Coursework 7 (Scala, Knight's Tour)

This coursework is about searching and backtracking, and worth 10%. The first part is due on 23 November at 11pm; the second, more advanced part, is due on 30 November at 11pm. You are asked to implement Scala programs that solve various versions of the *Knight's Tour Problem* on a chessboard. Make sure the files you submit can be processed by just calling `scala <<filename.scala>>`.

Disclaimer

It should be understood that the work you submit represents your own effort. You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Background

The *Knight's Tour Problem* is about finding a tour such that the knight visits every field on an $n \times n$ chessboard once. For example on a 5×5 chessboard, a knight's tour is:

4	24	11	6	17	0
3	19	16	23	12	7
2	10	5	18	1	22
1	15	20	3	8	13
0	4	9	14	21	2
	0	1	2	3	4

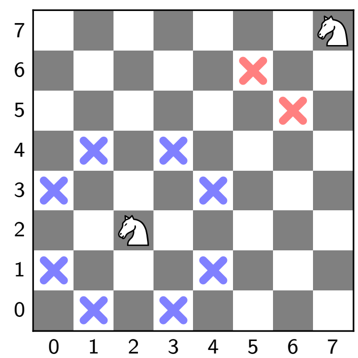
The tour starts in the right-upper corner, then moves to field $(3,2)$, then $(4,0)$ and so on. There are no knight's tours on 2×2, 3×3 and 4×4 chessboards, but for every bigger board there is.

A knight's tour is called *closed*, if the last step in the tour is within a knight's move to the beginning of the tour. So the above knight's tour is <u>not</u> closed (it is open) because the last step on field $(0,4)$ is not within the reach of the first step on $(4,4)$. It turns out there is no closed knight's tour on a 5×5 board. But there are on a 6×6 board and bigger, for example

10	5	18	25	16	7
31	26	9	6	19	24
4	11	30	17	8	15
29	32	27	0	23	20
12	3	34	21	14	1
33	28	13	2	35	22

where the 35th move can join up again with the 0th move.

If you cannot remember how a knight moves in chess, or never played chess, below are all potential moves indicated for two knights, one on field $(2, 2)$ (blue moves) and another on $(7, 7)$ (red moves):



Part 1 (6 Marks)

You are asked to implement the knight's tour problem such that the dimension of the board can be changed. Therefore most functions will take the dimension as an argument. The fun with this problem is that even for small chessbord dimensions it has already an incredably large search space—finding a tour is like finding a needle in a haystack. In the first task we want to see far we get with exhaustively exploring the complete search space for small chessboards.

Let us first fix the basic datastructures for the implementation. The board dimension is an integer (we will never go boyond board sizes of 100×100). A *position* (or field) on the chessboard is a pair of integers, like $(0, 0)$. A *path* is a list of positions. The first (or 0th move) in a path is the last element in this list; and the last move in the path is the first element. For example the path for the 5×5 chessboard above is represented by

$$\texttt{List(}\underbrace{\texttt{(0, 4)}}_{24}\texttt{, }\underbrace{\texttt{(2, 3)}}_{23}\texttt{, ..., }\underbrace{\texttt{(3, 2)}}_{1}\texttt{, }\underbrace{\texttt{(4, 4)}}_{0}\texttt{)}$$

Suppose the dimension of a chessboard is n, then a path is a *tour* if the length of the path is $n \times n$, each element occurs only once in the path, and each move follows the rules of how a knight moves (see above for the rules).

Tasks (file knight1.scala)

(1a) Implement a is-legal-move function that takes a dimension, a path and a position as argument and tests whether the position is inside the board and not yet element in the path. [1 Mark]

(1b) Implement a legal-moves function that calculates for a position all legal onward moves. If the onward moves are placed on a circle, you should produce them starting from "12-oclock" following in clockwise order. For example on an 8×8 board for a knight on position $(2,2)$ and otherwise empty board, the legal-moves function should produce the onward positions

```
List((3,4), (4,3), (4,1), (3,0), (1,0), (0,1), (0,3), (1,4))
```

in this order. If the board is not empty, then maybe some of the moves need to be filtered out from this list. For a knight on field $(7,7)$ and an empty board, the legal moves are

```
List((6,5), (5,6))
```

[1 Mark]

(1c) Implement two recursive functions (count-tours and enum-tours). They each take a dimension and a path as arguments. They exhaustively search for **open** tours starting from the given path. The first function counts all possible open tours (there can be none for certain board sizes) and the second collects all open tours in a list of paths. [2 Marks]

Test data: For the marking, the functions in (1c) will be called with board sizes up to 5×5. If you only search for open tours on 5×5 board starting from field $(0,0)$, there are 304 of them. If you try out every field of a 5×5-board as a starting field and add up all open tours, you obtain 1728. A 6×6 board is already too large to be searched exhaustively.[1]

Tasks (file knight2.scala)

(2a) Implement a first-function. This function takes a list of positions and a function f as arguments. The function f takes a position as argument and produces an optional path. The idea behind the first-function is as follows:

$$
\begin{aligned}
\mathit{first}(\texttt{Nil}, f) &\stackrel{\text{def}}{=} \texttt{None} \\
\mathit{first}(x :: xs, f) &\stackrel{\text{def}}{=} \begin{cases} f(x) & \textit{if } f(x) \neq \texttt{None} \\ \mathit{first}(xs, f) & \textit{otherwise} \end{cases}
\end{aligned}
$$

That is, we want to find the first position where the result of f is not `None`. [1 Mark]

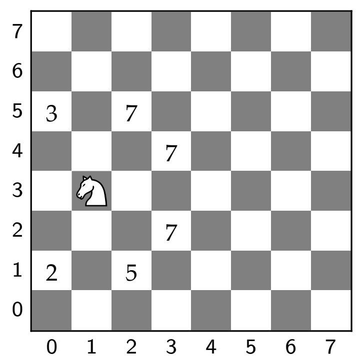
[1] For your interest, the number of open tours on 6×6, 7×7 and 8×8 are 6637920, 165575218320, 19591828170979904, respectively.

(2b) Implement a first-tour function. Using the first-function from (2a), search recursively for an open tour. As there might not be such a tour at all, the first-tour function needs to return an `Option[Path]`. [2 Marks]

Testing The first tour function will be called with board sizes of up to 8×8.

Part 2 (4 Marks)

As you should have seen in Part 1, a naive search for open tours beyond 8×8 boards and also for searching for closed tours takes too much time. There is a heuristic (called Warnsdorf's rule) that can speed up finding a tour. This heuristice states that a knight is moved so that it always proceeds to the square from which the knight will have the <u>fewest</u> onward moves. For example for a knight on field $(1, 3)$, the field $(0, 1)$ has the fewest possible onward moves, namely 2.



Warnsdorf's rule states that the moves on the board above sould be tried out in the order

$$(0, 1), (0, 5), (2, 1), (2, 5), (3, 4), (3, 2)$$

Whenever there are ties, the correspoding onward moves can be in any order. When calculating the number of onward moves for each field, we do not count moves that revisit any field already visited.

Tasks (file knight3.scala)

(3a) orderered-moves

(3b) first-closed tour heuristics; up to 6×6

(3c) first tour heuristics; up to 50×50