farmer.pl

The Farmer, the Wolf, the Goat and the Cabbage

Tom Cornell

November 1, 1999

This program implements the search for a solution to the following problem, taken from an exercise in Kowalski's *Logic for Problem Solving*. The scene is a river. On its north bank stand a farmer, a wolf, a goat and a cabbage. Also a boat. The farmer's task is to use the boat to get the wolf, goat and cabbage safely to the south side of the river. The key word here is "safely": the wolf would like to eat the goat and the goat would like to eat the cabbage, and they will do so as soon as the farmer turns his back on them. To make matters worse, the boat can only fit one passenger besides the farmer. Our problem is to find a sequence of river crossings that will assure that the wolf is never left alone with the goat and the goat is never left alone with the cabbage on either bank of the river, and which will eventually put all four protagonists on the south bank of the river.

The crucial information that we need at any point in the computation so as to figure out what to do next is just the current locations of each of the four protagonists. So the four individual locations taken together give us a notion of *state*. Each river crossing represents a change of state in that it leads to a new location for the farmer (minimally) and perhaps one other member of our cast of characters. Note that the farmer is allowed to row the boat across the river empty, so there are four possible passengers: the wolf, the goat, the cabbage and no one. The constraints on the problem—that the farmer has to be the one rowing the boat, that he can take only one passenger, that the wolf and the goat cannot be together without the farmer, and likewise the goat and the cabbage—tell us which new states are accessible from the state we are currently in. For example, from the initial state in which everyone is on the north shore, the only accessible state is the one in which the farmer and the goat are on the south shore, the farmer having rowed the goat across in our first move.

We can represent solutions to the problem as Prolog lists of atoms drawn from the set `wolf`, `goat`, `cabbage` and `no_one`. So, for example, the list

```
[goat,no_one,wolf,goat,cabbage|Rest]
```

represents a partial solution in which the farmer first rows the goat across the river, returns with an empty boat, gets the wolf, returns with the goat, and then gets the cabbage. Initially all four actors (excluding no-one) are on the north shore. At the end of this partial solution the farmer, wolf and cabbage are on the south shore, and the goat is on the north.

Of course, not all lists over these elements will represent solutions to the farmer's dilemma. We can think of the solutions as forming a *language*, that is, a subset of the set of strings $\{\texttt{wolf}, \texttt{goat}, \texttt{cabbage}, \texttt{no\_one}\}^*$. This language is *regular*: it can be recognized by a finite automaton whose states represent the current locations of the farmer, wolf, goat and cabbage, and whose transition relation reflects the problem constraints. However, we can take advantage of the logical nature of the problem and the solution automaton and write a program which uses logical *descriptions* of states and transitions, rather than cranking out the entire finite automaton. That is, we can use the resources of Prolog to collapse the common parts of related collections of states and transitions, yielding a program which is at once smaller and more readable, in that it reflects more accurately the structure of the problem it is meant to solve.

Note that there are infinitely many solutions. That is, our solution-language is an infinite language. For example, once we get everyone over to the south bank of the river, we have the option of returning them all to the north bank and starting over. Because the language is infinite, the search tree for our program must contain an infinite branch. In fact, it contains a number of infinite branches. So we must be careful to make sure we find all our solutions before we disappear down one of those branches. It turns out that the only case we really have to be careful of is the case where we take passenger x across the river and then immediately bring x back. Clearly, we can do this as often as we like. So we take care to keep a record of who the last passenger was, and we add the following condition to our transition relation: the next passenger must not be the same as the last passenger. Initially, and sensibly enough, the last passenger is set to `no_one`.

The code is as follows. The main predicate is `farmer/1`, which holds of a list of elements drawn from the set:

$$\{\texttt{farmer,wolf,goat,cabbage,no\_one}\}$$

just in case that list represents a solution to the problem. `farmer/1` calls `solution/4`, which acts like `delta_star/3` in the finite state automaton implementation. The extra argument is used to hold the last passenger taken across. As noted above, it is initialized to `no_one`.

The predicate `solution/4` calls `crossing/4` to move to select an accessible state `Q1`, reachable by rowing `NextOver` across the river. The solution, whatever it may ultimately be, must satisfy the constraint that at this time `NextOver` is its head. Then we call `solution/4` recursively to work out the remainder of the solution, `PartSol`. Note that in the recursive call the current value of `NextOver` will become the next value of `LastOver`.

That was the code for the general problem solver. The rest of the code amounts to a detailed description of the particular problem to be solved.

The initial and final states are described as any objects which simultaneously satisfy four `where/3` constraints. This makes the statement of the problem more

```
*/
:- use_module( library(lists), [member/2,select/3] ).

farmer( TheSolution ):-
        initial( Q0 ),
        final( Q ),
        solution( Q0, no_one, TheSolution, Q ).
/*
```

```
*/
solution( Q0, _, [], Q ):- Q0 = Q.
solution( Q0, LastOver, [NextOver|PartSol], Q ):-
        crossing( Q0, LastOver, NextOver, Q1 ),
        solution( Q1, NextOver, PartSol, Q ).
/*
```

or less immune to changes in the particular data structure used to implement states.

```
*/
initial( Q ):-
        where( farmer, north, Q ),
        where( wolf, north, Q ),
        where( goat, north, Q ),
        where( cabbage, north, Q ).

final( Q ):-
        where( farmer, south, Q ),
        where( wolf, south, Q ),
        where( goat, south, Q ),
        where( cabbage, south, Q ).
/*
```

The complexity of the problem is all located in the definition of `crossing/4`. First, we determine where the farmer is in the current state (`Bank1`). Then we select a possible passenger `Next` subject to the additional constraints that `Next` currently be located on `Bank1`, that is, with the farmer, and also that `Next` not be the same as `Last`. This may lead to some backtracking, until the call to `select/3` finally comes up with a viable candidate.

Once we have a viable passenger in `Next`, we proceed to construct the next state. First, we have to take care of a problem which can cause considerable difficulties in more complex Artificial Intelligence programs, so it even has its own name: the *frame problem*. This is essentially the problem of implementing inertia: all data elements not affected by the current change of state have to be placed in their old positions in the new state. In our case that means that the characters not selected as the next passenger have to remain in their current locations in the next state. That is what the `frame/3` constraint enforces. After getting our background frame all set up, we put the farmer and `Next` on `Bank2`, opposite to `Bank1`. Then finally we check to see if our newly constructed state satisfies `safe/1`. Note that here again we may have to backtrack. If `Q2` is not safe, then the last choice point will turn out to be in the call to `select/3`: we will backtrack all the way to the passenger selection and try again. If we run out of passengers, then we will have to backtrack all the way back to some previous crossing, and redo that. In this way we blindly search the space of possible crossings for a sequence that works.

```
*/
crossing( Q1, Last, Next, Q2 ):-
        where( farmer, Bank1, Q1 ),
        select( Next, [no_one,wolf,goat,cabbage], Others ),
        where( Next, Bank1, Q1 ), % such that X is with the farmer
        neq( Next, Last ),        % and was not the last across
        /* Construct new state: */
        frame( Others, Q1, Q2 ),
        opposite( Bank1, Bank2 ),
        where( farmer, Bank2, Q2 ),
        where( Next, Bank2, Q2 ),
        safe( Q2 ).

frame( [], _, _ ).
frame( [A|As], Q1, Q2 ):-
        where( A, ALoc, Q1 ),
        where( A, ALoc, Q2 ),
        frame( As, Q1, Q2 ).
/*
```

A state `Q` is safe, that is, it satisfies the predicate `safe/1`, just in case it is both safe for the goat (the wolf cannot eat it) and safe for the cabbage (the goat cannot eat it). Each member of this conjunction of constraints is composed of a disjunction, which is essentially the same in both cases. On the one hand, the goat (cabbage) is safe in `Q` if the wolf (goat) is on the opposite bank in `Q`. On the other hand, the goat (cabbage) is safe in `Q` even if the wolf (goat) is on the same bank as it is, as long as the farmer is also on that bank in state `Q`.

The data structure that we use for our states is defined by the predicate `where/3`. A call to `where( X, Y, Q )` succeeds just in case character `X` is in location `Y` in state `Q`. We implement states as simple 4-place structures under the functor `q`. The first subterm of such a `q`-term represents the farmer's location, the second the wolf's, the third the goat's and the fourth the cabbage's. Of course `no_one` has no specific location. In a certain sense (*ahem*) no one is

```
*/
safe( Q ):-
        safe_for_goat( Q ),
        safe_for_cabbage( Q ).

safe_for_goat( Q ):-
        where( goat, NoSo, Q ),
        where( wolf, SoNo, Q ),
        opposite( NoSo, SoNo ).
safe_for_goat( Q ):-
        where( goat, NoSo, Q ),
        where( wolf, NoSo, Q ),
        where( farmer, NoSo, Q ).

safe_for_cabbage( Q ):-
        where( cabbage, NoSo, Q ),
        where( goat, SoNo, Q ),
        opposite( NoSo, SoNo ).
safe_for_cabbage( Q ):-
        where( cabbage, NoSo, Q ),
        where( goat, NoSo, Q ),
        where( farmer, NoSo, Q ).
/*
```

everywhere....

```
*/
where( farmer, F, q(F,_,_,_) ).
where( wolf, W, q(_,W,_,_) ).
where( goat, G, q(_,_,G,_) ).
where( cabbage, C, q(_,_,_,C) ).
where( no_one, _, _ ).
/*
```

Finally, we have some predicates for expressing essentially negative information. The relation `opposite/2` is trivial; note only that if we want it to be a symmetrical relation (and of course we do) then we must explicitly make it so somehow.

The predicate `neq/2` implements another way—even more compact but even less efficient—of implementing inequality over elements of a finite set: after selecting `X` from our domain of elements, as long as that domain contains no duplicates, then `Y` can only be in the remainder of the domain if it is not equal to `X`.

```
*/
```

```
opposite( south, north ).
opposite( north, south ).

neq( X, Y ):-
        select( X, [no_one,wolf,goat,cabbage], Rest ),
        member( Y, Rest ).

/*
```