Search Algorithms

Dr. Subarna Chatterjee

subarna.cs.et@msruas.ac.in

Intended Learning Outcome

- The students should understand the state space representation, and gain familiarity with some common problems formulated as state space search problems.
- The student should be familiar with different search algorithms, and should be able to code the algorithms
  i. greedy search ii. DFS iii. BFS
  iv. uniform cost search v. iterative deepening search vi. bidirectional search
- Given a problem description, the student should be able to formulate it in terms of a state space search problem.
- The student should be able to understand and analyze the properties of these algorithms in terms of
  - time complexity
  - space complexity
  - termination
  - optimality

Cont..

- Be able to apply these search techniques to a given problem whose description is provided.
- The student should understand how implicit state spaces can be unfolded during search.
- Understand how states can be represented by features.
- The student will learn the following strategies for uninformed search:
  - Breadth first search
  - Depth first search
  - Iterative deepening search
  - Bidirectional search

Cont..

- For each of these they will learn
  - The algorithm
  - The time and space complexities
  - When to select a particular strategy
- At the end of this lesson the student should be able to do the following:
  - Analyze a given problem and identify the most suitable search strategy for the problem.
  - Given a problem, apply one of these strategies to find a solution for the problem.

Outline

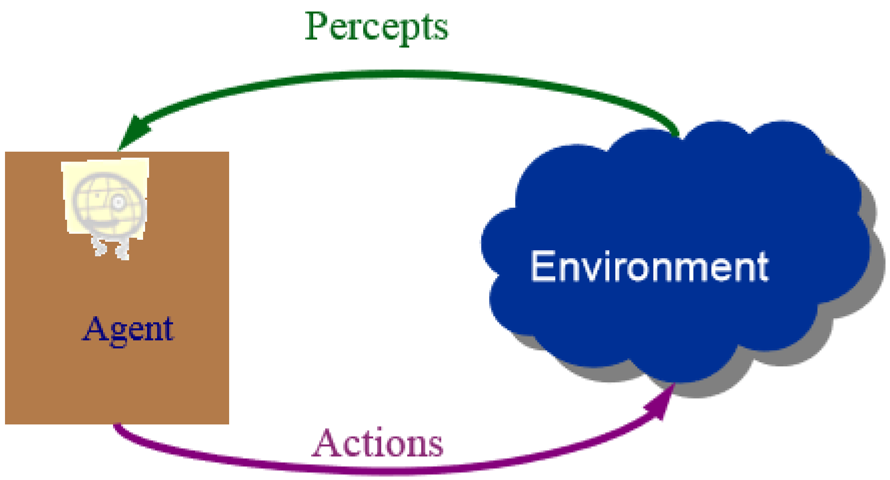
- Problem-solving
- Search Algorithms
- Uninformed Search Algorithm
- Informed Search Algorithms
- Hill Climbing Algorithm
- Means-Ends Analysis

State space search

- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states .
- A state is defined by the specification of the values of all attributes of interest in the world
- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The initial state is where you start
- The goal state is the partial description of the solution

Goal Directed Agent

- A goal directed agent needs to achieve certain goals. Such an agent selects its actions based on the goal it has. Many problems can be represented as a set of states and a set of rules of how one state is transformed to another. Each state is an abstract representation of the agent's environment. It is an abstraction that denotes a configuration of the agent.
- Initial state : The description of the starting configuration of the agent; An action/ operator takes the agent from one state to another state. A state can have a number of successor states. A plan is a sequence of actions.



Cont..

- A goal is a description of a set of desirable states of the world. Goal states are often specified by a goal test which any goal state must satisfy.
- Let us look at a few examples of goal directed agents.

1. 15-puzzle: The goal of an agent working on a 15-puzzle problem may be to reach a configuration which satisfies the condition that the top row has the tiles 1, 2 and 3. The details of this problem will be described later.
2. The goal of an agent may be to navigate a maze and reach the HOME position.

- The agent must choose a sequence of actions to achieve the desired goal.

State Space Search Notations

- An initial state is the description of the starting configuration of the agent
- An action or an operator takes the agent from one state to another state which is called a successor state. A state can have a number of successor states.
- A plan is a sequence of actions.
- The cost of a plan is referred to as the path cost. The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path.

Cont..

- **Problem formulation** means choosing a relevant set of states to consider, and a feasible set of operators for moving from one state to another.
- **Search** is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.
- We are now ready to formally describe a search problem. A search problem consists of the following:
- S: the full set of states
- s_0 : the initial state
- $A:S \rightarrow S$ is a set of operators
- G is the set of final states. Note that $G \subseteq S$
- These are schematically depicted in Figure

Cont..

- **Search Problem :**
- We are now ready to formally describe a search problem.
- A search problem consists of the following:
  - S: the full set of states
  - s_0 : the initial state
  - A: S→S is a set of operators
  - G is the set of final states. Note that G ⊆S
  - These are schematically depicted in Figure

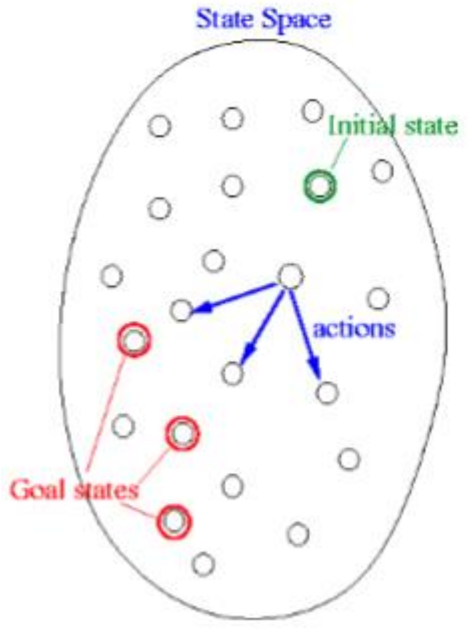


Figure 2

Cont..

- The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g ∈ G.
- A search problem is represented by a 4-tuple {S, s_0, A, G}.
  - S: set of states
  - s_0 ∈ S : initial state
  - A: S → S operators/ actions that transform one state to another state
  - G : goal, a set of states. G ⊆ S

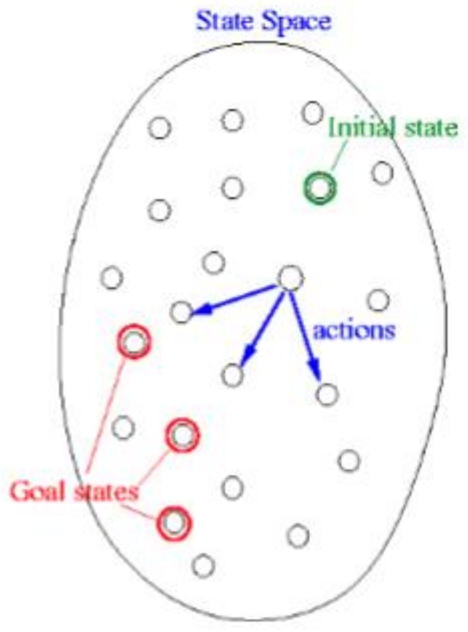


Figure 2

Cont..

- This sequence of actions is called a solution plan.
  - It is a path from the initial state to a goal state.
  - A plan P is a sequence of actions.
- $P = \{a_0, a_1, \ldots, a_N\}$ which leads to traversing a number of states $\{s_0, s_1, \ldots, s_{N+1} \in G\}$.
- A sequence of states is called a path.
  - The cost of a path is a positive number.
  - In many cases the path cost is computed by taking the sum of the costs of each action.

Cont..

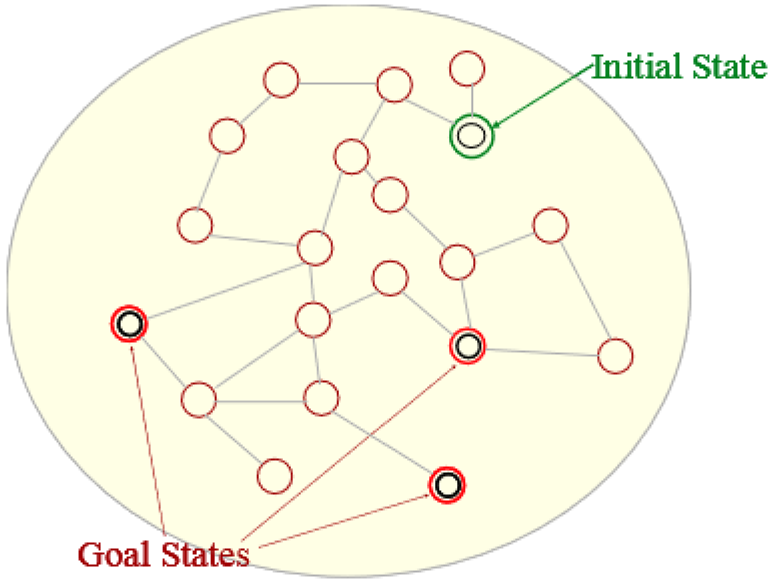
- Representation of search problems
  - A search problem is represented using a directed graph.
  - The states are represented as nodes.
  - The allowed actions are represented as arcs.
- Searching process
- The generic searching process can be very simply described in terms of the following steps:

Do until a solution is found or the state space is exhausted.

1. **Check the current state**
2. **Execute allowable actions to find the successor states.**
3. **Pick one of the new states.**
4. **Check if the new state is a solution state If it is not, the new state becomes the current state and the process is repeated**

Illustration of a search process

- We will now illustrate the searching process with the help of an example. Consider the problem depicted in Figure
  - s_0 is the initial state.
  - The successor states are the adjacent states in the graph.
  - There are three goal states.



Cont..

- The two successor states of the initial state are generated.

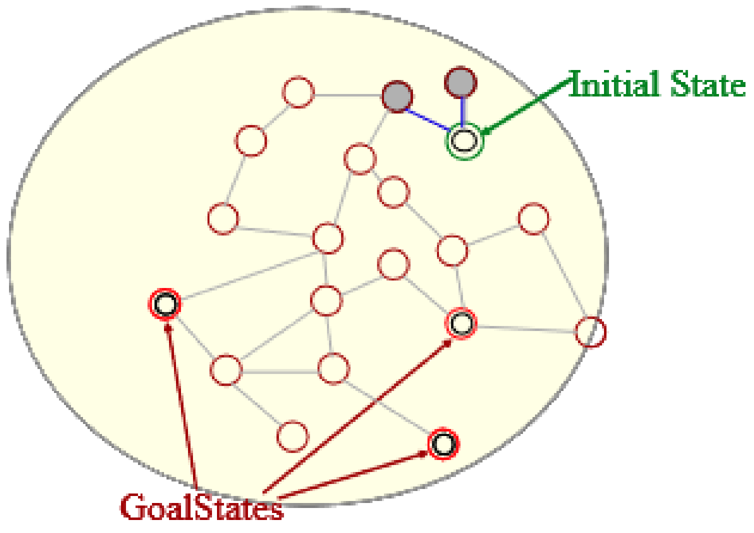


Fig 2

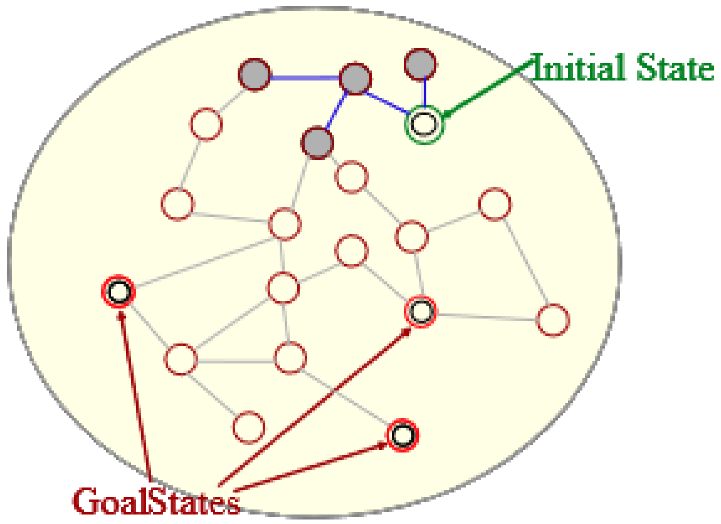


Fig 3

- The successors of these states are picked and their successors are generated.

Cont..

- Successors of all these states are generated.

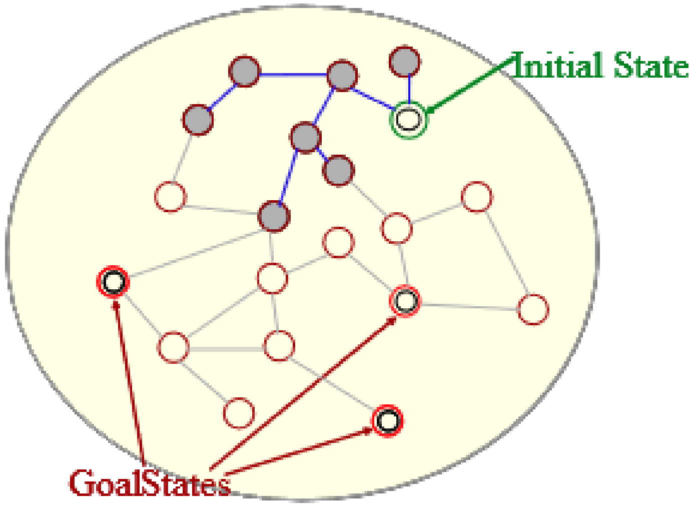


Fig 4

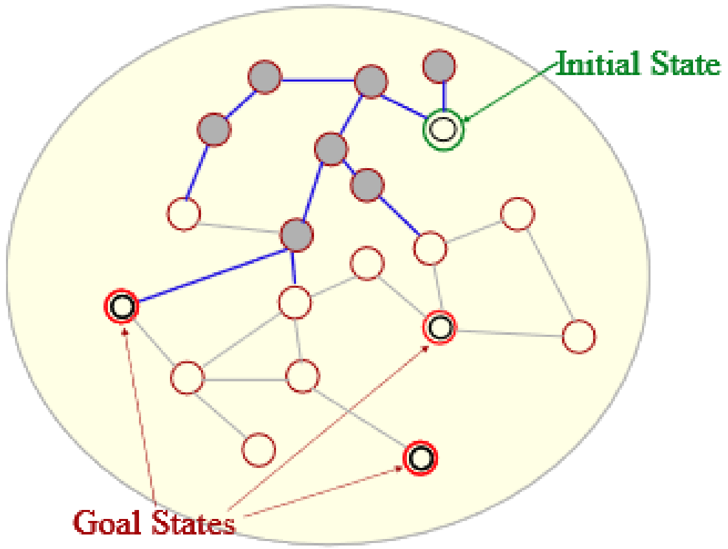
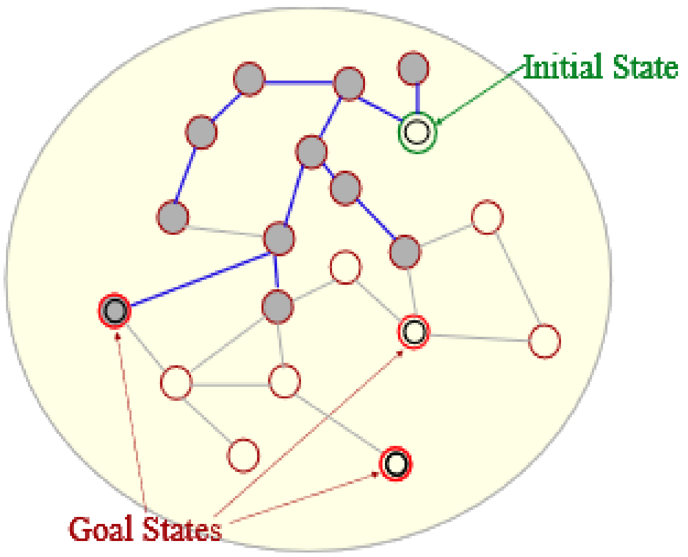


Fig 5

- The successors are generated.

Cont..

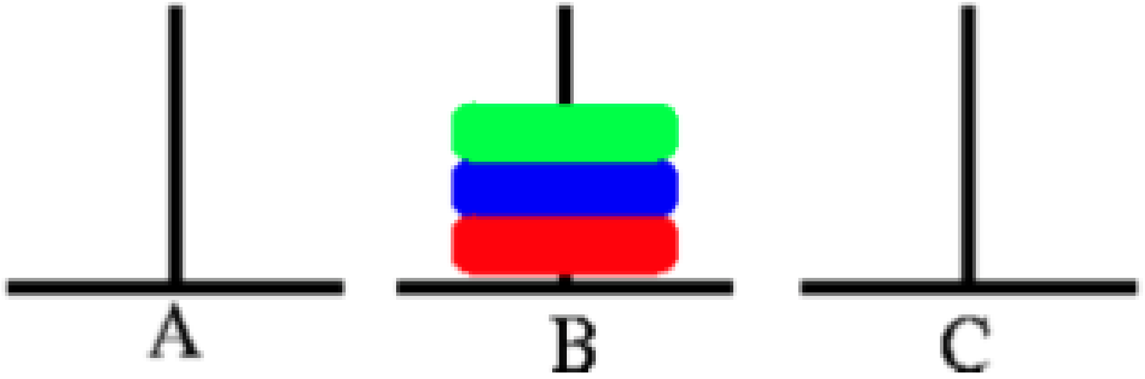
- A goal state has been found.
- The above example illustrates how we can start from a given state and follow the successors, and be able to find solution paths that lead to a goal state. The grey nodes define the search tree. Usually the search tree is extended one node at a time.



- The order in which the search tree is extended depends on the search strategy.
- We will now illustrate state space search with one more. We will illustrate a solution sequence which when applied to the initial state takes us to a goal state.

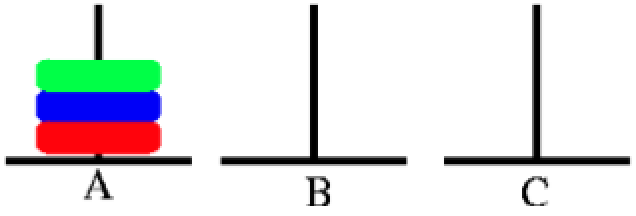
Example : Pegs and Disks problem(Tower of Hanoi)

- Consider the following problem.
- We have 3 pegs and 3 disks.
- **Operators**: one may move the topmost disk on any needle to the topmost position to any other needle
- In the goal state($G \in S$) all the pegs are in the needle B as shown in the figure below.

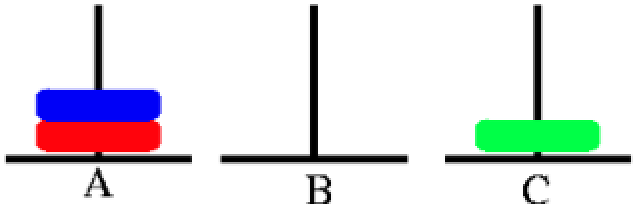


Cont..

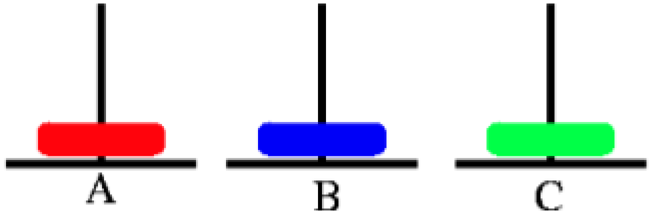
- The initial state($s_0 \in S$) is illustrated below.



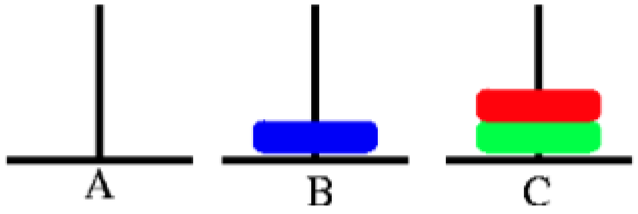
- Now we will describe a sequence of actions that can be applied on the initial state. A: $s_0 \rightarrow \ldots \rightarrow G$
  - Step 1: Move A → C



  - Step 2: Move A → B

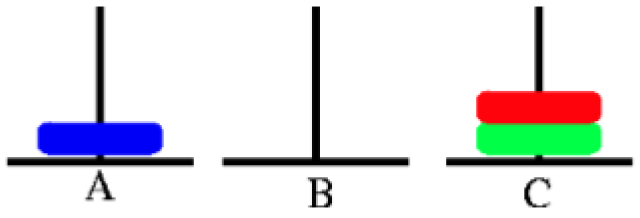


  - Step 3: Move A → C

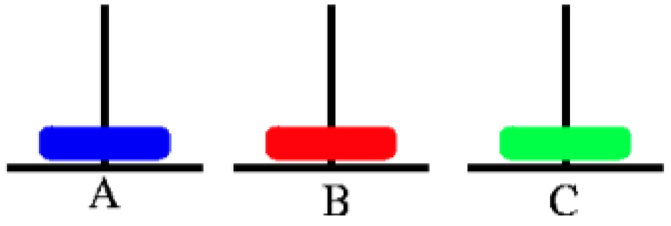


Cont..

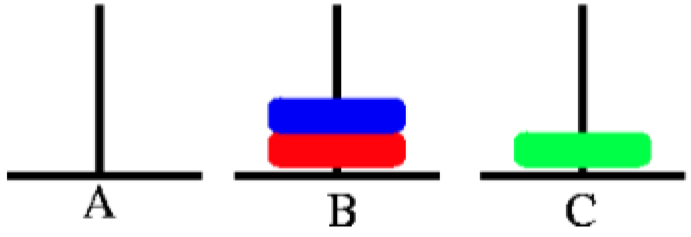
- Step 4: Move B→ A



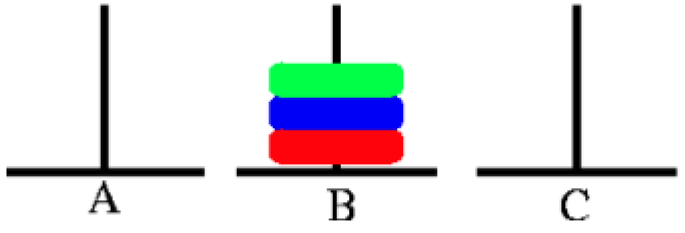
- Step 5: Move C → B



- Step 6: Move A → B



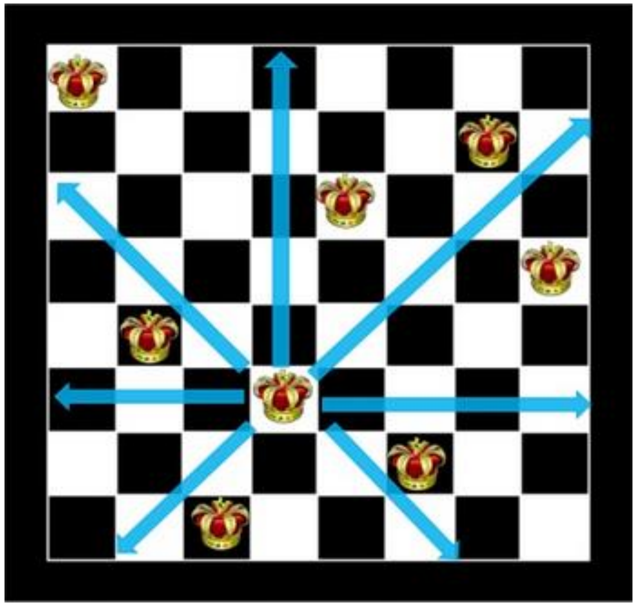
- Step 7: Move C→ B



- 8-queens problem, which can be generalized to the N-queens problem.

Example : 8 queens problem

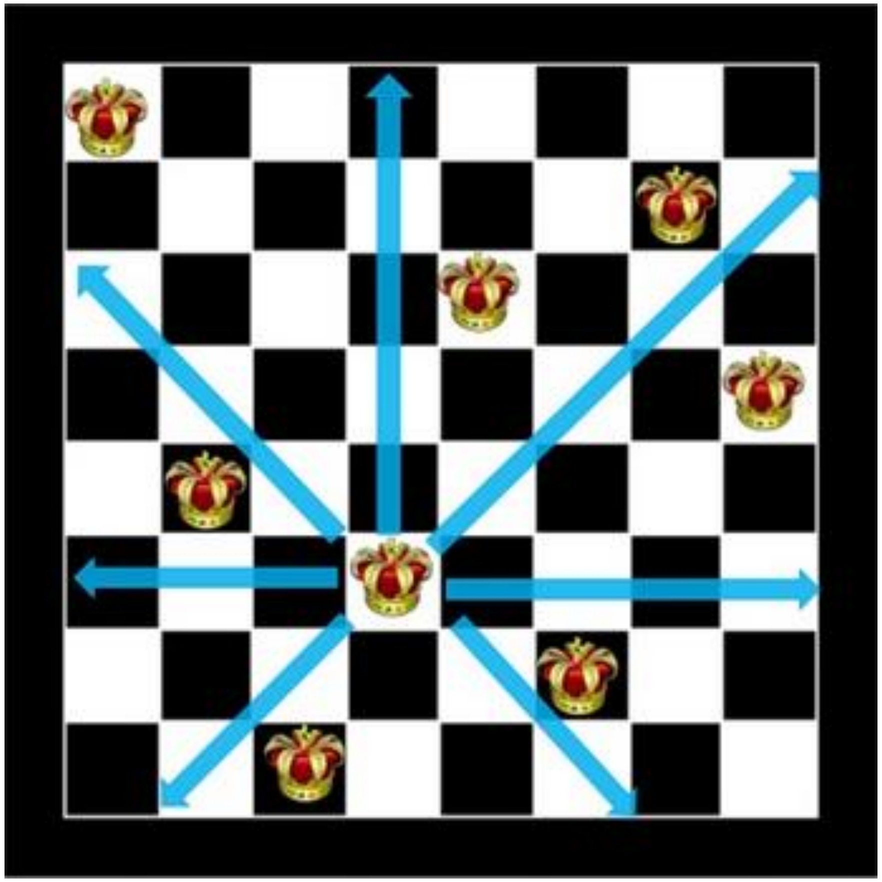
- The problem is to place 8 queens on a chessboard so that no 2 queens are in the same row, column or diagonal
- The picture on the right shows a solution of the 8-queens problem. The picture is a correct solution, because no 2 queens are attacking each other.



- How do we formulate this in terms of a **state space search problem**?
- The problem formulation involves deciding the representation of the **states**, selecting the **initial state** representation, the description of the **operators**, and the **successor states**. We will now show that we can formulate the search problem in several different ways for this problem.

Formulation :

- States: any arrangement of 0 to 8 queens on the board
- Initial state: 0 queens on the board
- Successor function: add a queen in any square
- Goal test: 8 queens on the board, none attacked

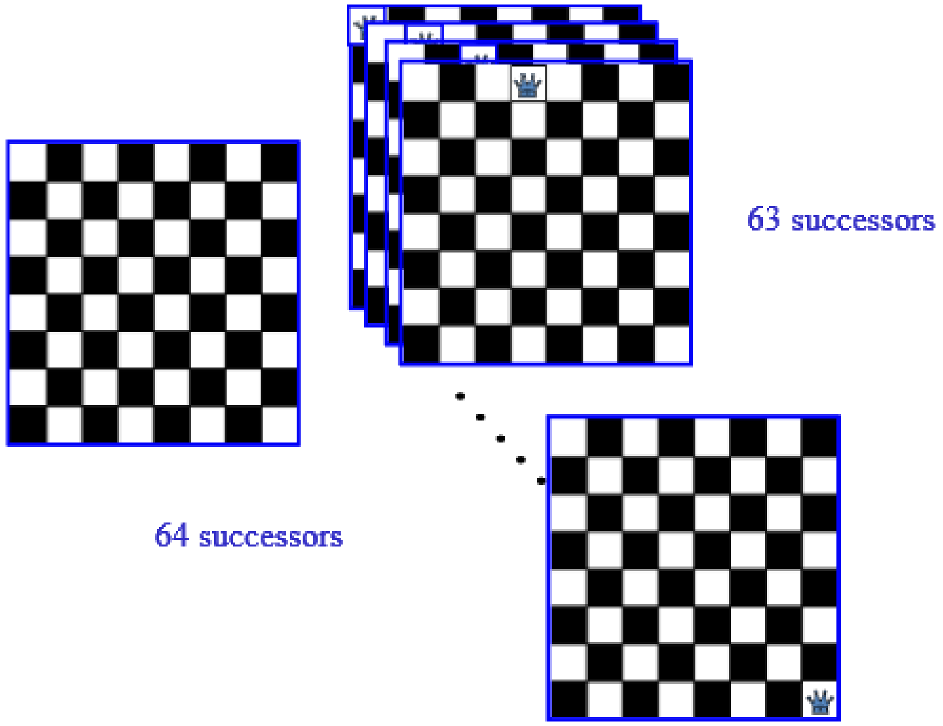


Cont..

- **N queens problem formulation 1**
- **States:** Any arrangement of 0 to 8 queens on the board
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen in any square
- **Goal test:** 8 queens on the board, none are attacked

Cont..

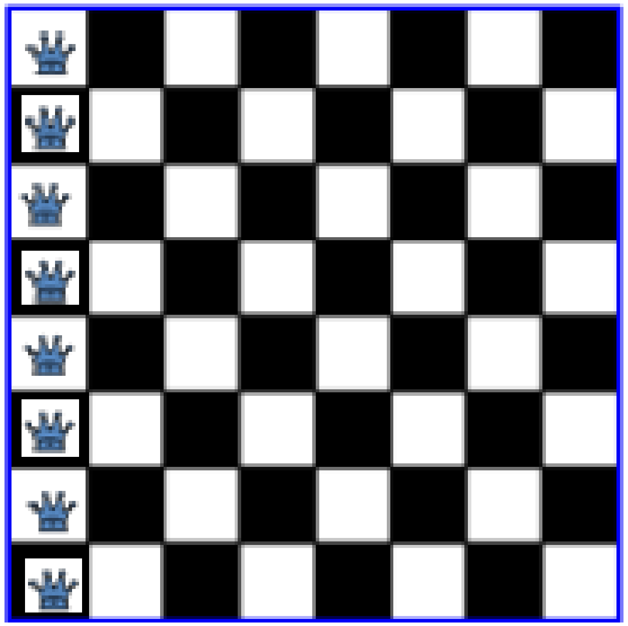
- The initial state has 64 successors.
- Each of the states at the next level have 63 successors, and so on.
- We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other.



- To do that we have to check the new queen against all existing queens on the board.
- The solutions are found at a depth of 8.

Cont..

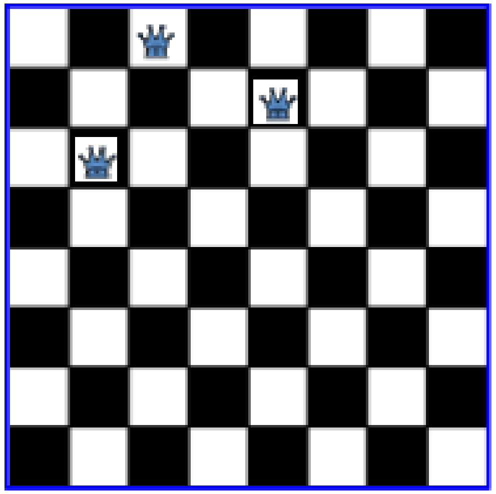
- **N queens problem formulation 2**
- **States:** Any arrangement of 8 queens on the board
- **Initial state:** All queens are at column 1
- **Successor function:** Change the position of any one queen
- **Goal test:** 8 queens on the board, none are attacked

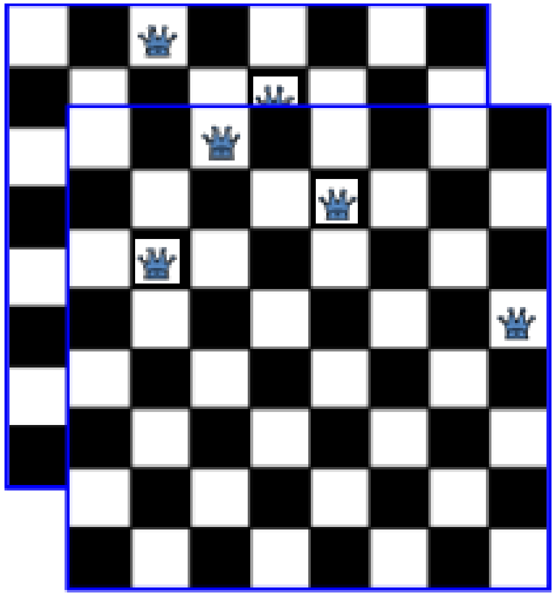


- If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

Cont..

- **N queens problem formulation 3**
- **States:** Any arrangement of k queens in the first k rows such that none are attacked
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen to the $(k+1)^{th}$ row so that none are attacked.





- **Goal test :** 8 queens on the board, none are attacked
- We will now take up yet another search problem, the 8 puzzle.

Example : 8 puzzle

- In the 8-P problem we have a 3×3 square board and 8 numbered tiles. The board has one blank position.
- Bocks can be slid to adjacent blank positions.

5	4	
6	1	8
7	3	2

Initial State

1	4	7
2	5	8
3	6	

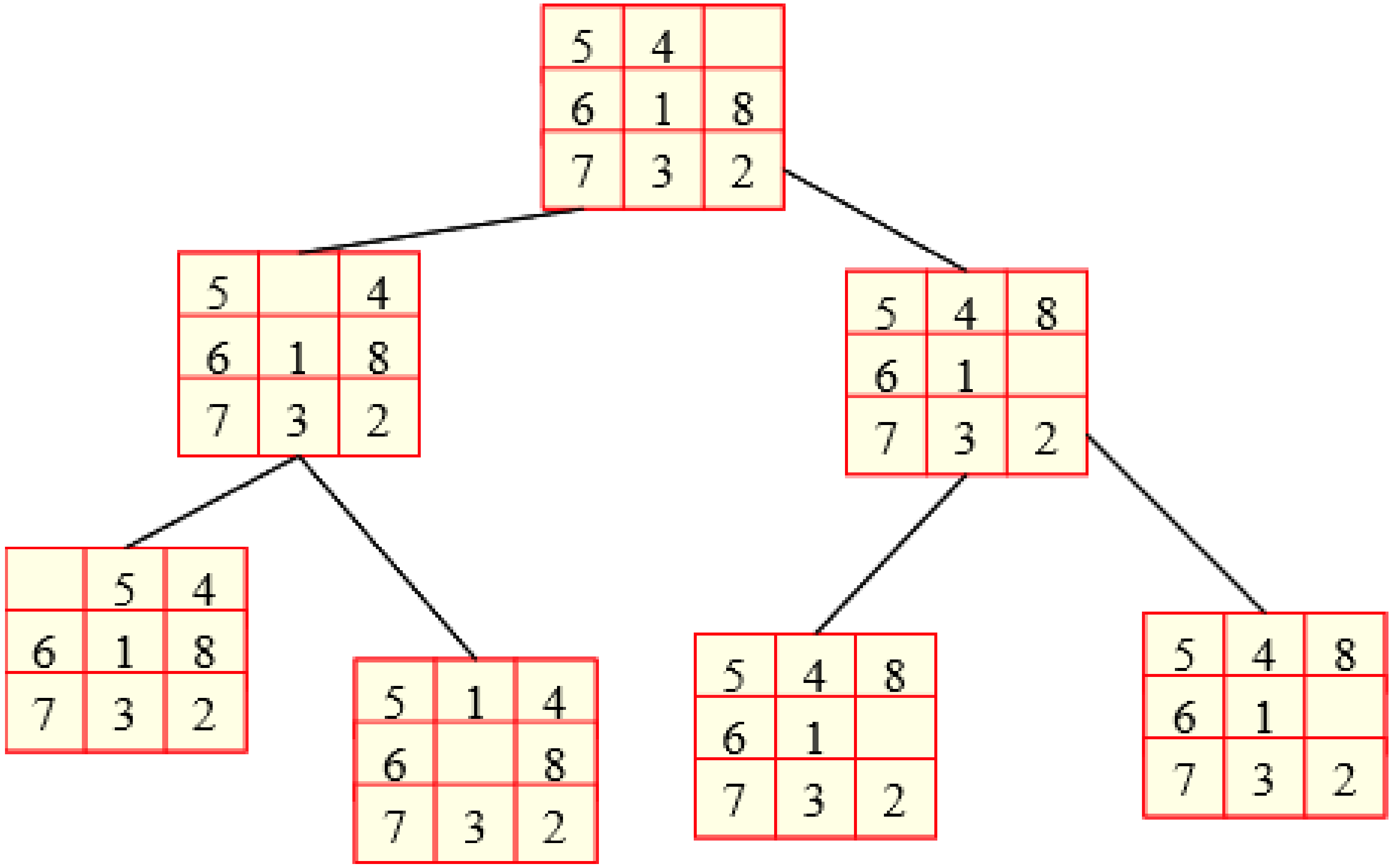
Goal State

- We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right.
- The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.
- The 15-puzzle problems is similar to the 8-puzzle. It has a 4×4 square board and 15 numbered tiles

Cont..

- The state space representation for this problem is summarized below:
- States: A state is a description of each of the eight tiles in each location that it can occupy.
- Operators/Action: The blank moves left, right, up or down
- Goal Test: The current state matches a certain state
- Path Cost: Each move of the blank costs 1
- A small portion of the state space of 8-puzzle is shown below. Note that we do not need to generate all the states before the search begins. The states can be generated when required.

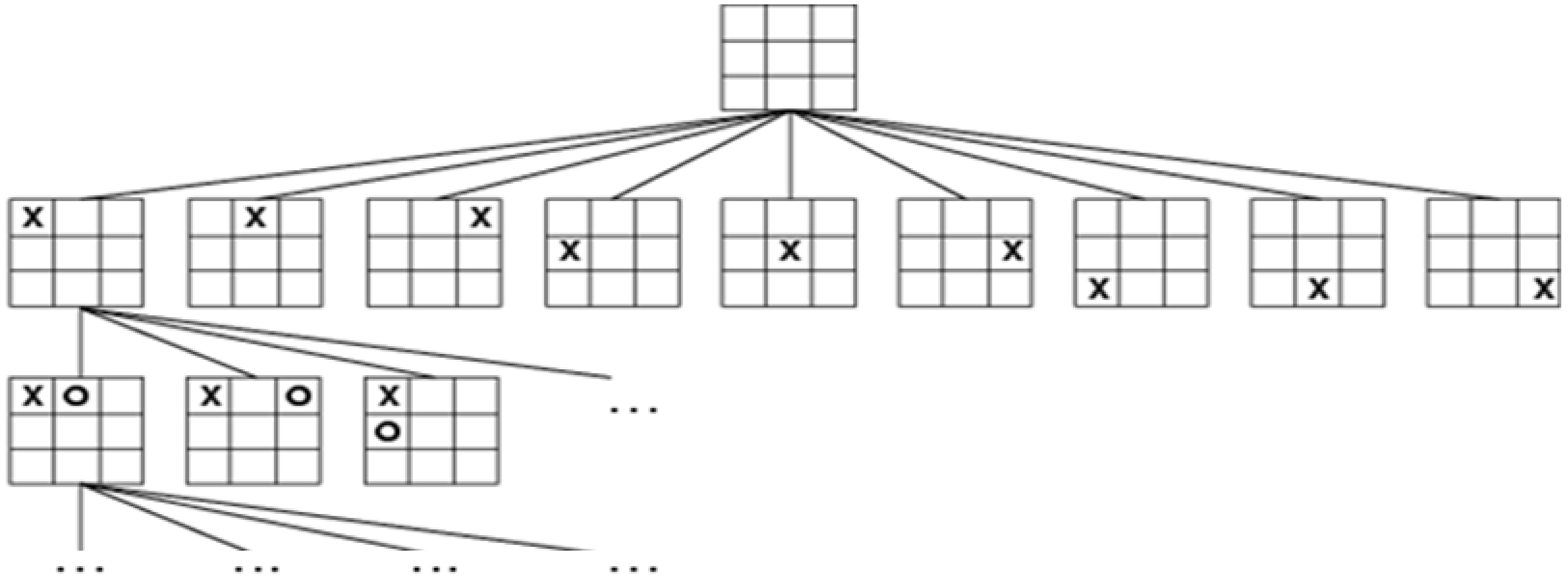
Cont..



8-puzzle partial state space

Example : tic-tac-toe

- Another example we will consider now is the game of tic-tac-toe.
- This is a game that involves two players who play alternately.
- Player one puts a X in an empty position.
- Player 2 places an O in an unoccupied position. The player who can first get three of his symbols in the same row, column or diagonal wins. A portion of the state space of tic-tac-toe is depicted below.



Example : Water jug problem

- You have three jugs measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You need to measure out exactly one gallon.
- Initial state: All three jugs are empty
- Goal test: Some jug contains exactly one gallon.
- Successor function: Applying the
  - action *tranfer* to jugs i and j with capacities C_i and C_j and containing G_i and G_j gallons of water, respectively, leaves jug i with $\max(0, G_i - (C_j - G_j))$ gallons of water and jug j with $\min(C_j, G_i + G_j)$ gallons of water.
  - Applying the action fill to jug i leaves it with C_i gallons of water.
- Cost function: Charge one point for each gallon of water transferred and each gallon of water filled

Explicit vs Implicit state space

- The state space may be explicitly represented by a graph. But more typically the state space can be implicitly represented and generated when required. To generate the state space implicitly, the agent needs to know
  - the initial state
  - The operators and a description of the effects of the operators
- An operator is a function which "expands" a node and computes the successor node(s). In the various formulations of the N-queen's problem, note that if we know the effects of the operators, we do not need to keep an explicit representation of all the possible states, which may be quite large.

Search

- Searching through a state space involves the following:
  - A set of states
  - Operators and their costs
  - Start state
  - A test to check for goal state

- We will now outline the basic search algorithm, and then consider various variations of this algorithm.

Search Algorithms

- **Search algorithms** are one of the most important areas of AI. Here we will explain about the search algorithms in AI.

- **Problem-solving agents:**
  - In AI, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. Here, we will learn various problem-solving search algorithms.

Basic Search Algorithm

- We need to denote the states that are generated as nodes.
- DS of node will keep track of the state, its parent state or the operator, i.e. applied to get this state.
- Additionally, the SA maintains a list of nodes called the ***fringe***, that track the generated nodes & yet to be explored.
- The ***fringe*** represents the frontier of the search tree generated.
- Initially, ***fringe*** contains a single node corresponding to the **start state**. Here we use only the **OPEN** list or **fringe**.
- Algorithm always picks the 1st node from fringe for expansion.

Basic Search Algorithm

- Let L be a list containing the initial state (L = the **fringe**)
- Loop
  - if L is empty return failure
  - Node ← select (L)
  - if Node is a goal
    - then return Node (the path from initial state to Node)
  - else generate all successors of Node, and
    - merge the newly generated states into L
- End Loop

Cont..

- If the node is goal state, the path is returned.
- The path corresponding to a goal node can be found by following the parent nodes.
- Otherwise all successor nodes are generated and they are added to the fringe.
- The successors of the current expanded node are put in fringe. It will determine the property of the search algorithm.

Search algorithm: Key issues

- Corresponding to a ***SA***, we get a ***ST*** which contains the generated and the explored nodes.
  - The *ST* may be unbounded.
  - This may happen if the state space is ∞.
  - This can also happen if there are loops in the search space. How can we handle loops?
- Corresponding to a SA, should we return a path or a node?
  - The answer to this depends on the problem.
  - For problems like **N-queens** we are only interested in the **goal state**.
  - For problems like **15-puzzle**, we are interested in the **solution path**.
- We see that in the basic SA, we have to select a node for expansion.
- Which node should we select?

Cont..

- Alternatively, how would we place the newly generated nodes in the fringe?
- Depending on the search problem, we will have different cases.
  - The search graph may be weighted or unweighted.
  - In some cases we may have some knowledge about the quality of intermediate states and this can perhaps be exploited by the SA.
  - Also depending on the problem, our aim may be to find a minimal cost path or any to find path as soon as possible.

Cont..

- **Which path to find?**
- The **objective** of a search problem is to find a path from the initial state to a goal state.
- If there are several paths which path should be chosen?
- Our objective could be
  - to find any path, or
  - we may need to find the shortest path or
  - least cost path.

Evaluating Search strategies

- We will look at various search strategies and evaluate their problem solving performance. What are the characteristics of the different search algorithms and what is their efficiency? We will look at the following three factors to measure this.

  1. **Completeness:** Is the strategy guaranteed to find a solution if one exists?
  2. **Optimality:** Does the solution have low cost or the minimal cost?
  3. What is the **search cost** associated with the **time** and **memory** required to find a solution?
     a. **Time complexity:** Time taken (number of nodes expanded) (worst or average case) to find a solution.
     b. **Space complexity:** Space used by the algorithm measured in terms of the maximum size of fringe

Search Algorithm Terminologies

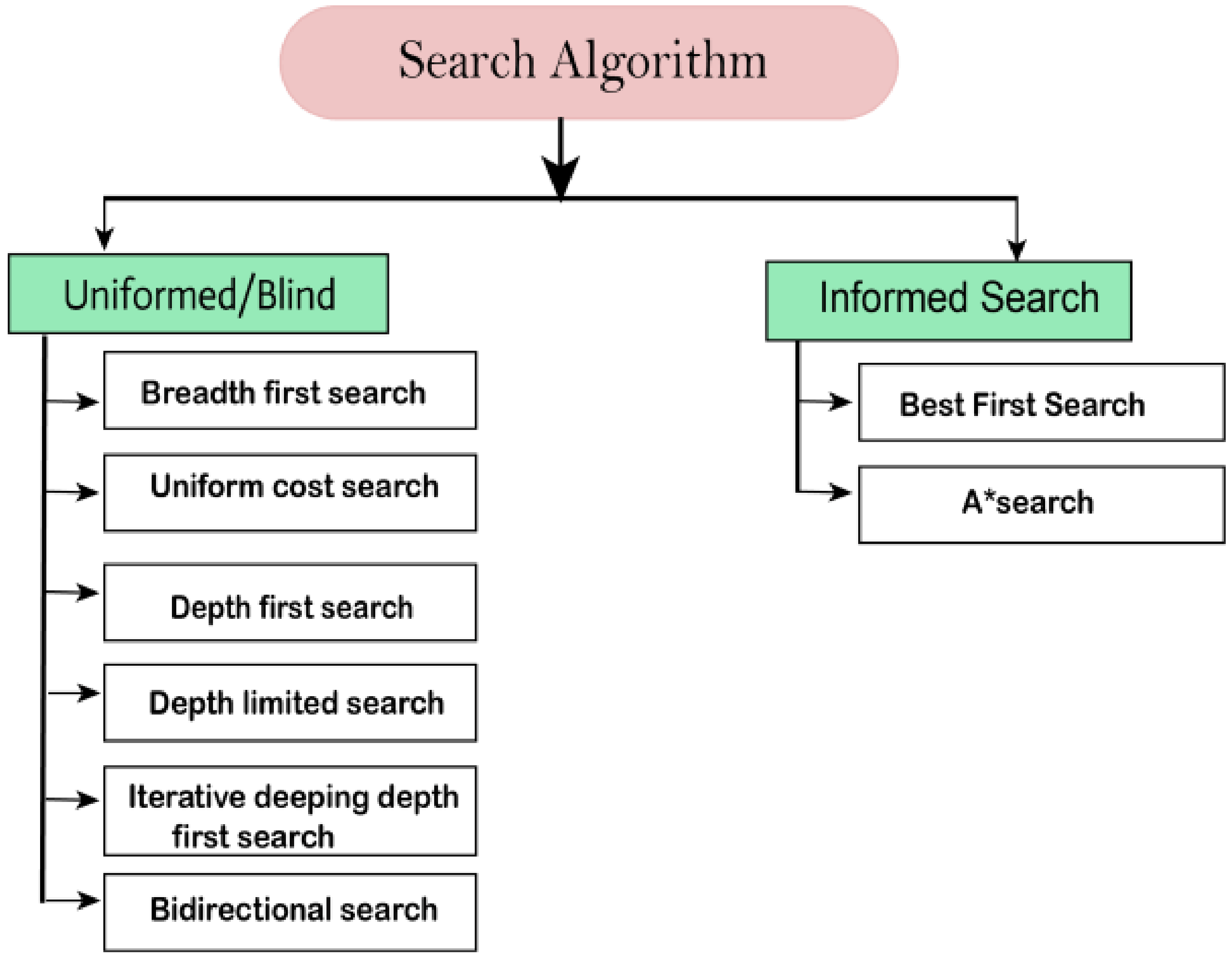
- **Search:** Searching is a step by step procedure to solve a search-problem in a given **search space**. A search problem can have 3 main factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent begins **the search**.
  - **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions of the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms

- Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:
  - **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
  - **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
  - **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
  - **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

- Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Cont..

- The different search strategies that we will consider include the following:
  1. Blind Search strategies or Uninformed search
     a. Depth first search
     b. Breadth first search
     c. Iterative deepening search
     d. Iterative broadening search
  2. Informed Search
  3. Constraint Satisfaction Search
  4. Adversary Search

Uninformed/Blind Search

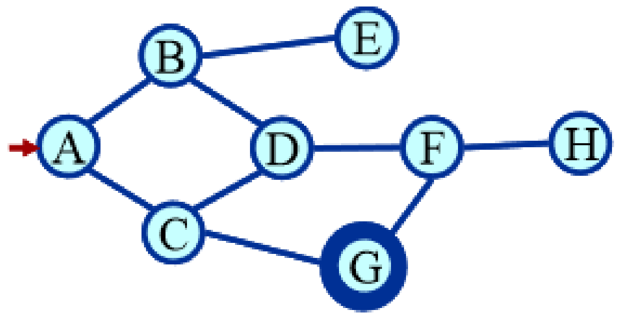
- It does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- It applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, only how to traverse the tree is available, so it is also called blind search.
- It examines each node of the tree until it achieves the goal node.
- 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

Cont..

- **Blind Search** : we will talk about blind search or uninformed search that does not use any extra information about the problem domain.
- The two common methods of blind search are:
  - BFS or Breadth First Search
  - DFS or Depth First Search
- It can be divided into five main types:
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Iterative deepening depth-first search
  - Bidirectional Search

Cont..

- **Search Tree:** Consider the state space graph shown in the figure.



- One may list all possible paths, eliminating cycles from the paths, and we would get the complete search tree from a state space graph. Let us examine certain terminology associated with a ST.
- A ST is a DS containing a root node, from where the search starts.
- Every node may have 0 or more children. If a node X is a child of node Y, node Y is said to be a parent of node X.

Search Tree – Terminology

- **Root Node:** The node from which the search starts.
- **Leaf Node:** A node in the search tree having no children.
- **Ancestor/Descendant:** X is an ancestor of Y is either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- **Branching factor:** the maximum number of children of a non-leaf node in the search tree
- **Path:** A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.
- We also need to introduce some DS that will be used in the search algorithms.

Node data structure

- A node used in the search algorithm is a data structure which contains the following:
    1. A state description
    2. A pointer to the parent of the node
    3. Depth of the node
    4. The operator that generated this node
    5. Cost of this path (sum of operator costs) from the start state
- The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe.
- Initially only the start node is in OPEN

Cont..

- The search starts with the **root node**. The algorithm picks a node from **OPEN** for expanding and generates all the children of the node. Expanding a node from OPEN results in a closed node.
- Some search algorithms keep track of the closed nodes in a DS called **CLOSED**.
- A solution to the search problem is a sequence of operators that is associated with a path from a **start node** to a **goal node**.
- The **cost of a solution** is the sum of the arc costs on the solution path.
- For **large state spaces**, it is not practical to represent the whole space. State space search makes explicit a sufficient portion of an implicit state space graph to find a goal node.
- Each node represents a **partial solution path** from the **start node** to the **given node**. In general, from this node there are many possible paths that have this partial path as a prefix.

Cont..

- The search process constructs a search tree, where
  - **root** is the initial state and
  - **leaf nodes** are nodes
    - not yet expanded (i.e., in fringe) or
    - having no successors (i.e., “dead-ends”)
- ST may be ∞ because of loops even if state space is small. Search problem will return as a solution a path to a goal node.
- Finding a path is important in problems like path finding, 15-puzzle, etc.
- There are also problems like the N-queens problem for which the path to the solution is not important.
- For such problems the search problem needs to return the goal state only.

Breadth First Search Algorithm

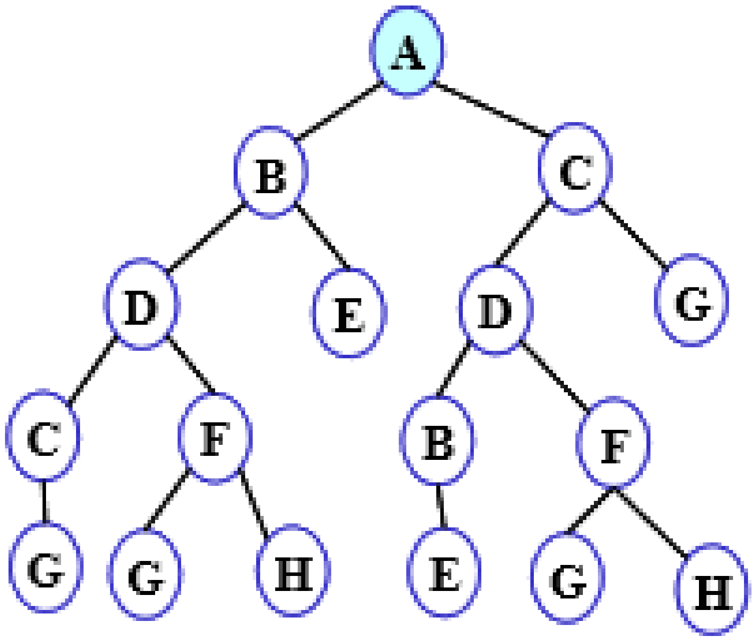
- Let fringe be a list containing the initial state
- **Loop**
  - if *fringe* is empty return failure
  - Node ← remove-first (fringe)
  - if Node is a goal
    - then return the path from initial state to Node
  - else generate all successors of Node, and
    - (merge the newly generated nodes into fringe)

      add generated nodes to the **back of fringe**
- **End Loop**

BFS, the newly generated nodes are put at the **back** of **fringe / OPEN** list. Nodes will be expanded in a **FIFO** order. Node that enters **OPEN** earlier will be expanded earlier.

Cont..

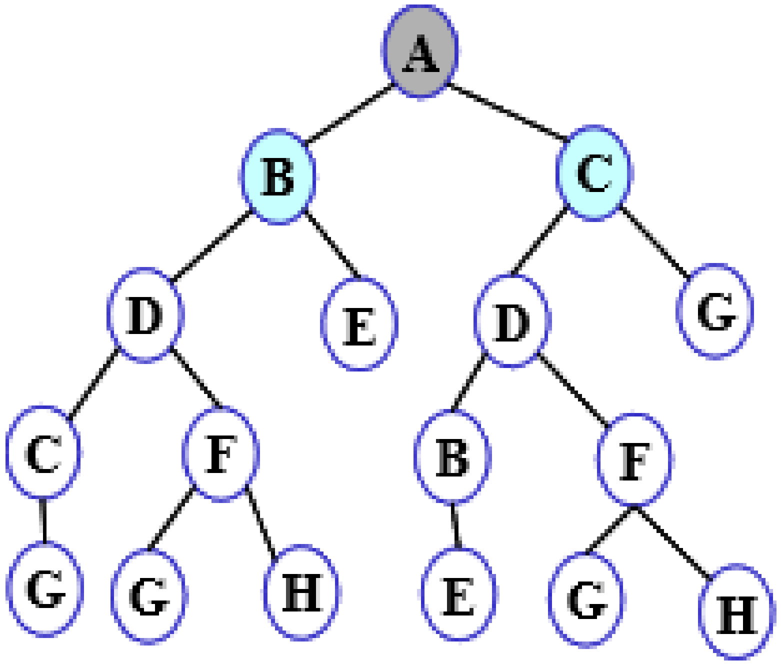
- We will now consider the search space in slide 49, and show how breadth first search works on this graph.
- Step 1: Initially fringe contains only one node corresponding to the source state A.



- FRINGE: A

Cont..

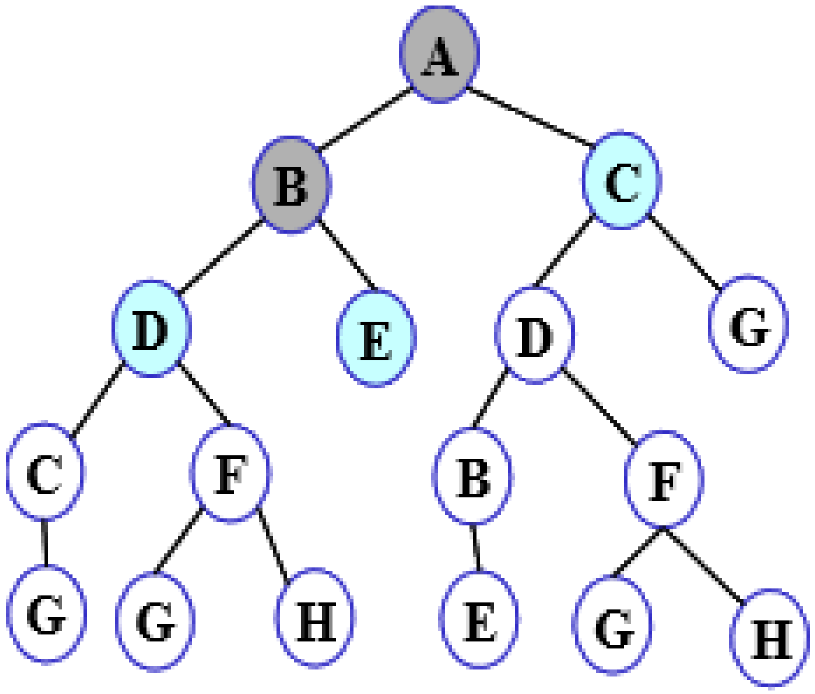
- Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.



- FRINGE: B,C

Cont..

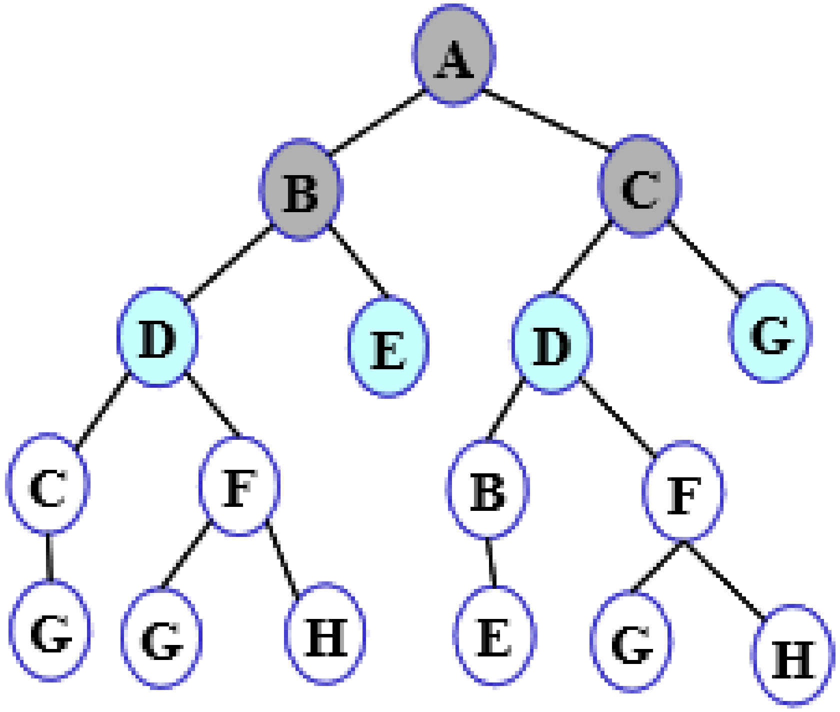
- Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.



- FRINGE: C, D, E

Cont..

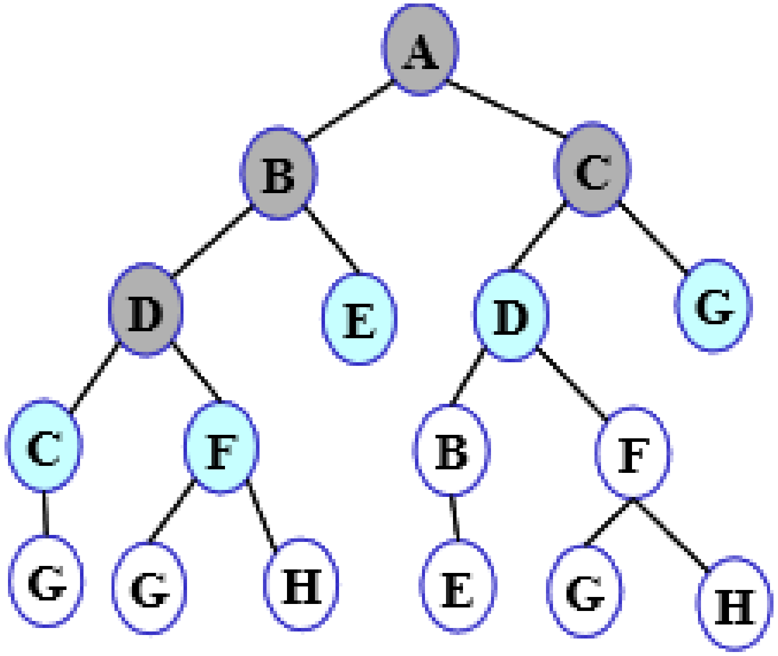
- Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.



- FRINGE: D, E, D, G

Cont..

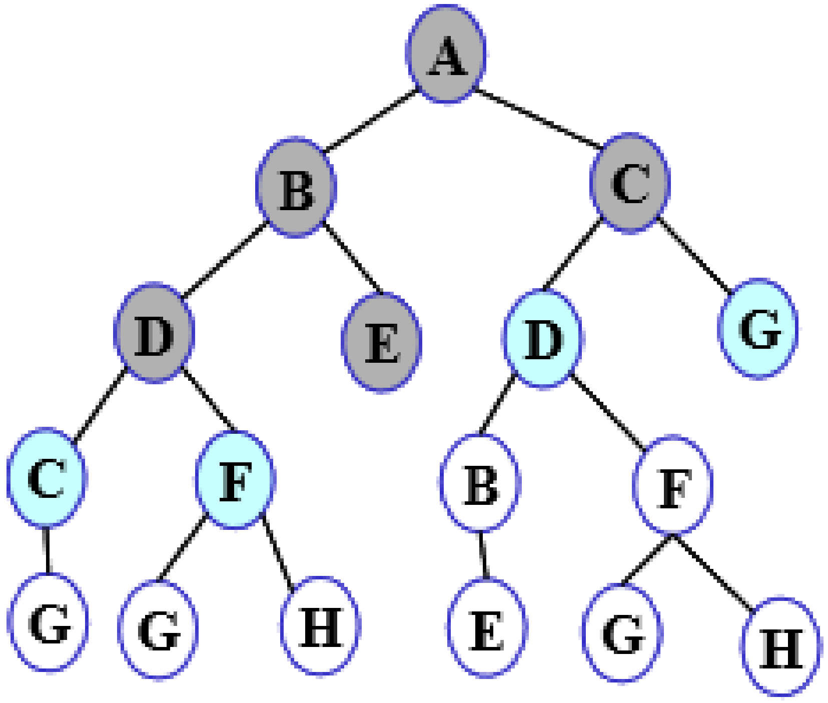
- Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.



- FRINGE: E, D, G, C, F

Cont..

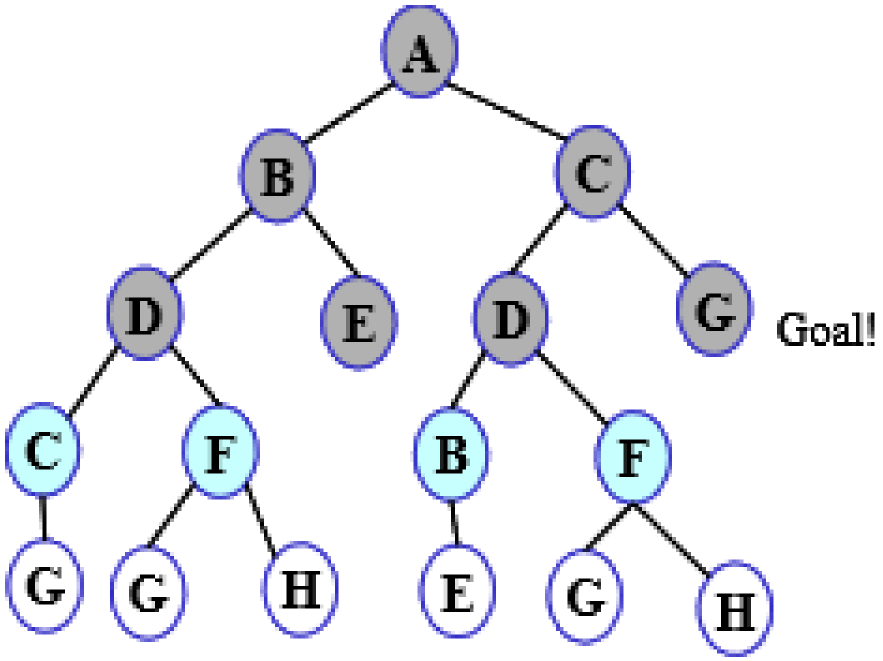
- Step 6: Node E is removed from fringe. It has no children.



- FRINGE: D, G, C, F

Cont..

- Step 7: D is expanded, B and F are put in OPEN or fringe.

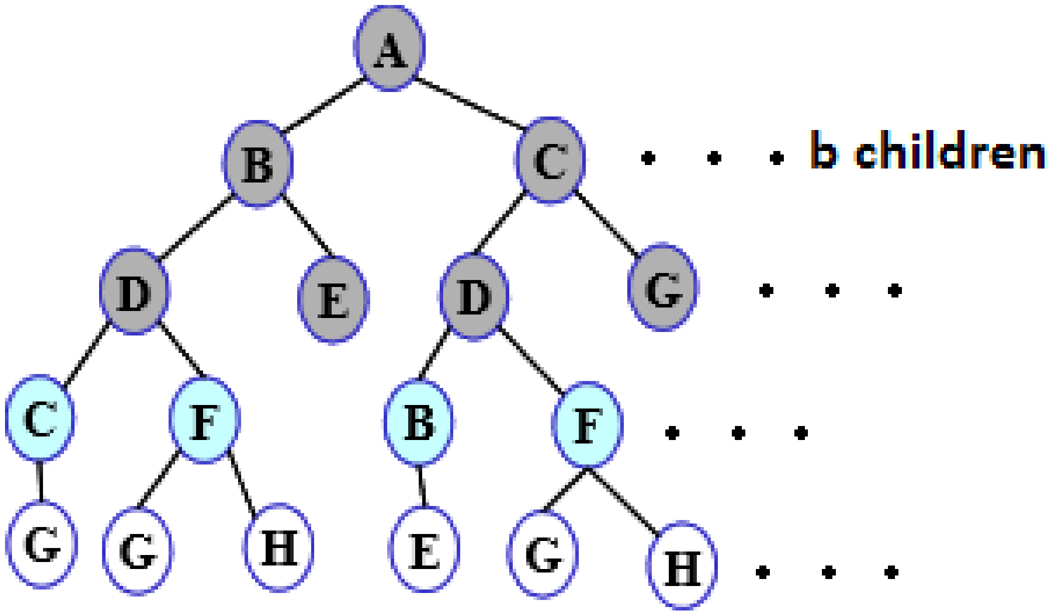


- FRINGE: G, C, F, B, F

- Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm **terminates**.

Properties of Breadth-First Search

- Let us consider a model of the search tree as shown in slide 55.
- Assume every non-leaf node has b children.
- Suppose that **d** is the depth **o** the shallowest goal node, and **m** is the depth of the node found 1st.



Properties of Breadth-First Search

- Complete.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, BFS finds a solution with the shortest path length.
- The algorithm has exponential time and space complexity.
- Suppose the search tree can be modeled as a b-ary tree as shown in slide 55. Then the time and space complexity of the algorithm is O(bd) where **d** is the depth of the solution and **b** is the branching factor (i.e., number of children) at each node.
- A complete search tree of depth **d** where each non-leaf node has **b** children, has a total of

$$1 + b + b^2 + \cdots b^d = {}^{(b^{(d+1)}+1)}/_{(b-1)} \quad \text{nodes}$$

Cont..

- Consider a complete search tree of depth 15, where every node at depths 0 to 14 has 10 children and every node at depth 15 is a leaf node.
- The complete search tree in this case will have $O(10^{15})$ nodes.
- If BFS expands 10000 nodes / sec and each node uses 100 bytes of storage, then BFS will take 3500 years to run in the worst case, and it will use 11100 terabytes of memory.
- So you can see that the BFS algorithm cannot be effectively used unless the search space is quite small.
- You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.
- **Advantages:** Finds the path of minimal length to the goal.
- **Disadvantages :** Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node

Breadth-first Search

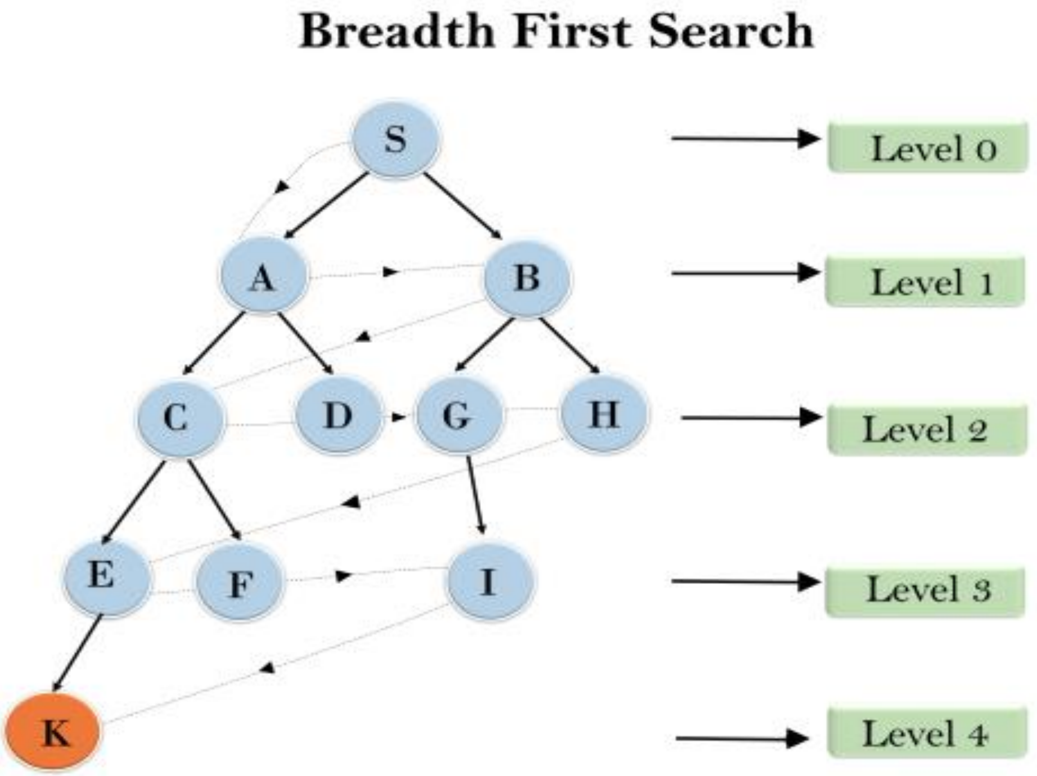
1. Breadth-first Search

- It 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.
- This algorithm is an example of a general-graph search algorithm.
- This is 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.

Cont..

- **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.
- **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:



$$S \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow G \rightarrow H \rightarrow E \rightarrow F \rightarrow I \rightarrow K$$

Cont..

- **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.

$$\boldsymbol{T\ (b)\ =\ 1 + b^2 + b^3 + \cdots \ldots . + b^d =\ O\ (b^d)}$$

- **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.

Depth-first Search Algorithm

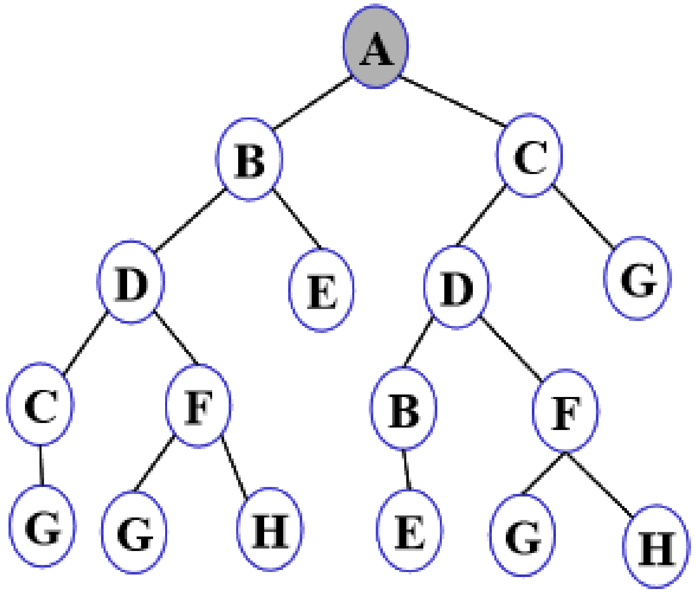
- Let fringe be a list containing the initial state
- **Loop**
  - if *fringe* is empty return failure
  - Node ← remove-first (fringe)
  - if Node is a goal
    - then return the path from initial state to Node
  - else generate all successors of Node, and
    - merge the newly generated nodes into fringe

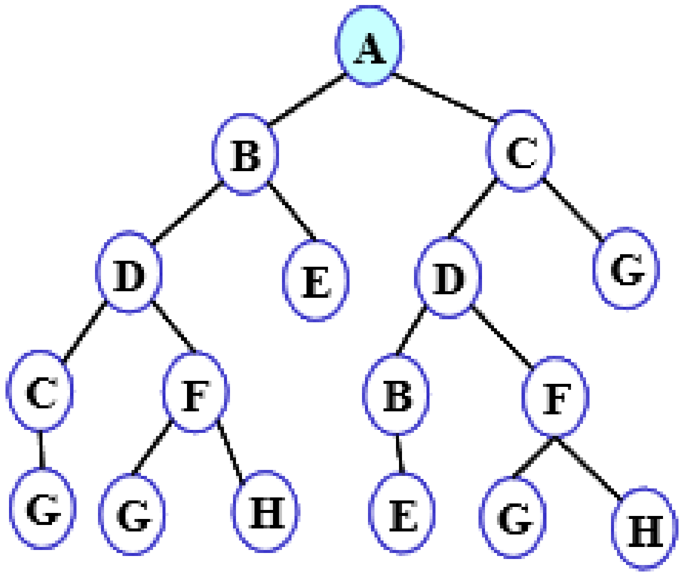
      add generated nodes to the **front of fringe**
- **End Loop**

The DFS algorithm puts newly generated nodes in the **front** of **OPEN**. This results expands the deepest node 1st. Thus the nodes in **OPEN** follow a **LIFO** order. **OPEN** is implemented using a **stack** DS.

Cont..

- Let us now run Depth First Search on the search space given in Figure, and trace its progress.

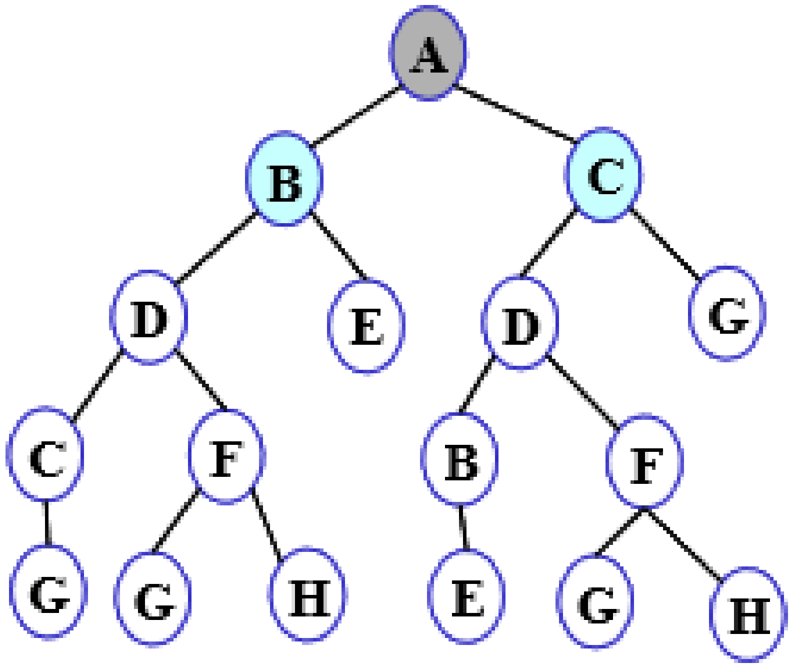




- Step 1: Initially fringe contains only the node for A.
- FRINGE: **A**

Cont..

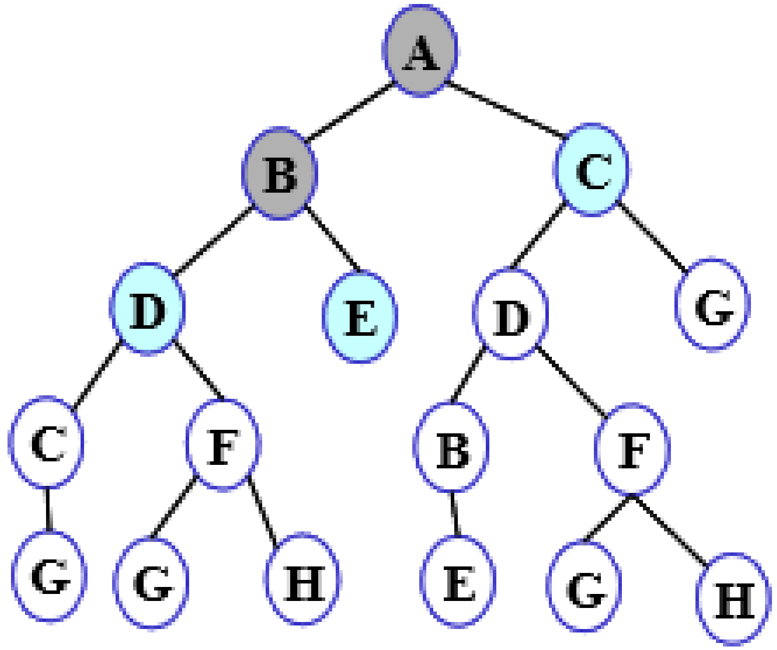
- Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



- FRINGE: **B, C**

Cont..

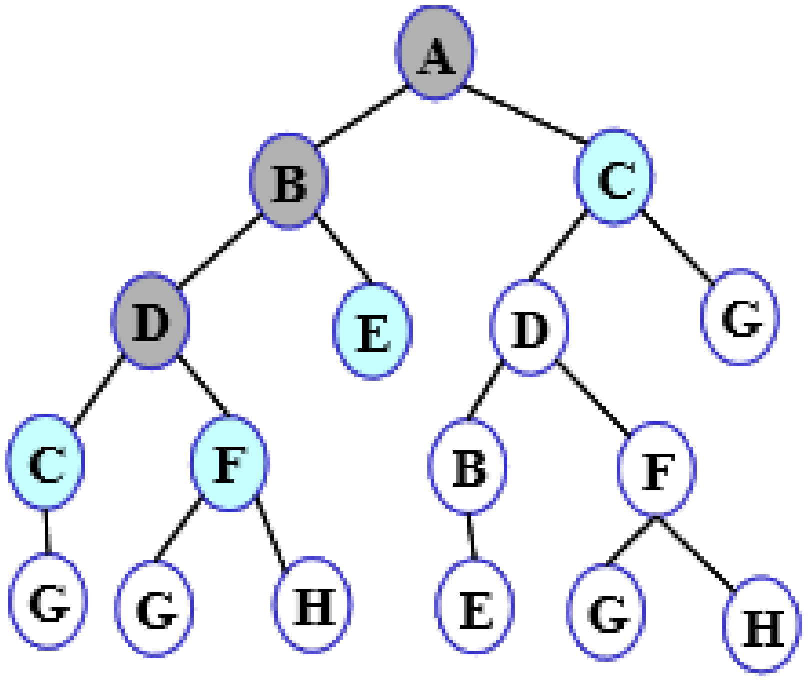
- Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



- FRINGE: **D, E, C**

Cont..

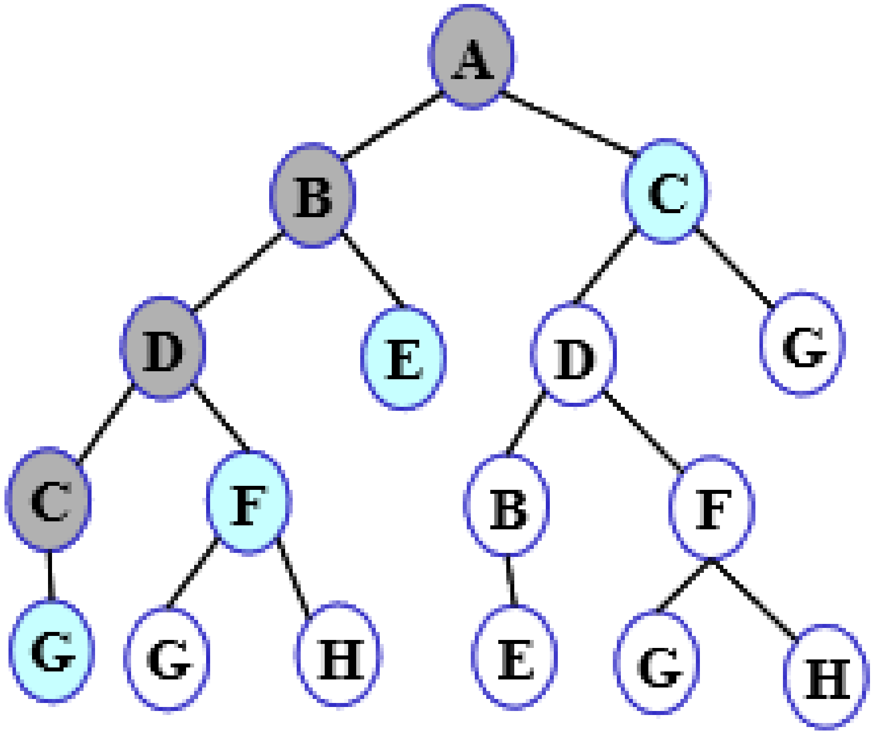
- Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



- FRINGE: **C, F, E, C**

Cont..

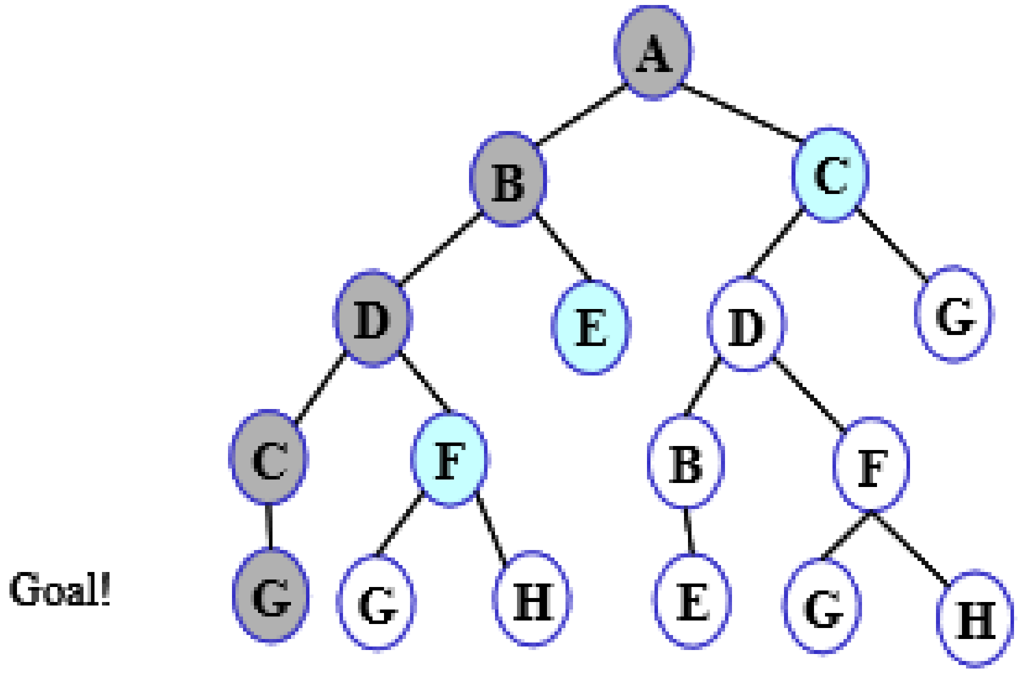
- Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



- FRINGE: **G, F, E, C**

Cont..

- Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm **terminates**.



- FRINGE: **G, F, E, C**

Properties of Depth First Search

- Let us now examine some properties of the DFS algorithm. The algorithm takes exponential time. If **N** is the maximum depth of a node in the search space, in the worst case the algorithm will take time $\boldsymbol{O(b^d)}$.
- However the space taken is linear in the depth of the search tree, $\boldsymbol{O(b^N)}$.
- Note that the time taken by the algorithm is related to the **maximum depth** of the search tree.
- If the search tree has ∞ depth, the algorithm may not terminate.
- This can happen if the search space is ∞. It can also happen if the search space contains cycles. Thus DFS is not complete.

Depth-first Search

2. Depth-first Search

- It 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.

- **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).

- Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Cont..

- **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.
- **Example:**

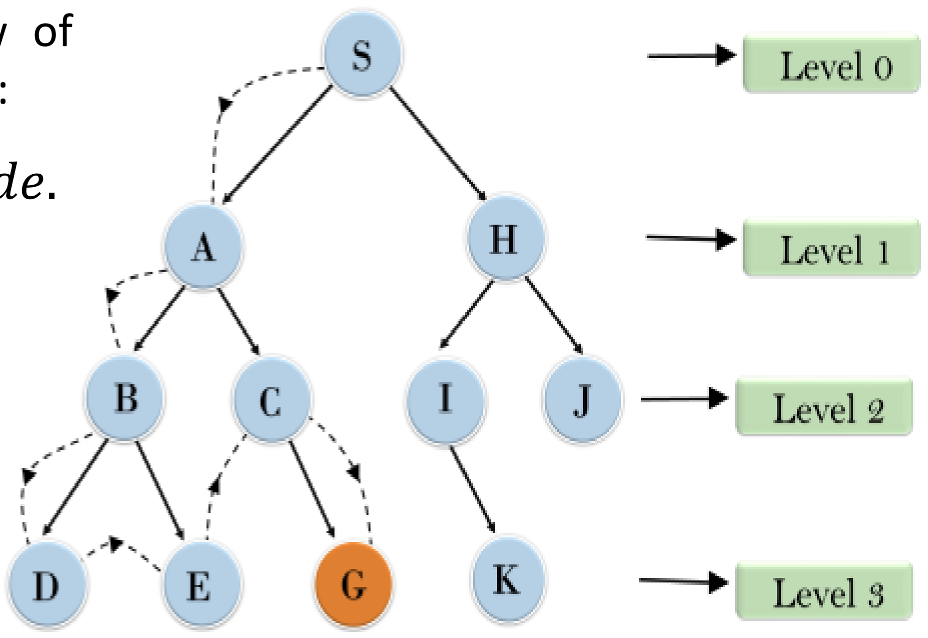
In the search tree, we have shown the flow of depth-first search, and it will follow the order as:

$$Root\,node \text{ —> } Left\,node \text{ —> } right\,node.$$

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.

$$S \text{ —> } A \text{ —> } B \text{ —> } D \text{ —> } E \text{ —> } C \text{ —> } G$$

Depth First Search



Cont..

- **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:

$$\boldsymbol{T(n) = 1 + n^2 + n^3 + \cdots \ldots \ldots + nm = O(nm)}$$

- 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.

Depth-Limited Search Algorithm

- A variation of DFS circumvents the above problem by keeping a depth bound. Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.
- **Depth-Limited Search Algorithm**
- Let fringe be a list containing the initial state
- **Loop**
  - if fringe is empty return failure
  - Node ← remove-first (fringe)
  - if Node is a goal
    - then return the path from initial state to Node
  - else if depth of Node = limit return cutoff
  - else add generated nodes to the **front of fringe**
- **End Loop**

Depth-Limited Search Algorithm

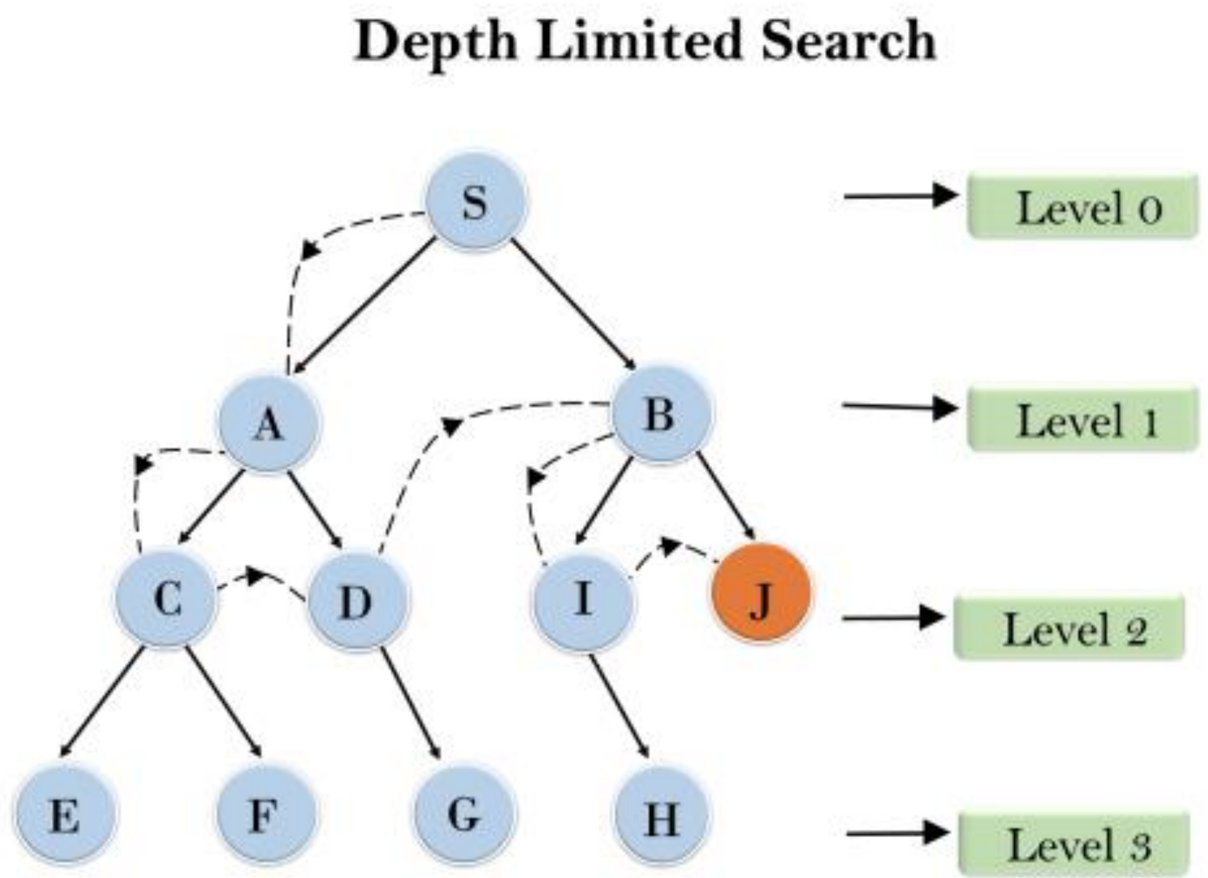
3. Depth-Limited Search Algorithm

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

Cont..

- Example



Cont..

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

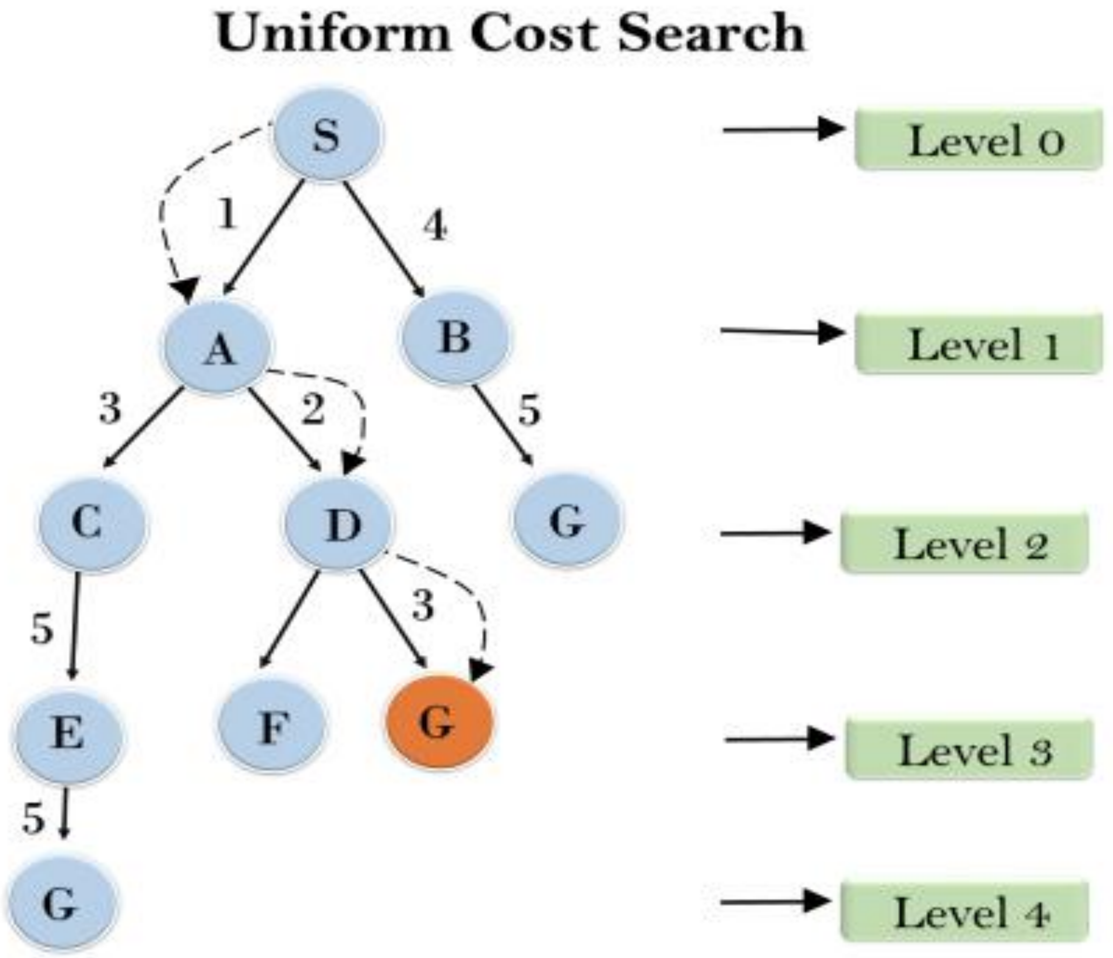
Uniform-cost Search Algorithm

4. Uniform-cost Search Algorithm

- It 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**. It 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**. It is **equivalent** to **BFS** algorithm if the **path cost** of all edges is the **same**.
- **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.

Cont..

- Example



Cont..

- **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 $= C*/\varepsilon + 1$. Here we have taken +1, as we start from state 0 and end to $C*/\varepsilon$.

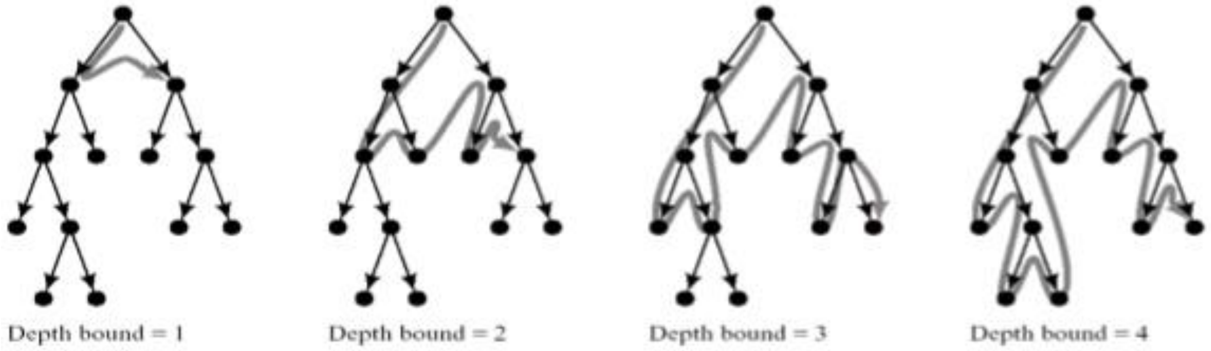
- Hence, the worst-case time complexity of Uniform-cost search is $\boldsymbol{O(b^{1} + [C*/\varepsilon])/}$.

- **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\boldsymbol{O(b^{1} + [C*/\varepsilon])}$.

- **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Depth-First Iterative Deepening (DFID)

- First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.
- DFID
  - until solution found do
  - DFS with depth cutoff c
  - c = c+1



- **Advantage**
  - Linear memory requirements of DFS
  - Guarantee for goal node of minimal depth
- Procedure
  - Successive DFS are conducted
  - each with depth bounds increasing by 1

Properties

- For large d the ratio of the number of nodes expanded by DFID compared to that of DFS is given by b/(b-1).
- For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative deepening search than BFS.
- The algorithm is
  - Complete
  - Optimal/Admissible if **all operators** have the **same cost**. Otherwise, not optimal but guarantees finding solution of **shortest length** (like BFS).
  - **Time complexity** is a **little worse** than **BFS** or **DFS** because **nodes** near the **top** of the search tree are generated **multiple times**, but because **almost all** the nodes are **near the bottom** of a tree, the worst case time complexity is still exponential, $\boldsymbol{O}(\boldsymbol{b}^{\boldsymbol{d}})$

Properties

– If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.

– Hence

$$b^d + 2b^{(d-1)} + \cdots + db <= {b^d}\Big/{\left(1 - {1}/{b}\right)^2} = O\left(b^d\right)$$

– Linear space complexity, O(bd), like DFS

Cont..

- DFID combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly).

- This algorithm is generally preferred for large state spaces where the solution depth is unknown.

- There is a related technique called iterative broadening is useful when there are many goal nodes.

- This algorithm works by 1st constructing a search tree by expanding only one child per node. In the 2nd iteration, 2 children are expanded, and in the ith iteration i children are expanded.

Iterative deepening depth-first Search

5. Iterative deepening depth-first Search

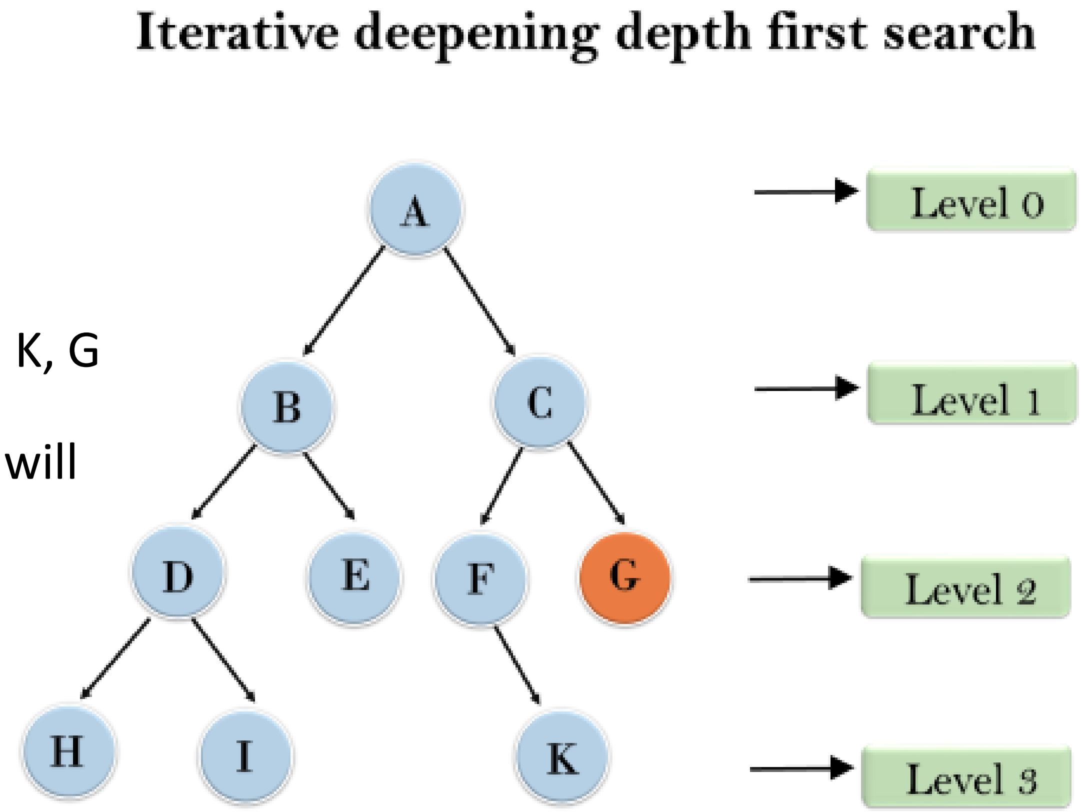
- This is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- It performs DFS up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node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

Cont..

- 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 4th iteration, the algorithm will find the goal node.



Cont..

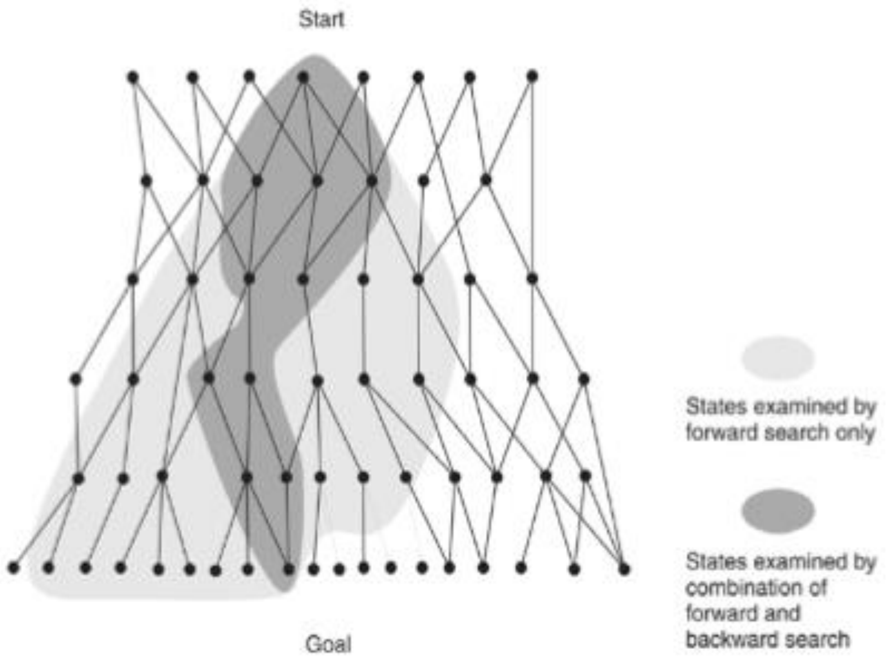
- **Completeness:** This algorithm is complete is if the branching factor is finite.
- **Time Complexity:** Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $\boldsymbol{O(b^d)}$.
- **Space Complexity:** The space complexity of IDDFS will be $\boldsymbol{O(bd)}$.
- **Optimal:** IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Bidirectional Search

- Let the search problem is such that the arcs are **bidirectional** i.e., if there is an **operator** that **maps** from state A to state B, there is **another operator** that **maps** from state B to state A.
- Many search problems have **reversible arcs**. **8-puzzle, 15-puzzle, path planning** etc. are examples of this. However there are other state space search formulations which do not have this property. The **water jug problem** is a problem that does not have this property.
- But if the arcs are **reversible**, you can see that instead of starting from the **start state** and searching for the **goal**, one may start from a **goal state** and try reaching the **start state**. If there is a single state that satisfies the goal property, the search problems are identical.
- How do we search backwards from goal?
- One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor. This is the motivation to consider bidirectional search.

Cont..

- Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when the **frontiers intersect**.
- A search algorithm has to be selected for each half. How does the algorithm know when the frontiers of the search tree intersect? For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
- Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure
- Also note that the algorithm works well only when there are unique start and goal states. Question: How can you make bidirectional search work if you have 2 possible goal states?


Start
Goal
States examined by forward search only
States examined by combination of forward and backward search

Cont..

- **Time and Space Complexities**
- Consider a search space with branching factor b. Suppose that the goal is d steps away from the start state. Breadth first search will expand $\boldsymbol{O(b^d)}$ nodes.
- If we carry out bidirectional search, the frontiers may meet when both the forward and the backward search trees have depth = d/2.
- Suppose we have a good hash function to check for nodes in the fringe. IN this case the time for bidirectional search will be $\boldsymbol{O(b^{d/2})}$.
- Also note that for at least one of the searches the frontier has to be stored.
- So the space complexity is also $\boldsymbol{O(b^{d/2})}$.

Bidirectional Search Algorithm

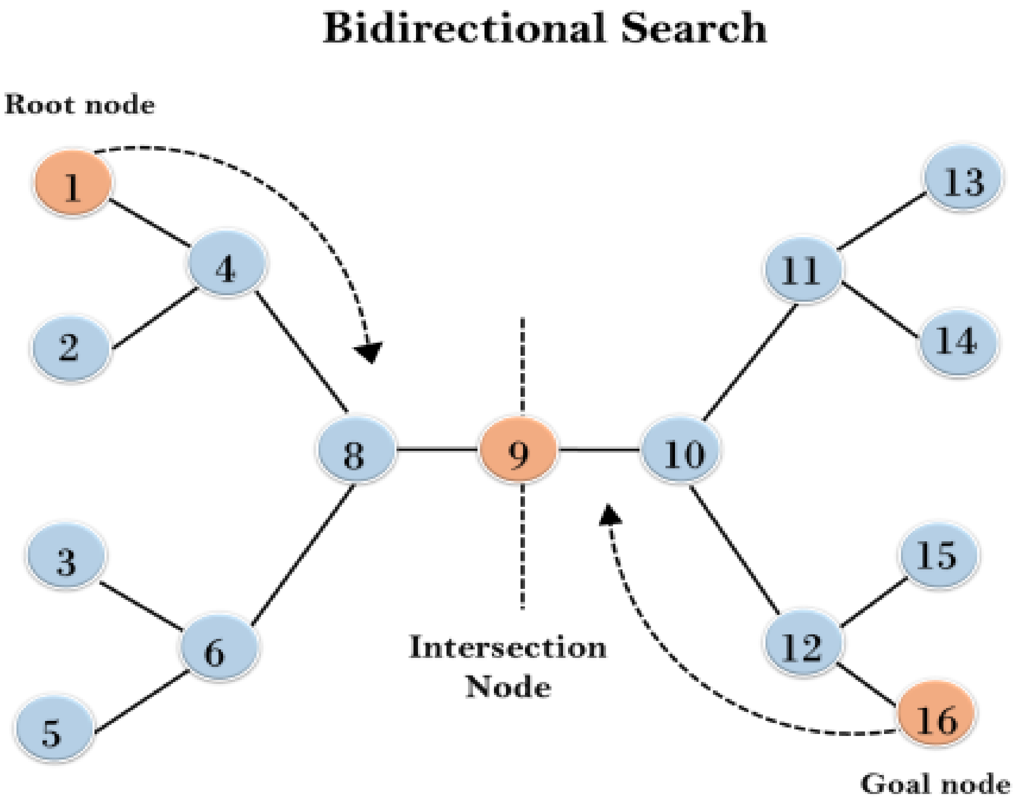
6. Bidirectional Search Algorithm

- It 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.
- It 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.
- It can use search techniques such as BFS, DFS, DLS, etc.
- 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.

Cont..

- 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.



Cont..

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

Search Graphs

- If the **search space** is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state.
- It is easy to consider a pathological example to see that the **search space size** may be exponential in the **total number of states**.
- In many cases we can modify the SA to avoid **repeated state expansions**. The way to avoid generating the same state again when not required, the SA can be modified to check a node when it is being generated.
- If a **node** corresponding to the **state** is already in **OPEN**, the **new node** should be discarded. But if the **state** was in **OPEN** **earlier** but has been **removed** and **expanded:-** **To keep track** of this phenomenon, we use another list called **CLOSED**, which records all the **expanded nodes**.
- The newly generated node is checked with the nodes in **CLOSED** too, and it is put in **OPEN** if the corresponding state cannot be found in **CLOSED**.

Graph search Algorithm

- Let fringe be a list containing the initial state
- Let closed be initially empty
- **Loop**
  - if fringe is empty return failure
  - Node ← remove-first (fringe)
  - if Node is a goal
    - then return the path from initial state to Node
  - else put Node in closed
    - generate all successors of Node S
      - for all nodes m in S
        - » if m is not in fringe or closed
        - » merge m into fringe
- **End Loop**

Cont..

- This algorithm is expensive.
- Apart from the fact that the CLOSED list has to be maintained, the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED.
- Unless there is a very efficient way to index nodes, this will require additional overhead for every node.
- In many search problems, we can adopt some less computationally intense strategies.
- Such strategies do not stop duplicate states from being generated, but are able to reduce many of such cases.

Cont..

- The simplest strategy is to not return to the state the algorithm just came from. This simple strategy avoids many node re-expansions in **15-puzzle** like problems.
- 2nd strategy is to check that you do not create paths with cycles in them. This algorithm only needs to check the nodes in the current path so is much more efficient than the full checking algorithm. Besides this strategy can be employed successfully with depth first search and not require additional storage.
- 3rd strategy is as outlined in the table. Do not generate any state that was ever created before.
- Which strategy one should employ must be determined by considering the frequency of “loops” in the state space.

Informed Search

- We have seen that uninformed search methods that systematically explore the state space and find the goal.
- They are inefficient in most cases. Informed search methods use problem specific knowledge, and may be more efficient.
- At the heart of such algorithms there is the concept of a heuristic function.
- **Heuristics**
- Heuristic means "rule of thumb".
- To quote Judea Pearl, "Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal".
- In heuristic search or informed search, heuristics are used to identify the most promising search path.

Informed Search

- 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.
  1. Greedy Search
  2. A* Search

Example : Heuristic Function

- A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. It is denoted by h(n).
- h(n) = estimated cost of the cheapest path from node n to a goal node
- **Example 1**: We want a path from Kolkata to Guwahati Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati h(Kolkata) = euclideanDistance(Kolkata, Guwahati)
- **Example 2**: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.

2	8	3
1	6	4
	7	5

Initial State

1	2	3
8		4
7	6	5

Goal State

Cont..

- 1st picture shows the current state n, and the 2nd picture the goal state.
  - ***h(n) = 5*** because the tiles 2, 8, 1, 6 and 7 are out of place.
- Manhattan Distance Heuristic: Another heuristic for 8-puzzle. This heuristic sums the distance that tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.
- For the above example, using the Manhattan distance heuristic,
  - h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6

2	8	3
1	6	4
	7	5

Initial State

1	2	3
8		4
7	6	5

Goal State

Heuristics function

- It is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. This method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- It estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.
- **Admissibility of the heuristic function is given as:**

$$h(n) \;<=\; h*(n)$$

- Here h(n) is **heuristic cost**, and h*(n) is the **estimated cost**. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search

- It is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, **OPEN** and **CLOSED** list.
- In the **CLOSED** list, it places those nodes which have already expanded and in the **OPEN** list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.
- In the informed search we will discuss two main algorithms which are given below:
- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

Best-first Search (Greedy Search)

- Uniform Cost Search is a special case of the BFS algorithm. It maintains a **priority queue** of nodes to be explored. A **cost function** f(n) is applied to **each node**. The nodes are put in **OPEN** in the order of their f values. Nodes with smaller f(n) values are expanded earlier.
- **Best-first Search Algorithm (Greedy Search)**
- Let fringe be a priority queue containing the initial state
- **Loop**
  - if *fringe* is empty return failure
  - Node ← remove-first (*fringe*)
  - if Node is a goal
    - then return the path from initial state to Node
  - else generate all successors of Node, and
    - put the newly generated nodes into fringe according to their f values
- **End Loop**

Consider different ways of defining the function f. This leads to different search algorithms.

Greedy Search

- In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.
- We use a heuristic function
  - f(n) = h(n); h(n) estimates the distance remaining to a goal.
- Greedy algorithms often perform very well. They tend to find good solutions quickly, although not always optimal ones.
- The resulting algorithm is not optimal.
- The algorithm is also incomplete, and it may fail to find a solution even if one exists. This can be seen by running greedy search on the following example.
- A good heuristic for the route-finding problem would be straight-line distance to the goal.

Best-first Search Algorithm (Greedy Search)

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.

$$f(n) = g(n)$$

- Were, 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.

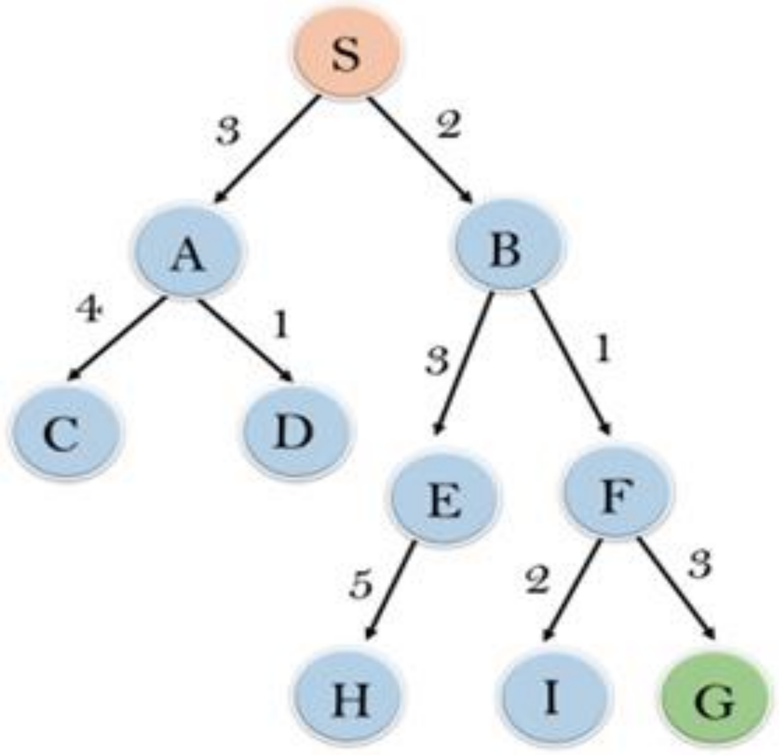
Cont..

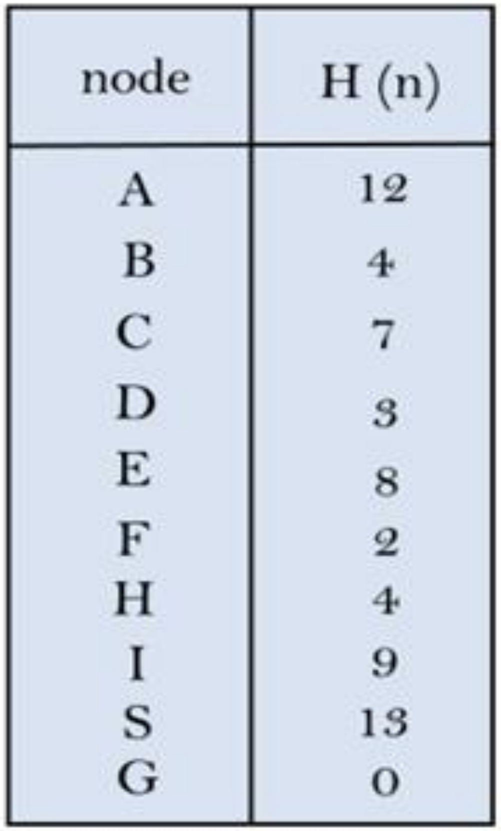
- **Advantages:**
  - Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
  - This algorithm is more efficient than BFS and DFS algorithms.
- **Disadvantages:**
  - It can behave as an unguided depth-first search in the worst case scenario.
  - It can get stuck in a loop as DFS.
  - This algorithm is not optimal.
- Example :

  Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.

Cont..

- Example : Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





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

Cont..

- In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.
- **Expand the nodes of S and put in the CLOSED list**

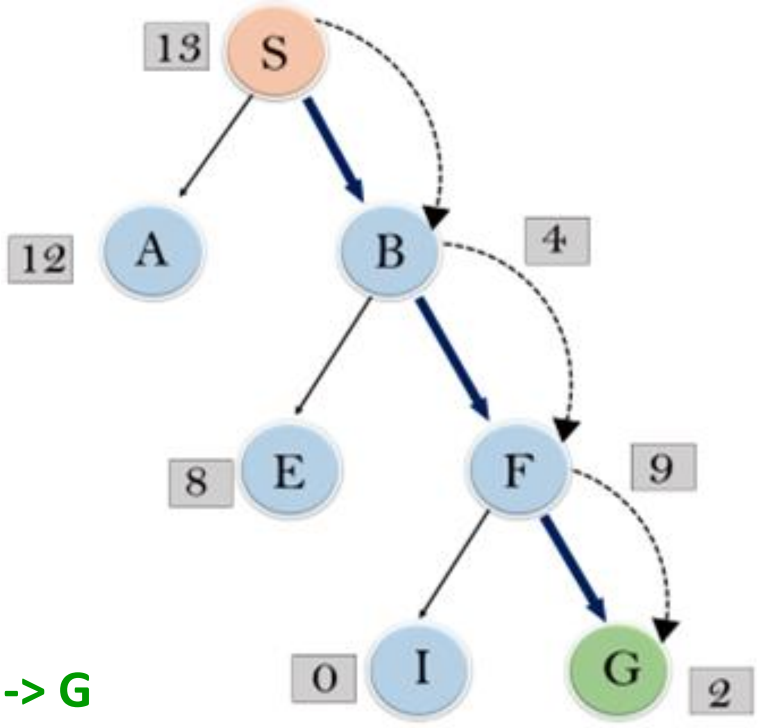
Initialization : Open [S], Closed []

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

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

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

Iteration 4 : 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**

Cont..

- **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.

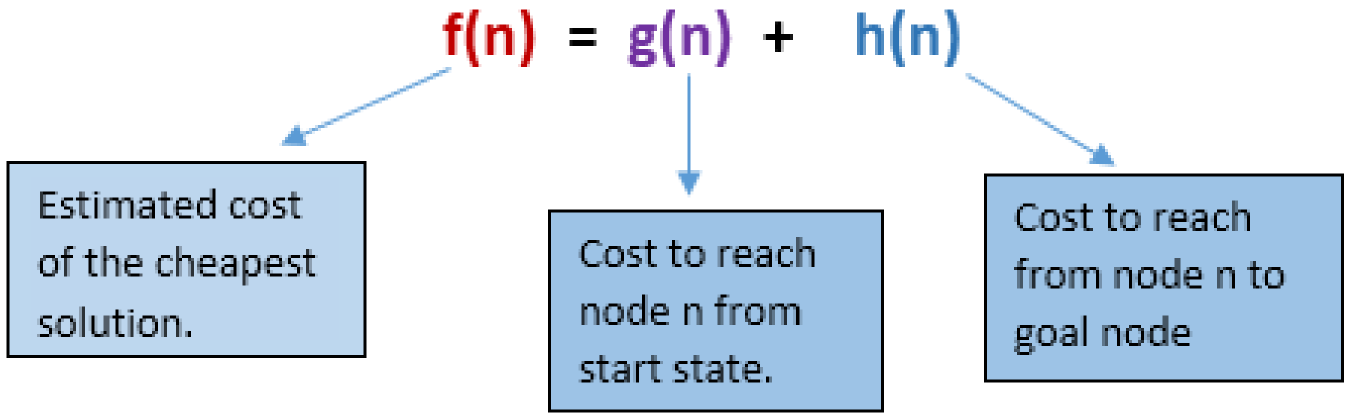
A* Search Algorithm

2. A* Search Algorithm

- A* search is the most commonly known form of BFS. 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 BFS, by which it solve the problem efficiently.
- It 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).

Cont..

- Here, 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

- At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

A* search Algorithm

- **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**.

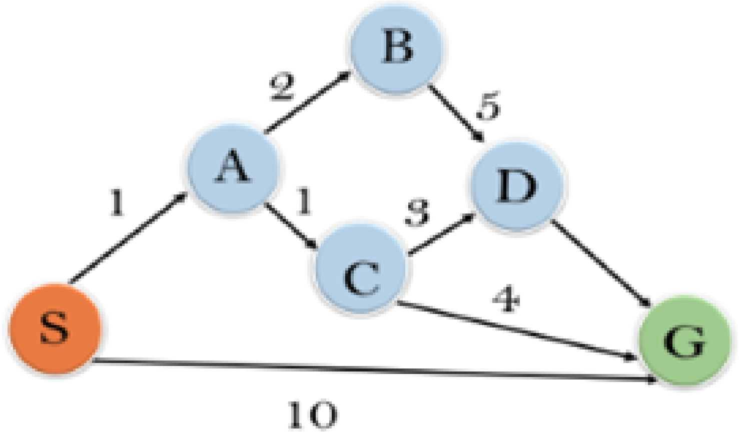
Cont..

- **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.

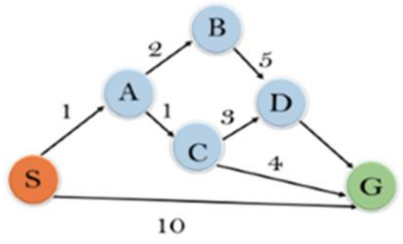
- **Disadvantages:**
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - A* search algorithm has some complexity issues.
  - The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various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

Cont..

- **Example:** **Here**, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
- Here we will use OPEN and CLOSED list.



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

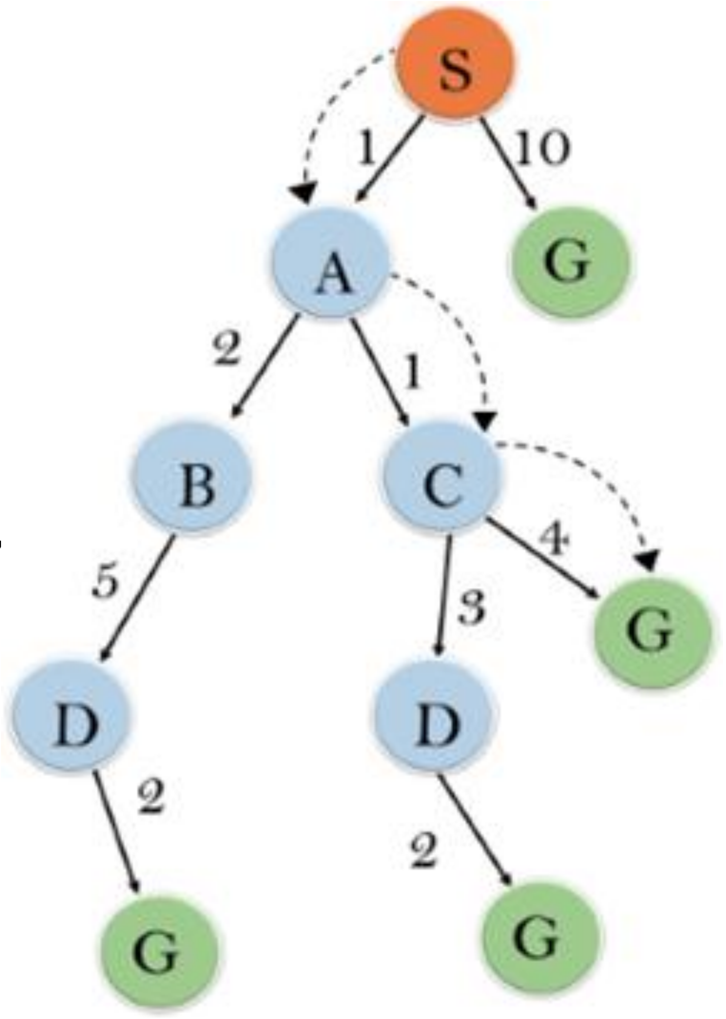


State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Cont..

- **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)}

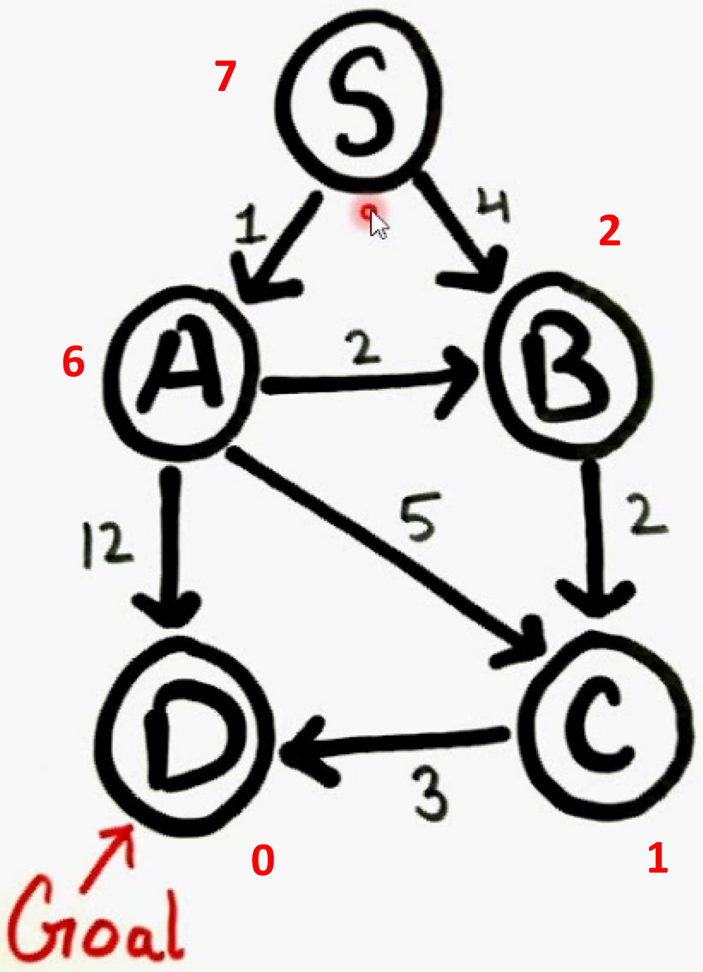
- **Iteration 4** will give the final result, as **S->A->C->G** it provides the optimal path with cost 6.



Cont..

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until final state is reached.

Example 1



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

for S,
0+7=7

S→A,
1+6=7

S→B,
4+2=6

S→B→C
6+1=7

S→A→B
3+2=5

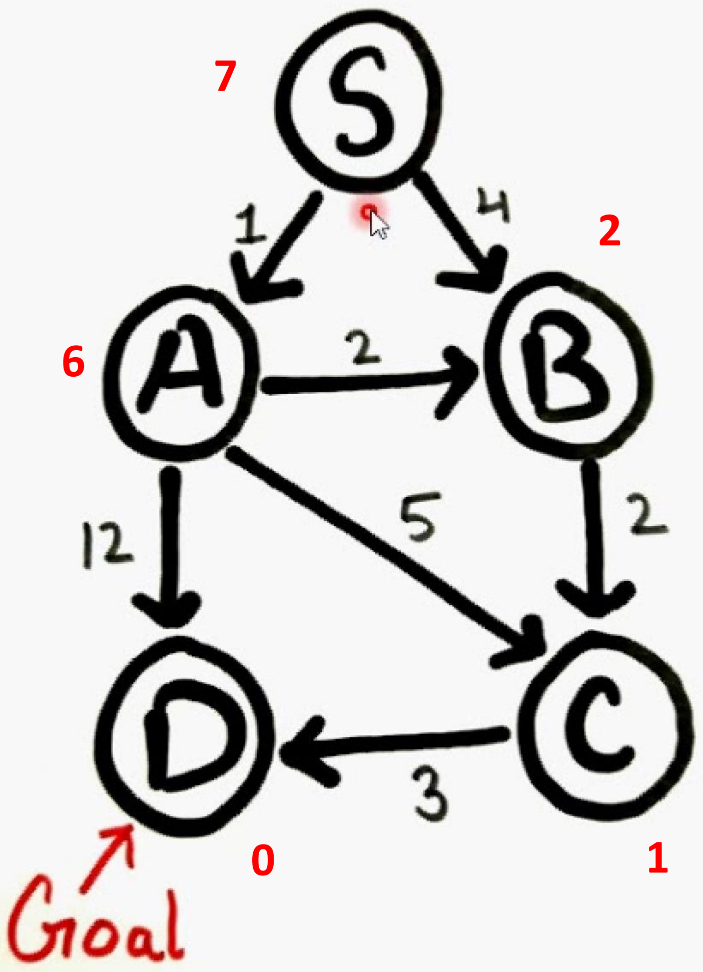
S→A→C
6+1=7

S→A→D
13+0=13

Heuristic Value	
S	7
A	6
B	2
C	1
D	0

S→A=7 ✓
S→B=6 ✓
S→B→C=7
S→A→B=5
S→A→C=7
S→A→D=13

Cont..



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

for S,
0+7=7

S→A,
1+6=7

S→B,
4+2=6

S→B→C
6+1=7

S→A→B
3+2=5

S→A→C
6+1=7

S→A→D
13+0=13

S ->B->C->D
9+0=9

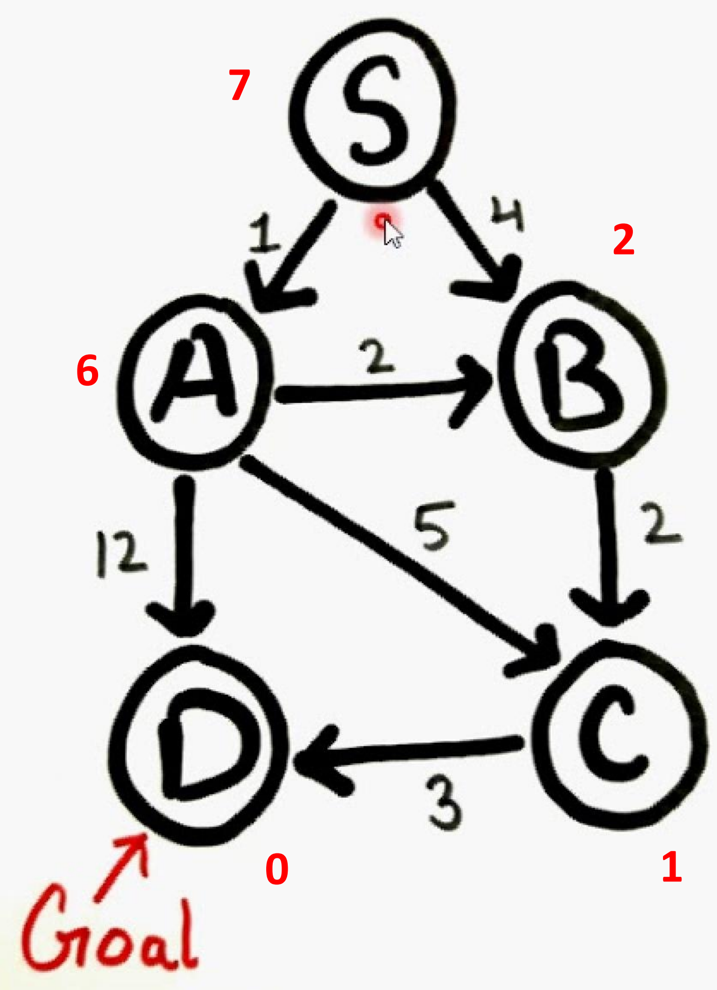
S -> A -> B->C->D
8+0=8

Heuristic Value	
S	7
A	6
B	2
C	1
D	0

S→A=7 ✓
S→B=6 ✓
S→B→C=7
S→A→B=5
S→A→C=7
S→A→D=13

S ->B->C->D = 9
S ->A->B->C->D = 8

Cont..



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

for S,
0+7=7

S→A,
1+6=7

S→B,
4+2=6

S→B→C
6+1=7

S→A→B
3+2=5

S→A→C
6+1=7

S→A→D
13+0=13

S ->B->C->D
9+0=9

S -> A -> B->C->D
8+0=8

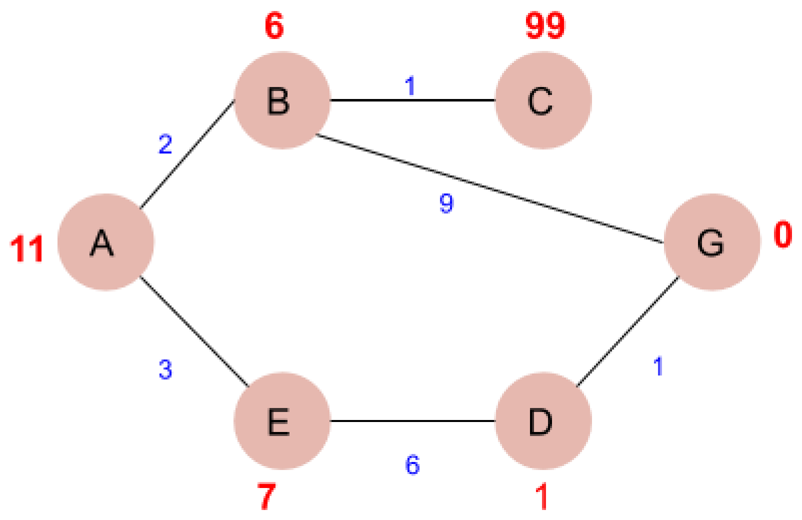
Heuristic Value	
S	7
A	6
B	2
C	1
D	0

S→A=7 ✓
S→B=6 ✓
S→B→C=7
S→A→B=5
S→A→C=7
S→A→D=13

S ->A->B->C->D = 8

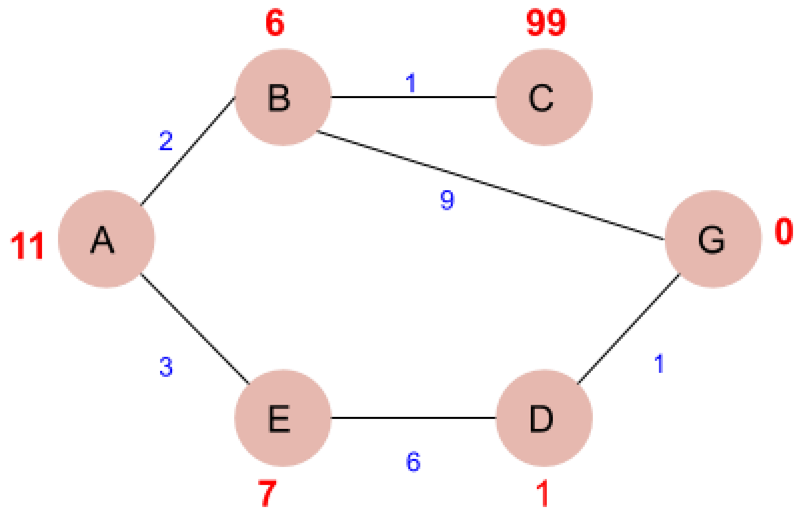
Now comparing all the paths that lead us to the goal, we conclude that S → A → B → C → D is the most cost-effective path to get from S to D.

Example 2



- Apply the steps of the A* Search algorithm to find the shortest path from A to G using the above graph.
  - The numbers on edges represent the distance between the nodes.
  - Numbers on nodes represent the heuristic value.

Cont..



State	h(n)
A	11
B	6
C	99
D	1
E	7
G	0

- *Step 1:* Let’s start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero and from the graph, we get the heuristic value of A is 11,

  ∴ f(x) = g(x) + h(x) = 0+ 11 =11

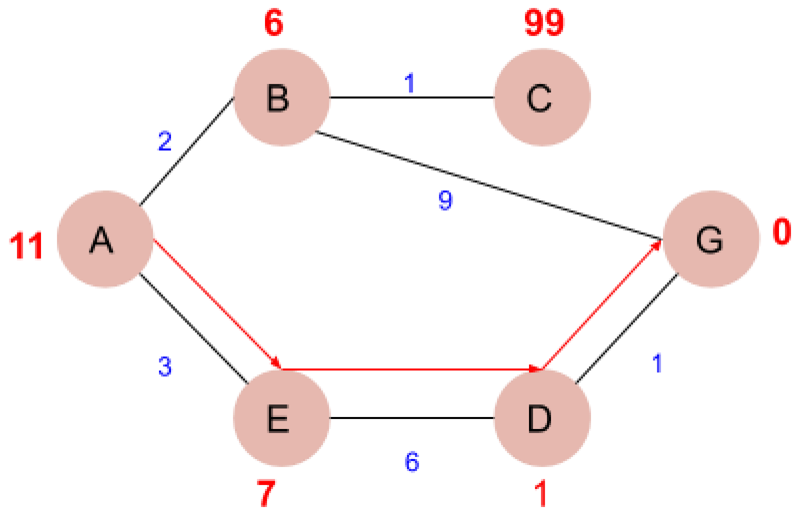
  Thus for A, we can write A=11

Cont..

- *Step 2:* Now from A, we can go to point B or point E, so we compute f(x) for each of them
  - A → B = f(B) = 2 + 6 = 8
  - A → E = f(E) = 3 + 6 = 9
- *Step 3:* Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B
  - A → B → C = f(C) = (2 + 1) + 99 = 102
- *Step 4:* Since there is no path between C and G, the heuristic cost is set infinity or a very high value
  - A → B → G = f(G) = (2 + 9) + 0 = 11
- *Step 5:* Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further
  - A → E → D = f(D) = (3 + 6) + 1 = 10

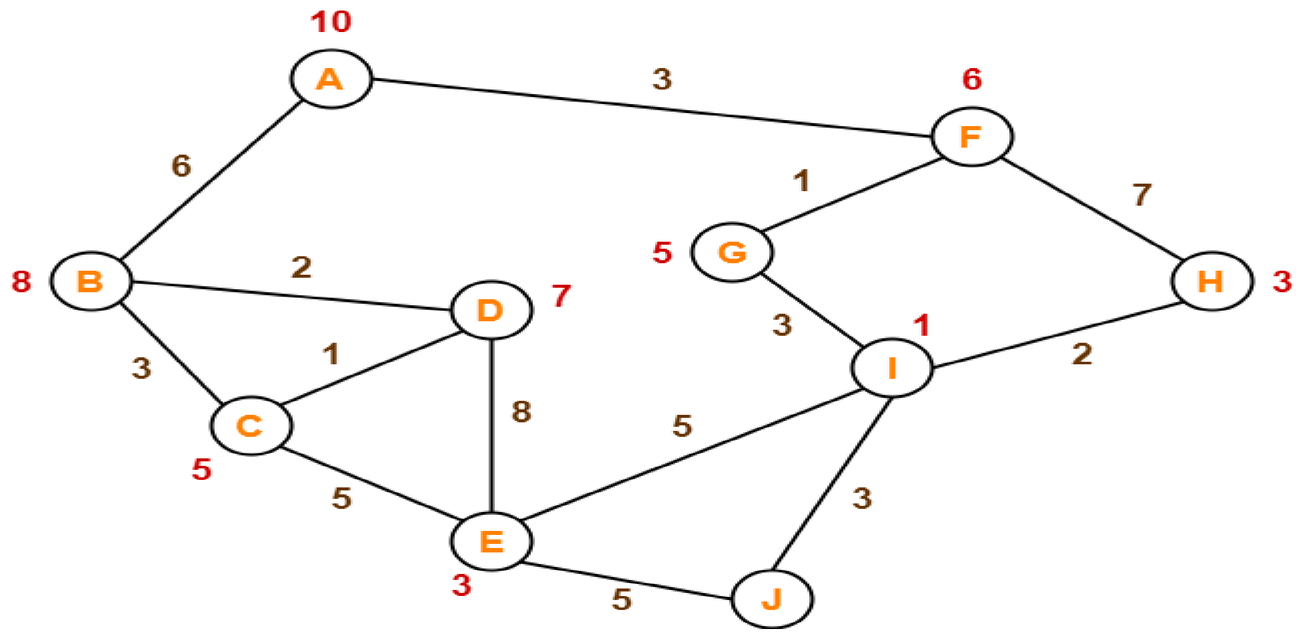
Cont..

- *Step 6:* Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D
  - A → E → D → G = f(G) = (3 + 6 + 1) + 0 = **10**
- *Step 7:* Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.



Example 3

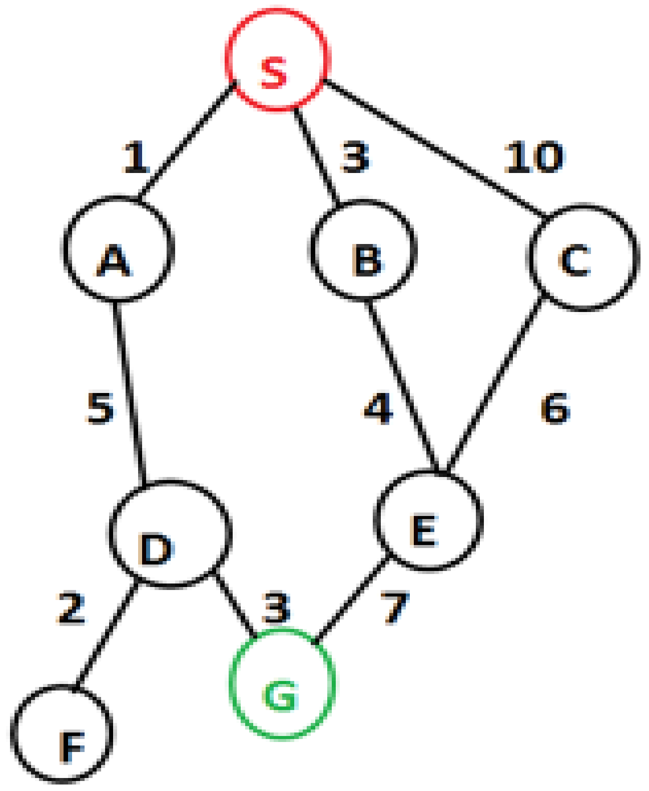
- Consider the following graph-



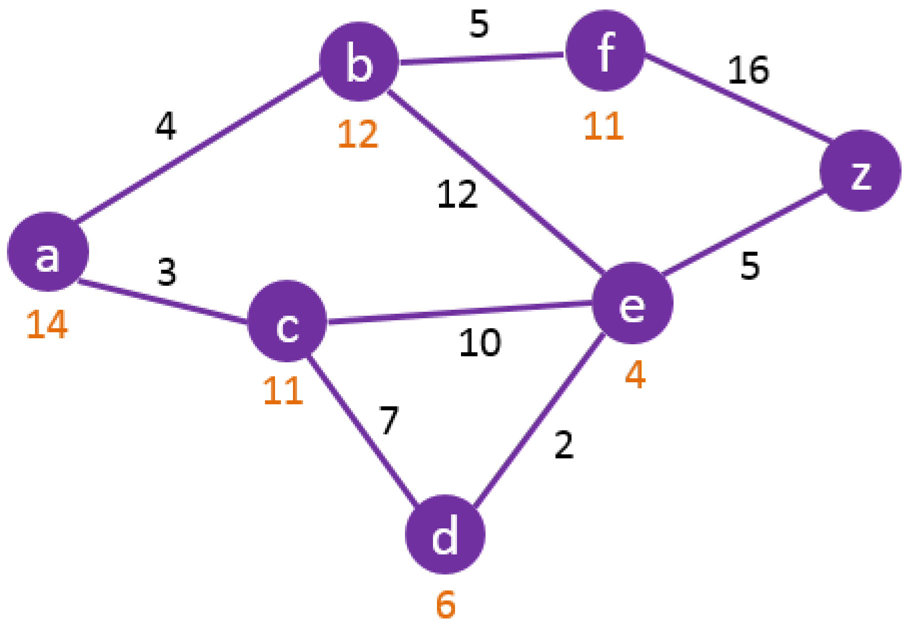
- The numbers written on edges represent the distance between the nodes.
- The numbers written on nodes represent the heuristic value.
- Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

Example 4

- Consider the following graph-
- The numbers written on edges represent the heuristic value.
- Find the most cost-effective path to reach from start state S to final state G using A* Algorithm.

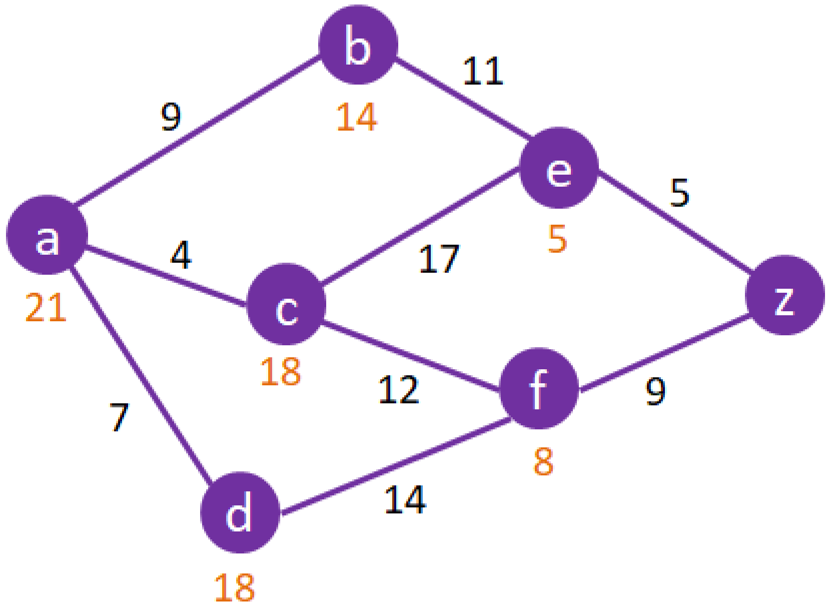


Example 5



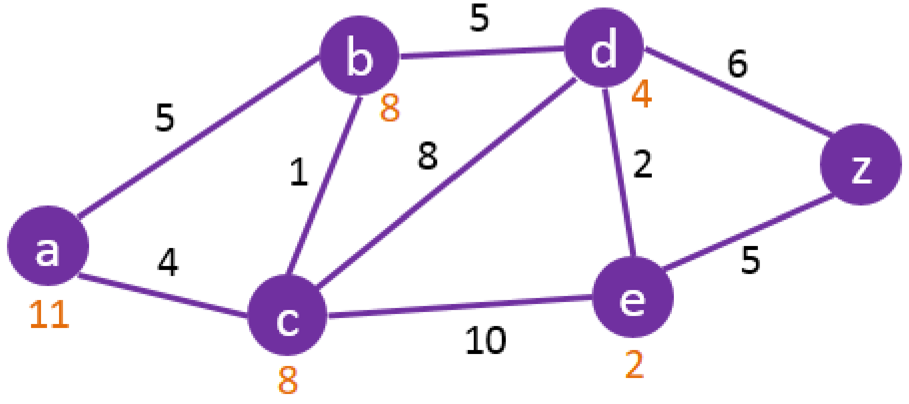
- The numbers on edges represent the distance between the nodes.
- Numbers on nodes represent the heuristic value.
- Find the shortest path to travel from A to Z using A* Algorithm.

Example 6



- Apply the steps of the A* Search algorithm to find the shortest path from A to Z using the following graph:

Example 7



- Apply the steps of the A* Search algorithm to find the shortest path from A to Z using the following graph:

Example 8

- Given an initial state of a 8-puzzle problem and final state to be reached-

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

- Find the most cost-effective path to reach the final state from initial state using A* Algorithm.
- Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Cont..

- Notes:-
- A* Algorithm is one of the best path finding algorithms.
- But it does not produce the shortest path always.
- This is because it heavily depends on heuristics.

Cont..

- *Applications of A* Star algorithm*
- A Star algorithm is often used to search for the lowest cost path from the start to the goal location in a graph of cells/voronoi/visibility/quad tree.

- *Some real life problems which uses the above algorithm*
- The algorithm solves problems like *8-puzzle problem* and *missionaries & Cannibals problems*.

- *A** is generally considered to be the best pathfinding algorithm. Also Djikshtra's algorithm is essentially the same as *A** except that there is no heuristic(H is always 0). Because it has no heuristic, it searches by expanding out equally in every direction. So it usually ends up exploring a much larger area before the target is found. This generally makes it slower than *A**.

Cont..

- **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.

Cont..

- 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 $\boldsymbol{O(b^\wedge d)}$

Local Search

- Local search methods work on complete state formulations. They keep only a small number of nodes in memory.
- Local search is useful for solving optimization problems: Often it is easy to find a solution; But hard to find the best solution Algorithm goal: find optimal configuration (e.g., TSP),
  - Hill climbing
  - Gradient descent
  - Simulated annealing
- For some problems the state description contains all of the information relevant for a solution. Path to the solution is unimportant.
  - **Examples:**
    - map coloring
    - 8-queens

Cont..

- Start with a state configuration that violates some of the constraints for being a solution, and make gradual modifications to eliminate the violations.
- One way to visualize iterative improvement algorithms is to imagine every possible state laid out on a landscape with the height of each state corresponding to its goodness.
- Optimal solutions will appear as the highest points. Iterative improvement works by moving around on the landscape seeking out the peaks by looking only at the local vicinity.

Iterative improvement

- In many optimization problems, the path is irrelevant; the goal state itself is the solution.
- An example of such problem is to find configurations satisfying constraints (e.g., n-queens).
- Algorithm:
  - Start with a solution
  - Improve it towards a good solution
- Example:
  - N queens
  - Goal: Put n chess-game queens on an n x n board, with no two queens on the same row, column, or diagonal.

Chess board reconfigurations

- Here, goal state is initially unknown but is specified by constraints that it must satisfy
- Hill climbing (or gradient ascent/descent)
- Iteratively maximize “value” of current state, by replacing it by successor state that has highest value, as long as possible.
- Note: minimizing a “value” function v(n) is equivalent to maximizing –v(n), thus both notions are used interchangeably.
- Hill climbing – example
- Complete state formulation for 8 queens Successor function: move a single queen to another square in the same column Cost: number of pairs that are attacking each other. Minimization problem

Hill Climbing Algorithm in AI

- Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.
- *Note: minimizing a "value" function v(n)* is equivalent to *maximizing –v(n)*, thus both notions are used interchangeably.
- **Algorithm:**
  1. determine successors of current state
  2. choose successor of maximum goodness (break ties randomly)
  3. if goodness of best successor is less than current state's goodness, stop
  4. otherwise make best successor the current state and go to step 1
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.

Cont..

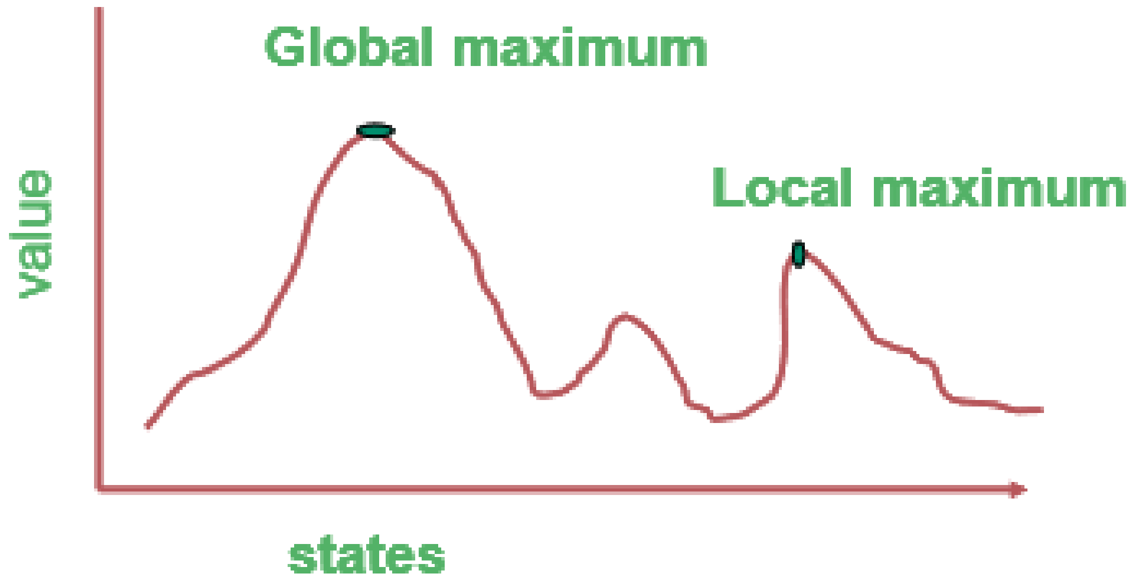
- Problems:
  - **Local maxima:** Once the top of a hill is reached the algorithm will halt since every possible step leads down.
  - **Plateaux:** If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.
  - **Ridges:** If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

Cont..

- Shape of state space landscape strongly influences the success of the search process. A very spiky surface which is flat in between the spikes will be very difficult to solve.
- Can be combined with nondeterministic search to recover from local maxima.
- **Random-restart hill-climbing** is a variant in which reaching a local maximum causes the current state to be saved and the search restarted from a random point. After several restarts, return the best state found. With enough restarts, this method will find the optimal solution.
- **Gradient descent** is an inverted version of hill-climbing in which better states are represented by lower cost values. Local minima cause problems instead of local maxima.

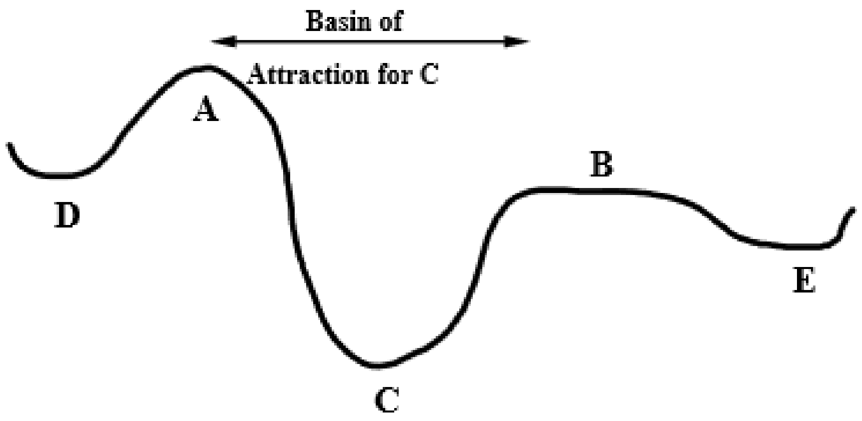
Cont..

- **Hill climbing - example**
- **Complete state formulation for 8 queens**
  - Successor function: move a single queen to another square in the same column
  - Cost: number of pairs that are attacking each other.
- **Minimization problem**
- **Problem:** depending on initial state, may get stuck in local extremum.



Cont..

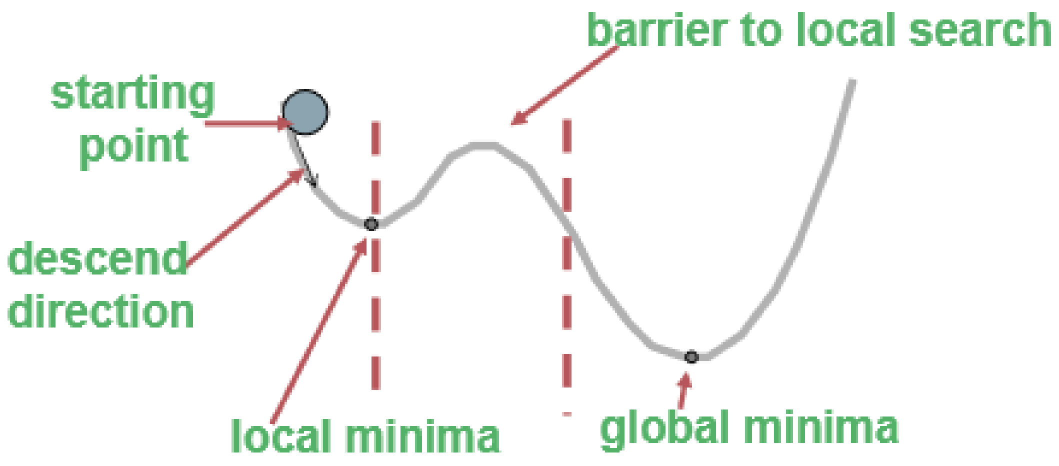
- Minimizing energy
- Compare our state space to that of a physical system that is subject to natural interactions
- Compare our value function to the overall potential energy E of the system.
- On every updating, we have DE ≤ 0



- Hence the dynamics of the system tend to move E toward a minimum.
- We stress that there may be different such states — they are local minima. Global minimization is not guaranteed.

Cont..

- Problem: How do you avoid this local minima?



- **Simulated annealing: basic idea**
  - From **current state**, pick a **random successor state**;
  - If it has **better value** than current state, then "**accept the transition**," that is, use successor state as current state;
  - Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).
  - So we accept to sometimes "**un-optimize**" the value function a little with a non-zero probability.

Hill Climbing Algorithm in AI

- It 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.
- It 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**.
- It is mostly used when a **good heuristic** is available.
- Here, 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.

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**.

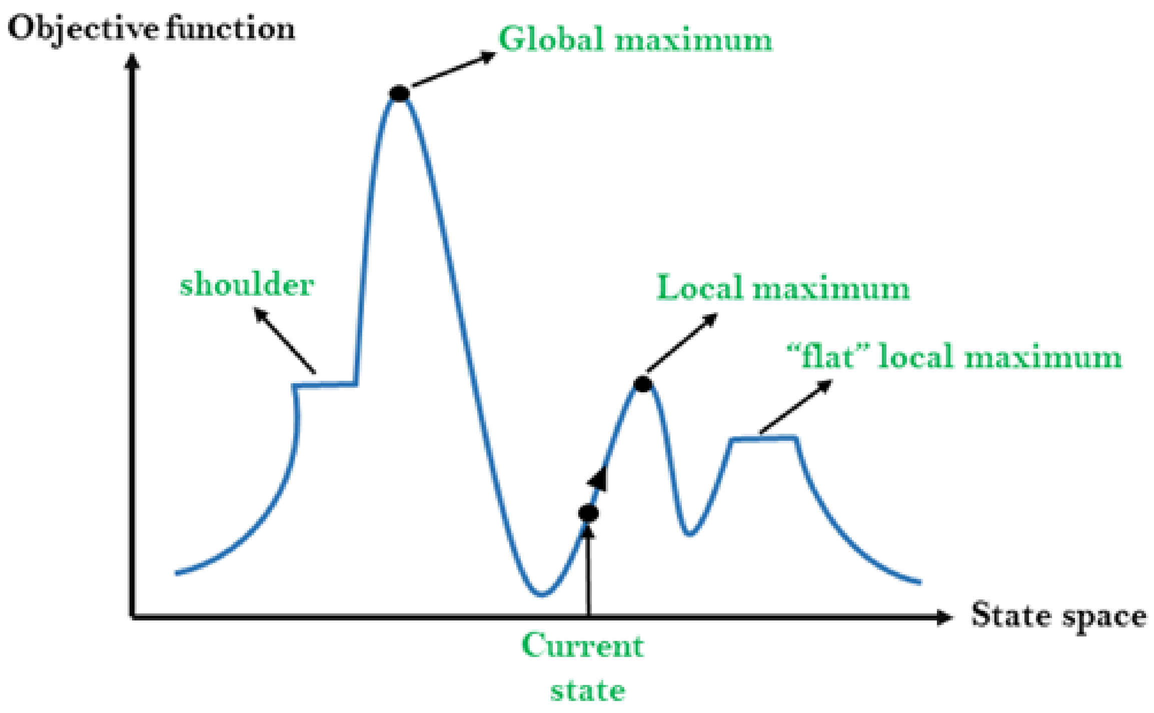
- If the function of Y-axis is **Objective function**, then the **goal** of the **search** is to find the **global maximum** and **local maximum**.

State Space diagram for Hill Climbing

- **State space diagram** is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).
- **X-axis :** denotes the state space i.e. states or configuration our algorithm may reach.
- **Y-axis :** denotes the values of objective function corresponding to a particular state.
- The **best solution** will be that state space where **objective function** has maximum value(global 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

- Types of Hill Climbing Algorithm
    1. Simple hill Climbing:
    2. Steepest-Ascent hill-climbing:
    3. 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

Simple Hill Climbing Algorithm

- **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:** Perform these to 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.

Steepest-Ascent hill climbing

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

Steepest-Ascent hill climbing Algorithm

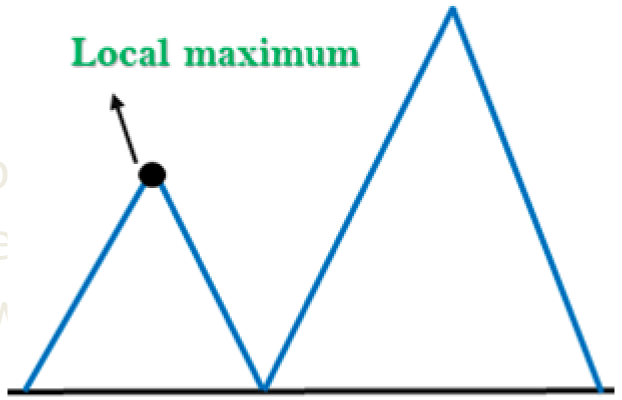
- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
  - Let SUCC be a state such that any successor of the current state will be better 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.

Stochastic hill climbing

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:**
  - i. **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.

lo state in the landscape which is better than e states, but there is another state also present w local maximum

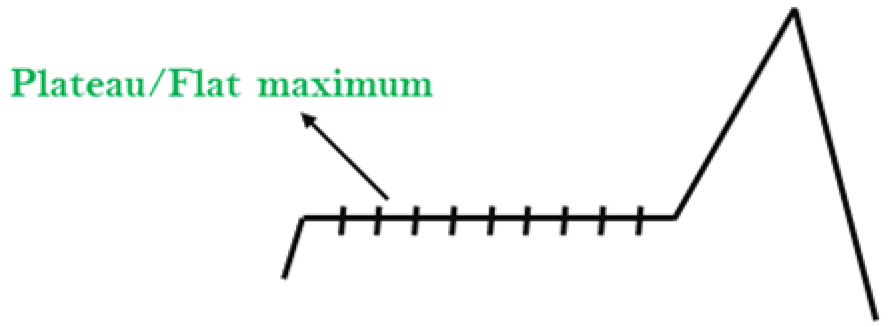


– **Solution:** Backtracking technique can be a solution of the **local maximum** in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

Plateau / flat local maximum

ii. Plateau / flat local maximum

- It 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

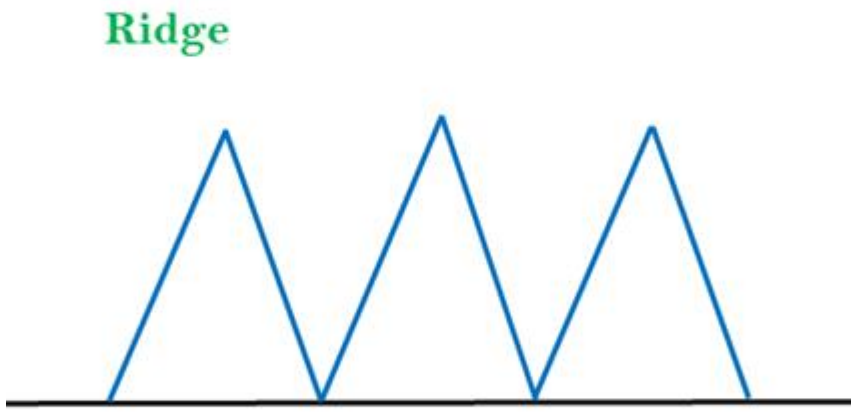


- **Solution:** The solution for the plateau is to take **big steps or very little steps** while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.

Ridges

iii. 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.

Means-Ends Analysis in AI

- We have studied the strategies which can reason either in forward or backward, but a mixture of the two directions is appropriate for solving a complex and large problem. Such a mixed strategy, make it possible that first to solve the major part of a problem and then go back and solve the small problems arise during combining the big parts of the problem. Such a technique is called **Means-Ends Analysis**.
- Means-Ends Analysis is problem-solving techniques used in Artificial intelligence for limiting search in AI programs.
- It is a mixture of Backward and forward search technique.
- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works

- The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving.

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

Operator Subgoaling

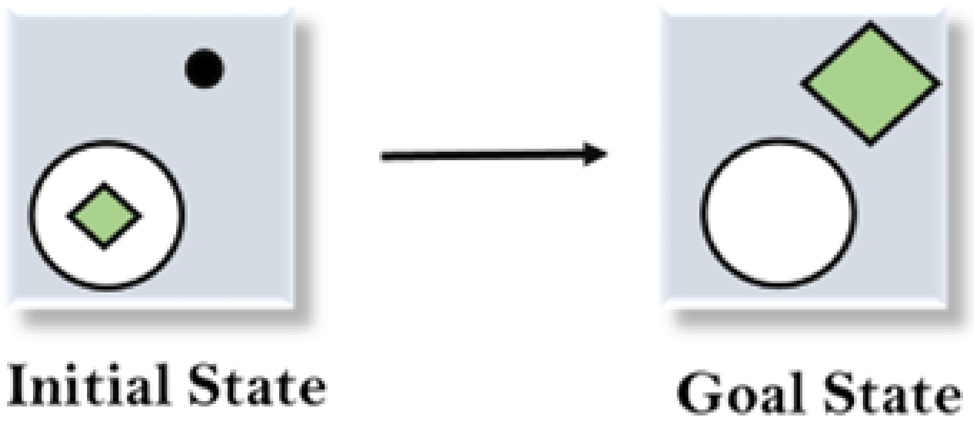
- In the MEA process, we detect the differences between the current state and goal state.
- Once these differences occur, then we can apply an operator to reduce the differences.
- But sometimes it is possible that an operator cannot be applied to the current state.
- So we create the sub-problem of the current state, in which operator can be applied, such type of backward chaining in which operators are selected, and then sub goals are set up to establish the preconditions of the operator is called **Operator Subgoaling**.

Algorithm for Means-Ends Analysis

- Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.
- **Step 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.
  - Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
  - Attempt to apply operator O to CURRENT. Make a description of two states.
    i) O-Start, a state in which O?s preconditions are satisfied.
    ii) O-Result, the state that would result if O were applied In O-start.
  - If **(First-Part <-- MEA (CURRENT, O-START)**
    And **(LAST-Part <-- MEA (O-Result, GOAL)**, are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.
- The above-discussed algorithm is more suitable for a simple problem and not adequate for solving complex problems.

Example of Mean-Ends Analysis

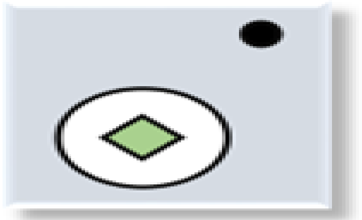
- Let's take an example where we know the initial state and goal state as given below. In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.



Cont..

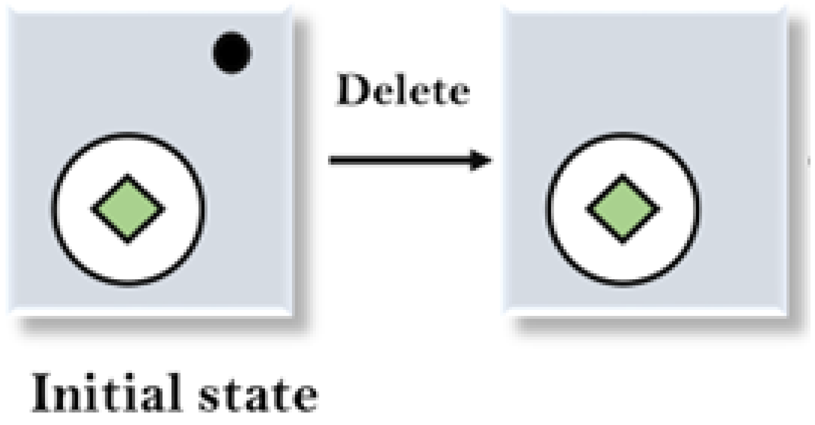
- **Solution:**
- To solve the above problem, we will first find the differences between initial states and goal states, and for each difference, we will generate a new state and will apply the operators.
- The operators we have for this problem are:
  - **Move**
  - **Delete**
  - **Expand**

1. **Evaluating the initial state:** In the first step, we will evaluate the initial state and will compare the initial and Goal state to find the differences between both states.


Initial state

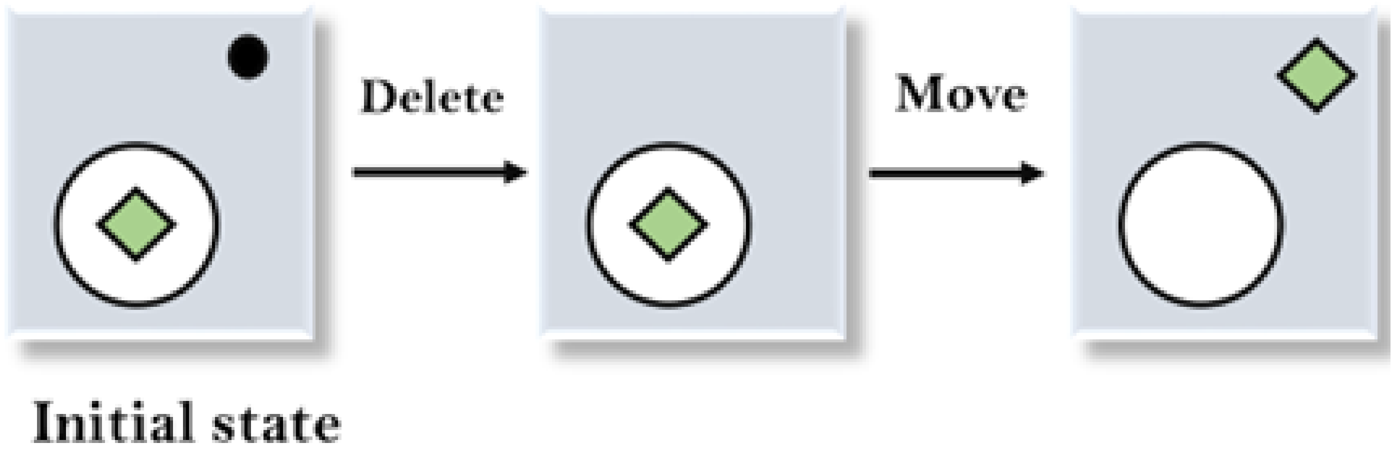
Cont..

2. Applying Delete operator: As we can check the first difference is that in goal state there is no dot symbol which is present in the initial state, so, first we will apply the **Delete operator** to remove this dot.



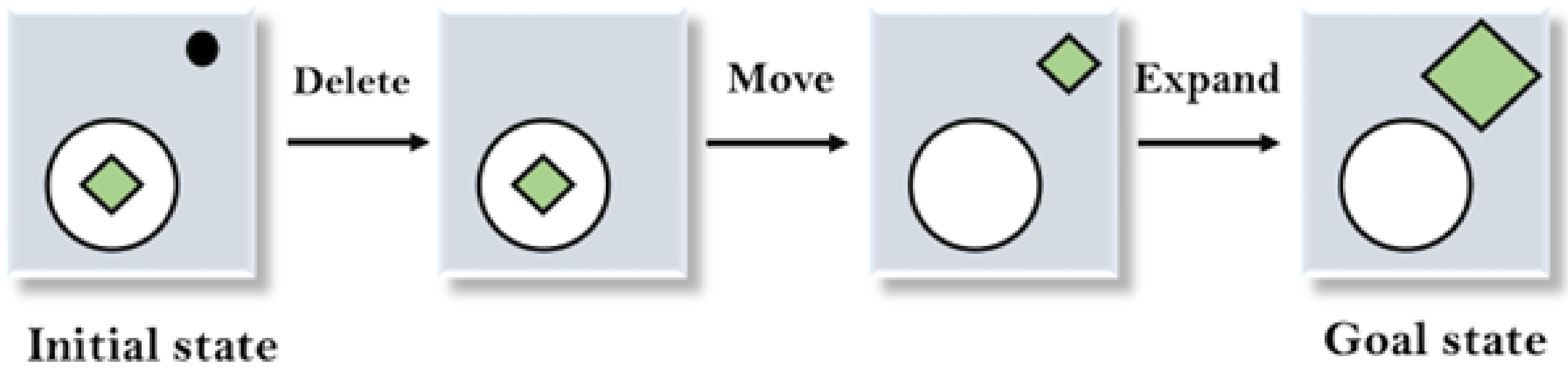
Cont..

3. Applying Move Operator: After applying the Delete operator, the new state occurs which we will again compare with goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the **Move Operator**.



Cont..

4. Applying Expand Operator: Now a new state is generated in the third step, and we will compare this state with the goal state. After comparing the states there is still one difference which is the size of the square, so, we will apply **Expand operator**, and finally, it will generate the goal state.



Thank You