Practical Exercise 2: River Crossing

October 3, 2010

Assignment. In this exercise you are to implement a solver for a particularly silly instance of river-crossing games. The rules are as follows: At the shore of a river is a family consisting of a father, a mother, two daughters, and two sons. Furthermore, there is a policeman and a thief. Using a raft (also on the same side of the river) everyone has to get across to the other side, with the following restrictions:

- Only the father, the mother and the policeman can operate the raft.
- No more than two people can be shipped at a time.
- The father cannot stay with any of the daughters, without their mother's presence.
- The mother cannot stay with any of the sons, without their father's presence.
- The thief cannot stay with any family member, if the Policeman is not there.

You can play the game on `http://www.robmathiowetz.com/` (needs Flash).

Synopsis. We solve the problem by state space search. The search space C is the set of all possible configurations. Given an initial configuration $c_I \in C$ and a set of possible target configurations $C_T \subset C$ the problem is to identify a path $p = (p_1, \ldots, p_n)$ in C with $p_1 = c_I$ and $p_n \in C_T$. Thereby every member p_s of p with $s > 1$ has to be a valid successor of p_{s-1} in accordance to the the rules of the game, which is captured by a successor relation $S : C \times C$ where $(c_1, c_2) \in S$ iff c_2 is a valid successor of c_1. We generate the sub-space of C that contains only configurations that are reachable from c_I by applying the transitive closure of S to c_I.

Instructions. Specify a data type that models the configurations of the game. Do not hesitate to introduce auxiliary data types and/or type synonyms. Consider using the module `Data.MultiSet` from the `multiset` package. Include an instance declaration of `Show` that pretty-prints the configuration in a concise way.

data *River* = ...

instance *Show River* **where** ...

Define a constant representing the initial configuration, and a predicate whether a configuration is the sought one:

$initial :: River$
$isSolution :: River \to Bool$

Implement the relation S as a function, that maps a configuration to all configurations that can result from a single ride with the raft to the other side. Make sure to include exclusively *admissible* successors, according to the rules of the game.

$admissible :: River \to Bool$
$successors :: River \to [River]$

The transitive closure of S maps a configuration to all reachable configurations by an arbitrary amount of steps. Make sure that the elements in the result are unique.

$successors' :: River \to [River]$

Using this function, assert that the puzzle is solvable by searching a solution in the sub-space that is reachable by `initial`. Of course one also wants to know the sequence of moves that was taken to reach the target configuration. You need not worry, we will deal with this in the following part.

Generalise. The algorithm applies to a large class of similar search problems. In a new module `Search` introduce a type class `Problem` that models these search problems and generalise your solver to find solutions for all instances of that class. Create a data type `Traced` that can be parametrised by any instance of `Problem` and solves the same problem while keeping track of the intermediate configurations. In your original module, make `River` an instance of `Search.Problem` and print a traced solution of the puzzle.

Bonus. Come up with other (more interesting) search space problems and implement them as an instance of your type class. Generalise your solver further in order to supported different strategies. Carry laziness to its extremes.

Don't panic. If you have problems, or if you are looking for a certain library function, ask the advisors for help. This is after all the purpose of their existence.