CSE541-HW2

xinliu

October 2023

Discuss with Charles Troutman

1 Question 1

Algorithm Pseudocode

The algorithm to accomplish this is as follows:

```
ALGORITHM SIMPLE_PATHS(G, s, t)
    TOPOLOGICAL_SORT(G)
    Initialize an array DP[1...|V|] to infinity
    Determine the vertices between s and t in the topologically sorted order
    and store them in array v[0...k], where v[0] = s and v[k] = t
    Set DP[k] = 1 (base case)
    FOR i = k-1 DOWNTO 0
        FOR each vertex u adjacent to v[i]
            IF u is between v[i] and t in the sorted order
                DP[i] += DP[position of u in v]
    RETURN DP[0]
```

Correctness Proof

To prove the correctness of the SIMPLE_PATHS algorithm, we will use induction on the vertices of the graph, following their topological order.

Base Case: For the base case, consider the last vertex t. There is exactly one simple path from t to itself, hence $DP[k] = 1$.

Inductive Step: Assume that for a vertex $v[i]$, where $i < k$, the number of simple paths from each vertex $v[j]$, with $j > i$, to t has been correctly computed in $DP[j]$. Then, the number of simple paths from $v[i]$ to t is the sum of all simple paths from vertices adjacent to $v[i]$ (and between $v[i]$ and t in the topological order) to t, which is stored in $DP[i]$. This sum is computed iteratively for each $v[i]$, starting from $v[k-1]$ and moving backwards to $v[0]$ (which is s).

2 Question2,3,4,5

2.1 Question 2

In an $m \times n$ matrix with horizontal wrapping, each cell in the first column can advance to three adjacent cells in the next column. The top and bottom cells can move to an additional fourth cell due to the wrapping, but this does not increase the total count significantly, as the overlap is minimal. Therefore, the total number of distinct paths is given by:

$$\text{Total Paths} = m \times 3^{(n-1)} \tag{1}$$

2.2 Question 3

The largest number of distinct shortest paths in a wrapped $m \times n$ matrix depends on the distribution of weights within the matrix. If the weights are uniformly distributed, then potentially every path could be the shortest path. so the number of shortest paths is equal to the total number of paths in this case.

Therefore:

$$\text{Max Shortest Paths} = m \times 3^{(n-1)} \tag{2}$$

2.3 Question 4

```
ALGORITHM FindShortestPath(matrix)
    m, n = size of matrix

    // Create a cost matrix to store the minimum path cost to reach each cell
    Initialize cost[m][n] with infinity
    Initialize path[m][n] to store the actual path

    // Set up the cost for the first column
    for i from 0 to m-1:
        cost[i][0] = matrix[i][0]

    // Fill in the cost matrix considering wrapping
    for j from 1 to n-1:
        for i from 0 to m-1:
            // Get the wrapped indices for the top and bottom rows
            above = (i - 1 + m) mod m
            below = (i + 1) mod m

            // Update the cost considering the three possible moves
            cost[i][j] = min(cost[above][j-1], cost[i][j-1],

            cost[below][j-1]) + matrix[i][j]

            // Update the path to reflect the chosen move
            if cost[i][j] == cost[above][j-1] + matrix[i][j]:
                path[i][j] = 'from_above'
            else if cost[i][j] == cost[i][j-1] + matrix[i][j]:
                path[i][j] = 'from_left'
            else:
                path[i][j] = 'from_below'

    // Find the minimum cost in the last column
    min_cost = min(cost[0...m-1][n-1])

    // Backtrack to find the actual path
    actual_path = []
    for j from n-1 down to 0:
        // Find the row index with the minimum cost for this column
        i = row index of min_cost in column j

        // Prepend the direction to the actual path
        actual_path.prepend(path[i][j])

        // Move to the previous column according to the direction
        if path[i][j] == 'from_above':
            i = (i - 1 + m) mod m
        else if path[i][j] == 'from_below':
            i = (i + 1) mod m
        // No need to change i if the move was 'from_left'

    return actual_path
```

2.4 Question 5

The asymptotic worst-case running time of the dynamic programming algorithm for a cylindrical matrix is $O(mn)$, which is polynomial in the size of the matrix. This time complexity accounts for the additional consideration of wrapping.