Ex. No. 1.A **UNINFORMED SEARCH ALGORITHM - BFS**

Date:

Aim:

To write a Python program to implement Breadth First Search (BFS).

Algorithm:

Step 1. Start
Step 2. Put any one of the graph's vertices at the back of the queue.
Step 3. Take the front item of the queue and add it to the visited list.
Step 4. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
Step 5. Continue steps 3 and 4 till the queue is empty.
Step 6. Stop

Program:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Viva questions:

1. What is BFS and how does it differ from other search algorithms such as DFS or A* search?
2. Can you describe the steps of a BFS algorithm and explain how it works?
3. Can you explain the time and space complexity of a BFS algorithm?
4. Can you give an example of a real-world problem that can be solved using BFS?
5. What does the "visited array" in BFS refer to?

Result:

Thus the Python program to implement Breadth First Search (BFS) was developed successfully.

Ex. No.1.B **UNINFORMED SEARCH ALGORITHM - DFS**

Date:

Aim:

To write a Python program to implement Depth First Search (DFS).

Algorithm:

Step 1. Start
Step 2. Put any one of the graph's vertex on top of the stack.
Step 3. After that take the top item of the stack and add it to the visited list of the vertex.
Step 4. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
Step 5. Repeat steps 3 and 4 until the stack is empty.
Step 6. Stop

Program:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Viva Questions:

1. What is DFS and how does it differ from other search algorithms such as BFS or A* search?
2. Can you describe the steps of a DFS algorithm and explain how it works?
3. How doe DFS handles loops or repeated states in graph?
4. Can you explain the time and space complexity of a DFS algorithm?
5. Can you give an example of a real-world problem that can be solved using DFS?

Result:

Thus the Python program to implement Depth First Search (DFS) was developed successfully.

Ex. No.2. A **INFORMED SEARCH ALGORITHM**

Date: **A* SEARCH**

Aim:

To write a Python program to implement A* search algorithm.

Algorithm:

Step 1: Create a priority queue and push the starting node onto the queue.Initialize minimum value (min_index) to location 0.
Step 2: Create a set to store the visited nodes.
Step 3: Repeat the following steps until the queue is empty:
  3.1: Pop the node with the lowest cost + heuristic from the queue.
  3.2: If the current node is the goal, return the path to the goal.
  3.3: If the current node has already been visited, skip it.
  3.4: Mark the current node as visited.
  3.5: Expand the current node and add its neighbors to the queue.
Step 4: If the queue is empty and the goal has not been found, return None (no path found).
Step 5: Stop

Program:

```
import heapq

class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start, goal, graph):
    heap = []
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))
    visited = set()

    while heap:
        (cost, current) = heapq.heappop(heap)

        if current.state == goal:
            path = []
            while current is not None:
                path.append(current.state)
```

```
            current = current.parent
        # Return reversed path
        return path[::-1]

    if current.state in visited:
        continue

    visited.add(current.state)

    for state, cost in graph[current.state].items():
        if state not in visited:
            heuristic = 0  # replace with your heuristic function
            heapq.heappush(heap, (cost, Node(state, current, current.cost + cost, heuristic)))

  return None  # No path found

graph = {
    'A': {'B': 1, 'D': 3},
    'B': {'A': 1, 'C': 2, 'D': 4},
    'C': {'B': 2, 'D': 5, 'E': 2},
    'D': {'A': 3, 'B': 4, 'C': 5, 'E': 3},
    'E': {'C': 2, 'D': 3}
}
start = 'A'
goal = 'E'

result = astar(start, goal, graph)
print(result)
```

Viva Questions:

1. What is A* search and what makes it different from other search algorithms?
2. How does the A* algorithm choose which node to expand next?
3. Can you explain how the heuristic function is used in the A* algorithm and what role it plays in the search process?
4. How does the cost function used in A* search differ from the heuristic function?
5. What are the advantages and disadvantages of using A* search compared to other search algorithms like breadth-first search or depth-first search?

Result:

Thus the python program for A* Search was developed and the output was verified successfully.

Ex. No.2.B **INFORMED SEARCH ALGORITHM**

Date: **MEMORY-BOUNDED A***

Aim:

To write a Python program to implement memory- bounded A* search algorithm.

Algorithm:

Step 1: Create a priority queue and push the starting node onto the queue.
Step 2: Create a set to store the visited nodes.
Step 3: Set a counter to keep track of the number of nodes expanded.
Step 4: Repeat the following steps until the queue is empty or the node counter exceeds the max_nodes:
    4.1: Pop the node with the lowest cost + heuristic from the queue.
    4.2: If the current node is the goal, return the path to the goal.
    4.3: If the current node has already been visited, skip it.
    4.4: Mark the current node as visited.
    4.5: Increment the node counter.
    4.6: Expand the current node and add its neighbors to the queue.
Step 5: If the queue is empty and the goal has not been found, return None (no path found).
Step 6: Stop

Program:

```
import heapq

class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start, goal, graph, max_nodes):
    heap = []
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))

    visited = set()

    node_counter = 0

    while heap and node_counter < max_nodes:
        (cost, current) = heapq.heappop(heap)

        if current.state == goal:
```

```
            path = []
            while current is not None:
                path.append(current.state)
                current = current.parent
            return path[::-1]

        if current.state in visited:
            continue

        visited.add(current.state)

        node_counter += 1

        for state, cost in graph[current.state].items():
            if state not in visited:
                heuristic = 0
                heapq.heappush(heap, (cost, Node(state, current, current.cost + cost, heuristic)))

    return None

# Example usage

graph = {'A': {'B': 1, 'C': 4},
         'B': {'A': 1, 'C': 2, 'D': 5},
         'C': {'A': 4, 'B': 2, 'D': 1},
         'D': {'B': 5, 'C': 1}}

start = 'A'
goal = 'D'
max_nodes = 10

result = astar(start, goal, graph, max_nodes)
print(result)
```

Viva Questions:

1. What is memory bounded A* search and how does it differ from traditional A* search?
2. How does memory bounded A* search help in handling large state spaces?
3. What is the basic idea behind memory bounded A* search and how does it work?
4. Can you explain the trade-off between optimality and memory usage in memory bounded A* search?
5. How does memory bounded A* search handle the problem of node replanning and how does it impact the performance of the search?

Result:

Thus the python program for memory-bounded A* search was developed and the output was verified successfully.