MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING

ARTIFICIAL INTELLIGENCE

CSE4082

HOMEWORK 2

Group Members:

ONURCAN İŞLER 150120825

ERKAM KARACA 150118021

Table of Contents

1. METHODS AND CLASSES 3

2. MINIMAX ALGORITHM WITH ALPHA BETA PRUNNING 4

3. HEURISTICS 4

3.1. Expert Heuristic 5

3.2. Apprentice Heuristic 6

3.3. Novice Heuristic 7

1. METHODS AND CLASSES

The game starts with `start()` function. It is used to keep the game running all the time. Each time a game is played the user can return back to the main menu and start another game. In our code, we kept everything simple. We have written a separate function for each game type.

- `ai_ai()` function is for AI player vs AI player game,
- `human_ai()` function is for AI player vs Human player game,
- `human_human()` function is for Human player vs Human player game.

- To perform the movements for Human players, `get_human_move()` function is used.
- To perform the movements for AI players, `get_AI_move()` function is used. In fact, `get_AI_move()` function then calls minimax() function to perform AI moves.

We have also defined Utility functions to perform simple operations:

- `is_winner(table,isRed)` to check if a player with color `isRed` is winner,
- `perform_move(table,col,isRed)` to perform move for the color `isRed`. Move is performed in '`col`'th column.
- `initialize_table()` function is used to initialize the table with size 7x8.
- We have also used `print_table()` function to print the tables before and after each move.

Lastly, we have defined 3 separate Python classes to perform heuristic evaluations. We will be discussing inner workings of these heuristics in the later parts of the report. What is good about our code is that we have defined many error checking mechanisms that will prevent the user to enter invalid inputs. That way, the game will be more playable and fairer. Each input reading code is put inside an infinity while loop so that it will run until player enters something valid.

2. MINIMAX ALGORITHM WITH ALPHA BETA PRUNNING

We have used alpha-beta pruning to reduce running time of the minimax algorithm. This function takes `heuristic_type` argument to decide on the type of the heuristic to be used. It has `depthlim` variable to limit the depth when deciding on possible moves. It also has `isRed` variable which is used to decide the current player to play. If `isRed` is True, then Red player is playing. Black is otherwise. The first move is always made by Red player. In minimax algorithm, Red tries to maximize its utility, whereas Black tries to minimize Red's utility. So, Red is maximizer and Black is minimizer. We have written heuristic functions accordingly.

- So, how these heuristic values are used?

First thing to note is that heuristic evaluations are only performed at the deepest level where `depth==depthlim`. By using these heuristic values, AI players decide on the moves they play.

Now, let's discuss about the idea of alpha-beta pruning. The main idea is to prune the tree so that number of computations are reduced. We do that with simple idea:
If we find a utility value worse than the other player's guaranteed utility value, we always play the move that yields that awful utility value. But on the other hand, since the other player will not be playing from this awful utility (because he has a better chance of moving), we do not think beyond the state that has awful utility. Because it is unnecessary, the other player will never play here. In this way, computations are minimized.

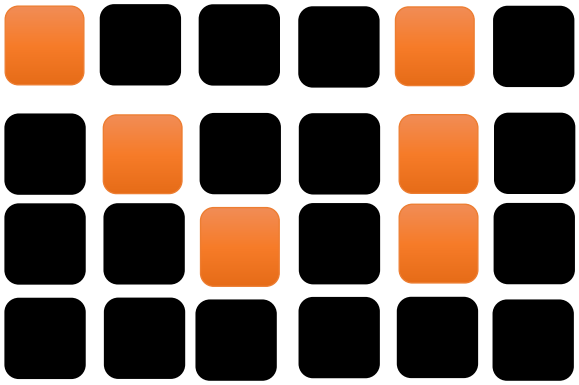
3. HEURISTICS

We have defined three cute heuristics for the evaluations of the board. All these heuristics returns positive values when Red has better advantage, and returns negative values when Black

has better advantage. Here, better advantage means being closer to the finishing move. Now, let's go over these heuristic functions one by one.

3.1. Expert Heuristic

We start with the best heuristic. The idea is entirely our own. In fact, the idea is a bit intuitive. We are counting the length of the longest line can be made by placing a piece at an empty slot. If we can make a longer line by placing the piece, then we are lucky. Our utility will be good. Here by 'line' we mean pieces that are in order. See,



There are 2 lines with length of 3 in this pile of pieces. The main idea is to compute the line with maximum length after placing a piece. But notice in Connect-4 game, we cannot place our pieces anywhere we want. Consider:

```
For example consider the table:
| | | | | | | | |
| | | | | | | | |
|R|B| | | | | | |
|R|B| | | | | |B|
|R|R|R|B|B| | |R|
|B|B|B|R|R|R| |R|
|B|R|B|B|B|R| |R|
Here, we consider the empty slots represented by letter 'X'.
| | | | | | | | |
|X|X| | | | | | |
|R|B| | | | | |X|
|R|B|X|X|X| | |B|
|R|R|R|B|B|X| |R|
```

```
|B|B|B|R|R|R| |R|
|B|R|B|B|B|R|X|R|
We compute the length of the largest line we can make if we place
R or B into these X slots. We then compute the balance accordingly.
```

So, we are computing the maximum length of lines at these 'X' slots. This idea may look simple, but in fact, it is so powerful that it beats us all the time even if depth limit is defined as 1. But one weakness of this heuristic is that it plays extremely savage and does not play safe. Of course, it can defend some moves where contiguous 3 opponent pieces are placed. But, in most cases it simply ignores when 2 opponent pieces are piled. Recall that we said this heuristic considers the maximum length of the line after the move is made. So, it mostly focuses on winning the game not defending itself.

To get the evaluation value, we perform following computation for each empty slot.

```
balance = balance + Heuristic3.getmax_profit(table,row,col,"R")
balance = balance - Heuristic3.getmax_profit(table,row,col,"B")
```

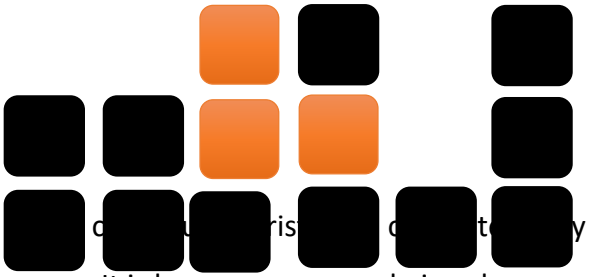
Also, `getmax_profit()` function defined as:

```
return 30**(LENGTH_OF_THE_LONGEST_LINE)
```

So, we use power of thirty as a piece is added. 30 here is completely result of our own observations. If you increase it the algorithm will play even more savage and eventually will start ignoring opponent's adjacent stones. If you decrease this value, then it will try to prevent even if only two pieces are adjacent. In this case, it will not be playing for winning but instead to prevent opponent to win. The game will most likely end up with draw in this case.

3.2. Apprentice Heuristic

This heuristic basically counts all the adjacent same pieces. So, a greater number of pieces close together is good. In most cases, this heuristic prefers putting the pieces at corners so that the pieces will pile up. For example, consider the following arrangement:



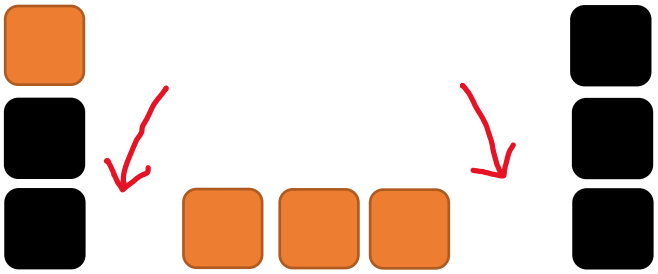
o ris c y for red player as follows: 1 + 2 + 1 = 3.

- It is because upper red piece has one adjacent red piece,
- The piece on the right has one adjacent red piece,
- The middle piece has two adjacent red pieces.

It makes 1 + 2 + 1 = 3. So, the greater the number of same-colored pieces the greater the greater the utility. At the end, we return subtraction of `(RedAdj–BlackAdj)`. We do this because the red is the maximizer, and the black is the minimizer. One huge weakness of this heuristic is that it may not be able to react when three pieces are piled up. So, it will probably lose the game if depth is defined to be small.

3.3. Novice Heuristic

The whole idea of our dumbest heuristic is that checking if Red or Black has won. Otherwise, it returns 0. In most cases it will handle the situations when opponent has piled up 3 pieces. But one very huge disadvantage is that once a player has able to win on multiple sides than it simply performs dumb move. See,



We can see red can win this game with two possible moves: by placing a red piece on either side. In this case, Novice Heuristic has nothing to do about it. It did not see that coming even if depth is around 5 or so. Certain defeat is at hand.