**Uninformed search algorithm**

**(Foundation of ai with r programming)**

**(XAI302/XAI305)**



**Submitted to : Submitted by:**

SUGAVANESHWARAN, **S.SRIRANJANI**

**(FOUNDATION OF AI BSC AI 2ND YEAR**

**WITH R PROGRAMMING) DEPT OF (CS)**

**Uninformed search algorithm:**

|  |
| --- |
| * DEPTH FIRST SEARCH * BREADTH FIRST SEARCH * DEPTH LIMITED SEARCH * ITERATIVE DEEPENING SEARCH * BIDIRECTIONAL SEARCH * UNIFORM COST SEARCH |

**depth first search :**

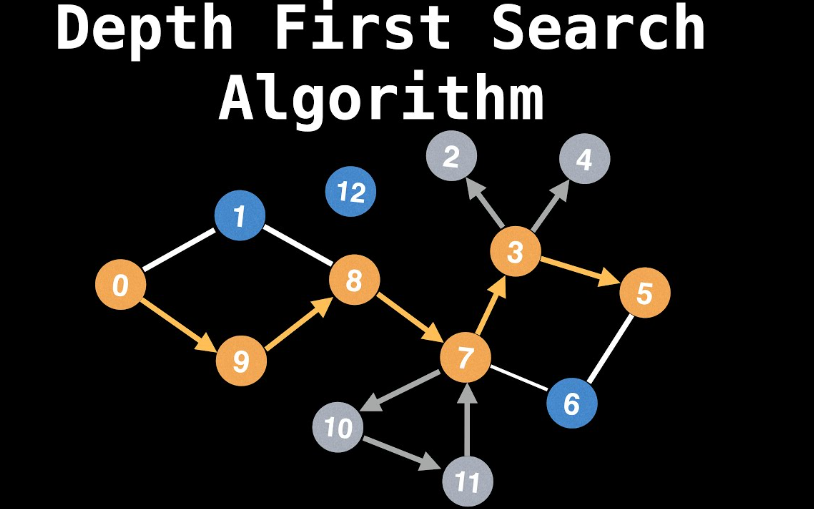
[**Depth-first search (DFS)** is a graph traversal algorithm that explores as far as possible along each branch before backtracking](https://en.wikipedia.org/wiki/Depth-first_search). [It starts at the root node and explores as far as possible along each branch before backtracking](https://www.javatpoint.com/depth-first-search-algorithm). [DFS can be implemented using a **stack** data structure](https://www.javatpoint.com/depth-first-search-algorithm). [The algorithm is recursive in nature and can be used to search all the vertices of a tree data structure or a graph](https://www.javatpoint.com/depth-first-search-algorithm).

[Here is the step-by-step process to implement the DFS traversal](https://www.javatpoint.com/depth-first-search-algorithm):

1. Create a stack with the total number of vertices in the graph.
2. Choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. Push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack’s top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

[DFS algorithm can be used to implement **topological sorting**, **finding paths between two vertices**, **detecting cycles in the graph**, **determining if a graph is bipartite or not**, and for one solution puzzles](https://www.bing.com/search?q=detail+about+depth+first+serach+algorithm&cvid=1242733d44ea4ad8ad5a6171adf98d1c&aqs=edge..69i57j69i64j0l7.57167j0j8&FORM=ANAB01&PC=HCTS).

[Here’s an example of how DFS works](https://www.javatpoint.com/depth-first-search-algorithm):



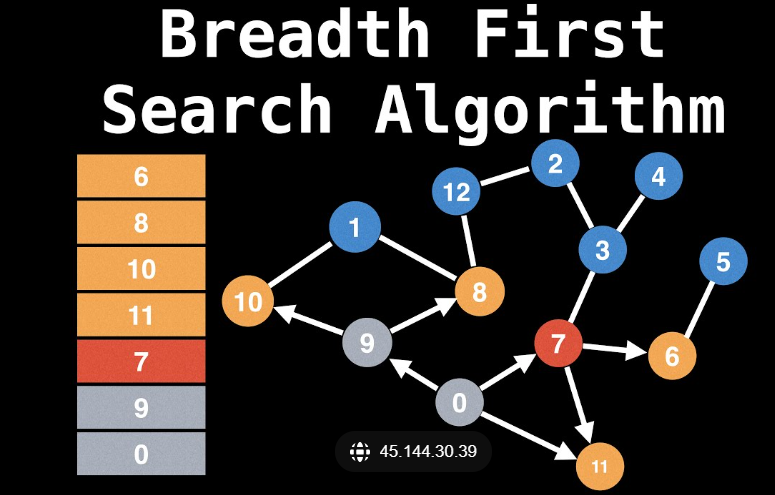
Starting from node A, we explore as far as possible along each branch before backtracking. The order in which nodes are visited is: A, B, D, E, C,F.

**Breadth first search:**

Breadth-first search (BFS) is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. [While using BFS for traversal, any node in the graph can be considered as the root node](https://www.bing.com/search?q=detail+about+breadth++first+serach+algorithm&qs=n&form=QBRE&sp=-1&lq=0&pq=detail+about+breadth++first+serach+algorithm&sc=5-44&sk=&cvid=39EBD98C37FB475BAC0499669E7A7079&ghsh=0&ghacc=0&ghpl=).

Here are the steps involved in BFS algorithm to explore a graph:

1. SET STATUS = 1 (ready state) for each node in G
2. Enqueue the starting node A and set its STATUS = 2 (waiting state)
3. Repeat Steps 4 and 5 until QUEUE is empty
4. Dequeue a node N. Process it and set its STATUS = 3 (processed state).
5. Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
6. EXIT



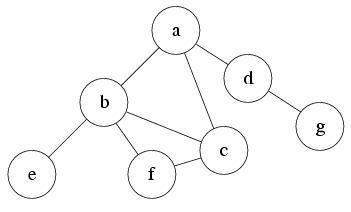
[BFS algorithm can be used to find the neighbouring locations from a given source location, find all the neighbouring nodes in a peer-to-peer network, index web pages, determine the shortest path and minimum spanning tree, compute maximum flow in a flow network, and duplicate garbage collection](https://www.javatpoint.com/breadth-first-search-algorithm)

**depth limited search:**

The depth-limited search (DLS) method is almost equal to depth-first search (DFS), but DLS can work on the infinite state space problem because it bounds the depth of the search tree with a predetermined limit L. Nodes at this depth limit are treated as if they had no successors.

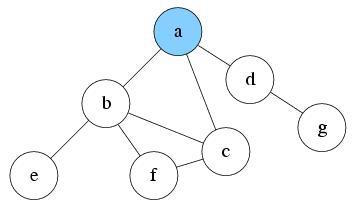
Now use the example in DFS to see what will happen if we use DLS with L=1�=1.

Below is the graph we will traverse. Same as DFS, we use the stack data structure S1 to record the node we’ve explored. Suppose the source node is node a.



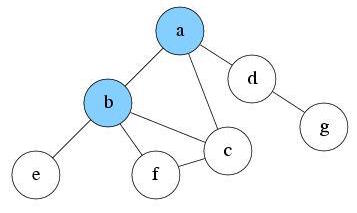
###### S1:

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



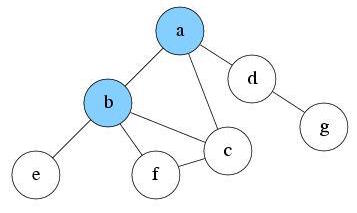
###### S1: a

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.



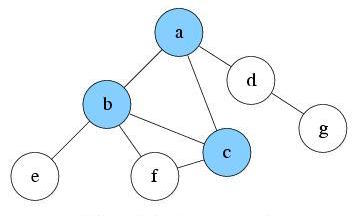
###### S1: b, a

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.



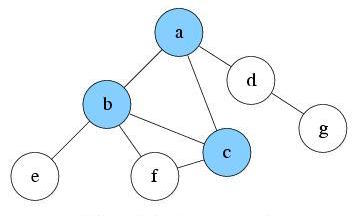
###### S1: a

Explore a 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.



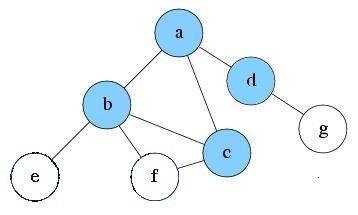
###### S1: c, a

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.



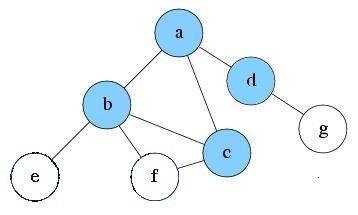
###### S1: a

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



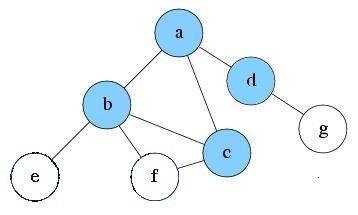
###### S1: d, a

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:

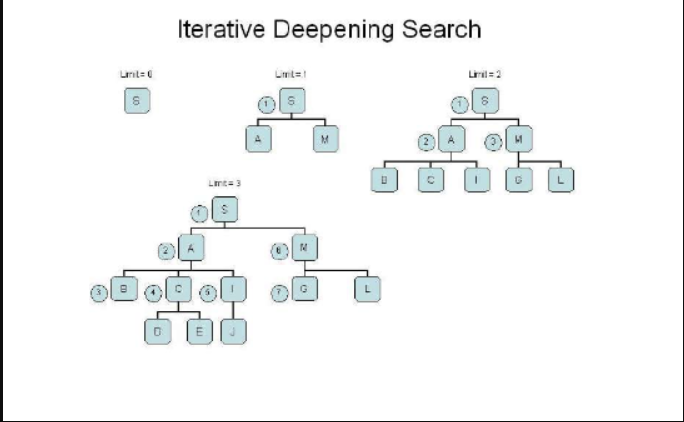
S1 is empty now. DLS will be finished.

**iterative deepening search:**

[Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search (IDDFS) is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph](https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/) .

It is an amalgam of Breadth First Search (BFS) and Depth First Search (DFS) algorithms. [The algorithm works by performing DFS to a certain depth, then backing up to the root node and performing DFS again with a deeper depth limit](about:blank)  [This process continues until the goal node is found or the maximum depth is reached](about:blank) .

[The algorithm is space-efficient, as it only stores the nodes in the current path from the root to the current node](about:blank) .  [It also guarantees that the shortest path to the goal node i s found, as it performs DFS with increasing depth limits](about:blank) .

[However, it may visit some nodes multiple times, which can make it slower than other algorithms in some cases](about:blank) . 

**bidirectional search:**

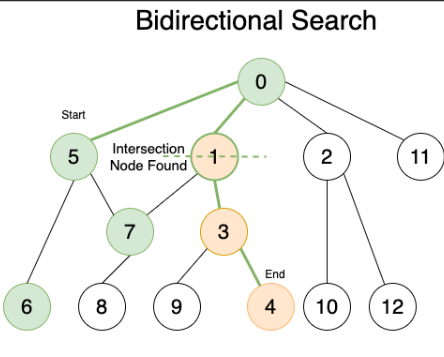
Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. [It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet1](https://en.wikipedia.org/wiki/Bidirectional_search).

Here is an example diagram of bidirectional search using breadth-first search (BFS) as the base algorithm:

[Bidirectional search example]

In this example, the start vertex is S and the goal vertex is G. The forward search expands the vertices in the order S, A, B, C, D, E, F. The backward search expands the vertices in the order G, L, I, H, K, J. The two searches meet at vertex E, which is the intersection point. The shortest path from S to G is then S-A-B-C-D-E-L-G, which has a length of 7.

Bidirectional search can reduce the time and space complexity of unidirectional search algorithms, such as BFS or DFS. [However, it requires that both the initial and goal states are unique and completely defined, and that the branching factor is exactly the same in both directions](about:blank).



**uniform cost search algorithm:**

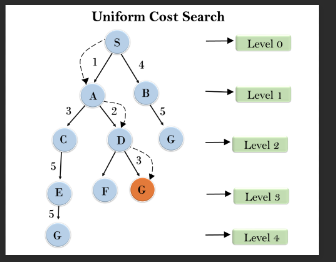
Uniform cost search algorithm is a type of uninformed search algorithm that finds the shortest path from a start node to a goal node in a weighted graph. [It is also known as Dijkstra’s algorithm for large graphs1](https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/).

The algorithm works by using a priority queue to store the nodes to be explored, where the priority of each node is its current cost from the start node. The algorithm then repeatedly pops the node with the lowest cost from the queue and expands its adjacent nodes, updating their costs and parents if a lower-cost path is found. [The algorithm terminates when the goal node is popped from the queue or when the queue becomes empty](about:blank).

Here is an example of uniform cost search algorithm on a simple graph:

![Uniform cost search example]

In this example, the start node is S and the goal node is G. The numbers on the edges represent the costs of moving from one node to another. The algorithm starts by adding S to the queue with a cost of 0. Then it pops S and adds its adjacent nodes A and B to the queue with their costs of 2 and 1 respectively. Then it pops B and adds its adjacent nodes D and E to the queue with their costs of 5 and 5 respectively. Then it pops A and adds its adjacent node C to the queue with its cost of 4. Then it pops C and adds its adjacent node F to the queue with its cost of 7. Then it pops D and adds its adjacent nodes F and G to the queue with their costs of 10 and 7 respectively. Then it pops E and does nothing since it has no unvisited adjacent nodes. Then it pops G and returns it as the goal node with a cost of 7. [The shortest path from S to G is then S-B-D-G, which has a length of 4 and a cost of 7](about:blank).



program:

# Import heapq module for priority queue

import heapq

# Define a function for the uniform cost search algorithm

def uniform\_cost\_search(graph, start, goal):

# Initialize the priority queue, the visited set, and the parent dictionary

queue = [(0, start)]

visited = set()

parent = {}

# Loop until the queue is empty

while queue:

# Pop the node with the lowest cost

cost, node = heapq.heappop(queue)

# Check if the node is the goal

if node == goal:

# Construct the path and return it

path = []

while node != start:

path.append(node)

node = parent[node]

path.append(start)

return path[::-1]

# Check if the node is already visited

if node not in visited:

# Mark the node as visited

visited.add(node)

# Loop through the adjacent nodes

for neighbor in graph[node]:

# Get the cost to the neighbor and the total cost

edge\_cost = graph[node][neighbor]

total\_cost = cost + edge\_cost

# Push the neighbor to the queue

heapq.heappush(queue, (total\_cost, neighbor))

# Update the parent of the neighbor

parent[neighbor] = node

# Return None if no path is found

return None