Gobang Game

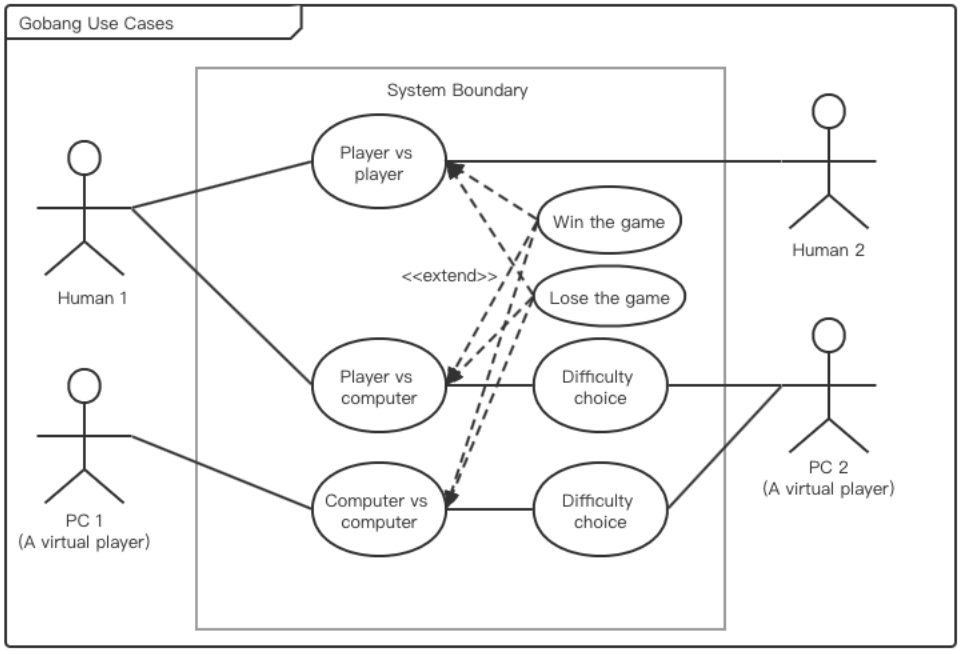
Haoxiang Wang（482006282）

1. Introduction

Gobang is a fun and challenging game with simple rules. In this game, two players take turns playing chess on a 15*15 board, one plays black and the other plays white. Black put chess first. When someone's five chess pieces are in a row, he wins the game. In this project, I design four modes for this game: player vs player, player vs computer, computer vs player, and computer vs computer. The computer has three levels of difficulty: easy, medium, and hard. User can set which player and computer play chess first.

2. Demand analysis

The use case diagram for my project is:

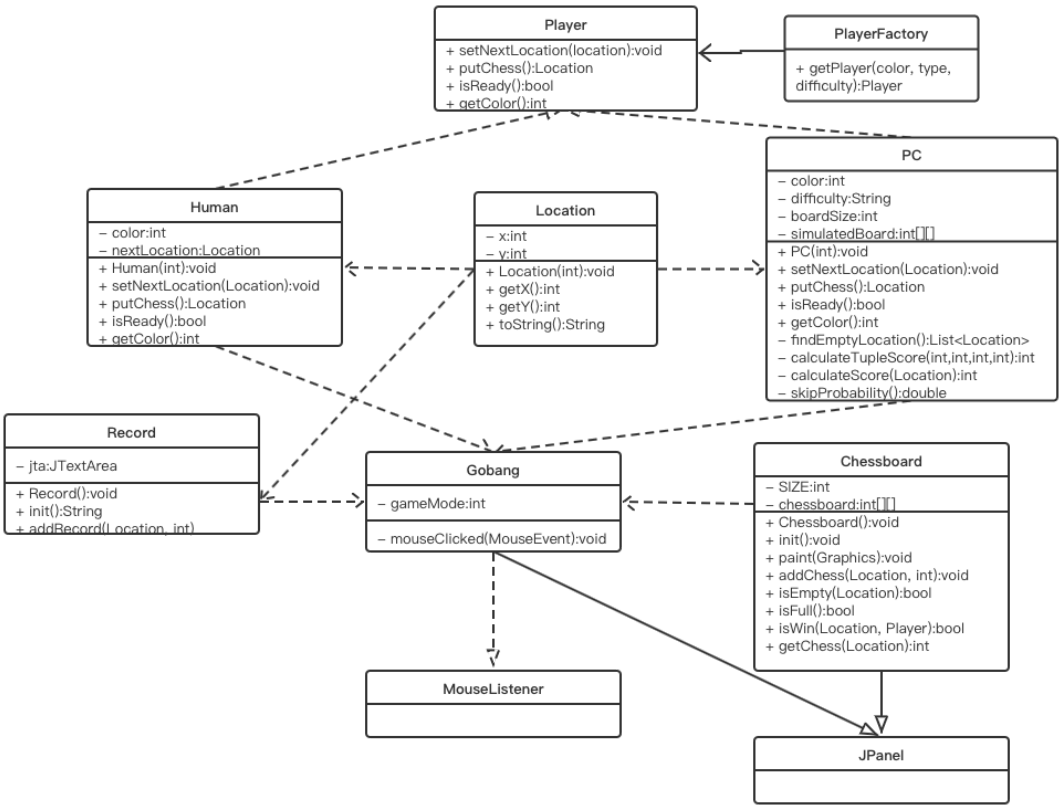


There are two kinds of chess, black and white, and each can be played by a human

or a PC, there are 2*2=4 modes in total. The user can choose the player of the black side and the player of the white side, and if there is a computer as the player, he can also choose the difficulty of the game (the intelligence of the computer).

3. Class Design

The UML class diagram for the GoBang is:



Since each color of chess can be a computer or a human, I used the factory pattern to automatically generate a PC or a human based on the user's settings. For humans, only when the mouse activity is captured, a legal position is obtained, and after setNextLocation(), chess can be played on the chessboard. At this time, isReady() is True. The PC can make the next move at any time according to the situation on the field, so the PC's isReady() is always True. In addition, the PC class also includes functions for calculating the position score and judging the optimal solution.

4. Implementation

Environment: IntelliJ IDEA 2021.3.1 (Ultimate Edition) on MacBook (M1 processor)

4.1.Factory Pattern

Player interface:

```
interface Player {
    void setNextLocation(Location location);
    Location putChess(Chessboard chessboard, Record record);
    boolean isReady();
    int getColor();
}
```

Human class:

```
public class Human implements Player {
    private int color;
    private Location nextLocation;

    public Human(int color) {
        this.color = color;
        nextLocation = null;
    }

    @Override
    public void setNextLocation(Location location) { nextLocation = location; }

    @Override
    public Location putChess(Chessboard chessboard, Record record) {...}

    @Override
    public boolean isReady() { return nextLocation != null; }

    @Override
    public int getColor() { return color; }
}
```

PC class:

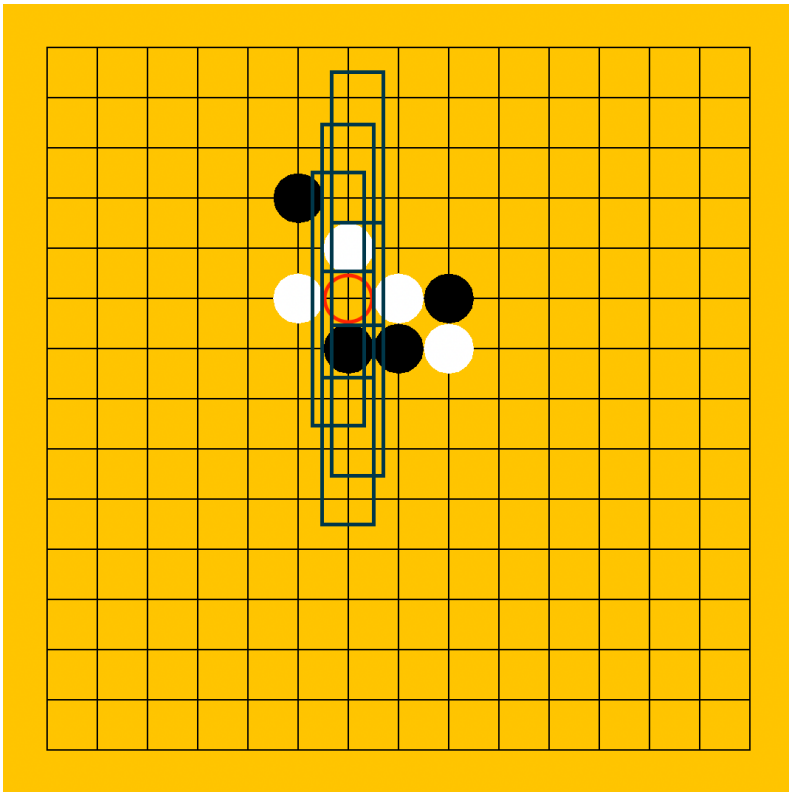
```
public class PC implements Player {

    private int boardSize;
    private int[][] simulatedBoard;
    private int color;
    private String difficulty;

    public PC(int color, String difficulty) {...}

    private List<Location> findEmptyLocation() {...}

    private int calculateTupleScore(int x, int y, int dx, int dy) {...}

    private int calculateScore(Location location) {...}

    public double skipProbability() {...}

    @Override
    public void setNextLocation(Location location) { return; }

    @Override
    public Location putChess(Chessboard chessboard, Record record) {...}

    @Override
    public boolean isReady() { return true; }

    @Override
    public int getColor() { return color; }
}
```

Player factory:

```
public class PlayerFactory {
    public Player getPlayer(int color, String type, String difficulty) {
        if (type == "Human") {
            return new Human(color);
        }
        else if (type == "PC") {
            return new PC(color, difficulty);
        }
        return null;
    }
}
```

4.2.Evaluation Method

For the score of a position on the board (for example, the red position), we first count the quintuple with it as the bottom, that is, the first rectangle, then the second, the third, and finally the quintuple with this position as the top. In this way, we count the scores in the y-axis direction.



In the same way, then count the scores of the quintuple in the x-axis direction and the two diagonal directions. Add them together to get the score for this position. Traverse the scores of all positions and select the point with the highest score probabilistically according to the difficulty. This position is the position of the computer. Besides, if two locations have the same score, pick the location closer to the middle of the board.

The scoring of the different quintuplets is as follows (assuming this move is for Black to play). The reason why black scored higher for the same number of pawns is that this move is black's turn to play. For example, there are four black chess pieces in a certain quintuple, and the black side can win here with this move. But with four white pieces, black can play here to prevent white from winning. The reason why the score

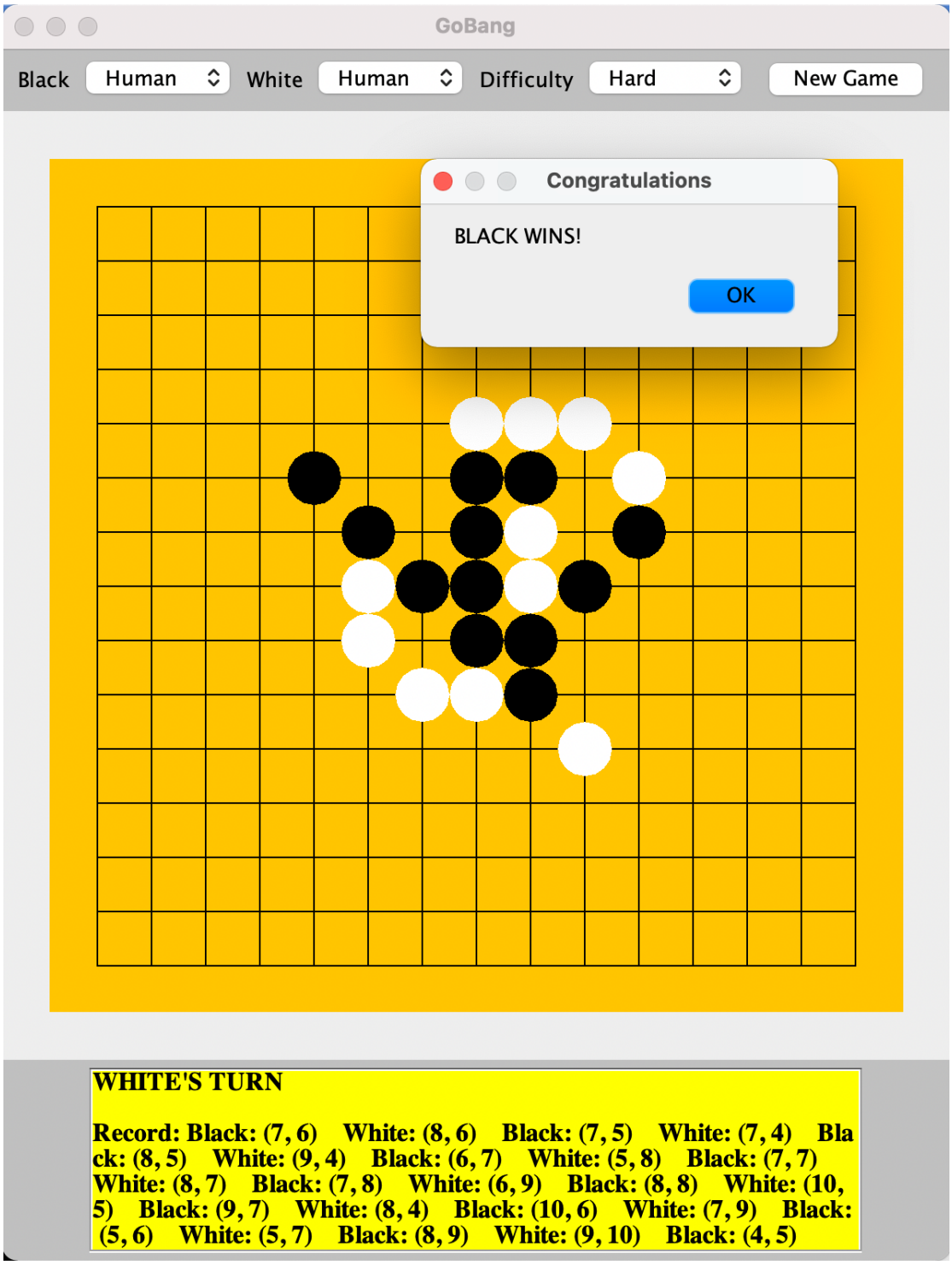
of more than 1 black & more than 1 white is 0 is that this quintuple can't possibly be the key to winning.

Quintuplet	Score
0 black & 0 white	7
1 black & 0 white	35
2 black & 0 white	800
3 black & 0 white	15000
4 black & 0 white	800000
0 black & 1 white	15
0 black & 2 white	400
0 black & 3 white	1800
0 black & 4 white	100000
More than 1 black & more than 1 white	0

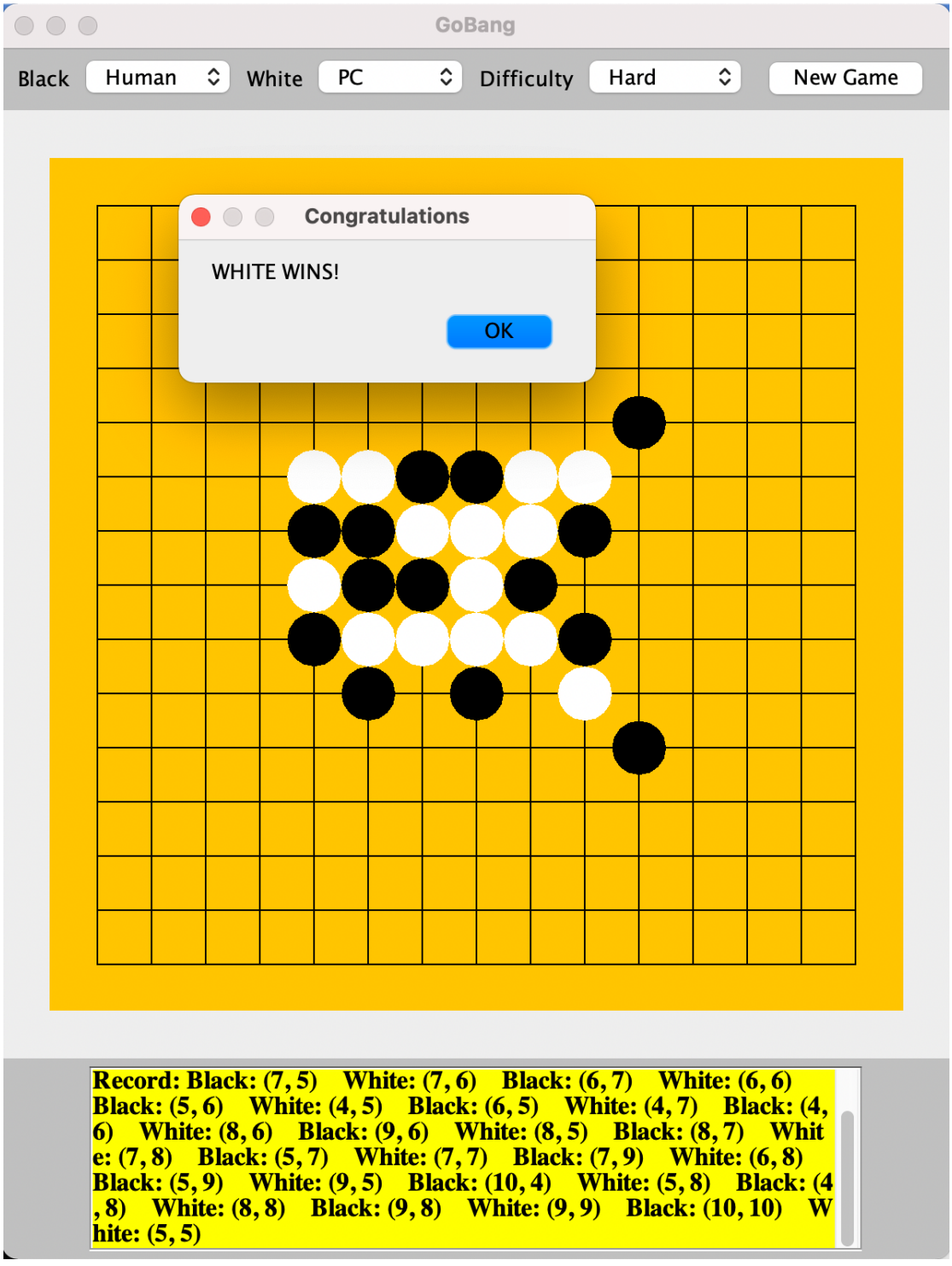
This scoring method can achieve better chess effect after many experiments. In my many games with the computer, the computer still has a high win rate when I go first, and the computer always wins when I go back.

5. Test

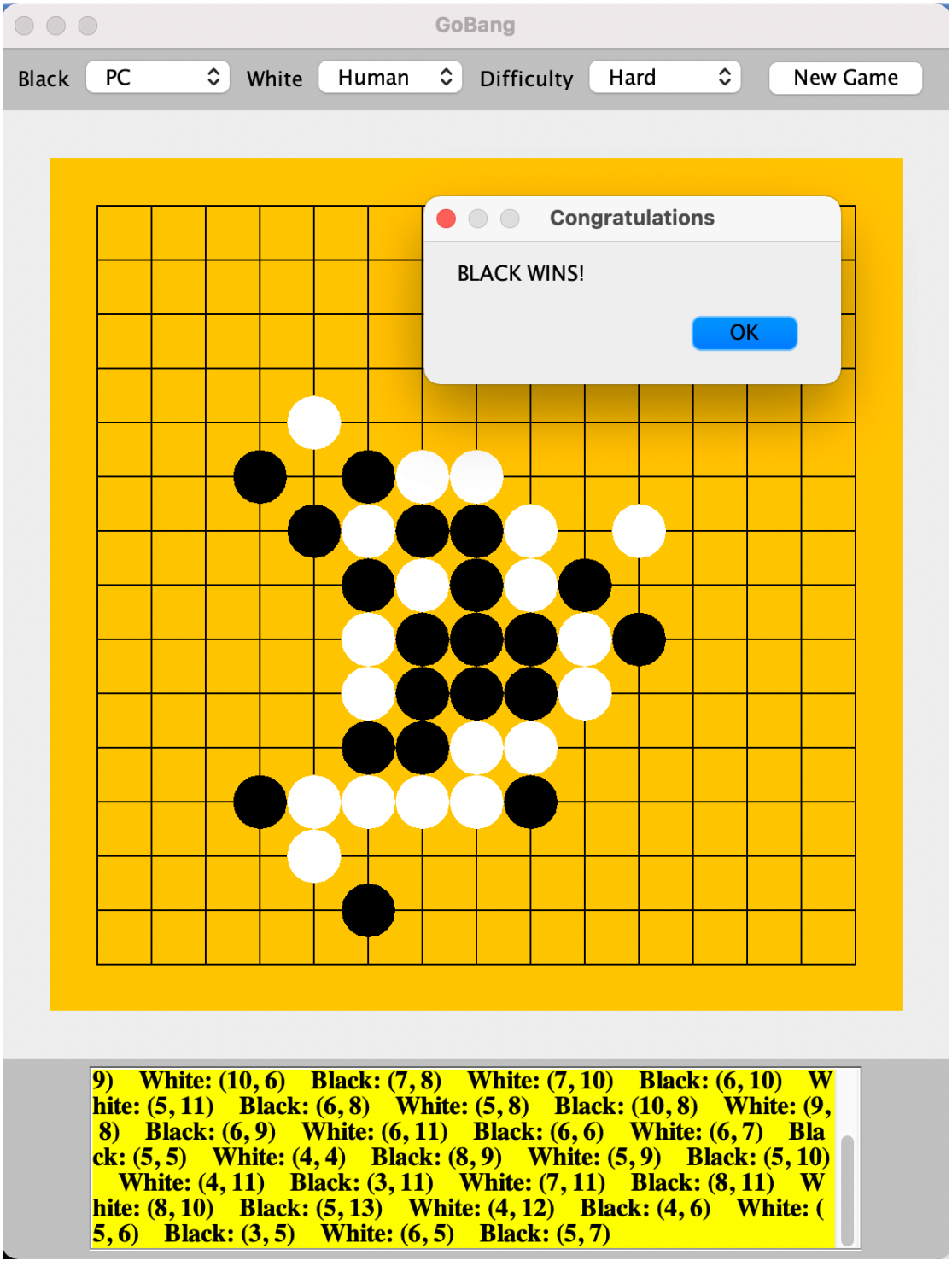
Black: Human vs White: Human (black win)



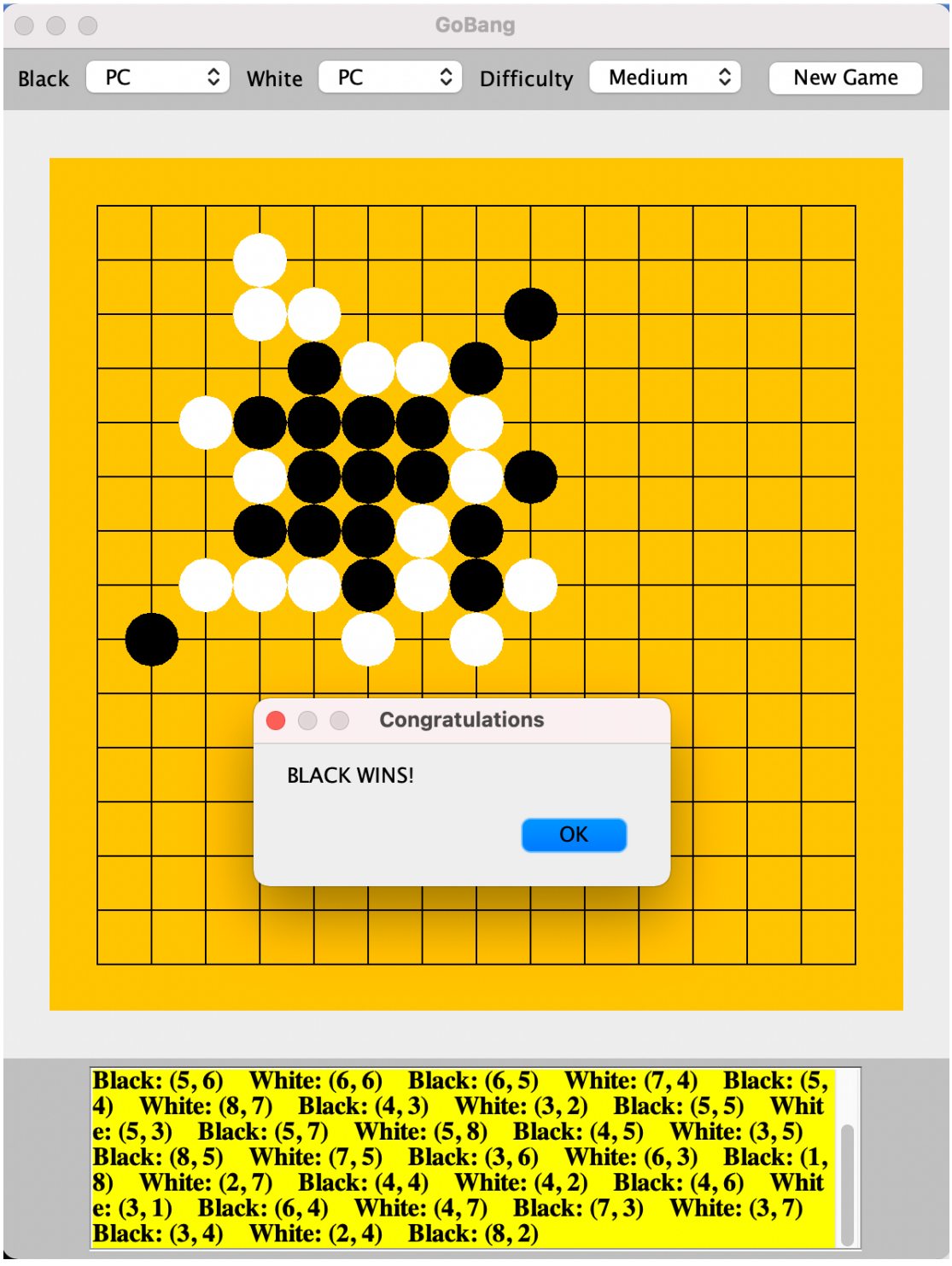
Black: Human vs White: PC, hard module (white win)



Black: PC vs White: Human, hard module (black win)



Black: PC vs White: PC, medium module (black win)



After click the New Game button: