A Game for Kings

AND COMPUTING STUDENTS, PROJECT MANUAL PART 2

Part 2 of the Project

Part 2 of the project is involved with modifying and extending existing structures for knowledge representation to develop your technical and practical skills for developing algorithms used in game playing. But first lets consider Figure 1 below. Essentially this part of the project is involved with designing and building Intelligent Software agents. As we can see in Figure 1 the software agent interprets the percepts coming from the environment (board position) and acts on the environment by making a move. There are many strategies involved with understanding what move should be made. For this part of the project you also need to demonstrate and describe the use of the structures for knowledge representation and your logical reasoning systems

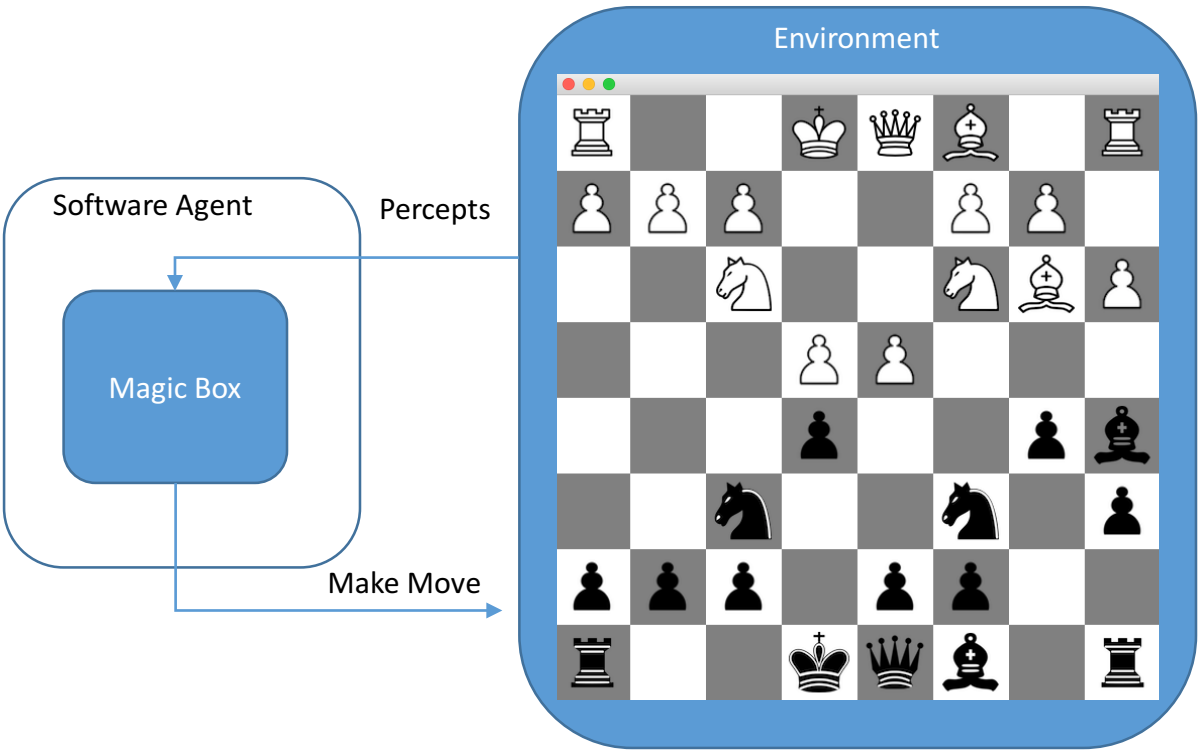

Figure 1: Agent interacts with the environment through the percepts received and communicates by making a move

The project manual will guide you through the project. First we need to revert back to the King Movements and create better structures for knowledge representation.

King Movements

Figure 1 below gives you a quick reminder of how the King should be able to move.

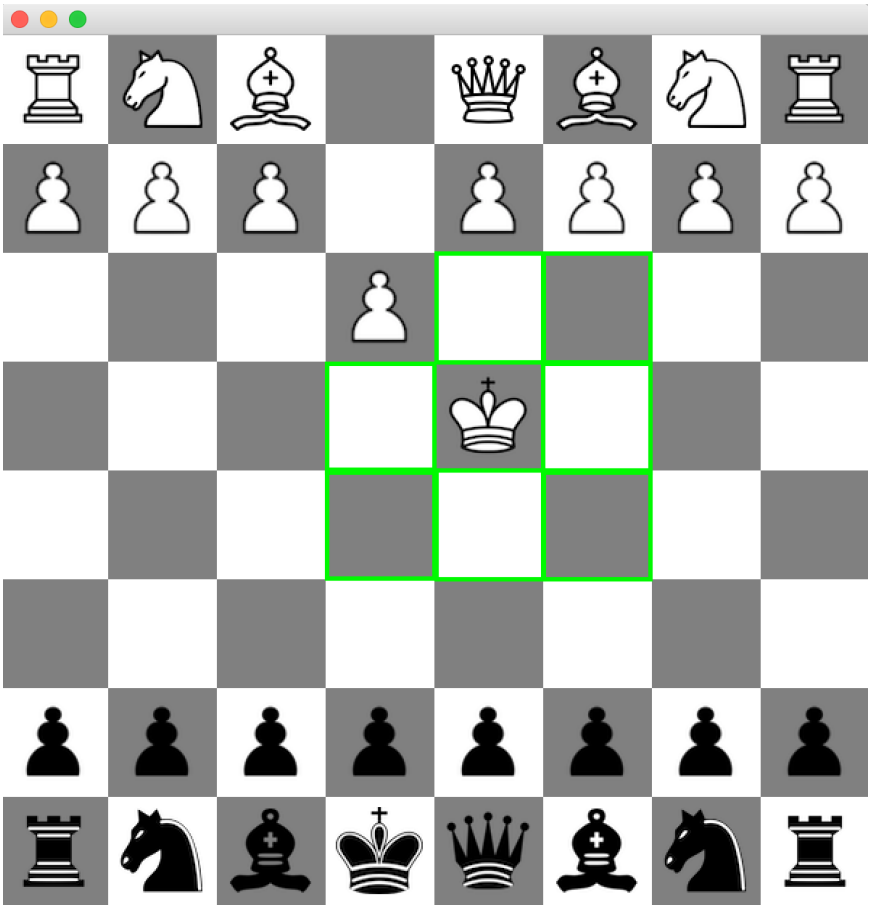

Figure 1: The King can only move one square at a time

The final condition that we ignored in Part 1 of the project was that you should make sure that you cannot move your king into a position that a piece is going to be directly attacking your King, i.e. you would be in Check! This will require you to create a method that loops through all the enemy chess pieces and finds out if they can see / attack the square you are moving too.

The complex part comes from understanding what squares your opponent is currently attacking. To help solve the King movement problems, we need to construct a new Object to represent a Square. This object should have an x-coordinate, y-coordinate and the name of a piece on the square if present, see Code Snippet 1 below.

```java
class Square{
  public int xCoor;
  public int yCoor;
  public String pieceName;

  public Square(int x, int y, String name){
    xCoor = x;
    yCoor = y;
    pieceName = name;
  }

  public Square(int x, int y){
    xCoor = x;
    yCoor = y;
    pieceName = "";
  }

  public int getXC(){
    return xCoor;
  }

  public int getYC(){
    return yCoor;
  }

  public String getName(){
      return pieceName;
  }
}
```

Code Snippet 1: The Square object enabling easier knowledge representation

This is great now as we have an object to represent squares on the board. However, we need to get the current state of where all enemy pieces are attacking. To help us achieve this we will be using our new class Square and the built in Stack from java.util library. The first method we need to create is to find all the pieces on the board, for example if we wanted to find where all the White pieces are on the board and store this representation in a Stack…check out Code Snippet 22 below.

```
private Stack findWhitePieces(){
      Stack squares = new Stack();
      String icon;
      int x;
      int y;
      String pieceName;
      for(int i=0;i < 600;i+=75){
        for(int j=0;j < 600;j+=75){
          y = i/75;
          x=j/75;
          Component tmp = chessBoard.findComponentAt(j, i);
          if(tmp instanceof JLabel){
            chessPiece = (JLabel)tmp;
            icon = chessPiece.getIcon().toString();
            pieceName = icon.substring(0, (icon.length()-4));
            if(pieceName.contains("White")){
              Square stmp = new Square(x, y, pieceName);
              squares.push(stmp);
            }
          }
        }
      }
      return squares;
    }
```

Code Snippet 2: Method to find the positions of all the White pieces on the board

Okay this is great, we now know were all the White pieces are located on the board. We know how the pieces move for our earlier work, so we just need to use the above Code Snippets to find out what squares White is attacking, check out Code Snippet 3 below for the code to find all the squares that are currently being attacked by White Knights.

```
private Stack getWhiteAttckingSquares(Stack pieces){
  while(!pieces.empty()){
    Square s = (Square)pieces.pop();
    String tmpString = s.getName();
    if(tmpString.contains("Knight")){
      tempK = getKnightMoves(s.getXC(), s.getYC(), s.getName());
      while(!tempK.empty()){
        Square tempKnight = (Square)tempK.pop();
        knight.push(tempKnight);
      }
    }
    else if(tmpString.contains("Bishop")){

    }
  }
}
```

Code Snippet 3: Method to return all the squares that are being attacked by White Pieces

We can see in Code Snippet 3 that the framework is created for finding out what squares are being attacked by White pieces. The method *getKnightMoves* can be seen in Code Snippet 4 below. Essentially you need to follow this method to be able to find the squares being attacked for each piece. One of the simplest strategies in Chess is trying to see how many pieces are attacking a particular square and the value of these pieces. This would be very important information to capture for your AI components.

```
private Stack getKnightMoves(int x, int y, String piece){
    Stack moves = new Stack();
    Stack attacking = new Stack();
    /*
        Essentially the knight can move in L shapes, so if we take in any square (x, y)
        the following list is the complete possible movements:
        (x+1, y+2), (x+1, y-2),(x-1, y+2), (x-1, y-2)
        (x+2, y+1), (x+2, y-1),(x-2, y+1), (x-1, y-1)
        You just need to make sure that the L shaped move stays on the board and doesn't
        take its own piece.
    */
    Square s = new Square(x+1, y+2);
    moves.push(s);
    Square s1 = new Square(x+1, y-2);
    moves.push(s1);
    Square s2 = new Square(x-1, y+2);
    moves.push(s2);
    Square s3 = new Square(x-1, y-2);
    moves.push(s3);
    Square s4 = new Square(x+2, y+1);
    moves.push(s4);
    Square s5 = new Square(x+2, y-1);
    moves.push(s5);
    Square s6 = new Square(x-2, y+1);
    moves.push(s6);
    Square s7 = new Square(x-2, y-1);
    moves.push(s7);
    for(int i=0;i < 8;i++){
        Square tmp = (Square)moves.pop();
        if((tmp.getXC() < 0)||(tmp.getXC() > 7)||(tmp.getYC() < 0)||(tmp.getXC() > 7)){
        }
        else if(piecePresent(((tmp.getXC()*75)+20), (((tmp.getYC()*75)+20)))){
            if(piece.contains("White")){
                if(checkWhiteOponent(((tmp.getXC()*75)+20), ((tmp.getYC()*75)+20))){
                    attacking.push(tmp);
                }
                else{
                    System.out.println("Its our own piece");
                }
            }
            else{
                if(checkBlackOponent(tmp.getXC(), tmp.getYC())){
                    attacking.push(tmp);
                }
            }
        }
        else{
            attacking.push(tmp);
        }
    }
    Stack tmp = attacking;
    colorSquares(tmp);
    return attacking;
}
```

Code Snippet 4: Finding out the squares being attacked by a Knight

The last very useful method that we will discuss in this manual is a method that is used to highlight squares. For beginner Chess players it may be nice to show them were a piece can move to. For example, if a user clicks on a piece, the possible squares that the piece can move to should be highlighted. Check out Code Snippet 5 for this functionality.

```
/*
    Method to color a number of squares...
*/
private void colorSquares(Stack squares){
    Border greenBorder = BorderFactory.createLineBorder(Color.GREEN,3);
    while(!squares.empty()){
        Square s = (Square)squares.pop();
        int location = s.getXC() + ((s.getYC())*8);
        JPanel panel = (JPanel)chessBoard.getComponent(location);
        panel.setBorder(greenBorder);
    }
}
```

Code Snippet 5: Method to colour a number of squares

Check out Figure 2 to see the application of using *colorSquares*.

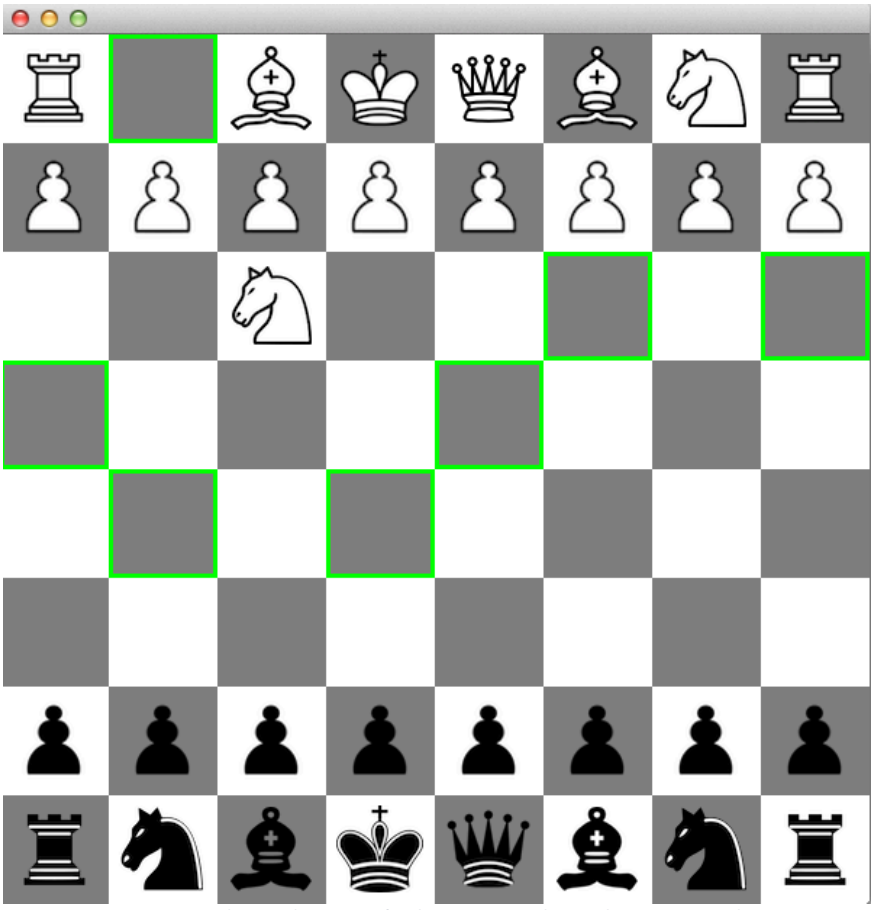

Figure 2: The application of colorSquares when selecting a Knight

Perfect. Now you have all the components to be able to complete your Intelligent Software Agents. To complete the project you need to be able to create three modes for the AI component. The AI component will always move as the White player. The following strategy is recommended:

1. While not considered an "AI" solution marks will be awarded if you create an automated player that simply makes random **valid** moves. To achieve this the logic needs to be embedded to play the game, i.e. the game starts with a White piece moving followed by a black piece moving. This sequence continues until one player is placed in Check Mate. The simplest way of implementing this is to have a method that returns all the possible moves for a player and then you simply make a random selection, i.e. extending the code above to identify the attacking squares for White.

2. The second level of AI can take many forms and is a simple extension of the logic from the above level. One possibility for the AI component would be an aggressive player that exchanges pieces when making a move if possible, or trying to take control of the board. Essentially this AI component should look at all possible moves and make an informed decision based on the strategy being implemented. Like the above approach you need to find all the squares that the opponent can move into and evaluate the possibilities at each stage. However, the approach should be limited by a single state perspective (one level deep).

3. The final level of AI should apply a searching strategy and look ahead at least two moves and make an informed decision. This is exactly the same as the second level however the evaluation function should be applied after two moves and then using the Min-Max approach traverse back up the tree to make the appropriate move. Obviously the conditions of the Min-Max approach will not be win, lose or draw but should be based on something else, perhaps the score of the pieces left on the board.