Data Structures and Algorithms
Assignment 1
Submit Deadline: 5pm 16th April 2021

1. Problem: Hex Game

Hex is a strategy board game for two players, Red and Blue, played on a hexagonal grid, theoretically of any size and several possible shapes, but traditionally arranged in a shape of rhombus with 11×11 hexagonal cells. **Figure** 1 shows such a traditional Hex game board. The top and bottom edges are painted in red, called the red sides, and the left and right edges are the blue sides.

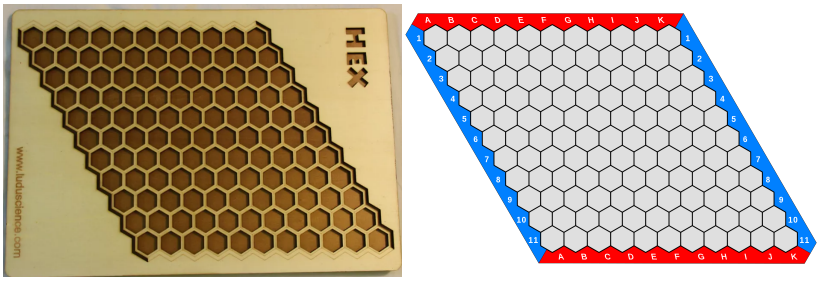

Figure 1: *Traditional Hex Game board.*

Players alternate placing red/blue markers (red markers for Red player and blue markers for Blue player) on unoccupied hexagonal spaces. Once a marker is placed, it remains until the game terminates. The goal for each player is to form a connected path of their own markers linking the opposing sides of the board marked by their colours[1]. The first player to complete his or her connection wins the game. Note that a path of a player must contain a sequence of consecutive adjacent markers in the player's colour. Since the spaces are in shape of hexagon, the spaces that contain two adjacent markers must share one edge. Figure 2 shows a case in which Blue player has a path links the blue sides whereas Red player does not have a connected path to link his sides therefore Blue player wins.

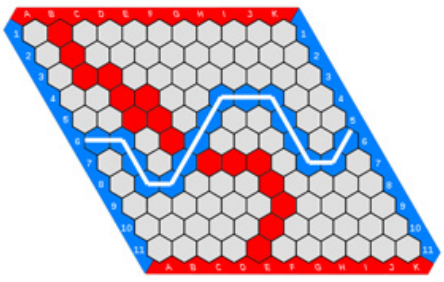

Figure 2: *Blue player has a connected path thus wins.*

[1] The four corner hexagons each belong to both adjacent sides.

The game is simple but has deep strategy, sharp tactics and a profound mathematical underpinning. It was invented in the 1940s independently by two mathematicians, Piet Hein and John Nash. Interestingly, it was mathematically proved that any Hex game must have a winner (there are no draws), even though it does not seem the case. See Wikipedia https://en.wikipedia.org/wiki/Hex_(board_game) for more details.

The task of this assignment is to write a computer program that allows human players or computer players to play a hex game on any size of m*m rhombus board where m>2. To make your program simple and also make the task unique[2], you are provided with a C++ code and you are restricted to write your code based on the given code. You may extend the base code but the game board must be in **pure text** mode as shown in Figure 3. **No graphic output** is allowed.

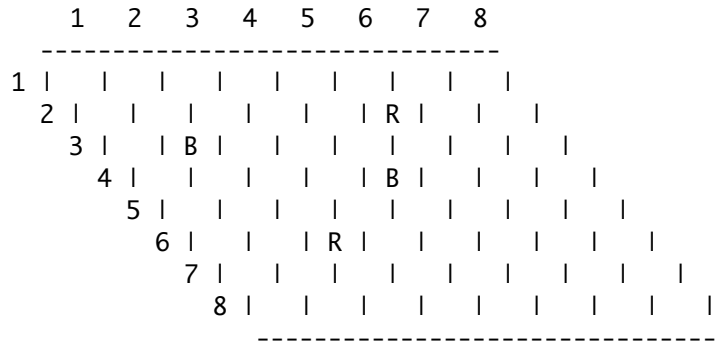

Figure 3: *An example layout of 8x8 Hex board and players' moves.*

Letter 'R' represents Red player's moves and 'B' Blue player's moves. In Figure 3, Red player made two moves (6, 3) and (2,6) while Blue player made moves (4,5) and (3,2). Note that although a Hex game board is in the shape of rhombus, we do not need to invent a specific data structure to represent a Hex game board. In the base code, we simply implement a game board as a two-dimensional array of integers, where '1' represents Red player's moves, '-1' for Blue player's moves and '0' represents blank cells. The only thing you need to concern is which cells can connect each other.

The following two figures show two examples of winning status. Figure 4 shows a case that the Red player wins and Figure 5 is an example of the Blue player wins.

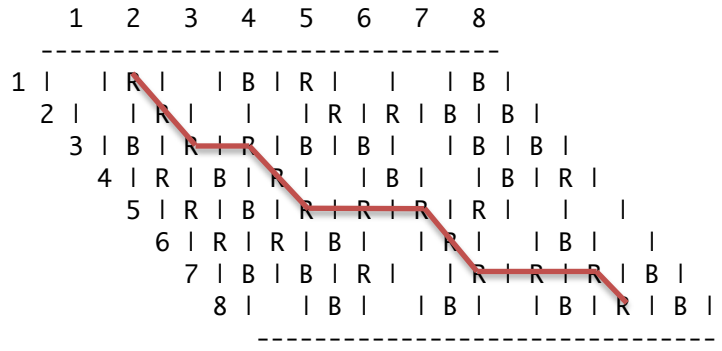

Figure 4: *An example of Red player wins.*

[2] There are many implementations of this game available in the Internet. Most of them are graphic based. You are not allowed to use any whole or part existing solution, no matter how it was implemented.

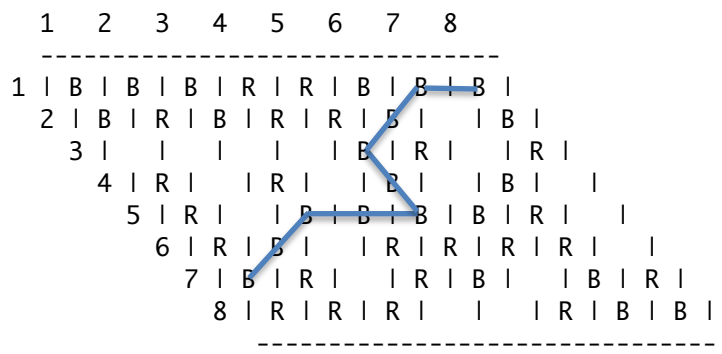

Figure 5: *An example of Blue player wins.*

The aims of this assignment are:

- Extend the base code to implement a game-playing platform on which human players and/or computer players can play Hex game against each other.
- Design computer players that can play Hex game smartly by applying the data structures and algorithms we learn from this unit.

The base code has implemented a simple game-playing platform for Hex game with the following functionalities:

1) Display a Hex game board in text format[3].
2) Accept players' moves and update game board accordingly.
3) Players take turns automatically.

The base code is written in C++. You are required to extend the existing code to implement further functionalities as specified in the next section. If you want to implement the platform in other language, such as Java or Python, you are required to literately translate the C++ code into other languages and further extend it. The game board must be text-based.

2. Task specification and requirements

You may claim marks for any of the following tasks you have completed but you are highly recommended to work in sequence.

Task 1 (10%): Download *HexGameBaseCode.zip*, which contains a simple C++ implementation of Hex game-playing platform. Add a new method to the Board class to check if the current board is full. Make appropriate changes on the program so that a game can terminate when the board is full.

Requirement and hints:

- Import the program into an IDE. Make necessary changes so that it is runnable in your IDE.
- The game board is represented as *n*n* two-dimensional array of integers with 1 for Red player, -1 for Blue player and 0 for empty cell.

Task 2 (20%) Implement a random player by extending the Board class with additional data members and functions as well as extending the abstract player class to a random

[3] Board size can be any but no less than 3.

player class.

Requirements and hints:

- Create a Random player class in which you override *getMove* function so that it can generate a valid random move each time it is called.
- Make your random player plays against the human player.
- Your mark is subject to the efficiency of your algorithm. Up-to 5 marks if you use the simplest way to generate random moves (repeatedly generate a row and col, and check if the move is valid).
- You are highly recommend to use *rand()* to generate random moves to make efficiency estimation easier. In general, the fewer calls of *rand()*, the better.

Task 3 (10%): Add a member function to the Board class (or elsewhere) to check if a player has a complete straight line.

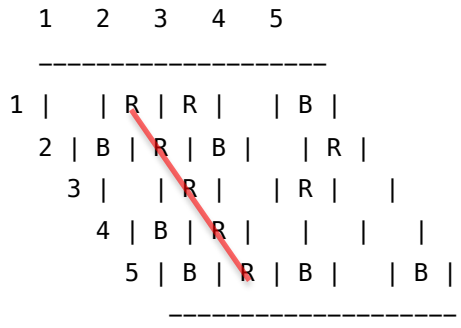

Figure 7: *An example of complete straight line of Red player.*

Requirements and hints:

- A complete straight line of a player is a complete path in which all moves in the same column (Red player) or in the same row (Blue player).
- This task is an intermediate step towards Task 5 to check winning paths of players. If you have a correct solution for Task 5, you will gain the marks of this task.
- Call the function in *HexGame* class to check if a player wins with a straight line.

Task 4 (10%) Add a member function in the Board class (or elsewhere) which stores all the neighbours (connected cells) that contain the moves of the same player for each given cell.

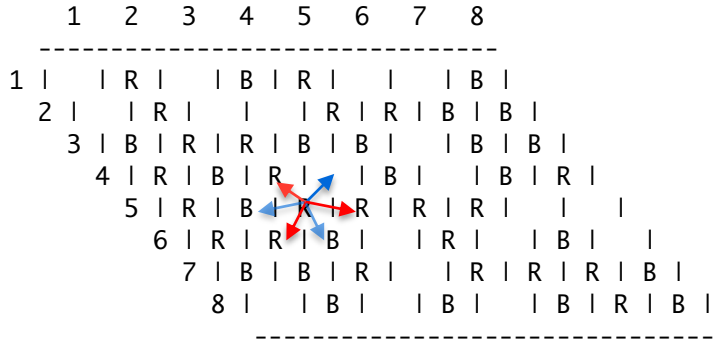

Figure 8: *The neighbours of a cell.*

Requirements and hints:

- This is also an intermediate step towards other tasks. If you have a back-tracking solution for Task 5, you will gain the marks of this task.
- For each cell, there are at most six neighbour cells (can be less if the cell is in a corner and on an edge): *down-left*, *down-right*, *left*, *right*, *up-left* and *up-right*. For instance, if the current cell is (*x,y*), the down-left cell is (*x+1,y-1*), the up-left is (*x-1,y*) and so on. Note that the rhombus shape of the board and the hexagonal shage of the markers matter, which determine which cells are connected.
- The prototype of the function may look like the following:

```
stack<int> Board::getNeighbours(int playerType, int x, int y)
```

  Given the cell (5,3) in Figure 8, the function should return a stack which contains the indices of *up-left*, *right* and *down-left* cells (the cells pointed by red arrows).
- You may invent your own representation for a cell, can either be an object of coordinates or the position index (*i*boardsize +j*), where *i* and *j* are the indices of the cell in the two-dimensional array.
- If you just want to pass the assignment by providing a perfect solution to Tasks 1-5 only, you can but your program must be runnable with a simplified game in which a player wins if the player has a winning straight line. Whenever a player makes a new move, your program must display all the neighbour cells that contain the player's moves to demonstrate your solution to Task 4.

Task 5 (20%): Add a member function to the Board class to check if a player wins, i.e., the player has *a* connected path to link its sides.

Requirements and hints:

- Your program must terminate a game whenever a player wins.
- You may create a few member functions for Board class or other classes to help you to identify all possible winning conditions, including the one for Task 4.
- You are highly recommended to use backtracking algorithm introduced in Lecture 3. Your marks will be determined by the quality of your code, the more generic and shorter your algorithm is, the more marks you will receive. If it is too hard for you to find a full solution, you may supply a partial solution for this task, which can recognize some of the winning situations but all. In this case, you will receive a portion of the total marks for this task.

Task 6 (15%): Implement a smart computer player so that it can win random player almost all the time.

Requirements and hints:

- Your computer player may not be as smart as human but should be able to win a random player in almost any situation.
- You may use any approach to implement your algorithm but the smarter your program is, the more marks you will receive.
- The simplest strategy to win a random player is to make a straight line (you will receive up to 5 marks in this case). You may also use Best-First Search to find the longest path to extend. Other path finder algorithms can also be implemented.

- Your smart player must be an extended class to the Player class.

Task 7 (15%): Implement a Monte Carlo computer player.

Requirements and hints:

- You must use recursive algorithm to implement a Monte Carlo player as demonstrated in Lecture 5.
- If you have implemented a Monte Carlo player for Task 5, redo Task 5 so that you can have two smarter computer players compete each other. Add some code to the main function so that computer players can run ten or more games automatically and show which player outperforms the other.

Tasks for Advanced students or students like more challenges

Task 8: Change your code to Java and embed it into the GGP game platform.

Requirements and hints:

- GGP platform is a game design platform implemented to facilitate General Game Playing Competitions (http://games.stanford.edu). The system can be downloaded from http://ggp.org.
- I will provide you with the rule description of Hex Game.
- I can give a separate tutorial on GGP platform and Monte Carlo approach to the students who are interested in taking the challenge.

3. Deliverables

3.1 Source code

Your program must be written in C++. You may convert the base code into Java or Python and extend it with the required functionalities. You can use any IDE to demonstrate your program provided it is available during your demonstration. You are allowed and highly recommended to demonstrate your program on your laptop. **All comments must be deleted from your source code when you demonstrate.** Your code must be purely written by yourself. No group work is allowed for the assignments of this unit. You may use any code I provided in the assignments, lectures, practicals or tutorials, but no part of your code is taken from any other resources unless specified in your declaration.

All students are required to submit a document contain the following

DECLARATION

I hereby certify that no other part of this submission has been copied from any other sources, including the Internet, books or other student's work except the ones I have listed below. No part of the code has been written/produced for me by another person or copied from any other source.

I hold a copy of this assignment that I can produce if the original is lost or damaged.

4. Submission

All source code and documentation should be submitted via vUWS before the deadline for documentation purpose. Your programs (.h, .cpp or .java) can be put in separate files (executable files are not required). All these files, including your declaration text file, should be zipped into one file **with your student id as the zipped file name**. Submission that does not follow the format will not be accepted. **Email submission is not acceptable (strict rule).**

4. Demonstration

You are required to demonstrate your program during **your scheduled** practical session in Week 8. Your tutor will check your code and your understanding of the code. **You will receive no marks if you fail the demonstration, especially if you miss the demo time.** Note that it is students' responsibility to get the appropriate compilers or IDEs to run their programs. **The feedback on your work will be delivered orally during the demonstration.** No further feedback or comments are given afterward.

The demonstrated program should be exactly the same as the one you submitted to vUWS. If you fail this assignment at your first demonstration, you are allowed to improve your work within the semester-break week (**maximal grade is pass in this case**).

Again, do not send us your work via email. Programming doesn't work in that way. Face-to-face is the most efficient way to check your work.

6. Additional information

The executable files of my solution will be downloadable from vUWS soon (no guarantee runnable in your machine). I will also demonstrate my solution for each level during PASS sessions. Additional training will be given as tasks in the practical sessions. Make sure you can do these tasks with the help of your tutor. A discussion forum will be created in vUWS to encourage you learn from each other. Note that we encourage students to learn from each other, but work has to be done individually.

Detailed marking criteria will be specified in a separate document.

Have fun!