CLIENT

index.html

```
<!doctype html>


<html>

<style>
    body {
    background-color: #D7D7D7;

    }

  .buttons {
          background-color: #65BFAD;
          -moz-border-radius:28px;
          -webkit-border-radius:28px;
  border-radius:28px;
          border:0px solid #18ab29;
          display:inline-block;
          cursor:pointer;
          color: #ffffff;
          font-family:arial;
          font-size:17px;
          padding:16px 31px;
          text-decoration:none;
          text-shadow:0px 1px 0px #2f6627;
  position: fixed;
  top: 50px;
  right: 680px;
}
  .buttons:hover {
          background-color:#66CCCC;
}

  canvas {
    position: fixed;
}

ul#ruleList
{
    display:none;
}
```

```
.actions:hover ul#ruleList
{
    display: block;
    margin-left: 150px;
    margin-right: auto;
    color: #000000;
    font-family:arial;
    font-size:15px;
    text-decoration:none;
    position: fixed;
    top: 90px;
}


    </style>



 <body>

    <!--button id="test">TEST</button-->
     <canvas id="edgeCanvas" width="1500" height="900"></canvas>


  <div class="actions">
    <button id="rules" class="buttons">How To Play</button>
      <ul id="ruleList">
        <p> ON EACH TURN, you can make one of two moves: add a new node or add a new edge between two existing and unconnected nodes. </p>
        <p> TO ADD a new node, click the board to create a new node and click another node to connect them. </p>
        <p> TO ADD a new edge, click two consecutive nodes. </p>
        <p> WHEN you connect any two nodes, the colors of their neighbors become inverted. Nodes can have at most 3 neighbors. </p>
        <p> Your color is green. The computer's color is purple. TO WIN: have the most nodes of your color when the game ends. </p>

  </ul>
</div>

  </body>

  <script src="jquery.min.js"></script>
   <script src="nodes.js"></script>
```

```
</html>
```

nodes.js

```
window.onload = function() {
  //set global edge canvas width and height
edgeCanvas.width = window.innerWidth;
edgeCanvas.height = window.innerHeight;
edgeCanvas.onclick = function(e) {
  if (!edgeDrawn && !isFirst) {
    window.alert("Illegal move: must connect new node to another")
    }
    else {
  var x = e.clientX;
  var y = e.clientY;
  doesIntersect(x, y, function(intersect) {
      if (!intersect) {
        var c = new Circle(x, y);
  }
    else window.alert("Illegal move: nodes cannot intersect");
  });

  }
};


} //end onload

function tryXY() {
  randomX = Math.random()*innerWidth*0.8;
  randomY = Math.random()*innerHeight*0.8;
  if (randomX < innerWidth/2) randomX += 100;
  else randomX -= 100;
  if (randomY < innerHeight/2) randomY += 100;
  else randomY -= 100;
  var c;
 doesIntersect(randomX, randomY, function(intersect) {
          if (!intersect) {
            c = drawResponse(randomX, randomY);
          }
          else c = tryXY();
  });
 return c;
}
```

```
function drawResponse(x,y) {
var c = new Circle(x,y);
return c;
}

 function createRequestString() {
    /*
    -To pass to server:
  -for each node that exists
    -its color
    -the indices of its neighbors
    -the nodecap
    */
    var requestString = "";
    $.each(allNodes, function(index, value) {
      //requestString += value.color + toString(value.neighbors)+ ",";
      requestString += value.color + getIndices(value)+ ",";
    });
    requestString+=nodeCap;

  return requestString;
  }

 function toString(arr) {
    var str = "";
    $.each(arr, function(index,value) {
      str = str+","+value.color;
    });
    return str;

  }

  function getIndices(node) {
    //get indices of neighbors
    var str = "";
    $.each(node.neighbors, function(index, value) {
      $.each(allNodes, function(i, v) {
        if (value == v) str += " " + i;
      });
    });
    return str;
  }

function doesIntersect(x, y, callback) {
var intersect = false;
if (allNodes.length == 0) callback(false);
```

```
else {
$.each(allNodes, function(index, v) {

  //call the node in the loop x
  //if node.top is in x.top + or - nodeRadius*2 and node.left x.left is in x.left + or -
nodeRadius*2, return true

  if (y > (v.windowY - nodeRadius*2) && y < (v.windowY+nodeRadius*2) && x >
(v.windowX - nodeRadius*2) && x < (v.windowX + nodeRadius*2)) {
    //window.alert("interseeeect");
    intersect = true;
  }
  if (index == allNodes.length-1) callback(intersect);

});
}

}

nodeRadius = 50;
edgeCanvas = document.getElementById("edgeCanvas");
newEdge = [];
allNodes = [];
nodeCap = 13; //HOW MANY NODES BEFORE THE GAME ENDS
lastColor = "purple";
isFirst = true;
edgeDrawn = false;

function findWinner() {
var p = 0;
var g = 0;
$.each(allNodes, function(index, node) {
  if (node.color == "green") g++;
  else p++;
});

if (p > g) return "Purple";
else return "Green";

}


function Circle(x, y) {
  edgeDrawn = false;
  if (allNodes.length > 0) isFirst = false;
  var that = this;
  var el = document.createElement("canvas");
  document.body.appendChild(el);
  var height = nodeRadius*2;
  el.width = nodeRadius*2;
  el.height = nodeRadius*2;
  el.style.top = (y-nodeRadius)+"px";
  el.style.left = (x-nodeRadius)+"px";
  this.element = el;

  this.element.onclick = function() {
    //HIGHLIGHTING:
  //that.strokeWidth = 2;
  //that.strokeColor = "#ffffff";
  //that.draw();

  //console.log("newEdge length: " + newEdge.length);
  if (that.neighbors.length >= 3) window.alert("Illegal move: nodes can have at most 3
neighbors");
  else if (newEdge.length == 0) newEdge.push(that);
  else if (newEdge.length == 1) {
    if ($.inArray(newEdge[0], that.neighbors) != -1) {
      window.alert("Illegal move: edges can only join unconnected nodes");
    }
    else if (newEdge[0] == that) window.alert("Illegal move: cannot connect a node to
itself");
    else {
    newEdge.push(that);
    drawEdge(newEdge[0], newEdge[1], function() {
      var a = newEdge[0];
      var b = newEdge[1];
      invertNeighbors(a,b);
      a.neighbors.push(b);
      b.neighbors.push(a);
      newEdge = [];
      lastColor = "green";
      setTimeout(getResponse(), 300000);
      });
    }
  }
  else window.alert("Illegal move: edges can only connect two nodes");
    //highlighting:

  }

  this.neighbors = [];
  this.x = nodeRadius;
  this.y = this.x;
  this.windowX = x;
  this.windowY = y;
  this.radius = nodeRadius;
  this.color = "green";
  if (lastColor == "green") {
    this.color = "purple";
    lastColor = "purple";
  }
  else {
    this.color = "green";
    lastColor = "green";
  }

  this.strokeWidth = 0;
  if (this.color == "green") this.strokeColor = "#65BFAD";
  else this.strokeColor = '#9999CC';
  newEdge = [this];
  allNodes.push(this);
  that.draw();
  if (isFirst) {
    setTimeout(getResponse(), 300000);
  }

}

function getResponse() {
    var requestString = createRequestString(allNodes);
    //console.log(requestString);
    var arr = [];
    $.getJSON("http://127.0.0.1:8888/"+"move.json"+"?"+requestString, function(data) {
      $.each(data, function(index, value) {
        arr.push(allNodes[parseInt(value)]);
      });
      if (arr.length > 1) {
      drawEdge(arr[0], arr[1], function() {
      a = arr[0];
      b = arr[1];
      invertNeighbors(arr[0],arr[1]);
      a.neighbors.push(b);
      b.neighbors.push(a);
      newEdge = [];
      });
      }

      else {
```

```
   var one = tryXY();
   var two = arr[0];
   drawEdge(one, two, function() {
    invertNeighbors(two);
    one.neighbors.push(two);
    two.neighbors.push(one);
    newEdge = [];
   });
    }
    lastColor = "purple";
  });

}

 Circle.prototype.draw = function() {
 var ctx = this.element.getContext("2d");
 if (this.color == "purple") ctx.fillStyle = '#9999CC';
 else ctx.fillStyle = "#65BFAD";
 //ctx.strokeWidth = this.strokeWidth;
 //ctx.strokeStyle = this.strokeColor;
 ctx.strokeStyle = ctx.fillStyle;
 ctx.lineWidth = this.strokeWidth;
 ctx.beginPath();
 ctx.arc(this.x,this.y,this.radius,0,2*Math.PI);
 ctx.fill();
 ctx.stroke();
 }

//drawEdge: need to take in a start and end nodes
//new Circle(350, 400);

function invertNeighbors(a,b) {
 $.each(a.neighbors, function(index, node) {
  if (node.color == "green") node.color = "purple";
  else node.color = "green";
  node.draw();
 });
if (b) {
$.each(b.neighbors, function(index, node) {
  if (node.color == "green") node.color = "purple";
  else node.color = "green";
  node.draw();
 });
}
}
```

```
function drawEdge(a,b, callback) {
var ctx = edgeCanvas.getContext("2d");
ctx.lineWidth = 15;
ctx.strokeStyle = "#ffffff"
ctx.beginPath();
ctx.moveTo(a.windowX, a.windowY);
ctx.lineTo(b.windowX,b.windowY);
ctx.stroke();
edgeDrawn = true;
if (allNodes.length >= nodeCap) {
  window.alert("GAME OVER: The winner is: " + findWinner() + "!");
  document.open();
}
callback();
}
```

SERVER

WebService.java

```
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.InetSocketAddress;
import javax.swing.JOptionPane;
import java.util.*;

public class WebService
{
        public static void main(String[] args)
        {
      // default port and delay
      int port = 8888;

                // parse command line arguments to override defaults
      if (args.length > 0)
      {
         try
         {
            port = Integer.parseInt(args[0]);

         }
```

```
                  catch (NumberFormatException ex)
   {
      System.err.println("USAGE: java YahtzeeService [port]");
      System.exit(1);
   }
          }

          // set up an HTTP server to listen on the selected port
          try
          {
                  InetSocketAddress addr = new InetSocketAddress(port);
                  HttpServer server = HttpServer.create(addr, 1);

                  server.createContext("/move.json", new MoveHandler());

                  server.start();
                  System.out.println("server started");
          }
          catch (IOException ex)
          {
                  ex.printStackTrace(System.err);
                  System.err.println("Could not start server");
          }
  }

  public static class MoveHandler implements HttpHandler {
           @Override
public void handle(HttpExchange ex) throws IOException
{

  //System.err.println(ex.getRequestURI());
   String q = ex.getRequestURI().getQuery();
  // System.out.println(q);
   String[] split = q.split(",");
   State s = new State(0);
   //add nodes to the state
   int nodeCap = Integer.parseInt(split[split.length-1]);
   for (int i = 0; i < split.length-1; i++) {
          String str = split[i];
          String[] sep = str.split(" ");
          String color = sep[0];
          int c;
          if (color.equals("green")) c = 0;
          else c = 1;
          s.state.add(new Node(c));
   }
```

```
        //add neighbors to each node
        for (int i = 0; i < split.length-1; i++) {
            String str = split[i];
            Node node = s.state.get(i);
            String[] sep = str.split(" ");
            if (sep.length > 1) {
                //node has neighbors
                for (int j = 1; j < sep.length; j++) {
                    int n = Integer.parseInt(sep[j]); //index of the neighbor
                    Node neighbor = s.state.get(n);
                    if (!node.hasNeighbor(neighbor)) {
                        node.initNeighbor(neighbor);
                    }

                }
            }

        }


        MCTree tree = new MCTree(nodeCap);
        LinkedList<Integer> result = tree.getResponse(s);
        //response section: "response" should instead get output from MCTree
        StringBuilder response = new StringBuilder("{");
        for (int i = 0; i < result.size(); i++) {
            response.append("\""+i+"\":"+"\""+result.get(i)+"\",");
        }
        response = response.deleteCharAt(response.length()-1);
        response.append("}");
                ex.getResponseHeaders().add("Access-Control-Allow-Origin", "*");
        byte[] responseBytes = response.toString().getBytes();
        ex.sendResponseHeaders(HttpURLConnection.HTTP_OK,
responseBytes.length);
        ex.getResponseBody().write(responseBytes);
        ex.close();
      }



       }


}
```

MCTree.java

```
import java.util.*;

public class MCTree {

public LinkedList<State> visited = new LinkedList<State>();
public LinkedList<State> allStates = new LinkedList<State>();
public int totalNodes; //this needs to come from webservice
public State start;
public int numPlayouts;
public double balance = 1e-6;
//NOTE: HUMAN = COLOR 0, COMPUTER = COLOR 1

/* while current node is not a leaf, use selection policy to select a child, then reiterate.
if current node is not a leaf, instantiate the children list.
Play out the game until a finish (totalMoves). Iterate through the visited nodes list and
update their values. */

public MCTree(int m) {
	totalNodes = m;
	numPlayouts = m*1000;
//note: start should be parsed from a string into a state in WebService and sent to getResponse
//web service should also send over max number of moves

}

public static void main(String[] args) {
//for testing

	MCTree tree = new MCTree(3);
	State s = new State(0);
	Node a = new Node(0);
	Node b = new Node(1);
	b.initNeighbor(a);
	a.initNeighbor(b);
	s.state.add(a);
	s.state.add(b);
	LinkedList<Integer> result = tree.getResponse(s);
	System.out.println("response: " + result.get(0));
```

```
}


public LinkedList<Integer> getResponse(State s) {
    allStates.add(s);
    //System.out.println(s.children.size());
    //s.printChilds();
    for (int i = 0; i < numPlayouts; i++) {
        visited = new LinkedList<State>();
        runPlayout(s);
        //s.printChilds();
    }
    State best = selectChild(s);

    //response should return the index of the node to attach an edge to, or the
indices of nodes to draw an edge between
    LinkedList<Integer> response = s.getResponse(best);
    //System.out.println(response.get(0));
    return response;

}


public void runPlayout(State n) {
    n.numVisits++;
    visited.add(n);
    if (n.isEnd(totalNodes)) BP(n); //should call backpropagate method here
    else if (n.isLeaf) {
        //n.print();
        expand(n);
        //runPlayout(n.children.get(1));
        runPlayout(selectChild(n));
    }

    else runPlayout(selectChild(n));
}

public void BP(State s) {
    //backpropagation
    //check if won or lost
    //if won, add 1 to the value of all nodes in visited

    int zeroCount = 0;
    int oneCount = 0;
    for (Node n: s.state) {
        if (n.color == 0) zeroCount++;
```

```
                else oneCount++;
        }
        if (oneCount > zeroCount) {
                //computer won
                for (State state: visited) {
                        state.value += 1;
                }
        }

}


 public State selectChild(State n) {
        //select the best child using UCT (if value does not exist, it will return a random
child)
        Random r = new Random();
        double max = -10.0;
        State bestChild = null;
        for (State c: n.children) {
                double UCB = 0.0;
                if (c.numVisits == 0) UCB = Math.sqrt((2*(Math.log(n.numVisits)))/
(n.numVisits*r.nextDouble()));
                else UCB = c.value + Math.sqrt((2*(Math.log(n.numVisits)))/(c.numVisits));
                if (UCB > max) {
                        max = UCB;
                        bestChild = c;
                }

        }
        //System.out.println(bestChild == n.children.get(0));
        return bestChild;

  }


/*
 public State selectChild(State n) {
        //select the best child using UCT (if value does not exist, it will return a random
child)
        Random r = new Random();
        double max = Double.MIN_VALUE;
        State bestChild = null;
        for (State c: n.children) {
                double UCB = c.value / (c.numVisits + balance) +
Math.sqrt(Math.log(n.numVisits) / (c.numVisits + balance)) + r.nextDouble()*balance;
                if (UCB > max) {
```

```
                max = UCB;
                bestChild = c;
            }
        }
        //System.out.println(bestChild == n.children.get(0));
        return bestChild;

  }
*/

public void expand(State n) {
//for each node: if there are less than three neighbors, then try adding a purple and a
green: each of these make a new state
            //then, try to make an edge with each other node that has less than three
neighbors. Each of these is a new state.
            //Remember, whenever you deal with a new state, first check the tree's
master list of States to see if that state already exist.
            //If not: create a new State, but remember to save it's state by the proper
sorting rules.
for (int i = 0; i < n.state.size(); i++) {
        Node node = n.state.get(i);
        if (node.numNeighbors < 3) {
            //add neighbors of different colors
            int c;
            if (n.color == 0) c = 1;
            else c = 0;
            State child1 = new State(c);
            child1.copy(n);
            Node neighbor1 = new Node(c);
            child1.state.get(i).addNeighbor(neighbor1);
            neighbor1.addNeighbor(child1.state.get(i));
            child1.state.add(neighbor1);
            child1.sort();
            child1 = check(child1); //checks if the state exists already, if so, returns
that state
            n.addChild(child1);

//now, generate the states resulting from drawing edges from this node to all other free
nodes
            State child3;
            for (int j = 0; j < n.state.size(); j++) {
                Node that = n.state.get(j);
                if ((j != i) && (that.numNeighbors < 3) && !
(node.hasNeighbor(that))) {
                    child3 = new State(c);
                    child3.copy(n);
                Node one = child3.state.get(i);
                Node two = child3.state.get(j);
                one.addNeighbor(two);
                two.addNeighbor(one);
                child3.sort();
                child3 = check(child3);
                n.addChild(child3);
            }
        }

    }

}


n.isLeaf = false;
}


public State check(State s) {
    //check if state already exists
    State result = s;
    for (State state: allStates) {
        if (state.equals(s)) result = state;
    }

    if (result == s) allStates.add(s); //if its not already existent, add the new state to
all states
    return result;
}

public void print() {
    for (State s: allStates) {
        s.print();
    }
}

}
```

Node.java

```
import java.util.*;

public class Node {
```

```
int color; //0: green 1: purple
int neighborColorSum = 0;
LinkedList<Node> neighbors = new LinkedList<Node>();
int numNeighbors = 0;

public Node(int c) {
        color = c;
}

public void initNeighbor(Node n) {
        this.neighbors.add(n);
        numNeighbors++;
        neighborColorSum+=n.color;
}

public void addNeighbor(Node n) {
        invertNeighbors();
        this.neighbors.add(n);
        numNeighbors++;
        neighborColorSum+=n.color;
}

public void setNeighbor(Node n) {
        this.neighbors.add(n);
}

public void invertNeighbors() {
        for (Node n: neighbors) {
                n.invert();
        }

}

public void print() {
        System.out.println("node: " + color);
        System.out.print("neighbors: ");
        for (Node n: neighbors) {
                System.out.print(n.color + " ");
        }
        System.out.println();
        //System.out.println("neighborSum: " + neighborColorSum);

}

public void invert() {
        if (color == 0) color = 1;
        else color = 0;
    }

    public void copy(Node n) {
        this.color = n.color;
        this.numNeighbors = n.numNeighbors;
        this.neighborColorSum = n.neighborColorSum;
        /*
        for (int i = 0; i < n.neighbors.size(); i++) {
            Node a = n.neighbors.get(i);
            Node b = new Node(a.color);
            b.copy(a);
            neighbors.add(b);
        }
        */

    }

    public boolean hasNeighbor(Node n) {
        for (Node a: neighbors) {
            if (n == a) return true;
        }
        return false;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Node)) return false;

        else {
        Node n = (Node) o;
        if (n.color != this.color) return false;
        if (n.neighbors.size() != this.neighbors.size()) return false;
        if (n.numNeighbors != this.numNeighbors) return false;
        /*
        for (int i = 0; i < this.neighbors.size(); i++) {
            if (!(this.neighbors.get(i).equals(n.neighbors.get(i)))) return false;
        }
        */
        //comparing neighborColorSum instead of the above loop, which will do a
lot of recursive calls
        if (n.neighborColorSum != this.neighborColorSum) return false;
    }
        return true;
}


}
```

State.java

```
import java.util.*;

public class State {

public LinkedList<State> children = new LinkedList<State>(); //these are the children game states.
public int numVisits = 0;
public double value = 0;
public boolean isLeaf = true;
public boolean isEnd = false;
public LinkedList<Node> state = new LinkedList<Node>();
public LinkedList<Node> oldState = new LinkedList<Node>();
public int color; //the player's color who just added to create this state

public State(int c) {
color = c;
}

public LinkedList<Integer> getResponse(State c) {
LinkedList<Integer> result = new LinkedList<Integer>();

for (int i = 0; i < state.size(); i++) {

    if (!(state.get(i).numNeighbors == c.oldState.get(i).numNeighbors)) {
        //there was a change to this node
        result.add(i);
    }
}

return result;

}


public void addChild(State s) {
    children.add(s);
}

public boolean isEnd(int m) {
    return (m == state.size());
}
```

```
public void copy(State s) {
        for (int i = 0; i < s.state.size(); i++) {
                Node n = s.state.get(i);
                Node add = new Node(n.color);
                add.copy(n);
                state.add(add);

        }
for (int j = 0; j < s.state.size(); j++) {
        Node a = s.state.get(j);
        for (int k = 0; k < s.state.size(); k++) {
                Node b = s.state.get(k);
                if (a.hasNeighbor(b)) {
                        Node c = this.state.get(j);
                        Node d = this.state.get(k);
                        if (!c.hasNeighbor(d)) {
                        c.setNeighbor(d);
                        d.setNeighbor(c);
                }
                }
        }
}
}

public void saveState() {
        //copying state into oldstate
        for (int i = 0; i < state.size(); i++) {
                Node n = state.get(i);
                Node add = new Node(n.color);
                add.copy(n);
                oldState.add(add);

        }
for (int j = 0; j < state.size(); j++) {
        Node a = state.get(j);
        for (int k = 0; k < state.size(); k++) {
                Node b = state.get(k);
                if (a.hasNeighbor(b)) {
                        Node c = oldState.get(j);
                        Node d = oldState.get(k);
                        if (!c.hasNeighbor(d)) {
                        c.setNeighbor(d);
                        d.setNeighbor(c);
                }
                }
        }
}
}

public boolean equals(Object o) {
if (!(o instanceof State)) return false;
else {
State s = (State) o;
if (s.state.size() != this.state.size()) return false;
for (int i = 0; i < s.state.size(); i++) {
    if (!(s.state.get(i).equals(this.state.get(i)))) return false;
}

}
return true;

}

public void print() {
    System.out.println("STATE");
        for (Node n : state) {
            n.print();
        }

        System.out.println();
}

public void printOldState() {
    System.out.println("OLDSTATE");
        for (Node n : oldState) {
            n.print();
        }

        System.out.println();
}
public void printChilds() {
    System.out.println("CHILD VALUES");
    for (State c: children) {
        System.out.print(c.value + " ");
    }
}

public void sort() {
    //sorting rules:
    //each node will have a color and a list of neighbors
    //every list will be sorted in the game state by 1 first and > # neighbors first and >
neighbor colors first
```

```
    saveState(); //save the old order so we can compare with parent if bestChild
    for (int i = 0; i < state.size(); i++) {
        for (int j = 0; j < state.size()-1; j++) {
            Node first = state.get(j);
            Node next = state.get(j+1);
            Node best = first;
            if (next.color > best.color) best = next;
            if (next.neighbors.size() > best.neighbors.size()) best = next;
            else if (next.neighbors.size() == best.neighbors.size()) {
                if (next.neighborColorSum > best.neighborColorSum) best =
next;
            }

            if (best == next) {
                //swap
                state.set(j,next);
                state.set(j+1,first);
            }
        }

    }
}

}
```