



Informed Search

Informed Search

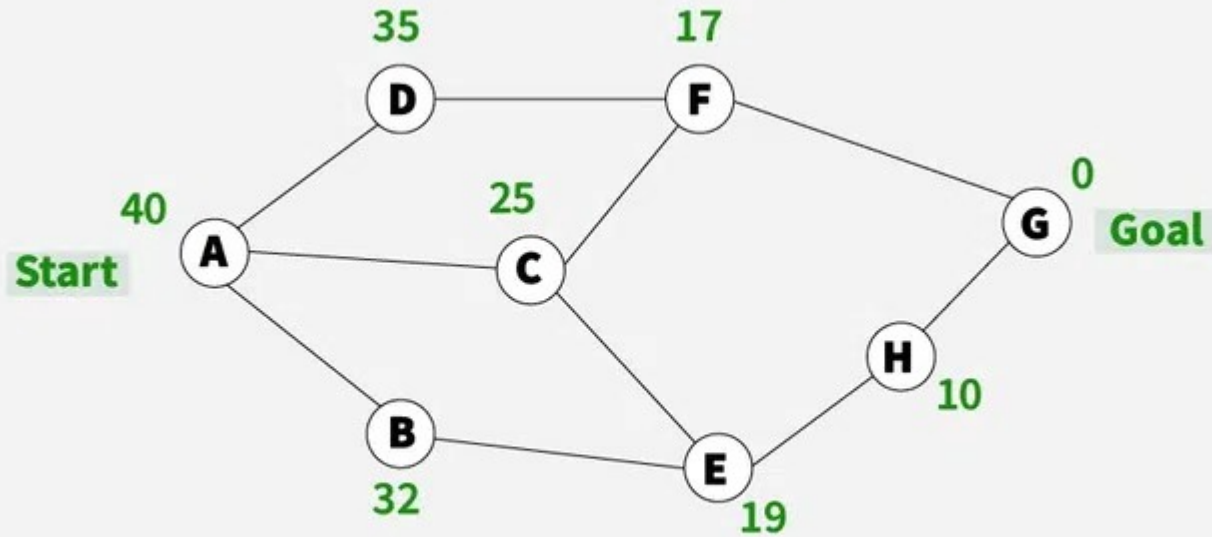
- The algorithms of an informed search contain information regarding the goal state.
- It helps an AI make more efficient and accurate searches.
- A function obtains this data/info to estimate the closeness of a state to its goal in the system.

1. Greedy First Search
2. A* Search
3. Hill-Climbing Search

Greedy First Search

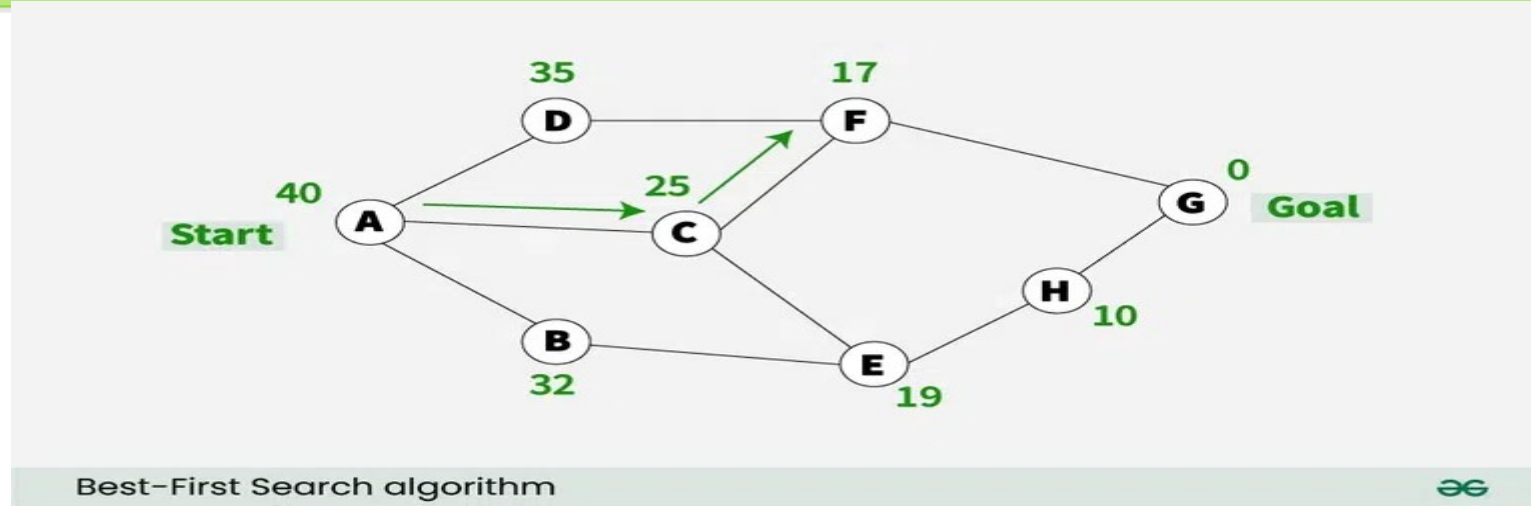
- Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a heuristic function to determine which path is the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

Greedy First Search - Example



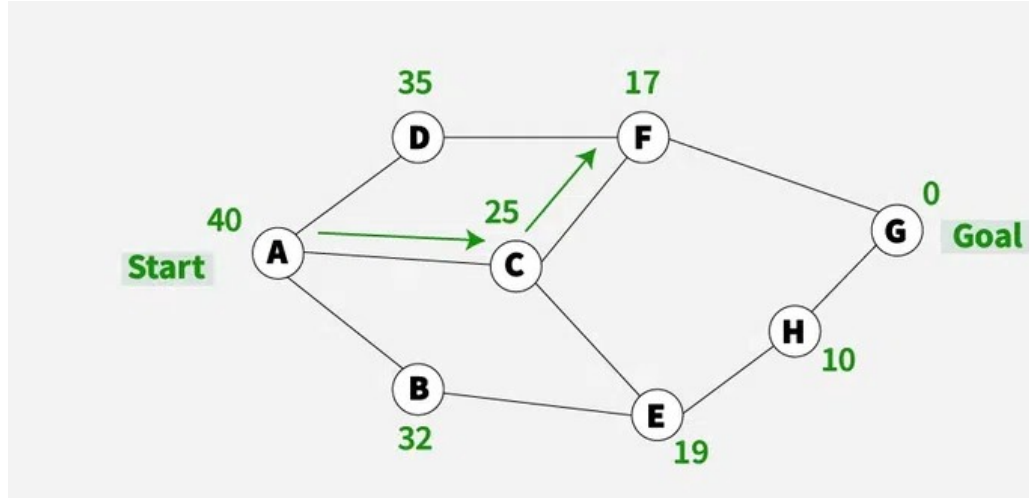
Best-First Search algorithm

Greedy First Search - solution



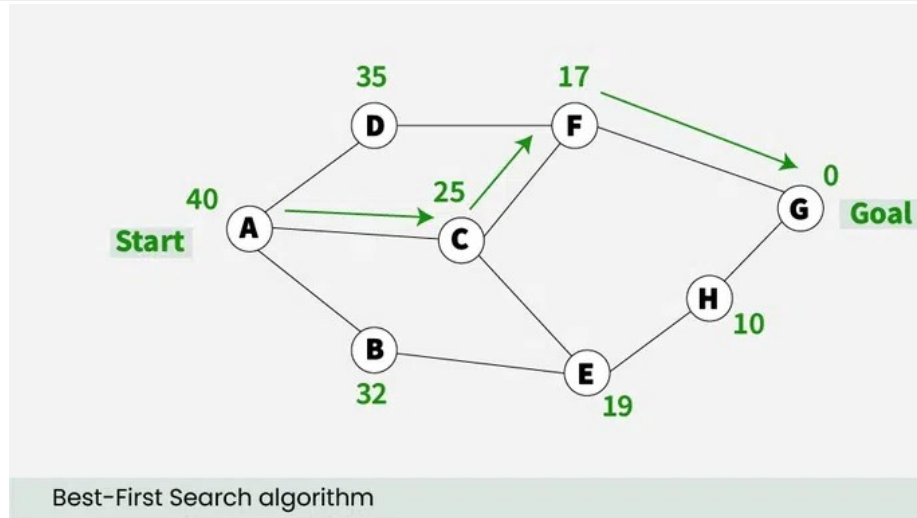
- From A there are direct path to node B(with heuristics value of 32), from A to C (with heuristics value of 25) and from A to D(with heuristics value of 35) choose the path with lowest heuristics value , currently C has lowest value among above node . **A to C**

Greedy First Search - solution



- from C we have direct paths as C to F(with heuristics value of 17) and C to E(with heuristics value of 19) , so we will go from C to F.

Greedy First Search - solution



- From F we have direct path to go to the goal node G (with heuristics value of 0)
- A->C->F->G .

Advantages of Greedy Best-First Search:

- Simple and Easy to Implement: Greedy Best-First Search is a relatively straightforward algorithm, making it easy to implement.
- Fast and Efficient: Greedy Best-First Search is a very fast algorithm, making it ideal for applications where speed is essential.
- Low Memory Requirements: Greedy Best-First Search requires only a small amount of memory, making it suitable for applications with limited memory.
- Flexible: Greedy Best-First Search can be adapted to different types of problems and can be easily extended to more complex problems.
- Efficiency: If the heuristic function used in Greedy Best-First Search is good to estimate, how close a node is to the solution, this algorithm can be a very efficient and find a solution quickly, even in large search spaces.

Disadvantages of Greedy Best-First Search:

- **Inaccurate Results:** Greedy Best-First Search is not always guaranteed to find the optimal solution, as it is only concerned with finding the most promising path.
- **Local Optima:** Greedy Best-First Search can get stuck in local optima, meaning that the path chosen may not be the best possible path.
- **Heuristic Function:** Greedy Best-First Search requires a heuristic function in order to work, which adds complexity to the algorithm.
- **Lack of Completeness:** Greedy Best-First Search is not a complete algorithm, meaning it may not always find a solution if one exists. This can happen if the algorithm gets stuck in a cycle or if the search space is too much complex.
-

Applications of Greedy Best-First Search:

- **Pathfinding:** Greedy Best-First Search is used to find the shortest path between two points in a graph. It is used in many applications such as video games, robotics, and navigation systems.
- **Machine Learning:** Greedy Best-First Search can be used in machine learning algorithms to find the most promising path through a search space.
- **Optimization:** Greedy Best-First Search can be used to optimize the parameters of a system in order to achieve the desired result.
- **Game AI:** Greedy Best-First Search can be used in game AI to evaluate potential moves and choose the best one.
- **Navigation:** Greedy Best-First Search can be used to navigate to find the shortest path between two locations.
- **Natural Language Processing:** Greedy Best-First Search can be used in natural language processing tasks such as language translation or speech recognition to generate the most likely sequence of words.
- **Image Processing:** Greedy Best-First Search can be used in image processing to segment an image into regions of interest.

Hill Climbing

- Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.
- Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem.
- This solution may not be the global optimal maximum.
- In the above definition, mathematical optimization problems implies that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs.

Hill Climbing

- Example- Travelling salesman problem where we need to minimize the distance traveled by the salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in reasonable time.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information.
- It helps the algorithm to select the best route out of possible routes.

Hill Climbing

- 1) **Variant of generate and test algorithm** : It is a variant of generate and test algorithm. The generate and test algorithm is as follows :
 1. Generate possible solutions.
 2. Test to see if this is the expected solution.
 3. If the solution has been found quit else go to step 1.
- Hence we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from the test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.
- 2) **Uses the Greedy approach** : At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

Hill Climbing

Types of Hill Climbing

- 1) Simple Hill climbing
- 2) Steepest-Ascent Hill climbing
- 3) Stochastic hill climbing

Simple Hill climbing

Simple Hill climbing : It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.

Algorithm for Simple Hill climbing :

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

Step 2 : Loop until the solution state is found or there are no new operators present which can be applied to the current state.

- a) Select a state that has not been yet applied to the current state and apply it to produce a new state.
- b) Perform these to evaluate new state
 - i. If the current state is a goal state, then stop and return success.
 - ii. If it is better than the current state, then make it current state and proceed further.
 - iii. If it is not better than the current state, then continue in the loop until a solution is found.

Step 3 : Exit.

Steepest-Ascent Hill climbing

It first examines all the neighboring nodes and then selects the node closest to the solution state as of next node.

Algorithm

Step 1 : Evaluate the initial state. If it is goal state then exit else make the current state as initial state

Step 2 : Repeat these steps until a solution is found or current state does not change

- i. Let 'target' be a state such that any successor of the current state will be better than it;
- ii. for each operator that applies to the current state
 - a. apply the new operator and create a new state
 - b. evaluate the new state
 - c. if this state is goal state then quit else compare with 'target'
 - d. if this state is better than 'target', set this state as 'target'
 - e. if target is better than current state set current state to Target

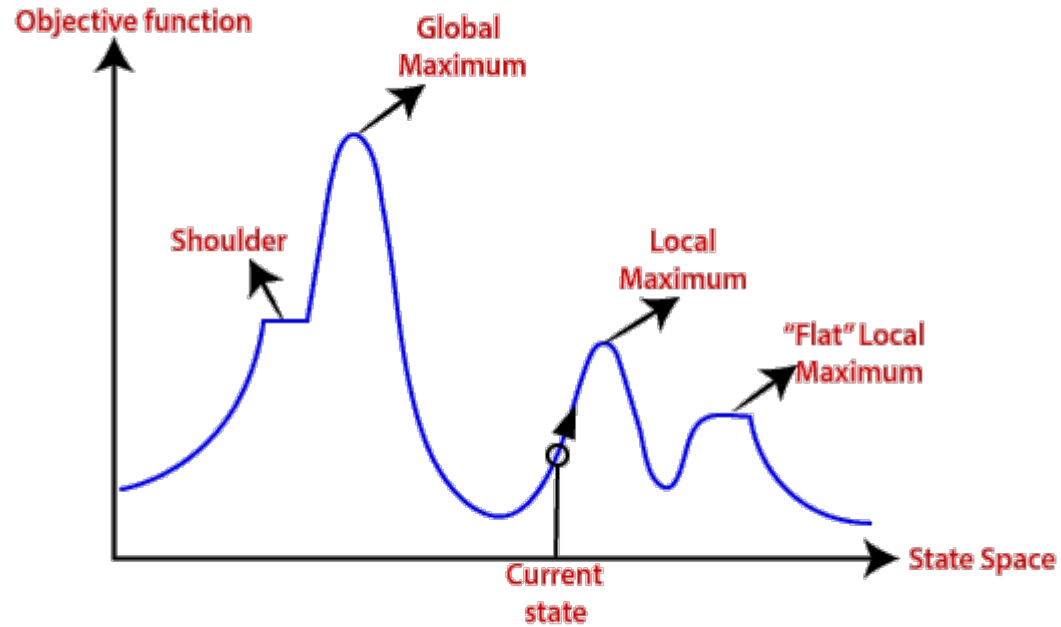
Step 3 : Exit

3) Stochastic hill climbing

- It does not examine all the neighboring nodes before deciding which node to select.
- It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

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 ie 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).



A one-dimensional state-space landscape in which elevation corresponds to the objective function

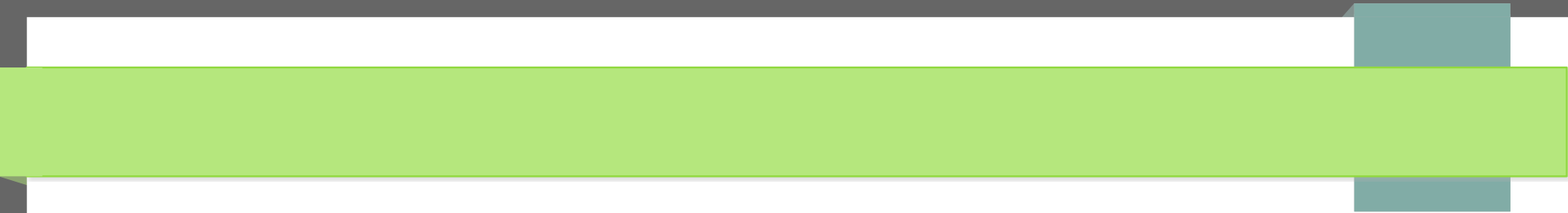
Different regions in the State Space Diagram

1. Local maximum: It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.

2.Global maximum : It is the best possible state in the state space diagram. This because at this state, objective function has highest value.

3.Plateau /flat local maximum : It is a flat region of state space where neighboring states have the same value.

4.Ridge : It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.

- 
- **5.Current state** : The region of state space diagram where we are currently present during the search.
 - **6.Shoulder** : It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

1) Local maximum : At a local maximum all neighboring states have a values which is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.

To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.



2) **Plateau** : On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

To overcome plateaus : Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non- plateau region

3) **Ridge** : Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.

To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

A* Algorithm

- A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
- A* (pronounced "A-star") is a powerful **graph traversal and pathfinding** algorithm widely used in artificial intelligence and computer science.
- It is mainly used to **find the shortest path between two nodes** in a graph, given the estimated cost of getting from the current node to the destination node.
- Algorithm A* combines the advantages of two other search algorithms: **Dijkstra's algorithm and Greedy Best-First Search**.
- Like Dijkstra's algorithm, A* ensures that the path found is as short as possible but does so more efficiently by directing its search through a heuristic similar to Greedy Best-First Search.

A* Algorithm

- A heuristic function, denoted $h(n)$, estimates the cost of getting from any given node n to the destination node.
- The main idea of A* is to evaluate each node based on two parameters:
- **$g(n)$** : the actual cost to get from the initial node to node n . It represents the sum of the costs of node n outgoing edges.
- **$h(n)$** : Heuristic cost (also known as "estimation cost") from node n to destination node n .
- $f(n) = g(n) + h(n)$.

A* Algorithm

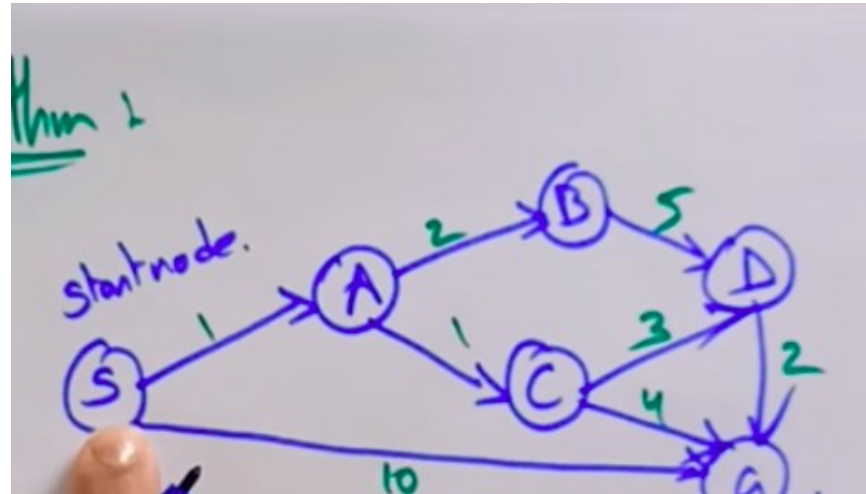
1. Create an **open list** of found but not explored nodes.
2. Create a **closed list** to **hold already explored nodes**.
3. Add a starting node to the open list with an initial value of g
4. **Repeat the following steps** until the open list is empty or you reach the target node:
 - a) Find the **node with the smallest f-value** (i.e., the node with the minor $g(n) + h(n)$) in the open list.
 - b) Move the selected node from the open list to the closed list.
 - c) Create all valid descendants of the selected node.

A* Algorithm

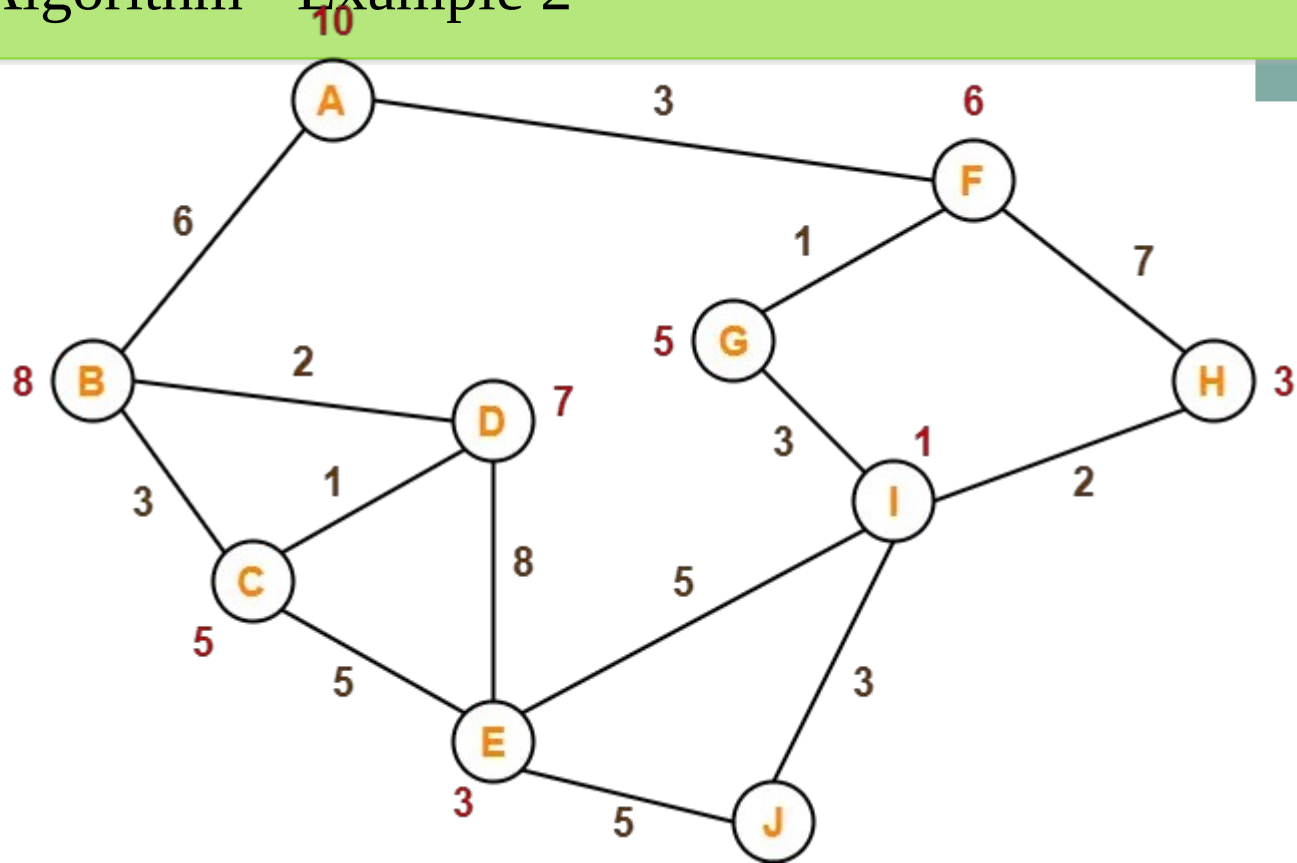
- d) For each successor, calculate its **g-value** as the sum of the current node's g value and the cost of moving from the current node to the successor node.
- e) If the follower is not in the open list, add it with the calculated g-value and **calculate its h-value**.
- f) Repeat the cycle.
- Algorithm A* terminates when the target node is reached or when the open list empties, indicating no paths from the start node to the target node.
- The A* search algorithm is widely used in various fields such as robotics, video games, network routing, and design problems because it is efficient and can find optimal paths in graphs or networks.

A* Algorithm – Example 1

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



A* Search Algorithm - Example 2



Find the most cost-effective path to reach from start state **A** to final state **J** using A* Algorithm.

Solution-

Step-01:

- We start with node A.
- Node B and Node F can be reached from node A.

A* Algorithm calculates $f(B)$ and $f(F)$.

- $f(B) = 6 + 8 = 14$
- $f(F) = 3 + 6 = 9$

Since $f(F) < f(B)$, so it decides to go to node F.

Path- A → F

Step-02:

Node G and Node H can be reached from node F.

A* Algorithm calculates $f(G)$ and $f(H)$.

- $f(G) = (3+1) + 5 = 9$
- $f(H) = (3+7) + 3 = 13$

Since $f(G) < f(H)$, so it decides to go to node G.

Path- A \rightarrow F \rightarrow G

Step-03:

Node I can be reached from node G.

A* Algorithm calculates $f(I)$.

- $f(I) = (3+1+3) + 1 = 8$

It decides to go to node I.

Path- $A \rightarrow F \rightarrow G \rightarrow I$

Step-04:

Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates $f(E)$, $f(H)$ and $f(J)$.

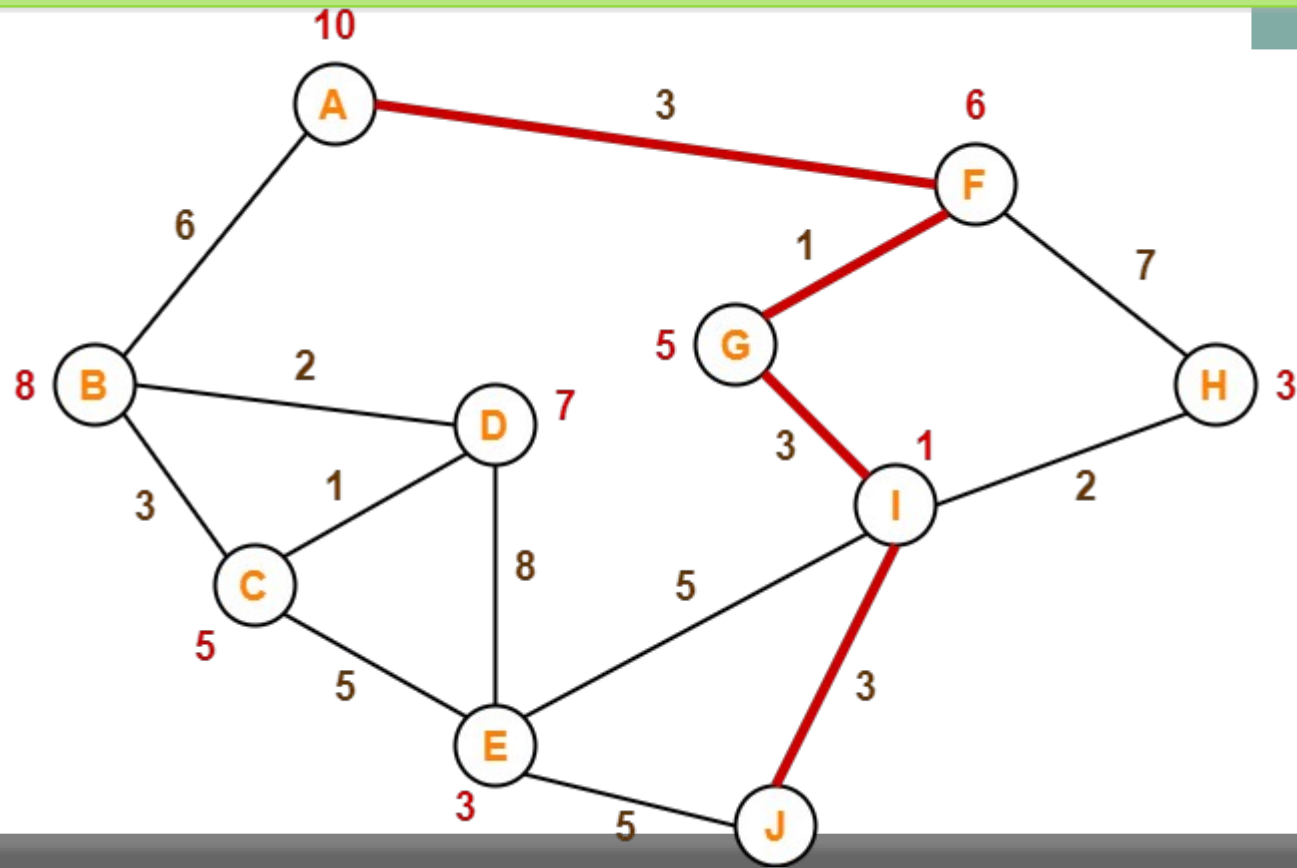
- $f(E) = (3+1+3+5) + 3 = 15$
- $f(H) = (3+1+3+2) + 3 = 12$
- $f(J) = (3+1+3+3) + 0 = 10$

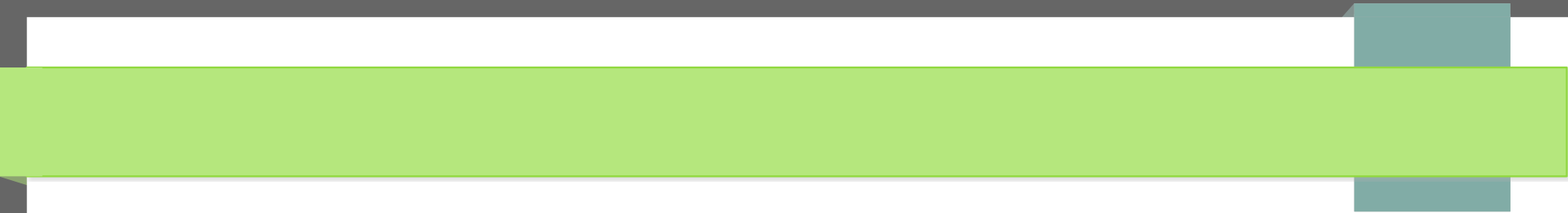
Since $f(J)$ is least, so it decides to go to node J.

Path- A → F → G → I → J

This is the required shortest path from node A to node J.

Path - Output



- 
- 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.

- **Video link for Reference :**

<https://www.youtube.com/watch?v=tvAh0JZF2YE>