LinelandNavigation Writeup

In my algorithm, I represented Lineland as a digraph. The nodes of the graphs were locations in Lineland, while the edges were forwards or backwards jumps (by which I mean legal movements) that LePoint could make between these locations. I kept a boolean array of all possible locations that LePoint might visit (from 0 to a bit past the goal). Whenever a location was declared to be a mine through a call to `addMinedLineSegment()`, I flipped its element in the array to `true`.

I implemented `solveIt()` as a Breadth-First Search. I did so by building a BFS tree and determining the depth of each location as it was added to the tree; in a BFS tree, depth represents the minimum number of edges needed to reach the node. When I found the goal, I returned its depth. Since each edge is a single jump, the depth of the goal is the minimum number of jumps necessary to reach it. To ensure that the BFS tree would not contain any mines, I marked each location as it was mined in a boolean array (as described above), and then used that same array in my BFS to determine if a node was already visited, ensuring that mined nodes would not be visited, even if they would otherwise have led to a shorter path.

In order to reduce the cost of the algorithm, I made some modifications to BFS. Rather than creating a graph ahead of time and passing it to the algorithm, I created the graph as I traversed it. Whenever I visit a node `location`, I add `location + m` to the BFS queue (the forward jump from that node). I then add backwards jumps, starting at `location - 1`, but as soon as I find a node that has already been added to the queue (as proven by its depth being nonzero, so it isn't confused with a mine), I stop looking for more edges incident on `location`.

The reason I can do this is that my algorithm has an invariant that all nodes before a node added to the queue were also added to the queue (unless they were mines). This can be proven using (*sigh*) induction. In the base case of jump 0 (before the first jump), there are no (valid) nodes before node 0, so all the nodes before it are on the queue. In the inductive case of jump *n*, I only need to add nodes working backwards until I reach a node already on the queue, which we will call node *q*. Since all the locations before *q* were already added to the queue (using strong induction), all the nodes before *n* have now been added to the queue (unless they were mines).

I added this order of growth analysis before I realized you didn't care, but I don't want to take it out after spending time on it. The order of growth of any BFS is O(V + E). V in this case is `g`, because no node before 0 can be considered, and only one node at or beyond `g` can be considered before the algorithm returns. E in this case is proportional to V. I can perform up to V forward jumps, because each valid vertex (non-mine) has exactly one forward jump edge. Additionally, the number of back jump edges (by which I mean forward edges pointing backwards on the number line, not actual back edges) is also proportional to V. Each vertex can be examined for the first time, and be added to the queue, once. Once I examine an edge and discover its vertex to have already been added to the queue, I do not check any edges before it. I can examine a number of edges already on the queue to the number of vertices for which I am examining back jump edges. If a vertex is a mine, it can also be examined no more than twice: once for the first time, whether from a forward or backward jump; and once when being revisited when checking for nodes from the non-mine vertex immediately before the mine. All other edges are not even examined, and so might as well not exist. Because of this, `solveIt()`'s order of growth is proportional to the number of vertices, or O(`g`).