Seneca College of Applied Arts & Technology **February 11, 2019**

SCHOOL OF INFORMATION & COMMUNICATION TECHNOLOGY

JAC444 **Due date: March 7, 2019**

Workshop 6

Description[1]:
This assignment lets you practice various concepts including (more in-depth) Swing/JavaFX GUI.

In this assignment, you will be implementing the Knight's Tour problem. Can the knight piece move around an empty chessboard and touch each of the 64 squares once and only once?

The knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). Thus, as shown in the following figure, from a square near the middle of an empty chessboard, the knight (labeled K) can make eight different moves (numbered 0 through 7).

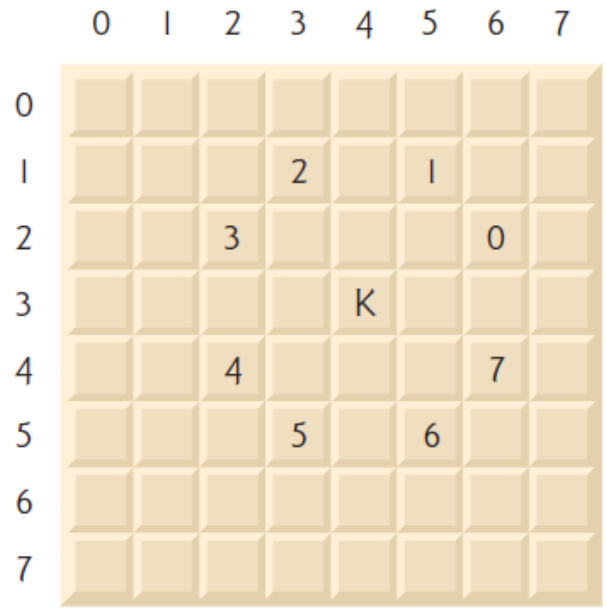

Figure 1- The eight possible moves of the knight.

a) Draw an eight-by-eight chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a 1 in the starting square, a 2 in the second square, a 3 in the third, and so on. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

[1] From: Deitel and Deitel, *Java, How to Program, 10th Edition*, Prentice-Hall, 2014. ISBN 0-13-380780-0

b) Now let's develop an application that will move the knight around a chessboard. The board is represented by an eight-by-eight two-dimensional array *board*. Each square is initialized to zero. We describe each of the eight possible moves in terms of its horizontal and vertical components. For example, a move of type 0, as shown in the figure, consists of moving two squares horizontally to the right and one square vertically upward. A move of type 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, *horizontal* and *vertical*, as follows:

```
horizontal[ 0 ] = 2      vertical[ 0 ] = -1

horizontal[ 1 ] = 1      vertical[ 1 ] = -2

horizontal[ 2 ] = -1     vertical[ 2 ] = -2

horizontal[ 3 ] = -2     vertical[ 3 ] = -1

horizontal[ 4 ] = -2     vertical[ 4 ] = 1

horizontal[ 5 ] = -1     vertical[ 5 ] = 2

horizontal[ 6 ] = 1      vertical[ 6 ] = 2

horizontal[ 7 ] = 2      vertical[ 7 ] = 1
```

Let the variables *currentRow* and *currentColumn* indicate the row and column, respectively, of the knight's current position. To make a move of type *moveNumber*, where *moveNumber* is between 0 and 7, your application should use the statements:

```
currentRow += vertical[ moveNumber ];

currentColumn += horizontal[ moveNumber ];
```

Write an application to move the knight around the chessboard. Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Test each potential move to see if the knight has already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Run the application. How many moves did the knight make?

c) After attempting to write and run a Knight's Tour application, you've probably developed some valuable insights. We'll use these insights to develop a *heuristic* (i.e., a common-sense rule) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so that when the board gets congested near the end of the tour, there will be a greater chance of success.

We could develop an "accessibility heuristic" by classifying each of the squares according to how accessible it is and always moving the knight (using the knight's L-shaped moves) to the most inaccessible square. We label a two-dimensional array *accessibility* with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each of the 16 squares nearest the center is rated as 8, each corner square is rated as 2, and the other squares have accessibility numbers of 3, 4 or 6 as follows:

```
2  3  4  4  4  4  3  2
3  4  6  6  6  6  4  3
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
3  4  6  6  6  6  4  3
2  3  4  4  4  4  3  2
```

Write a new version of the Knight's Tour, using the accessibility heuristic. This time, the knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [*Note:* As the knight moves around the chessboard, your application should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.]

Run this version of your application. Did you get a full tour? Modify the application to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

d) ***(If you'd like to improve your solution,)*** you could write a version of the Knight's Tour application that, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your application should move to the tied square for which the next move would arrive at a square with the lowest accessibility number.

e) ***(If you'd like to explore more,)*** in part (c), we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently. As computers continue to increase in power, we'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let's call this approach "brute-force" problem solving. Now this time:

e-1) Use random-number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves) at random. Your application should run one tour and display the final chessboard. How far did the knight get?

e-2) Most likely, the application in part (e-1) produced a relatively short tour. Now modify your application to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your application finishes attempting the 1000 tours, it should display this information in neat tabular format. What was the best result?

e-3) Most likely, the application in part (e-2) gave you some "respectable" tours, but no full tours. Now let your application run until it produces a full tour. [*Caution:* This version of the application could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length, and display this table when the first full tour is found. How many tours did your application attempt before producing a full tour? How much time did it take?

e-4) Compare the brute-force version of the Knight's Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute-force approach? Argue the pros and cons of brute-force problem solving in general.

f) ***(And if you'd like to explore even more,)*** as you already know, in the Knight's Tour, a full tour occurs when the knight makes 64 moves, touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the application you wrote to test for a closed tour if a full tour has occurred.

Marking Criteria and Tasks:
Please note that you should:

a- have appropriate indentation.
b- have proper file structures and modularization.
c- follow Java naming conventions.
d- document all the classes properly.
e- not have debug/useless code and/or file(s) left in assignment.
f- have good intra and/or inter class designs.

in your code!

- Task: Produce a graphical version of the Knight's Tour problem (parts b and c.) As each move is made, the appropriate cell of the chessboard should be updated with the proper move number. If the result of the program is a *full tour* or a *closed tour*, the program should display an appropriate message. Developing and running the desired solution correctly, using Swing/JavaFX GUI: **5 marks**.

Deliverables and Important Notes:
You are supposed to show up AND hand in your solution in person (run the solution and/or answer related Qs) in lab 8.

In case you don't show up OR hand in/run the required task in the lab, you could submit your final solution (described below) on the same due date but note that there would be a 50% penalty! Late submissions would result in additional 10% penalties for each day or part of it.

In this case, you should zip *only the Java files* to a file named after your Last Name followed by the first 3 digits of your student ID. For example, if your last name is **`Savage`** and your ID is **`354874345`** then the file should be named **`Savage354.zip`**. Finally email your zip file to me at reza.khojasteh@senecacollege.ca

Remember that you are encouraged to talk to each other, to the instructor, or to anyone else about any of the assignments, but the final solution may not be copied from any source.