CS 2210a — Data Structures and Algorithms
Assignment 2: (n, k)**-Tic-Tac-Toe**
Due Date: October 17 at 11:59 pm
Total marks: 20

1 Overview

For this assignment you are required to write a Java program that plays (n, k)-tic-tac-toe; (n, k)-tic-tac-toe is played on a board of size $n \times n$ and to win the game a player needs to put k symbols on adjacent positions of the same row, column, or diagonal. The program will play against a human opponent. You will be given code for displaying the gameboard on the screen.

2 The Algorithm for Playing (n, k)-Tic-Tac-Toe

The human player always starts the game. The human uses 'X's and the computer uses 'O's. In each turn the computer examines all possible plays or moves and selects the best one; to do this, each possible move is assigned a score. Your program will use the following 4 scores for moves:

- `HUMAN_WINS` = 0: this score is given to a move that will ensure that the human player wins
- `UNDECIDED` = 1: this score is give to a move for which it is not clear which player will win
- `DRAW` = 2: this score is given to a move that will lead to a draw (i.e. no player wins the game)
- `COMPUTER_WINS` = 3: this score is given to a move that will ensure that the computer wins

For example, suppose that the gameboard looks like the one in Figure 1(a). If the computer plays in position 8, the game will end in a draw (see Figure 1(b)), so the score for the computer playing in position 8 is 2 (`DRAW`). Similarly, the score for the computer playing in position 9 is 0 (`HUMAN_WINS`).

A *configuration* is the positioning of the symbols on the gameboard. For example, the configuration shown in Figure 1(b) corresponds to a game that ends up in a draw. The configuration in Figure 1(c) corresponds to a game won by the human player. Each configuration is also assigned one of the 4 above scores. So, the configuration in Figure 1(b) gets a score of 2 and the configuration in Figure 1(c) gets a score of 0.

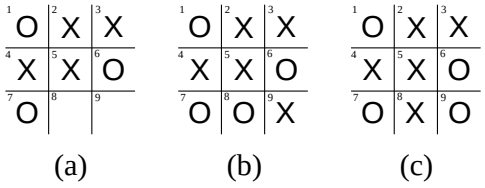

Figure 1: Board configurations.

Note. You can skip the rest of this section and Section 2.1, as you do not need this to complete the assignment. However, if you are interested in knowing how a computer program that plays a 2-player game works, then keep on reading.

To compute scores, your program will use a recursive algorithm that repeatedly simulates a move from the computer followed by a move from the human, until an outcome for the game has been decided. This recursive algorithm will implicitly create a tree formed by all the moves that the players can make starting at the current configuration. This tree is called a *game tree*. An example of a game tree is shown in Figure 2.

Assume that after several moves the gameboard is as the one shown at the top of Figure 2 and suppose that it is the computer's turn to move. The algorithm for computing scores will first try all

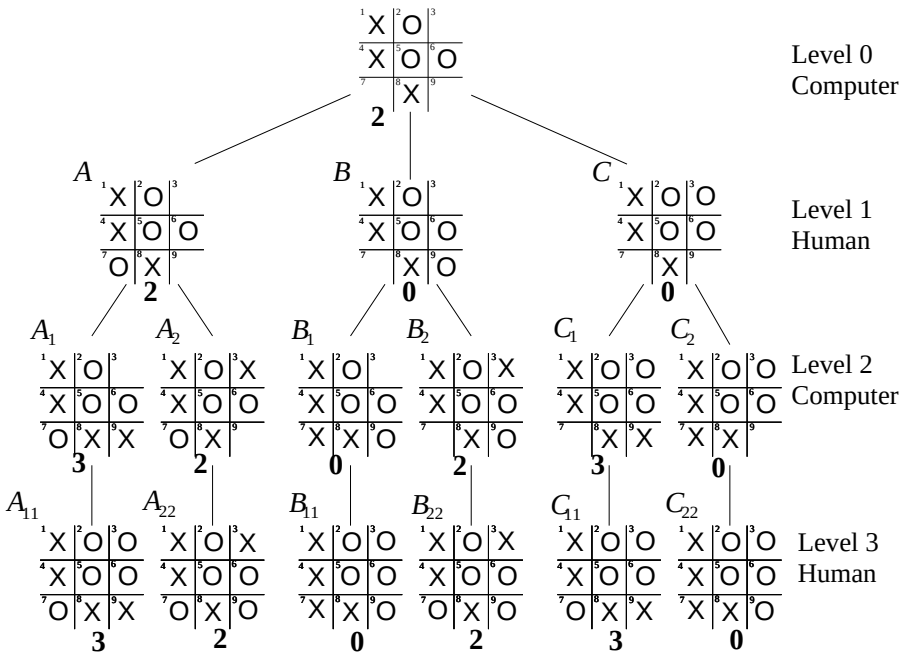

Figure 2: A game tree.

possible moves that the computer can make: A (play at position 7), B (play at position 9), and C (play at position 3). For each one of these moves, the algorithm will then consider all possible moves by the human player: A_1 and A_2 (if the computer plays as in configuration A), B_1 and B_2 (if the computer plays as in configuration B), and C_1 and C_2 (if the computer plays as in configuration C). Then, all possible responses by the computer are attempted, and so on until the outcome of each possible sequence of plays is determined.

In Figure 2 each level of the tree is labelled by the player whose turn is next. So levels 0 and 2 are labelled "computer" and the other 2 levels are labelled "human". After reaching final configurations A_{11}, A_{22}, B_{11}, B_{22}, C_{11}, and C_{22}, the algorithm computes a score for each one of them depending on whether the computer wins, the human wins, or the game is a draw. These scores are propagated upwards as follows:

- For a configuration b on a level labelled "computer", the highest score of the adjacent configurations in the next level is selected as the score for b. This is because the higher the score is, the better the outcome is for the computer.
- For a configuration b on a level labelled "human", the score of b is equal to the minimum score of the adjacent configurations in the next level, because the lower the score is, the better the outcome is for the human player.

The scores for the configurations in Figure 2 are the numbers in boldface. For example, for the configuration at the top of Figure 2, putting an 'O' in position 7 yields the configuration with the highest score (2), hence the computer will choose to play in position 7. Similarly, for configuration A in Figure 2, placing an 'X' in position 3 yields the configuration with the smallest score (2), so the human will choose to play in position 3.

We give below the algorithm for computing scores and for selecting the best available move. The algorithm is given in Java, but we have omitted variable declarations and some initialization steps. A full version of the algorithm can be found inside class `Play_nk_TTT.java`, which can be downloaded from the course's website.

```
private PosPlay computerPlay(char symbol, int highestScore, int lowestScore, int level) {
    if (level == 0) configurations = t.createDictionary();
    if (symbol == 'X') opponent = 'O'; else opponent = 'X';
    for(int row = 0; row < board_size; row++)
      for(int column = 0; column < board_size; column++)
        if(t.squareIsEmpty(row,column)) { // Empty position found
          t.storePlay(row,column,symbol);
          if (t.wins(symbol)||t.isDraw()||(level == max_level))
            reply = new PosPlay(t.evalBoard(),row,column);
          else {
(*)         lookupVal = t.repeatedConfig(configurations);
(*)           if (lookupVal != -1)
              reply = new PosPlay(lookupVal,row,column);
            }
            else {
              reply = computerPlay(opponent, highestScore, lowestScore, level + 1);
              t.insertConfig(configurations,reply.getScore());
            }
          }
          t.storePlay(row,column,' ');
          if((symbol == COMPUTER && reply.getVal() > value) || // A better play was found
            (symbol == HUMAN && reply.getVal() < value)) {
            bestRow = row; bestColumn = column;
            value = reply.getVal();
            if (symbol == COMPUTER && value > highestScore) highestScore = value;
            else if (symbol == HUMAN && value < lowestScore) lowestScore = value;
            if (highestScore >= lowestScore) /* alpha/beta cut */
              return new PosPlay(value, bestRow, bestColumn);
          }
        }
    return new PosPlay(value, bestRow, bestColumn);
}
```

The fist parameter of the algorithm is the symbol (either 'X' or 'O') of the player whose turn is next. The second and third parameters are the highest and lowest scores for the board positions that have been examined so far. The last parameter is used to bound the maximum number of levels of the game tree that the algorithm will consider. Since the number of configurations in the game tree could be very large, to speed up the algorithm the value of the last parameter specifies the highest level of the game tree that will be explored. Note that the smaller the value of this parameter is, the faster the algorithm will be, but the worse it will play.

Also note that if we bound the number of levels of the game tree, it might not be possible to determine the outcome of the game for some of the configurations in the lowest level of the tree. For example, if in the game tree of Figure 2 we set the maximum level to 2, then the algorithm will explore only levels 0, 1, and 2. At the bottom of the tree will appear configurations A_1, A_2, B_1, B_2, C_1, and C_2. Among these configurations, the scores for B_1 and C_2 are 0, as the human player wins in those cases; however, the scores for the remaining configurations are not known as in none of these configurations any player has won, and the configurations still include empty positions, so they do not denote game draws. In this case, configurations A_1, A_2, B_2, and C_1 will receive a score of `UNDECIDED` = 1.

2.1 Speeding-up the Algorithm with a Dictionary

The above algorithm includes several tests that allow it to reduce the number of configurations that need to be examined in the game tree. For this assignment, the most important test used to speed-up the program is the one marked (*). Every time that the score of a board configuration is computed, the configuration and its score are stored in a dictionary, that you will implement using a hash table. Then, when algorithm `computerPlay` is exploring the game tree trying to determine the computer's best move, before it expands a configuration b it will look it up in the dictionary. If b is in the dictionary, then its score is simply extracted from the dictionary, instead of exploring the part of the game tree below b.

For example, consider the game tree in Figure 3. The algorithm examines first the left branch of the game tree, including configuration D and all the configurations that appear below it. After exploring the configurations below D, the algorithm computes the score for D and then it stores D and its score in the dictionary. When later the algorithm explores the right branch of the game tree, configuration D will be found again, but this time its score is simply obtained from the dictionary instead of exploring all configurations below D, thus reducing the running time of the algorithm.

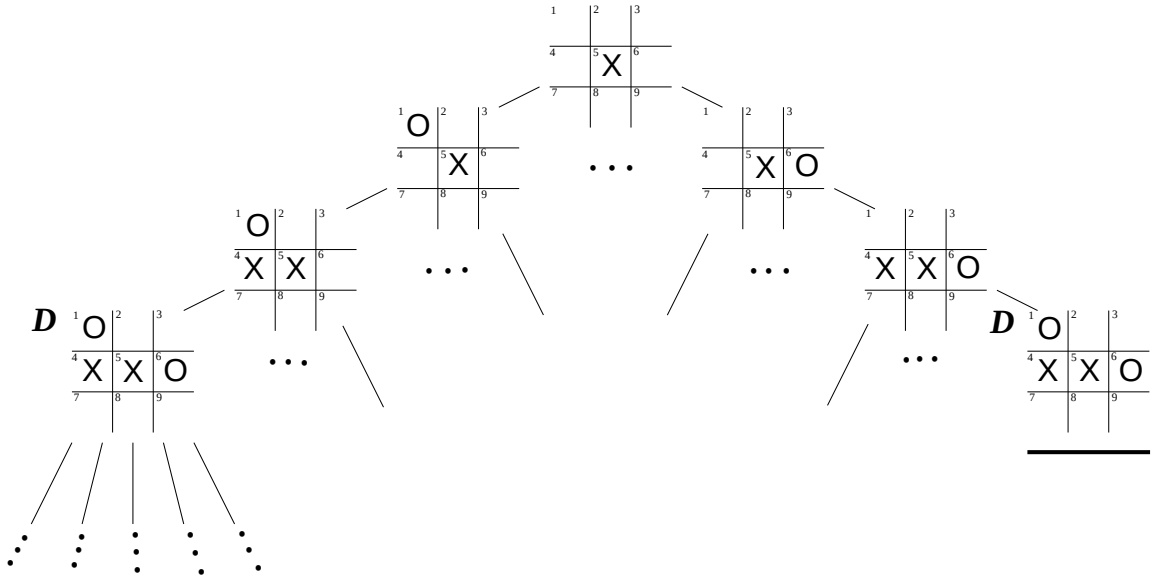

Figure 3: Detecting repeated configurations.

3 Classes to Implement

You are to implement at least 3 Java classes: `Dictionary.java`, `nk_TicTacToe.java`, and `Record.java`. You can implement more classes if you need to. You must write all the code yourself. You cannot use code from the textbook, the Internet, or any other sources. You **cannot** use the standard Java classes `Hashtable`, `HashMap`, `HashSet` or any other Java provided class that implements a hash table. You **cannot** use the `hashCode()` method.

3.1 Record

This class represents an entry in the dictionary, associating a configuration with its integer score. Each board configuration will be represented as a string as follows: concatenate all characters placed

in the board starting at the top left position and moving from left to right and from top to bottom. For example, for the configurations in Figure 1, their string representations are "`OXXXXOO␣␣`"[1], "`OXXXXOOOX`", and "`OXXXXOOXO`".

For this class, you must implement all and only the following `public` methods:

- `public Record(String config, int score)`: A constructor which returns a new `Record` with the specified configuration and score. The string `config` will be used as the key attribute for every `Record` object.
- `public String getConfig()`: Returns the configuration stored in this `Record`.
- `public int getScore()`: Returns the score in this `Record`.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

3.2 Dictionary

This class implements a dictionary. You must implement the dictionary using a hash table with separate chaining. You will decide on the size of the table, keeping in mind that the size of the table must be a prime number. A table of size between 6000-8000, should work well.

You must design your hash function so that it produces few collisions. (A bad hash function that induces many collisions will result in a lower mark.) Collisions must be resolved using separate chaining.

For this class, you must implement all the `public` methods in the following interface:

```
public interface DictionaryADT {
    public int insert (Record pair) throws DictionaryException;
    public void remove (String config) throws DictionaryException;
    public int get (String config);
    public int numElements();
}
```

The description of these methods follows:

- `public int insert(Record pair) throws DictionaryException`: Inserts the given `Record pair` in the dictionary. This method must throw a `DictionaryException` (see below) if `pair.getConfig()` is already in the dictionary.

  You are required to implement the dictionary using a hash table with separate chaining. To determine how good your design is, we will count the number of collisions produced by your hash function.

  Method `insert` must return the value 1 if the insertion of `pair` produces a collision, and it will return the value 0 otherwise. In other words, if for example your hash function is h(key) and the name of your table is T, this method will return the value 1 if $T[h($`pair.getConfig()`$)]$ already stores at least one element; it will return 0 if $T[h($`pair.getConfig()`$)]$ was empty before the insertion.

- `public void remove(String config) throws DictionaryException`: Removes the entry with the given `config` from the dictionary. Must throw a `DictionaryException` (see below) if the configuration is not in the dictionary.

[1] note that there are two blank spaces at the end of the string representing the two empty places in the board of Figure 1(a)

- `public int get(String config)`: A method which returns the score stored in the dictionary for the given configuration, or -1 if the configuration is not in the dictionary.
- `public int numElements()`: A method that returns the number of `Record` objects stored in the dictionary.

Since your `Dictionary` class must implement all the methods of the `DictionaryADT` interface, the declaration of your method should be as follows:

```
public class Dictionary implements DictionaryADT
```

You can download the file `DictionaryADT.java` from the course's website. The only other public method that you can implement in the `Dictionary` class is the constructor method, which must be declared as follows

```
public Dictionary(int size)
```

this returns an empty dictionary of the specified size.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

3.3 nk_TicTacToe

This class implements all the methods needed by algorithm `computerPlay`, which are described below. The constructor for this class must be declared as follows

```
public nk_TicTacToe (int board_size, int inline, int max_levels)
```

The first parameter specifies the size of the board, the second is the number of symbols in-line needed to win the game, and the third is the maximum level of the game tree that will be explored by the program. So, for example, to play the usual $(3, 3)$-tic-tac-toe, the first two parameters will have value 3.

This class must have an instance variable `gameBoard` of type `char[][]` to store the gameboard. This variable is initialized inside the above constructor method so that every entry of `gameBoard` stores a space ' '. Every entry of `gameBoard` stores one of the characters 'X', 'O', or ' '. This class must also implement the following public methods.

- `public Dictionary createDictionary()`: returns an empty `Dictionary` of the size that you have selected.
- `public int repeatedConfig(Dictionary configurations)`: This method first represents the content of `gameBoard` as a string as described above; then it checks whether the string representing the `gameBoard` is in the `configurations` dictionary: If it is in the dictionary this method returns its associated score, otherwise it returns the value -1.
- `public void insertConfig(Dictionary configurations, int score)`: This method first represents the content of `gameBoard` as a string as described above; then it inserts this string and `score` in the `configurations` dictionary.
- `public void storePlay(int row, int col, char symbol)`: Stores `symbol` in `gameBoard[row][col]`.
- `public boolean squareIsEmpty (int row, int col)`: Returns true if `gameBoard[row][col]` is ' '; otherwise it returns false.
- `public boolean wins (char symbol)`: Returns true if there are k adjacent occurrences of `symbol` in the same row, column, or diagonal of `gameBoard`, where k is the number of required symbols in-line needed to win the game.

- `public boolean isDraw()`: Returns true if `gameBoard` has no empty positions left and no player has won the game.
- `public int evalBoard()`: Returns one of the following values:
  - 3, if the computer has won, i.e. there are k adjacent 'O's in the same row, column, or diagonal of `gameBoard`;
  - 0, if the human player has won.
  - 2, if the game is a draw, i.e. there are no empty positions in `gameBoard` and no player has won.
  - 1, if the game is still undecided, i.e. there are still empty positions in `gameBoard` and no player has won.

4 Classes Provided

You can download classes `DictionaryADT.java`, `PosPlay.java`, `Play_nk_TTT.java` and `DictionaryException.java` from the course's website. Class `PosPlay` is an auxiliary class used by `Play_nk_TTT` to represent plays. Class `Play_nk_TTT` has the main method for your program, so the program will be executed by typing

```
java Play_nk_TTT size in_line max_levels
```

where `size` is the size of the gameboard, `in_line` is the number of symbols that need to be placed in-line to win the game, and `max_levels` is the maximum number of levels of the game tree that the program will explore.

Class `Play_nk_TTT` also contains methods for displaying the gameboard on the screen and for entering the moves of the human player.

5 Testing your Program

We will perform two kinds of tests on your program: (1) tests for your implementation of the dictionary, and (2) tests for your implementation of the program to play (n, k)-tic-tac-toe. For testing the dictionary we will run a test program called `TestDict` which verifies whether your dictionary works as specified. We will supply you with a copy of `TestDict` so you can use it to test your implementation.

6 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and white spaces should be used to improve readability.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared `private` (not `protected`), to maximize information hiding. Any access to these variables should be done with accessor methods (like `getScore()` for `Record`).

7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `Dictionary` tests pass: 4 marks.
- `nk_TicTacToe` tests pass: 4 marks.
- Coding style: 2 marks.
- `Record` and `Dictionary` implementation: 4 marks.
- `nk_TicTacToe` program implementation: 4 marks.

8 Handing In Your Program

You must submit an electronic copy of your program. To submit your program, login to OWL and submit your java files from there. Please **do not** put your code in sub-directories. **Do not** compress your files or submit a .zip, .rar, .gzip, or any other compressed file. Only your .java files should be submitted. Remember that the TA's will test your program on the computers of the Department.

When you submit your program, we will receive a copy of it with a datestamp and timestamp. If you re-submit your program after the due date, please send me an email message to ensure that the TA's mark your latest submission. We will take the last program submitted as the final version, and will deduct marks accordingly if it is late.

It is your responsibility to ensure that your assignment was received by the system.