Michael Rallo
Kara Hryszko
Dustin Rios
CS 4750, Fall 2016

Homework 6

<u>Coding environment:</u> JavaFX

<u>Programs used:</u> NetBeans 8.1

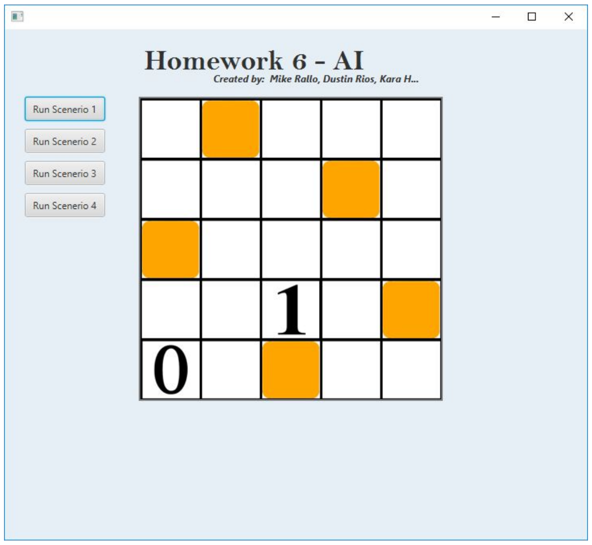
<u>Github Project Directory:</u> https://github.com/msr5zb/Homework6AI

Algorithms and Implementation

In the implementation of the Latin Square Puzzles problem, the search algorithms we used was BackTracking with Minimum Remaining Value (MRV), using degree heuristic and forward checking.

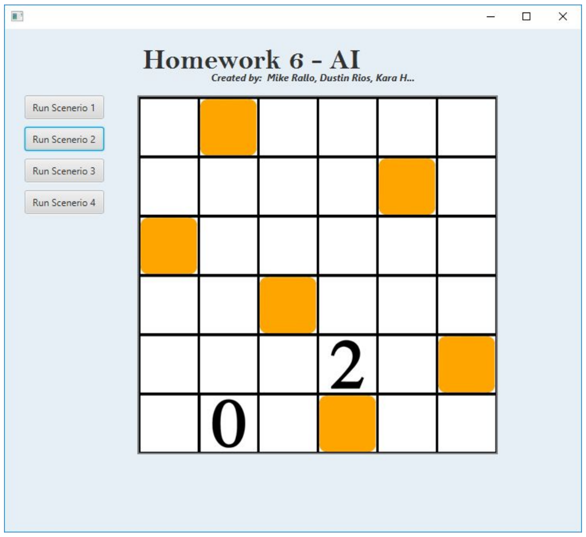
Here are the Statistics for Each Scenario:

Solution of Scenario #1 (on 5 x 5 board, with given scenario)



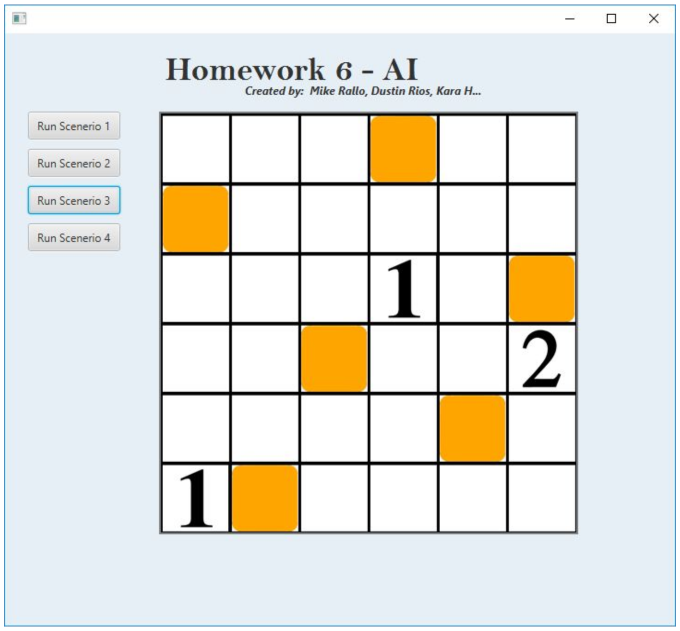
Run Information
<u>Number of search nodes:</u> 11
<u>CPU execution time (in milliseconds):</u> 5

Solution of Scenario #2 (on 6 x 6 board, with given scenario)



Run Information
<u>Number of search nodes:</u> 7
<u>CPU execution time (in milliseconds):</u> 4

Solution of Scenario #3 (6 x 6 board)

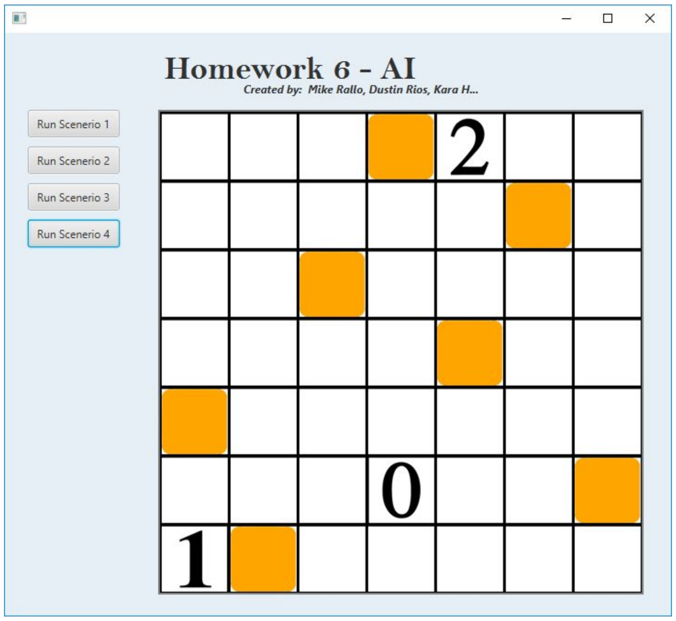


Run Information

<u>Number of search nodes:</u> 51

<u>CPU execution time (in milliseconds):</u> 120

Solution of Scenario #4 (on 6 x 6 board, with given scenario)



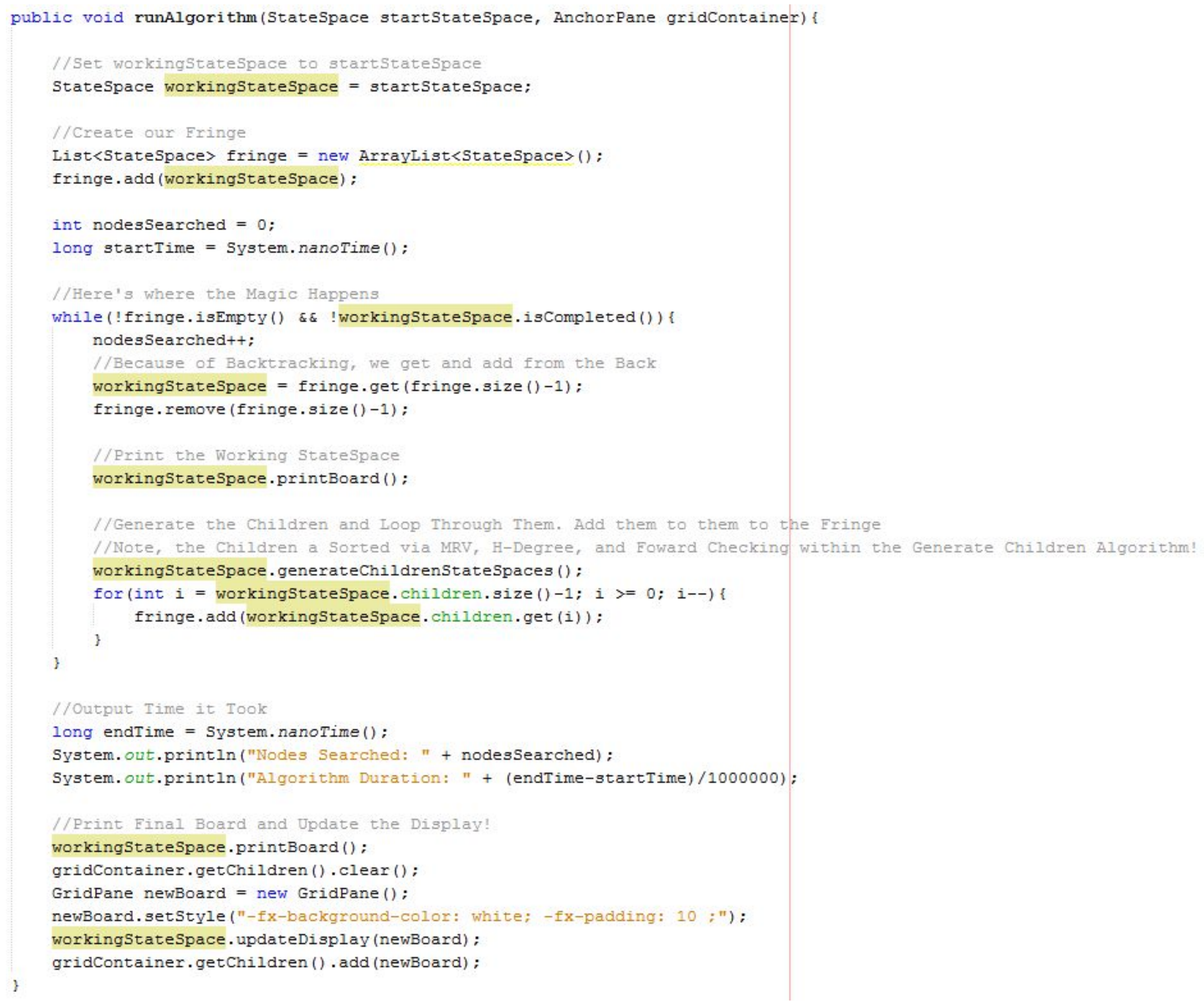
Run Information

<u>Number of search nodes:</u> 60

<u>CPU execution time (in milliseconds):</u> 204

The BackTracking Algorithm

First, let’s take a look at the main algorithm - Backtracking.

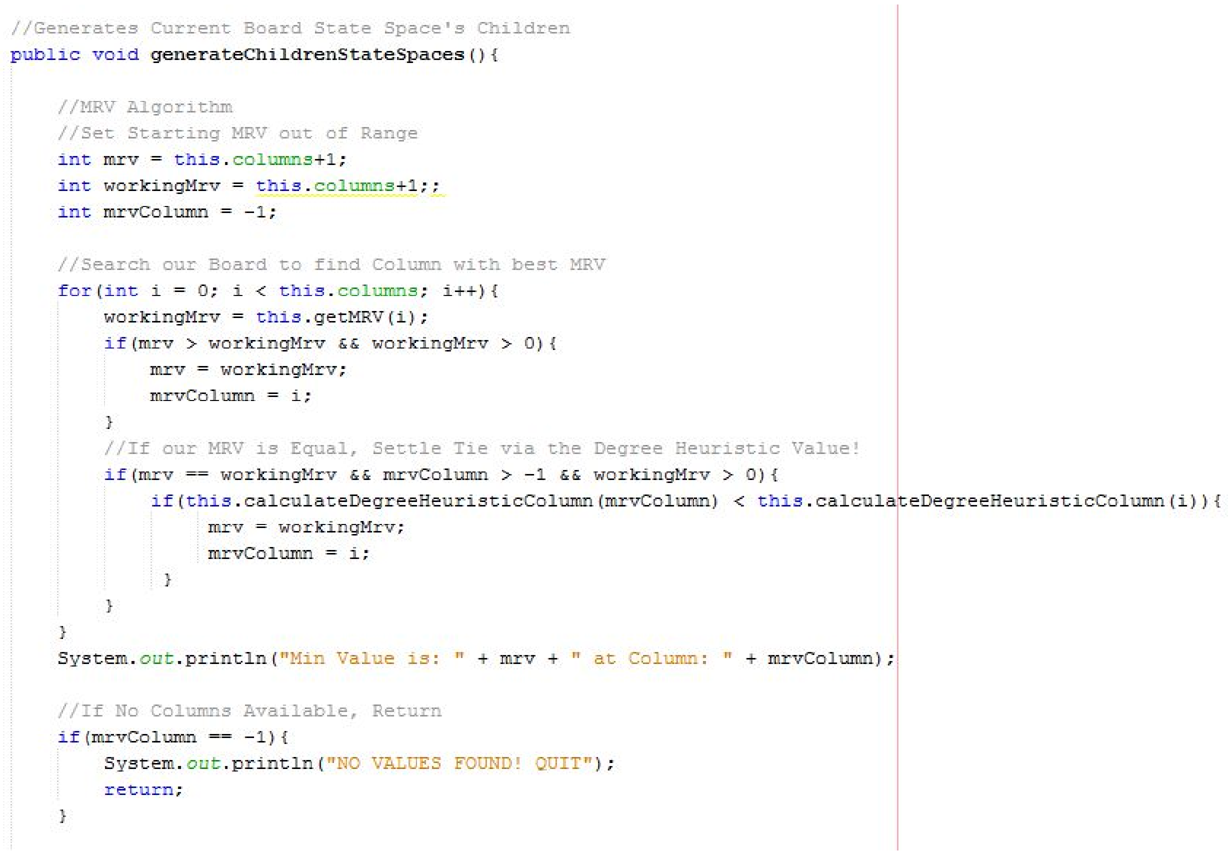


```java
public void runAlgorithm(StateSpace startStateSpace, AnchorPane gridContainer){

    //Set workingStateSpace to startStateSpace
    StateSpace workingStateSpace = startStateSpace;

    //Create our Fringe
    List<StateSpace> fringe = new ArrayList<StateSpace>();
    fringe.add(workingStateSpace);

    int nodesSearched = 0;
    long startTime = System.nanoTime();

    //Here's where the Magic Happens
    while(!fringe.isEmpty() && !workingStateSpace.isCompleted()){
        nodesSearched++;
        //Because of Backtracking, we get and add from the Back
        workingStateSpace = fringe.get(fringe.size()-1);
        fringe.remove(fringe.size()-1);

        //Print the Working StateSpace
        workingStateSpace.printBoard();

        //Generate the Children and Loop Through Them. Add them to them to the Fringe
        //Note, the Children a Sorted via MRV, H-Degree, and Foward Checking within the Generate Children Algorithm!
        workingStateSpace.generateChildrenStateSpaces();
        for(int i = workingStateSpace.children.size()-1; i >= 0; i--){
            fringe.add(workingStateSpace.children.get(i));
        }
    }

    //Output Time it Took
    long endTime = System.nanoTime();
    System.out.println("Nodes Searched: " + nodesSearched);
    System.out.println("Algorithm Duration: " + (endTime-startTime)/1000000);

    //Print Final Board and Update the Display!
    workingStateSpace.printBoard();
    gridContainer.getChildren().clear();
    GridPane newBoard = new GridPane();
    newBoard.setStyle("-fx-background-color: white; -fx-padding: 10 ;");
    workingStateSpace.updateDisplay(newBoard);
    gridContainer.getChildren().add(newBoard);
}
```

Note, the MRV, Degree Heuristic, and Forward Checking are All done in the Generate Children Function - to which they find the Column with best MRV, and returns it’s sorted Children, whilst doing forward checking.

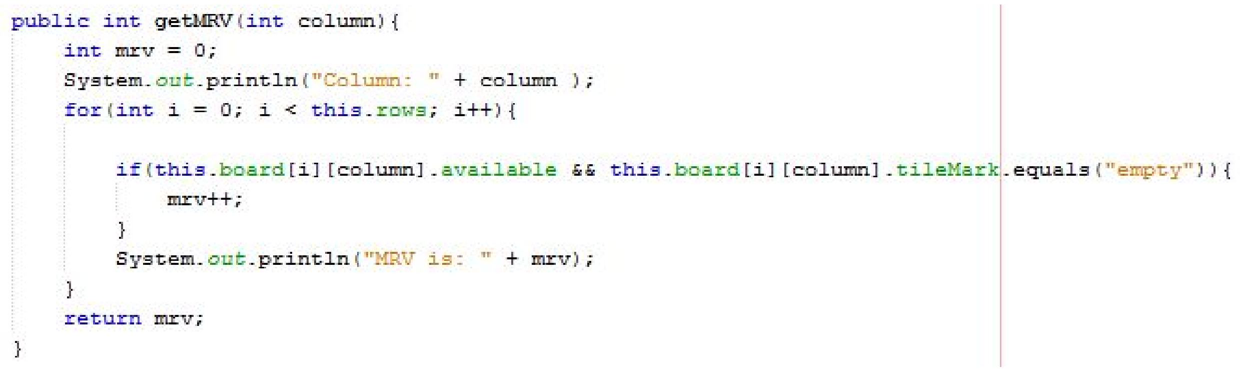
MRV - Within GenerateChildren

This Alg finds the Column with the best MRV. Note on Tie, we settle it via best Degree Heuristic Value!



```java
//Generates Current Board State Space's Children
public void generateChildrenStateSpaces(){

    //MRV Algorithm
    //Set Starting MRV out of Range
    int mrv = this.columns+1;
    int workingMrv = this.columns+1;;
    int mrvColumn = -1;

    //Search our Board to find Column with best MRV
    for(int i = 0; i < this.columns; i++){
        workingMrv = this.getMRV(i);
        if(mrv > workingMrv && workingMrv > 0){
            mrv = workingMrv;
            mrvColumn = i;
        }
        //If our MRV is Equal, Settle Tie via the Degree Heuristic Value!
        if(mrv == workingMrv && mrvColumn > -1 && workingMrv > 0){
            if(this.calculateDegreeHeuristicColumn(mrvColumn) < this.calculateDegreeHeuristicColumn(i)){
                 mrv = workingMrv;
                 mrvColumn = i;
             }
        }
    }
    System.out.println("Min Value is: " + mrv + " at Column: " + mrvColumn);

    //If No Columns Available, Return
    if(mrvColumn == -1){
        System.out.println("NO VALUES FOUND! QUIT");
        return;
    }
```

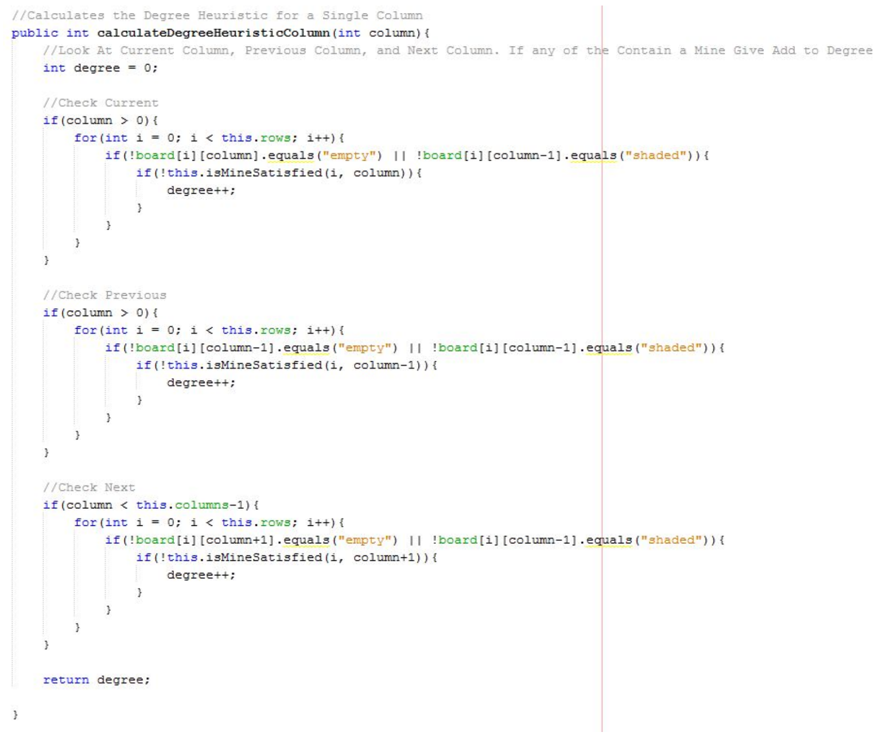
getMRV Helper Function:



```java
public int getMRV(int column){
    int mrv = 0;
    System.out.println("Column: " + column );
    for(int i = 0; i < this.rows; i++){

        if(this.board[i][column].available && this.board[i][column].tileMark.equals("empty")){
            mrv++;
        }
        System.out.println("MRV is: " + mrv);
    }
    return mrv;
}
```

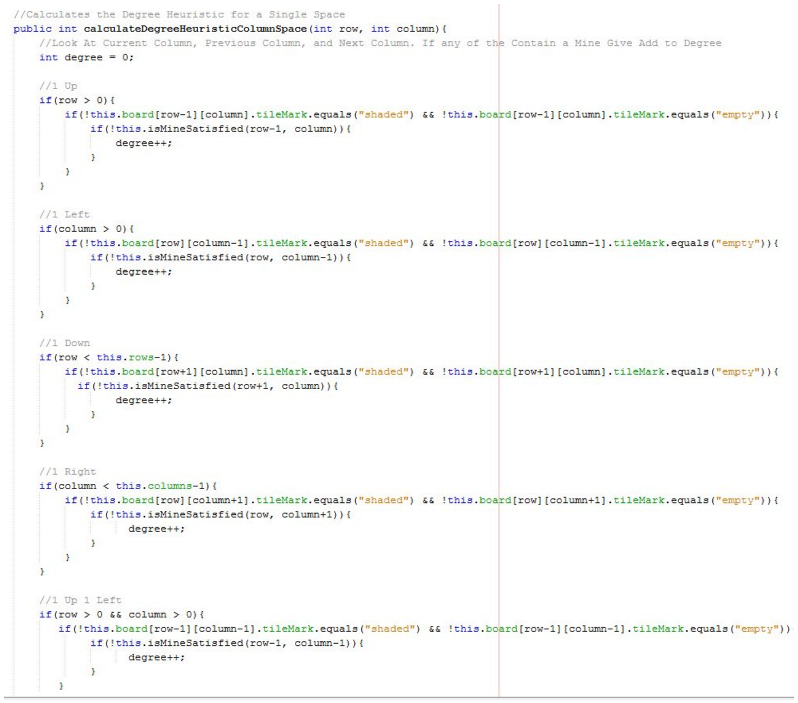
Degree Heuristic

We used Degree Heuristic Values not only for Columns, but also for Spaces. This Degree Heuristic is Calculated by searching how many Active Number Mines Are affected and can be affected by placing a particular value in a particular space. Below is a function for calculating the Degree Heuristic of Column.



```java
//Calculates the Degree Heuristic for a Single Column
public int calculateDegreeHeuristicColumn(int column){
    //Look At Current Column, Previous Column, and Next Column. If any of the Contain a Mine Give Add to Degree
    int degree = 0;

    //Check Current
    if(column > 0){
        for(int i = 0; i < this.rows; i++){
            if(!board[i][column].equals("empty") || !board[i][column-1].equals("shaded")){
                if(!this.isMineSatisfied(i, column)){
                    degree++;
                }
            }
        }
    }

    //Check Previous
    if(column > 0){
        for(int i = 0; i < this.rows; i++){
            if(!board[i][column-1].equals("empty") || !board[i][column-1].equals("shaded")){
                if(!this.isMineSatisfied(i, column-1)){
                    degree++;
                }
            }
        }
    }

    //Check Next
    if(column < this.columns-1){
        for(int i = 0; i < this.rows; i++){
            if(!board[i][column+1].equals("empty") || !board[i][column-1].equals("shaded")){
                if(!this.isMineSatisfied(i, column+1)){
                    degree++;
                }
            }
        }
    }

    return degree;

}
```

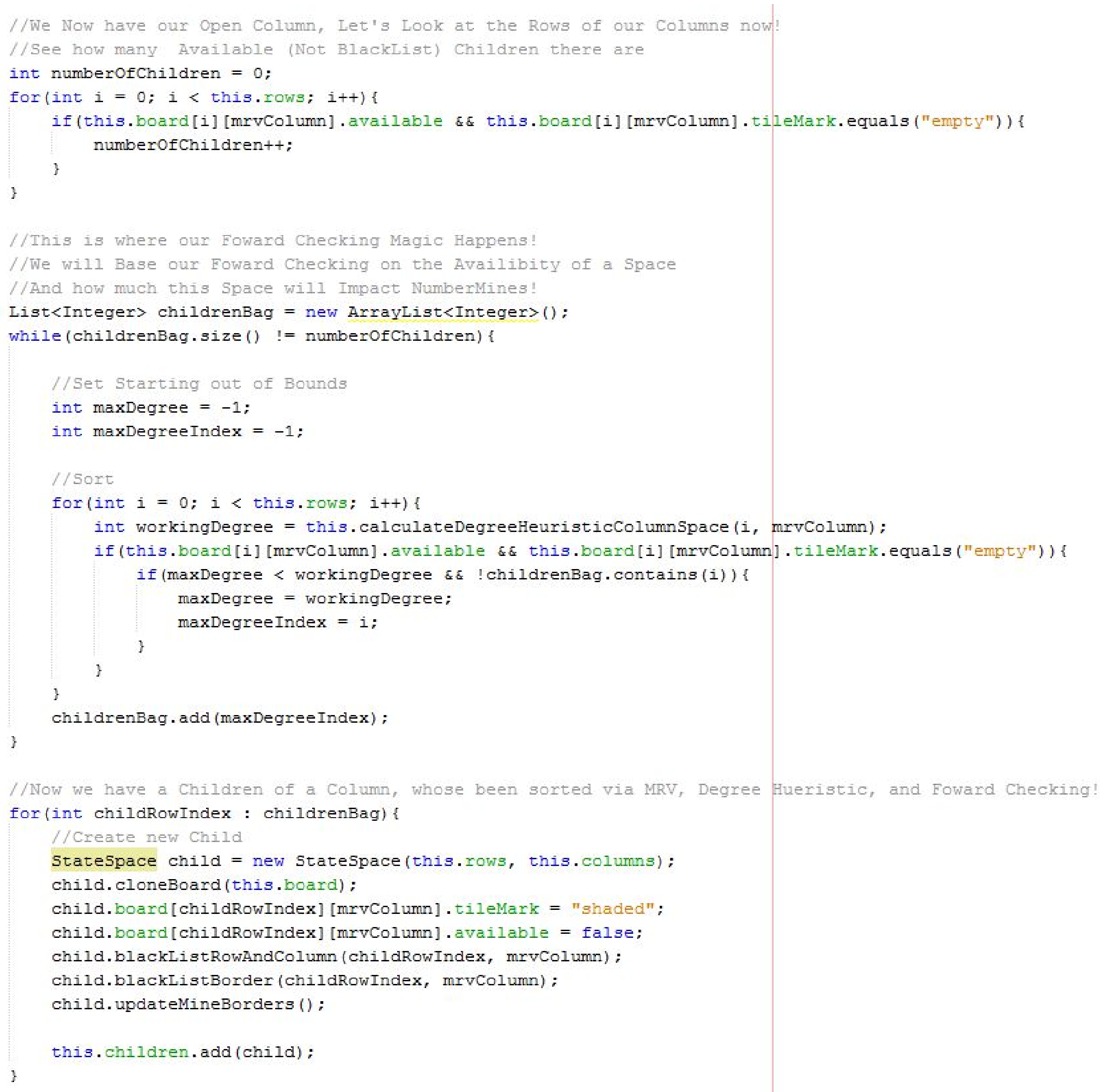
Below is for Calculating Degree Heuristic of Space (Cut Short).



```java
//Calculates the Degree Heuristic for a Single Space
public int calculateDegreeHeuristicColumnSpace(int row, int column){
    //Look At Current Column, Previous Column, and Next Column. If any of the Contain a Mine Give Add to Degree
    int degree = 0;

    //1 Up
    if(row > 0){
        if(!this.board[row-1][column].tileMark.equals("shaded") && !this.board[row-1][column].tileMark.equals("empty")){
            if(!this.isMineSatisfied(row-1, column)){
                degree++;
            }
        }
    }

    //1 Left
    if(column > 0){
        if(!this.board[row][column-1].tileMark.equals("shaded") && !this.board[row][column-1].tileMark.equals("empty")){
            if(!this.isMineSatisfied(row, column-1)){
                degree++;
            }
        }
    }

    //1 Down
    if(row < this.rows-1){
        if(!this.board[row+1][column].tileMark.equals("shaded") && !this.board[row+1][column].tileMark.equals("empty")){
          if(!this.isMineSatisfied(row+1, column)){
                degree++;
            }
        }
    }

    //1 Right
    if(column < this.columns-1){
        if(!this.board[row][column+1].tileMark.equals("shaded") && !this.board[row][column+1].tileMark.equals("empty")){
            if(!this.isMineSatisfied(row, column+1)){
                  degree++;
            }
        }
    }

    //1 Up 1 Left
    if(row > 0 && column > 0){
       if(!this.board[row-1][column-1].tileMark.equals("shaded") && !this.board[row-1][column-1].tileMark.equals("empty"))
            if(!this.isMineSatisfied(row-1, column-1)){
                  degree++;
            }
        }
```

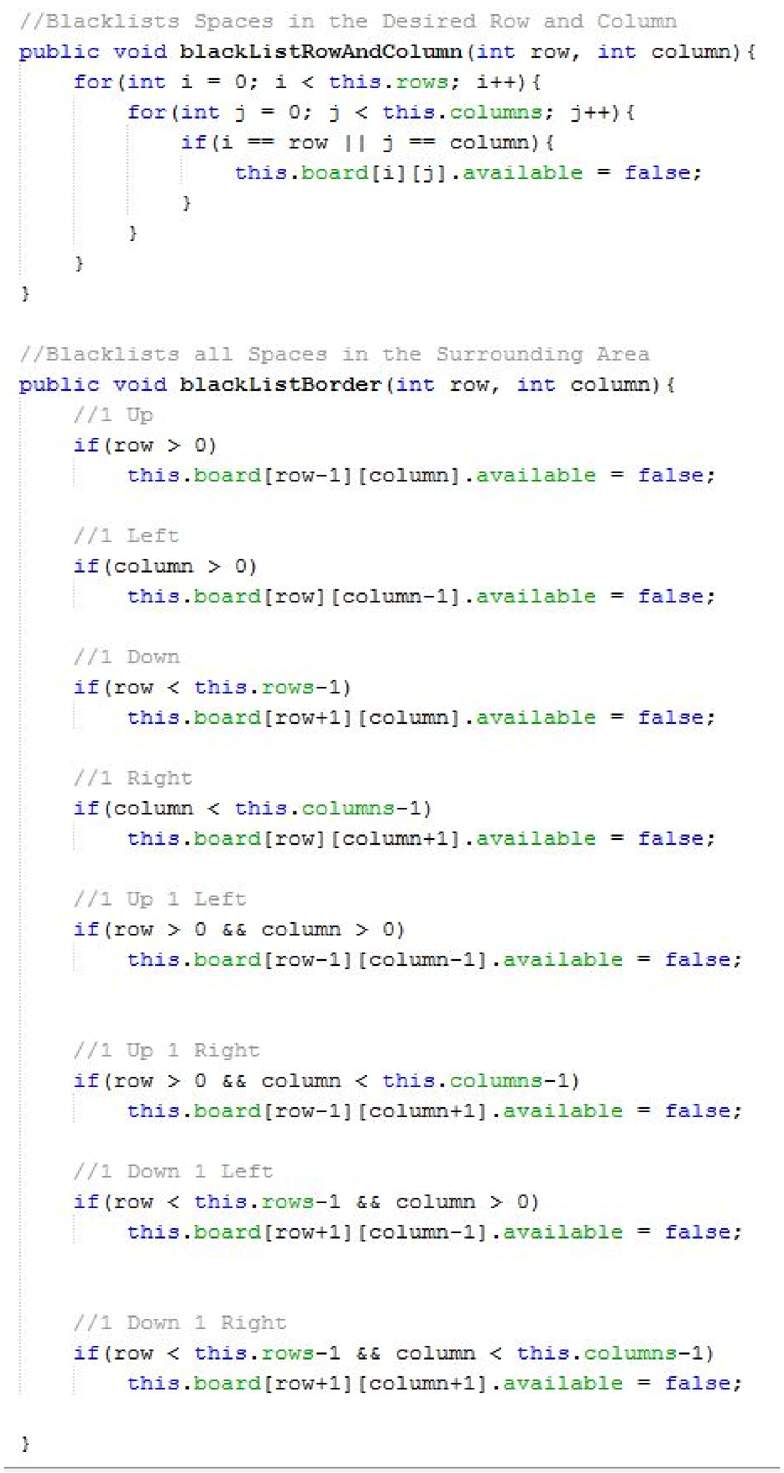
Forward Checking

With Forward Checking, we are able to determine which moves are available to use using lookaheads and comparable neighbor values. We sort our children using this, as well as previous functions to return the desired bag. Note, a lot of Checking for Availability is done with other functions, they will be liste below as well! Note availability goes hand ‘n’ hand with the constraints!



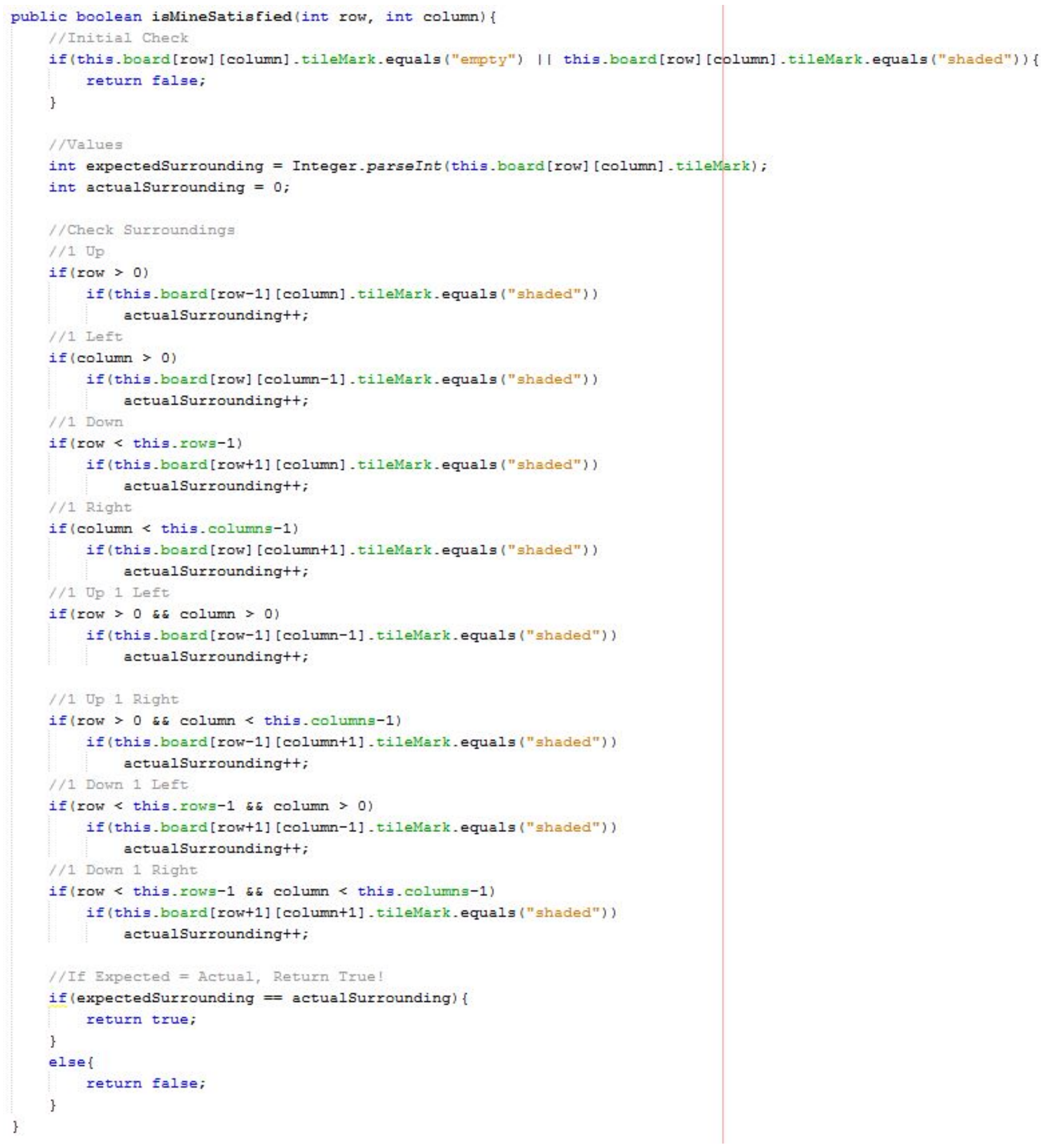
```java
//We Now have our Open Column, Let's Look at the Rows of our Columns now!
//See how many  Available (Not BlackList) Children there are
int numberOfChildren = 0;
for(int i = 0; i < this.rows; i++){
    if(this.board[i][mrvColumn].available && this.board[i][mrvColumn].tileMark.equals("empty")){
        numberOfChildren++;
    }
}

//This is where our Foward Checking Magic Happens!
//We will Base our Foward Checking on the Availibity of a Space
//And how much this Space will Impact NumberMines!
List<Integer> childrenBag = new ArrayList<Integer>();
while(childrenBag.size() != numberOfChildren){

    //Set Starting out of Bounds
    int maxDegree = -1;
    int maxDegreeIndex = -1;

    //Sort
    for(int i = 0; i < this.rows; i++){
        int workingDegree = this.calculateDegreeHeuristicColumnSpace(i, mrvColumn);
        if(this.board[i][mrvColumn].available && this.board[i][mrvColumn].tileMark.equals("empty")){
            if(maxDegree < workingDegree && !childrenBag.contains(i)){
                maxDegree = workingDegree;
                maxDegreeIndex = i;
            }
        }
    }
    childrenBag.add(maxDegreeIndex);
}

//Now we have a Children of a Column, whose been sorted via MRV, Degree Hueristic, and Foward Checking!
for(int childRowIndex : childrenBag){
    //Create new Child
    StateSpace child = new StateSpace(this.rows, this.columns);
    child.cloneBoard(this.board);
    child.board[childRowIndex][mrvColumn].tileMark = "shaded";
    child.board[childRowIndex][mrvColumn].available = false;
    child.blackListRowAndColumn(childRowIndex, mrvColumn);
    child.blackListBorder(childRowIndex, mrvColumn);
    child.updateMineBorders();

    this.children.add(child);
}
```

Function to Blacklist Spaces that can no longer be placed based off constraints!



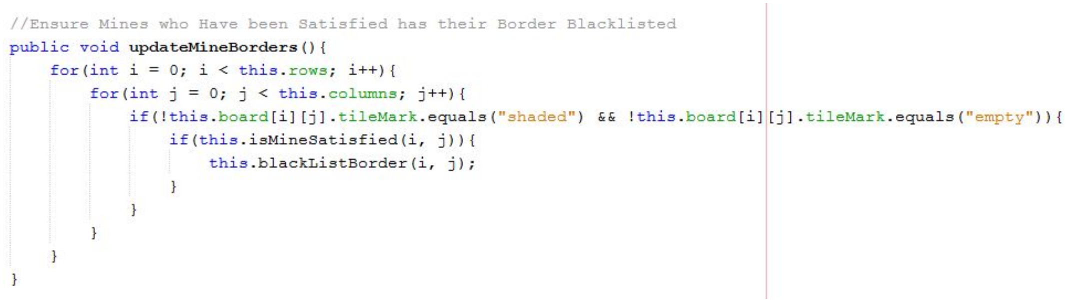
```java
//Blacklists Spaces in the Desired Row and Column
public void blackListRowAndColumn(int row, int column){
    for(int i = 0; i < this.rows; i++){
        for(int j = 0; j < this.columns; j++){
            if(i == row || j == column){
                this.board[i][j].available = false;
            }
        }
    }
}

//Blacklists all Spaces in the Surrounding Area
public void blackListBorder(int row, int column){
    //1 Up
    if(row > 0)
        this.board[row-1][column].available = false;

    //1 Left
    if(column > 0)
        this.board[row][column-1].available = false;

    //1 Down
    if(row < this.rows-1)
        this.board[row+1][column].available = false;

    //1 Right
    if(column < this.columns-1)
        this.board[row][column+1].available = false;

    //1 Up 1 Left
    if(row > 0 && column > 0)
        this.board[row-1][column-1].available = false;


    //1 Up 1 Right
    if(row > 0 && column < this.columns-1)
        this.board[row-1][column+1].available = false;

    //1 Down 1 Left
    if(row < this.rows-1 && column > 0)
        this.board[row+1][column-1].available = false;


    //1 Down 1 Right
    if(row < this.rows-1 && column < this.columns-1)
        this.board[row+1][column+1].available = false;

}
```

Function to check if a Mine has been Satisfied



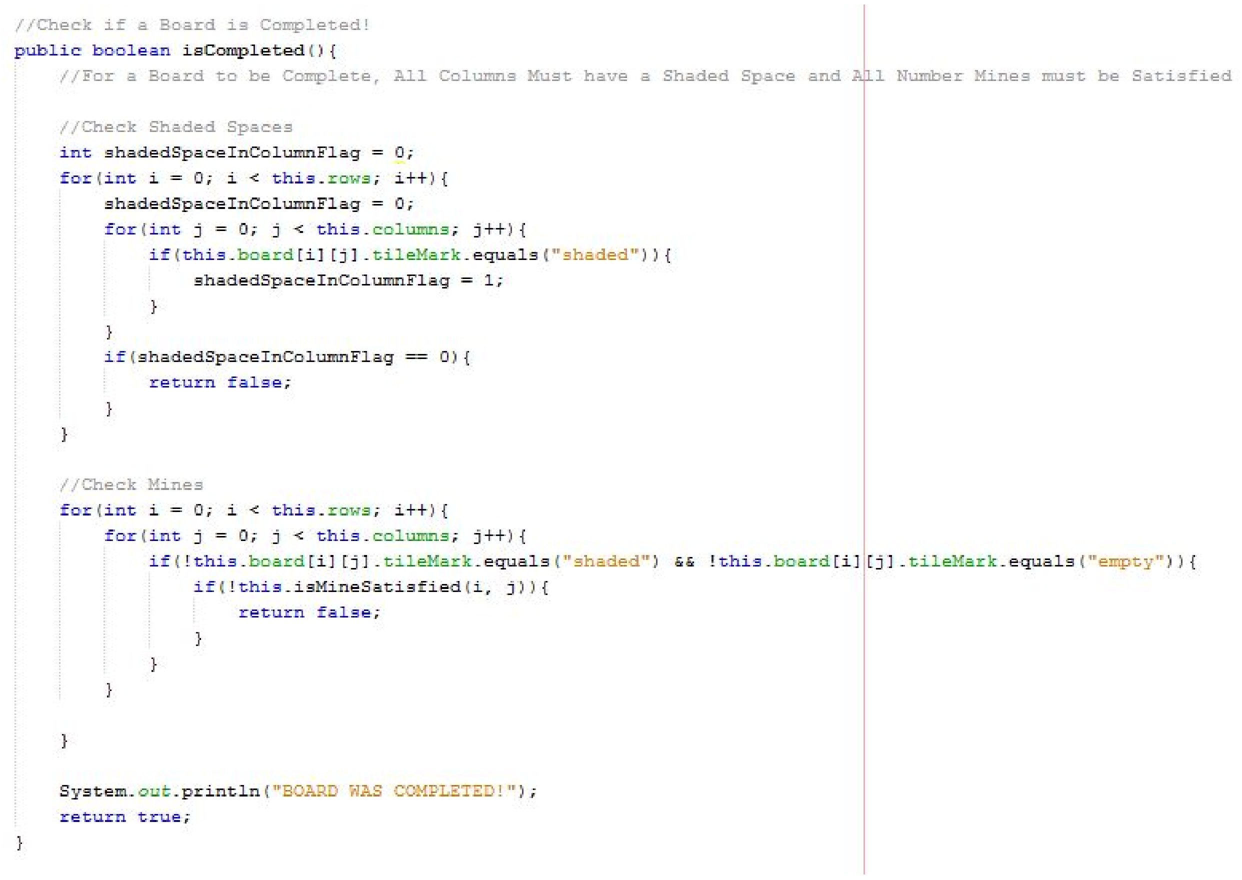
```
public boolean isMineSatisfied(int row, int column){
    //Initial Check
    if(this.board[row][column].tileMark.equals("empty") || this.board[row][column].tileMark.equals("shaded")){
        return false;
    }

    //Values
    int expectedSurrounding = Integer.parseInt(this.board[row][column].tileMark);
    int actualSurrounding = 0;

    //Check Surroundings
    //1 Up
    if(row > 0)
        if(this.board[row-1][column].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Left
    if(column > 0)
        if(this.board[row][column-1].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Down
    if(row < this.rows-1)
        if(this.board[row+1][column].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Right
    if(column < this.columns-1)
        if(this.board[row][column+1].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Up 1 Left
    if(row > 0 && column > 0)
        if(this.board[row-1][column-1].tileMark.equals("shaded"))
            actualSurrounding++;

    //1 Up 1 Right
    if(row > 0 && column < this.columns-1)
        if(this.board[row-1][column+1].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Down 1 Left
    if(row < this.rows-1 && column > 0)
        if(this.board[row+1][column-1].tileMark.equals("shaded"))
            actualSurrounding++;
    //1 Down 1 Right
    if(row < this.rows-1 && column < this.columns-1)
        if(this.board[row+1][column+1].tileMark.equals("shaded"))
            actualSurrounding++;

    //If Expected = Actual, Return True!
    if(expectedSurrounding == actualSurrounding){
        return true;
    }
    else{
        return false;
    }
}
```

Function to Update Number Mine Borders when Spaces are Filled to ensure Constraints



```
//Ensure Mines who Have been Satisfied has their Border Blacklisted
public void updateMineBorders(){
    for(int i = 0; i < this.rows; i++){
        for(int j = 0; j < this.columns; j++){
            if(!this.board[i][j].tileMark.equals("shaded") && !this.board[i][j].tileMark.equals("empty")){
                if(this.isMineSatisfied(i, j)){
                    this.blackListBorder(i, j);
                }
            }
        }
    }
}
```

Function to Check if a Board is Completed!



```java
//Check if a Board is Completed!
public boolean isCompleted(){
    //For a Board to be Complete, All Columns Must have a Shaded Space and All Number Mines must be Satisfied

    //Check Shaded Spaces
    int shadedSpaceInColumnFlag = 0;
    for(int i = 0; i < this.rows; i++){
        shadedSpaceInColumnFlag = 0;
        for(int j = 0; j < this.columns; j++){
            if(this.board[i][j].tileMark.equals("shaded")){
                shadedSpaceInColumnFlag = 1;
            }
        }
        if(shadedSpaceInColumnFlag == 0){
            return false;
        }
    }

    //Check Mines
    for(int i = 0; i < this.rows; i++){
        for(int j = 0; j < this.columns; j++){
            if(!this.board[i][j].tileMark.equals("shaded") && !this.board[i][j].tileMark.equals("empty")){
                if(!this.isMineSatisfied(i, j)){
                    return false;
                }
            }
        }

    }

    System.out.println("BOARD WAS COMPLETED!");
    return true;
}
```