**TUTORIAL 3**

1)What are the two types of searching algorithms? How do they differ? Give two example algorithms for each search type.

Informed Search

* Best first search
* A\* Search

Uninformed Search

* Breadth first search
* Depth first search
* Uniform cost search
* Depth limited search
* Iterative deepening depth first search

2) Define the terms: completeness, optimality, time complexity and space complexity as applied in search algorithms.

Completeness: Completeness refers to whether a search algorithm is guaranteed to find a solution if one exists within the search space. A search algorithm is considered complete if it is guaranteed to find a solution when one exists. In other words, it will not fail to find a solution.

Optimality: Optimality refers to whether a search algorithm is guaranteed to find the best or optimal solution among all possible solutions. An optimal search algorithm is one that will find the best solution, typically defined as the one with the lowest cost or highest value, depending on the problem domain.

Time Complexity: Time complexity measures the amount of time or number of computational steps required for a search algorithm to find a solution. It quantifies the efficiency of the algorithm in terms of the time it takes to complete, typically expressed as a function of the input size. Lower time complexity indicates faster execution and better efficiency.

Space Complexity: Space complexity measures the amount of memory or storage required by a search algorithm to solve a problem. It quantifies the efficiency of the algorithm in terms of the space it uses, typically expressed as a function of the input size. Lower space complexity indicates the algorithm uses less memory, which can be desirable in resource-constrained environments.

3) Define the terms Breadth First Search (BFS) and Depth First Search (DFS) and mention at least 5 differences between them.

Breadth first search – It is the most common search strategy to traverse a tree or graph. First the root node is expanded first then all the successors of the root node are expanded next then their successors and so on. Breadth first search starts searching from the root node and expands all success nodes at the current level before moving to the node of the next level.

Depth first search - It is a recursive algorithm to traverse a tree or a graph data structure. It is called depth first as it starts from the root node and follows each path to the greatest depth node before moving to the next step.

|  |  |
| --- | --- |
| Breadth first  search | Depth first  search |
| Tree traversal is done level wise - explores all the neighboring nodes at the current depth level before moving to the next depth level | Tree traversal is done depth-wise. |
| BFS uses a queue data structure to store the nodes that needs to be explored | DFS uses a stack data structure to store the nodes that needs to be explored, it is easy for backtracking |
| BFS finds the optimal solution - BFS guarantees finding the shortest path or the optimal solution if all edge costs are equal. | It does not find the optimal solution - BFS guarantees finding the shortest path or the optimal solution if all edge costs are equal |
| BFS generally requires more memory compared to DFS. This is because BFS needs to store all the nodes at the current level before moving to the next level | DFS only needs to store the nodes along the current path from the root to the current node. Therefore it has a lower memory usage. |
| DFS only needs to store the nodes along the current path from the root to the current node. Therefore it is complete | DFS only needs to store the nodes along the current path from the root to the current node. |

4) Explain how you would use Queue for implementing BFS and a Stack for implementing a DFS.

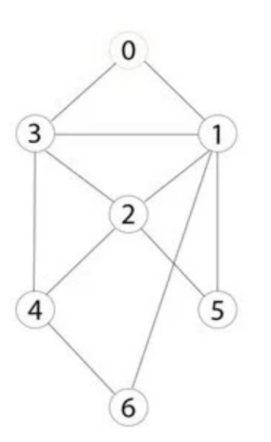
To implement Breadth First Search (BFS) using a Queue, you can follow these steps:

1. Initialize a Queue data structure.
2. Enqueue the starting node or root into the Queue.
3. Create an empty set or array to keep track of visited nodes.
4. Start a loop until the Queue becomes empty.
5. Dequeue a node from the front of the Queue.
6. Check if the node has been visited before. If not, mark it as visited and process it.
7. Enqueue all the unvisited neighboring nodes of the current node into the Queue.
8. Repeat steps 5 to 7 until the Queue is empty.

To implement Breadth First Search (BFS) using a Queue, you can follow these steps:

1. Initialize a Queue data structure.
2. Enqueue the starting node or root into the Queue.
3. Create an empty set or array to keep track of visited nodes.
4. Start a loop until the Queue becomes empty.
5. Dequeue a node from the front of the Queue.
6. Check if the node has been visited before. If not, mark it as visited and process it.
7. Enqueue all the unvisited neighboring nodes of the current node into the Queue.
8. Repeat steps 5 to 7 until the Queue is empty.

5) Consider a BFS traversal from the node 0 to the node 6 using queue-based implementation. Mention the updated visited nodes and nodes in the queue for each step. Hence derive final path of BFS traversal.



* 1. First push 0 to the queue
  2. Then 0 is popped from the queue then add its successors 3 and 1

Pop – 0

Queue – 1,3

* 1. Then the first element in the queue is popped and its successor is added to the stack which is 2 and 5

Pop – 1

Queue – 3,2,5

* 1. Then in the above queue 3 is dequeued from the front of the queue and its successors are added which is 4 and 2 but since 2 is already visited it is not added again

Pop – 3

Queue – 2,5,4

* 1. Then the above queue 2 is in the front of the queue so it is removed so its successors are added which is 5 and 4 which is already in the list

Pop – 2

Queue – 5,4

* 1. Then in the above queue 5 is removed as it’s the first element so 5 is popped and its successors are added which there are no successors

Pop – 5

Queue – 4

* 1. Then in the above queue 4 is removed and its successors are added which is 6

Pop – 4

Queue – 6

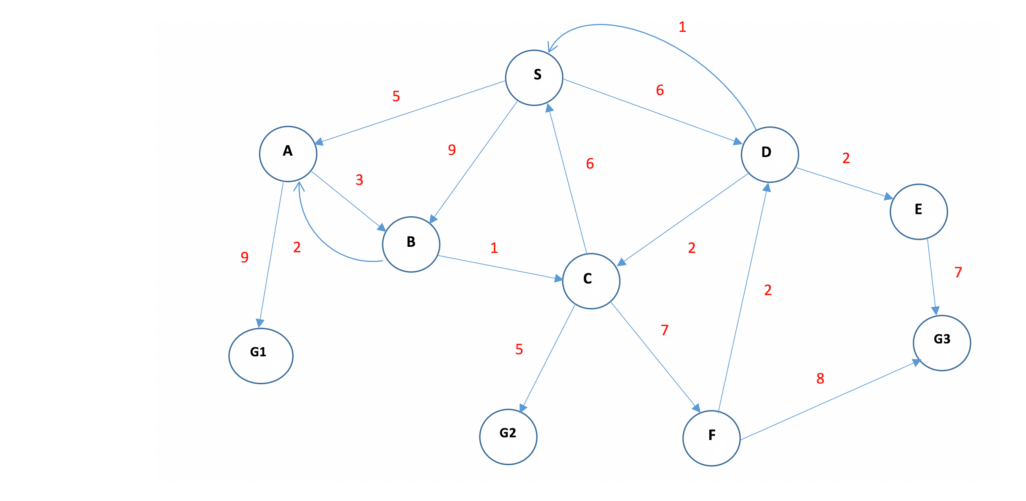
* 1. Then 6 is removed and its successors are 1 which is already visited so it stops from here.

Pop – 6

Queue:

So the traversal order is 0 – 1 – 3 – 2 – 5 – 4 – 6

6) What is the change in implementation discussed in Question 4, if the graph you are traversing is a directed one? Execute BFS and DFS individually for below directed graph. Produce the visited nodes list and final path for each case individually.



**BREADTH FIRST SEARCH**

1)First push S into the queue

2)Then S is popped and its successors are added

Pop – S

Queue – A, B, D

3)Then A is popped and added to the list and its successors are G1 and B since B is already in the queue G1 is added

Pop – S, A

Queue – B, D, G1

4) Then B is popped and added to the list and its successors are A and C so C is added as A is already visited

Pop – S, A, B

Queue – D, G1, C

5) Then D is popped and added to the list and its successors are C,E,S so E is added

Pop – S, A, B, D

Queue – G1, C, E

6)Next G1 Is popped and added to the list it has no successors.,

Pop – S, A , B, D, G1

Queue – C, E

Since G1 is a goal state it terminates from here so the traversal order is

S – A – B – D – G1

**DEPTH FIRST SEARCH**

1)First S is pushed into the stack

2) Then S is popped

Pop – S

Stack - ---

3)Then its successors are added A,B,D

Pop – S

Stack – A, B, D

4) Then D is popped from the stack and its successors are added C,E,S since S is already there C AND E are added

Pop – S,D

Stack – A, B, C, E

5) Then A is popped from the stack and its successors are added G1 and B

Pop – S, D, A

Stack – B, C, E, G1

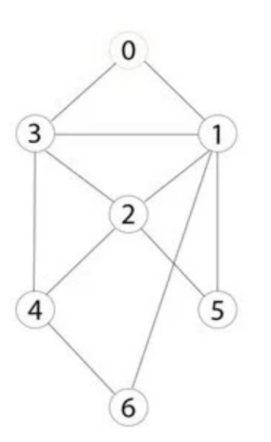
6) Then G1 is popped from the stack and it has no successors

Pop – S, D, A, G1

Stack – B, C, E

Since G1 is a goal node it terminates so the traversal order is S – D – A – G1

**5) DFS SEARCH**



1. VL – -

UL – 1 , 3

1. VL – 0,3

UL – 1, 2,4

1. VL – 0,3,4

UL – 1,2,6

1. VL – 0,3,4,6

UL – 1,2

7) Derive the time complexity for BFS.

The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

8) List down real-life applications of BFS and DFS. Explain for each case why the selected strategy is good compared to the other search strategy.

**BREADTH FIRST SEARCH APPLICATIONS**

1. Shortest Path Finding in Unweighted Graphs:
   * BFS is widely used to find the shortest path in unweighted graphs. It guarantees finding the shortest path between two nodes since it explores nodes level by level. This property makes BFS an ideal choice when all edges have the same weight or when the focus is on finding the path with the minimum number of edges.
2. Web crawling

* Crawlers create indexes based on breadth-first. The goal is to start at the original page and follow all of the links there, then repeat. It helps systematically explore the web by visiting web pages in a breadth-first manner. By traversing the web using BFS, one can ensure that all pages at the same level (or depth) are visited before moving on to the next level. This approach efficiently discovers and indexes new web pages.

1. Social Network Analysis:

* BFS finds application in social network analysis, where the goal is to analyze the relationships and connections between individuals. BFS can help find the shortest path between two individuals, identify communities or clusters, and determine the reachability of a person within a network. It ensures a systematic exploration of connections, making it suitable for various social network analysis tasks.

1. Network Broadcasting:

* BFS is often used in network broadcasting scenarios. For example, in computer networks or social media platforms, when a message or information needs to be disseminated to all connected nodes or users, BFS can efficiently achieve this goal. By starting the broadcast from a specific node and exploring its neighbors level by level, BFS ensures that the information reaches all nodes in the shortest possible time.

1. Puzzle Solving:

* BFS can be applied to solve various puzzles that involve exploring a search space. For instance, solving the sliding tile puzzle, solving maze problems, or finding optimal game moves in certain board games. BFS guarantees finding a solution with the fewest steps or moves, making it suitable for solving puzzles that prioritize the shortest path to a solution.

**DEPTH FIRST SEARCH APPLICATIONS**

1. Maze Solving:
   * DFS is commonly used to solve maze problems. It explores a path as deeply as possible before backtracking, which makes it suitable for finding a path from the start to the goal in a maze. DFS can efficiently traverse through dead ends, making it advantageous when the focus is on reaching the goal rather than finding the shortest path.
2. Graph Traversal:
   * DFS is often applied in graph traversal problems. It helps explore all nodes or vertices in a graph by traversing a branch as deeply as possible before backtracking. This characteristic is useful in scenarios where you want to visit all connected components of a graph or find connected clusters.
3. Detecting Cycles:
   * DFS can be used to detect cycles in a graph. By keeping track of visited nodes and parent-child relationships during the traversal, DFS can identify if there are any back edges or cycles in the graph. This makes DFS a suitable strategy for cycle detection in various scenarios, such as dependency resolution, circuit analysis, or deadlock detection.
4. Topological Sorting:
   * DFS is employed in topological sorting of directed acyclic graphs (DAGs). Topological sorting orders the nodes of a DAG in such a way that for every directed edge (u, v), node u comes before node v. DFS can be used to perform topological sorting by exploring the graph in a depth-first manner and assigning an order to each node based on the finishing times of the traversal.
5. Tree Traversal:
   * DFS is commonly used to traverse trees in various applications. It allows for a depth-first exploration of a tree's nodes, making it suitable for operations such as searching for a specific node, calculating tree height, generating prefix or postfix expressions, and more.