Student: Dăscălescu Dana
Group: 507 - Artificial Intelligence
Date: March 16, 2023

Uninformed Search

BFS vs DFS

1 Problem description

In classroom 308, students taking the Artificial Intelligence exam are seated in rows of two-person benches arranged in two columns. Some spots are free, meaning there are no students occupying those seats. Diana wants to help her friend Calin cheat on the exam by sending him the answers through a message. However, there are restrictions on how messages can be passed. Students can only pass the message to the bank colleague behind or in front of them, or the one right beside them, but not diagonally. Additionally, passing messages between rows is challenging since the teacher can easily notice it. Only the penultimate and final benches on each row can be used for message transfer. To ensure that she does not get lost in the classroom, Diana wants to include the path she needs to take from one colleague to the next on the note.

2 Solution

We will solve this problem using two uninformed search algorithms: DFS and BFS.

DFS and BFS are two fundamental algorithms used in graph traversal. Both algorithms aim to visit all the vertices in a graph, but they differ in the order of traversal and the way they explore the graph. DFS explores a graph by visiting as far as possible along each branch before backtracking, while BFS explores a graph layer by layer, visiting all the nodes at a given depth before moving on to the next depth. DFS uses less memory than BFS, as it only needs to remember the path from the start node to the current node, while BFS needs to store all the nodes at the current depth. DFS is typically implemented using a stack, while BFS is typically implemented using a queue. The time complexity of both algorithms is $O(V + E)$, where V is the number of vertices and E is the number of edges. Both algorithms have advantages and disadvantages. Whether to choose DFS or BFS depends on the specific problem and the graph characteristics. DFS is great for finding a path between two nodes, while BFS is perfect for finding the shortest path. DFS is a memory saver, while BFS is a memory hog

To solve the given problem, we can model it as a graph where the nodes represent the students in the classroom, and the edges represent the possible paths that the message can take between them. We can start by identifying the neighbors of each student and creating the corresponding edges in the graph.

We can represent each student using a tuple (row, column) where the row represents the row number (1 to n) and the column represents the column number (1 or 2). The neighbors of a student can be found as follows:

- The student in front of them (if they are not in the first row)
- The student behind them (if they are not in the last row)

- The student to their left (if they are in the even column)
- The student to their right (if they are in the odd column)

However, we also need to take into account the *constraints* of the problem. Only the penultimate and final banks on each row can be utilized for message transfer, which means that not all neighbors are valid. For example, a student in the first column cannot pass the message to a student in the second column, and a student in the penultimate bank cannot pass the message to a student in the first or second bank of the same row. Passing the message between columns is not allowed in all cases, meaning students in different columns cannot be neighbors. With these constraints in mind, we can create the graph and apply BFS or SFs to find the shortest path from Diana's location to Calin's location.

3 Implementation

BFS: We can start by adding Diana's location to the queue, then visit all its valid neighbors according to the constraints. For each neighbor, we add it to the queue and mark it as visited. We continue this process until we reach Calin's location or we visit all reachable nodes without finding him.

```
    def BFS(self):
        """
            Performs a breadth-first search on a given problem and returns
            the path from the start node to the scope node
        """
        frontier = deque([self.problem_configuration.start_node])
        visited = set()

        while frontier:
            node = frontier.popleft()
            visited.add(node)

            if node == problem.scope_node:
                path = self.reconstruct_path(node)
                print("The length of the path:", len(path))
                self.show_path(path)
                return path

            successors = self.get_successors(node)
            for successor in successors:
                if successor not in visited and successor not in frontier:
                    frontier.append(successor)

        print("Path not found!")
```

To keep track of the path, we can use a list that maps each visited node to its parent node. This way, when we find Calin's location, we can backtrack from it to Diana's location using the parent pointers and record the path.

DFS: To adapt the previous code for Depth-First Search (DFS), we can start by adding Diana's location to the stack, then visit its valid neighbors according to the constraints. For each neighbor, we add it to the stack and mark it as visited. We continue this process until we reach Calin's location or we visit all reachable nodes without finding him.

```python
    def DFS(self):
        """
            Performs a depth-first search on a given problem and returns
            the path from the start node to the scope node
        """
        frontier = [self.problem_configuration.start_node]
        visited = set()

        while frontier:
            node = frontier.pop()
            visited.add(node)

            if node == problem.scope_node:
                path = self.reconstruct_path(node)
                print("The length of the path:", len(path))
                self.show_path(path)
                return path

            successors = self.get_successors(node)
            for successor in successors:
                if successor not in visited and successor not in frontier:
                    frontier.append(successor)

        print("Path not found!")
```

Output:

```
*****************************************************************************************
The length of the path: 18
Reconstructed path:
        diana ---> daria ---> andrei ---> petru ---> ana ---> bianca ---> victoria
        ---> dionisie ---> marius ---> iustina ---> denisa ---> daniel ---> razvan
        ---> iuliana ---> alex ---> daniela ---> mihaela ---> calin
Execution time of BFS search: 0.0011 seconds
*****************************************************************************************

*****************************************************************************************
The length of the path: 20
Reconstructed path:
        diana ---> daria ---> andrei ---> petru ---> ana ---> maria ---> anastasia
        ---> cosmin ---> victoria ---> dionisie ---> marius ---> iustina ---> mihail
        ---> matei ---> razvan ---> iuliana ---> alex ---> daniela ---> mihaela
        ---> calin
Execution time of DFS search: 0.0008 seconds
*****************************************************************************************
```