VNUHCM

University Of Science

PROJECT REPORT

Introduction to Artificial Intelligence

Lab 1: Searching

Student:	21127702 - Bùi Nguyễn Tin
Course section:	CSC14003 - 22CLC08
Faculty advisor:	Bùi Tiến Lên
	Phạm Trọng Nghĩa
	Nguyễn Thanh Tình

Ho Chi Minh City – 2024

ASSESSMENT

No.	Details	Self assessment
01	Implement BFS correctly.	10%
02	Implement DFS correctly.	10%
03	Implement UCS correctly.	10%
04	Implement IDS correctly.	10%
05	Implement GBFS correctly.	10%
06	Implement A* correctly.	10%
07	Implement Hill-climbing correctly.	10%
08	Generate at least 5 test cases for all algorithms with different attributes. Describe them in the experiment section of your report.	10%
09	Report your algorithm, experiment with some reflection or comments.	20%
Total		100%

TABLE OF CONTENTS

Contents

1. Algorithms

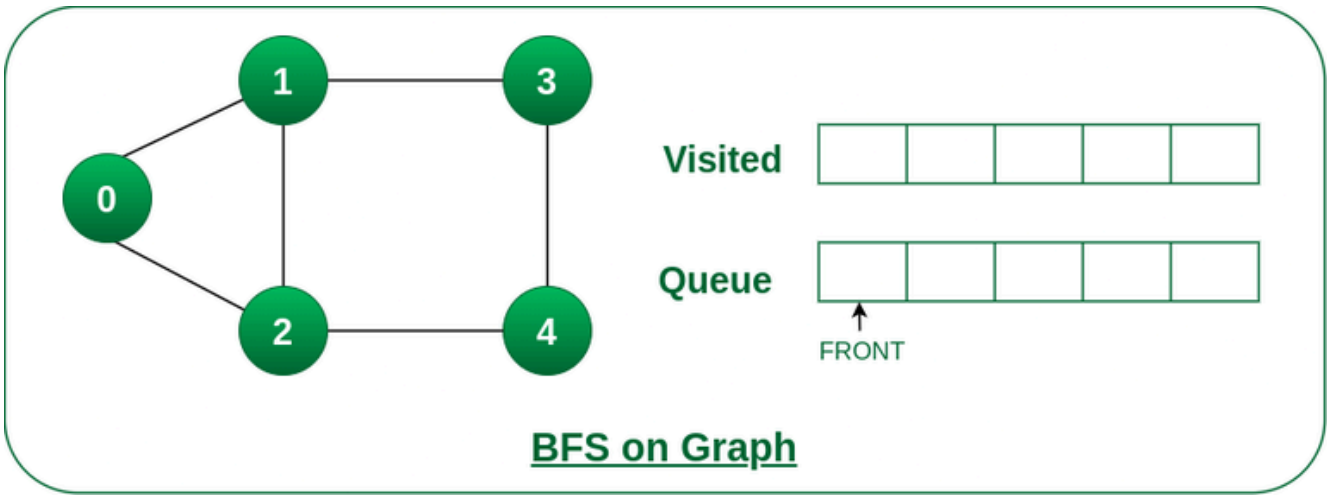
1.1. Breadth-first search (BFS)

1.1.1. Description

Breadth-First Search (BFS) is a graph traversal algorithm that explores a graph level by level. It starts at a designated starting node and systematically visits all its neighbors at the same depth before moving on to the next level of neighbors.

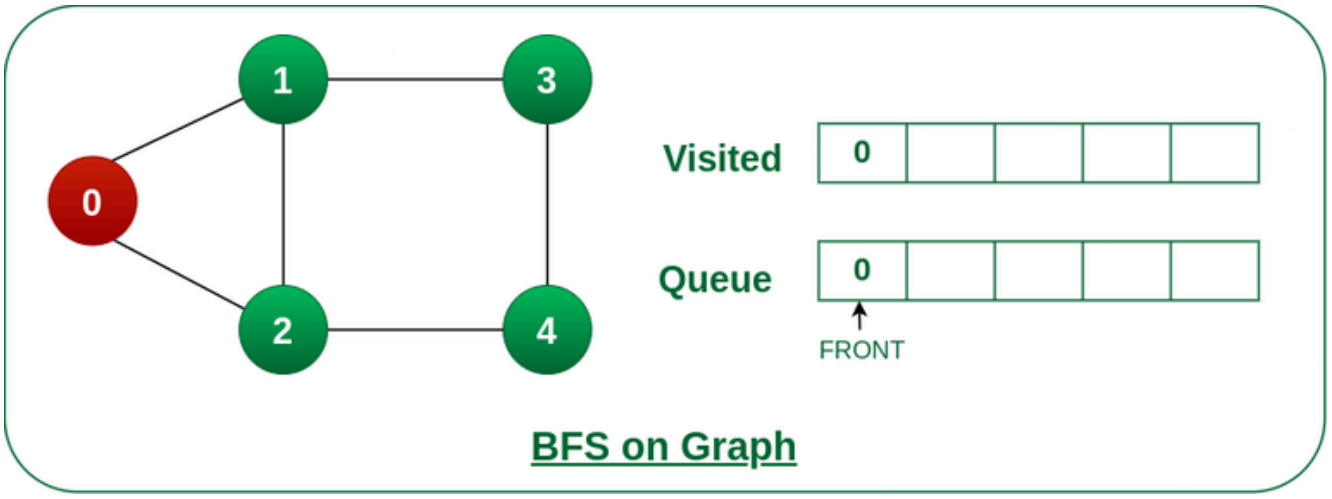
1.1.2. Steps of execution

- **Step1:** Initially the queue and visited arrays are empty.



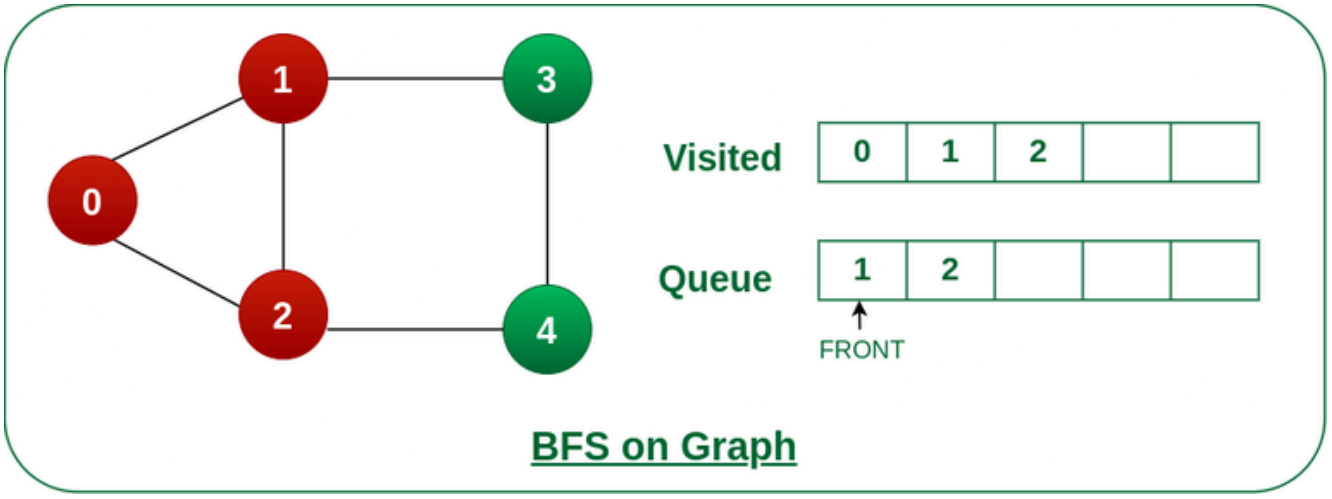
Queue and visited arrays are empty initially.

- **Step2:** Push node 0 into the queue and mark it visited.



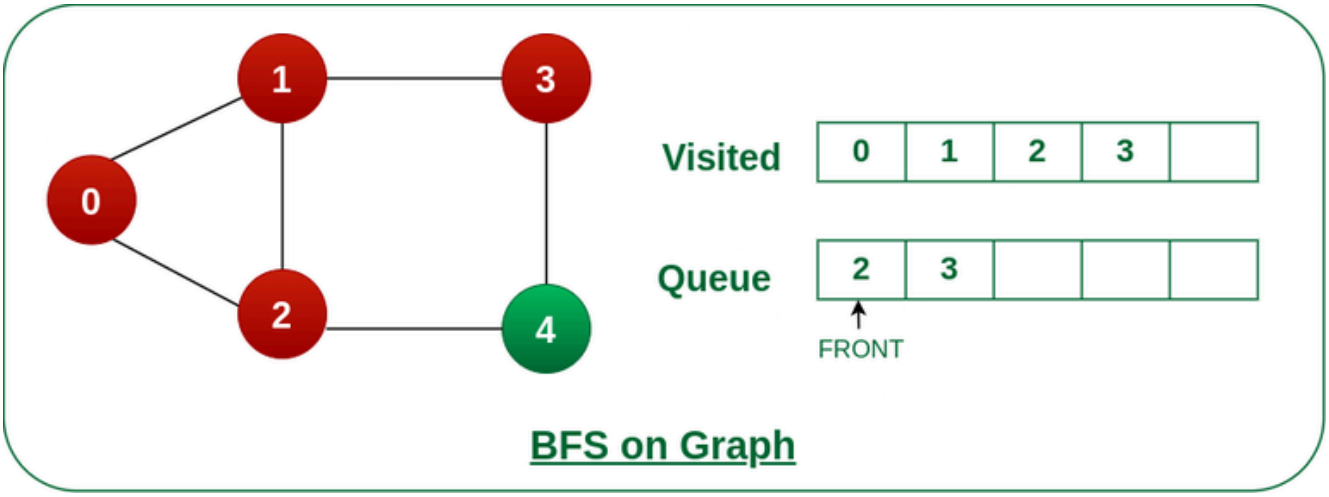
Push node 0 into the queue and mark it visited.

- **Step 3:** Remove node 0 from the front of the queue and visit the unvisited neighbors and push them into the queue.



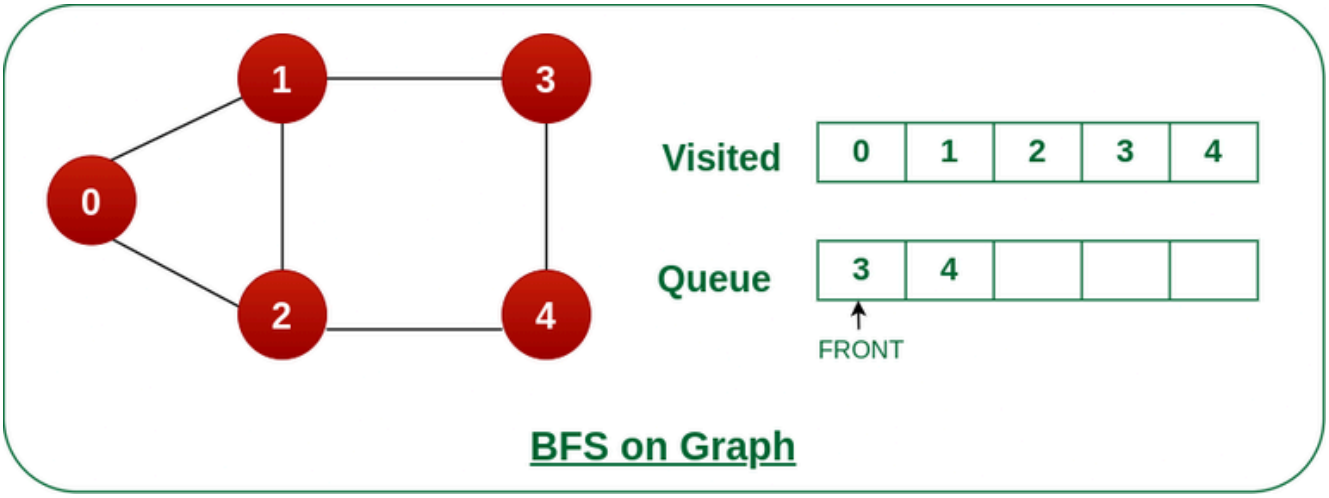
Remove node 0 from the front of queue and visited the unvisited neighbors and push into queue.

- **Step 4:** Remove node 1 from the front of the queue and visit the unvisited neighbors and push them into the queue.



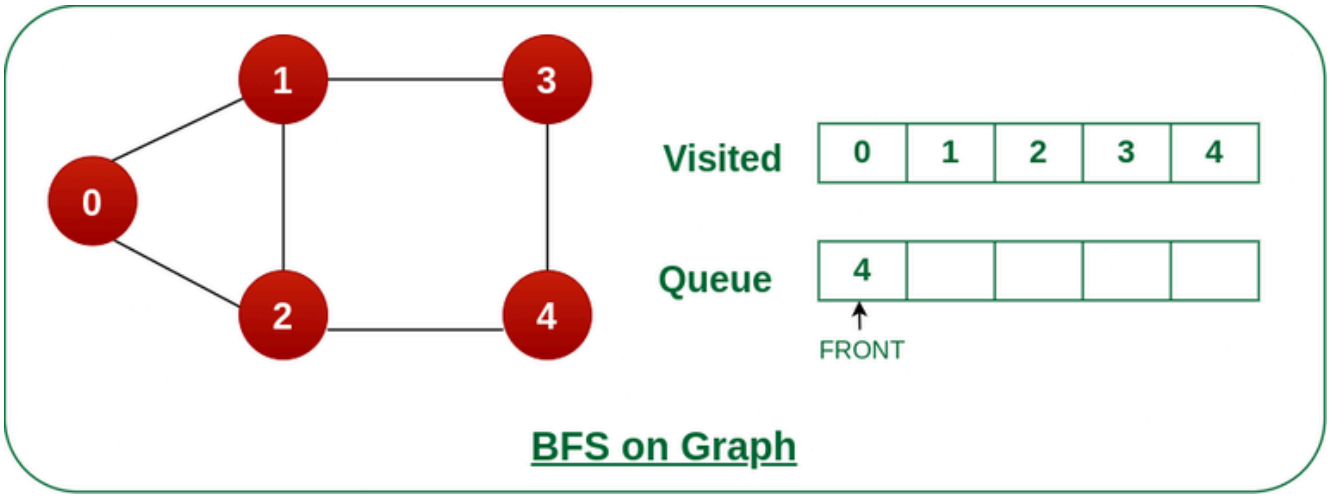
Remove node 1 from the front of queue and visited the unvisited neighbors and push

- **Step 5:** Remove node 2 from the front of the queue and visit the unvisited neighbors and push them into the queue.



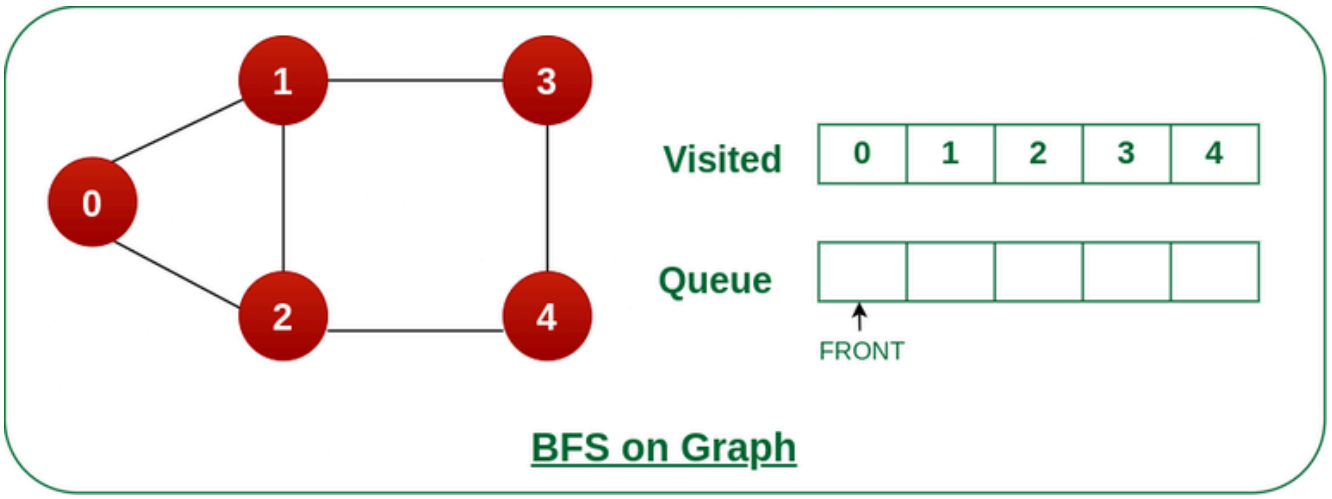
Remove node 2 from the front of queue and visit the unvisited neighbors and push them into the queue.

- **Step 6:** Remove node 3 from the front of the queue and visit the unvisited neighbors and push them into the queue. As we can see that every neighbor of node 3 is visited, so move to the next node that is in the front of the queue.



Remove node 3 from the front of the queue and visit the unvisited neighbors and push them into the queue.

- **Step 7:** Remove node 4 from the front of the queue and visit the unvisited neighbors and push them into the queue. As we can see that every neighbor of node 4 is visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of the queue and visit the unvisited neighbors and push them into the queue.

Now, Queue becomes empty, So, terminate this process of iteration.

1.2. Depth-first search (DFS)

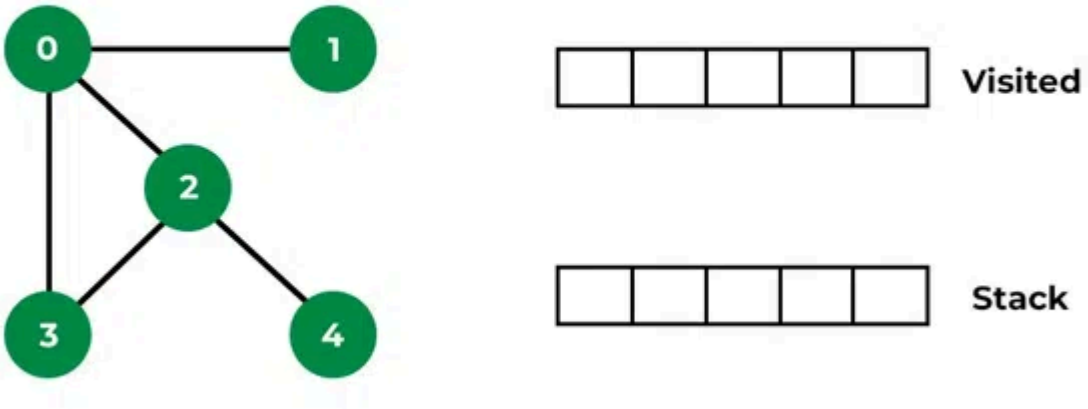
1.2.1. Description

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It systematically visits all reachable nodes in a graph,

starting from a given source node. DFS is similar to the Depth First Traversal of a tree, but it handles cycles in graphs by using a visited array to prevent revisiting nodes.

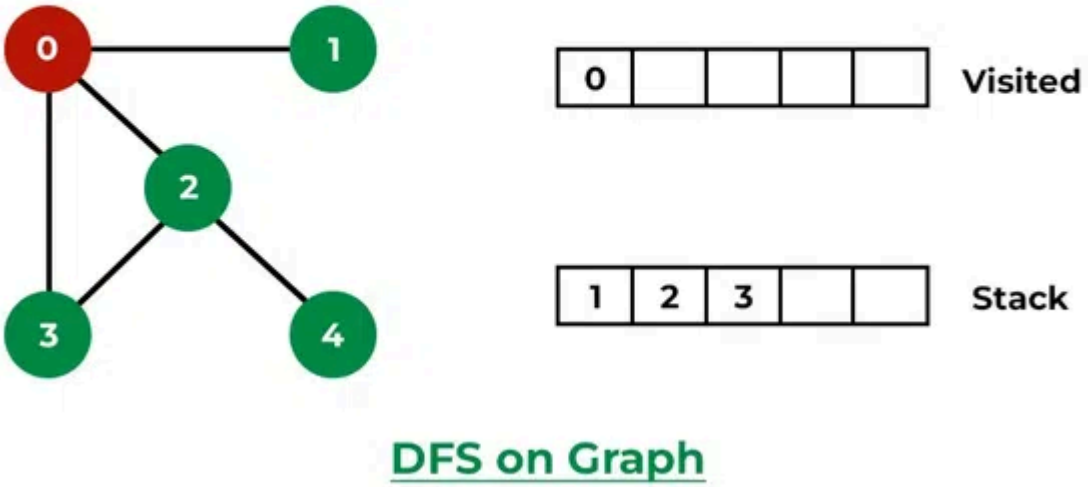
1.2.2. Steps of execution

- **Step1:** Initially stack and visited arrays are empty.



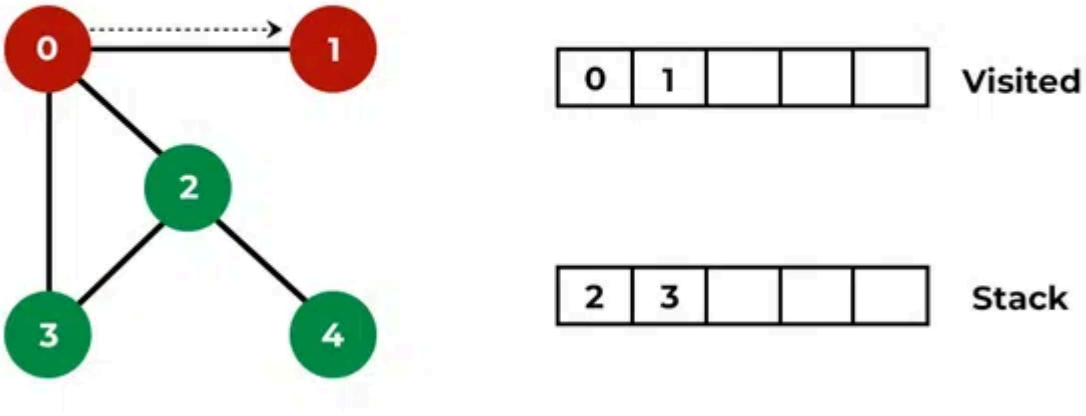
Stack and visited arrays are empty initially.

- **Step 2:** Visit 0 and put its adjacent nodes which are not visited yet into the stack.



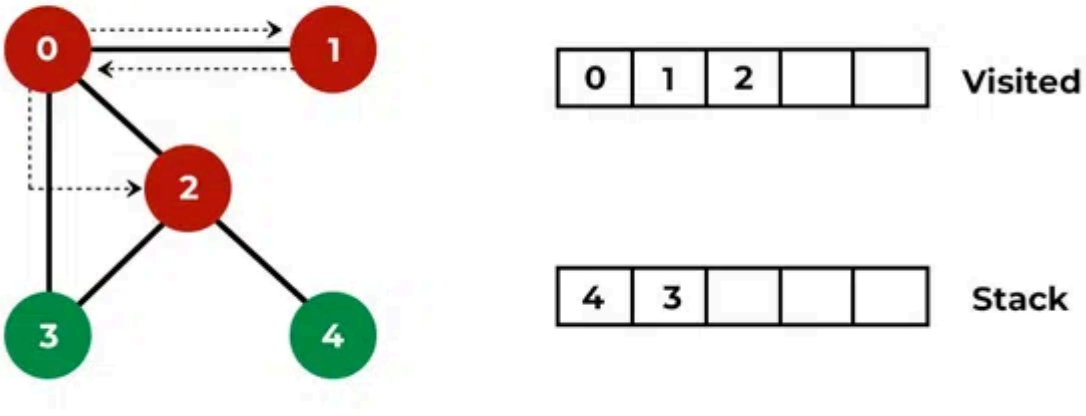
Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

- **Step 3:** Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



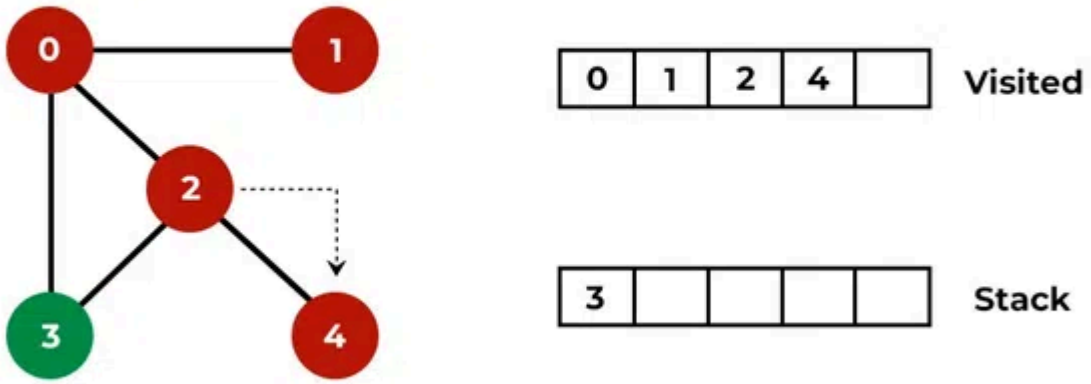
Visit node 1

- **Step 4:** Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



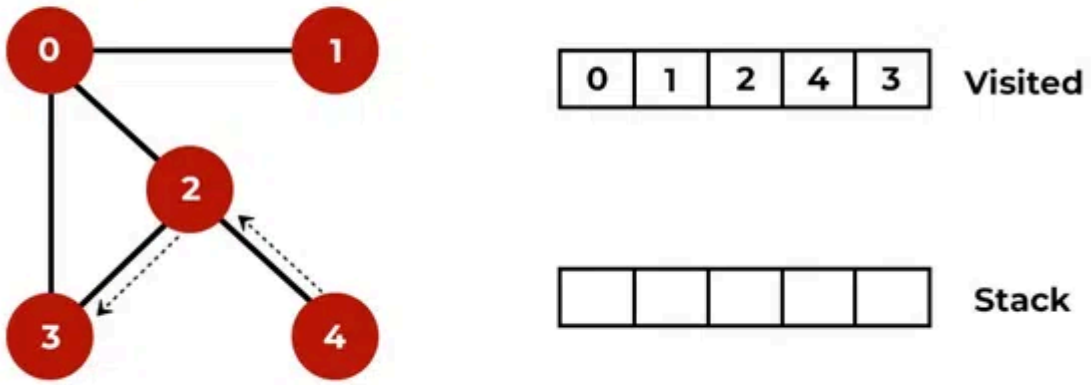
Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

- **Step 5:** Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Visit node 4

- **Step 6:** Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Visit node 3

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

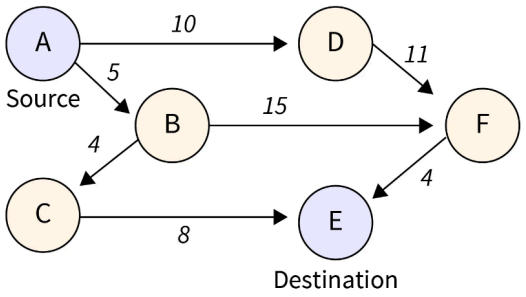
1.3. Uniform-cost search (UCS)

1.3.1. Description

The Uniform Cost Search (UCS) algorithm is a search algorithm used to find the shortest path from a starting node to a goal node in a weighted graph. It is a variation of Dijkstra's algorithm and is considered an uninformed search algorithm because it does not use any prior knowledge about the graph or the goal node.

1.3.2. Steps of execution

- **Input:** Let the graph be as below with source node being A and destination E.



- **Output**: 17
- The shortest path here is $A -> B -> C -> E$ forming the cost 17.

We will have an empty priority queue and a boolean visited array. We will insert A with cost 0 into the queue.

Queue:

A - 0					

visited:

A	B	C	D	E	F
False	False	False	False	False	False

- **Step 1:**
  - Pop A from the queue.
  - A is not destination node.
  - Cost of A = 0, cst = 0.
  - Append B and D to the queue with the costs cst + 10 and cst + 5.
  - And mark A as visited.

Queue:

B - 5	D - 10				

visited:

A	B	C	D	E	F
True	False	False	False	False	False

- **Step 2:**
  - Pop B from the Queue. As the cost of B is less than D, it appears first in the queue.
  - B is not the destination node.
  - Cost of B = 5, cst = 5.
  - We will append C with cost 4 + cst and F with cost 15 + cst to the queue.
  - Mark B has visited.

Queue:

C - 9	D - 10	F - 20			

A	B	C	D	E	F
True	True	False	False	False	False

- **Step 3:**
  - Similarly, C is popped, and E is pushed in the stack.
  - Similarly, D is popped, and F is pushed into the stack as it is still not visited.

Queue:

E - 17	F - 20	F - 21			

visited:

A	B	C	D	E	F
True	True	True	True	False	False

- **Step 4:**
  - Pop E from the Queue. As the cost of E is least it appears first in the queue.
  - E is the destination node.
  - Cost of E = 17, cst = 17.
  - The initial value of min_cost is Infinity or Integer.MAX_VALUE. Thus, min_cost = cst = 17 is updated.
  - We also don't push any adjacent nodes.
  - As E is the destination node we don't mark it visited.

Queue:

F - 20	F - 21				

visited:

A	B	C	D	E	F
True	True	True	True	True	False

- **Step 5:**
  - Pop F from the Queue.
  - F is not the destination node.
  - Cost of F = 20, cst = 20.
  - F has the outgoing edge to E. We will push it to the queue with the cost 4 + cst.
  - Mark F has visited.

Queue:

F - 21	E - 24				

visited:

A	B	C	D	E	F
True	True	True	True	False	True

- **Step 6:**
  - Pop F from the Queue.
  - F is not the destination node.
  - Cost of F = 21, cst = 21.
  - F has the outgoing edge to E. We will push it to the queue with the cost 4 + cst.
  - Mark F has visited.

Queue:

E - 24	E - 25				

visited:

A	B	C	D	E	F
True	True	True	True	False	True

- **Step 7:**
  - Pop E from the Queue.
  - E is the destination node.
  - Cost of E = 24 cst = 24.
  - The min_cost = 17 and the current cost is 24. Thus, we are not required to update any values.
  - Again Pop E from the Queue.
  - The min_cost = 17 and the current cost is 25. Thus, we are not required to update any values.
  - As the queue is now empty and the value of min_cost is not infinity, we found the node and now we will print the value in the output.

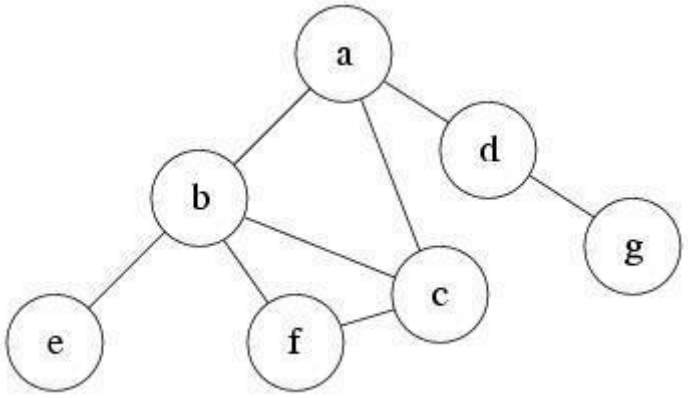
1.4. Depth limited search (DLS)

1.4.1. Description

Depth-Limited Search (DLS) is a search algorithm used in artificial intelligence that improves upon Depth-First Search (DFS) by introducing a depth limit, effectively preventing the algorithm from exploring excessively deep paths and improving search efficiency. The article explains how DLS works, explores its applications in pathfinding robotics, network routing, and puzzle solving, and illustrates its use with a Python code example. Finally, it addresses frequently asked questions regarding DLS, such as its strengths, weaknesses, and the process of choosing an appropriate depth limit.

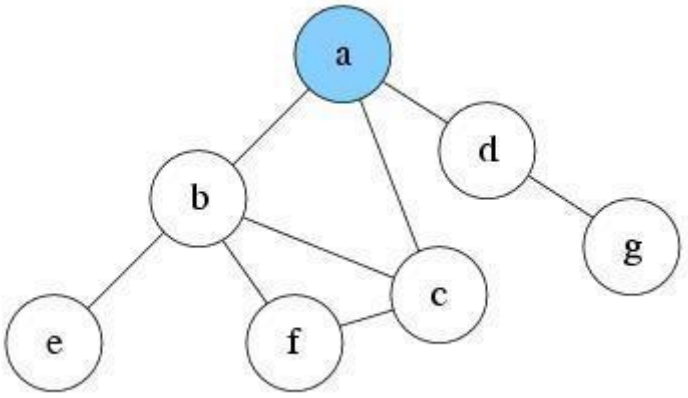
1.4.2. Steps of execution

- Depth limit = 1. Same as DFS, we use the stack data structure S1 to record the node we've explored. Suppose the source node is node a.



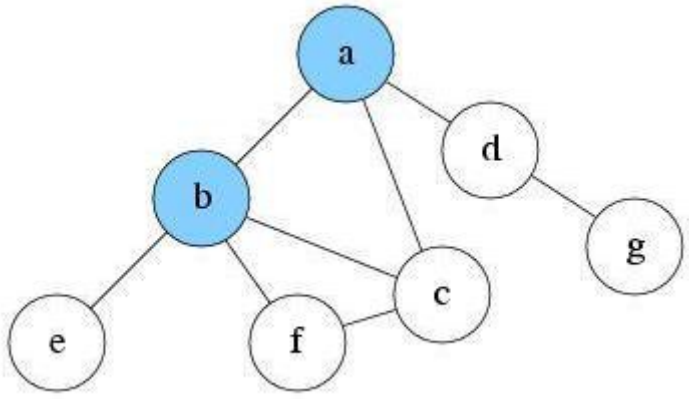
S1:

At first, the only reachable node is a. So push it into S1 and mark as visited. Current level is 0.



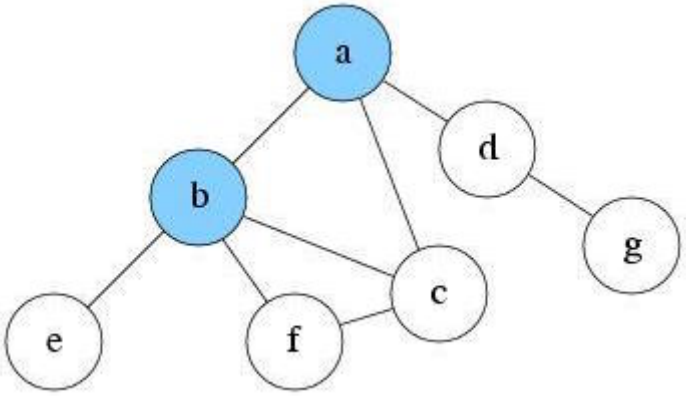
S1: a

After exploring a, now there are three nodes reachable: node b, c and d. Suppose we pick node b to explore first. Push b into S1 and mark it as visited. Current level is 1.



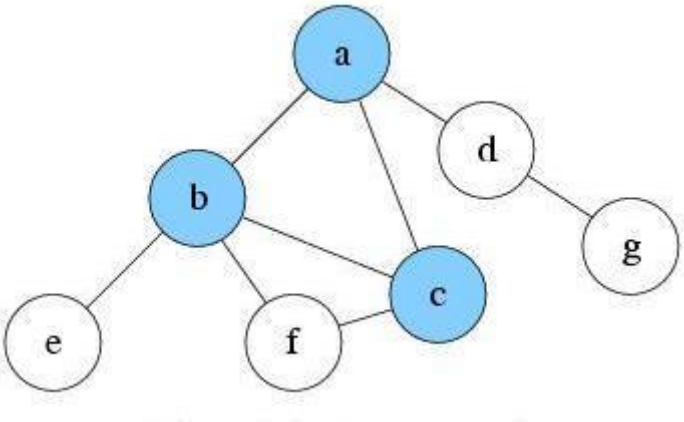
S1: b, a

Since current level is already the max depth L. Node b will be treated as having no successor. So there is nothing reachable. Pop b from S1. Current level is 0.



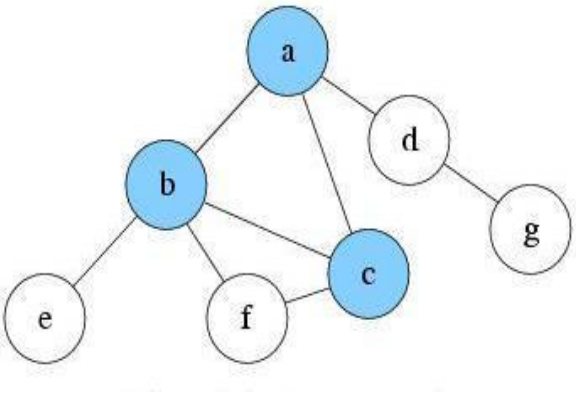
S1: a

Explore again. There are two unvisited nodes c and d that are reachable. Suppose we pick node c to explore first. Push c into S1 and mark it as visited. Current level is 1.



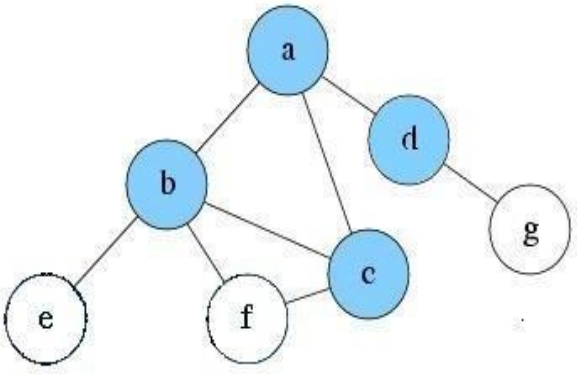
S1: c, a

Since current level is already the max depth L. Node c will be treated as having no successor. So there is nothing reachable. Pop c from S1. Current level is 0.



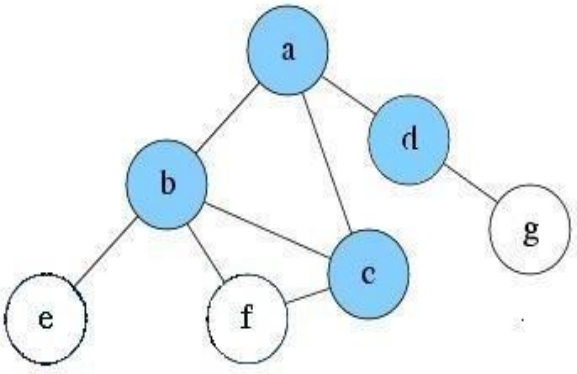
S1: a

Explore again. There is only one unvisited node d reachable. Push d into S1 and mark it as visited. Current level is 1.



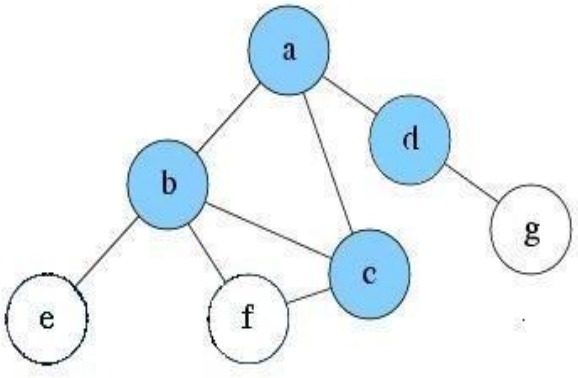
S1: d, a

Explore d and find no new node is reachable. Pop d from S1. Current level is 0.



S1: a

Explore a again. No new reachable node. Pop a from S1.



S1:
S1 is empty now. DLS will be finished.

1.5. Iterative deepening search (IDS)

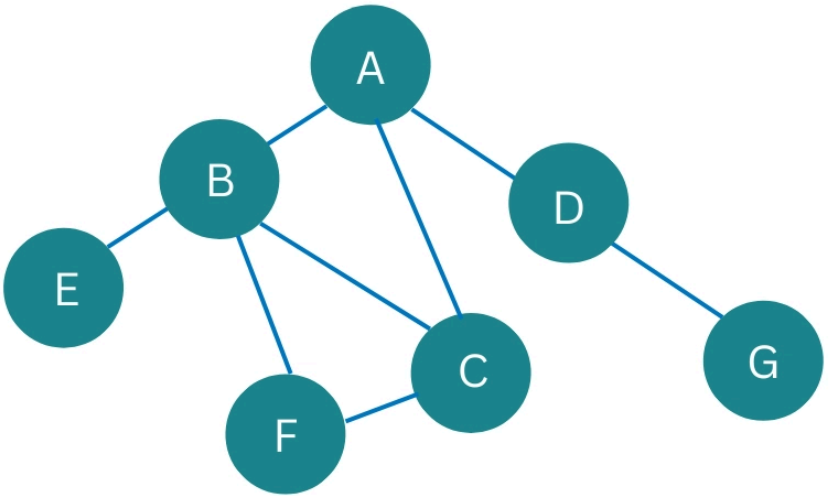
1.5.1. Description

Iterative Deepening Search (IDS) is a search algorithm used in artificial intelligence that combines the advantages of both Depth-First Search (DFS) and Breadth-First Search (BFS). It is particularly useful for finding the shortest path to a goal in a state space where the depth of the solution is unknown.

1.5.2. Steps of execution

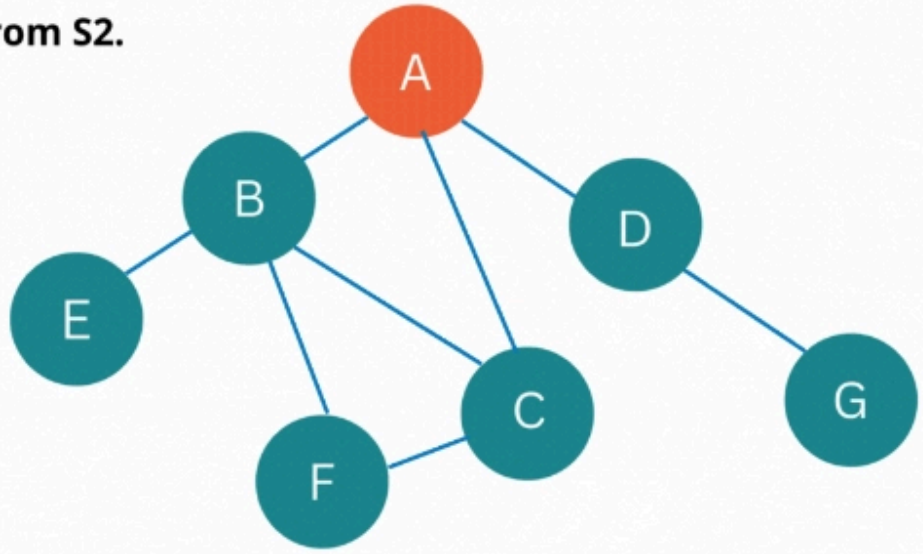
In this Graph, we use the stack data structure S to keep track of the nodes we've visited.

- Assume node 'A' is the source node.
- Assume that node 'D' is the solution node.

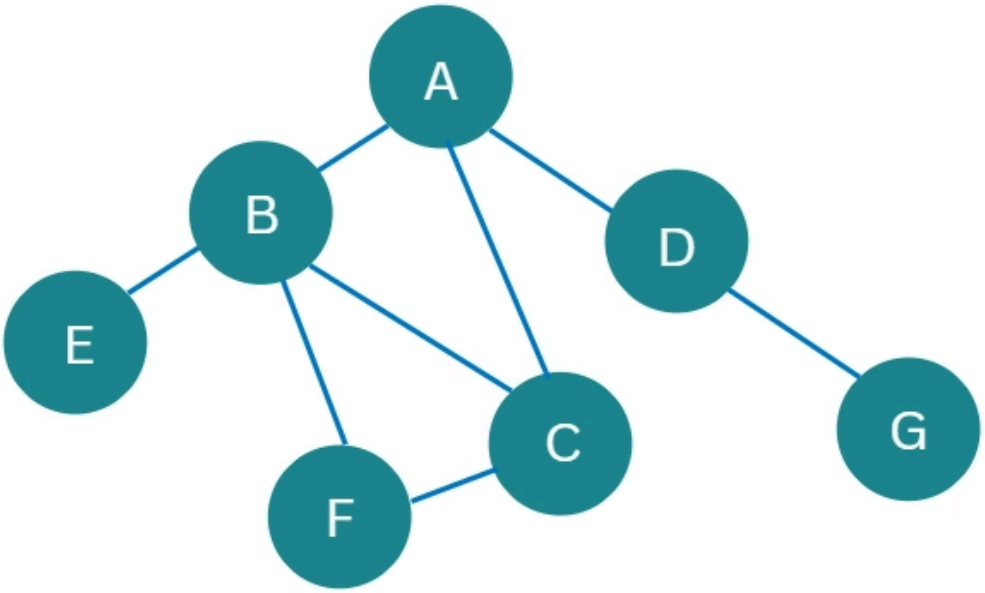


At first, the depth limit L =0. Only the node is reachable. Place it in S1 and mark it as visited. The current value is zero.

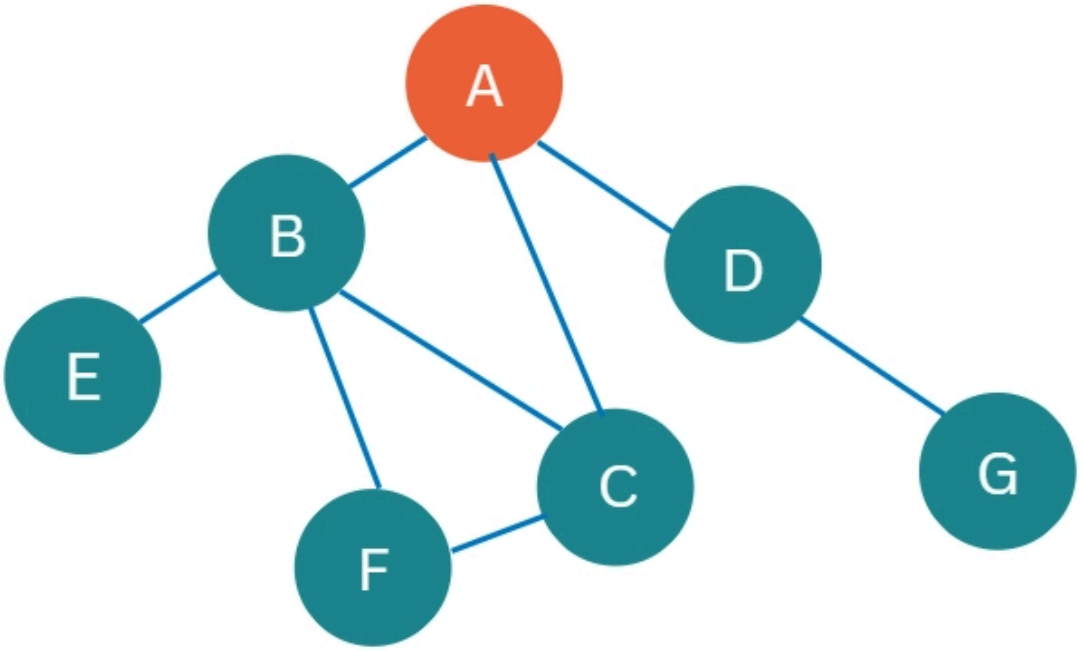
- **S2: A**
- **Explore A.**
- **Because the current level is already at the maximum depth L.**
- **There will be no new reachable nodes discovered.**
- **Take A from S2.**



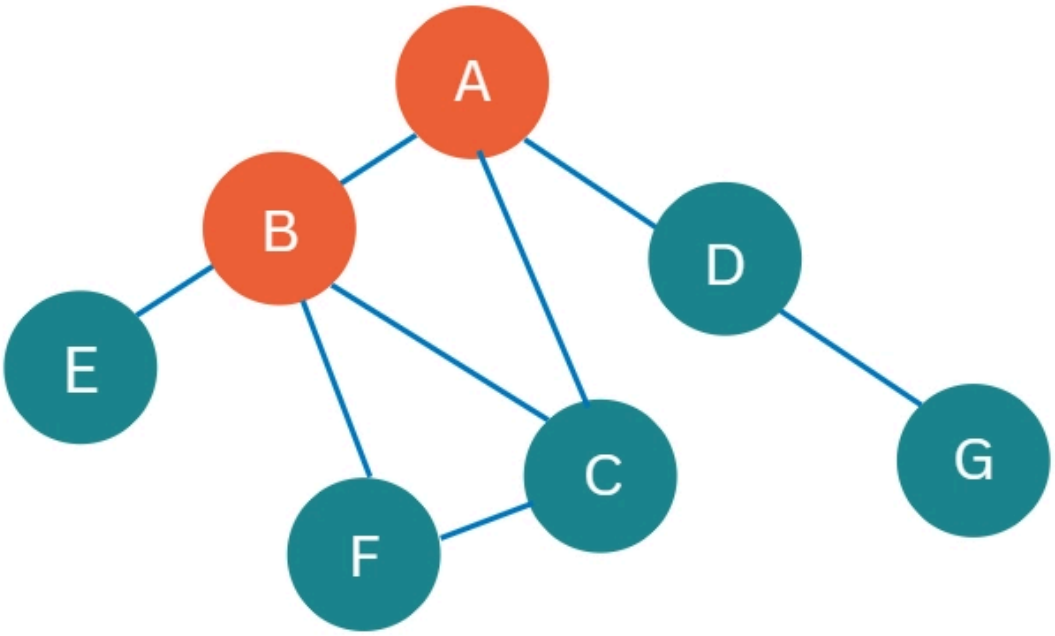
- S2 is now empty. Since no solution has been found and the maximum depth has not been reached, set the depth limit L to 1 and restart the search from the beginning.



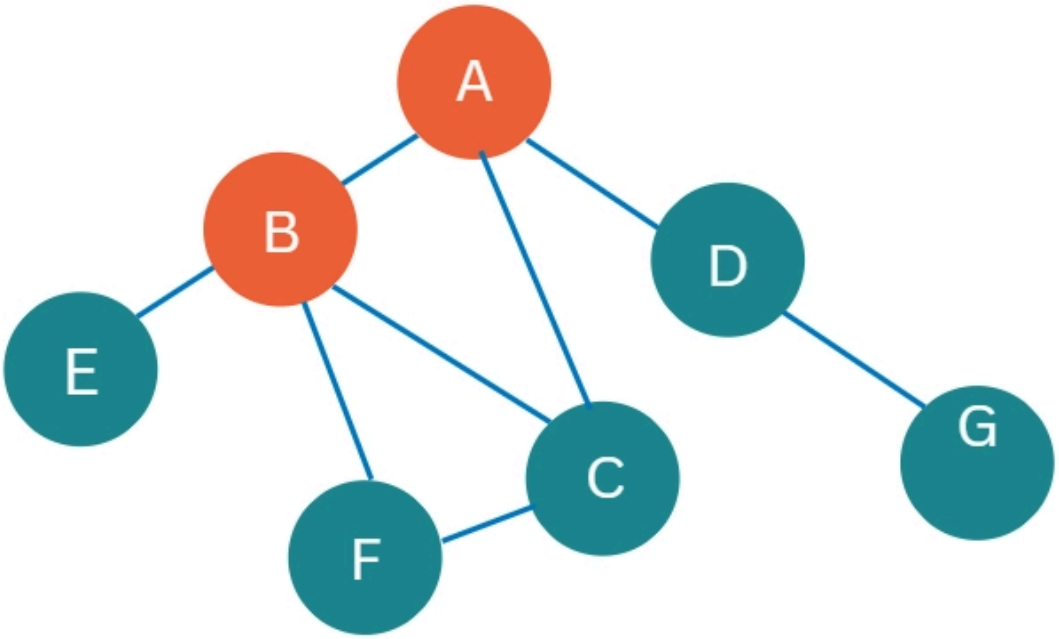
- S2:
  - Initially, only node A is reachable. So put it in S2 and mark it as visited.
  - The current level is 0.



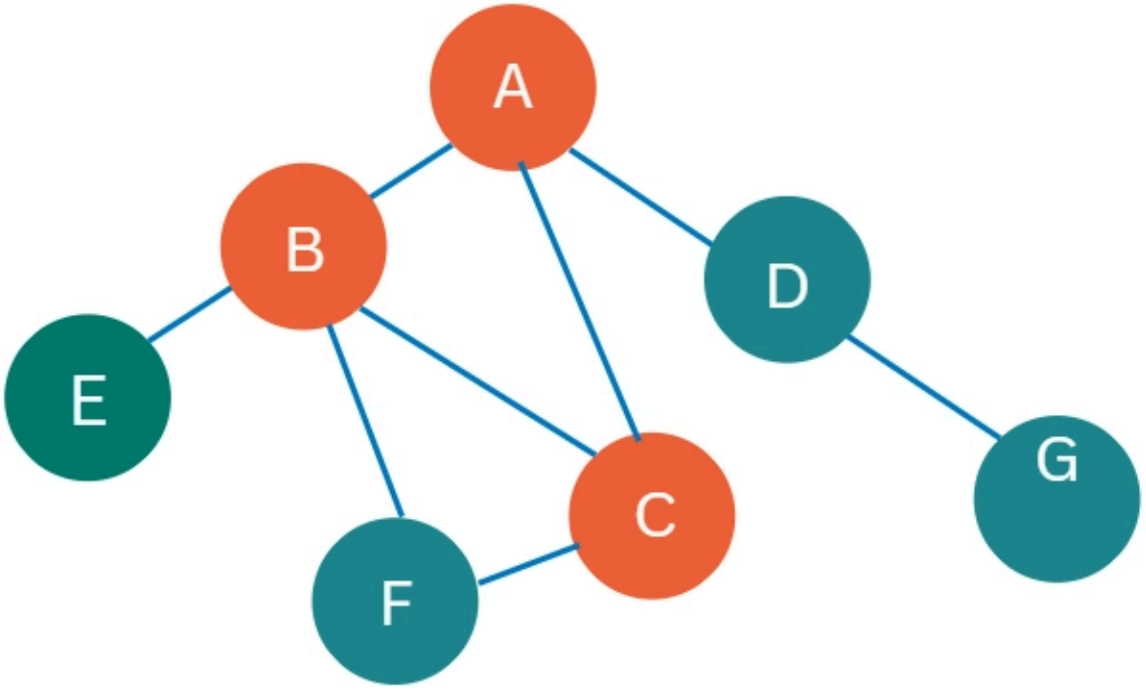
- S2: A
  - After exploring A, three nodes are now accessible: B, C, and D.
  - Assume we begin our exploration with node B.
  - B should be pushed into S2 and marked as visited.
  - The current level is one.



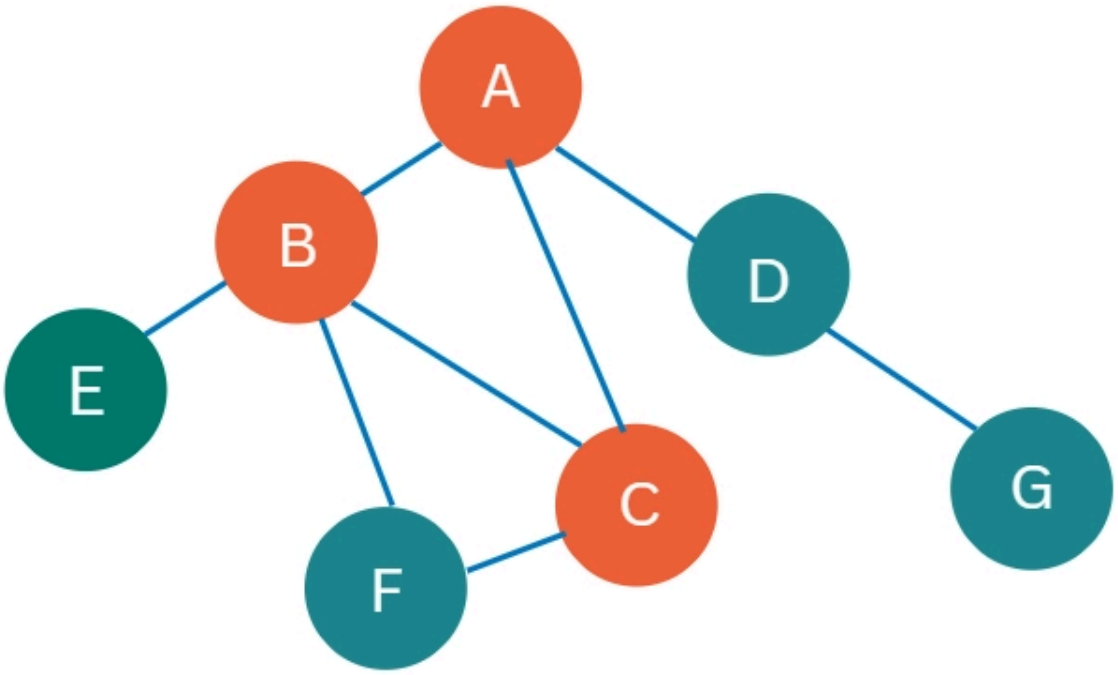
- S2: B, A
  - Node B will be treated as having no successor because the current level is already the limited depth L.
  - As a result, nothing is reachable.
  - Take B from S2.
  - The current value is 0.



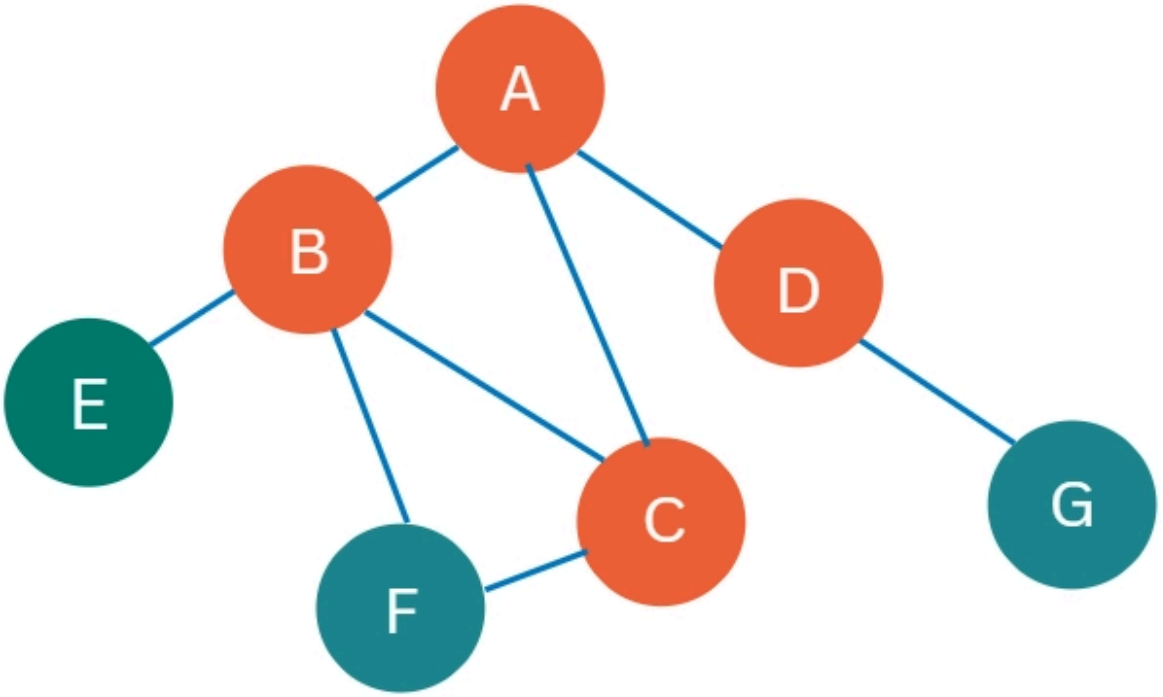
- S2: A
  - Explore A once more.
  - There are two unvisited nodes, C and D, that can be reached.
  - Assume we begin our exploration with node C.
  - C is pushed into S1 and marked as visited.
  - The current level is one.



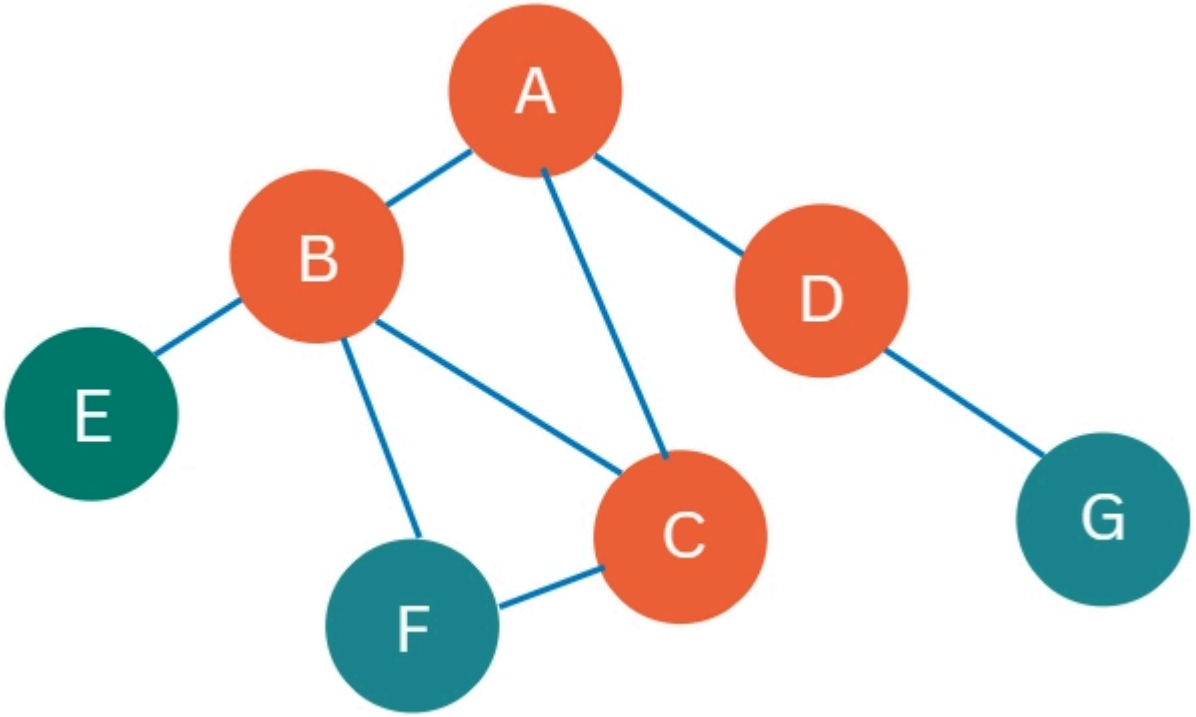
- S2: C, A
  - Because the current level already has the limited depth L, node C is considered to have no successor.
  - As a result, nothing is reachable.
  - Take C from S2.
  - The current value is 0.



- S2: A
  - Explore A once more.
  - There is only one unvisited node D, that can be reached.
  - D should be pushed into S2 and marked as visited.
  - The current level is one.



- S2: D, A
  - D is explored, but no new nodes are found.
  - Take D from S2.
  - The current value is 0.
- S2: A
  - Explore A once more.
  - There is no new reachable node.
  - Take A from S2.



- Similarly at depth limit 2, IDS has already explored all the nodes reachable from a; if the solution exists in the Graph, it has been found.

1.6. Greedy best-first search (GBFS)

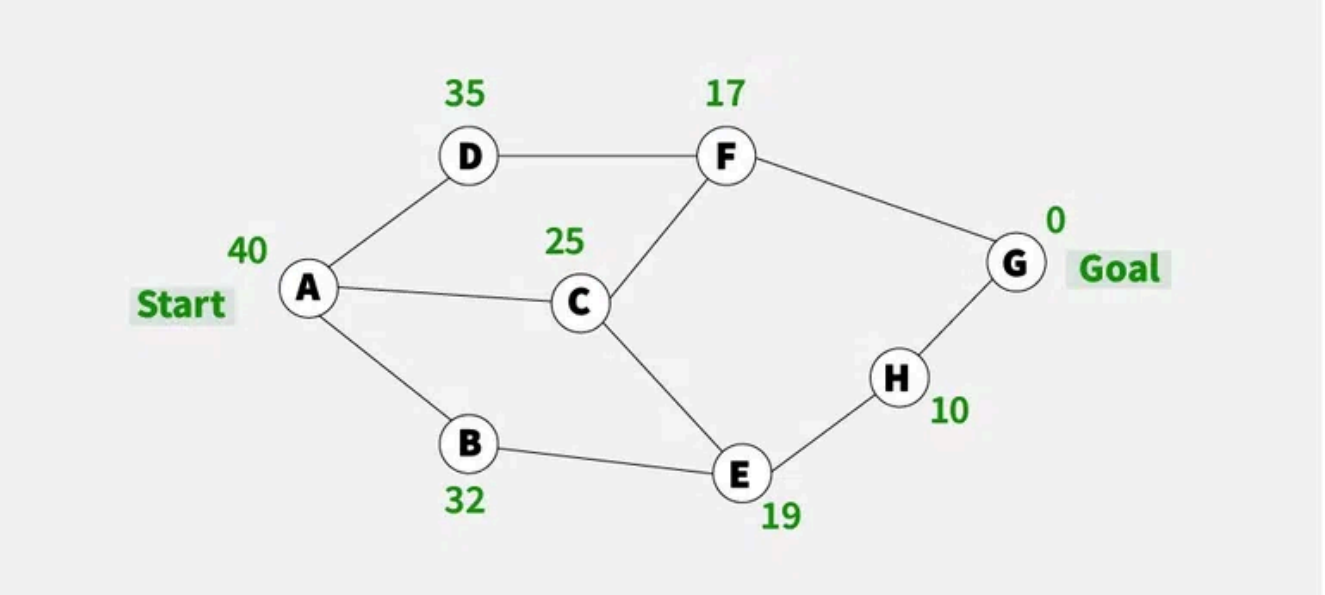
1.6.1. Description

Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm 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.

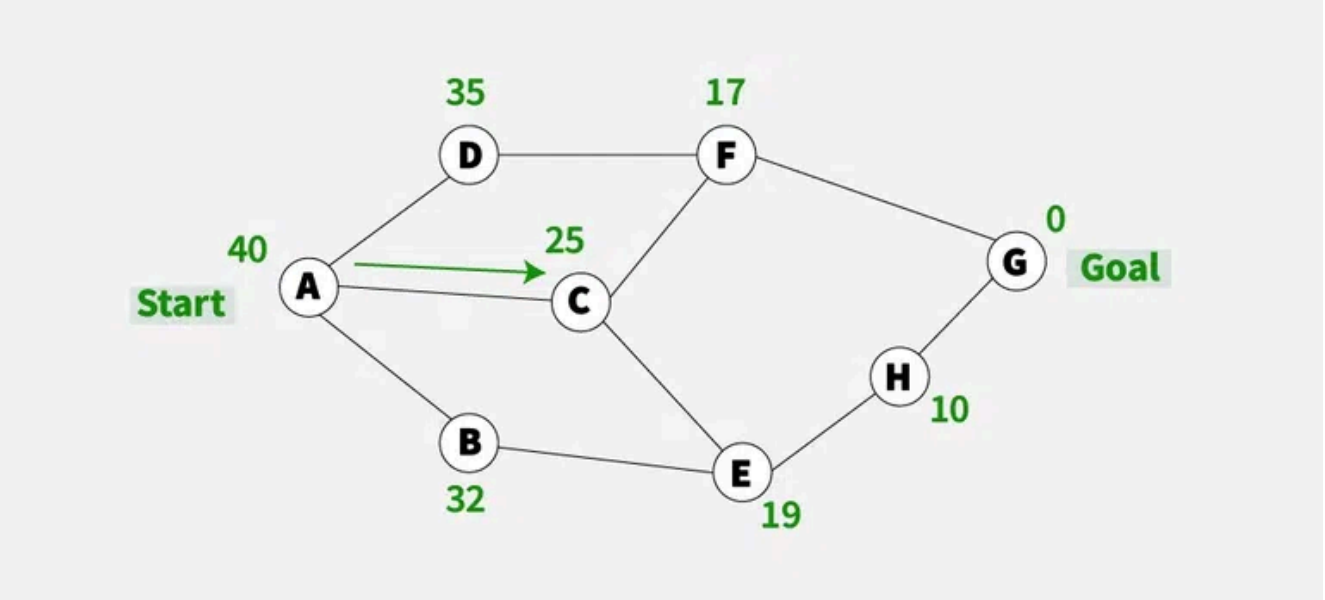
1.6.2. Steps of execution

- We have to find the path from A to G.

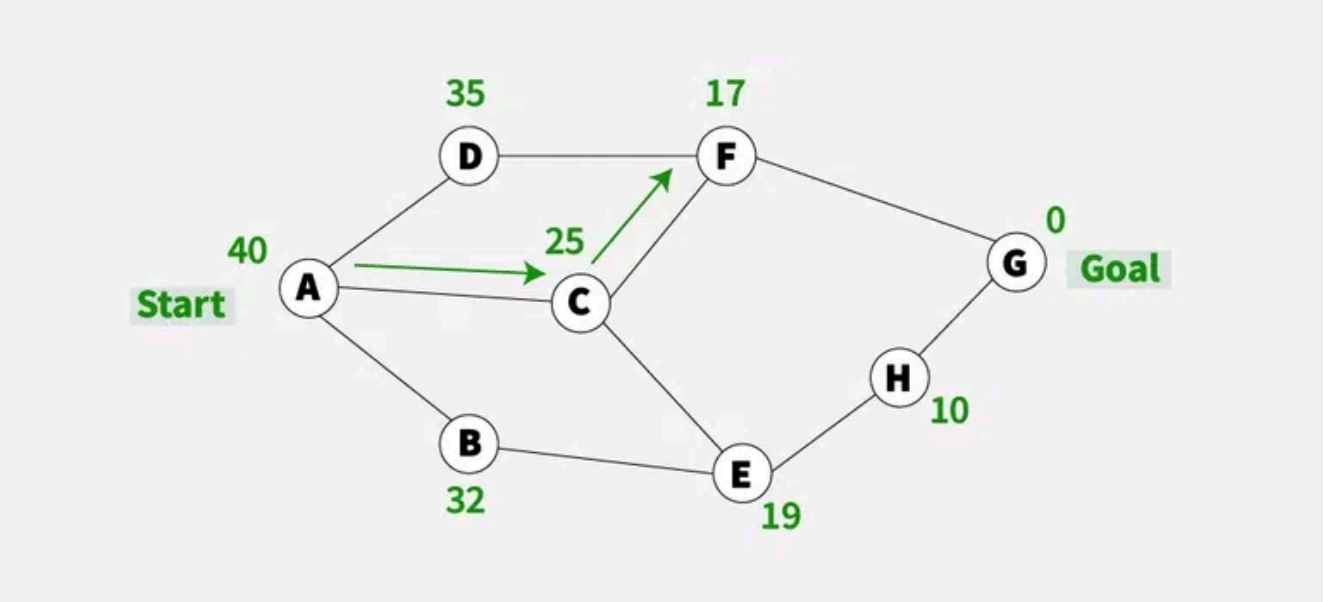
The values in red color represent the heuristic value of reaching the goal node G from current node



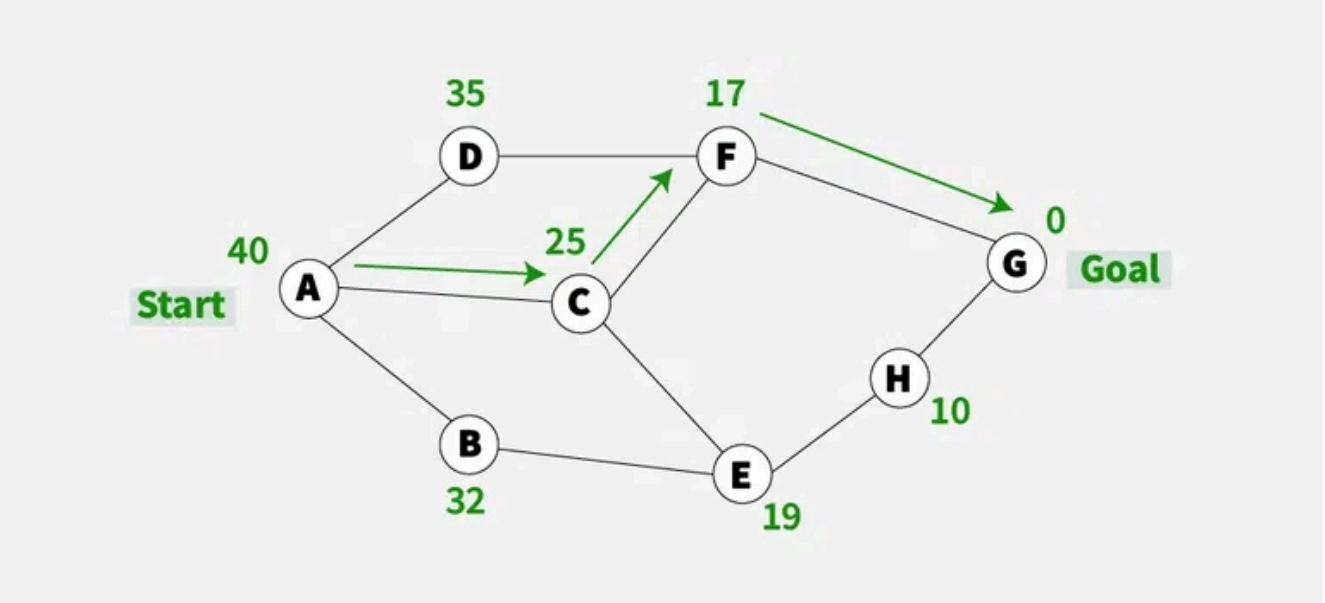
- **Step 1:** We are starting from A , so from A there is a direct path to node B (with heuristics value of 32) , from A to C (with heuristics value of 2) and from A to D(with heuristics value of 35).
- **Step 2:** So as per best first search algorithm choose the path with lowest heuristics value , currently C has lowest value among above nodes . So we will go from A to C.



- **Step 3:** Now 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.



- **Step 4:** Now from F we have a direct path to go to the goal node G (with heuristics value of 0) , so we will go from F to G.



- **Step 5:** So now the goal node G has been reached and the path we will follow is **A->C->F->G** .

1.7. Graph-search A* (A*)

1.7.1. Description

Graph-Search A* (A*) is an advanced search algorithm that is widely used in artificial intelligence for finding the shortest path from a start node to a goal node in a weighted graph. A* combines the strengths of Dijkstra's algorithm and Best-First Search, making it both complete and optimal under certain conditions.

1.7.2. Steps of execution

Algorithm A* selects the nodes to be explored based on the lowest value of f(n), preferring the nodes with the lowest estimated total cost to reach the goal.

- Create an open list of found but not explored nodes.
- Create a closed list to hold already explored nodes.
- Add a starting node to the open list with an initial value of g
- Repeat the following steps until the open list is empty or you reach the target node:
    - Find the node with the smallest f-value (i.e., the node with the minor g(n) h(n)) in the open list.
    - Move the selected node from the open list to the closed list.
    - Create all valid descendants of the selected node.
    - For each successor, calculateits 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. Update the g-value of the tracker when a better path is found.
    - If the following is not in the open list, add it with the calculated g-value and calculate its h-value. If it is already in the open list, update its g value if the new path is better.
    - 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* The 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.

1.8. Hill-climbing (HC) variant

1.8.1. Description

Hill-climbing (HC) is a local search algorithm that iteratively moves towards the optimal solution by choosing the best neighboring state. However, it can get stuck in local optima. Several variants of hill-climbing have been developed to address its limitations and improve performance. Here are some of the notable variants:Simple Hill-Climbing,

Steepest-Ascent Hill-Climbing, Stochastic Hill-Climbing, Random-Restart Hill-Climbing, First-Choice Hill-Climbing, Simulated Annealing,...

1.8.2. Steps of execution

Algorithm for Simple Hill Climbing:

- **Step 1**: Evaluate the initial state, if it is a goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    - If it is a goal state, then return success and quit.
    - Else if it is better than the current state then assign a new state as a current state.
    - Else if not better than the current state, then return to step 2.

2. Results

2.1. Results of Input files and Output files

- Input file 01:

Input	Output
5 0 3 0 4 5 0 0 4 0 2 5 6 5 2 0 3 0 0 5 3 0 1 0 6 0 1 0 8 5 3 0 1	BFS: Path: 0 -> 1 -> 3 Time: 0.0001696000 seconds Memory: 1.484375 KB DFS: Path: 0 -> 2 -> 3 Time: 0.0000965000 seconds Memory: 0.6904296875 KB UCS: Path: 0 -> 2 -> 3 Time: 0.0001538000 seconds Memory: 0.8388671875 KB DLS: Path: 0 -> 1 -> 2 -> 3 Time: 0.0000621000 seconds Memory: 1.25 KB IDS: Path: 0 -> 1 -> 3 Time: 0.0000835000 seconds Memory: 1.640625 KB GBFS: Path: 0 -> 2 -> 3 Time: 0.0000760000 seconds Memory: 0.6904296875 KB AStar: Path: 0 -> 2 -> 3 Time: 0.0001052000 seconds Memory: 0.8779296875 KB HC: Path: 0 -> 2 -> 3 Time: 0.0000486000 seconds Memory: 0.7060546875 KB

- Input file 02:

Input	Output
5 0 4 0 1 2 0 0 1 0 0 3 0 2 0 0 1 4 0 3 1 0 1 0 0 4 1 0 7 6 2 1 0	BFS: Path: 0 -> 2 -> 4 Time: 0.0003108000 seconds Memory: 1.4091796875 KB DFS: Path: 0 -> 2 -> 4 Time: 0.0001147000 seconds Memory: 0.7138671875 KB UCS: Path: 0 -> 2 -> 3 -> 4 Time: 0.0002327000 seconds Memory: 0.8076171875 KB DLS: Path: 0 -> 1 -> 3 -> 2 -> 4 Time: 0.0001243000 seconds Memory: 1.5234375 KB IDS: Path: 0 -> 2 -> 4 Time: 0.0001627000 seconds Memory: 1.7294921875 KB GBFS: Path: 0 -> 2 -> 4 Time: 0.0001240000 seconds Memory: 0.6904296875 KB AStar: Path: 0 -> 2 -> 3 -> 4 Time: 0.0002281000 seconds Memory: 0.9716796875 KB HC: Path: 0 -> 2 -> 4 Time: 0.0000817000 seconds Memory: 0.7060546875 KB

- Input file 03:

Input	Output
4 1 3 0 2 0 1 2 0 3 0 0 3 0 1 1 0 1 0 10 5 2 0	BFS: Path: 1 -> 0 -> 3 Time: 0.0001323000 seconds Memory: 1.4013671875 KB DFS: Path: 1 -> 2 -> 3 Time: 0.0000488000 seconds Memory: 0.6904296875 KB UCS: Path: 1 -> 0 -> 3 Time: 0.0000772000 seconds Memory: 0.7451171875 KB DLS: Path: 1 -> 0 -> 3 Time: 0.0000348000 seconds Memory: 0.984375 KB IDS: Path: 1 -> 0 -> 3 Time: 0.0000532000 seconds Memory: 1.640625 KB GBFS: Path: 1 -> 2 -> 3 Time: 0.0000509000 seconds Memory: 0.6669921875 KB AStar: Path: 1 -> 2 -> 3 Time: 0.0000648000 seconds Memory: 0.8076171875 KB HC: Path: 1 -> 2 -> 3 Time: 0.0000333000 seconds Memory: 0.7060546875 KB

- Input file 04:

Input	Output
6 0 5 0 1 0 0 2 0 1 0 1 0 0 2 0 1 0 1 0 0 0 0 1 0 3 1 2 0 0 3 0 1 0 2 0 1 1 0 6 4 3 2 1 0	BFS: Path: 0 -> 1 -> 5 Time: 0.0003055000 seconds Memory: 1.609375 KB DFS: Path: 0 -> 4 -> 5 Time: 0.0001198000 seconds Memory: 0.7138671875 KB UCS: Path: 0 -> 1 -> 5 Time: 0.0003044000 seconds Memory: 0.8701171875 KB DLS: Path: 0 -> 1 -> 2 -> 3 -> 4 -> 5 Time: 0.0001520000 seconds Memory: 1.9609375 KB IDS: Path: 0 -> 1 -> 5 Time: 0.0001411000 seconds Memory: 1.640625 KB GBFS: Path: 0 -> 4 -> 5 Time: 0.0001276000 seconds Memory: 0.6904296875 KB AStar: Path: 0 -> 4 -> 5 Time: 0.0001752000 seconds Memory: 0.8779296875 KB HC: Path: 0 -> 4 -> 5 Time: 0.0000818000 seconds Memory: 0.7060546875 KB

- Input file 05:

Input	Output
3 2 0 0 2 3 2 0 1 3 1 0 5 3 0	BFS: Path: 2 -> 0 Time: 0.0001576000 seconds Memory: 1.46875 KB DFS: Path: 2 -> 0 Time: 0.0001219000 seconds Memory: 0.6669921875 KB UCS: Path: 2 -> 0 Time: 0.0001361000 seconds Memory: 0.7373046875 KB DLS: Path: 2 -> 0 Time: 0.0000522000 seconds Memory: 0.7265625 KB IDS: Path: 2 -> 0 Time: 0.0000586000 seconds Memory: 1.0234375 KB GBFS: Path: 2 -> 0 Time: 0.0001099000 seconds Memory: 0.6669921875 KB AStar: Path: 2 -> 0 Time: 0.0001400000 seconds Memory: 0.8076171875 KB HC: Path: -1 Time: 0.0000482000 seconds Memory: 0.6201171875 KB

- Input file 06:

Input	Output
7 0 6 0 1 0 0 0 2 0 1 0 2 0 0 0 3 0 2 0 2 0 0 0 0 0 2 0 1 0 0 0 0 0 1 0 3 2 2 0 0 0 3 0 1 0 3 0 0 2 1 0 10 8 6 4 2 1 0	BFS: Path: 0 -> 1 -> 6 Time: 0.0001844000 seconds Memory: 1.640625 KB DFS: Path: 0 -> 5 -> 6 Time: 0.0000547000 seconds Memory: 0.7138671875 KB UCS: Path: 0 -> 5 -> 6 Time: 0.0001770000 seconds Memory: 0.8544921875 KB DLS: Path: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 Time: 0.0000812000 seconds Memory: 2.21875 KB IDS: Path: 0 -> 1 -> 6 Time: 0.0000613000 seconds Memory: 1.640625 KB GBFS: Path: 0 -> 5 -> 6 Time: 0.0000887000 seconds Memory: 0.6904296875 KB AStar: Path: 0 -> 5 -> 6 Time: 0.0000770000 seconds Memory: 0.8779296875 KB HC: Path: 0 -> 5 -> 6 Time: 0.0000400000 seconds Memory: 0.7060546875 KB

2.2. Review

2.2.1. Path finding capability

- BFS: Successfully finds paths in all cases.
- DFS: Successfully finds paths in all cases.
- UCS: Successfully finds paths in all cases.
- DLS: Successfully finds paths but may explore unnecessary nodes.
- IDS: Successfully finds paths similar to BFS.
- GBFS: Successfully finds paths in all cases using heuristics.
- A*: Successfully finds paths with the shortest path guarantee.
- HC: Finds paths using heuristics, but may get stuck and report no path.

2.2.2. Runtime

- Fastest: HC generally has the fastest runtime.
- Slowest: BFS and UCS tend to have the longest runtimes.
- Consistency: DLS and IDS provide consistent runtimes.

2.2.3. Memory usage

- Lowest: HC, DFS, and GBFS typically use the least memory.
- Highest: BFS, IDS, and DLS tend to use more memory.
- Consistency: UCS and A* have moderate memory usage.

2.2.4. Overall

- BFS: Reliable but can be slow and memory-intensive.
- DFS: Fast and low memory but may not find the shortest path.
- UCS: Finds the shortest path but can be slow.
- DLS: Efficient but may explore unnecessary nodes.
- IDS: Combines DFS and BFS benefits but can be memory-intensive.
- GBFS: Fast with low memory but may not find the optimal path.
- A*: Finds the shortest path with moderate resources.
- HC: Very fast and low memory but may fail to find a path.

REFERENCES

[1] Search Algorithms Implementation

https://chatgpt.com/share/ee043d91-9d19-4307-90fe-7bd3eabc5d40

[2] Depth First Search or DFS for a Graph - Geeks for geeks

Depth First Search or DFS for a Graph - GeeksforGeeks

[3] Breadth First Search or BFS for a Graph

Breadth First Search or BFS for a Graph - GeeksforGeeks

[4] A* Search Algorithm

A* Search Algorithm - GeeksforGeeks

[5] Introduction to Hill Climbing

Introduction to Hill Climbing | Artificial Intelligence - GeeksforGeeks

[6] Uniform-Cost Search (Dijkstra for large Graphs)

Uniform-Cost Search (Dijkstra for large Graphs) - GeeksforGeeks

[7] Greedy Best first search algorithm

Greedy Best first search algorithm - GeeksforGeeks

[8] Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS) - GeeksforGeeks

[9] Depth Limited Search for AI

Depth Limited Search for AI - GeeksforGeeks

[10] Uniform-Cost Search Algorithm - Scaler Topics

Uniform-Cost Search Algorithm - Scaler Topics

[11] What is depth-limited search?

What is depth-limited search

[12] Iterative Deepening Search (IDS) vs. Iterative Deepening Depth First Search (IDDFS)

Iterative Deepening Search(IDS) vs. Iterative Deepening Depth First Search(IDDFS) - Naukri Code 360

[13] Hill Climbing Algorithm in Artificial Intelligence

Hill Climbing Algorithm in AI - Javatpoint

[14] A* Search Algorithm in Artificial Intelligence

Informed Search Algorithms in AI - Javatpoint