Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)

531: Prolog
'Crossings' — Assessed

Issued: 22 October 2018 **Due:** 1 November 2018

Note You may use the Sicstus built-in list predicates `append/3`, `member/2`, `nonmember/2`, `memberchk/2`, `length/2`, but *do not use* the built-in predicate `sort/2`, or the Sicstus `lists` library for this exercise.

Aims

- To give you practice writing both declarative and procedural style programs
- To introduce you to the representation of an AI problem in Prolog
- To give you practice with recursive list processing tasks

1 Introduction

This is a variation of the classic farmer-wolf-goat-cabbage puzzle. We have added another item (a bag of fertilizer) and a number of other features.

1.1 The Problem

A river has a north bank and a south bank. On the north bank there stands a farmer with a wolf, a goat, a cabbage, and a bag of fertilizer.

The farmer's goal is to transport all of these and himself to the south bank. He can cross the river in a boat, either alone or accompanied by just *one* other item. However, it is not *safe* to leave behind the wolf with the goat, as the wolf might eat the goat if the farmer is not present. Likewise it is not *safe* to leave behind the goat with the cabbage (the goat might eat the cabbage).

The farmer will therefore have to make several journeys back and forth, in such a way that all items get transported and nothing gets eaten.

For each journey across the river the farmer has to pay a fee, depending on how many items (including himself) are transported in that journey. He wants to calculate how the total cost of achieving the goal varies according to how he solves it.

1.2 Prolog Representation

A *state*, describing the situation immediately before or after a journey, is represented by a Prolog term of the form `North-South`.

- `North` is a list of the items (Prolog constants) currently on the north bank
- `South` is a list of the items (Prolog constants) currently on the south bank

The constants representing the figures in the puzzle: the farmer, wolf, goat, cabbage, and bag of fertilizer; are `f`, `w`, `g`, `c` and `b`, respectively. So, a state `[f,w,g,c,b]-[]` corresponds to the start of the puzzle. In order to make some of the program more interesting to write, we generalise the definition of a state to allow other items to appear on the river bank. So, the lists `North` and `South` can include any constant, meaning `[w,g,tim,lukeskywalker]-[f,c]` and `[w,w,w,w]-[]` are states.

2 What To Do

2.1 Obtain the Exercise Files

You will need to clone the skeleton respository to your DoC home directory in order to work on it. Later you will need to *push* your changes back to the server.

- You can get your skeleton repository by issuing the following command (all on one line, replacing the occurrence of *`login`* with your DoC login):

  ```
  git clone https://gitlab.doc.ic.ac.uk/lab1819_autumn/531_crossings_login.git
  ```

  Or, if you have set up ssh key access you can use:

  ```
  git clone git@gitlab.doc.ic.ac.uk:lab1819_autumn/531_crossings_login.git
  ```

- This will create a new directory called `531_crossings_login`. Inside you will find the following files and directories:
  - `crossings.pl` — this is the source file you should edit to implement the procedures required for this exercise.
  - `.git` and `.gitignore`

2.2 Write Your Program

Follow the steps outlined below. Each of the named procedures will be separately tested when you submit your answer, so ensure you write them all as specified. Make sure you regularly *add*,

commit and *push* your work back to the Gitlab repository.

Task 1 Write a Prolog program for `safe(+Bank)` which holds when `Bank` is a given list of items (those currently on one or other river bank) that is safe as defined above. For example, `safe([w,c])` and `safe([f,w,c])` should succeed, but `safe([g,c,b])` should fail. **Note:** the items in a bank may occur in any order — allow for this. You can assume that `Bank` is always ground when `safe(Bank)` is called.

Task 2 Write a Prolog program for `goal(+State)` which holds when `State` (given, ground) is a goal state. A goal state is specific to the puzzle outlined above and occurs when the five puzzle figures (exactly one of each) are on the south bank, there are no other items on the south bank, and the north bank is empty. For example, `goal([]-[c,w,g,f,b])` should succeed, but `goal([w,f,c]-[b,g])` should fail.

Task 3 Write a Prolog program for the predicate `equiv(+State1, +State2)` which holds when the two (ground) states are equivalent, that is, when they have the same number of each item in their north banks and the same number of each item in their south banks. For example, `equiv([b,c]-[f,w,g], [c,b]-[g,f,w])` should succeed, but `equiv([b]-[f,w,c,g], [c,b]-[g,f,w])` should fail.

Task 4 A *sequence* is a list of states. Write a Prolog program for `visited(+State, +Sequence)` which holds when `State` is equivalent (as defined above) to some member of `Sequence`. For example, `visited([b,c]-[f,w,g], [ [c,b]-[g,f,w], [c,b,f,w]-[g] ])` should succeed. You can assume that both `State` and `Sequence` are ground.

Task 5 Write a Prolog program for the predicate `choose(-Items, +Bank)` which, given `Bank`, returns `Items` as a list of either one item or two items (including `f`) such that if `Items` make a journey to the other bank then the remainder left behind would be safe. For example, `choose(Items, [g,f,b])` could return `Items` such as `[f]`, `[f,g]` or `[f,b]`, but `choose(Items, [g,f,c])` should not return `Items` such as `[f]` (the remainder left behind is not safe) or `[c]` (a journey without the farmer is not possible).

Task 6 Write a Prolog program for the predicate `journey(+State1, -State2)` which holds when state `State2` results from a possible *safe* journey from (given) state `State1`. Note that in general there will be many possible values for `State2` from any given state `State1`.

Task 7 Write a Prolog program for the predicate `succeeds(-Sequence)` which returns a `Sequence` that starts with an initial state, ends with a goal state, and is such that each state (other than the first) is obtained from its predecessor by a safe journey. `Sequence` should contain no 'loops', i.e., `succeeds(Sequence)` should call `visited/2` defined earlier.

Define `succeeds` in terms of an auxiliary predicate `extend` as follows. Start with the clause

```
succeeds(Sequence) :-
   extend([ [f,w,g,c,b]-[] ], Sequence).
```

Now write two clauses (base case and recursive case) to define `extend`. The first argument in `extend` represents the sequence of states visited so far. The recursive clause for `extend` first tests that in the sequence of states visited so far, the last state is not a goal state; then it makes a journey to yield some next state, checks that this state has not already been visited, adds it to the current sequence of visited states, and recurses.

For efficiency, you might prefer to store the sequence of states visited so far in reverse order. If so, ensure that `Sequence` as returned by `succeeds(Sequence)` contains the states in the order they are visited, that is, from initial state to goal state. Alternatively, you might prefer to define `succeeds/1` in terms of `extend/3` where `extend(VisitedSeq, LastState, Sequence)` is such that `VisitedSeq` is the sequence of states visited so far and `LastState` is the last state visited in `VisitedSeq`.

Task 8 Write a Prolog program for the predicate `fee(+State1, +State2, -Fee)` which returns `Fee` as the fee that must be paid for a journey from (given) `State1` to (given) `State2`. You can assume that `State1` and `State2` represent a valid journey (i.e. `journey(State1, State2)` would succeed). The fee if the farmer travels alone is some number `F1` and is otherwise some number `F2`. Define your own choice of these numbers by including a clause of the form `fees(F1, F2)`.

Task 9 Write a program defining `cost(-Sequence, -Cost)` which returns a `Sequence` satisfying `succeeds(Sequence)` together with `Cost` as the sum of the fees for all the journeys undertaken within `Sequence`.

2.3 Testing

You should test your programs on a range of suitable examples before submission. You can automate your testing using the `plunit` Sicstus library. See the Sicstus documentation for details.

Your submitted work will be automatically tested, so you must ensure that:

- Your Prolog program is written in `crossings.pl`.
- Your program *COMPILES WITHOUT ERRORS* on the **Linux Sicstus** system installed on the lab machines.
- You have *NOT* included any print or write statements in your submitted code. Use them by all means for tracing/debugging but do not include them in your submitted program.

You can verify that your code runs and produces sensible output by requesting an autotest online via `https://teaching.doc.ic.ac.uk/labts`. This is NOT the full set of tests that will be used in assessing your work, so you cannot assume that your program is correct if all the tests pass. However, you will be able to check that your code was loaded and executed without errors. (In particular look at the log output as well as the test results.)

3 Submission

Submit By 1 November 2018

1. **Push To Gitlab.** Use `git add`, `git commit` and `git push` to update the Gitlab server with the changes you have made to the skeleton repository. Use `git status` to confirm that you have no local changes you have not pushed, and then inspect the files on Gitlab: `https://gitlab.doc.ic.ac.uk`.
2. **Submit to CATE.** Click the **Submit to CATe** button in LabTS (`https://teaching.doc.ic.ac.uk/labts`) for the version of your code that you want to be marked. Complete the declaration and submission in CATE.

4 Assessment and Feedback

This exercise is worth 20% of the marks allocated for coursework on this module. Your solutions will be marked on correctness, code design and readability.

Grades

`F-E:` The answer shows a clear lack of understanding of how to write Prolog code. The behaviour of the program is likely to be incorrect.

`D-C:` The submission shows that the student can write Prolog, but the behaviour of the program is incorrect. Solutions may have serious efficiency problems. Solutions may be grossly overcomplicated, or otherwise poorly designed. Code style may be poor.

`B-A:` The behaviour of the program is mostly or entirely correct. Solutions may have efficiency problems, and/or some design flaws. Code style is good.

`A+:` The behaviour of the program is correct. Some minor efficiency problems may exist in some procedures, but otherwise code is well designed, simple and elegant. Code style is good.

`A*:` There are no obvious deficiencies in the solution, including efficiency of the program, program design and the student's coding style.

4.1 Return of Work and Feedback

This exercise will be marked and returned by **16th November 2018**. Feedback on your solution will be given on the returned copy, and feedback will be given to the class as a whole in the support lectures.