Dijkstra's Algorithm Applied to Chutes and Ladders

By Jonathan Ruel

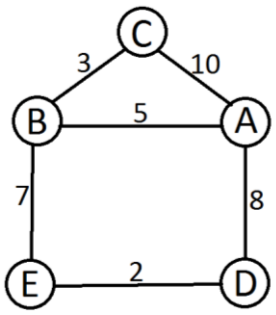
In graph theory, the shortest path problem is defined by the problem of finding a path between two vertices such that the sum of the weights of its constituent edges is minimized. An example of this type of problem would be to find the quickest route from one town to another on a map. In this example, vertices represent locations and edges represent the distance relationship between locations. Other real world applications that contain the shortest path problem are, GPS mapping, network protocols, and telecommunication networks.

In order to solve the shortest path problem there are many algorithms available to find the shortest path between two vertices. Dijkstra's algorithm, being one example, solves the single pair, single source, and single destination shortest path problems for a graph with non negative edge path costs. It does this by constructing the shortest path tree edge by edge. At each step of the algorithm, adding one new edge, to construct the shortest path to the current new vertex.

When first starting to analyze Dijkstra's algorithm I created a simple example to test my code on. The graph had only five nodes and six edges. The example I created was simple just to understand the concepts of the algorithm.

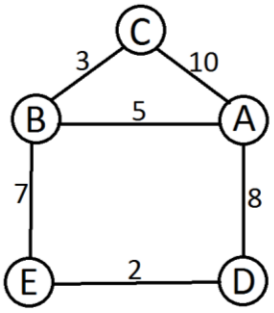
Using this graph, starting from node A, Dikstra's algorithm can be used to find the shortest path to all other nodes in the graph. To do this it helps to keep track of the distances in a table.

Node	Distance from A	Visited
B	∞	No
C	∞	No
D	∞	No
E	∞	No



All initial distances from A are set to infinity to ensure that any path found will be shorter than the initial value in the table. Starting from node A, examine all possible paths from node A. There is a path to node B with a weight of five. There's a path to node C with a weight of 10. Finally, there is a path to node D with a weight of 8. Now the table can be updated because these new values are less than infinity.

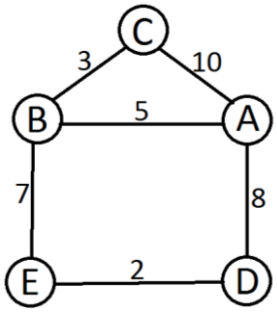
Node	Distance from A	Visited
B	5	No
C	10	No
D	8	No
E	∞	No



Now move to the nearest node from A. In this case node B is the closet. At node B there are paths available to node C and node E. the weight of the path from B to C is three. But to get the distance of C from A the weight of the path from A to B, which is five, must be added. So the new of C from A is now eight, which is shorter than the direct route from A to C. The same can be done for node E, which now has a distance from A of thirteen, which is better than infinity. Although the new route to C is not a direct route, but it is the shortest route. An advantage of Dijkstra's algorithm is that it can be used to find the shortest route, even when that route is not the most direct path.

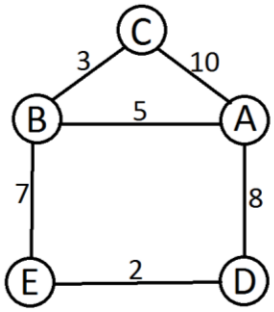
Now all paths from B have been checked. So, node B can now be marked as visited. Node C can also be marked as visited because it is the closet node to B but node C only has a path to nodes A and B which have already been visited. It's important to note that nodes are only visited once. Once a node has been marked visited, the path to that node is known to be the shortest route from the initial node.

Node	Distance from A	Visited
B	5	Yes
C	8	Yes
D	8	No
E	13	No



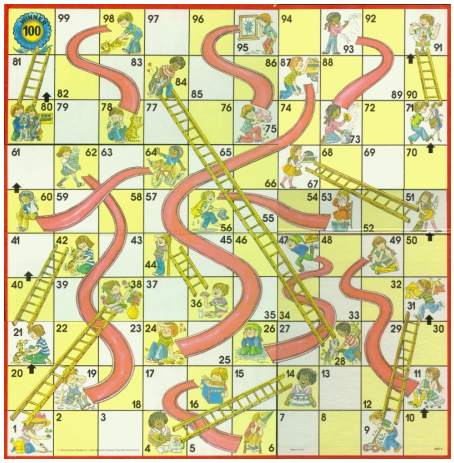
Now consider all the paths from the second closest node to A. In this case node D is the next node. There is only one path from node D to node E with a weight of two. Adding the weight of the path from node A to D, which is eight, to this new found path gives a distance of ten. This new distance from node A to E is shorter than the path previously discovered.

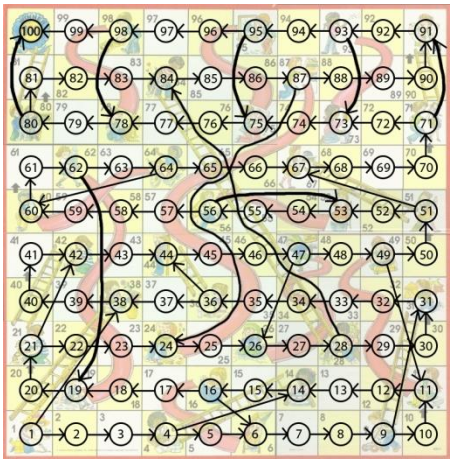
Node	Distance from A	Visited
B	5	Yes
C	8	Yes
D	8	Yes
E	10	Yes



Now all the nodes in the graph have been visited and now the table displays the correct shortest paths to all nodes from node A.

I first created this exmalpe to study the algorithm with a simple problem. I was able to create a program in java to solve the shortest path problem on this example using Dijktra's algorithm. Once the program was working I then modified it to apply it to a different problem. I chose to base my project on the childrens board game, Chutes and Ladders. The objective of the game is to roll a die and be the first player to the one hundreth square. Along the way the player encounters chutes and ladders that can transport them to other squares. Ladders help the player reach the end much faster, but chutes force the player to move back spaces. Looking at the gameboard, it resembles a graph like structure. Each square has an edge to the next square. The chutes and ladders are also edges that connect squares. So I created a program, using Dijkstra's algorith to find the shortest path in order to win.





To start off, the node class defines what properties the vertices of the graph will have. Each node contains a data item, in this case the square number. Each node also has an adjacency list, implemented as an array. Finally, by default, each node's distance is set to infinity.

```
class Node implements Comparable<Node>
{
    public final String data;
    public Edge[] adjacencies;
    public double minDistance = Double.POSITIVE_INFINITY;
    public Node previous;

    // constructor for node
    public Node(String data)
    {
        // data of the node
        this.data = data;
    }

    // return distance of edge between nodes
    public int compareTo(Node other)
    {
        return Double.compare(minDistance, other.minDistance);
    }

    // return string representation of data
    public String toString()
    {
        return data;
    }
}
```

Next is the edge class, which defines the property of the edges of the graph. Each edge has a target node that will be connected to the current node. Also each edge is given a weight to distinguish the distance between nodes. In the case of this application, the weight will always be one because by the rules of the game the player may only move one space at a time. Thus making game squares only a distance of one apart.

```
class Edge
{
    public final Node target;
    public final double weight;

    // constructor for edge
    public Edge(Node target, double weight)
    {
        // target node
        this.target = target;
        // weight of an edge
        this.weight = weight;
    }
}
```

Then once the nodes and edges are defined they can then be created. In the method buildGameboard(), a node is created for each square on the Chutes and Ladders board game, and stored in an array. The data item associated with each node is the square number it represents on the board. There are one-hundred spaces, so one-hundred nodes must be created.

```
public static Node[] buildGameboard()
{
    // create a node for each square on the board game, chutes & ladders
    Node n1 = new Node("1"); //start if 1st roll is 1
    Node n2 = new Node("2"); //start if 1st roll is 2
    Node n3 = new Node("3"); //start if 1st roll is 3
    Node n4 = new Node("4"); //start if 1st roll is 4
    Node n5 = new Node("5"); //start if 1st roll is 5
    Node n6 = new Node("6"); //start if 1st roll is 6
    Node n7 = new Node("7");
    Node n8 = new Node("8");
    Node n9 = new Node("9");
    Node n10 = new Node("10");
    Node n11 = new Node("11");
```

After all the vertices have been created, representing each square in the board game, the edges that connect each vertex must be created. This is all done within the same method to fully represent Chutes and ladders as a graph. The formula, $E < V \log V$ defines a sparse graph. There are a total of one hundred nodes and one hundred eighteen edges. Thus $118 < (100 \log 100 = 200)$. Therefore the game of Chutes and Ladders is a sparse graph and edges are better represented by an adjacency list. The game board is overall linear because the player is supposed to travel from one square to the next in ascending order. Having an adjacency matrix would take up too much space. Although there are a lot of nodes, this is not a very dense graph. At most

nodes have only two edges, in the case that there is a chute or a ladder present on that square. In the game the player can only travel up ladders and travel down chutes. So, the edges are created to mimic these properties, therefore creating a directed graph.

```
// create edges that connect each node to other nodes
// agencency lists for each node
n1.adjacencies = new Edge[]{new Edge(n2,1),new Edge(n38,1)};//ladder 1 to 38
n2.adjacencies = new Edge[]{new Edge(n3,1)};
n3.adjacencies = new Edge[]{new Edge(n4,1)};
n4.adjacencies = new Edge[]{new Edge(n5,1),new Edge(n14,1)};//ladder 4 to 14
n5.adjacencies = new Edge[]{new Edge(n6,1)};
n6.adjacencies = new Edge[]{new Edge(n7,1)};
n7.adjacencies = new Edge[]{new Edge(n8,1)};
n8.adjacencies = new Edge[]{new Edge(n9,1)};
n9.adjacencies = new Edge[]{new Edge(n10,1),new Edge(n31,1)};//ladder 9 to 31
n10.adjacencies = new Edge[]{new Edge(n11,1)};
```

```
// array implementation of the vertices of the graph just created
Node[] vertices =
{n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,
 n11,n12,n13,n14,n15,n16,n17,n18,n19,n20,
 n21,n22,n23,n24,n25,n26,n27,n28,n29,n30,
 n31,n32,n33,n34,n35,n36,n37,n38,n39,n40,
 n41,n42,n43,n44,n45,n46,n47,n48,n49,n50,
 n51,n52,n53,n54,n55,n56,n57,n58,n59,n60,
 n61,n62,n63,n64,n65,n66,n67,n68,n69,n70,
 n71,n72,n73,n74,n75,n76,n77,n78,n79,n80,
 n81,n82,n83,n84,n85,n86,n87,n88,n89,n90,
 n91,n92,n93,n94,n95,n96,n97,n98,n99,n100};

 return vertices;
}
```

Next, Dijkstra's algorithm can be implemented to find the shortest path to win the game. The method first sets the starting node's distance to zero. Then a priority queue is used to keep track of the nodes that can be visited. These are sorted by the distance from the starting node. When the algorithm is started, only the source node will be in the queue. While the destination has not been reached, visit the closet node to the starting node. This will then become the next node in the queue. When each node is then visited, all of its neighboring nodes are added to the queue except any they may have already been visited. As nodes are visited, the sum of the distance traveled is assigned to the current node's minimum distance from the starting node. This

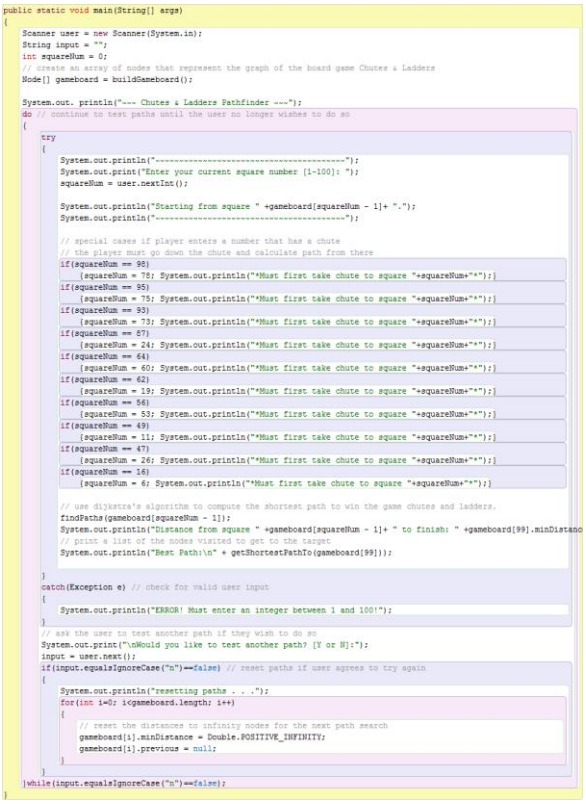
distance will later be displayed as the smallest number of spaces the player must travel in order to win the game. This process is then repeated for all paths.

```
// dijkstra's algorithm to calculate shortest path
public static void findPaths(Node source)
{
    source.minDistance = 0;
    PriorityQueue<Node> vertexQueue = new PriorityQueue<Node>();
    vertexQueue.add(source);

    while (!vertexQueue.isEmpty())
    {
        Node u = vertexQueue.poll();
        // visit each edge exiting u
        for (Edge e : u.adjacencies)
        {
            Node v = e.target;
            double weight = e.weight;
            double distanceThroughU = u.minDistance + weight;
            if (distanceThroughU < v.minDistance)
            {
                vertexQueue.remove(v);
                v.minDistance = distanceThroughU;
                v.previous = u;
                vertexQueue.add(v);
            }
        }
    }
}
```

After Dijkstra's algorithm has done its work to compute the shortest path to the destination, the path can be displayed. This next method returns a list of the nodes or in this case squares on the board game that should be traveled on to win the game in the shortest path. The loop works its way backwards following the nodes previous to the target node, all the way to the starting node. The list is then returned in reverse so that it is read in the proper order from source to destination.

```
public static List<Node> getShortestPathTo(Node target)
{
    List<Node> path = new ArrayList<Node>();

    for (Node vertex = target; vertex != null; vertex = vertex.previous)
        path.add(vertex);

    Collections.reverse(path);
    return path;
}
```

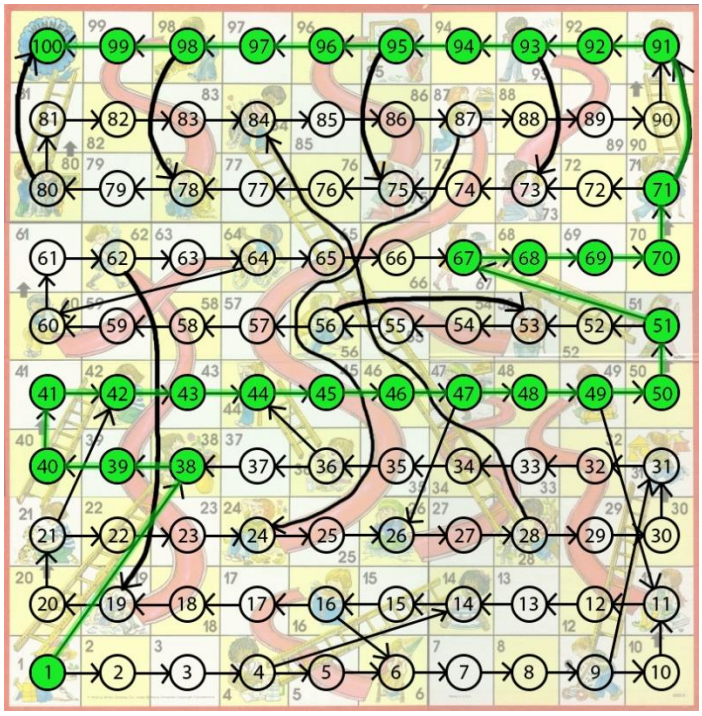
Finally, when all these pieces come together in the main method, the application to find the shortest path to win Chutes and ladders is complete. The user is asked to enter the number for a square on the board game that will be the starting point. The one-hundredth square is hard coded to always be the destination because it is the space the player is trying to reach to win the game. By allowing the user to enter square number, the application can be used to calculate the shortest path to win from anywhere on the board. The player may go off course of the best path during the game because the number of spaces a player is allowed to move each turn is dictated by the random roll of a die. So the user can enter their current square number after each turn in the game and then have an idea of what path to hope to take. For debugging purposes, if the user enters a square number that happens to be the top of a chute, the square number is changed to calculate the shortest path starting from the square at the bottom. The player is also reminded that they must first travel down the chute to continue, otherwise the player is cheating. This had to be added because from the testing I conducted, taking a chute was never an efficient route. The algorithm always seemed to show ladders. Perhaps there is a square number where taking a chute is efficient but this is probably a rare case. The user is asked if they would like to test another path after each test, so multiple routes may be tested in a single run of the application.



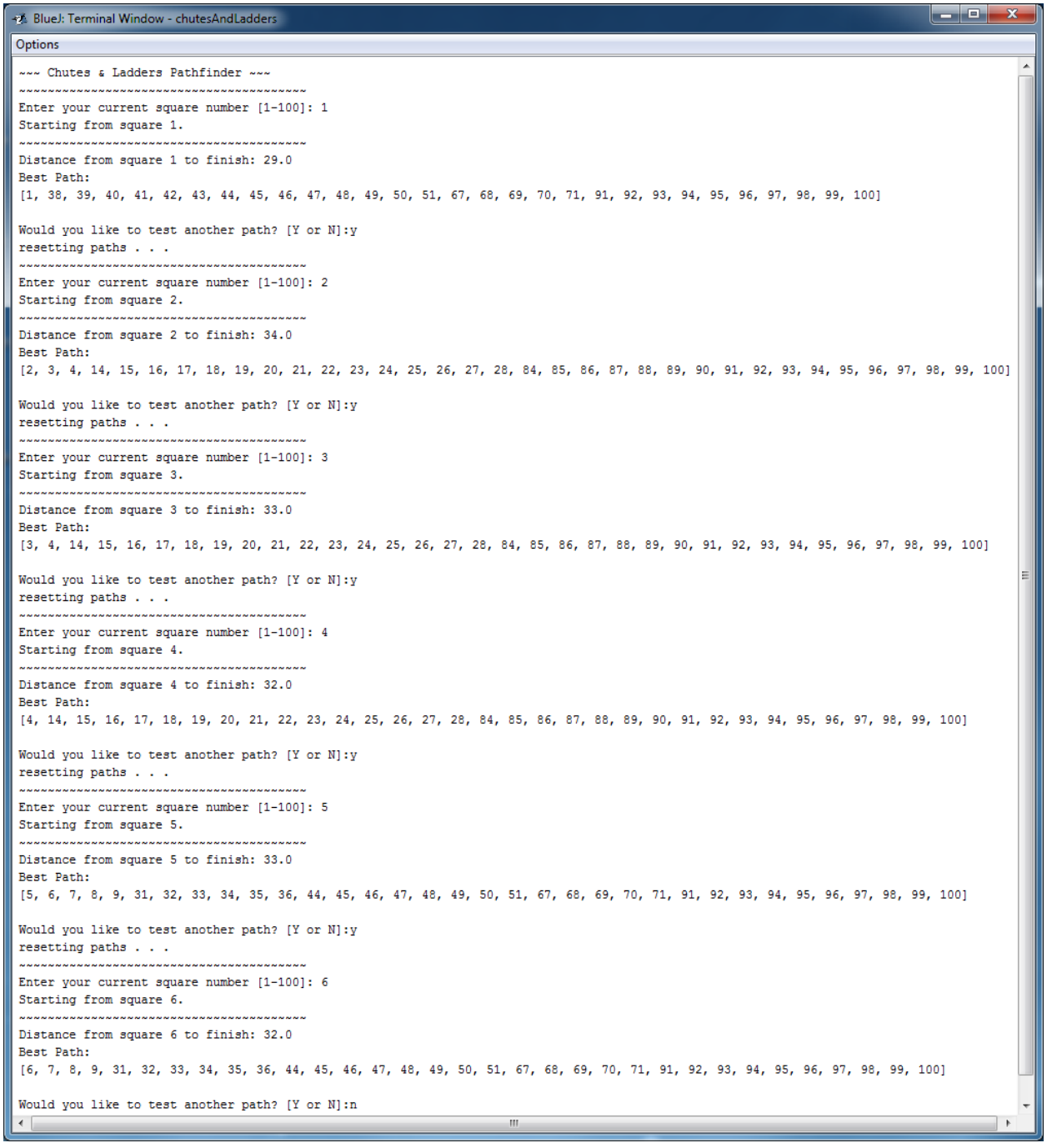
```
public static void main(String[] args)
{
    Scanner user = new Scanner(System.in);
    String input = "";
    int squareNum = 0;
    // create an array of nodes that represent the graph of the board game Chutes & Ladders
    Node[] gameboard = buildGameboard();

    System.out. println("~~~ Chutes & Ladders Pathfinder ~~~");
    do // continue to test paths until the user no longer wishes to do so
    {
        try
        {
            System.out.println("----------------------------------------");
            System.out.print("Enter your current square number [1-100]: ");
            squareNum = user.nextInt();

            System.out.println("Starting from square " +gameboard[squareNum - 1]+ ".");
            System.out.println("----------------------------------------");

            // special cases if player enters a number that has a chute
            // the player must go down the chute and calculate path from there
            if(squareNum == 98)
                {squareNum = 78; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 95)
                {squareNum = 75; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 93)
                {squareNum = 73; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 87)
                {squareNum = 24; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 64)
                {squareNum = 60; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 62)
                {squareNum = 19; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 56)
                {squareNum = 53; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 49)
                {squareNum = 11; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 47)
                {squareNum = 26; System.out.println("*Must first take chute to square "+squareNum+"*");}
            if(squareNum == 16)
                {squareNum = 6; System.out.println("*Must first take chute to square "+squareNum+"*");}

            // use dijkstra's algorithm to compute the shortest path to win the game chutes and ladders.
            findPaths(gameboard[squareNum - 1]);
            System.out.println("Distance from square " +gameboard[squareNum - 1]+ " to finish: " +gameboard[99].minDistanc
            // print a list of the nodes visited to get to the target
            System.out.println("Best Path:\n" + getShortestPathTo(gameboard[99]));

        }
        catch(Exception e) // check for valid user input
        {
            System.out.println("ERROR! Must enter an integer between 1 and 100!");
        }
        // ask the user to test another path if they wish to do so
        System.out.print("\nWould you like to test another path? [Y or N]:");
        input = user.next();
        if(input.equalsIgnoreCase("n")==false) // reset paths if user agrees to try again
        {
            System.out.println("resetting paths . . .");
            for(int i=0; i<gameboard.length; i++)
            {
                // reset the distances to infinity nodes for the next path search
                gameboard[i].minDistance = Double.POSITIVE_INFINITY;
                gameboard[i].previous = null;
            }
        }
    }while(input.equalsIgnoreCase("n")==false);
}
```

In Chutes and Ladders the players start off the board and must roll a die to determine where they begin on the board. From sample runs of the application, rolling a one will start the player off on the shortest path to win the game. So an example of the ideal scenario would be for a player to roll a one on their first turn. Since there's a ladder on square one to square thirty-eight, the player moves to square thirty-eight. The player will then want to roll a six on their second and third turns. Then roll a one on their fourth turn, thus landing on the ladder that is on square fifty-one. The ladder then leads the player to square sixty-seven. From there the player will want to roll a four on their fifth turn. This leads the player to the ladder on square seventy-one to square ninety-one. Then on the players sixth turn, the ideal roll would be a six. Now the player is on square ninety-seven and needs to roll a tree on their seventh turn to win. These are just example rolls; other combinations of numbers are possible. As long as the player doesn't

land on a chute, the game can be won in only seven turns, using the shortest path found by Dijkstra's algorithm.



Here is an example run of my application showing the shortest path to win the game, as well as all the starting possibilities.



Works cited:

Goodrich, Michael T., and Roberto Tamassia. "Graph Algorithms." *Data Structures and Algorithms in Java*. New York: J. Wiley & Sons, 2010. 597-657. Print.

"Dijkstra's Algorithm." *Dijkstra's Algorithm*. N.p., n.d. Web. 10 Dec. 2014. <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/dijkstraAlgor.htm>.

"Dijkstra's Algorithm." *Wikipedia*. Wikimedia Foundation, 10 Dec. 2014. Web. 10 Dec. 2014. <http://en.wikipedia.org/wiki/Dijkstra's_algorithm>.

Groeneveld, Savannah. "Rules for Chutes & Ladders | EHow." *EHow*. Demand Media, 18 May 2010. Web. 10 Dec. 2014. <http://www.ehow.com/list_6115868_chutes-ladders-rules.html>.