Maze Game

By

Erik Marrero

Nely Rodriguez

**CS 253 - 70**

**CENTRAL CONNECTICUT STATE UNIVERSITY**

**Introduction:**

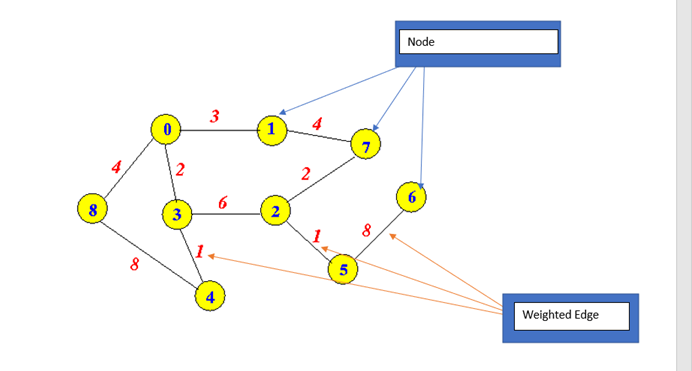
Maze’s have existed for centuries, the word maze dates back to the 13th century. A maze is a puzzle with set paths that may contain deadends. In a maze the objective is to find your way out to the other side. Since mazes are set up with deadends it is expected for someone to get lost before finding the optimal way out. This project takes the approach of finding the optimal path out of a maze in game form. Since this is a game we added user input as a key part of this project. The user input is important because it was used to compare the computer generated path to a human generated path. The computer generated path uses a graphing algorithm to find the best path to take in the maze. The optimal way out of the maze is calculated using a weighted graph and dijkstra's algorithm. Dijkstra's algorithm finds the optimal path by doing calculations and is a straightforward set of thinking. The algorithm will be able to calculate which path to take based on the paths between the nodes in the weighted graph. When using a weighted graph we need to take into consideration it is a simple graph with weighted edges. When doing calculations for a weighted graph we need to use a set of vertices, a set of edges and a number that is set by length. With the weighted graph and Dijkstra's algorithm we can find the best and quickest way out of the maze.

Since this project was based on finding a graphing algorithm that interested us we chose Dijkstra's algorithm. After picking the algorithm we decide that a game would be the best implementation of the algorithm. The game we chose was a maze because it is the best way we can show the shortest path. The reason we also chose to do a maze was because we could implement user input better. The user input added a different dimension to the project. Most programs that implemented Dijkstra's algorithm only displayed the shortest path. The results from the user will be used to compare the accuracy of the algorithm.

**Weighted Graph:**

A graph is a conceptual notation used to represent the connection between pairs of objects. A graph consists of Vertices which are Interconnected objects in a graph. Vertices are also known as nodes. They also contain Edges which are the links that connect the vertices.There are two types of graphs a the first one is a directed graph which its edges have direction, i.e., edges go from one vertex to another.The next one is a Undirected graph and in an undirected graph, edges have no direction. A Weighted Graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers. Weighted graphs are used for applications where we need to take into account some cost or measurement between vertices of the graph. For example, the weights can represent the time it costs to travel from one location to another. Or, they can represent a measurement, such as the distance between the locations.

We picked a weighted graph because we wanted to calculate the shortest of three paths in a maze. So what we did was wrote code that takes the edges of the program and compared them to each other and whichever edge was the shortest we saved until we found a better edge with a smaller weight / distance from point start to point end.



As Shown in the picture above the weighted edges would be the line going from one node/vertices to another. In this example the weighted edge from node 0 to node 1 is 3. Weighted edge between 1 and 7 is 4. From node 1 to 2 node the weight is 6. Etc… The point of explaining this was to give a visual understanding of what a weighted edge is and what A node/ vertices is. So for an example let's say we are trying to get from node 8 to node and we want to choose the shortest path to get to node . From node 8 we would go from 8->0->3->2->5

With weights of 4,2,6,1 which would have a total weight of 13. If we were to calculate each way to get from node 8 to node 5 this would have the smallest weight therefore being the best way to get from nodes 8 to 5.

|  |
| --- |
| Graph Implementation:  Graph Maze = new Graph(18);    Maze.makeEdge(0, 1, 1);  Maze.makeEdge(0, 11, 1);  Maze.makeEdge(1, 2, 1);  Maze.makeEdge(1, 3, 1);  Maze.makeEdge(3, 4, 1);  Maze.makeEdge(3, 10, 1);  Maze.makeEdge(4, 5, 1);  Maze.makeEdge(4, 6, 1);  Maze.makeEdge(6, 7, 1);  Maze.makeEdge(6, 8, 1);  Maze.makeEdge(8, 9, 1);  Maze.makeEdge(8, 10, 1);  Maze.makeEdge(10, 11, 1);  Maze.makeEdge(11, 12, 1);  Maze.makeEdge(12, 13, 1);  Maze.makeEdge(13, 14, 1);  Maze.makeEdge(13, 15, 1);  Maze.makeEdge(15, 17, 1);  Maze.makeEdge(15, 16, 1);    \* Adds vertices to the graph in a bidirectional way, with the weight of the edge  \*/  public void makeEdge(int vertex1, int vertex2, int time) {  vertices[vertex1][vertex2] = time;  vertices[vertex2][vertex1] = time;  } |

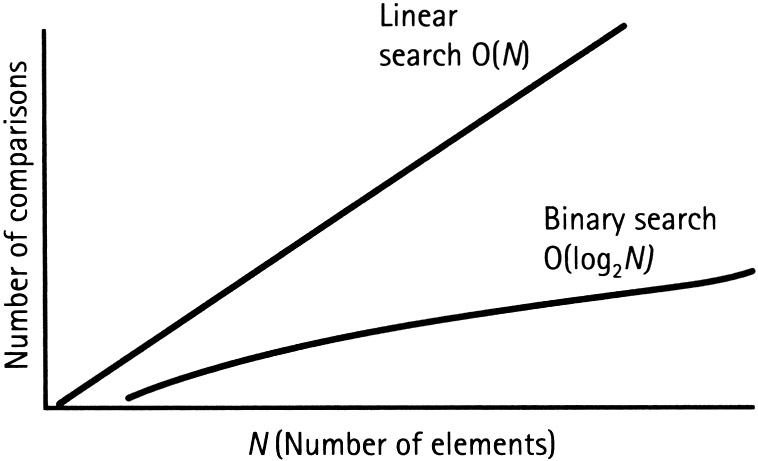
The way this was implemented in our project is by first making the edges, This is done with the function makeEdges which puts the 2 vertices and the weight and then we store the new edge into a graph maze. After all the edges are in the graph they are all compared to each other to see which has the shortest path.

**Dijkstra's algorithm:**

Dijksta’s algorithm is used to find the shortest path between nodes in a graph. According to “Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction” by Estefania Cassingena Navone, the main way to use the algorithm is by finding “the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree”(Navone 2). Navone explains that the basics of dijksta’s algorithm consists of 4 basic steps to find the shortest path tree. The first basic step is finding the shortest path between the initial node and the other nodes in the graph. The second basic step is the algorithm keeps updated on the value between nodes. When a shorter path is discovered between nodes the value is updated. The third step is to make sure when the shortest path is found the node is marked. The node needs to be marked because otherwise the algorithm will run again for that node and it will cause time efficiency errors in the code. The final basic step is to repeat the previous 3 steps until all nodes in the graph are marked as visited which means that the shortest path has been found for each node. Before we apply time complexity to dijksta’s algorithm let's talk about what time complexity is. As discussed in “An Introduction to the Time Complexity of Algorithms'' by Aditya Dehal time complexity is

“the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time). The algorithm that performs the task in the smallest number of operations is considered the most efficient one in terms of the time complexity”(Dehal 1).

Time complexity is associated with Big O Notation, the notation lets us see the time complexity between to set variables. Big o notation can use linear search and binary search as a basis for the algorithms implemented. The following graph shows big o notation for linear and binary search.



If we take a look at the binary aspect it shows that the time complexity of a binary search is calculated using O(log2N). This is important to know because binary elements are implemented into the algorithm. For Dijksta’s algorithm the time complexity and big o notation comes from the fact that the algorithm is implemented with a binary heap. The biggest difference between the 2 calculations is that the algorithm impaments the number of edges(E) and the number of vertices(V) in the graph used buy Dijksta’s algorithm.The runtime efficiency is calculated using the time complexity of Dijksta’s algorithm. The base of the algorithm is O(log2N) to get it to fit the algorithm. We need to add the edges and vertices that are used to find the shortest path, this creates 2 calculations after which we combine to get the final calculation of the algorithm. For the first calculation we first add vertices to the function making it O(V). Next we need to add a vertex which makes the calculation O(log(v)). The next step is to multiply edges in to create O((E)log(v)). The next calculation is almost the same as the first but instead of multiplying by the edges we need to multiply by the vertices. So the calculation would be O((v)log(v)). To get the simplified calculation we just combine the two to get O((E+V)log(v)).

The reason we chose Dijksta’s algorithm is because we wanted to get the most accurate calculation of the shortest path. Since we used a weighted graph we need to use Dijksta’s algorithm because it only works with positive values. Dijksta’s algorithm minimizes the chances of errors because it cannot function with negative values which fits perfectly with the set values of edges and vertices in the weighted graph. This works best with our maze because our maze contains more than one way of getting out of it. Dijksta’s algorithm affects the maze because it takes into account all of the paths we created to exit the maze. The algorithm makes sure the best path is represented. The following pseudocode is how we decided to implement the algorithm into our game code.

|  |
| --- |
| function Dijkstra(Graph, source):  for each vertex v in Graph: // Initialization  Distance of V is set to infinity  node -1 is set to undefined // Previous node in optimal path from source  Distance of source node is set to 0 // Distance from source to source  i is the set of all nodes in the graph // all nodes in the graph are unoptimized  while i is set with a condition: // main loop  k = node in i with smallest distance  remove k from i  for each neighbor v of i: // where v has not been removed  add is distance k and the distance between(k, v)  if add < distance of v  Distance v is = add  v-1 := k  return v-1 |

The pseudocode form above is just a basic way of implementing the algorithm. We used the pseudocode as a guide for our more detailed and specific version of the algorithm. With the way we programed the maze there are deadends and we needed to take those into account with our algorithm. Our use of the algorithm takes in the multiple ways to exit the maze and applies the algorithm to each path. The following code is the detailed and accurate implementation for the maze game.A

|  |
| --- |
| /\*\*  \* Implementation of the Dijkstra algorithm  \*/  public List<Integer> path(int from, int to) {  // Create lists of travel costs so far  int cost[] = new int[vertices.length];  // Create list with vertices predecessors to the visitor today  int prev[] = new int[vertices.length];  // Create lists with unvisited vertices,  Set<Integer> unvisited = new HashSet<>();  // Initializes the cost of the initial vertex with 0  cost[from] = 0;  for (int v = 0; v < vertices.length; v++) {  if (v != from) {  // Initializes the cost of the other initial vertices with a very high value  cost[v] = Integer.MAX\_VALUE;  }  // Defines all values ​​as undefined  prev[v] = UNDEFINED;  // Adds all vertices to the unvisited listv  unvisited.add(v);  }  //Search  while (!unvisited.isEmpty()) {  // Obtains the lowest cost unvisited vertex  int near = closest(cost, unvisited);  // We removed the vertex from the unvisited list  unvisited.remove(near);  // For each neighbor we calculate the total cost of traveling to it  for (Integer neighbor : getNeighbors(near)) {  // We add the cost of the node so far plus the cost of the edge connecting the two vertices  int totalCost = cost[near] + getCost(near, neighbor);  // If the total cost is less than the current cost to  if (totalCost < cost[neighbor]) {  // Update the cost  cost[neighbor] = totalCost;  // We mark the predecessor vertex  prev[neighbor] = near;  }  }  // Verification if the processed vertex is the destination  if (near == to) {  return makePathList(prev, near);  }  }  //No path found  return Collections.emptyList();  } |

The results of Dijksta’s algorithm provides us with the shortest path from the entry of the maze to the exit of the maze. The algorithm provides us with the optimal path which is great because especially when we can compare the optimal path to the path chosen by the user. The algorithm above runs through each path thoroughly to find the shortest path.

**Evaluate The steps in the code:**

First in our program what we did was create a graph class we started by calculating the number of vertices with the numvertices. Then we created a Function named makeEdge that Adds vertices to the graph bidirectionally, with the weight as well as a function remove edge to remove them. The graph class represents our adjacency matrix. In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal.In the Graph class we created functions to calculate the number of vertices and then Adds vertices to the graph in a bidirectional way, with the weight of the edge. An Unidirectional Graph Is a graph with edges which are directed only in one direction, like in the case of a directed graph. Bidirectional Graph: Is a graph in which each edge is given an independent orientation (or direction, or arrow) at each end. Then we created a remove edge function which Removes vertices of the graph in a bidirectional way. Then we created a get cost function to just return the edge cost. After that is the getNeighbors function which returns all vertices connected to the vertex passed by parameter. Where we perform a for loop on the line that represents the vertex, and add the vertices whose index is greater than 0. Then we implement the Dijkstra algorithm.

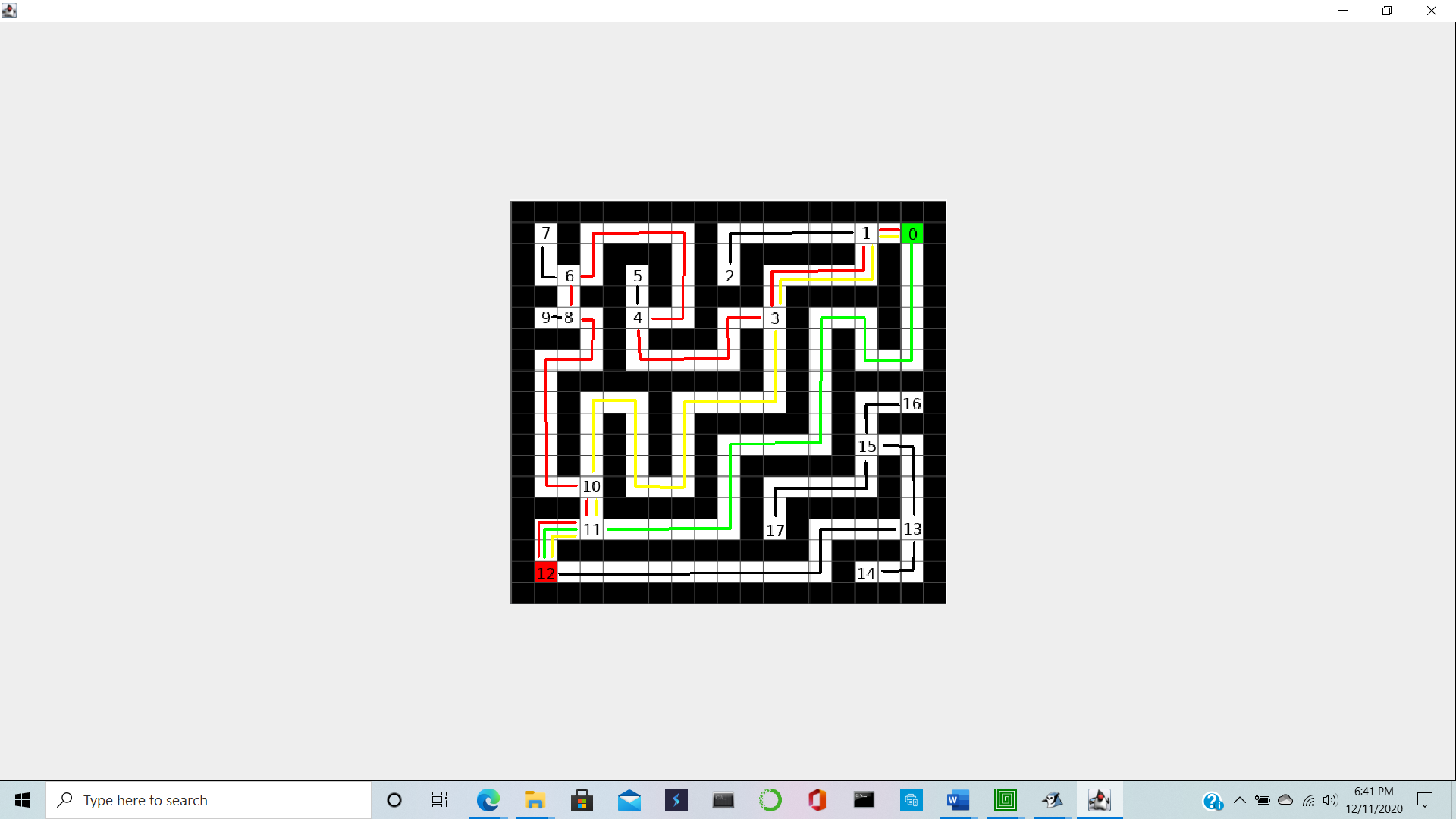
We Create lists of travel costs so far. Then we Create list with vertices predecessors to the visitor toda.after that Create lists with unvisited vertices. Next Initializes the cost of the other initial vertices with a very high value and defines all values ​​as undefined.lastly we use unvisited.add(v); which adds all vertices to the unvisited list. Then what we do is create a loop to search. First it Obtains the lowest cost unvisited vertex and // we removed the vertex from the unvisited list.For each neighbor we calculate the total cost of traveling to it.We then add the cost of the node so far plus the cost of the edge connecting the two vertices.If the total cost is less than the current cost to and then we Update the cost.We mark the predecessor vertex.we then need verification if the processed vertex is the destination. The next function Obtains the lowest cost unvisited vertex by returning The min Index. After that we get a path to the target vertex and make a path list. The last step was to create the main method which will put these functions to use.

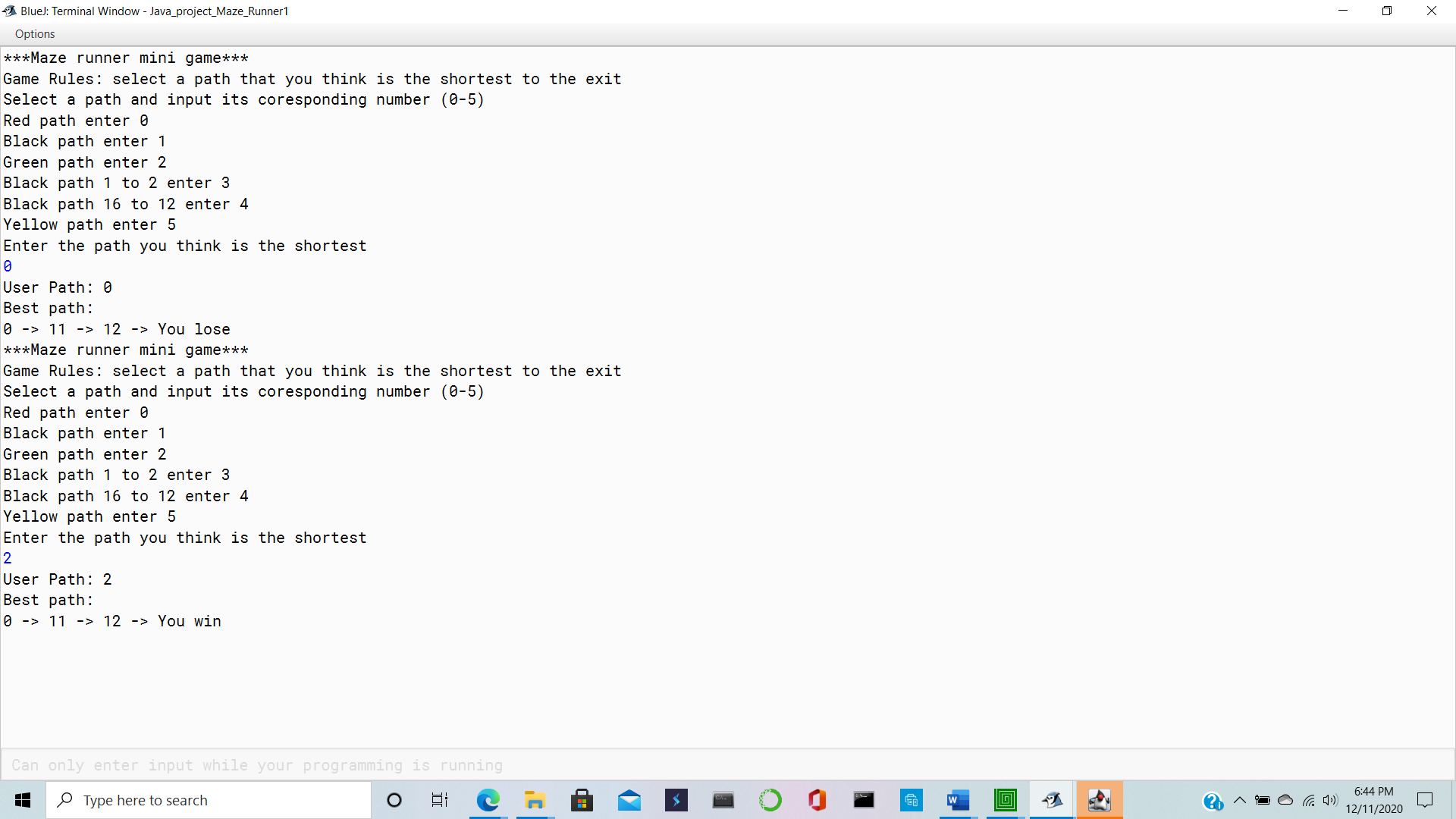
**Summary of paper and end results:**

The result of our project was a mini game where the person gets to choose which path they think is the shortest path. If they can guess whether the red, yellow, green, or black path is the shortest path they win the game. Each colored path corresponds to a number as the game explains which color goes to which number. After the user inputs their choice of what path they think is the shortest. The program then uses a weighted graph and Dijksta’s algorithm to calculate which path was the shortest. After calculating the shortest path the algorithm then checks to see if the user chooses the shortest path. If the player did choose the shortest path the game would display that the player won otherwise the game would display that the user lost. As demonstrated below. The first user input was 0 and the player lost because that isn’t the shortest path and then the user input path 2 which is the green path and it displayed that the user won. When the user choose their path they only saw the visual representations so even though the maze was completed they did not pick the shortest path that was calculated by the algorithm.

Overall our objective was to use a weighted graph in our program and implement Dijksta’s algorithm to develop a game where the user tries to guess the shortest path. With our implementation of the weighted graph and Dijksta’s algorithm we were able to meet our objective of producing a maze game with user input. We also applied the theories in a way where we could successfully find the shortest path out of the maze. Without our use of graphs and a graphing algorithm our program would have not been successful.

**Pictures of end result:**

****

****

**Work Cited**

A Akshaya Kumar Biswal @akshayakumarbis added a Short note 28 Jul 2018, et al.

“Binary Search.” *Techtud*.

“A Brief History of Mazes.” *National Building Museum*, 17 Mar. 2017.

“CS 312 Lecture 20 Dijkstra's Shortest Path Algorithm.” *Shortest Path Algorithm*,

Cornell University .

Dehal, Aditya. “An Introduction to the Time Complexity of Algorithms.” *FreeCodeCamp.org*

,FreeCodeCamp.org, 11 June 2019.

Huang, Chung-Yang (Ric), et al. “Fundamentals of Algorithms.” *Electronic Design Automation*

, Morgan Kaufmann, 29 Jan. 2010.

Navone, Estefania Cassingena. “Dijkstra's Shortest Path Algorithm - A Detailed and

Visual Introduction.” *FreeCodeCamp.org*, FreeCodeCamp.org, 18 Nov. 2020.