Assignment: Graph Algorithms – \_II

1. **Draw Minimum Spanning Tree** 
   1. **Draw minimum spanning tree for the below graph.**

A diagram of a network

Description automatically generated with low confidence



Cost = 7

1 + 1 + 2 + 3 = 7

Steps:

1. choose smallest cost( we can choose either (C-B) or (E-C)): we’ll choose (E-C).

A diagram of a network

Description automatically generated with low confidence



2. next we choose next smallest cost from edges connected to vertex C or vertex E & repeat process making sure there are no cycles.

A picture containing yellow

Description automatically generated

* 1. Draw spanning Tree that is not minimum

A picture containing circle, clock, screenshot, line

Description automatically generated

cost = 8

1. **MST Implementation:**
   1. Implement Prims algorithm. **Prims(G).** Include function in the file **MST.PY.** You can either use brute force approach or ***priority queue***. Try to see if you can come up with a solution using *priority queue.*
   2. What is the difference between the Kruskal’s and the Prim’s algorithm?

Both algorithms aim to find a minimum spanning tree for a given graph, but they differ in their strategies for selecting edges and growing the MST. Kruskal's algorithm focuses on sorting edges and considering them individually, while Prim's algorithm focuses on growing the tree from a single vertex outward.

* Approach:
  1. Kruskal's algorithm is a greedy algorithm that starts with an empty set of edges and iteratively selects the edge with the minimum weight that does not create a cycle.
  2. Prim's algorithm, on the other hand, starts with a single vertex and gradually grows a tree by adding the minimum weight edge that connects the tree to a new vertex.
* Data Structure:
  1. Kruskal's algorithm typically uses a disjoint set data structure to efficiently detect cycles and maintain connectivity information.
  2. Prim's algorithm often uses a priority queue to select the minimum weight edges.
* Edge Selection:
  1. Kruskal's algorithm considers all the edges in the graph and selects the minimum weight edge that does not create a cycle.
  2. Prim's algorithm selects the minimum weight edge that connects the current tree to a new vertex, expanding the tree gradually.
* Spanning Tree Construction:
  1. Kruskal's algorithm constructs the minimum spanning tree by iteratively adding edges.
  2. Prim's algorithm constructs the minimum spanning tree by iteratively growing a connected subgraph from a single vertex.

Both algorithms aim to find a minimum spanning tree for a given graph, but they differ in their strategies for selecting edges and growing the MST. Kruskal's algorithm focuses on sorting edges and considering them individually, while Prim's algorithm focuses on growing the tree from a single vertex outward.

**3. Apply Graph traversal to solve a problem (Portfolio Project Problem):**

The algorithm step by step and explain the reasoning behind each step:

1. Create an empty queue to store the cells to be explored.
   * We use a queue data structure to implement the breadth-first search (BFS) traversal. It helps us explore the cells in a first-in, first-out order, ensuring that we process the cells level by level.
     1. BFS: **Guarantees finding the shortest path from the source to the destination.** Since BFS explores cells in a level-by-level manner, the first path it finds will be the shortest. This is because BFS visits cells in non-decreasing order of their distance from the source. As soon as it reaches the destination, it means the path followed is the shortest possible because BFS would have explored all closer cells before reaching the destination.
     2. DFS: The main reason DFS may not find the shortest path is its depth-first nature, lack of predefined search order, and potential for backtracking. While DFS can find valid paths, it does not prioritize the shortest path and may explore longer paths before discovering shorter ones. **In contrast, BFS is more reliable for finding the shortest path as it explores cells in a breadth-first manner, ensuring completeness and optimality.**
2. Enqueue the source cell with its position and an empty path.
   * We start the traversal from the source cell, so we enqueue it in the queue. Along with the position of the cell, we also maintain a path that represents the sequence of cells visited so far. At the beginning, the path is empty.
   * Here we can use either a queue or deque data structure. It really is preference.
3. Create an empty set to keep track of visited cells.
   * We need to keep track of the cells that have been visited to avoid revisiting them. Using a set allows for efficient lookup and insertion operations, ensuring that we don't process the same cell multiple times.
4. While the queue is not empty:
   * We continue the traversal as long as there are cells in the queue that need to be explored. If the queue becomes empty, it means we have explored all reachable cells and haven't found the destination.

a. Dequeue a cell from the front of the queue.

* + We dequeue the next cell from the queue to process it. This cell represents the current position from which we will explore its neighboring cells.

b. If the dequeued cell is the destination cell, return its path.

* + If the current cell is the destination cell we were looking for, it means we have found a valid path. We can return the path taken to reach this cell as the result.

c. If the dequeued cell is not visited:

* + We only proceed if the current cell has not been visited before. This check prevents unnecessary exploration of already visited cells.

i. Mark the cell as visited. - We mark the current cell as visited by adding it to the set of visited cells. This ensures that we don't process the same cell multiple times.

ii. For each neighboring cell (left, right, up, down) that is a valid and unvisited empty cell: - We explore each neighboring cell of the current cell. To be considered valid, the neighboring cell must be within the puzzle boundaries and must be an empty cell (not a barrier). Additionally, we ensure that the neighboring cell has not been visited before.

vbnet

1. - For each valid neighboring cell, we enqueue it into the queue along with its position and the updated path. The updated path is obtained by appending the corresponding direction (L, R, U, D) to the path of the current cell.
2. If the queue becomes empty and the destination cell is not reached, return None (no valid path found).
   * If we exhaust all possible cells to explore and the destination cell is not reached, it means there is no valid path from the source to the destination. In this case, we return None to indicate the absence of a path.

By following these steps, the algorithm explores the puzzle cells systematically, using BFS to efficiently search for the destination cell while keeping track of the visited cells and constructing the path along the way.