Artificial Intelligence & Expert Systems (CT-361)
Lab 03: Search Problems

Objectives

The objective of this experiment to get familiar with

1. Search problems
2. Depth First Search Algorithm Implementation
3. Breadth First Search Algorithm Implementation
4. Best First Search Algorithm

Tools Required

Jupyter IDE

Course Coordinator
– Course Instructor
– Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Introduction

Artificial Intelligence (AI) has revolutionized the way we approach search problems, offering innovative solutions across various domains. By leveraging techniques like informed search, uninformed search, constraint satisfaction problems, and reinforcement learning, AI algorithms can effectively analyze vast amounts of data, make informed decisions, and find optimal solutions.

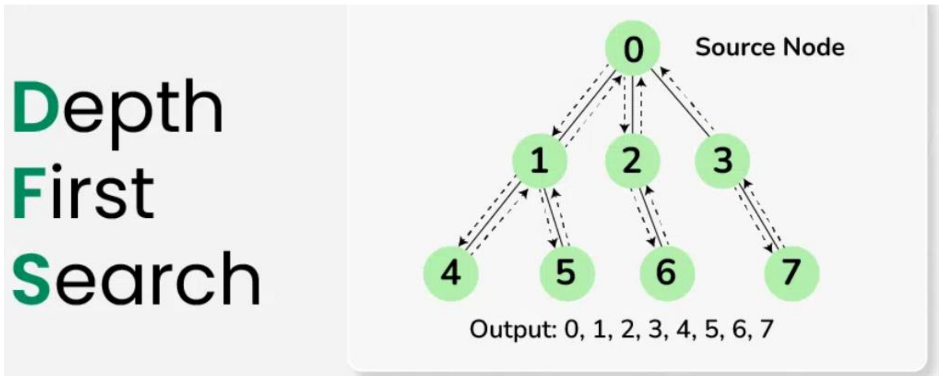
One of the key AI techniques for search problems is informed search. This approach utilizes heuristic functions to estimate the distance to the goal, guiding the search process towards promising paths. Algorithms like A* search and greedy best-first search combine heuristic information with path costs to find efficient solutions. Uninformed search, on the other hand, explores the search space without any prior knowledge of the goal, relying on techniques like breadth-first search and depth-first search.

Constraint satisfaction problems (CSPs) involve finding assignments for variables that satisfy a set of constraints. AI algorithms like backtracking and local search are commonly used to solve CSPs. Backtracking involves exploring different assignments and backtracking when a conflict arises, while local search starts with a random solution and iteratively improves it by making small changes. Reinforcement learning, another powerful AI technique, enables agents to learn optimal actions through trial and error by maximizing rewards. Q-learning and deep Q-networks (DQNs) are popular reinforcement learning algorithms that have been applied to various search problems.

AI in search problems has a wide range of applications, including game AI, robotics, optimization, recommendation systems, and natural language processing. In game AI, AI agents use search algorithms to make strategic decisions, such as in chess, Go, and real-time strategy games. In robotics, search algorithms help robots plan paths, navigate environments, and perform tasks like object manipulation. AI can also be used to optimize various problems, such as scheduling, resource allocation, and logistics. Recommendation systems leverage search techniques to suggest products, movies, or other items based on user preferences.

Depth First Search
Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

DFS Pseudocode (recursive implementation)
The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.



```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:

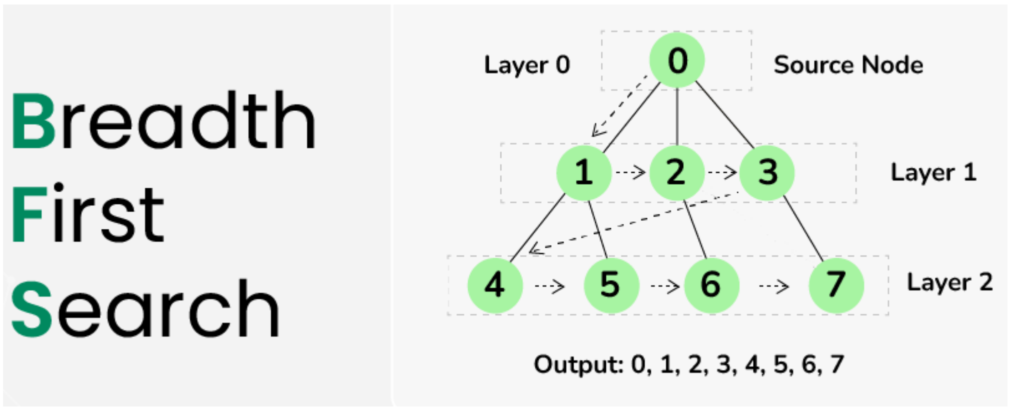
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Code:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

Implement the given code and write your output below:

Breadth first search
Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.



BFS pseudocode

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
```

Breadth First Search Algorithm
A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

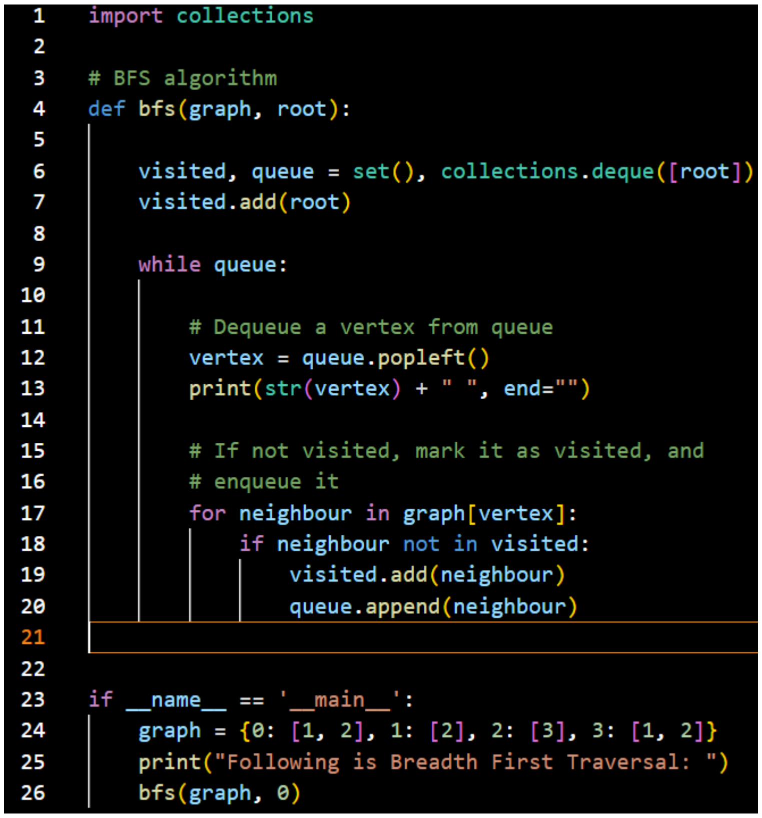
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.

2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

Code:



```python
import collections

# BFS algorithm
def bfs(graph, root):

    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:

        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)


if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

Implement the given code and write your output below:

Exercise:

1. Explain the Best First Search Algorithm (BFS) and its variants;
   1. Greedy Best First Search
   2. A Best First Search* (A*)?
2. Write down implementation code of each variant of BFS.
3. Write down the difference among searching algorithm discussed so far.