CSE431 HW4

Siqi Cheng, 50388579

Problem 1

Based on Zuckerberg's autobiographical movie: "The Social Network" we come to a hypothetical question. I'm trying to develop a social networking application similar to Facebook, where one of the key features is a recommendation system for "friends people might know". Given a list of users and their friendships, your task is to find out, for each user, which friends they might know. Principle: If A and B are friends, and B and C are friends, but A and C are not friends, then the system should recommend C to A, because they have a common friend B.

Given an undirected graph G(V, E) containing n nodes, where V is the set of nodes and E is the set of edges. Two nodes are friends if an edge exists between them. So we have to find for each node i a list of people who may know each other.

So we have the problem:

Design a dynamic programming algorithm to implement the "Facebook" feature.

We can use a 2D Boolean dynamic programming array dp[n][n], where dp[i][j] is true if node i and node j are friends. If dp[i][k] and dp[k][j] are both true, but dp[i][j] is false, then dp[i][j] can be set to true, indicating that i and j could be friends. For all pairs of nodes (i, j) should be like this.

B--A—C

| |

E—D In this graph, A is friends with B and C, B is friends with A and E, E is friends with B and D.

Now, we have the DP matrix should be: (1 means that the user can potentially be friends, and 0 means they cannot

A B C D E

A 0 1 1 0 1

B 1 0 1 1 1

C 1 1 0 0 1

D 0 1 0 0 1

E 1 1 1 1 0

Set dp[i][j] = 1 if node i and node j are directly connected. Set dp[i][j] = 0 if node i and node j are not directly connected, or i equals j.

If dp[i][k] == 1 and dp[k][j] == 1 but dp[i][j] == 0, then set dp[i][j] = 1. This indicates that i and j could be friends because they have a common friend k. (for node k)

For every node i, a friend is a node j if dp[i][j] = 1 and i and j are not already friends.

The time complexity is O(n^3), base on this algorithm, it uses three for loops to make sure it checks into every user's social circle.

The space complexity is O(n^2), this is because this algorithm is a 2D matrix to store the friendship status between every possible pair of users. The size of this matrix is n x n.

Problem 2

Define dp[i][j][k][d] to be the maximum fraction that can be obtained when k turns have been made at position (i, j) and the direction of the last move is d. Here d = 0 means down and d = 1 means right. Now we have:

If the last move was down (d=0), we have two choices:

Continue moving down: dp[i][j][k][0] = grid[i][j] + dp[i-1][j][k][0] (assuming i-1 >= 0)

Make a 90-degree turn and move right: dp[i][j][k+1][1] = grid[i][j] + dp[i-1][j][k][0] (assuming i-1 >= 0 and k+1 <= K)

If the last move was right (d=1), we also have two choices:

Continue moving right: dp[i][j][k][1] = grid[i][j] + dp[i][j-1][k][1] (assuming j-1 >= 0)

Make a 90-degree turn and move down: dp[i][j][k+1][0] = grid[i][j] + dp[i][j-1][k][1] (assuming j-1 >= 0 and k+1 <= K)

Base cases: dp[0][0][0][0] = dp[0][0][0][1] = grid[0][0].

Therefor final answer is the maximum over dp[n-1][n-1][k][d] for all k, d.

```
function (grid, K):
    n = length(grid)
    dp = 5-D array of size [n][n][K+1][2] filled with -infinity
    //Base cases
    dp[0][0][0][0] = grid[0][0]
    dp[0][0][0][1] = grid[0][0]
    for i from 0 to n-1:
        for j from 0 to n-1:
            for k from 0 to K:
                for d from 0 to 1:
                    if dp[i][j][k][d] is not -infinity:
                        if i+1 < n:
```

```
            // move down without turn
            dp[i+1][j][k][0] = max(dp[i+1][j][k][0], dp[i][j][k][d] + grid[i+1][j])
            //move down with turn if last move was right
            if d == 1 and k+1 <= K:
                dp[i+1][j][k+1][0] = max(dp[i+1][j][k+1][0], dp[i][j][k][d] + grid[i+1][j])

        if j+1 < n:
            // move right without turn
            dp[i][j+1][k][1] = max(dp[i][j+1][k][1], dp[i][j][k][d] + grid[i][j+1])
            // move right with turn if last move was down
            if d == 0 and k+1 <= K:
                dp[i][j+1][k+1][1] = max(dp[i][j+1][k+1][1], dp[i][j][k][d] + grid[i][j+1])

max_score = -infinity
for k from 0 to K:
    for d from 0 to 1:
        max_score = max(max_score, dp[n-1][n-1][k][d])
return max_score
```

Time Complexity: The time complexity of this algorithm would be O(n^2 * K), where n is the size of the grid and K is the maximum number of turns.

Space Complexity: The space complexity would also be O(n^2 * K) because we need to maintain a 4-dimensional DP array.

Problem 3

This problem can be solved using BFS, because BFS it explores all possible moves step by step, extending the search boundaries equally in all directions. In order to find the shortest path, an attempt is made to find the minimum number of rolls to pass through the level.

the block can be in two states: standing (occupies one cell) and flat (occupies two adjacent cells). We represent these states as (x1, y1, x2, y2). If the block is standing, (x1, y1) equals (x2, y2). If it's flat, (x1, y1) and (x2, y2) are the coordinates of the two cells occupied by the block.

The initial state is the cell marked with S. It starts in the standing position, so the initial state is (x3, y3, x3, y3), where (x3, y3) is the coordinate of the cell marked with S.

Our goal is for the block to fall into the cell labeled D in a standing position. We need to define that in each state the cube can try to move in one of four directions: up, down, left and right. However, movement is only valid if it does not cause the block to fall off the grid or fall into an empty cell while lying flat. A move is also invalid if it causes half of the block to lie flat in an empty cell and the other half to not be in a cell. The cost of a move is always 1, so the cost of a path is 1.

After construct the graph, use a shortest path algorithm, Dijkstra's algorithm, to find the shortest path from the starting state to the target state.

It assumes

valid_moves(state) - this function generates all possible moves that can be made from the current state.

make_move(state, move) - this function updates the current state according to the move selected.

is_goal(state) - this function checks if the current state is the goal (destination) state.

```
function Bloxorz_ShortestPath(grid):
    Initialize state as the starting cell in standing position
    Initialize an empty Priority Queue pq
    Initialize an empty dictionary distance_map with state as key and 0 as value
    pq.push(state, 0)
    while pq is not empty:
        current_state = pq.pop()
        if is_goal(current_state):
            return distance_map[current_state
        for move in valid_moves(current_state):
            next_state = make_move(current_state, move)
            new_distance = distance_map[current_state] + 1

            if next_state not in distance_map or new_distance < distance_map[next_state]:
                distance_map[next_state] = new_distance
                pq.push(next_state, new_distance)
    return no_solution
```

The time complexity of this algorithm is of O(E + V log V). For a graph with E edges and V vertices, Dijkstra's algorithm has a time complexity of O(E + V log V) .

the time complexity of the algorithm would be O(n log n).