Lab 4 – Instructions

This lab is about working with graphs, specifically using an algorithm to find a particular kind of path in a graph to solve a couple of puzzles.

Logistics

The logistics of this lab are as before:

1. Download the skeleton project and unpack it.

2. cd into its top-level directory, `edaa40lab4`. Start the Leiningen REPL there.

3. Have a look at the file

```
edaa40lab4/src/edaa40/lab4.clj
```

This is the file you are supposed to edit. As in previous labs, it contains commented-out incomplete code skeletons for you to provide implementations in. Also as before, it is a good idea to do this one step at a time, top to bottom, while making sure one set of tests pass before you proceed to the next part.

The package can be loaded, as you would expect, using:

```
(use 'edaa40.lab4 :reload)
```

As you implement the various functions, **do not worry about efficiency**. There is an entire course devoted to that topic, for now let’s focus on producing something that works, even if it works very slowly.

1. Square sum problem and Hamiltonian paths

Start by watching the following video (a clickable link is also on the course page):

https://www.youtube.com/watch?v=G1m7goLCJDY

In part A and B of this lab, we solve the square sum problem in two steps:

A) Create the graph representing the square sum relation

This part consists of implementing the function `create-square-sum-relation`. It's a very short function (roughly a single line of code, depending on formatting, of course), just a little warm-up.

B) Implementing the algorithm searching for a Hamiltonian path in such a graph.

The next part is about implementing the actual path search. The function we want to call is `H`, and it has two arguments: `V`, a set of vertices, and `E`, an edge relation over `V` (i.e. a set of two-element vectors, like all the other relations we have worked with so far).

As you see from the code, `H` uses a helper function, `H'`. That's the one you are supposed to implement. `H'` has four arguments:

1. `E`, the edge relation, in other words: the arrows of our graph,
2. `a`, the "current" vertex,
3. `S`, the set of vertices we haven't visited yet, and
4. `P`, the partial path we have traveled so far.

That `:pre` and `:post` business after the argument list is a "condition map" containing pre- and postconditions for this function.[1] (Preconditions and postconditions together are also called "assertions".) Those are fancy names for boolean expressions that must be true at the beginning (preconditions) or the end (postconditions) of the execution of the function body, whenever it is called. Just uncomment them with the rest of the skeleton and leave them there, they can help you detect errors. As they also take some time to test, you can turn off testing when you want to run the code on larger graphs. You do that by typing

```
(set! *assert* false)
```

into the REPL, and you can turn assertion testing back on with, you guessed it,

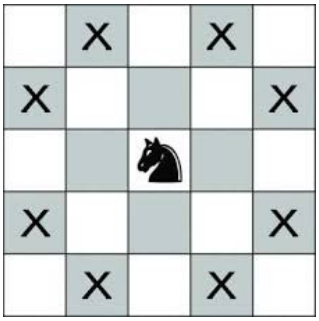
```
(set! *assert* true)
```

It might help you with implementing `H'` to understand those assertions, and also how `H'` is called initially from `H`. In particular, have a look at the documentation for the Clojure function `some` and how it is used in `H` to "try" the different vertices in `V` as starting points for the path. It will be useful in `H'`, too. Note that it returns `nil` when the function it evaluates returns `nil` for all values in the set. Otherwise, it returns the first non-`nil` (and non-`false`) value it gets from applying the predicate.

NOTE: `H'` also does not require an awful lot of code. My version's body consists, fluffily formatted, of four lines. If you find yourself writing loops and complex if/cond/case constructions, you might want to look for a simpler solution. It's out there! :-)

1 Take a look at https://clojure.org/reference/special_forms for more information on this.

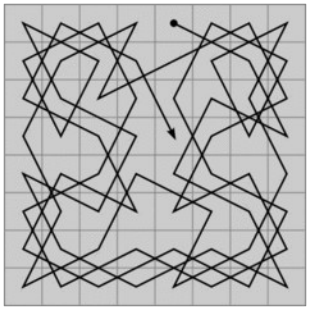
2. Knight's tours



In the game of chess, the knight is the horse-shaped piece, and it moves either one square up or down and two to the right or left, or two squares up or down, and one to the right or left.

Suppose you have an n by m grid of squares, e.g. with 8 by 8 it would be a chess board. A *knight's tour* is a sequence of knight's moves, starting from anywhere on the grid, that touches each square exactly once. Depending on the dimensions of the grid, the number of different knight's tours can vary widely, and for some grids there aren't any.

C) Using `H` to find knight's tours.



In the third part of this lab, we use the Hamiltonian path search to find Knight's tours on n by m grids.[2] In order to do this, all we need to do is create (1) a set of all squares/positions on an n by m board and (2) create a relation that represents when one of these squares can be reached by a knight's move from another.

For example, we will represent a 5 by 5 board by a set of tuples that goes something like this:

```
#{ [0 0] [0 1] [0 2] [0 3] [0 4] [1 0] [1 1] … [4 2] [4 3] [4 4]}
```

That is our `V`, our set of vertices. `[0 0]` would be the square in the lower left corner, `[1 0]` the one to its right, `[0 1]` the one above the lower left corner square and so forth, up to `[4 4]`, the square in the upper right corner.

The edge relation, the `E` parameter in the Hamiltonian path search, contains a pair of these pairs iff one is a knight's move away from the other. So, for example, since `[0 1]` can reach `[1 3]` with a knight's move, the set `E` would then contain the pair `[[0 1] [1 3]]`, as well as the symmetric pair `[[1 3] [0 1]]`.

You need to implement two functions. `next-positions` computes the set of squares on the board that can be reached by a knight's move from a given position. It needs the board as a parameter so that it can determine whether a square that results from adding the relative coordinates of a move (one of the vectors in `Moves`) to its position parameter is, in fact, on the board.

Using this function, you can then implement `create-knights-move-relation`, which does just that: it computes, for a given board, i.e. for a set of positions, the relation that represents whether they can be reached from each other by a knight's move.

2 As it turns out, while general Hamiltonian path search does solve the problem of finding Knight's tours, it does so very, very poorly in the sense that it scales terribly – in fact, there are much better algorithms for finding Knight's tours with much better scaling behavior. If you want to know more about this, take a look at the Wikipedia page on the topic and the references it contains: https://en.wikipedia.org/wiki/Knight%27s_tour