My approach to this problem was to solve it using a Breadth First Search algorithm. I had to make a few tweaks in the implementation of this algorithm to handle the given constraints of the assignment, like the constant jump magnitude and mined line segments.

In the constructor, all I did was set my class variable g and m to the g and m passed into the constructor and did some error checking. In my addMinedLineSegment method I add all positions in range from start to end to my mined line segment set.

I use the BFS algorithm in the solveIt method. To set up for the algorithm I create a queue to use for the breadth-first search. Then I initialize an integer array called distances to the size of MAX_FORWARD(which is 1,000,000) + 1 and fill the array with infinity at every index. Then I add 0 to the queue and set the 0 index in distances to 0 because the distance to the 0 node is always 0. While the queue isn't empty I set the current position to the first element in the queue and if the current position is greater than or equal to g then we have reached the endpoint and return the value of distances[currentPosition]. Otherwise, I make another queue of the current positions neighbor's by doing Queue<Integer> neighbors = getNeighbors(currentPosition);

The getNeighbors(int currentPosition) method is where I take into consideration this assignment's special constraints. I create a queue of neighbors and if the currentPosition + m(magnitude of the jump) is <= MAX_FORWARD and does't fall on a mined line segment then I add currentPosition + m to my neighbors queue. What this is accomplishing is only adding nodes that can be reached by a jump of m to the possible neighbors of the currentPosition. Then I add all backward neighbors from 1 step behind to m-1 steps behind to the queue as long as the node doesn't fall within a mined line segment.

Once I break out of the getNeighbors method and return to the solveIt method I iterate through the neighbors queue for (int neighbor : neighbors) if (distances[neighbor] == Integer.*MAX_VALUE*) meaning the node has not been visited yet then I update the distance to the neighbor distances[neighbor] = distances[currentPosition] + 1 and add neighbor to the queue. If no path to g was found I return 0.