Algorithms and Data Structures
Programming Project 3
Phase two

Pedram Bashiri
Yogarakshith Muralidhar Rao

The game is being played in a JFrame consisting of 6*7 JButtons each represents a piece of the game grid. These buttons are only used to interact with the player while actual status of the game is stored in a two dimensional array named VirtualBoard in which each element can be -1, 0, 1. 0 means the element is empty, 1 means is used by player and -1 means assigned to CPU (computer).

This documentation specifies the strategies that our team came up while improving the program.

1.CPU play

The original strategy used was using the java random function for the computer to choose a position in the board as opposed to choosing the positions that block the player from scoring.

1.1. Using the player's last column play data(Defensive Strategy)

The first strategy we came up was to use the data of the player's last move. Based on the move made by the player,one of the three playable neighbouring columns are chosen by the program and filed by the CPU's coin. This is done because the next corresponding move is most likely to be made based on the last move of the player. Theoretically, this would be able to raise the difficulty and the accuracy of the CPU's moves. This can be implemented by initializing a global variable that stores the players last move and choose a neighbouring position near that location and put a coin.

Pseudo Code:

```
playerturn : player turn value
int columnnumber = random(neighbours of playerturn)
while(true)
{
        while (Board[i][columnnumber] is not empty)
        {       iterate i through 0 till empty
                if i = -1, column is full;
        }
        if(column is full)
                choose different columnnumber
        else
```

Enter CPU coin in Board[i][columnnumber]
}

Performance:

The proposed code is similar to the old random code, except that a global variable narrow down the random to atmost 3 columns as opposed to all 6 columns in the previous method. The runtime is therefore going to be the same as the previous program.

Accuracy:

This entity changes in comparison to the previous method. The bounds for placing the CPU's coin has been reduced at least by 50% for a 6 x 7 board (atmost 3 columns). Mathematically, this would mean that the chances that the player's next move will be blocked will be more as the CPU will drop the coin in a max of 5 positions (unless the player places a coin in the edge columns). Hence the factor of difficulty becomes double.

Storage requirement:

The storage requirement too doesn't change much either. Space optimization is restrained to the virtual board, players' connect coins and the logical variables. Only a global variable is declared and updated with every time the player plays his/her turn. For a 6 x 7 board, the variable is updated 21 times. The variable is updated instead of creating new variables each move. Hence the Space Complexity is the same as the previous case.

1.2. Considering all possible options

In this strategy which is some sort of greedy approach, each time CPU considers all possible moves and tries to calculate the maximum number of possible connected coins. If this move can lead to a 4-connect, CPU will choose this option. In the next stage, it looks to see if it is possible to make a 3-connect. If yes, CPU chooses this option. After that it considers all possible 2-connects. If none of them exists, it just randomly chooses a point from all possible choices.

This is the strategy that we used in our implementation, so the code is already a part of the program.

Pseudo Code:

MovePredictionmethod (arguments)
{
Initialize newMove by setting possible connections

```
for (Moves move : PossibleMoves)
{
        if (move.getPossible4Conneteds() > 0){
                Chosen = move;
                found =true;
        }
}
for(Moves move:PossibleMoves){
        if (move.getPossible3Conneteds() > 0){
                Chosen = move;
                found =true;
        }
}
for(Moves move:PossibleMoves){
        if (move.getPossible2Conneteds() > 0){
                Chosen = move;
                found =true;
        }
}
else chooseRandomly();
```

Accuracy:
As we made two different strategies to play against each other, it turned out that the second strategy won 9 out of ten games. This result makes sense considering two rules that the second strategy uses to makes its choices more wisely. Especially in early moves (before at least half of the board is occupied) the second strategy plays more goal-oriented which leads him to better results.

1.3. Comparison of the 2 strategies.

After trial runs and implementation, we decided to go with greedy algorithm approach for these reasons.

a. Though the first strategy increases the accuracy by at least 50% for a 6 x 7 board(comparing to the strategy we used in the first phase), it still employs a random function on those 3 columns, thereby not actually raising the CPU's chances of winning by much significance. Whereas the greedy algorithm approach chooses the most optimal position that the player has a chance to score. This means if 3 or more coins are consecutively placed, then the CPU will choose the corresponding position to make a 4-connects. This in actuality, proves a bit more significant than the first approach.

b. The Storage vs Performance Tradeoff is very favourable to Performance and is very minimal. The second method has a little more space complexity versus the first strategy,

due to the usage of many checking conditions and variables for the moves. But the accuracy and performance is more effective than the cost of space it uses and the computer's connect coins are placed so as to challenge the player. Considering the machines that are used, space complexity is not a big problem as opposed to performance.

2. RESULT DETERMINATION

2.1. Point evaluation for individual connection dots

In this strategy, the algorithm is being used for counting the quadruples works like this. For each piece of the grid the program counts all possible quadruples which start from that piece in every possible direction (vertical, horizontal, and diagonal directions). For each direction we have a method to see if there is a quadruple in that direction which starts with that piece. Total number of all quadruples equals to the players score. Time complexity of iterating through all elements is apparently O(m*n). Also for each method we have another O(m*n) . As a result the total complexity in terms of Big O is (m^2 * n^2). But, in practice we narrow the iterations down in quadruples methods (horizontalQuadruples, verticallQuadruples, …) using some simple facts, e.g. for an element which is in the 5th column we do not count horizontal quadruples because no horizontal quadruple can start from this column(there are 2 other columns in the right). All in all, using this approaches we decreased the average case by more than 2.
the good point about this strategy is that it does not require any secondary memory. All calculations are done on the original matrix, *VirtualBoard.*

2.2. Using longest connection method

This methods involves evaluating the longest connection for a given color and calculating the number of points scored by each player. This is inspired from the DAG tracing algorithms. Unlike DAG, here the tracing will be limited to straight horizontal,vertical or diagonal lines. We would find the lines that have a length of at least 4 or more and less than or equal to 7 (coz of 6 x 7 dimensions). This would require scanning through the board for connections of lengths 4 - 7. Then we would calculate 1 point for 4 connection line, 2 points for 5 connection line, 3 points for 6 connection line and 4 points for 7 connection line.

Pseudocode:

```
PointCalculationmethod(Arguments)
{
```

```
 for(iterator i: 4 through max(dimension))
 {
        if(Board contains instance of connectionlength = i)
                instances++;
        if(i =4)
          score = instances*1
        else if(i = 5)
          score = instances*2
        else if(i = 6)
          score = instances*3
        else(i = 7)
          score = instances*4
        instances = 0;
   }
   Return score;
}
```

Performance:

Performance for calculating points this method has to be done after eliminating duplicates over various instance of different lengths. That is, a line of length 4 can also be included in lines of length 5, 6 or 7. This means we must remove duplicates from being counted. This can be limited by evaluating the lengths in the reverse order and avoiding recounting the high length lines. But this leads to a different problem of having to remove lines having higher lengths before evaluating for lower length lines. This creates a problem when a non-accounted lower length line has one or more connections overlapping with the higher length connection. This would result in inconsistency, unless processed correctly.

Accuracy:

The Accuracy is 100% like the first strategy, given on the one important condition : that it doesn't involve duplicates. But as discussed in the performance strata, eliminating duplicates will involve more work and careful selective elimination to be accurate. Hence it is more risky option as opposed to the first strategy, where each of the connections are considered the end-point of a given connect, thereby fixing the boundary for the connection. The Time Complexity is O(n^(max(dimension)-connection length)); in this case it should be O(n^3). This is the processing time for 1 iteration. This is more than the first strategies time complexity.

Storage Requirement:

This method would require for more storage for processing removal of duplicates as instances of higher length lines must be stored in memory to compare and eliminate the recurring lines in the board. Though the memory can be closed later to avoid memory leak, it still requires

space in the system during run-time. Hence it has more space complexity in comparison to the first method.

2.3 Comparison two strategies.

In comparing the two strategies, we decided to go with the first method for 2 reasons.

a. Both the strategies produce 100% accuracy, but the second strategy requires more processing space and time than the first strategy, as it involves storing higher-length line data, deleting duplicates etc. in contrast to the first method, which doesn't require line data to be stored and each individual connection is evaluated, eliminating the necessity to worry about duplicate elimination.
b. If we had to use the same program for boards of larger sizes, then we have to rework a lot on the second strategy, which would require more instances*lengthsize calculation, whereas the first strategy can be easily incorporated and the connections are evaluated individually to find the connect dot, irrespective of the size of the board.