Authors: Aidan Dalgarno-Platt and Liam Coleman

Problem 1)

Algorithm 1

This algorithm fails to properly function as it detects a cycle when there is not one.



* When performing the DFS the algorithm goes A-B-C-D. It adds the nodes to the visited stack for each step along the way. Eventually it reaches D and can’t go any farther. As a result we must backtrack to A. Unlike problem 2, we don’t remove the nodes from the stack while backtracking. Then we travel to D. Since we are still in the iteration we detect this as a cycle and return true. If this graph wasn’t directed, then it would indeed be a cycle.

Algorithm 2

This algorithm always correctly detects cycles, no matter where the cycle is. Take the same directed graph from last time:



* The algorithm picks A to start and marks it as visited, then places it in the recursive call stack. It then goes to B, marks it as visited, and places it in the recursive call stack. It does the same with C, and then D.
* This algorithm backtracks during the DFS like the first algorithm, but critically, it removes each node from the stack once each recursive call has finished. So, when it backtracks from D to C, D is removed from the stack, then from C to B, C is removed from the stack, and from B to A, B is removed from the stack. That means only A remains in the stack when it visits D, meaning it correctly does not find a cycle

Problem 2)

Our algorithm works as a two part dynamic programming solution calculating first the shortest path to every tile without dynamite from the start, then the shortest path to every tile without dynamite from the end. It traverses tiles using BFS, and at the end, it finds all pairs of shortest starts-ends between a wall, and the shortest of such pairs is the answer.

Heart of the Solution:

What

* maze\_dp[i][j][0] is the shortest path from tile (0, 0) to tile (i, j) without using dynamite from the start of the maze (the top left corner)
* maze\_dp[i][j][1] is the shortest path from tile (d1, d2) to tile (i, j) without using dynamite from the end of the maze (the bottom right corner)

How

* Calculating maze\_dp[i][j][0]
  + If maze[i][j] is a wall, then maze[i][j][0] =
  + Else maze[i][j][0] = min(maze[i - 1][j][0] + 1, maze[i + 1][j][0] + 1, maze[i][j - 1][0] + 1, maze[i][j + 1][0] + 1)
* Calculating maze\_dp[i][j][1]
  + If maze[i][j] is a wall, then maze[i][j][1] =
  + Else maze[i][j][1] = min(maze[i - 1][j][1] + 1, maze[i + 1][j][1] + 1, maze[i][j - 1][1] + 1, maze[i][j + 1][1] + 1)

Where

* The solution is some pair of shortest-path-from-start and shortest-path-from-end that are separated by a single wall. The algorithm iterates through every wall, adds the sum of every adjacent shortest-path-from-start and shortest-path-from-end pair combination to a list, and then takes the minimum in that list

First, a matrix representing the maze, with 0s as paths and 1s as walls, is constructed, which takes O(d1\*d2) time. Next, a matrix of adjacencies is constructed for each tile, which doesn’t consider walls adjacent to other walls, which also takes O(d1\*d2) time. Then two more matrices of the same size are constructed, one 3D representing the dynamic programming array (size of 2\*d1\*d2) and another to keep track of whether each tile has been visited or not, parallel to the maze matrix. Both of these take O(d1\*d2) time.

Using Breadth First Search, the program traverses through the maze matrix, visiting every tile exactly once, and populates the first entry in each element of the dynamic programming array representing the shortest path without dynamite from the start to each tile. Unreachable tiles are initialized with a value of , which includes wall tiles and path tiles that cannot be accessed from the start without blowing up a wall tile. This takes O(d1\*d2) time.

Next, the algorithm does the same thing but starting from the end tile instead of the start tile, and it populates the second entry in each element of the dynamic programming array rather than the first. Likewise, this takes O(d1\*d2) time.

Finally, the algorithm iterates through every wall and adds every possible adjacent shortest-path-from-start and shortest-path-from-end pair to a list. There are a maximum of 12 pairs per wall that can be added (all permutations of the four adjacent tiles excluding a tile pairing up with itself), and there are guaranteed to be fewer than d1\*d2 walls, so this section runs, worst case, in O(12\*d1\*d2), which is O(d1\*d2).

Every section of this algorithm takes O(d1\*d2) time independently, so the total running time is O(d1\*d2).

Problem 3)

This algorithm uses a modified version of Prim’s Algorithm. First we run Prim’s Algorithm on the all reliable nodes which should take either O(n2) or O(m\*log(n)). This will give us the cost of the MST for the reliable nodes. Then we loop through every unreliable node and connect it with the least expensive reliable node. We then add the cost of the reliable nodes to the cost of connecting each unreliable node..

The algorithm first reads in all the input, which takes O(V + E), or O(n + m), where n is the number of vertices (spies) and m is the number of edges (communication channels). Next, it creates an adjacency matrix of size n2, which takes O(n2) time, and initializes all the relevant edges in the matrix, which takes O(m) time. It creates a selected\_node list of size n, which takes O(n) time, then it finds the first reliable spy to be the root, which takes O(n) time. It then creates a verticies\_selected list of size n, which takes O(n) time, before getting to the meat of the algorithm.

Next, Prim’s algorithm begins, looping through every adjacent edge to each selected node and taking the minimum cost one, although there’s the added caveat that neither node can be an unreliable spy. Therefore, the total cost of all connections between all reliable spies is calculated this way. This takes O(n\*log(n) + m\*log(n)), or just O(m\*log(n)) time total.

Next, the algorithm iterates through every row of the adjacency matrix, and for each unreliable node, it finds the minimum edge between it and a reliable spy (as there can never be a connection between two unreliable spies) and adds that to the total cost calculated in the previous section. This takes O(n2) time.

Finally, the algorithm loops through every node to check that all reliable nodes have been selected at some point. If a node hasn’t been selected, then that node was never connected to the graph, there is no solution, and “NONE” is printed. Otherwise, the total cost is printed. This takes O(n) time.

Overall, this algorithm takes O(n2 + m + n + n + n + m\*log(n) + n2 + n), or O(n2 + m\*log(n)). This is equivalent to whichever one is larger, O(n2) or O(m\*log(n)), so the algorithm is either O(n2) or O(m\*log(n)).