Cost Search
S
A
B
C
D
G
E
F
G
G
1
4
3
2
5
5
3
5
Level 0
Level 1
Level 2
Level 3
Level 4

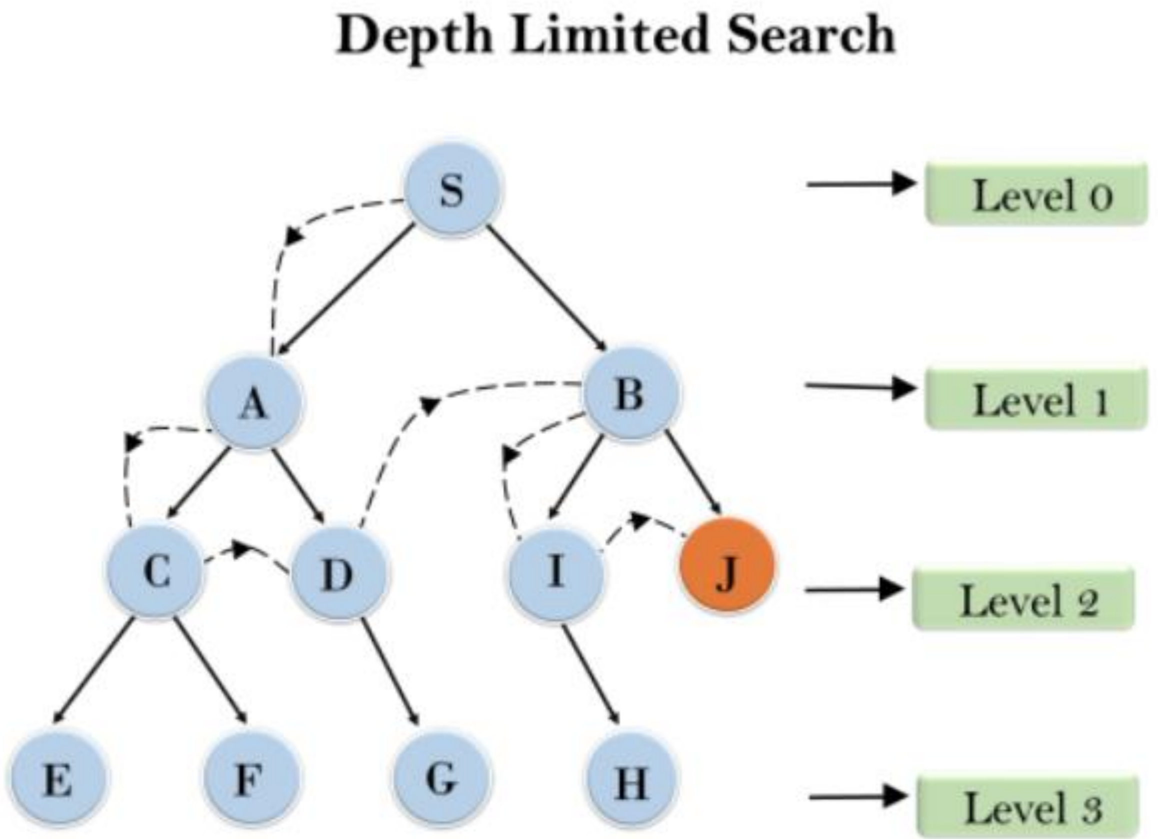
- **Completeness:**
- Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:**
- Let **C* is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is**$O(b^{1 + [C^*/\varepsilon]})$**.
- **Space Complexity:**
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.
- **Optimal:**
- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

- **Advantages:**
- Uniform cost search is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

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.

- Example:


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

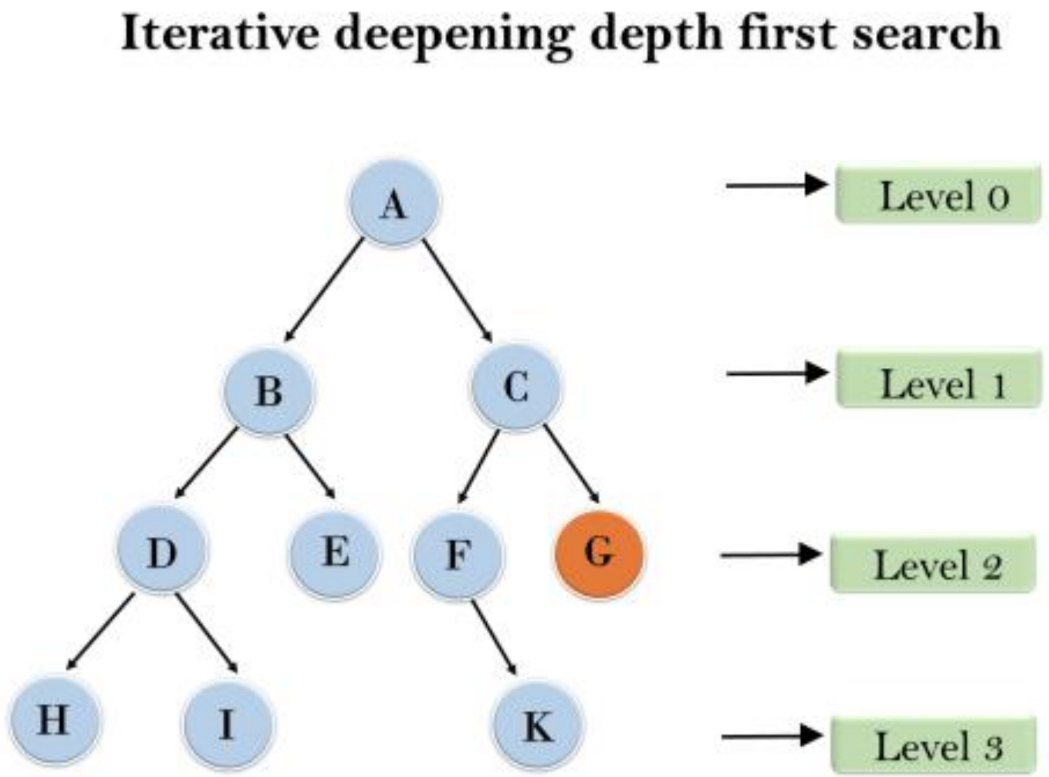
- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is $\mathbf{O(b^{\ell})}$.
- **Space Complexity:** Space complexity of DLS algorithm is O**($b \times \ell$)**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

- **Advantages:**
- Depth-limited search is Memory efficient.
- **Disadvantages:**
- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

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.**

- 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:
- 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 if the 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 **$O(b^d)$**.
- **Space Complexity:**
- The space complexity of IDDFS will be **$O(bd)$**.
- **Optimal:**
- IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

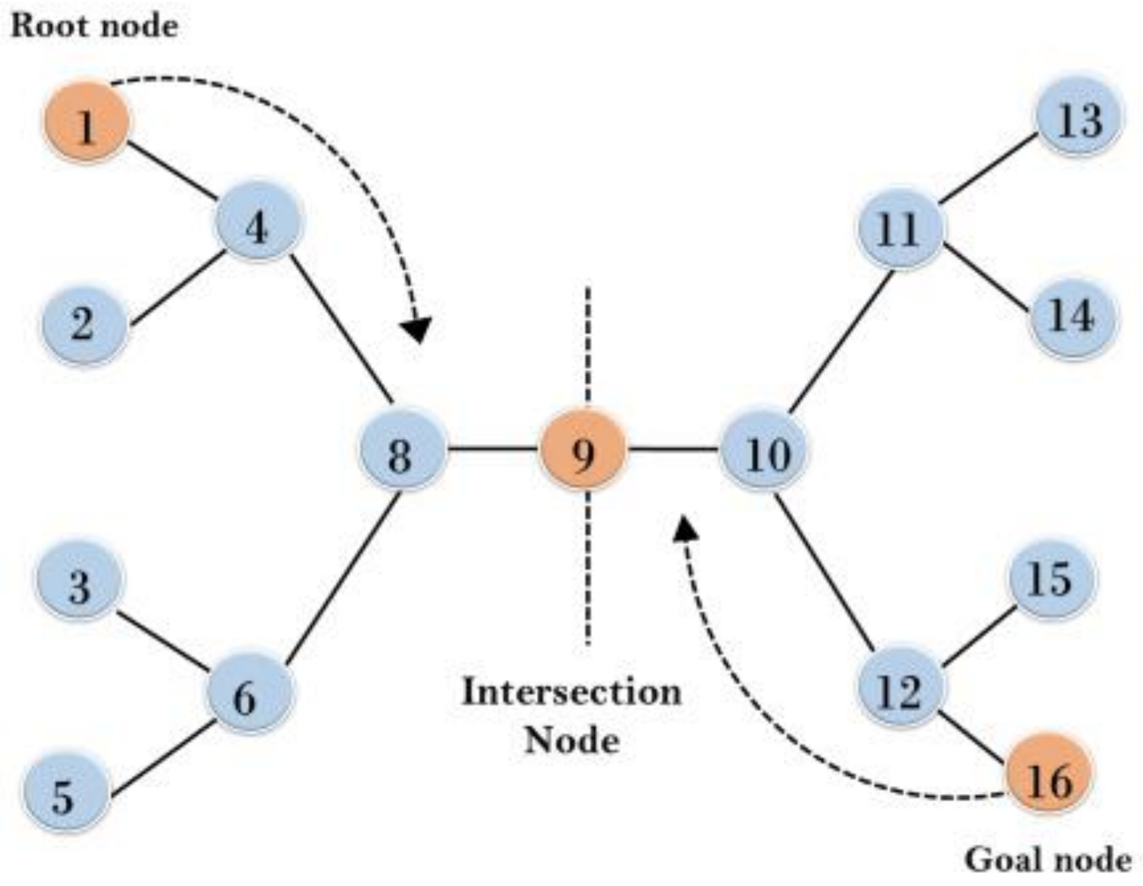
- **Advantages:**
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- **Disadvantages:**
- The main drawback of IDDFS is that it repeats all the work of the previous phase.

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.

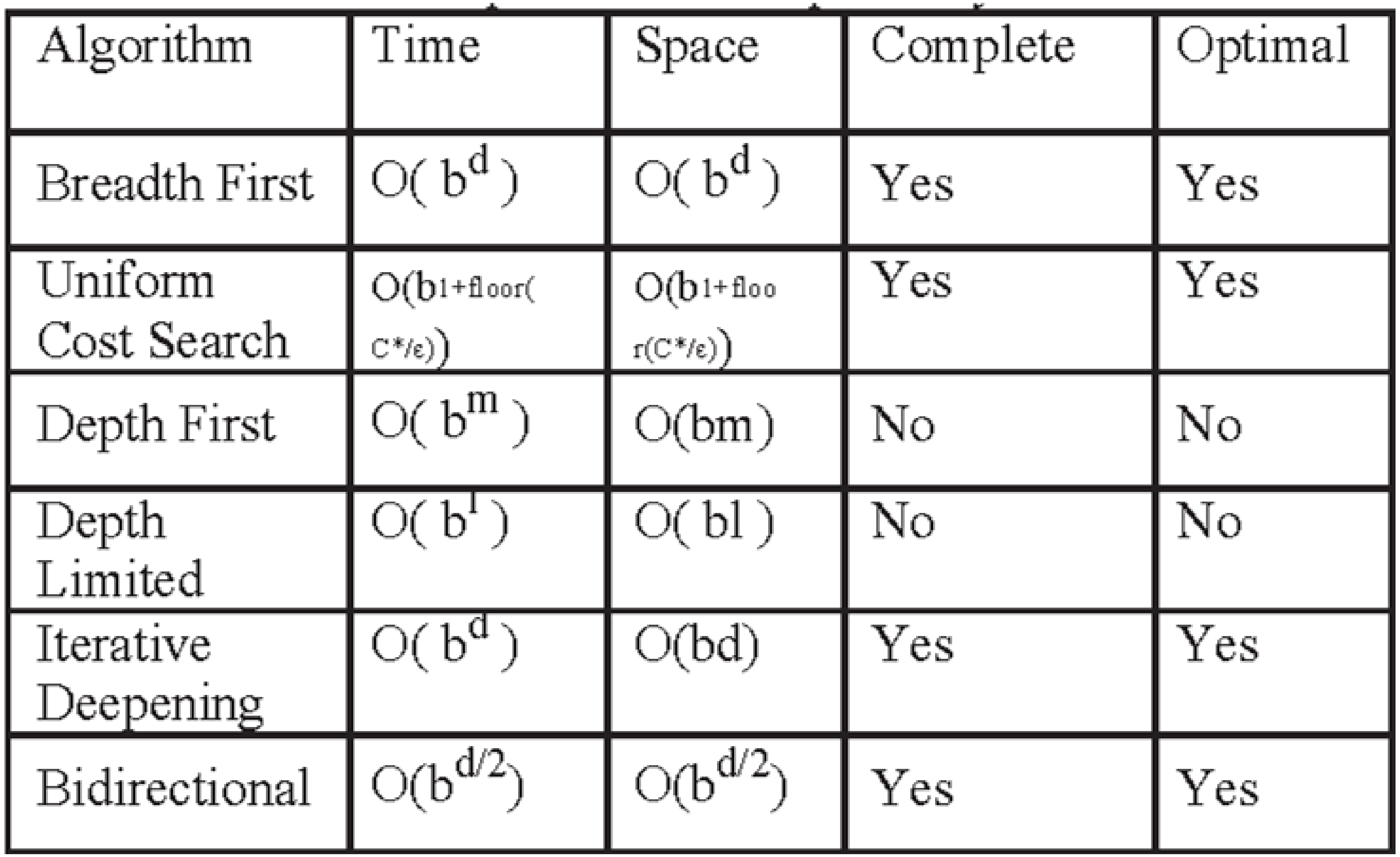
- Example:
- In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
- The algorithm terminates at node 9 where two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

Bidirectional Search



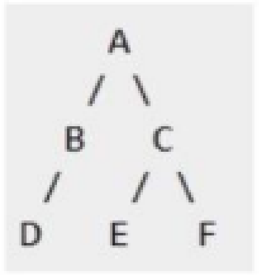
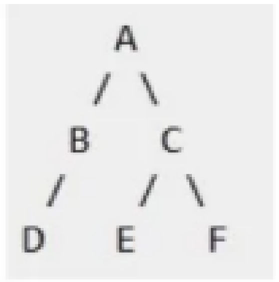
- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is $\mathbf{O(b^{d/2})}$.
- **Space Complexity:** Space complexity of bidirectional search is $\mathbf{O(b^{d/2})}$.
- **Optimal:** Bidirectional search is Optimal.
-

- **Advantages:**
- Bidirectional search is fast.
- Bidirectional search requires less memory
- **Disadvantages:**
- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**



Algorithm	Time	Space	Complete	Optimal
Breadth First	$O(b^d)$	$O(b^d)$	Yes	Yes
Uniform Cost Search	$O(b^{1+floor(C^*/\varepsilon)})$	$O(b^{1+floor(C^*/\varepsilon)})$	Yes	Yes
Depth First	$O(b^m)$	$O(bm)$	No	No
Depth Limited	$O(b^l)$	$O(bl)$	No	No
Iterative Deepening	$O(b^d)$	$O(bd)$	Yes	Yes
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$	Yes	Yes

S. No.	Parameters	BFS	DFS
1.	Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
4.	Technique	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
5.	Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
6.	Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
7.	Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
8.	Suitable for Decision Treestheirwinning	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.

9.	Visiting of Siblings/ Children	Here, siblings are visited before the children.	Here, children are visited before the siblings.
10.	Removal of Traversed Nodes	Nodes that are traversed several times are deleted from the queue.	The visited nodes are added to the stack and then removed when there are no more nodes to visit.
11.	Backtracking	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking
12.	Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and topological order etc.
13.	Memory	BFS requires more memory.	DFS requires less memory.
14.	Optimality	BFS is optimal for finding the shortest path.	DFS is not optimal for finding the shortest path.
15.	Space complexity	In BFS, the space complexity is more critical as compared to time complexity.	DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node.
16.	Speed	BFS is slow as compared to DFS.	DFS is fast as compared to BFS.
17.	When to use?	When the target is close to the source, BFS performs better.	When the target is far from the source, DFS is preferable.
18.	Tree	 A, B, C, D, E, F	 A, B, D, C, E, F

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.
- The informed search algorithm is more useful for **large search space**.
- Informed search algorithm uses the idea of **heuristic**, so it is also called **Heuristic search**.

Parameters	Informed Search	Uninformed Search
Known as	It is also known as Heuristic Search.	It is also known as Blind Search.
Using Knowledge	It uses knowledge for the searching process.	It doesn’t use knowledge for the searching process.
Performance	It finds a solution more quickly.	It finds solution slow as compared to an informed search.
Completion	It may or may not be complete.	It is always complete.
Cost Factor	Cost is low.	Cost is high.
Time	It consumes less time because of quick searching.	It consumes moderate time because of slow searching.
Direction	There is a direction given about the solution.	No suggestion is given regarding the solution in it.
Implementation	It is less lengthy while implemented.	It is more lengthy while implemented.
Efficiency	It is more efficient as efficiency takes into account cost and performance. The incurred cost is less and speed of finding solutions is quick.	It is comparatively less efficient as incurred cost is more and the speed of finding the Breadth-First solution is slow.

Computational requirements	Computational requirements are lessened.	Comparatively higher computational requirements.
Size of search problems	Having a wide scope in terms of handling large search problems.	Solving a massive search task is challenging.
Examples of Algorithms	Greedy Search A* Search AO* Search Hill Climbing Algorithm	Depth First Search (DFS) Breadth First Search (BFS)

Heuristics search

- Heuristics is an approach to problem-solving in which the objective is to produce a working solution within a **reasonable time frame**.
- Instead of looking for a perfect solution, heuristic strategies look for a **quick solution** that falls within an **acceptable range of accuracy**.
- A heuristic is a technique that is used to **solve a problem faster** than the classic methods.
- These techniques are used to find the approximate solution of a problem when classical methods do not.
- Heuristics are said to be the problem-solving techniques that result in **practical and quick solutions.**
- Heuristics are strategies that are derived from past experience with similar problems.
- Heuristics use practical methods and shortcuts used to produce the solutions that **may or may not be optimal**, but those solutions are sufficient in a given limited timeframe.

Why do we need heuristics?

- Heuristics are used in situations in which there is the requirement of a short-term solution.
- On facing complex situations with limited resources and time, Heuristics can help the companies to make quick decisions by shortcuts and approximated calculations.
- Most of the heuristic methods involve mental shortcuts to make decisions on **past experiences.**
- The heuristic method might **not always provide us the finest solution**, but it is assured that it helps us find a **good solution in a reasonable time**.
- Based on context, there can be different heuristic methods that correlate with the problem's scope.
- The most common heuristic methods are - trial and error, guesswork, the process of elimination, historical data analysis.
- These methods involve simply available information that is not particular to the problem but is most appropriate. They can include representative, affect, and availability heuristics.

- For finding a solution, by using the heuristic technique, one should carry out the following steps:

  1. Add domain—specific information to select what is the best path to continue searching along.
  2. Define a heuristic function h(n) that estimates the ‘goodness’ of a node n. Specifically, h(n) = estimated cost(or distance) of minimal cost path from n to a goal state.
  3. The term, heuristic means ‘serving to aid discovery’ and is an estimate, based on domain specific information that is computable from the current state description of how close we are to a goal.

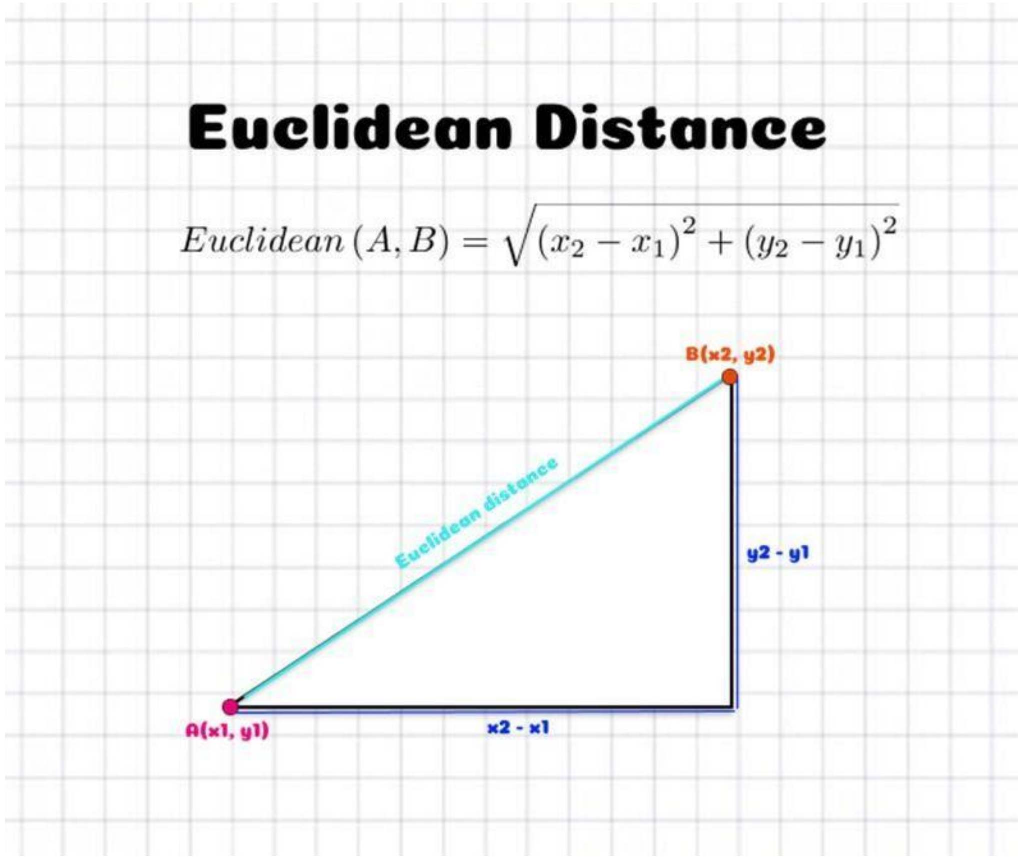
Heuristic function

- It is an function which gives an estimation of cost of getting from node n to goal node.
- Type :
- Admissible ($h(n) <= h'(n)$)
- Non - Admissible ($h(n) > h'(n)$)

How do we calculate heuristics value?

- No need to check all state but state with minimum heuristic value.

1. **Euclidean distance(straight line distance)**



2. **Manhattan distance (vertical and horizontal distance)**

START

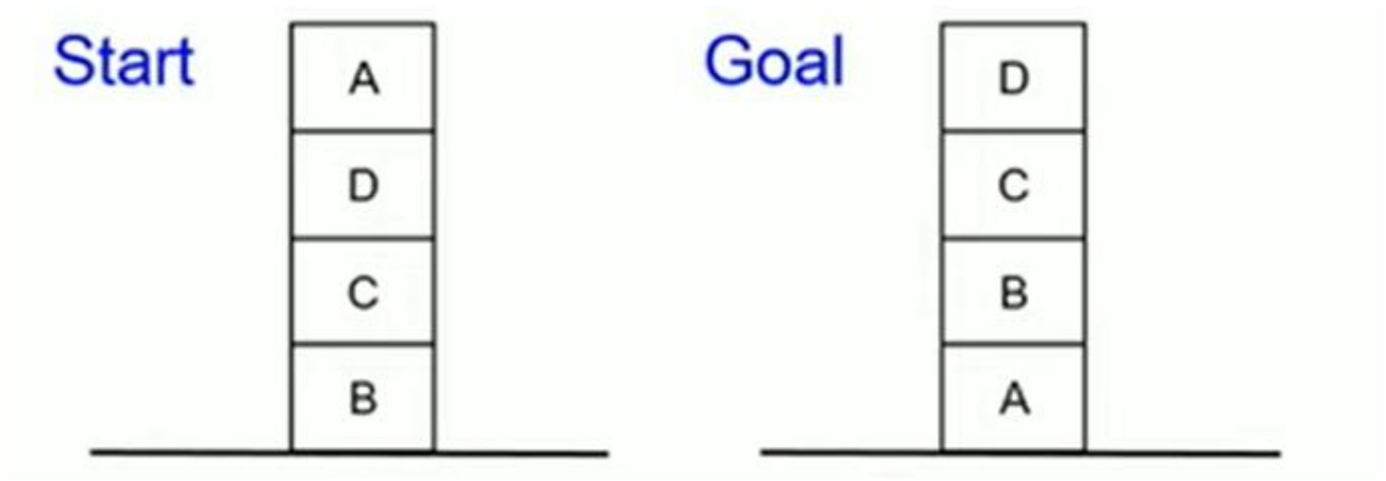
2	6	1
	7	8
3	5	4

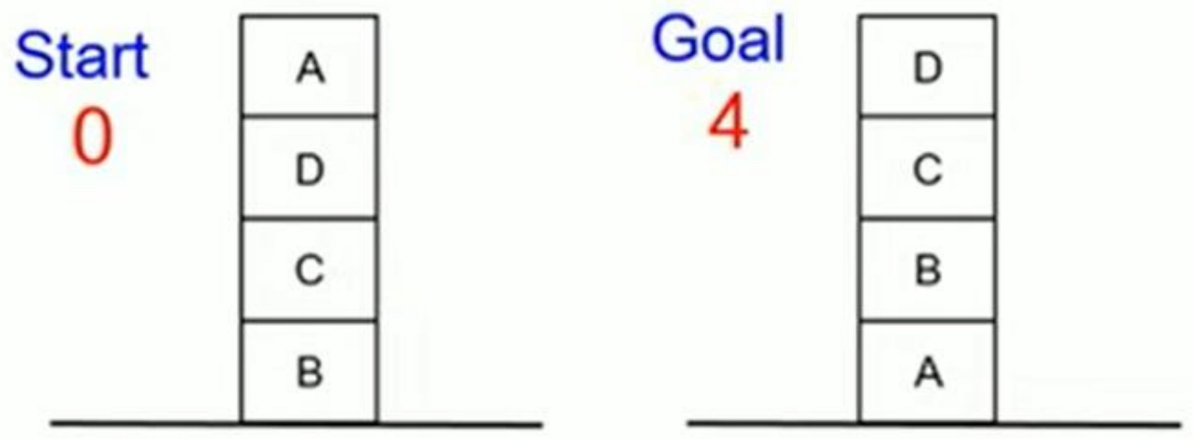
GOAL

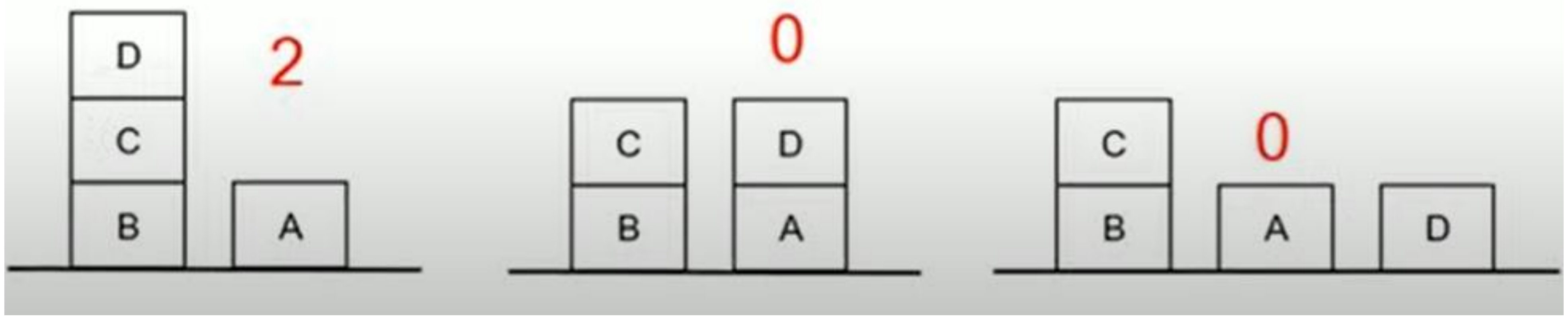
1	2	3
4	5	6
7	8	

Blocks World - Local heuristic function

- +1 for each block that is resting on the thing it is supposed to be resting on.
- -1 for each block that is resting on a wrong thing.
- Local optimum

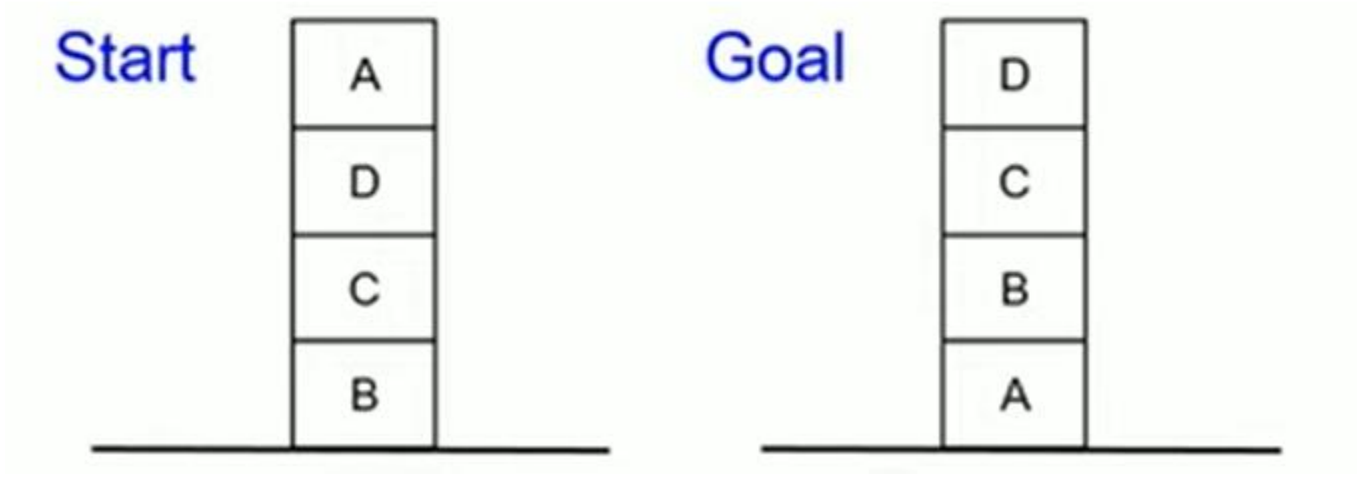


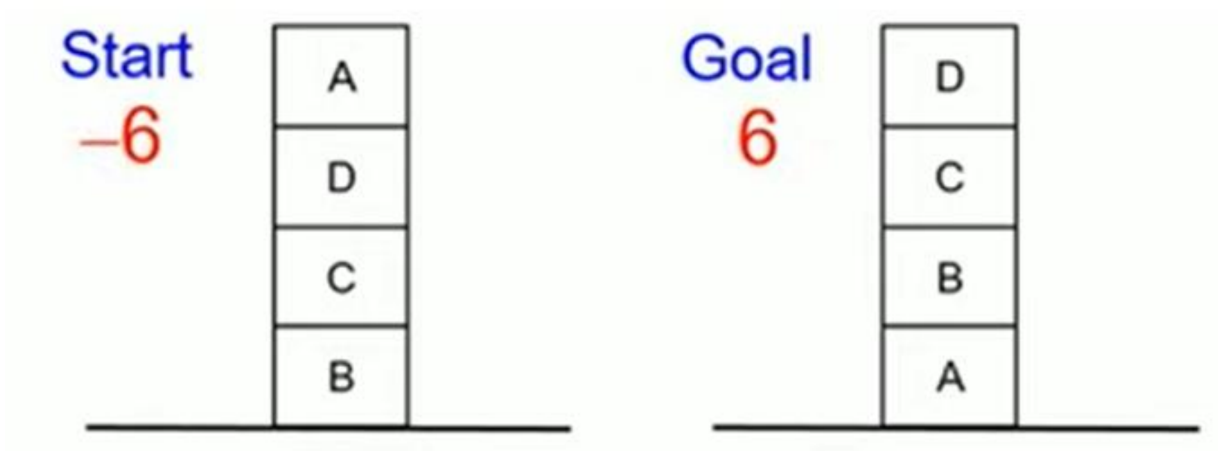

Start
0
A
D
C
B
Goal
4
D
C
B
A

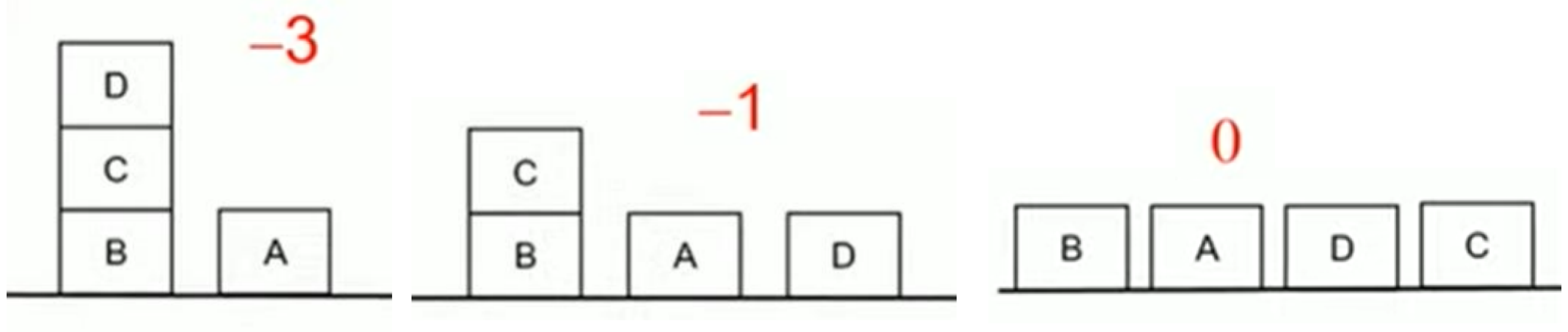

D
C
B
2
A
0
C
B
D
A
C
B
0
A
D

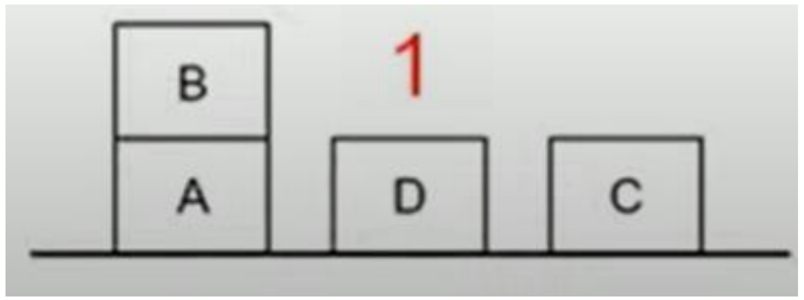
Blocks World - Global heuristic function

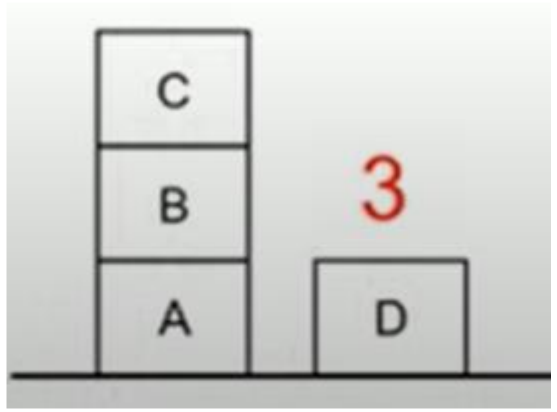
- For each block that has the correct support structure: +1 to every block in the support structure.
- For each block that has a wrong support structure: -1 to every block in the support structure.

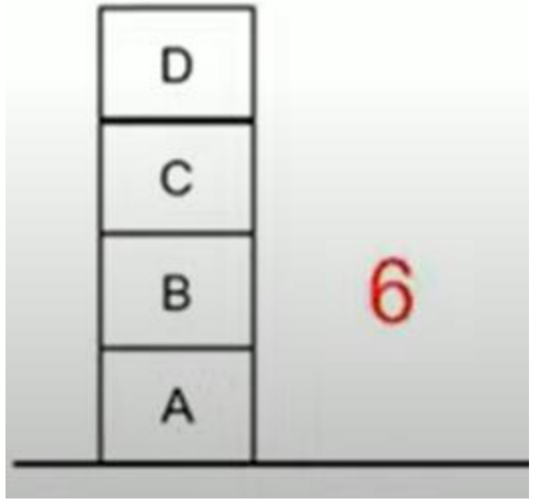



Start
−6
A
D
C
B
Goal
6
D
C
B
A


−3
D
C
B
A
−1
C
B
A
D
0
B
A
D
C


B
A
1
D
C


C
B
A
3
D

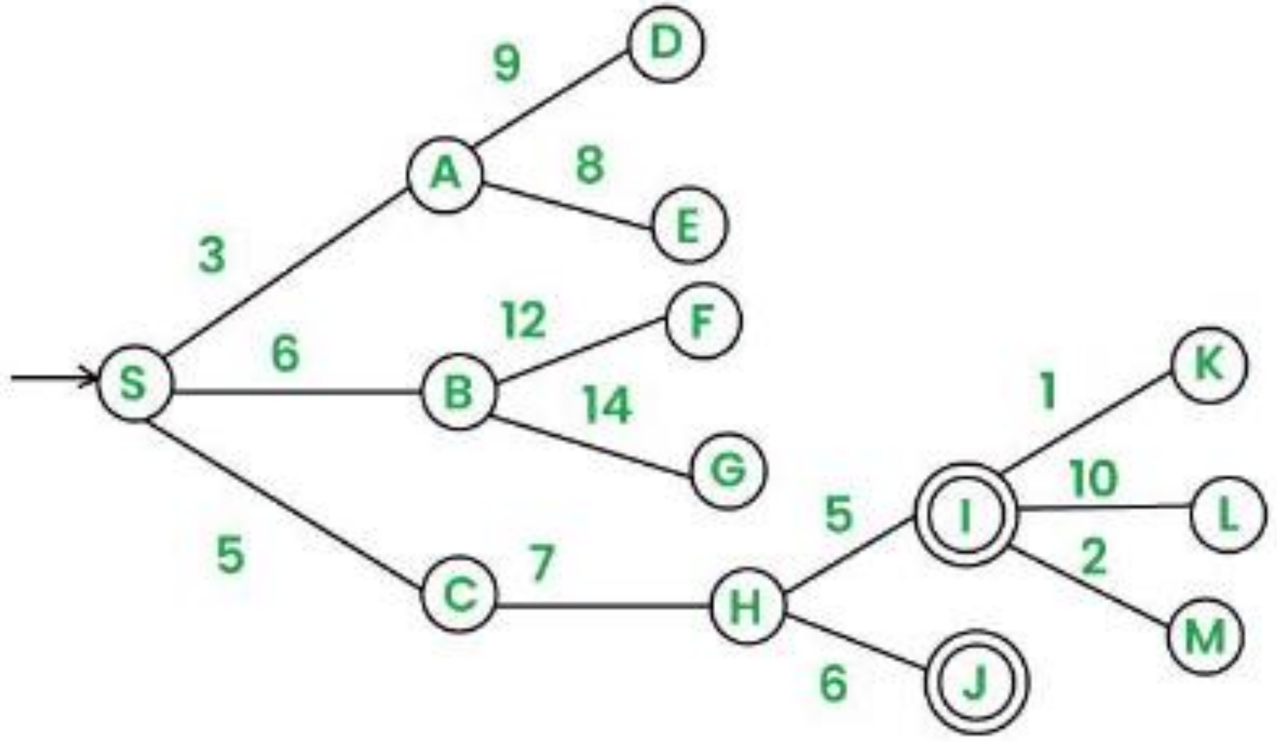

D
C
B
A
6

Best First Search

- In BFS and DFS, when we are at a node, we can consider any of the adjacent as the next node. So both BFS and DFS blindly explore paths without considering any cost function.
- The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.
- It is the combination of depth-first search and breadth-first search algorithms.

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 f(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.



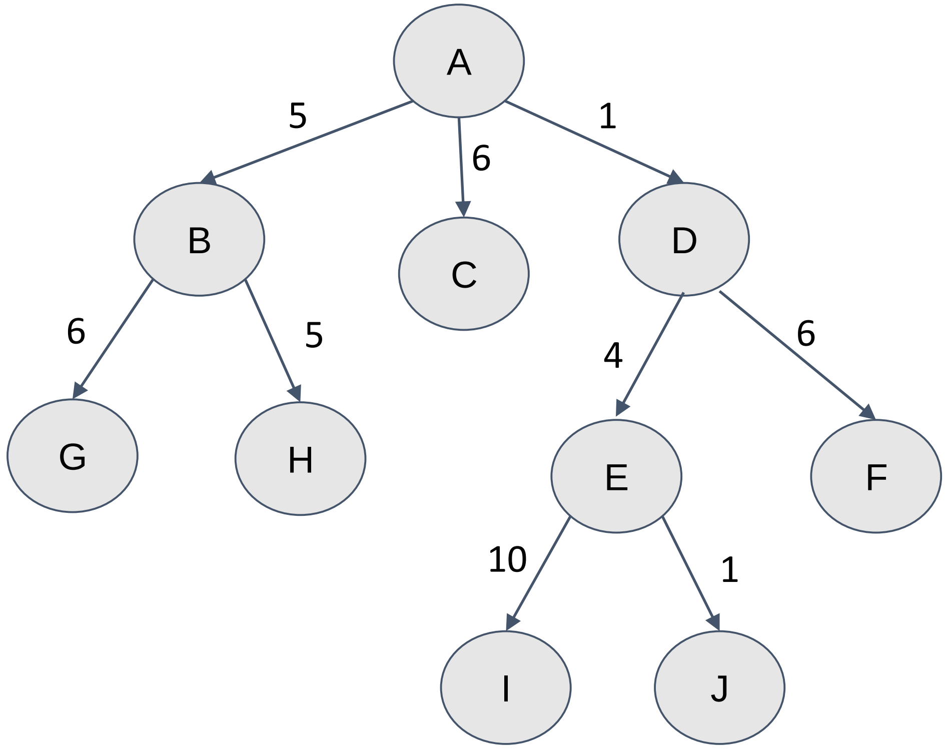
S
A
B
C
D
E
F
G
H
I
J
K
L
M
3
6
5
9
8
12
14
7
5
6
1
10
2

Open List (PRIORITY QUEUE)

A				
D	B	C		
E	B	C	F	
J	B	C	F	I
B	C	F	I	

Closed List

A			
A	D		
A	D	E	
A	D	E	J



Time Complexity: The worst case time complexity of best first search is O(bm).

Space Complexity: The worst case space complexity of best first search is O(bm). Where, m is the maximum depth of the search space.

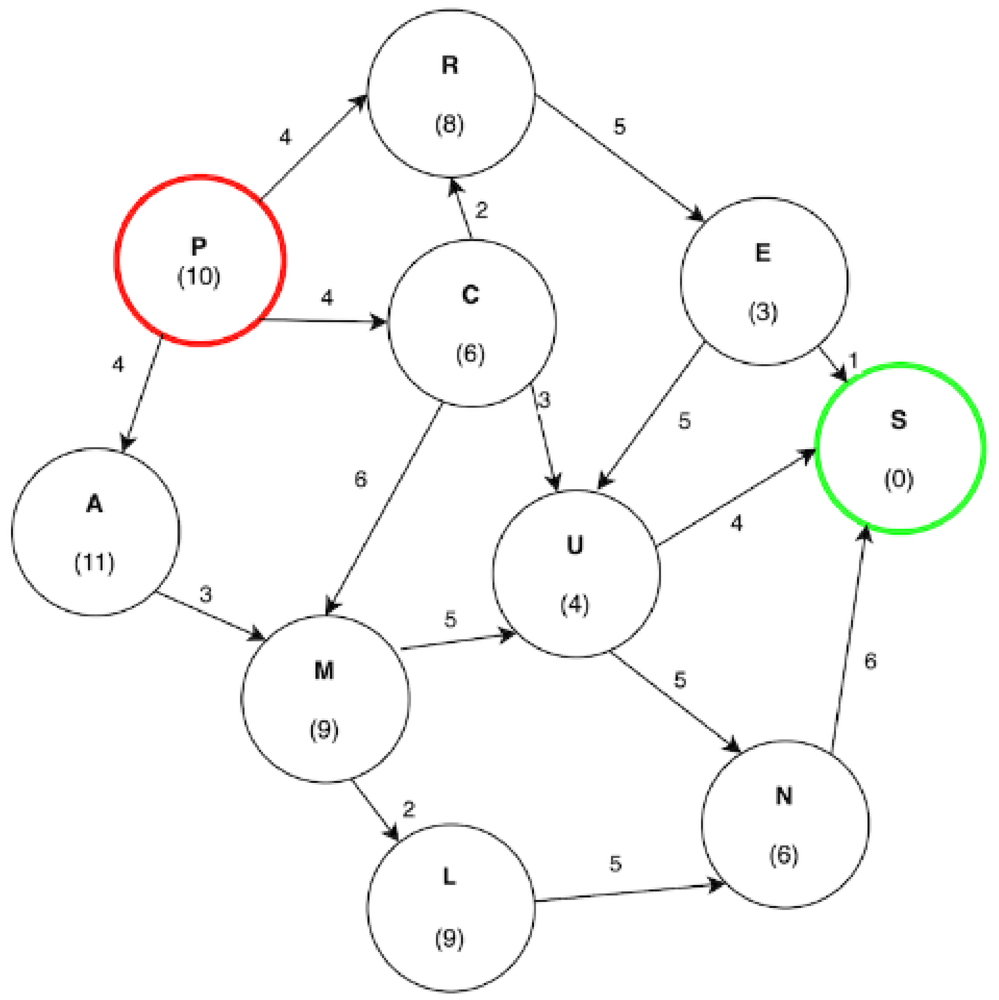
Complete:Best-first search is also incomplete, even if the given state space is finite.

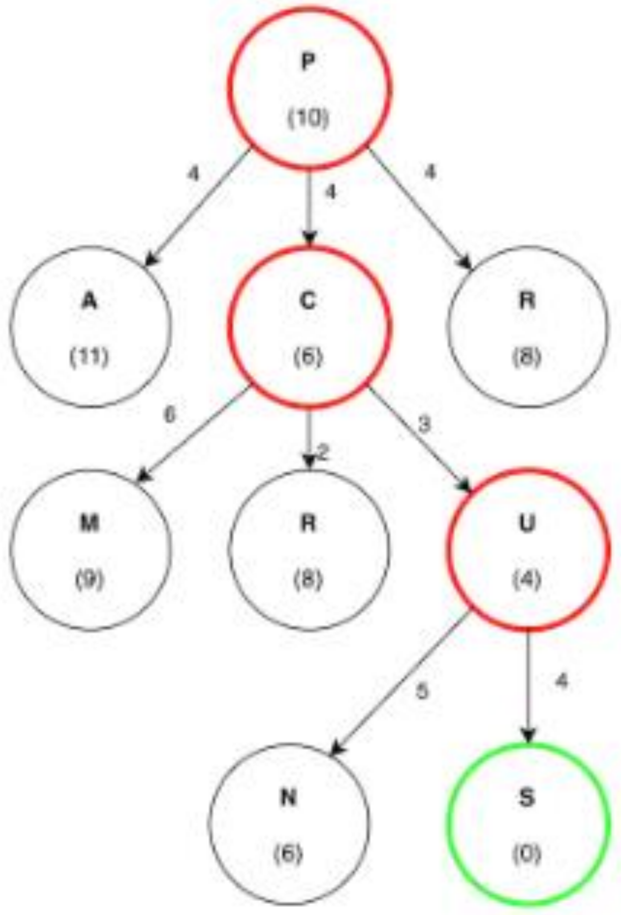
Optimal: Best first search algorithm is not optimal.

Greedy best-first search algorithm

- 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.
- 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 minimum cost is estimated by heuristic function

- The **evaluation function is f(n) = h(n)**
- Were, h(n)= estimated cost from node n to the goal.
- Greedy search ignores the cost of the path that has already been traversed to reach n
- Therefore, the solution given is not necessarily optimal.


R
(8)
P
(10)
C
(6)
E
(3)
S
(0)
A
(11)
U
(4)
M
(9)
N
(6)
L
(9)
4
5
2
4
4
1
3
5
6
4
3
5
5
6
2
5


P
(10)
4
4
4
A
(11)
C
(6)
R
(8)
6
2
3
M
(9)
R
(8)
U
(4)
5
4
N
(6)
S
(0)

- The total cost for the path (**P -> C -> U -> S**) evaluates to 11.
- The potential problem with a greedy best-first search is revealed by the path (**P -> R -> E -> S**) having a cost of 10, which is lower than (**P -> C -> U -> S**).
- Greedy best-first search ignored this path because it does not consider the edge weights.

- Greedy best-first search can start down an **infinite path** and never return to try other possibilities, it is **incomplete.**
- Because of its greediness the search makes choices that can lead to a **dead end**; then one backs up in the search tree to the deepest unexpanded node
- Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end
- The quality of the heuristic function determines the practical usability of greedy search

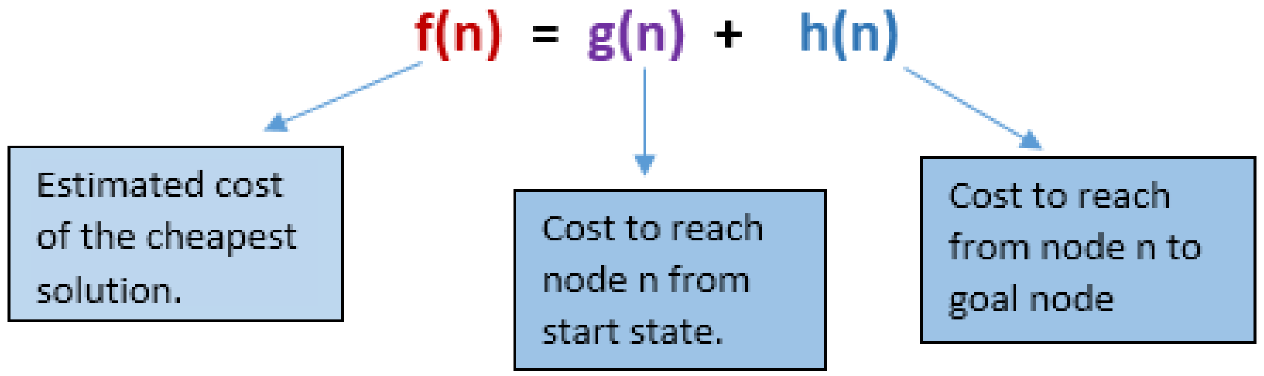
- Greedy search is not optimal
- Greedy search is incomplete without systematic checking of repeated states.
- In the worst case, the Time and Space Complexity of Greedy Search are both O(bm),

  Where

  - b is the branching factor and
  - m the maximum path length

A* search

- 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 combined features of **UCS and 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. This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it 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.**


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Algorithm of A* search:

Step 1: Place the starting node in the OPEN list.

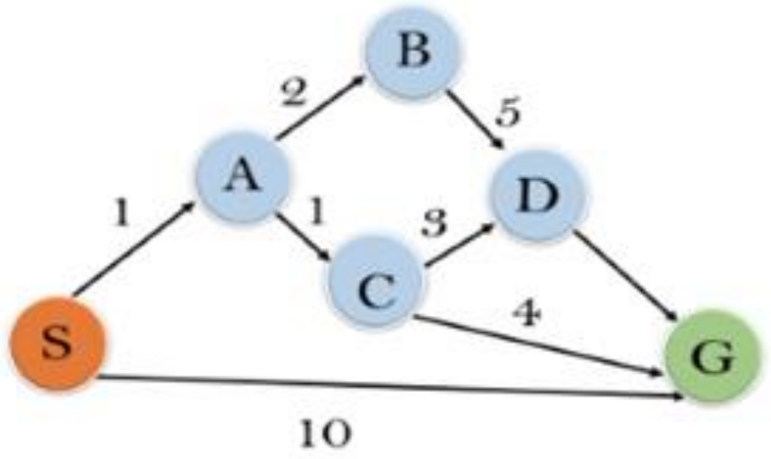
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

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

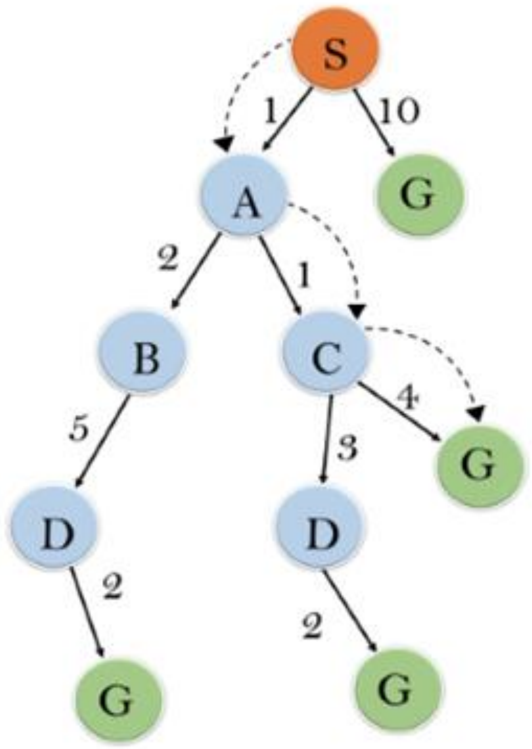
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

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.



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



Initialization: {(S, 5)}

Iteration 1: {(S--> A, 4), (S-->G, 10)}

Iteration 2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration 3: {(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)

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.

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.

Iterative Deepening A* algorithm (IDA*)

- **Iterative deepening A* (IDA*)** is a graph traversal and path-finding method that can determine the **shortest route in a weighted graph** between a defined start node and any one of a group of goal nodes.
- It is a kind of iterative deepening depth-first search that adopts the **A* search** algorithm idea of using a heuristic function to assess the remaining cost to reach the goal.
- A memory-limited version of A* is called IDA*. It performs all operations that A* does and has optimal features for locating the shortest path, but it **occupies less memory**.
- Iterative Deepening A Star uses a heuristic to choose which nodes to explore and at which depth to stop, as opposed to Iterative Deepening DFS, which utilizes simple depth to determine when to end the current iteration and continue with a higher depth.

How IDA* algorithm work?

Step 1: Initialization
Set the root node as the current node, and find the f-score.

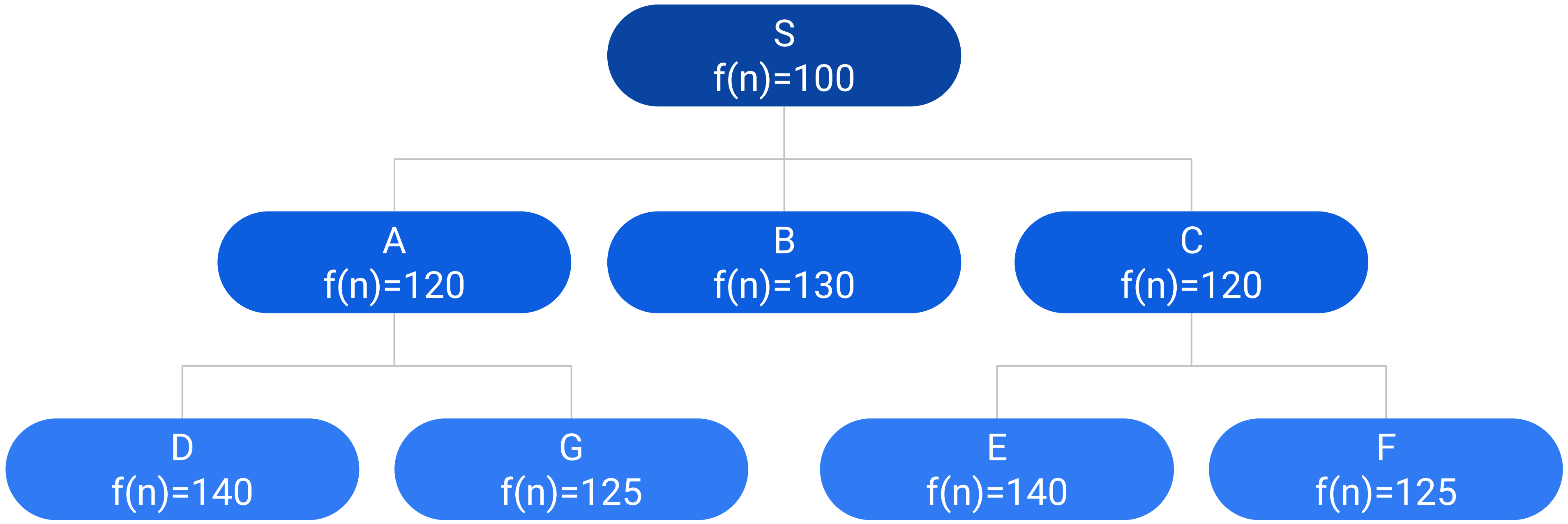
Sep 2: Set threshold
Set the cost limit as a **threshold** for a **node** i.e the **maximum f-score** allowed for that node for further explorations.

Step 3: Node Expansion
Expand the current node to its children and find f-scores.

Step 4: Pruning
If for any **node** the **f-score > threshold**, prune that node because it’s considered too expensive for that node. and store it in the **visited node list.**

Step 5: Return Path
If the **Goal node** is found then return the **path** from the start node Goal node.

Step 6: Update the Threshold
 If the Goal node is not found then **repeat from step 2** by changing the threshold with the minimum pruned value from the **visited node list**. And Continue it until you reach the goal node.


S
f(n)=100
A
f(n)=120
B
f(n)=130
C
f(n)=120
D
f(n)=140
G
f(n)=125
E
f(n)=140
F
f(n)=125

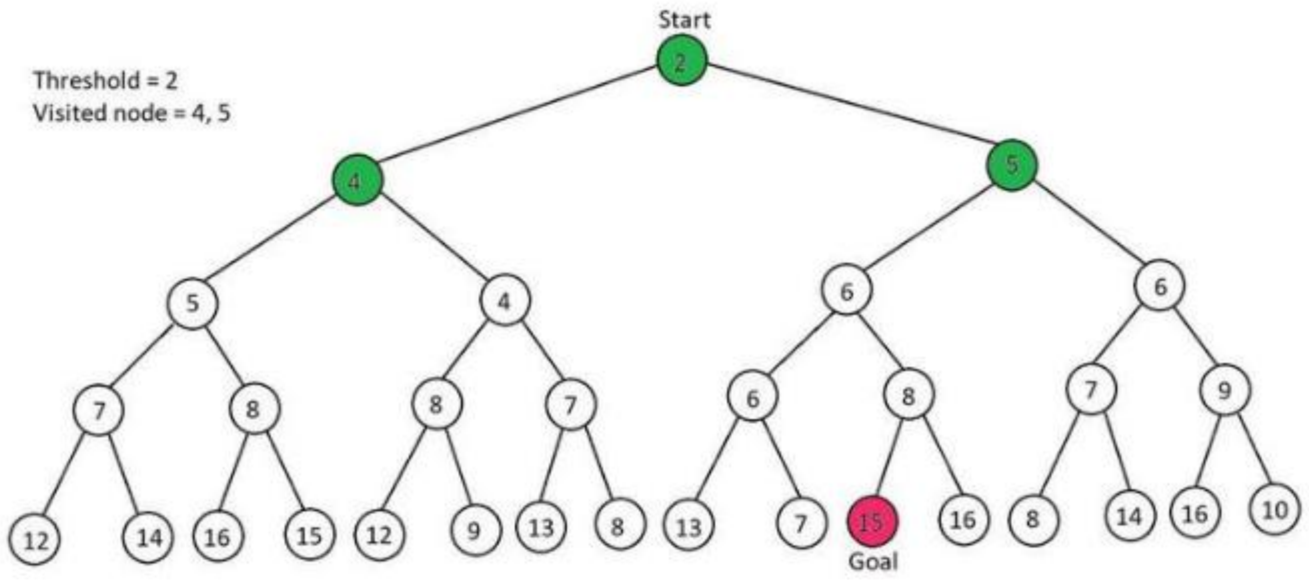
ITERATION 1 [LEVEL 0]
THRESHOLD_VALUE=100
A [120>100]
B [130>100]
C [120>100]

ITERATION 2 [LEVEL 1]
THRESHOLD_VALUE=120
D [140>120]
G [125>120]
B [130>120]
E [140>120]
F [125>120]

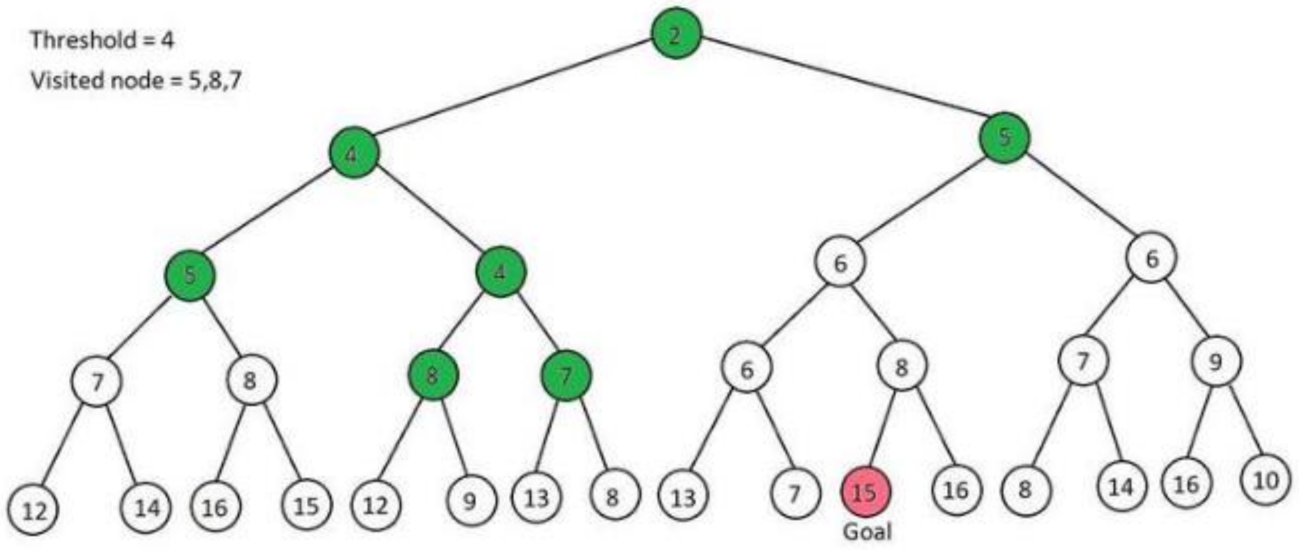
ITERATION 2 [LEVEL 2]
THRESHOLD_VALUE=125
D [140>120]
B [130>120]
E [140>120]

GOAL PATH : S -> A -> G

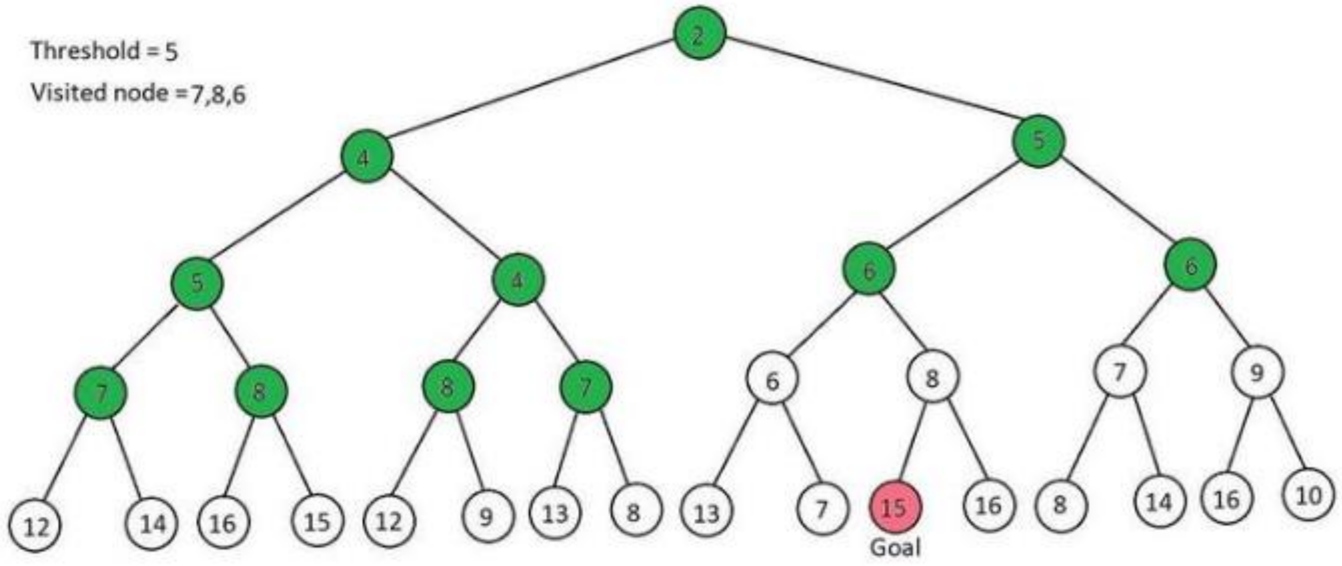
Iteration 1



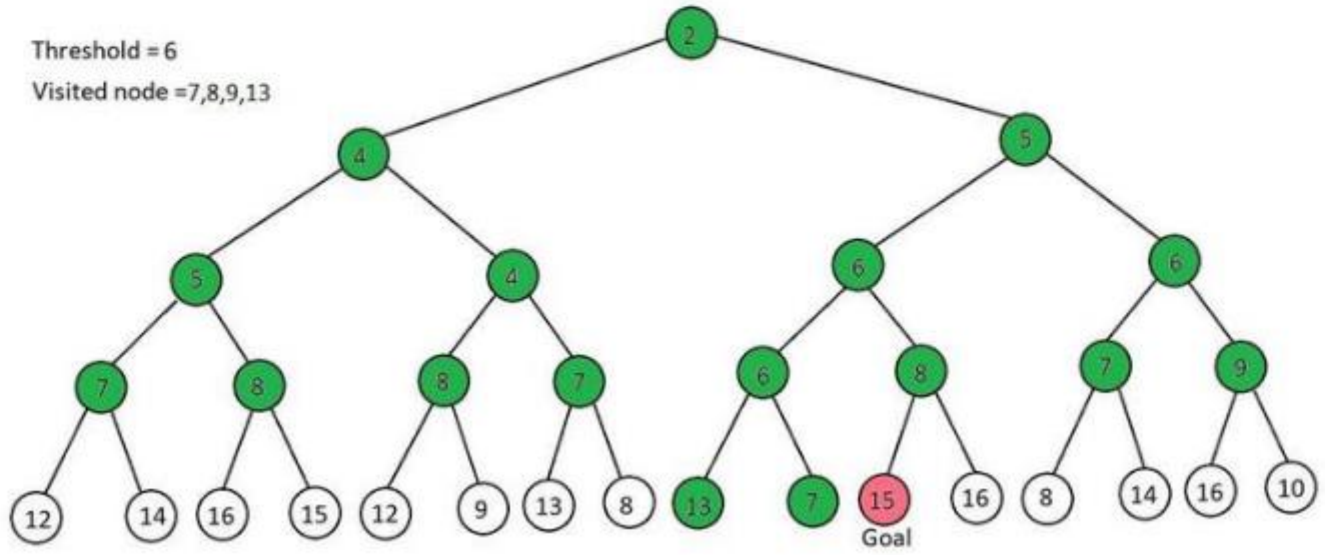
Iteration 2



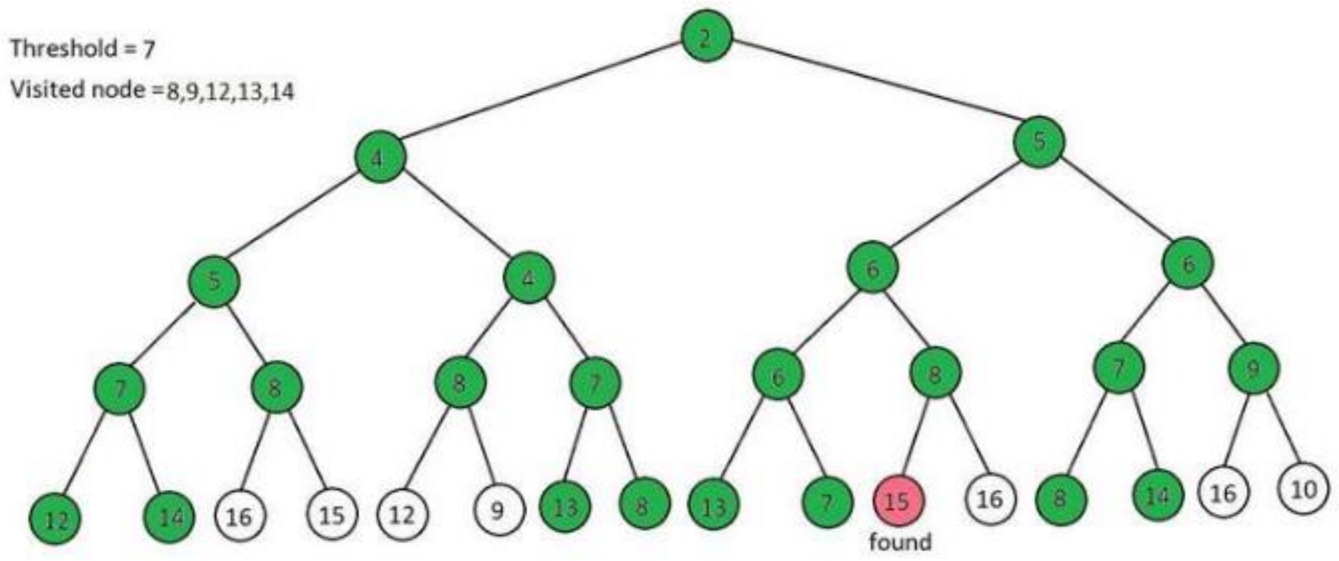
Iteration 3



Iteration 4

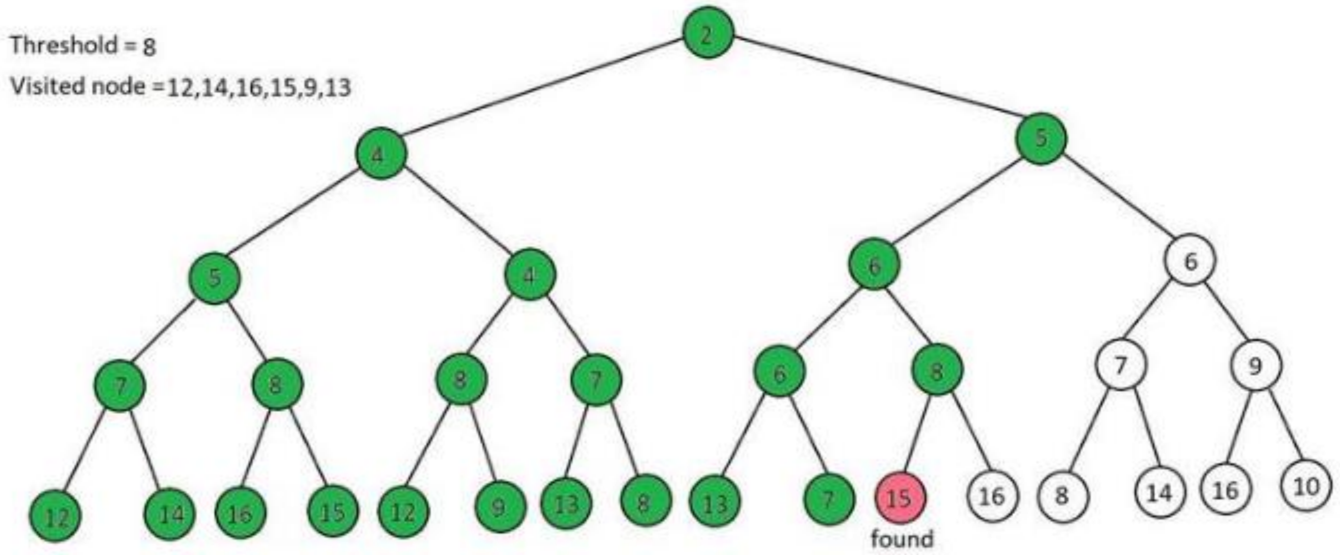


Iteration 5



Iteration 6

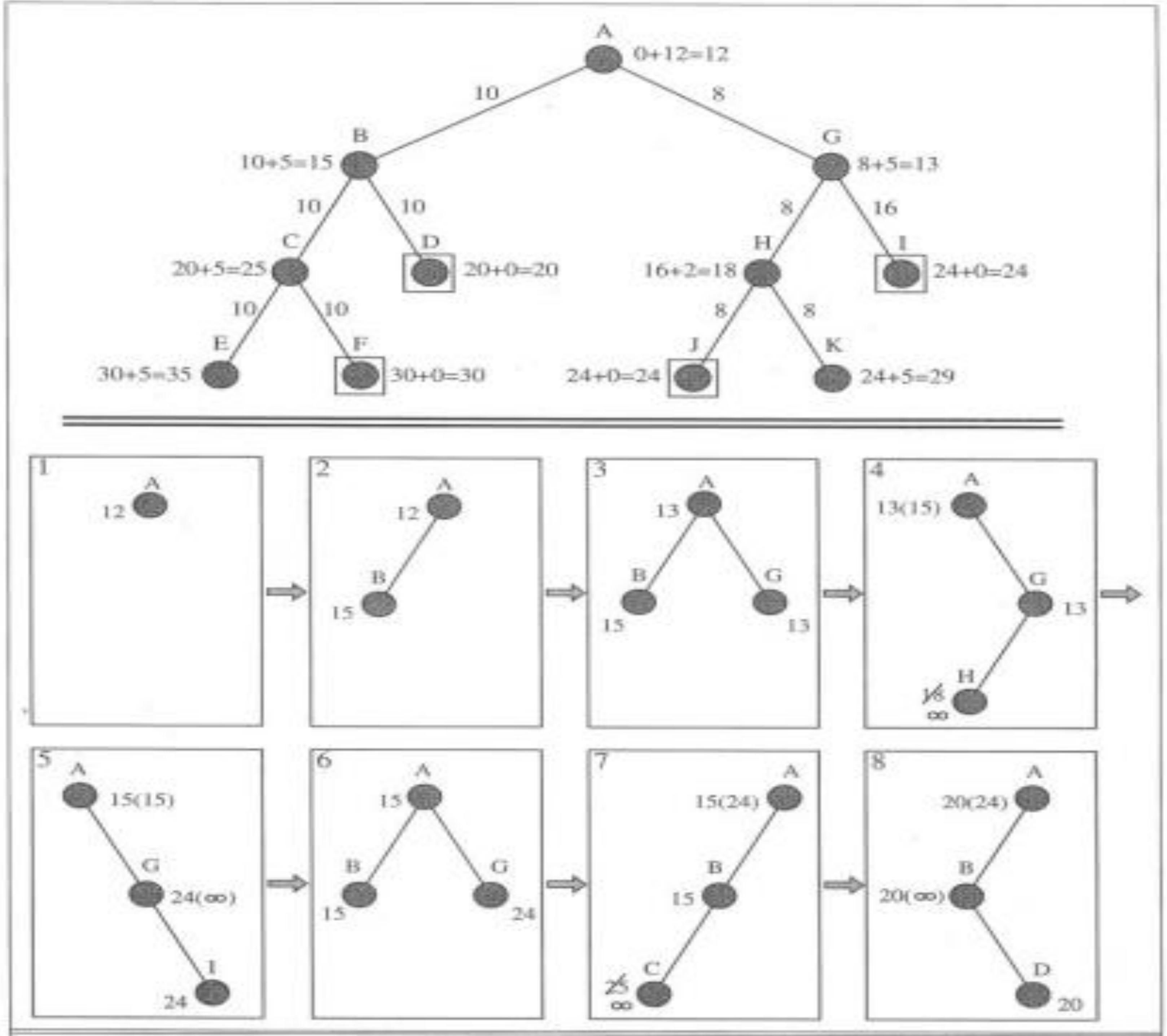
The goal path is **2–>5–>6–>8–>15**



SMA* (Simplified Memory Bounded A*) ALGORITHM

- IDA*'s difficulties in certain problem spaces can be traced to using too little memory.
- SMA algorithm, which can make **use of all available memory** to carry out the search.
- Using more memory can only improve search efficiency —one could always ignore the additional space, but usually it's better to remember a node than to have to regenerate it when needed.

- SMA has the following properties:
  - It will utilize whatever memory is made available to it.
  - It **avoids repeated states** as far as its memory allows.
  - It is complete if the available memory is sufficient to store the shallowest solution path.
  - It is **optimal** if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution that can be reached with the available memory.
  - When enough memory is available for the entire search tree, the search is optimally efficient.
- **forgotten nodes**


A 0+12=12
10
8
B 10+5=15
G 8+5=13
10
10
8
16
C 20+5=25
D 20+0=20
H 16+2=18
I 24+0=24
10
10
8
8
E 30+5=35
F 30+0=30
J 24+0=24
K 24+5=29
1 A 12
2 A 12 B 15
3 A 13 B 15 G 13
4 A 13(15) G 13 H 18 ∞
5 A 15(15) G 24(∞) I 24
6 A 15 B 15 G 24
7 A 15(24) B 15 C 25 ∞
8 A 20(24) B 20(∞) D 20

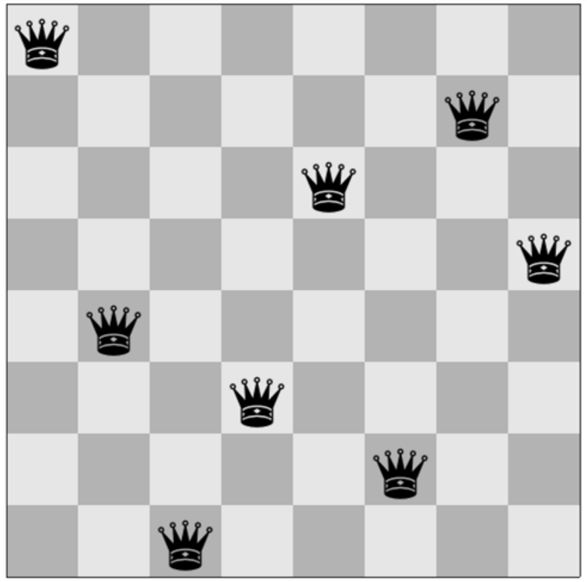
Beyond classical search

Local Search Algorithms and Optimization Problems

- Hill Climbing Search
- Simulated Annealing
- Local Beam Search
- Evolutionary Algorithm - Genetic Algorithm

Local Search Algorithm

- The Local search algorithm searches only the **final state**, not the path to get there.
- For example, in the 8-queens problem, we care only about finding a valid final configuration of 8 queens (8 queens arranged on chess board, and no queen can attack other queens) and not the path from initial state to final state.



- Local search algorithms operate by searching from a **start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.**
- They are **not systematic** - they might never explore a portion of the search space where a solution actually resides.
- They searches only the **final state**

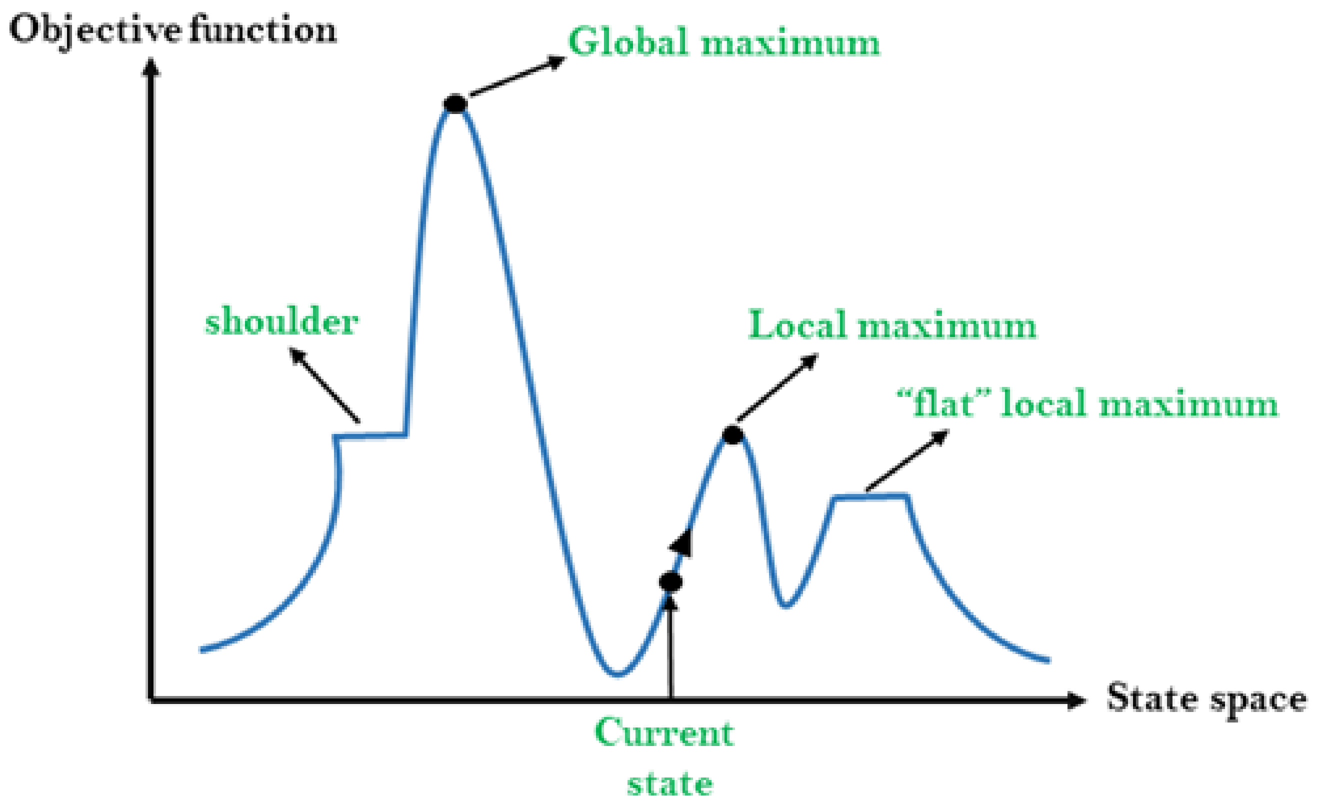
Hill Climbing

- Hill Climbing is a form of **heuristic search** algorithm which is used in solving optimization related problems in Artificial Intelligence domain.
- The algorithm starts with a **non-optimal state and iteratively improves** its state until some predefined condition is met. The condition to be met is based on the heuristic function.
- The aim of the algorithm is to reach an optimal state which is better than its current state. The starting point which is the non-optimal state is referred to as the base of the hill and it tries to constantly iterate (climb) until it reaches the peak value, that is why it is called **Hill Climbing Algorithm.**
- Hill Climbing Algorithm is a **memory-efficient** way of solving large computational problems.
- It takes into account the current state and immediate neighbouring state.

- The Hill Climbing Problem is particularly useful when we want to maximize or minimize any particular function based on the input which it is taking.
- The most commonly used Hill Climbing Algorithm is “**Travelling Salesman” Problem**” where we have to minimize the distance travelled by the salesman. Hill Climbing Algorithm may not find the global optimal (best possible) solution but it is good for finding local minima/maxima efficiently.
- Following are few of the **key features** of Hill Climbing Algorithm
- **Greedy Approach**: The algorithm moves in the direction of optimizing the cost i.e. finding Local Maxima/Minima
- **No Backtracking**: It cannot remember the previous state of the system so backtracking to the previous state is not possible
- **Feedback Mechanism**: The feedback from the previous computation helps in deciding the next course of action i.e. whether to move up or down the slope

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 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

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

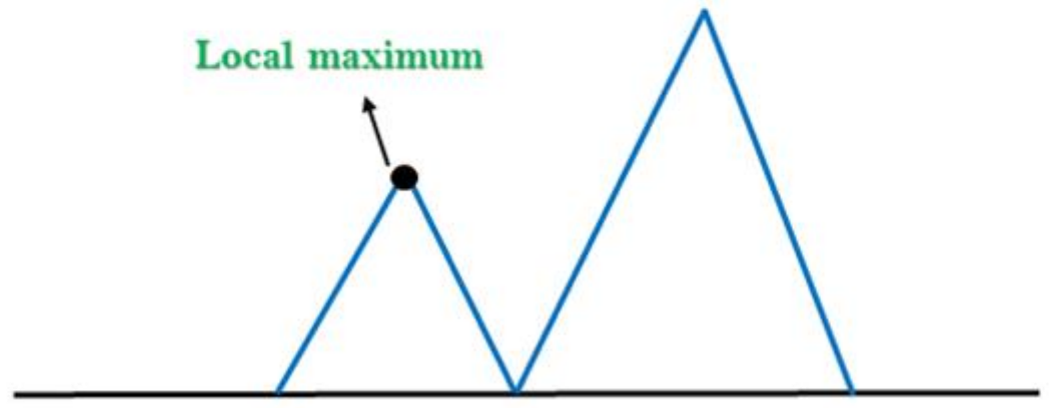
2. Steepest-Ascent hill climbing:

- 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

Problems in Hill Climbing Algorithm:

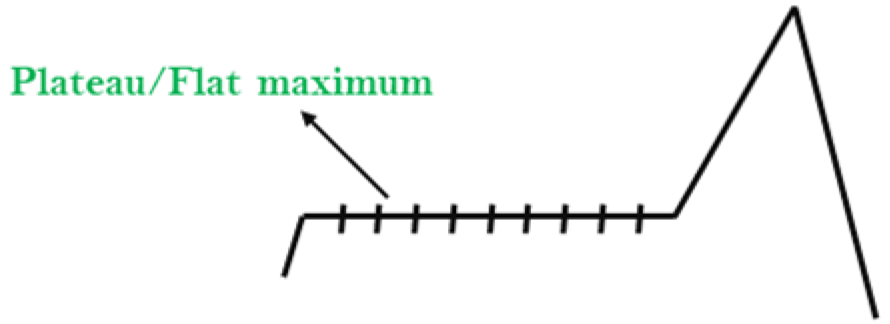
1. **Local Maximum**: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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.



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.



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.

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.

Advantages

- Easy to code for complex problems also
- Give good solution.
- Statistically guarantees finding of optimal solution.

Disadvantages

- Slow process.
- Can't tell whether the optimal solution is found.
- Some other method is also required.

Beam Search Algorithm

A heuristic search algorithm that examines a graph by **extending the most promising node in a limited set** is known as beam search algorithm.

The number of nodes n represents the **beam width**.

Beam width is 1, then hill climbing. For infinite , then best first search.

This algorithm only keeps the **lowest number of nodes on open list**.

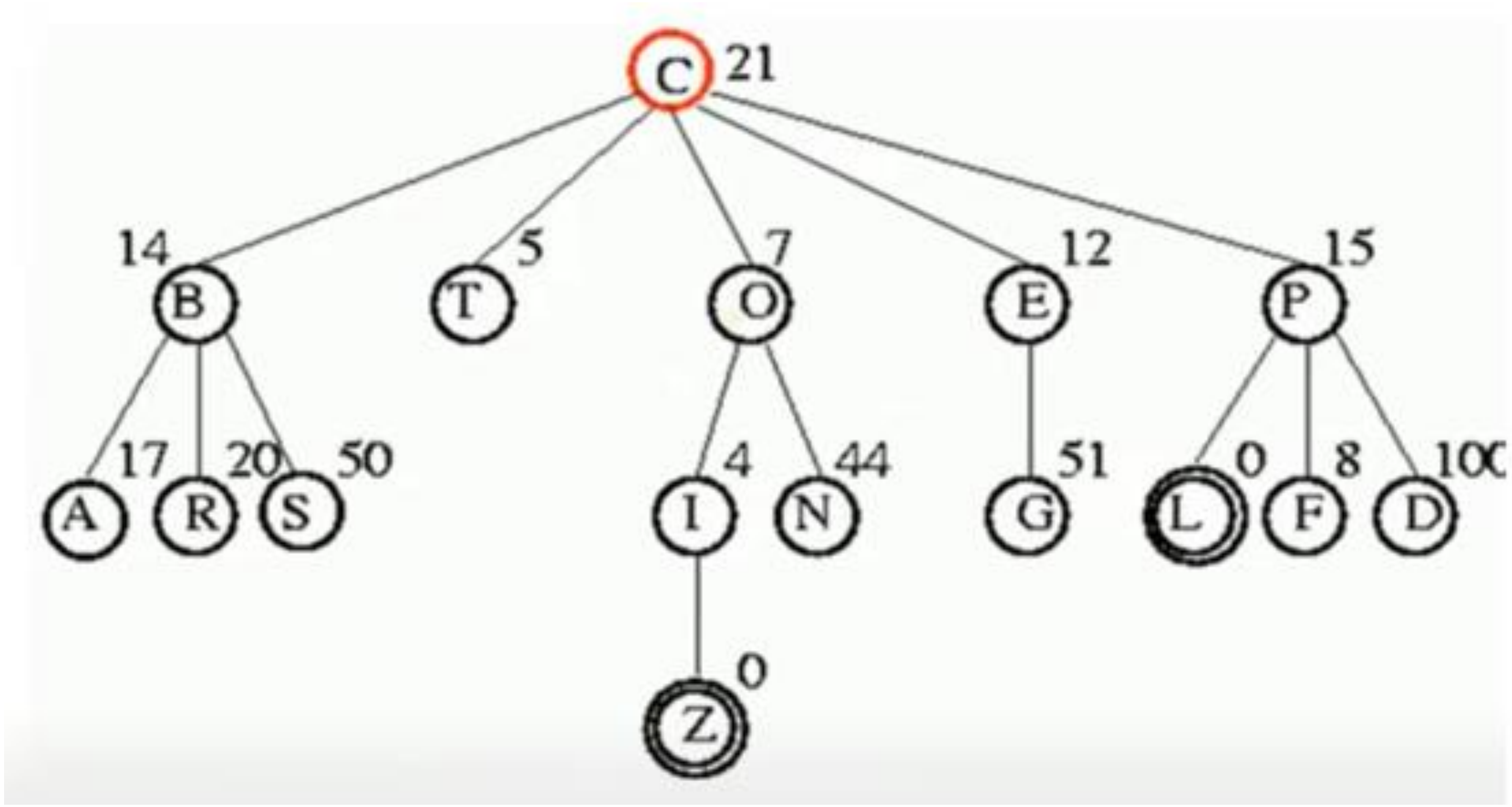
Components of Beam Search

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

1. The problem usually represented as **graph** and **contains a set of nodes** in which one or more of the nodes represents a goal.

2. The **set of heuristic rules for pruning**: are rules specific to the problem domain and **prune unfavorable nodes** from memory regarding the problem domain.

3. A memory with a **limited available capacity** : 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.


C 21
B 14
T 5
O 7
E 12
P 15
A 17
R 20
S 50
I 4
N 44
G 51
L 0
F 8
D 10
Z 0

- Time Complexity of Beam Search
  - The time complexity of the Beam Search algorithm depends on the following things, such as:
  - The accuracy of the heuristic function.
  - 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:
  - Beam Search's memory consumption is its most desirable trait.
  - Since the algorithm only stores B nodes at each level in the search tree.
  - The worst-case space complexity = ***O(B*m)***

Genetic algorithm

***A genetic algorithm is an adaptive heuristic search algorithm inspired by "Darwin's theory of evolution in Nature*."**

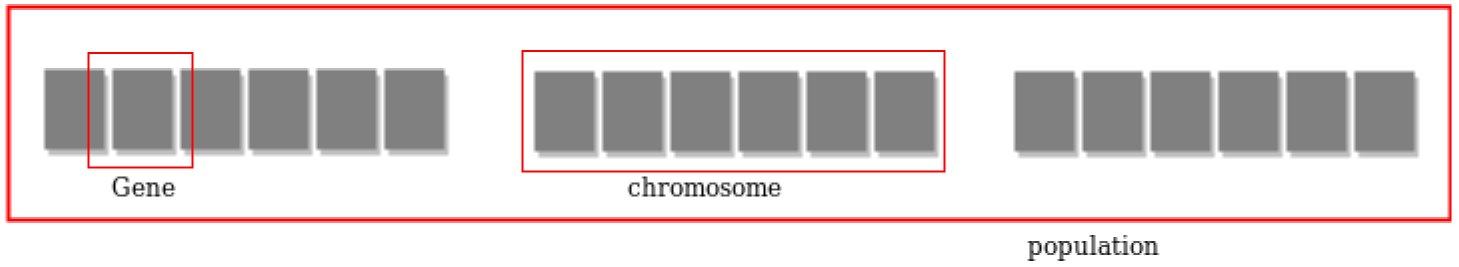
They are commonly used to generate high-quality solutions for optimization problems and search problems.

Basic terminologies

- **Population:** Population is the subset of all possible or probable solutions, which can solve the given problem.
- **Chromosomes:** A chromosome is one of the solutions in the population for the given problem, and the collection of gene generate a chromosome.
- **Gene:** A chromosome is divided into a different gene, or it is an element of the chromosome.

Search space

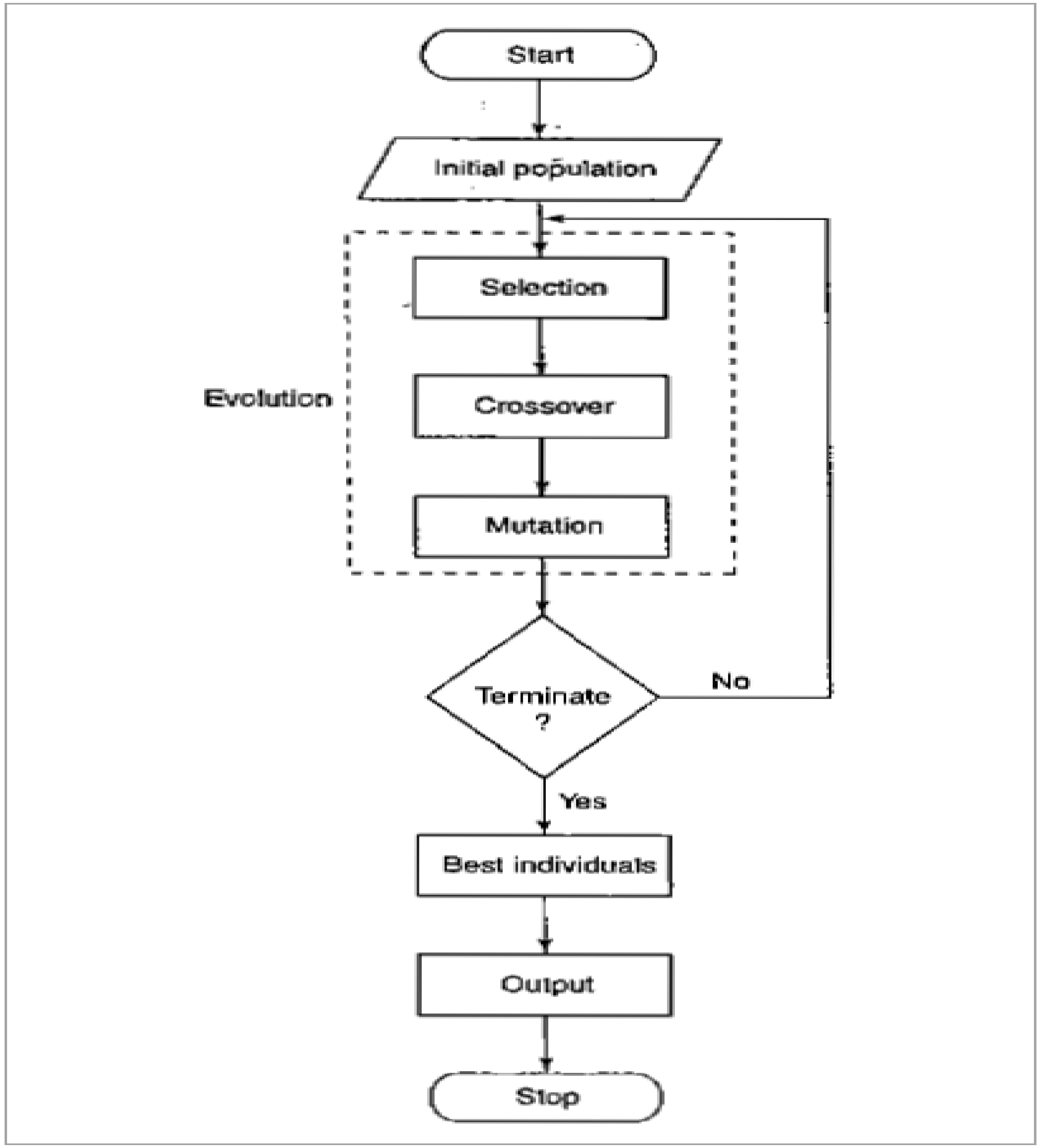
The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual)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



- **Fitness Function:** The fitness function is used to determine the individual's fitness level in the population. It means the ability of an individual to compete with other individuals. In every iteration, individuals are evaluated based on their fitness function.
- **Genetic Operators:** In a genetic algorithm, the best individual mate to regenerate offspring better than parents. Here genetic operators play a role in changing the genetic composition of the next generation.
- **Selection:** After calculating the fitness of every existent in the population, a selection process is used to determine which of the individualities in the population will get to reproduce and produce the seed that will form the coming generation.

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

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from “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.

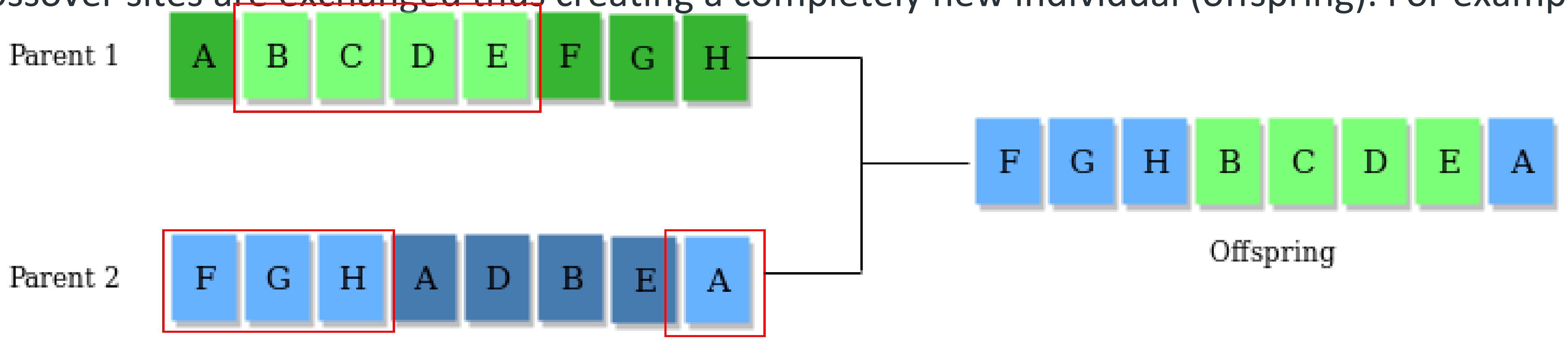

Start
Initial population
Selection
Evolution
Crossover
Mutation
Terminate ?
No
Yes
Best individuals
Output
Stop

Operators of Genetic Algorithms

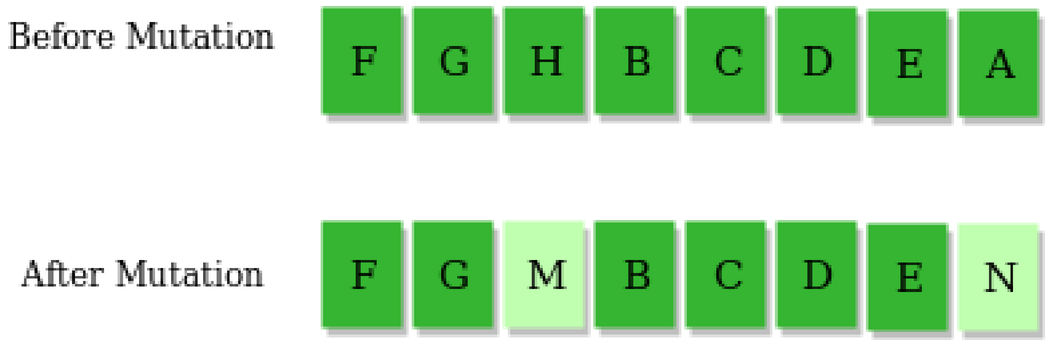
Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to 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 the population to avoid premature convergence. For example –



Search with Nondeterministic Actions

Belief State

• When the environment is **partially observable**, and the agent doesn't know for sure what state it is in; and

• when the environment is **nondeterministic**, the agent doesn't know what state it transitions to after taking an action

An agent will thinking "I'm either in state s1 or s2, and if I do action a, end up in state s2, s4 or s5."

• A set of physical states that the agent believes are possible a belief states.

Conditional Plan

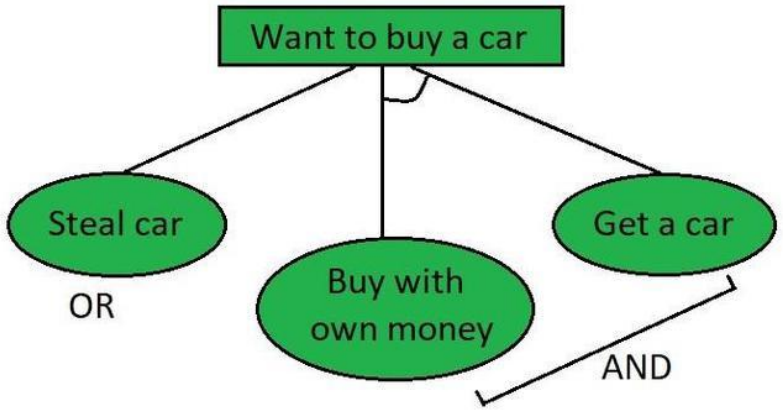
- In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence,
- but rather a conditional plan (sometimes called a contingency plan or a strategy)
- that specifies what to do, depending on what percepts agent receives, while executing the plan.

AO* algorithm

Best-first search is what the AO* algorithm does. The AO* method **divides** any given difficult **problem into a smaller group** of problems that are then resolved **using the AND-OR** graph concept.

AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems.

The **AND side** of the graph represents a set of tasks that must be completed to achieve the main goal, while the **OR side** of the graph represents different methods for accomplishing the same main goal.



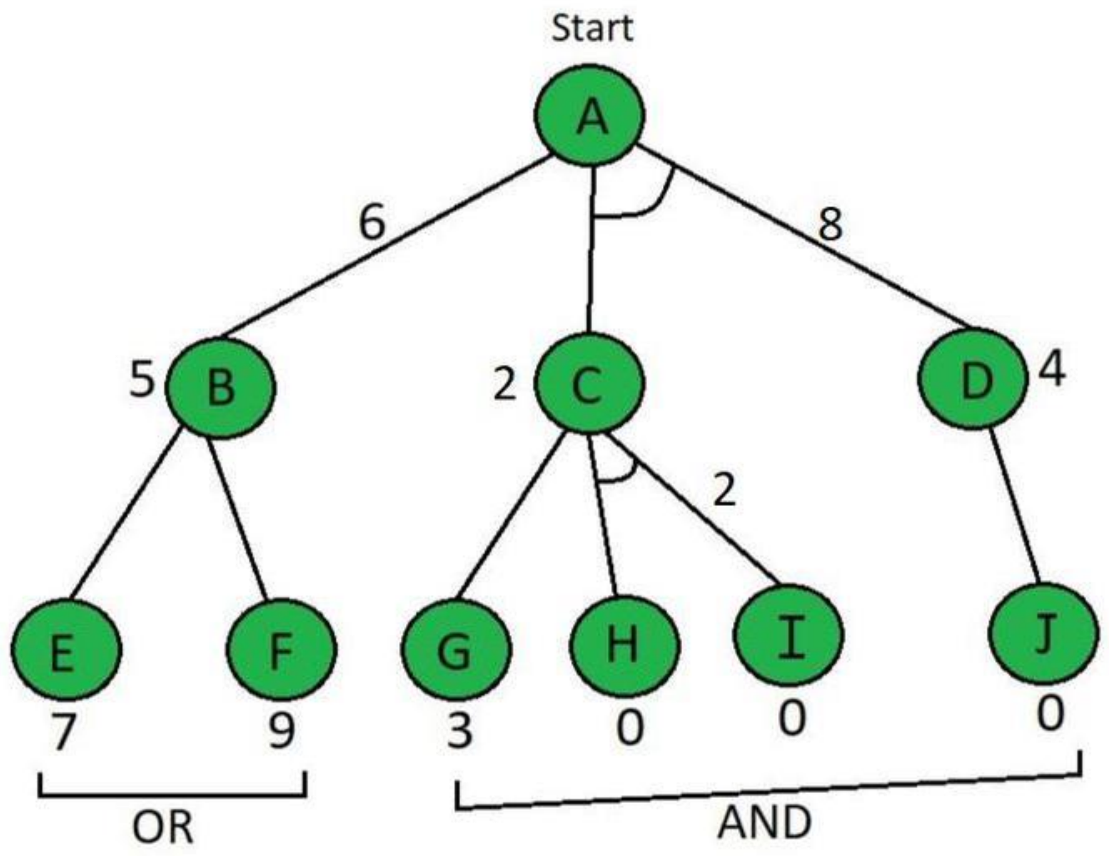
Working of AO* algorithm:

The evaluation function in AO* looks like this:
f(n) = g(n) + h(n)
f(n) = Actual cost + Estimated cost
here,
    f(n) = The actual cost of traversal.
    g(n) = the cost from the initial node to the current node.
    h(n) = estimated cost from the current node to the goal state.

Difference between the A* Algorithm and AO* algorithm

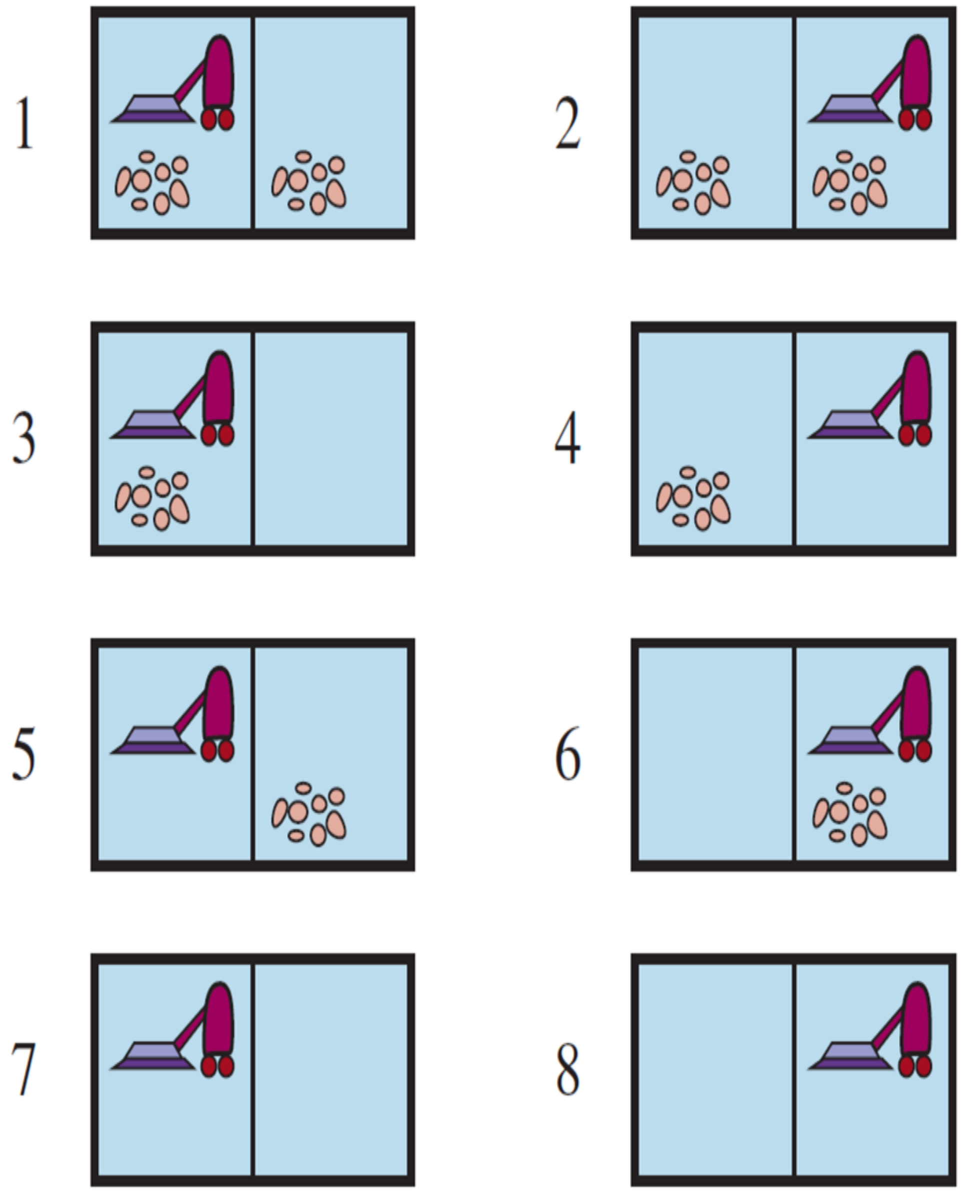
- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- **A*** always **gives** the **optimal solution** but AO* doesn’t guarantee to give the optimal solution.
- Once AO* got a solution **doesn’t explore** 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.**

Here in the example below the Node which is given is the heuristic value i.e **h(n)**. Edge length is considered as **1**.



The Erratic (Unpredictable) Vacuum World

- The vacuum world has eight states, and three actions-Right, Left, and Suck. The goal is to clean up all the dirt (state 7 or 8)



The Erratic (Unpredictable) Vacuum World

- If the environment is fully observable, deterministic, and completely known, then the problem is easy to solve and the solution is an action sequence.
- For example, if the initial state is 1, then the action sequence [Suck, Right, Suck] will reach a goal state, 8.

- In the erratic vacuum world, the environment is nondeterministic, then the Suck action works as follows:

  1. When applied to a dirty square the action cleans the square and sometimes, cleans up dirt in an adjacent square, too.

  2. When applied to a clean square, the action sometimes deposits dirt on the carpet.

The Erratic (Unpredictable) Vacuum World

- To provide a formulation of this problem, we need to generalize the notion of a transition model.
- Here, a RESULTS(s,a) function returns a set of possible outcome states.

- For example, in the erratic vacuum world, the 5 Suck action in state 1 cleans up either just the current location, or both locations:
- RESULTS(1, Suck) = {5,7}

Conditional Plan

- The Erratic Vacuum World, to get solution, the action sequence produces a conditional plan (not sequence plan)
- a conditional plan can contain if-then-else steps.
- [Suck, if State = 5 then [Right, Suck] else [ ]].
- Here, solutions are trees rather than sequences

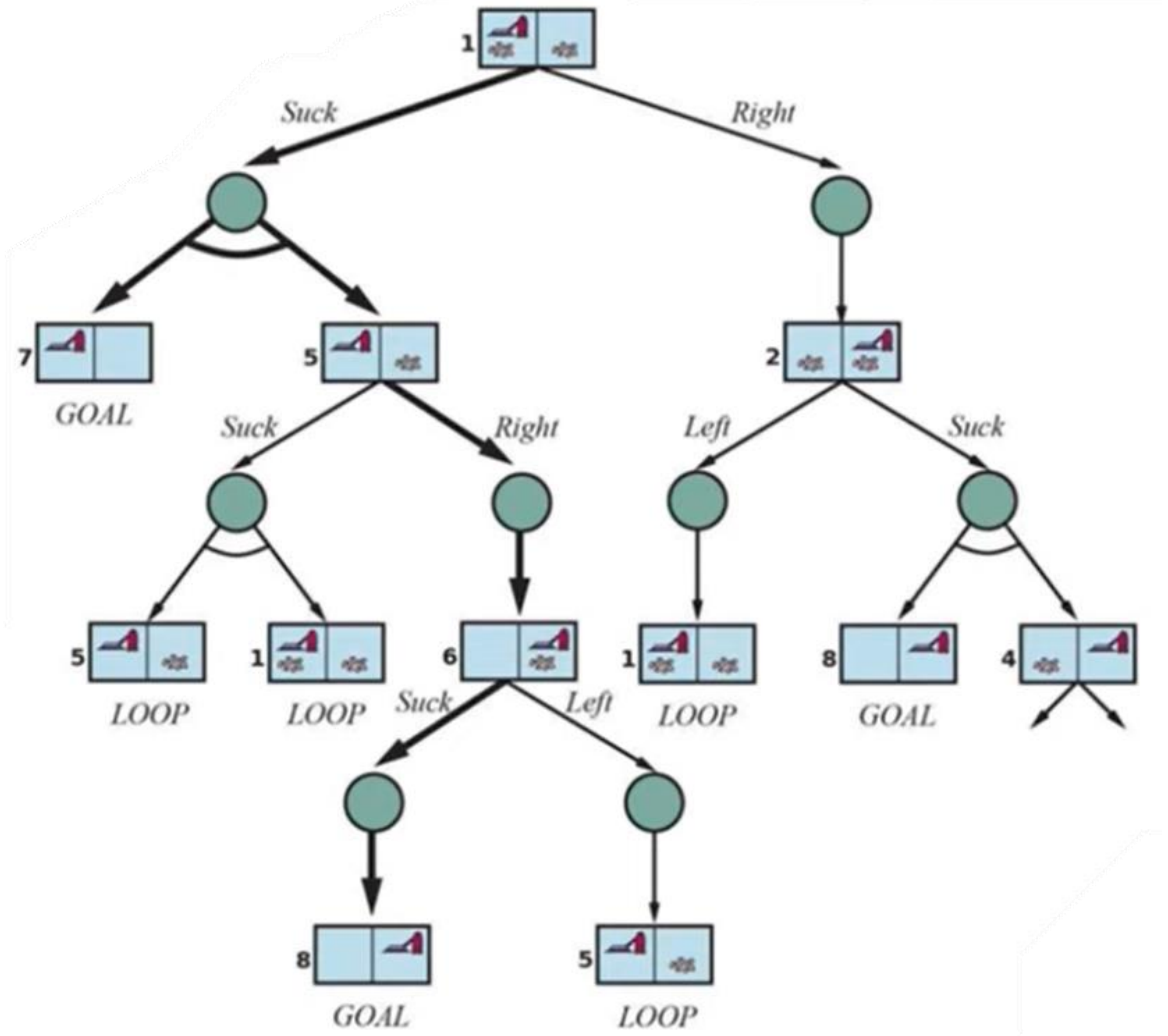
AND-OR Search Trees

- In a deterministic environment, the only branching is introduced by the agent's own choices in each state:
- I can do this action or that action.We call these nodes OR nodes.

- In a nondeterministic environment, branching is also introduced by the environment's choice of outcome for each action.
- We call these nodes AND nodes.

AND - OR tree for Vacuum World

- State nodes are OR nodes where some action must be chosen.
- At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches.
- In the vacuum world, at an OR node the agent chooses Left or Right or Suck.
- the Suck action in state 1 results in the belief state {5,7}, so the agent would need to find a plan for state 5 and for state 7.
- The solution found is shown in bold lines.


1
Suck
Right
7
GOAL
5
2
Suck
Right
Left
Suck
5
LOOP
1
LOOP
6
1
LOOP
8
GOAL
4
Suck
Left
8
GOAL
5
LOOP

A solution for an AND-OR search problem is a subtree of the complete search tree that

(1) has a goal node at every leaf,

(2) specifies one action at each of its OR nodes, and

(3) includes every outcome branch at each of its AND nodes.

(4) The solution should shown in bold lines as in the figure.

Searching with Partial observation

- In such a environment agent -
  1) Knows what is the effect of each action.
  2) Can calculate exactly which state results from any sequence of actions.
  3) Knows which state it is in.
  4) **Does not get new information after each action.**
- If environment is not fully observable and knowledge of the states and actions is incomplete then agent has different types of task environment.
- Such a environment leads to three distinct types of problems -
  1) Sensorless problems
  2) Contingency problems
  3) Exploration problems.

Sensorless Problems (Conformant Problems)

- When the agent's perception provide no information at all, is called a **sensorless problem.**

- In this type of problems -
  A) Agent has **no sensors** at all
  B) According its own knowledge, agent could be in one of several possible initial states.

  C) As agent has several possible initial states each action can lead to one of the several possible successor states.

- The solution to a sensorless problem is a **sequence of actions**, not a conditional plan.

Solving sensorless problems in a fully observable, completely known, deterministic environment

- In sensorless problem agent already have **predefined knowledge** about its various states.
- Out of these state set only one of the state is going to be initial state. Though agent has no sensor, hence no percepts, still from its own state and action it can reach to certain conclusion states, through which it can further reach to goal state.
- It means that we can say agent can coerce (**force to reach**) the world to reach at a particular expected state on his own.

- Steps taken to solve the problem are as follows

  1) First agent searches for **belief state** instead of physical state.
  [The belief state it is a set of states representing agent's current belief about the possible physical states it might be in. For finding such belief state agent should have reasoning for each state it can be in, based on its original knowledge].
  2) Initial state is belief state which can further mapped to another belief state.
  3) Action is applied to belief state by taking union of the results obtained from applying the action to each physical state in the belief state.
  4) We now get a path which connects several belief states.
  5) Solution is a path that leads to a belief state, all of whose members are goal states.

- Note: If the physical state space has 's' states, the belief state space will have 2^s belief states.

Contingency Problem

- If **environment is partially observable or if actions are uncertain**, then agent can sense and perceive new information after each action.
- This each action - this new information can lead to new problem (or critical situation) which is called as contingency. If this situation occurs then agent needs to plan next action to handle this contingency.
- **Solving contingency problem**
  1) For contingency problem one needs to form a tree, here each branch of a tree may be selected depending upon the percepts received up to that point in the tree.
  2) Then by searching and expanding previous level, agent can reach to a goal.
- In some situations for contingency problems we get purely **sequential solutions.**
- For example -
  Agent in game of chess.

Extreme Case of Contingency Problem

- **Exploration problem**: When state as well as the actions of the environment are unknown the agent must act to discover them. The interleaving of searching and execution would be useful to solve exploration problem.
- For example.
  The boat driving robotic agent.

Difference between Offline and Online Search

- The offline search algorithms knows complete solution before taking their first action.
- An offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand.
- Online searching is not searching in computer with internet connection.
- An online search agent first it takes an action, then it observes the environment and computes the next action.
- An online algorithm is one that can process its input piece-by-piece in a serial fashion,
- i.e., in the order that the input is fed to the algorithm, without having the entire input available from the beginning.

Online Search

- In unknown environments, where the agent does not know
  - what states exist, or
  - what its actions do
- the agent must use its actions as experiments in order to learn about the environment.
- Online search is suitable in dynamic or semi-dynamic environments.
- It is helpful in nondeterministic domains.

Example

- A robot placed in an new building.
- It must explore it to build a map that it can use for getting form A to B
- New Born Baby:
- It has many possible actions, but it does not knows the outcomes of them
- It has experienced only a few of the possible states that it can reach.
- The gradual discovery of how the world works is called as an online search process.

- An online search problem is solved by interleaving computation, sensing, and acting.
- The online agent knows the following:

  1. ACTIONS(s), the legal actions in state s;
  2. c(s,a,s'), the cost of applying action a in state s to arrive at state s'.
  3. Is-GOAL(s), the goal test.

- The agent cannot determine RESULT(s, a) except by actually being in states and doing action a.

Online search agents

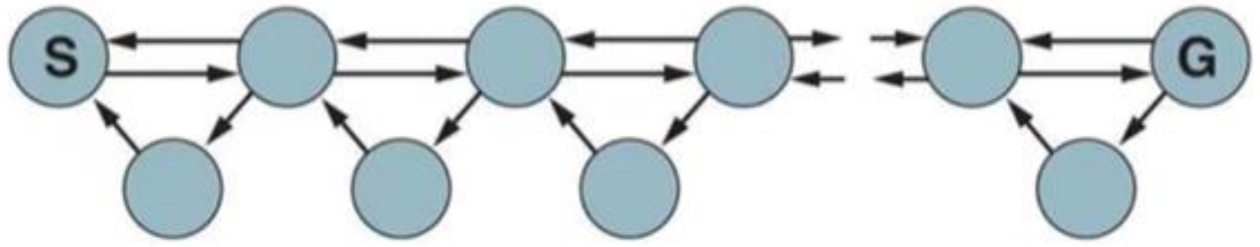
- After each action, an online agent in an observable environment receives a percept telling it what state it has reached;
- from this information, it can expand its map of the environment.
- The updated map is then used to plan where to go next.
- An online algorithm, can discover successors only for a state that it physically occupies.
- The next node expanded is a child of the previous node expanded.

An online depth-first exploration agent

- This agent stores its map in a table, result[s, a], that records the state resulting from executing action a in state s.
- To explore map, The difficulty comes when the agent has tried all the actions in a state.
- To avoid dead end,
- The algorithm keeps another table that lists, for each state, the parent state, to which the agent has not yet backtracked.
- If the agent has run out of states to which it can backtrack, then its search is complete.

- It keeps just one current state in memory, and it can do random walk to explore the environment.
- A random walk simply selects at random one of the available actions from the current state;
- Preference can be given to actions that not yet tried.
- This process will continue until it finds a goal or completes its exploration.
- Advantage
    - It provided that the space is finite and safely explorable.
- Drawback
    - The process can be very slow.

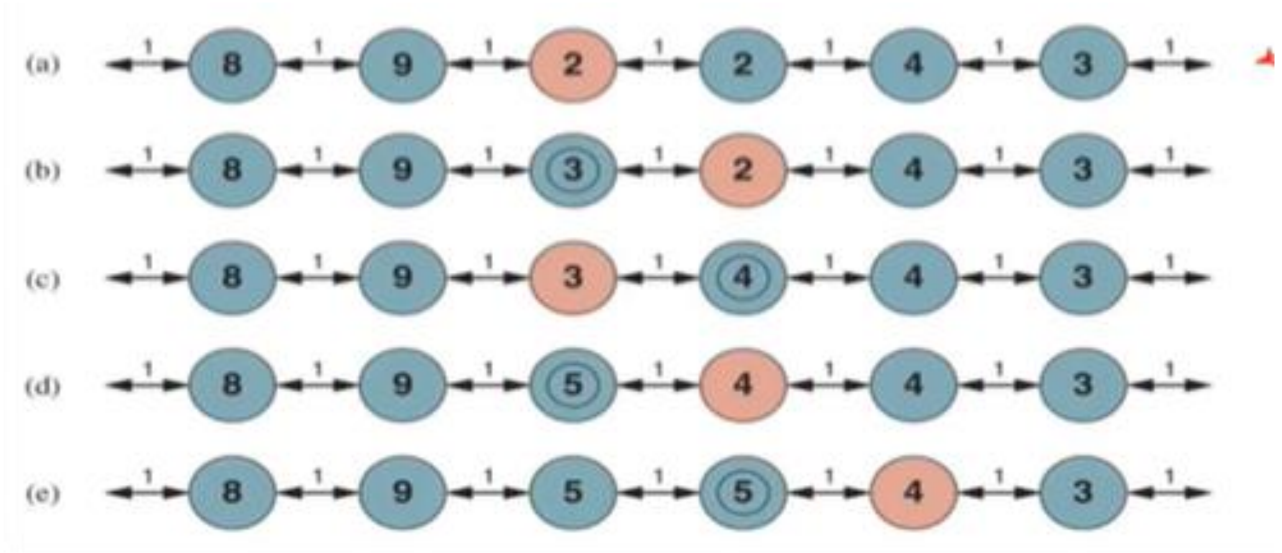
- An environment in which a random walk will take exponentially many steps to find the goal.
- for each state in the top row except S, backward progress is twice as likely as forward progress.



Learning Real-time A* (LRTA*) search Algorithm

- The LRTA* builds a map of the environment in the result table.
- It updates the cost estimate for the state it has just left, and then chooses the "apparently best" move according to its current cost estimates.
- Here h(s) is, The actions that not yet tried in a state, are always assumed to lead immediately to the goal, with the least possible cost.
- This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

- Five iterations of LRTA* on a one-dimensional state space.
- Each state is labeled with h(s), the current cost estimate to reach a goal, and every link has an action cost of 1.
- The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.



Questions

- Describe the problem formulation of Vacuum World problem.
- Explain following terms:

  i) State Space of problem ii) Path in State Space iii) Goal Test
  iv) Path Cost v) Optimal Solution to problem

- Give the outline of Breadth First Search algorithm with respect to Artificial Intelligence.
- With the Local Search algorithm, explain the following concepts;

  1) Shoulder ii) Global Maximum iii) Local Maximum

- Ilustrate Hill Climbing algorithm using 8 queen problem.
- Explain the mechanism of Genetic Algorithm.

Questions

- List and explain performance measuring ways for problem solving.
- Formulate the vacuum world problem.
- Write the uniform cost search algorithm. Explain in short.
- With suitable diagram explain the following concepts

  i. shoulder ii. Global maximum iii. Local maximum
- How genetic algorithm works?
- Explain the working of AND-OR search tree.

Questions

- Discuss in brief the formulation of single state problem.
- Give the outline of Breadth First Search algorithm.
- Give the outline of tree search algorithm.
- Explain the mechanism of genetic algorithm.
- Explain how transition model is used for sensing in vacuum cleaner problem.
- Give the illustration of 8 queen problem using hill climbing algorithm.

Questions

- Write the procedure for tree search.
- Explain the algorithm for breadth first Search.
- Give the outline of Uniform- cost search algorithm.
- Explain A* algorithm for the shortest path.
- Give the outline of Hill climbing algorithm.
- Explain the working mechanism of genetic algorithm.