ECM2418 Computer Languages and Representations
Continuous Assessment 2: Prolog

Dr Enrico Malizia

Handed out	Handed in
Friday 2nd November 2018 (T1:06)	Friday 30th November 2018 (midday) (T1:10)

This Continuous Assessment is worth 15% of the module mark.

Your submission for this CA will be both on paper, using BART, and electronic, using the EMPS Anonymous Electronic Coursework Submission system (`http://empslocal.ex.ac.uk/cgi-bin/submit/prepare`). In this system, you should submit to the folder entitled "2018-11-30 ~ Enrico Malizia ~ CA2 Prolog". *Remember that the submission deadline, for both the paper and the electronic version, is on* ***midday*** *of the hand in day. However, you will need to attach the receipt of the electronic submission to the paper submission, which means that before submitting your work via BART you will have already submitted your work electronically. Please keep this in mind while scheduling the time plan for your submission.*

All students are reminded of the University regulations on academic honesty and plagiarism.

Your task

For this coursework, your task is to write Prolog predicates to solve a number of problems regarding the underground train network of a city. In particular, we assume to model the underground network via a Prolog database. In this database, a fact `station(a)` states that in the city there is an underground station called `a`, a fact `line(red)` states that there is an underground line called `red`, and a fact `stop(red,3,e)` states that the station `e` is the third station on the line `red`.

- For simplicity, we assume that each line can be travelled in one direction only: from the first stop, i.e., the stop number one on the line, to the last stop, i.e., the stop with highest number on the line.
- If two (or more) lines have a stop in the same station `s`, it is always possible for a passenger to stop at station `s` and change line.
- There is a path from station `s1` to station `s2` if it is possible to reach `s2` from `s1` by operating zero, one, or more changes of lines.

You can download from ELE the file `underground.pl` containing the Prolog representation, according to the above mentioned scheme, of the underground network shown in Figure 1 (it is not a very realistic network, but it is enough for our purposes). Colours of the edges are used to distinguish different lines, and the arrows on the graph edges denote the direction of travel for the lines.

(1) Define a Prolog predicate `multiple_lines(S)` computing the stations `S` serving more than one line (you can ignore the fact that a station can be returned as answer multiple times). For example:

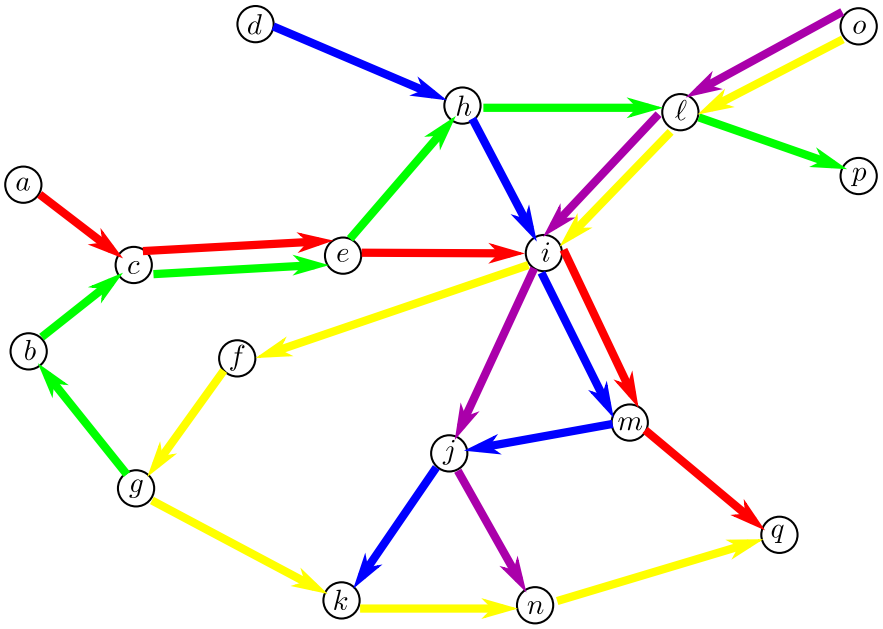

Figure 1: A graphic representation of the underground network stored in the Prolog database `underground.pl`

```
?- multiple_lines(S).
S = c ;
S = e ;
S = i ;
S = i ;
S = i ;
S = m ;
S = q ;
S = g ;
S = c ;
S = e ;
S = h ;
S = l ;
S = l ;
(... the answers continue...)
```

Observe that station `b` is not returned because station `b` is only on the green line, while station `c` is returned because station `c` is on the green and the red lines. **(5 marks)**

(2) Define a Prolog predicate `termini(L,S1,S2)` computing the two terminal stations `S1` and `S2` of the line `L`, more in particular `S1` and `S2` are the first station and the last station of the line `L`, respectively. For example,

```
?- termini(red,S1,S2).
S1 = a,
S2 = q.
```

(15 marks)

(3) Define a Prolog predicate `list_stops(L,List)` computing the list `List` of all the stations of the line `L`. Extra marks are awarded if the stations in the list are ordered from the first station of the line to the last station of the line. For example,

```
?- list_stops(red,List).
List = [a, c, e, i, m, q] ;
false.
```

(10 marks; 10 extra marks if correct order)

In the rest of the coursework, the tasks will be to write predicates to compute paths in the underground network. A path in the underground is represented via a list of functions `segment(·,·,·)`. For example, the list `[segment(red,a,e),segment(green,e,l),segment(purple,l,j)]` represents a path from station `a` to station `j` composed of three parts/segments: the first part of the path is on the `red` line from station `a` to station `e`; the second part of the path is on the `green` line from station `e` to station `l`; and the third part of the path is on the `purple` line from station `l` to station `j`. Observe that line changes are performed at stations in which the respective lines intersect.

(4) Define a Prolog predicate `path(S1,S2,Path)` computing a path from the station `S1` to the station `S2` which is returned in `Path`.

(**Hint:** Remember that, over cyclic graphs, Prolog might get stuck in a loop if the predicate is not properly designed to explore cyclic graphs, and the graph in the database provided on ELE *is* cyclic. Your code should check that it is not exploring parts of the network that has already been explored. In the appendix you can find more details on this and a code example.)

To get full marks your solution should fulfill two properties:

(a) The paths computed should not pass twice through the same station. For example, in the path

```
[segment(yellow,l,g),segment(green,g,h),segment(blue,h,k)]
```

the station `i` is traversed twice, once when on the yellow line, and once when on the blue line. Paths like this should not be returned in output by your code.

Observe that if this property is fulfilled, then the computed path is not cyclic. Hence, fulfilling this property is enough to avoid that the code loops while exploring the network.

(**Hint:** for this property, it can be useful to define the predicates `stations_traversed(Path,Set)` and `segment_adds_cycle(segment(L,S1,S2),Path)`.

The former predicate computes the set `Set` of the stations traversed in the path `Path`.

For example, `stations_traversed([segment(yellow,l,g), segment(green,g,h)], [l, i, f, g, b, c, e, h])` holds true.

The latter predicate decides whether the addition of the segment `segment(L,S1,S2)` to the path `Path` causes the generation of a cycle in the path, i.e., decides whether there is a station among the ones traversed by the new segment to be added that would be traversed twice.

For example `segment_adds_cycle(segment(blue,h,m),[segment(yellow,l,g),segment(green,g,h)])` holds true.)

(b) The paths computed should not suggest to take a line more than once. For example, in the path

```
[segment(red,a,e),segment(green,e,h),segment(blue,h,m),segment(red,m,q)]
```

the red line is taken twice (first and last segment). Paths like this should not be returned in output by your code. Fulfilling this property guarantee that paths like `[segment(red,a,i),segment(red,i,q)]` are not returned

For example,

```
?- path(i,k,Path).
Path = [segment(blue, i, k)] ;
Path = [segment(yellow, i, k)] ;
Path = [segment(red, i, m), segment(blue, m, k)] ;
Path = [segment(purple, i, j), segment(blue, j, k)] ;
false.
```

(25 marks for a basic solution; 15 extra marks if property (a) is fulfilled; 10 extra marks if property (b) is fulfilled)

(5) By exploiting the predicate `path(S1,S2,Path)` defined in the question above, define a Prolog predicate `minimum_path(S1,S2,Path)` where `Path` is a path from station `S1` to station `S2` with minimum number of changes (this implies that, if `S1` and `S2` are on the same line, then your code should return only the paths with no changes).

For example,

```
?- minimum_path(i,k,Path).
Path = [segment(blue, i, k)] ;
Path = [segment(yellow, i, k)] ;
false.

?- minimum_path(a,n,Path).
Path = [segment(red, a, i), segment(purple, i, n)] ;
Path = [segment(red, a, i), segment(yellow, i, n)] ;
false.
```

Observe that we are simply looking for the paths with minimum number of changes. In the example above, the path `[segment(red, a, i), segment(purple, i, n)]` actually traverses fewer stations than `[segment(red, a, i), segment(yellow, i, n)]`, but we do not care about that, since both paths have just one change of line. **(10 marks)**

What you should submit

Your submission should comprise both an electronic submission and a hard-copy submission, as follows:

1. The electronic submission should consist of one `.pl` file, containing the Prolog code for your solution, and one `.pdf` file, containing a descriptive account of how your program meets the required functionality (max 4 pages) *and* the listing of your Prolog code (which is not counted in the 4 pages limit).
2. The hard-copy submission should be the print out of the `.pdf` file above. A copy of a receipt from the Harrison E-submit system should also be attached to the BART sheet. Note that by receipt we mean a copy of the e-mail from the E-submit system requesting that you confirm the upload of your work by clicking a link.

How your submission will be assessed

Your program will be tested on the input database provided on ELE and on a different input database as well. Hence, your program is supposed to work on a generic input file, and not just on the one provided as example. You can assume that the input database is correct, i.e., you do not need to check for errors in the database (like, e.g., stations appearing twice in a line, or stations missing in a line, and so on). Marks will also be awarded in accordance with the following criteria:

- The program as a whole should exhibit good logic programming style. This means that solutions adopting a logic programming flavour will be valued better than solutions mimicking an imperative approach via Prolog.
- The program description should be correct, clear, and possibly concise. Aim of the program description is to show the reasoning behind your solution and hence show that you have not arrived to the solution only by chance.
- Your code should be readable and understandable. If your solution uses an approach that is not self-explanatory and in your program description your approach is not properly explained, then the question will get 0 marks.
- If either of the submissions (paper or electronic) is not done, then the coursework will get 0 marks.
- If your code contains errors and it is not correctly loaded by SWI-Prolog, then the coursework will get 0 marks.
- If parts of the code are missing in the report, then the questions whose code is missing will get 0 marks. (If the code is entirely missing in the report, then the coursework will get 0 marks).

Appendix

Prolog having a tour over graphs

In Prolog, computing a path in a graph could be a bit challenging, because, given the resolution strategy used by Prolog, on cyclic graphs a non-properly designed predicate could not stop. To avoid endless loops, a Prolog predicate looking for paths in (cyclic) graphs has to keep track of the vertices already visited (in order to avoid exploring areas of the graph that have already been visited). A Prolog predicate `path(S1,S2)` verifying the existence of a path *from* station `S1` *to* station `S2`, and correctly working also over cyclic graphs, can be devised as follows.

```
station_traversed(S,[segment(L,S1,S2)|_]):-stop(L,N,S),stop(L,N1,S1),stop(L,N2,S2),
  N1=<N,N=<N2,N1=<N2,!.
station_traversed(S,[_|R]):-station_already_traversed(S,R).

path(S1,S2):-pathHelper(S1,S2,[]).
pathHelper(S1,S2,_):-stop(X,N1,S1),stop(X,N2,S2),N1<N2.
pathHelper(S1,S2,ATTEMPT):-stop(X,N1,S1),stop(X,N_Med,S_Med),N1<N_Med,
  \+station_traversed(S_Med,ATTEMPT),pathHelper(S_Med,S2,[segment(X,S1,S_Med)|ATTEMPT]).
```

Predicate `station_traversed(S,Path)` checks whether station `S` is traversed in one of the segments of the path `Path`. For example, `station_traversed(i,[segment(yellow,l,g),segment(green,g,h)])` holds true. Predicate `path(S1,S2)` checks whether there is a path from station `S1` to station `S2`. Predicate `path` uses a predicate `pathHelper` whose third argument is a list of the segments collected up to that moment in order to verify whether the intermediate station `S_Med` has already been visited. The implementation of `path` shown here only checks the existence of a path between two stations, it does not return what this path is. To compute the path, the predicate needs to be changed a bit, which is left to the student. The predicate `station_traversed` as defined here, although it guarantees that the code does not get stuck in a loop, is not enough to fulfill the property "(a)" mentioned in question 4.