Prolog Assessed Exercise

Andrew Rice <andrew.rice@cl.cam.ac.uk>

November 20, 2007

The purpose of this exercise is to develop an implementation of Dijkstra's Shortest Path algorithm in Prolog. 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. Details about submission and marking are at the end of this document. This document consists of five pages.

1 Basic Operations

The shortest path algorithm maintains lists of pairs. Each pair represents a node in the graph and the cost of reaching it. We will represent them in the form `V-D` where `V` is the graph node and `D` is the cost. Begin your source file with these auxiliary functions for checking the equality of nodes and for comparing path lengths:

```
eq(A-_,A-_).
ne(A-_,B-_) :- A \= B.
lt(_-D1,_-D2) :- D1 < D2.
ge(_-D1,_-D2) :- D1 >= D2.
```

Include each of these in your source file with a brief explanatory comment.

2 Supporting Predicates

2.1 List Minimum

Write a predicate `min(+L,-Min,-Rest)` which takes a list `L` and unifies `Min` with the smallest value in `L` (according to `lt/2`) and `Rest` with a list containing all values in `L` excluding `Min`. Your implementation should use an accumulator and be amenable to last-call optimisation. It is not necessary to preserve the original order of `L` in `Rest`.

Include some test cases in your source file. Your tests should be executed when the file is loaded and should show that the predicate can generate the values of `Min` and `Rest` as well as check they are correct.

2.2 List Difference

The following predicate `diff(+L,+M,-N)` takes a list `L` and a list `M` and unifies `N` with a list containing all items in `L` which are not in `M`. The list `L` is assumed to contain no duplicate elements.

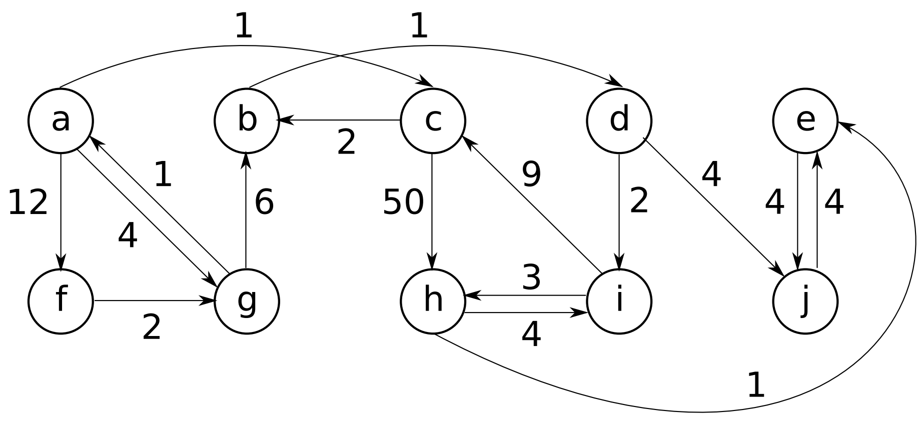

Figure 1: Example graph

```
diff(L,M,R) :- diff(L,M,M,R).
diff([],_,_,[]).
diff([H|T],[],M,[H|Rest]) :- diff(T,M,M,Rest).
diff([H|T],[H2|_],M,Rest) :- eq(H,H2),!,diff(T,M,M,Rest).
diff([H|T],[H2|T2],M,Rest) :- ne(H,H2),diff([H|T],T2,M,Rest).
```

Write a comment describing the operation of this predicate and include it and some test cases in your source file.

2.3 List Merge

Write a predicate `nodeMerge(+A,+B,-C)` which merge lists `A` and `B` together and unifies the result in `C`. Lists `A`, `B` and `C` contain no duplicate elements: i.e. within the each list there is no pair of nodes `V1`,`V2` such that `eq(V1,V2)`. You may assume that this is true of lists `A` and `B` and must ensure that its true for list `C`. For those vertices which feature in both list `A` and `B` your algorithm should include the vertex with the lowest distance i.e. given vertices `V1` (from `A`) and `V2` (from `B`) such that `eq(V1,V2)` you should include `V1` in `C` if `lt(V1,V2)` or `V2` in `C` if `ge(V1,V2)`. You should assume that lists `A` and `B` are arbitrarily ordered, it is not necessary to preserve this order in `C`.

Include your predicate in your source file along with some test cases and comments describing its operation.

3 Representing Graphs

We encode the graph which 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 which 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.

3.1 Start Nodes

Write a rule `node(+Graph,-Node)` which 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. 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 `node/2` 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 the built-in `member(E,L)` function which is true if `E` is an element of `L`, and the not operator.

3.2 Adjacent Nodes

Write a predicate `adjacent(+Graph,+Node,-Adj)` which is true if `Adj` is a list of all nodes reachable from the node `Node` in one hop in `Graph`. `Node` is of the form of a single graph vertex label (i.e. the minimum cost is not included), `Adj` should be a list of node-cost pairs.

For example, given the query adjacent([a-b-1,a-c-2,b-a-4,b-c-1],a,A) Prolog should return A = [b-1,c-2] *and no further results*.

Include your predicate in your source file along with some test cases.

3.3 Increasing Distance

Write a rule `addDist(+Nodes,+D,-NewNodes)` which is true if `NewNodes` is a list containing all the nodes in `Nodes` with their edge cost incremented by `D`.

4 Dijsktra's Shortest Path Algorithm

The operation of Dijsktra's shortest path algorithm is described in detail in the Algorithms II course. Students might also consult Wikipedia (search for "Dijkstra's algorithm"), and also view Carla Laffra's Java applet visualiser.[1]

We implement it in Prolog as follows:

```
dijkstra(Graph,MinDist,[],MinDist).
dijkstra(Graph,Closed,Open,MinDist):-
  min(Open,V-D,ReducedOpen),
  adjacent(Graph,V,AdjacentNodes),
  diff(AdjacentNodes,Closed,PrunedNodes),
  addDist(PrunedNodes,D,UpdatedPrunedNodes),
  nodeMerge(UpdatedPrunedNodes,ReducedOpen,NextOpen),
  dijkstra(Graph,[V-D|Closed],NextOpen,MinDist).
```

Include the above code in your source file with comments explaining the purpose of the various variables. Add an additional clause `dijkstra(+Graph,+Start,-MinDist)` which calculates the minimum distances from the node `Start` by querying the `dijkstra/4` predicate. Hint: you need to decide on a suitable value for the initial `Open` list.

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.

[1] `http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html`

5 Test Case

Append the following test clauses to your file:

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).

test(Name,Start) :-
  graph(Name,G),
  node(G,Start),
  dijkstra(G,Start,L),
  minCost(Name,Start,L2),
  perm(L,L2).

test(Name) :- findall(S,test(Name,S),L), graphNodes(Name,L2), perm(L,L2).

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

:- test(example).
```

Prolog will reply 'Yes' to the test/1 predicate 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-prolog07.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 SWI-Prolog 5.6.46 (the current stable release) 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 a different graph to your example above and an oral examination. Your oral viva examination will last for 7 and a half minutes and you will be expected to explain the functioning of your code and resolve any issues turned up 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 18-Apr-2008 12:00 noon
Submission deadline for your tick (by email)	Fri 25-Apr-2008 12:00 noon
Viva sign-up sheets taken down	Fri 25-Apr-2008 12:00 noon
Viva Examinations	Thu 8-May-2008 13:00 to 16:00
Viva Examinations	Fri 9-May-2008 13:00 to 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 25-Apr-2008 12:00 noon;
4. Submit your tick by email before 25-Apr-2008 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