Arai Leyva
Andrew Conti
Eliot Park
Quan Truong

Project 3

Algorithm 1: The Spread of Fire in a Forest

To solve this problem, we would use the Breadth-First-Search method. The idea is that BFS would process the matrix array one "layer" at a time. The algorithm will always return the shortest number of steps.

a. **Pseudocode**

```
Algorithm1(matrix_array) {
        If matrix_array is empty:
                return -1

        rows = number of rows in matrix_array
        cols = number of columns in matrix_array
        queue ← empty queue to hold positions of burned trees
        healthy_trees = 0
        days = 0

        For each cell in matrix_array:
                If matrix_array[r][c] == 2:
                        add (r, c) to queue   // burned tree
                Else if matrix_array[r][c] == 1:
                        healthy_trees += 1    // increment count of healthy trees overall
        If healthy_trees == 0:
                return days    // no trees to burn
        If queue is empty:
                return -1   // no fire to spread

        While queue is not empty:
                size = len(queue) // size of the queue
                For i from 1 to len(queue):
                        Pop the last tuple (r,c) added from the queue
                        // Check UP
                            nr ← r - 1
                            nc ← c
                            IF nr and nc are inside bounds AND forest[nr][nc] == 1:
                               forest[nr][nc] ← 2
```

```
                    healthy_trees -= 1
                    queue.enqueue((nr, nc))

                // Check DOWN
                nr ← r + 1
                nc ← c
                IF nr and nc are inside bounds AND forest[nr][nc] == 1:
                    forest[nr][nc] ← 2
                    healthy_trees -= 1
                    queue.enqueue((nr, nc))

                // Check LEFT
                nr ← r
                nc ← c - 1
                IF nr and nc are inside bounds AND forest[nr][nc] == 1:
                    forest[nr][nc] ← 2
                    healthy_trees -= 1
                    queue.enqueue((nr, nc))

                // Check RIGHT
                nr ← r
                nc ← c + 1
                IF nr and nc are inside bounds AND forest[nr][nc] == 1:
                    forest[nr][nc] ← 2
                    healthy_trees -= 1
                    queue.enqueue((nr, nc))

        If queue is not empty:
                days += 1
    If healthy_trees == 0:
        Return days
    Else:
        Return -1
}
```

b. Analysis and Efficiency

The wildfire spread algorithm is very efficient because it goes through each cell in the forest only once. First, it checks the whole grid to find burned and healthy trees, which takes time based on the size of the grid. Then, it uses a method called Breadth-First Search (BFS) to spread the fire from all burned trees to nearby healthy ones. Each tree can only catch fire once, and for every tree, the algorithm checks up to four nearby spots (up, down, left, right). So, the total time it takes depends on the number of rows and columns in the grid. We say the time and space used by this algorithm are both “O(m × n),” which just means it grows in a straight line with the size of the grid. This makes it one of the best and most practical ways to solve the problem.

c. **Output**

/usr/bin/python3 "/Users/quantruong/CSU Fullerton Dropbox/Hung Truong/CPSC 335 Algo/CPSC335_Project3/algo1.py"
Sample 1: [[2, 1, 1], [1, 1, 0], [0, 1, 1]]
Output:4
Sample 2: [[2, 1, 1], [0, 1, 1], [1, 0, 0]]
Output:-1
Sample 3: [[0, 2]]
Output:0
Sample 4: [[1, 1, 1, 2], [1, 2, 0, 1], [1, 1, 0, 1]]
Output:2

Algorithm 2: Delivery Route Planning

a. **Pseudocode**

```
Algorithm2(routes, source, destination, max_steps):
   nodes ← new empty Set
   add source to nodes
   add destination to nodes

   For each route IN routes:
      add route[0] to nodes  // Add the starting node to each route
      add route[1]  to nodes  // Add the ending node of each route
   End For

   // Get number of unique nodes
   num_nodes ← size of nodes

   // Initialize a constant representing infinity
   infinity ← a very large number  // Unreachable nodes

   // Map each node to a unique index based on sorted order
   node_to_index ← map each node in sorted(nodes) to unique index

   // Initialize the distance array with INF for all nodes
   distance ← array of size n filled with INF
```

```
// Set distance for the source node to 0
distance[node_to_index[source]] ← 0

// Loop k + 1 times to account for paths with at most max_stops
For each iteration from 0 to max_stops:
    distance_backup ← copy of distance
    For each route in routes:
        from_node ← route[0]
        to_node ← route[1]
        price ← route[2]

        // Get indices of the starting and ending nodes of the current route
        from_index ← node_to_index[from_node]
        to_index ← node_to_index[to_node]
        // Update cost if new cost is lower than current
        distance[to_index] ← minimum of (distance[to_index], distance_backup
            [from_index]  + price)
    End For
End For

If distance[node_to_index[destination]] IS infinity:
    Return -1
Else:
    Return distance[node_to_index[destination]]
End If
End Algorithm
```

b. Analysis and Efficiency

Building the nodes set; iterates over m routes, adds 2 per route makes it O(m). Creating node_to_index; sorts the node list and enumerates, which takes O(n log n). Initializing distance array is O(n). The main loop (k + 1 iterations) makes a copy of distance O(n) with the inner loop over all routes O(m) → total for loop is O((k ✕ m). With the final check being constant, the TOTAL time complexity becomes → O(m + n log n + (k + 1) ✕ m), which subsequently simplifies to O(n log n + k ✕ m). In terms of efficiency, the algorithm's performance depends on the relative sizes of n, m, and k. If k is small and m is not as large as n, the algorithm performs reasonably well. However, if k and m are large, the algorithm's runtime can increase considerably, the O(k ✕ m) term dominates, and performance worsens. It all depends on the value of k, and words well for small values of k.

c. **Output**

arai@arai-ubuntu:~/Documents/CPSC335_Project3$ /bin/python3 /home/arai/Documents/CPSC335_Project3/algo2.py
Routes: [[0, 1, 100], [1, 2, 100], [0, 2, 500]], Source: 0, Destination: 2, Stops: 1, **Cheapest Route: 200**
Routes: [[0, 1, 100], [1, 2, 100], [0, 2, 500]], Source: 0, Destination: 2, Stops: 0, **Cheapest Route: 500**
Routes: [[0, 1, 100], [1, 2, 100], [0, 2, 500]], Source: 0, Destination: 3, Stops: 1, **Cheapest Route: -1**
Routes: [], Source: 0, Destination: 1, Stops: 0, **Cheapest Route: -1**