CS:4420 Artificial Intelligence
Spring 2019

Homework 2

Part A

Due: Friday, Feb 22 by 11:59pm

This assignment has two parts, A and B, both to be done *individually*. This document describes Part A which consist of OCaml programming problems.

Download the accompanying OCaml source file `hw2A.ml` and enter your answers in there where indicated. When you are done, submit your version of `hw2A.ml` through the Assignments section of ICON. Make sure you *write your name in the file.*

Pay particular attention to these points:

- Each of your answers to programming problems *must be free of static errors.* You may receive no credit for problems whose code contains syntax or type errors.
- *Pay close attention to the specification of each problem and the restrictions imposed on its solution.* Solutions ignoring the restrictions may receive only partial credit or no credit at all.
- Do not worry about the efficiency of your solution. Strive instead for cleanness and simplicity. *Unnecessarily complicated code may not receive full credit.*
- This time you are allowed to use any OCaml constructs and library functions you want. You may use mutable variables and data structures although you strongly encouraged to avoid them.

1 n-Puzzle Problem

We consider the *n-puzzle problem*, a generalization of the 8-puzzle from the class notes. In the general case, the board has an arbitrarily large number m of rows/columns and n, the number of tiles, is $m^2 - 1$.

File `hw2A.ml` contains a possible model and definition in OCaml of the n-puzzle problem using a formulation with four operators, `Up`, `Down`, `Left`, and `Right`, each of which moves the empty space in one of the four directions. The code also contains data structures, auxiliary functions and sample implementations of some of the functions needed to solve the problem. In particular, it defines boards and states as a record type and the operators elements of as a discriminated union type.

1. Study the provided code to see how problem states and operators are implemented. Then define function `apply` which applies a given operator o to a given state s. This functions should return `None` when the operator does not apply to s, and return `(Some` s'`)` otherwise where s' is the new state generated by the operator.

2. The code defines also a `Plan` type that encodes sequences of operators meant to go from an initial state to a desired goal state. The provided function `execute` takes a plan p and a state s and applies the operators in p to s in order, returning the resulting state at the end. Based on the given code, define state variables `startState` and `goalState` with a board respectively like the following

```
+-------+
| 1 6 2 |
| 4   3 |
| 7 5 8 |
+-------+
```

```
+-------+
| 1 2 3 |
| 4 5 6 |
| 7 8   |
+-------+
```

   Manually identify a sequence of operators that leads from `startState` to `goalState`. Create a plan called `myPlan` with that sequence. Verify that it is indeed a solution by running `execute` on the state `startState` and checking that the returned state is equal to the state `goalState`.

3. We saw in class a couple of possible heuristics to solve the 8-Puzzle problem with greed search algorithms. Implement the *misplaced tiles* heuristics as a function `misplacedTiles` that takes a (goal) state s_g and a (current) state s_c and returns as an `int` value the number of misplaced tiles in s_c with respect to s_g.

4. The code contains an implementation of a generic informed-search function `treeSearch` which is a simplified version of the recursive best-first procedure described in Figure 3.26 of the textbook. It implements the search fringe using a priority queue. The function is parametrized by an initial state s, a heuristic function f and a goal function *isGoal*. Function f is supposed to be one that takes a node and returns as a `float` a *desirability* value for the node, which is used as its priority by `treeSearch`. Function *isGoal* is supposed to be one that takes a node and returns `true` if the node contains a goal state and returns `false` otherwise.

   Read the provided code carefully, including the various auxiliary functions, to see how informed search can be implemented in practice. Then implement the following instantiations of f:

   (a) `greedyBF` takes a node n and returns the value of n according to the greedy best-first heuristics where $h(n)$ is the the number of misplaced tiles.

   Run `treeSearch` using `startState`, `greedyBF` and the provided goal classifier `isMyGoal`. Report in a comment, as a list of operators, the solution found by `treeSearch`.

(b) `aStar` takes a node n and returns the value of n according to the A* heuristics where $h(n)$ is again the the number of misplaced tiles and $g(n)$ is the cost of generating node n from the start state.

Run `treeSearch` using `startState`, `aStar` and `isMyGoal`. Report in a comment, as a list of operators, the solution found by `treeSearch`.

Did `aStar` lead to a shorter solution than `greedyBF`? Answer this question in a comment.

5. [**Optional, extra credit**] Define a function `manhattanDist` similar to `misplacedTiles` but implementing the Manhattan distance heuristics.

   Define a new version of `greedyBF` and `aStar` that use `manhattanDist` instead of `misplacedTiles`.

   Run the search with the two new heuristics on the same input problem as before and report whether you get a shorter path in each case vs. using `misplacedTiles`.