

Coursework 7 (Scala, Knight's Tour)

This coursework is worth 10%. It is about searching and backtracking. The first part is due on 23 November at 11pm; the second, more advanced part, is due on 21 December at 11pm. You are asked to implement Scala programs that solve various versions of the *Knight's Tour Problem* on a chessboard. Note the second part might include material you have not yet seen in the first two lectures.

Important:

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the commandline.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot use `ListBuffers`, for example.
- Do not use `return` in your code! It has a different meaning in Scala, than in Java.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each part will be restricted to a maximum of 360 seconds on my laptop: If you calculate a result once, try to avoid to calculate the result again. Feel free to copy any code you need from files `knight1.scala`, `knight2.scala` and `knight3.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 |

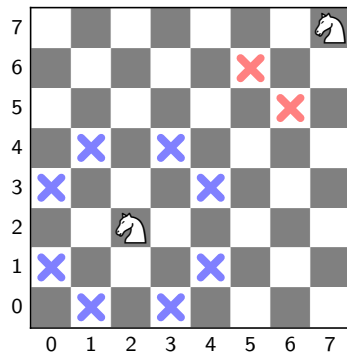
The tour starts in the right-upper corner, then moves to field (3,2), then (4,0) and so on. There are no knight's tours on 2×2 , 3×3 and 4×4 chessboards, but for every bigger board there is.

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 not closed 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 on bigger ones, 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 (7 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 of the board as an argument. The fun with this problem is that even for small chessboard dimensions it has already an incredibly large search space—finding a tour is like finding a needle in a haystack. In the first task we want to see how 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 beyond board sizes of 40×40). 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

$$\text{List}(\underbrace{(0, 4)}_{24}, \underbrace{(2, 3)}_{23}, \dots, \underbrace{(3, 2)}_1, \underbrace{(4, 4)}_0)$$

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 an `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-o'clock" 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 in this order:

$$\text{List}((3,4), (4,3), (4,1), (3,0), (1,0), (0,1), (0,3), (1,4))$$

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

$$\text{List}((6,5), (5,6))$$

[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 tours starting from the given path. The first function counts all possible tours (there can be none for certain board sizes) and the second collects all 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 search for tours on a 5×5 board starting only from field $(0, 0)$, there are 304 of tours. If you try out every field of a 5×5 -board as a

starting field and add up all 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. So f 's type is `Pos => Option[Path]`. The idea behind the first-function is as follows:

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

That is, we want to find the first position where the result of f is not `None`, if there is one. Note that you do not (need to) know anything about the function f except its type, namely `Pos => Option[Path]`. There is one additional point however you should take into account when implementing `first`: you will need to calculate what the result of $f(x)$ is; your code should do this only **once!**

[1 Mark]

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

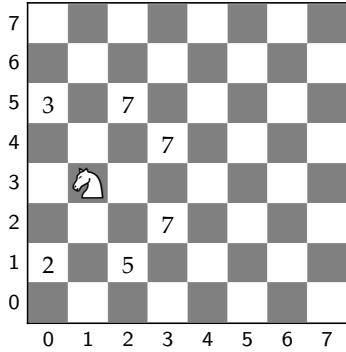
[2 Marks]

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

Part 2 (3 Marks)

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

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



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

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

Whenever there are ties, the corresponding 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) Write a function `ordered-moves` that calculates a list of onward moves like in (1b) but orders them according to the Warnsdorf's rule. That means moves with the fewest legal onward moves should come first (in order to be tried out first). [1 Mark]
- (3b) Implement a `first-closed-tour-heuristic` function that searches for a **closed** tour on a 6×6 board. It should use the `first-function` from (2a) and tries out onward moves according to the `ordered-moves` function from (3a). It is more likely to find a solution when started in the middle of the board (that is position $(dimension/2, dimension/2)$). [1 Mark]
- (3c) Implement a `first-tour-heuristic` function for boards up to 40×40 . It is the same function as in (3b) but searches for tours (not just closed tours). You have to be careful to write a tail-recursive version of the `first-tour-heuristic` function otherwise you will get problems with stack-overflows. [1 Mark]