# Assignment 4

COMP 250 - Winter 2018

Due date : April 13th, 2018

## 1 General Instructions (read carefully)

- You are provided some starter code that you should fill in as requested. **Add your code only where you are instructed to do so**. You can add some helper methods. Do not modify the code in any other way and in particular : do not change the methods or constructors that are already given to you, do not import extra code and do not touch the method headers. **Any failure to comply with these rules will give you an automatic 0**.

- **Submission instructions**

  - Assignment is due on April 13th 11:59 PM.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times but only the latest submission will be evaluated. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).
  - **Be careful, you have two submissions to do for this assignment !** First submit only the file `Graph.java` to the corresponding folder on myCouses. Finally, submit a .zip file containing only the files `Configuration.java` and `Game.java` to the other folder on myCourses.
  - **Submitting the wrong format of file or submitting in the wrong folder, or wrong file name (e.g. Graph (1).java) on myCourses will be an automatic 0** to the corresponding exercise.
  - Around one day before due date : we will run whatever you had submitted last and give you some feedback on it, mostly to warn

you if you currently have a grade of zero, so you can fix it if there is a problem. Take advantage of this by submitting early !

  - Morning after due date : we will run your last submission and warn you again if your code does not compile.
  - After this, you can resubmit at any moment for a period of three days after the initial due date, with a penalty of 20% per day.
  - The last acceptable file you submit will be used to determine your final grade for this assignment.

- The starter code includes a tester class. If your code fails those tests, it means that there is a mistake somewhere. **Even if your code passes those tests, it may still contain some errors**. We will test your code on a more challenging set of examples. We therefore highly encourage you to modify that tester class, expand it and share it with other students on the myCourses discussion board. Do not include it in your submission.

- You will automatically get 0 if your code does not compile.

- **Failure to comply with any of those rules will be penalized.** If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on myCourses.

# 2 Introduction

There are two exercises in this assignment. Be very careful about what files you submit on myCourses and where you upload them.

You will be less constrained for this assignment than for the previous ones. For many of the questions, there are more than one solution possible. You can write as many auxiliary methods as you want. However, you are not allowed to import any additional code (and you don't need it anyways).

The testers you are provided with are very partial. We have said it for every assignment, but let us repeat it for this one too : you need to expand this tester.

Each question has a weight associated to it. This is only to serve as an indicator and may be later changed without notice.

# 3 Exercise 1 : Graphs (28 points)

## 3.1 Structure of the code provided to you

A `Graph` has a number of nodes `nbNodes` and is characterized by its adjacency matrix `adjacency`. `adjacency[i][j]` states whether or not there is an edge from the i-th node to the j-th node.

The Graphs you will be dealing with are undirected, i.e. `adjacency[i][j] == adjacency[j][i]` is always true. They may contain self-loops and they may be non-connected.

## 3.2 Questions

**Question 1. (2 points each)** Code `addEdge` and `removeEdge`. It takes as input a pair of nodes and adds / removes the edge between the i-th and the j-th nodes.

**Question 2. (4 points)** Code `nbEdges`. `g.nbEdges()` returns the number of edges of `g`.

**Question 3. (10 points)** Code `cycle`. It takes as input an integer, `g.cycle(i)` returns true if the i-th node is part of a cycle (and false otherwise).

**Question 4. (10 points)** Code `shortestPath`. It takes as input a pair of integers, `g.shortestPath(i,j)` returns the length of the shortest path joining the i-th node to the j-h node. If there is no such path, it returns `g.nbNodes +1`.

# 4 Exercise 2 : Connect 4 (72 points)

I hope you have learnt things that are of interest to you during this class and in particular throughout the assignments. We are going to finish this final assignment by a small game that I used to play a lot with my sister when we were younger : Connect 4 (Puissance 4 pour les francophones).

For those of you who don't know about this game, I refer you to the wikipedia page for more details but let me sum up how the game works : you have a vertical grid that is 7-cell-long and 6-cell-high. Each player has some disks of a specific color (yellow for one player, red for the other one). At each round, the player whose turn it is picks a column and puts a disk in that column. The disk falls to the lowest available cell. After one player has played, it is the other player's turn.

The game can end in two cases. First, if one player has won, i.e. he is the first one to have four disks of his colour aligned (either horizontally, vertically or diagonally). Second, if there is no space left in the grid, in which case there is no winner.

## 4.1 Structure of the code that is provided to you

You have two classes provided to you : `Configuration` and `Game`.

The class `Configuration` corresponds to the grid. It has three fields. First `board` is the grid per say. It is initiated in the constructor to an array `int[7][6]` (containing only 0's by default). The field `available` is an array `int[7]` such that `available[i]` is the first available row in the i-th column. If the i-th column is already full, its value is 6. Finally the field `spaceLeft` is true if you still have some place somewhere on the grid to put a disk.

The class `Configuration` already has a method `print()` coded for you. It prints the grid, the disks that are contained in it and the number of the columns. For instance, it could print :

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   | 1 |   |   |
|   |   |   | 1 | 2 |   |   |
|   |   | 1 | 2 | 1 |   |   |
| 1 | 2 | 2 | 2 | 1 | 1 | 2 |
```

The row at the bottom is the 0-th row. So for instance the disk of the 2nd player in the 6th column corresponds to `board[6][0]` and the disks of the

1st player that are in column 4 correspond to `board[4][0]`, `board[4][1]` and `board[4][3]`.

The class `Game` corresponds to the game per say. It does not have any fields. You will have several methods to code in it. However, you are already provided with the method `play` that you can use at the end.

At the end of the tester, there are two lines that are commented. Once you have finished coding every methods, you can uncomment these two lines, these will allow you to play against the computer. It is a very good way of testing further your code.

## 4.2 Questions in Configuration

**Question 5. (8 points)** First code `addDisk` in `Configuration`. As the name suggests, `c.addDisk(5,1)` adds a disk corresponding to player 1 in the column 5 and at the first row available. `addDisk` takes as input the number of the column in which to add a disk and the number of the player whose disk is to be added. Here you can assume that the column has space left.

For instance, the grid showed in previous subsection would be updated to

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   | 1 |   |   |
|   |   |   | 1 | 2 |   |   |
|   |   | 1 | 2 | 1 | 1 |   |
| 1 | 2 | 2 | 2 | 1 | 1 | 2 |
```

Don't forget to update the other fields.

**Question 6. (16 points)** You can now code `isWinning`. It takes as argument the last column that was played and the player who played last (so you know that the disk that is last in that column belongs to said player) and it returns true if, and only if, the player managed to make a line of four disk in the last round. The line can be horizontal, vertical or diagonal.

For example, consider a configuration `c` with the following `board` :

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   | 1 | 2 |   |
|   |   |   | 1 | 2 | 1 |   |
|   |   | 1 | 2 | 1 | 1 |   |
| 1 | 2 | 2 | 2 | 1 | 1 | 2 |
```

`c.isWinning(5,2)` should return `true`, but `c.isWinning(3,1)` should return `false`.

**Question 7. (8 points)** Code `canWinNextRound`. It takes as argument the player `p` whose turn it is and returns `i` such that if player `p` places its disk in column `i`, he wins the game. If there are no such column, `i` is equal to -1. If there are two such columns, it returns the smallest one.

For example, consider a configuration `c` with the following `board` :

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   | 1 |   |   |
|   |   |   | 1 | 2 | 2 |   |
|   |   | 1 | 2 | 1 | 1 |   |
| 1 | 2 | 2 | 2 | 1 | 1 | 2 |
```

`c.canWinNextRound(2)` should return `5` but `c.canWinNextRound(1)` should return `-1`.

**Question 8. (16 points)** Code `canWinTwoTurns`. It takes as argument the player `p` whose turn it is and returns `i` such that if player `p` places its disk in column `i`, whatever the other player does on next round, player `p` can win when it's his turn again. Note that this requires that the other player does not win in between.

You can assume here that player `p` has no winning move at this turn.

For instance, consider a configuration `c` with the following `board` :

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
| 2 |   | 2 | 2 |   |   |   |
| 1 |   | 1 | 2 |   |   |   |
| 1 |   | 2 | 1 | 1 |   |   |
| 1 | 1 | 2 | 2 | 1 |   |   |
```

`c.canWinTwoTurns(1)` should return `-1` because player 1 has no way of making sure he is going to win in two rounds. `c.canWinTwoTurns(2)` should return `1` because if player 2 plays column 1, then there are two cases : either player 1 plays also in column 1 in which case by playing (again) in column 1, player 2 will complete a horizontal line, or player 1 does not play in column 1, in which case player 2 can play again in column 1 and complete a diagonal line.

## 4.3 Questions in Game

**Question 9. (10 points)** Code `getNextMove`. This method asks in which column the player wants to add his disk. It takes as argument a `BufferedReader` (where the player is writing his answer), a `Configuration` (the grid on which the player is playing) and an `integer` (the number of the player whose turn it is). You are allowed to print whatever you want on the screen (the grid, a warning if the other player has a winning move that should be prevented etc). But what is important is that if the player asks for a column with no space left, you should keep asking him until he gives a valid one.

Consider a configuration `c` with the following `board` :

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|   |   |   | 1 | 2 |   |   |
|   |   | 1 | 2 | 1 | 1 |   |
|   |   | 2 | 2 | 1 | 1 | 2 |
|   |   | 1 | 1 | 2 | 2 | 1 |
|   |   | 2 | 2 | 1 | 1 | 2 |
|   |   | 1 | 2 | 2 | 1 | 2 |
```

Consider a `BufferedReader keyboard`, then `getNextMove(keyboard, c, 2)` should ask player 2 for his move. If player 2 requests column 3, you should ask again. If player 2 then requests column 4, ask again. If player 2 then

requests column 3 (again), ask (again). If then player 2 requests column 2, return 2.

To test it, you should uncomment the corresponding lines in the tester.

**Question 10. (14 points)** Code `movePlayer1`. Player 1 is played by the computer. It takes as input an `integer` which is the column that the other player played last and a `Configuration` which is the current state of the grid. Here is the strategy that the player 1 is following :

1. if player 1 has a winning move, he plays that winning move.
2. if player 1 has a move that can make him win in two rounds, he plays that move.
3. if none of these are possible, player 1 tries to put his disk right above the one player 2 played last (call it `last`). If there is no more space left in column `last`, he tries the column on the left (`last -1`), then on the right (`last +1`). If that is still not possible, he plays the columns `last -2` then `last +2` etc. Player 1 skips every step that would correspond to a non-existing column. For instance, if `last` = 5, he would try the columns in the following order : 5, 4, 6, 3, 2, 1, 0.

You have just coded an AI ! It is not a very clever AI, but it is one nonetheless. You can now play against this AI `\o/`