Prolog Assessed Exercise

Lecturer: David Eyers <david.eyers@cl.cam.ac.uk>
Assessor: Sean Holden <sean.holden@cl.cam.ac.uk>

10th November 2010

The purpose of this exercise is to implement in Prolog the Bellman-Ford algorithm for computing single-source shortest paths in a weighted, directed graph. You should work through and complete all steps of this document. Ensure that each implemented predicate behaves correctly under backtracking. You may make reasoned use of the cut operator where appropriate. The date above indicates the document version. Check the subject web page for updates. The changes made between document versions will be listed on the subject web page. Details about submission and marking are at the end of this document. This document consists of six pages.

1 Basic Operations

During its operation, the shortest path algorithm maintains a list of triples. Each triple represents a node in the graph, the cost of reaching it, and the preceding node from which to reach it with that cost. We will represent them in the form `V+D+P` where `V` is the graph node, `D` is the cost, and `P` is the predecessor node.

While running the Bellman-Ford algorithm on a graph, we will not assume that the triples in a list are in any particular order. For ease of visual examination and comparison or results, however, we will define a standard ordering for the triples: first we will order triples in numerically increasing distance (`D`), and for verticies with the same (minimum) distance, we will order triples in alphabetical order of their vertex names. You may assume that any given vertex name only appears once in a list of triples.

2 Supporting Predicates

2.1 Triple comparison

Write a predicate `lessthan(+A,+B)` that is true iff[1] the triple `A` is "smaller than" the triple `B` for the notions of triple and order introduced above. Your predicate should fail if either or both of its arguments are not triples. Note that in Prolog, operators such as < (less than) are intended for arithmetic use, and will not order atoms such as the node names. Instead the operator `@<` and the like, apply the "standard order of terms", which includes ordering atoms alphabetically. Make sure your comments explain which operators you use and why.

Include some test cases in your source file. Your tests should be executed when your file is loaded. Recall that `not/1` evaluates negation by failure (as does the `\+` operator). You can use `not/1` (or equivalent) to implement negative tests: cases in which your predicate is expected to fail.

[1] "if and only if"

2.2 Merge sort

We will use the merge sort algorithm to effect the ordering of triples discussed above. Include the two predicates `msorttriples/2` and `divide/3` that are reproduced below in your source file. Add a brief explanatory comment to each. Note that you will implement the `mergelists/3` predicate in the next step of the assessed exercise.

```
msorttriples([],[]).
msorttriples([A],[A]).
msorttriples([A,B|Rest],S) :-
        divide([A,B|Rest],L1,L2),
        msorttriples(L1,S1),
        msorttriples(L2,S2),
        mergelists(S1,S2,S).

divide([],[],[]).
divide([A],[A],[]).
divide([A,B|R], [A|Ra], [B|Rb]) :-
        divide(R,Ra,Rb).
```

2.3 Merge sort's list merging predicate

Write a predicate `mergelists(+X,+Y,-Z)` that takes two lists of triples that have already been sorted, `X` and `Y`, and unifies `Z` with a list that selects all of the elements from `X` and `Y` such that `Z` is also sorted.

Include some test cases for the `mergelists/3` predicate. Show that your predicate can generate values of `Z` as well as check that provided values are correct. Now that you have defined `mergelists/3`, you should include some test cases for the `msorttriples/2` predicates in your source file also.

2.4 List replacement

We return to the triples discussed at the beginning of this document. As the Bellman-Ford algorithm runs, we store triples for the node distances that we know about. Any node not included in the triple list is unreachable (i.e. has an infinite shortest path cost). When we find a new minimum distance to a node, we want to lower the known distance to it, or to add it to the triple list if that node is not yet present.

Write a predicate `addreplace(+Find,+Replace,+InList,-OutList)` that tries to unify `Find` with each element of `InList`. If it succeeds, it replaces the corresponding element in `OutList` with `Replace`. If no elements of `InList` unify with `Find`, then `OutList` should be unified with a list containing all of the elements of `InList`, as well as a new element `Replace`. It can be assumed that only one replacement needs to be made.

3 Representing Graphs

We shall encode the graph that we wish to analyse as a list of edges. Each edge consists of a compound term `Src-Dest-Dist` where `Src` is the source node, `Dest` is the destination node and `Dist` is the edge cost between the nodes. Write a predicate `graph(example,-List)` which unifies `List` with the list that represents the graph in figure 1. We use the additional argument `example` to allow us to run our program with different graphs later.

You may assume that in any graph there is at most one edge between any two particular nodes.

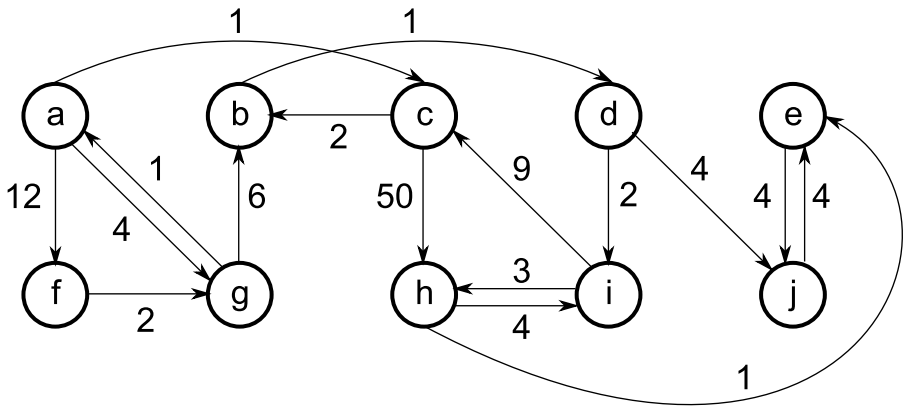

Figure 1: An example graph

3.1 Start Nodes

Write a rule `nodes(+Graph,-Nodes)` that unifies `Nodes` with a list that contains the name of each node in the graph once. Remember that a node can be at the start or the end of an edge in your graph representation. One approach to do this is to construct the `nodes/2` predicate from these auxiliary rules:

- `unroll`, which builds a list (containing duplicates) of all the nodes in the graph;
- `unique`, which removes duplicates from a list. Hint: use negation, and the built-in `member(E,L)` function which is true if `E` is an element of `L`.

Now write a rule `node(+Graph,-Node)` that unifies `Node` with a node in the graph. This rule should iterate through all nodes in the graph when backtracking but should only return each node once.

4 The Bellman-Ford algorithm

The Bellman-Ford algorithm for finding shortest paths from a single node is described in detail in the Algorithms II course. Students might also choose to consult Wikipedia[2], and decide to view the various Java applet visualisers available on the web[3].

Here we show most of a basic implementation in Prolog. Copy this code into your source file and add comments to explain the purpose and operation of each predicate. [Aside: Note that some of the predicates below have very similar structure to each other. As in functional languages, a "fold left" predicate can be defined and used to factor out this sort of similarity, but you are not required or expected to do so.]

```
bellmanford_inner(Graph,InitialVertexData,MinDists):-
        nodes(Graph,[_|Vs]),
        relaxEdgesRepeatedly(Vs,Graph,InitialVertexData,MinDists).

relaxEdgesRepeatedly([],_,VData,VData).
relaxEdgesRepeatedly([_|Vs],Edges,VData,NewVData):-
        relaxEdges(Edges,VData,NextVData),
        relaxEdgesRepeatedly(Vs,Edges,NextVData,NewVData).
```

[2] `http://en.wikipedia.org/wiki/Bellman-Ford_algorithm`
[3] e.g. `http://links.math.rpi.edu/applets/appindex/graphtheory.html`

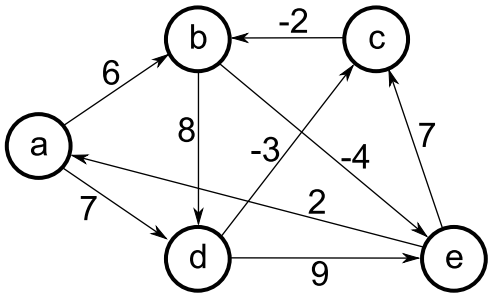

Figure 2: A graph with negative edge weights

```
relaxEdges([],X,X).
relaxEdges([E|Es],VData,NewVData):-
        relax(E,VData,NextVData),
        relaxEdges(Es,NextVData,NewVData).

relax(A-B-W,VData,NextVData):-
        member(A+ADist+_,VData),
        RelaxedDist is ADist + W,
        relaxable(B,VData,RelaxedDist),!,
        addreplace(B+_+_,B+RelaxedDist+A,VData,NextVData).
relax(_,VData,VData).

relaxable(B,VData,RelaxedDist):-
        member(B+BDist+_,VData),!,
        RelaxedDist < BDist.
relaxable(_,_,_).
```

Add an additional predicate `bellmanford(+Graph,+Start,-MinDist)` that calculates the minimum distances from the node `Start` by querying the `bellmanford_inner` predicate. Hint: you need to decide on a suitable value for the initial vertex data.

Include a few simple tests for your implementation. You must demonstrate that the algorithm is correct when there are multiple paths to the same node; there are loops in the graph; and some nodes are unreachable from the start node.

4.1 Negative edge cycles

The Bellman-Ford algorithm will correctly handle graphs that include negative edge weights, provided that they do not contain a negative cycle (i.e. a cycle whose edges' weights' sum is negative). However note that for graphs that have only non-negative edge weights, Dijkstra's shortest path algorithm will usually be more efficient.

Encode the second example graph, as shown in figure 2 in the predicate `graph(example2,-List)`. Note that you may need to put in extra parentheses (or equivalent) to avoid the parser becoming confused about negative weights and the `-` compound terms. For example, a hypothetical edge from `c` to `d` with weight `-3` could use the syntax: `c-d-(-3)`).

Copy your `graph(example2,-List)` predicate to an `graph(example3,-List)` predicate. In your `example3` graph, change the edge weight `b-d-8` to be `b-d-2`: this will create a negative cycle in the graph.

Modify your `bellmanford` predicate so that it unifies the minimum distances term with the atom `negative_cycle` when it detects a negative cycle in the graph.

4.2 Reverse shortest paths

Recall that the preceding node on the shortest path was recorded as part of each triple.

Write a predicate `rshortestpath(+Graph,+StartNode,+FinishNode,-Path)` that unifies `Path` with the list of nodes in reverse order along the shortest path from `StartNode` to `FinishNode`. As much as possible you should make use of predicates that you have already written to perform this task.

5 Test Cases

Append the following test clauses to your file:

```
test(Name,Start) :-
        graph(Name,G),
        node(G,Start),
        bellmanford(G,Start,L),
        msorttriples(L,Lsorted),
        minCost(Name,Start,Lsorted).
test(Name) :- findall(S,test(Name,S),L), sort(L,Lsort),
              graphNodes(Name,Lsort).

minCost(example,a,[a+0+_, c+1+_, b+3+_, d+4+_, g+4+_, i+6+_, j+8+_, h+9+_,
                   e+10+_, f+12+_]).
minCost(example,b,[b+0+_, d+1+_, i+3+_, j+5+_, h+6+_, e+7+_, c+12+_]).
minCost(example,c,[c+0+_, b+2+_, d+3+_, i+5+_, j+7+_, h+8+_, e+9+_]).
minCost(example,d,[d+0+_, i+2+_, j+4+_, h+5+_, e+6+_, c+11+_, b+13+_]).
minCost(example,e,[e+0+_, j+4+_]).
minCost(example,f,[f+0+_, g+2+_, a+3+_, c+4+_, b+6+_, d+7+_, i+9+_, j+11+_,
                   h+12+_, e+13+_]).
minCost(example,g,[g+0+_, a+1+_, c+2+_, b+4+_, d+5+_, i+7+_, j+9+_, h+10+_,
                   e+11+_, f+13+_]).
minCost(example,h,[h+0+_, e+1+_, i+4+_, j+5+_, c+13+_, b+15+_, d+16+_]).
minCost(example,i,[i+0+_, h+3+_, e+4+_, j+8+_, c+9+_, b+11+_, d+12+_]).
minCost(example,j,[j+0+_, e+4+_]).

minCost(example2,a,[e+(-2)+_,a+0+_,b+2+_,c+4+_,d+7+_]).
minCost(example2,b,[e+(-4)+_,a+(-2)+_,b+0+_,c+2+_,d+5+_]).
minCost(example2,c,[e+(-6)+_,a+(-4)+_,b+(-2)+_,c+0+_,d+3+_]).
minCost(example2,d,[e+(-9)+_,a+(-7)+_,b+(-5)+_,c+(-3)+_,d+0+_]).
minCost(example2,e,[e+0+_,a+2+_,b+4+_,c+6+_,d+9+_]).

graphNodes(example,[a,b,c,d,e,f,g,h,i,j]).
graphNodes(example2,[a,b,c,d,e]).

:- test(example).
:- test(example2).
:- graph(example3,G),bellmanford(G,b,negative_cycle).
```

Prolog will reply 'Yes' to the `test/1` predicates if the solutions returned by your implementation are correct for the chosen graph.

6 Deliverables and Deadlines

You should submit a single Prolog source file named `CRSID-prolog10.pl` (replace CRSID with your CRSID). This file should contain all the clauses above along with appropriate tests. The file should compile

and load in SWI-Prolog without errors, warnings about singleton variables, or failed clauses. For the avoidance of doubt: your code is expected to work correctly on the SWI-Prolog version running on PWF Linux.

Email your submission to `prolog-tick@cl.cam.ac.uk` .

Examination will take the form of a visual inspection of your source code, a test using different graphs to those in the examples above, and an oral examination. Your oral viva examination will last for seven and a half minutes and you will be expected to explain the functioning of your code and resolve any issues that are raised by your examiner. Ensure that you have re-familiarised yourself with your submission prior to attending your exam. You will be told at the end of your viva whether you have passed your tick.

6.1 Important Dates

Viva sign-up sheets placed outside Student Administration in the William Gates Building. Write your CRSID in an empty slot.	Fri 21-Jan-2011 12:00 noon
Submission deadline for your tick (by email)	Fri 28-Jan-2011 12:00 noon
Viva sign-up sheets taken down	Fri 28-Jan-2011 12:00 noon
Viva Examinations	Thu 10-Feb-2011 13:00-16:00
Viva Examinations	Fri 11-Feb-2011 13:00-16:00

6.2 Tick Checklist

In order to achieve your tick you must have achieved the following:

1. Implement and test the clauses described above providing comments where requested;
2. Your submitted code must pass visual inspection and a further test on a different example graph;
3. Sign up for a viva examination before Friday 28-Jan-2011 12:00 noon;
4. Submit your tick by email before Friday 28-Jan-2011 12:00 noon;
5. Attend your examination and answer questions about your submission to the examiner's satisfaction: *be prepared and punctual*.

6.3 Alternative: C & C++ Assessed Exercise

You need only complete either the Prolog tick or the C & C++ tick but you may complete both if you wish. No further examination credit is available for completing both ticks. The examination procedure for the C & C++ tick is of the same form as the above and will run concurrently with the Prolog tick examinations.

END OF TICK