Student: Dăscălescu Dana
Group: 507 - Artificial Intelligence
Date: April 17, 2023

Uninformed Search

BFS vs DFS

1 Introduction

The purpose of this project is to develop an application that utilizes two uninformed search techniques to solve a problem. In this document, we will outline the problem statement and provide an overview of the uninformed search techniques that will be used to solve it.

2 Task requirements and 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 Călin 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 the message doesn't get lost in the classroom, Diana wants to write on the note the path the message must take from one colleague to the next.

3 Solution

3.1 Problem and goal formulation

One possible way to tackle the problem is to model it as a graph, where each node represents a student in the classroom, and the edges represent the possible paths that the message can take between them. To create the graph, we can represent each student using a tuple (row, column), where the row represents the row number (from 1 to n), and the column represents the column number (1 or 2).

To find the neighbors of a student, we need to consider the students who are in front of, behind, left of, or right of them, depending on their position. For instance, a student in an even column has a neighbor to their left, whereas a student in an odd column has a neighbor to their right. However, we must also take into account the constraints of the problem, which specify which neighbors are valid for message transfer.

For example, the penultimate and final banks on each row are the only ones that can be utilized for message transfer. This means that 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. Additionally, passing the message between columns is not allowed in all cases, meaning students in different columns cannot be neighbors.

Given these constraints, we can create the graph from the maze representation of the problem configuration and apply an uninformed search strategy, such as BFS or DFS, to find the shortest path from Diana's location to Călin's location.

3.2 Search strategies

Our approach to solving this problem involves using four uninformed search algorithms: depth-limited search, breadth-first search, depth-limited search, and iterative deepening depth-first search. However, we will primarily focus on comparing DFS and BFS.

DFS and BFS are two widely used algorithms in graph traversal, both aiming to visit all the vertices in a graph, but differing in the order of traversal and the way they explore the graph. DFS explores a graph by traversing as far as possible along each branch before backtracking, whereas BFS explores a graph layer by layer, visiting all the nodes at a given depth before moving to the next depth. In terms of memory usage, DFS requires less memory than BFS since it only needs to remember the path from the start node to the current node, while BFS has to store all the nodes at the current depth. DFS is commonly 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 DFS and BFS have their advantages and disadvantages. Choosing which algorithm to use depends on the specific problem and the graph's characteristics. DFS is great for finding a path between two nodes, as it tends to reach the goal node faster than BFS. However, DFS does not necessarily find the shortest path, and it may get stuck in an infinite loop if the graph has cycles. On the other hand, BFS guarantees that the first path it finds is the shortest one. However, BFS may require more memory than DFS, and it may take longer to reach the goal node.

In addition to DFS and BFS, we will also test DLS (depth-limited search) and IDS (iterative deepening search) to solve this problem. DLS is a variant of DFS that limits the maximum depth of search, while IDS is a hybrid algorithm that combines the benefits of both BFS and DFS. IDS performs DFS with increasing depth limits until the goal is found.

For this problem, IDS and DLS are useful as we have prior knowledge of the graph's depth or structure. As a heuristic, we set the depth limit of the search at two times the number of rows in the class. DLS can be used to limit the search to a specific depth, preventing DFS from getting stuck in infinite loops or exploring the entire graph unnecessarily. IDS, on the other hand, is a memory-efficient algorithm that ensures the optimal solution while searching the graph layer by layer.

For this problem, since we are looking for the shortest path, BFS may be a more suitable choice. However, we will also test the other algorithms in terms of time execution and memory usage to ensure that we have found the optimal solution.

4 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.

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.

```python
    def BFS(self) -> List[Node]:
        """
        Performs breadth-first search on a given problem and returns the path from the start node to the scope node.

        Returns:
        path (List[Node]): The path from the start node to the scope node.
        """

        # Initialize the frontier and visited set
        frontier = deque([self.problem_configuration.start_node])
        visited = set()

        # Loop through each node in the frontier until the scope node is found or the frontier is empty
        while frontier:
            node = frontier.popleft()
            visited.add(node)

            # If the scope node is found, reconstruct the path and return it
            if node == problem.scope_node:
                path = self.reconstruct_path(node)
                print("The length of the path:", len(path))
                self.show_path(path)
                return path

            # Otherwise, add the node's successors to the frontier if they have not been visited or added to the frontier before
            successors = self.get_successors(node)
            for successor in successors:
                if successor not in visited and successor not in frontier:
                    frontier.append(successor)

        # If the scope node is not found, print an error message and return None
        print("Path not found!")
        return None
```

Output:

```
*************************************************************************************
The length of the path: 18
Reconstructed path:
        diana ---> daria ---> andrei ---> petru ---> ana ---> bianca ---> victoria ---> dionisie ---> marius ---> iustina ---> denisa ---> daniel ---> razvan ---> iulia ---> alex ---> daniela ---> mihaela ---> calin
Execution time of BFS search: 0.9096 seconds
Maximum memory usage of BFS: 40.5546875
*************************************************************************************
```

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) -> List[Node]:
        """
        Performs depth-first search on a given problem and returns the path from the start node to the scope node.

        Returns:
```

```python
        path (List[Node]): The path from the start node to the scope node.
        """

        # Initialize the frontier and visited set
        frontier = [self.problem_configuration.start_node]
        visited = set()

        # Loop through each node in the frontier until the scope node is found or the
    frontier is empty
        while frontier:
            node = frontier.pop()
            visited.add(node)

            # If the scope node is found, reconstruct the path and return it
            if node == problem.scope_node:
                path = self.reconstruct_path(node)
                print("The length of the path:", len(path))
                self.show_path(path)
                return path

            # Otherwise, add the node's successors to the frontier if they have not been
     visited or added to the frontier before
            successors = self.get_successors(node)
            for successor in successors:
                if successor not in visited and successor not in frontier:
                    frontier.append(successor)

        # If the scope node is not found, print an error message and return None
        print("Path not found!")
        return None
```

Output:

```
*******************************************************************************
The length of the path: 20
Reconstructed path:
        diana ---> daria ---> andrei ---> petru ---> ana ---> maria ---> anastasia --->
    cosmin ---> victoria ---> dionisie ---> marius ---> iustina ---> mihail ---> daniel
    ---> razvan ---> iulia ---> alex ---> daniela ---> mihaela ---> calin
Execution time of DFS search: 0.8857 seconds
Maximum memory usage of DFS: 40.55859375
*******************************************************************************
```

DLS: To adapt the previous code for Depth-Limited Search (DLS), we can modify the DFS algorithm to include a depth limit parameter. Instead of adding all unvisited neighbors to the stack, we only add neighbors that are within the current depth limit. Once we have visited all the nodes at the current depth, we backtrack to the previous node and increment the depth limit. We continue this process until we reach Calin's location or we reach the maximum depth limit without finding him.

By limiting the maximum depth of search, DLS can prevent the algorithm from getting stuck in an infinite loop or exploring the entire graph unnecessarily. However, setting the depth limit can be challenging as it depends on the problem's specific characteristics and constraints. A depth limit that is too small may not reach the goal node (as set in our case), while a depth limit that is too large may take a long time to execute and consume too much memory.

In our problem, we experimented with using a heuristic to set the maximum depth limit for IDS. The heuristic we used was based on the idea that the longest possible path in the graph is twice the number of rows since each row can be visited at most twice (once on the way to the destination and once on the way back). However, when we implemented this heuristic, we found that the depth limit was being reached before finding the shallowest goal.

This failure of the heuristic highlights an important point about IDS and heuristics in general. While heuristics can be very effective in guiding search algorithms toward promising regions of the search space, they are not always accurate and can sometimes lead to suboptimal or even incorrect results. In the case of our problem, the heuristic based on the longest possible path did not take into account the specific constraints and characteristics of the graph, which led to the failure of the algorithm to find the optimal solution.

Output:

```
************************************************************************************
The maximum depth to search to: 18
Path not found!
Execution time of DLS search: 0.8077 seconds
Maximum memory usage of DLS: 40.55859375
************************************************************************************
```

IDS: To adapt the previous code for the IDS algorithm to our problem, we can set an initial depth limit of one and perform a DFS search up to that depth limit, beginning with Diana's location. If Calin is not discovered, the depth limit is increased and the search is repeated until Calin is located or the maximum depth limit is reached.

IDS is a potentially beneficial algorithm for our problem because it can locate the shortest path while minimizing memory consumption. By conducting a series of DFS searches with increasing depth limits, IDS is able to effectively search through the graph without storing all the visited nodes in memory, which is particularly useful when the graph is large.

However, the optimal depth limit must be selected with care to achieve a balance between efficiency and optimality. If the depth limit is too small, the algorithm may not discover the optimal solution, whereas if it is too large, it may be too time-consuming. Therefore, we must carefully adjust the depth limit based on the particulars of our problem.

Output:

```
************************************************************************************
Current depth limit 1
The maximum depth to search to: 1
Path not found!
Current depth limit 2
The maximum depth to search to: 2
Path not found!

        .
        .
        .

Current depth limit 21
The maximum depth to search to: 21
The length of the path: 20
Reconstructed path:
        diana ---> daria ---> andrei ---> petru ---> ana ---> maria ---> anastasia --->
    cosmin ---> victoria ---> dionisie ---> marius ---> iustina ---> mihail ---> daniel
    ---> razvan ---> iulia ---> alex ---> daniela ---> mihaela ---> calin
Execution time of IDS search: 0.8185 seconds
Maximum memory usage of IDS: 40.6875
************************************************************************************
```