Assignment 2: Search Algorithms

The Problem

Consider the ship routing problem. A large container ship needs to move from port S to port G in the minimum amount of time, but at the same time it needs to avoid regions of land that it cannot pass through.

We can depict this situation in 2D using ASCII art:

```
+--------------------------------+
|                                |
|        XXXXXX       X          |
|         XXXXXXX     XX         |
|            XXXX     XG         |
|            XXXX     XXXX       |
|   XX                 XXX       |
|   XXX                          |
|    XXS            XXXXXXX      |
|   XXXXXX       XXX     X       |
|    XXX XXXXXXXXX       XX      |
|         XXXXXX          X      |
|                                |
+--------------------------------+
```

where `X` denotes the presence of land, `S` denotes the start port, `G` denotes the destination port, a space denotes open ocean, and the symbols `|`,`+`, and `-` denote impassable bounds of the map.

A path through the map can be denoted by a sequence of periods, e.g.

```
+--------------------------------+
|       .......... .......      |
|       .XXXXXX   ... X   .      |
|       . XXXXXXX     XX  .      |
|       .    XXXX     XG...      |
|       ...  XXXX     XXXX       |
|   XX    .            XXX       |
|   XXX   .                      |
|    XXS...         XXXXXXX      |
|   XXXXXX       XXX     X       |
|    XXX XXXXXXXXX       XX      |
|         XXXXXX          X      |
|                                |
+--------------------------------+
```

shows a route between the ports which is not particularly optimal but gets to the destination none-the-less.

We will assume for simplicity that all ships can only move one square north, south, east or west, at a time, and that the cost of each move is 1.

Part 1: Implement a data structure for paths

Before tackling the bigger problem of finding the shortest path between ports, we first of all need to construct a suitable data structure in Clojure for representing states, (a state being a map with a partial path).

One suggestion is to use a hashmap, e.g.

```
{:map ["XSX X",
       "X   X",
       "X   X",
       "XX  X",
       "XX GX"]
 :path [:south :south :east]}
}
```

but the actual representation you use is up to you.

Next, implement the following functions:

- `read-map-from-file`, which takes a filename and reads in a map. The map will be in text format. A newly created state with a zero length path is returned by this function.
- `print-state`, which takes a state and pretty prints it to the console. Both the map and the path associated with the state as a sequence of periods should be displayed.
- `position`, which takes a state and returns the coordinates of the ship on the map, which can be worked out by following the path from the start.
- `start`, which takes a state and returns the coordinates of the starting port marked by `S` on the map.
- `goal`, which takes a state and which returns the coordinates of the destination port marked by `G` on the map.
- `cost`, which takes a state and returns the cost of the state (i.e. the length of the path).
- `heuristic`, which takes a state and computes its heuristic value using the Euclidean distance metric which is given by:

$$h(p_x, p_y) = \sqrt{(p_x - g_x)^2 + (p_y - g_y)^2} \tag{1}$$

  where (p_x, p_y) is the ship's current position and (g_x, g_y) is the goal's position.

- `expand`, which takes a state returns a list of new states obtained by extending the length of the given state's path by one in all possible valid directions. A path is valid as long as it does not collide with (i) land and (ii) itself (i.e. no cycles allowed) and (iii) the border of the map.

Part 2: Implement Best First and A* algorithms

The state representation and data structure in the previous part should enable you to "easily" implement both the Best-First and A* search algorithms for finding a path across the map. The main difference between the algorithms is the value function: for Best-First search, it is simply the heuristic value of the state; but for A* it is the sum of the state's cost and heuristic value.

See the lecture notes or textbook for details on how the algorithms work. The requirements for this part:

- Provide two functions `best-first` and `a-star` to run the algorithms. They both take a single parameter, that being the name of a text file containing the map.
- Your program should have a global variable `verbose` flag that can be set to true or false for debugging. If it is true, then every single state that is evaluated is printed out so you can check its correctness. If verbosity is off, then only the final state is printed. Turn verbosity off when the map size is large.
- Count the number of state expansions performed by each algorithm and have your algorithms print that out as well.
- (Optional) States in Clojure are immutable and most of the functions in Part 1 are pure, so in theory using `memoize` might speed up your program. Does it?
- (Terminology Tip) Best-First search for this assignment is also called "greedy" search and "pure heuristic" search elsewhere. Some sources use the term "best first" to refer to the A* algorithm, so be careful when you are reading!
- (Tip) Start with a very small map, maybe even smaller than the map I have given you, to ensure that the program works without errors.

Part 3: Compare the algorithms

Three maps will be provided for this assignment. You should compare Best-First and A* on all three of these maps, and write up a short report showing (i) the best route found by both algorithms and (ii) the number of state expansions required by each algorithm.

Finally, create your own fourth map. The aim of the map is demonstrate that Best-First is not optimal, but A* is. Therefore you should construct your map so that the Best-First algorithm finds a sub-optimal route, and include this in your report.

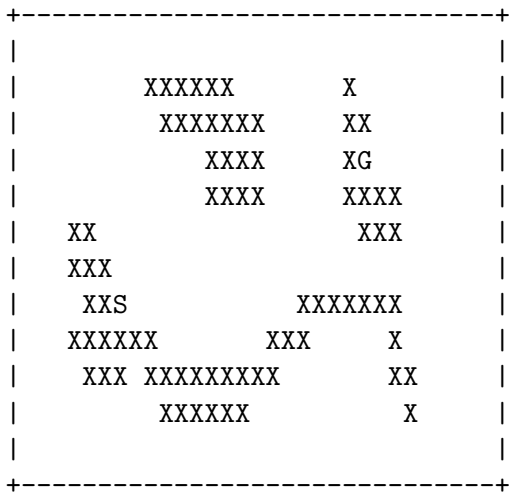
Overall, the report should be 3-4 pages or so, including all diagrams.

Submission

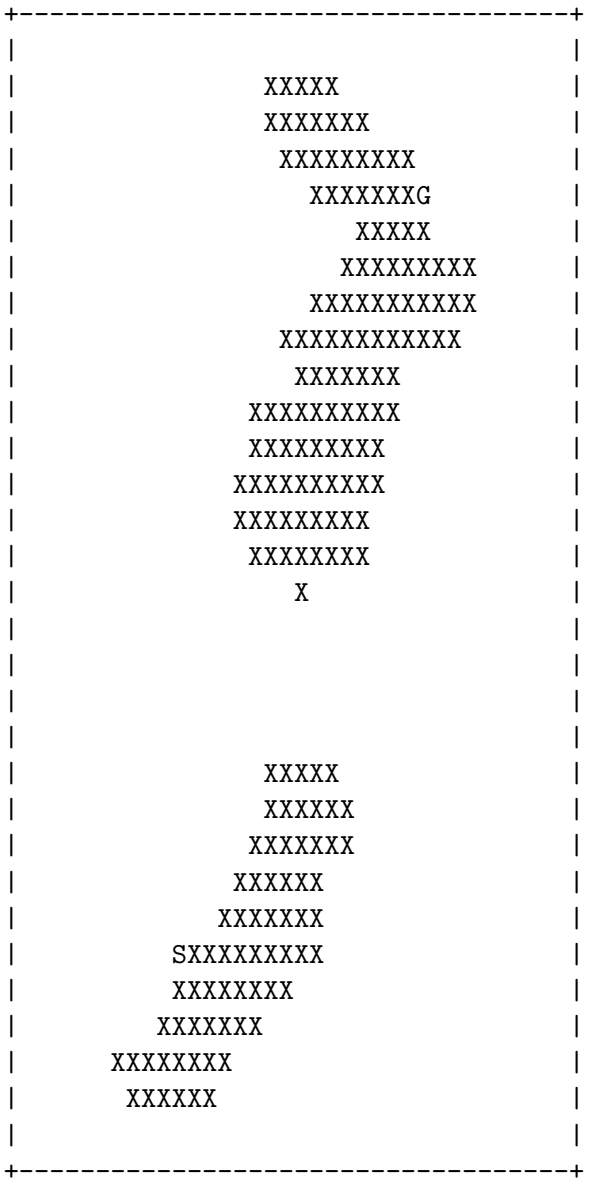
Your submission to Moodle should consist of a zip file containing one Clojure 1.8/9 project folders and one report in PDF format.

Appendix

Map 1 (small starter map):



Map 2 (larger map):



Map 3 (map with a trap):