Problem Wk.13.1.2: Farmer et al.: Search

Read Section 8.2 of the class notes. See also Section 8.3 of the class notes for an example.

One version of a standard puzzle called the *Farmer, Goat, Wolf, Cabbage* goes as follows:

- The farmer has a goat, a wolf and a head of cabbage (don't ask why a wolf...).
- They come to a river (they're on left bank) and need to get to the other side (the right bank).
- There's a boat there that fits at most two of them (it's a big head of cabbage); the farmer must always be one of the two in the boat.
- If the farmer leaves the goat and cabbage on the same side of the river, when he is not present the goat will eat the cabbage (so that's not a legal state).
- Similarly, if the farmer leaves the goat and the wolf on the same side of the river, when he is not present... well, that's not legal either.

So, the problem is to find a sequence of actions that go from the initial state where they are all on the left bank to the final state when they are all on the right bank.

We will implement this domain by defining a class which is a subclass of the `SM` class. We will then use `search.smSearch` to find sequences of actions.

The state of the state machine needs to keep track of which bank of the river everyone is on. So, a state will be:

```
(farmerLoc, goatLoc, wolfLoc, cabbageLoc)
```

where each of these locations is `'L'` or `'R'` (for left or right). The boat is always with the farmer so no need to keep track of that.

Assume that the actions are:

- `'takeNone'`,
- `'takeGoat'`,
- `'takeWolf'`, and
- `'takeCabbage'`.

Each action indicates a trip from whatever side of the river the farmer is on, to the opposite side of the river.

Define the state machine class `FarmerGoatWolfCabbageClass`. The state machine needs to have a `getNextValues` method and a `legalInputs` attribute, which will serve the role of the `successors` function and `actions` list respectively. The `startState` of the state machine indicates the initial state for the search and the `done` method of the state machine serves as the goal test function.

Make sure that no illegal or impossible actions are permitted, that is, states in which the goat and cabbage or the wolf and goat are alone together. Also, you must make sure that if the farmer is to take a passenger across the river, then the passenger must be on the same side of the river as the farmer.

If your machine receives an input action that is illegal or impossible, it should simply stay in the same state.

Use the indices defined below to index the elements of the tuple. So `state[wolf]` is `state[2]`, which is the third component of the state tuple.

Note that states need to be tuples, not lists, so that they can be keys in the dictionary used in the search code to implement dynamic programming. If you need to turn a tuple x into a list, you can do that with `list(x)`. If you need to turn a list x into a tuple, you can do that with `tuple(x)`.

You should write and debug your code in Idle, you can use the file

`swLab13Work.py`

which imports the relevant library files. When you have it working, paste your answer in below and check it.

```
# Indices into the state tuple.
(farmer, goat, wolf, cabbage) = range(4)

class FarmerGoatWolfCabbage(sm.SM):
   startState = None
   legalInputs = None
   def getNextValues(self, state, action):
      pass
   def done(self, state):
      pass
```

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011