A4_Report_122090337

Problem_1

1 Initial Analysis

It is actually a the shortest path question with some certain limitations (requirements) in the graph. We can solve it by the typical shortest path algorithm with some added conditions in the detail.

Why do we need to find the shortest path in this question? It is because that we are required to find **the least changes** to make to satisfy the demands. To find the least changes, we have to find the shortest way from i to j (or we may waste the changes on blocks that players will not pass on). Now the question is direct enough for solving.

2 Possible Solutions for the problem

Dijkstra

Then the task is how to find the shortest path in a graph. As taught by the lectures, Dijkstra approach will be a very nice way to solve this question. It has a rather small time complexity, and we will not encounter the troubles like circle or negative length.

3 How to Solve the Problem

Now the task is very direct and easy: find the shortest path, and check how many directions of each point is different from the supposed ones.

Point here means the block in the domain (map). The main function is just used for accepting the given data, and then check where are the start and end points. The most important thing is the Dijkstra function.

```
public static void dijkstra(Point[][] domain, Point start, int rows, int cols) {
    PriorityQueue<Point> points = new PriorityQueue<>(Comparator.comparingInt(p -> p.change));
    boolean[][] visitedPoints = new boolean[rows][cols];
```

- Initializes a priority queue that stores `Point` objects. The queue orders the points based on the `change` value in ascending order. This is used to determine the order in which nodes are processed.
- The 2D boolean array keeps track of which points in the domain have already been visited. It helps to avoid reprocessing the same point.

```
    points.add(start);
    visitedPoints[start.x][start.y] = true;
    domain[start.x][start.y].change = 0;

    while (!points.isEmpty()) {
        Point currentNode = points.poll();

        int nextPos_x = currentNode.x + 1;
        int nextPos_y = currentNode.y;
```

- These lines add the starting point to the priority queue, mark it as visited, and set its `change` value to 0, indicating that there is no cost to reach the start point.
- The loop continues as long as there are points left to process in the queue. They removes and returns the point with the lowest `change` value from the priority queue, representing the current node to process.

```
        if (nextPos_x>= 0 && nextPos_x < rows && nextPos_y >= 0 && nextPos_y < cols) {
            int isAddChange = ((currentNode.dir == 'i' || currentNode.dir == 's') ? 0 : 1);
            if (currentNode.change + isAddChange < domain[nextPos_x][nextPos_y].change) {
                domain[nextPos_x][nextPos_y].change = currentNode.change + isAddChange;
                points.add(domain[nextPos_x][nextPos_y]);
                visitedPoints[nextPos_x][nextPos_y] = true;
            }
        }
```

- The rest of the code in the snippet is about checking and updating the neighboring points (in this case, the point directly below the current point) based on their `change` values and directions (`dir`). If a more efficient path is found (i.e., a lower `change` value), the neighbor point is updated and added to the queue for further processing.

While the code blocks behind this part are used for checking the status in other directions (because the player can go to any direction within the border). So the logic is exactly the same.

This is the typical form of Dijkstra algorithm. Thus the shortest path will be found (with the required restrictions of course).

But one thing to note: even though the time complexity is good, we need to still focus on efficiency of functions and structures. Calling a function will cost time. **On OJ, calling outside functions for too many times will cause TLE in Q9.** Thus, all parts in Dijkstra is written together without calling outside functions.

Problem_2

1 Initial Analysis

This question can be divided into three subtasks:

- construct the graph with a good data structure
- operate on the graph, add some new edges when the conditions are satisfied
- display the result of the graph using BFS as required

The question is already very direct. Now we can think of how to solve each subtasks.

2 Possible Solutions for the problem

Constructing the graph

- **Graph Initialization**: First, initialize the graph with the given n vertices and m edges. This can be done using either an adjacency list or an adjacency matrix. Adjacency matrix will be chosen here because the data is not sparse. It is also really easy to code and understand.

Adding new edges

- **Adding New Edges**: Iterate over each vertex i (from 1 to n). For each vertex, check all its neighbors. If there is a k-times relationship in terms of IDs between any two neighbors (i.e., u = kv or v = ku), and there is no existing edge between u and v, add a new edge between them.
- **Avoiding Duplicate Edges**: When adding new edges, ensure that duplicate edges are not added. A two-dimensional boolean array can be used here.

BFS

- **Initialize BFS**: Use a queue to facilitate the BFS process. Initially, enqueue the starting vertex s. The process taught in the lecture will be adopted.
- **Traversal Process**: While the queue is not empty, dequeue a vertex. Visit all its neighbors in ascending order of their IDs, and enqueue those neighbors if they haven't been visited.
- **Recording Order**: As vertices are visited, record their order of visitation. This will be the final BFS order to output.

3 How to Solve the Problem

The codes used for constructing the graph and BFS are very typical forms taught in the lectures. So no need to elaborate them too much. The most important part in this question is the `edgeChanging` and `addNeighborEdge` functions.

```
static void edgeChanging(Graph graph, int n, int k) {
    for (int node = 1; node <= n; node++) {
        List<Integer> neighbors = graph.graph.get(node);
        if (neighbors != null) {
            addNeighborEdge(graph, neighbors, k);
        }
    }
    graph.graph.values().forEach(Collections::sort);
}
```

Purpose

- This method iterates through each node in the graph and applies the `addNeighborEdge` method to each node's neighbors to add new edges between neighbors that meet specific conditions.
- After traversing and adding all new edges, it sorts the neighbors list of each node.

Implementation

- **Iterating Nodes**: Iterates through each node from 1 to n.
- **Fetching Neighbors**: Retrieves the list of neighbors for each node.
- **Adding New Edges**: Calls the `addNeighborEdge` method to add edges between neighbor pairs that meet the condition, if the node has neighbors.
- **Sorting Neighbors**: Sorts the neighbor list of each node in ascending order after all new edges are added.

```
static void addNeighborEdge(Graph graph, List<Integer> neighbors, int k) {
    for (int mainNeighbor : neighbors) {
        for (int subNeighbor : neighbors) {
            if (mainNeighbor != subNeighbor && (mainNeighbor == subNeighbor * k || subNeighbor == mainNeighbor
* k)) {
                graph.addEdge(mainNeighbor, subNeighbor);
            }
        }
    }
}
```

Purpose

- Adds new edges between the neighbors of a given node that meet specific conditions.

Logic Implementation

- **Double Loop**: Iterates through the neighbors list using two nested loops.
- **Condition Checking**: For each pair of neighbors, checks if they satisfy the given k-times relationship (i.e., one neighbor is k times the other or vice versa).
- **Edge Addition**: If a pair of neighbors that meet the condition is found, and they are not the same node, adds an edge between these two neighbors.

Thus, all the requirements in the question are satisfied. So the answers will be satisfying.