Prolog Assessed Exercise

David Eyers <david.eyers@cl.cam.ac.uk>

2010-01-08 11:38:11Z

The purpose of this exercise is to implement Prolog predicates that explore the implications of the Four Colour Theorem[1]. Informally described, the Four Colour Theorem states that no more than four colours are required to colour each region in a planar map, so that no regions with sections of shared border (excepting single points of contact) are filled with the same colour. Familiar planar maps include geographical maps showing countries, states or counties (not counting where such land areas are disjoint). 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 eight pages.

[Aside: Note that the code below can often be shortened through the use of higher-order functions common within functional programming, such as "fold left" and "map", but that they are not covered in this course.]

1 Background

Figure 1 indicates the dual between regions on a map, and a graph showing nodes whose interconnections indicate map regions that have a shared boundary. The colouring algorithms explored in this assessed exercise operate on the graph representation.

During their operation, our colouring algorithms will need to maintain a number of items of state. There will be a representation of the graph being examined (described below). We are aiming to build a set of assignments of colours to nodes: this will be the output of any valid predicate implementation that colours a map. For this exercise, each item of the colouring result set will be a term of the form `N=C` where `N` is a node from the graph being examined, and `C` is a colour. Since each `N` should only occur once, the resulting set can be considered to be a form of a key-value dictionary. We will use the four colour atoms: `red`, `green`, `blue` and `yellow`. Note that we use Prolog lists to represent sets.

The initial colouring algorithm uses an inefficient search mechanism. We refine a subsequent predicate that uses constraints more intelligently: that is, if you have assigned `red` to a node, then `red` can be removed from the possible assignments of all of its neighbouring nodes. This will require maintaining the set of nodes, and coupled with each of those nodes, the remaining set of colours from which that node can take on a value without violating local colour constraints.

2 Supporting Predicates

2.1 Set comparison

While running the colouring algorithms on a graph, we will not assume that the elements in any of the internal lists are in any particular order. For ease of visual examination and comparison or results, however,

[1] Wikipedia has some starting points: `http://en.wikipedia.org/wiki/Four_color_theorem`

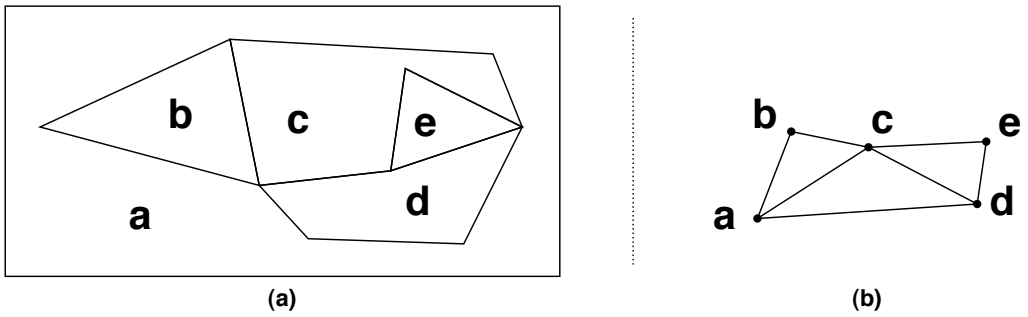

Figure 1: Representing a map in part (a) as regions on a plane and equivalently in part (b) using nodes to represent regions, and edges to indicate the regions that have common borders.

we will define a standard ordering for the data: we will order the colouring elements by their node name, i.e. the first argument of an `N=C` expression. Note that a node colouring list is invalid if it contains two elements of the form `N=C` with the same `N`. You do not need to check for this case.

Write a predicate `lessthan(+A,+B)` that is true iff[2] the colour assignment `A` is "smaller than" the colour assignment `B` for the notion of colour assignment introduced above. Your predicate should fail if either or both of its arguments are not colour assignments. 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.

You must 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. You must include at least one negative test.

2.2 Merge sort

We will use the merge sort algorithm to effect the standardised ordering of colours discussed above. Include the two predicates `msort_colours/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 `merge_lists/3` predicate in the next step of the assessed exercise.

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

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

[2] "if and only if"

2.3 Merge sort's list merging predicate

Write a predicate `merge_lists(+X,+Y,-Z)` that takes two lists of colours 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 `merge_lists/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 `merge_lists/3`, you should include some test cases for the `msort_colours/2` predicates in your source file also.

2.4 List search and modification

We will develop a predicate that aims to colour a graph in which some nodes have already had colour sets assigned to them. Write a predicate `get_with_default(+Key,+ListIn,+Default,-Value)` that expects to be presented with an `ListIn` list that contains elements of the form `K=V` (e.g. the colour sets described above). Your predicate should try to unify `Key` with each `K`. If it succeeds, unify the corresponding `V` with `Value`. Otherwise, unify `Default` with `Value`. Include some test predicates, and at least one negative test. This predicate will be used to set up initial colour sets for one of the colouring algorithms below.

Write a predicate `remove(+Item,+ListIn,-ListOut)` that unifies `ListOut` with the contents of `ListIn` except for the first element from `ListIn` that unifies with `Item`. Your predicate should not fail if no such `Item` can be found. If `Item` can be unified with multiple elements of `ListIn`, only the first removal should be performed, and no further answers be offered on backtracking.

3 Representing Graphs

We shall encode the graph that we wish to analyse as a list of (undirected) edges. Each edge consists of a compound term `N1-N2` where `N1` and `N2` are node identifiers within a given graph. Write a predicate `graph(example1,-List)` that unifies `List` with the list that represents the graph shown in part (b) of figure 1. Note that if you include an edge `x-y`, then you need not, and should not, include an edge `y-x`. We use the additional argument `example1` to allow us to run our program with different graphs later.

3.1 Graph 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 Graph Colouring

We first develop an inefficient, but nonetheless brief and correct, implementation of a colouring algorithm.

```
% colour_graph_simple(+Graph,-Colours).
colour_graph_simple(Graph,Colours) :-
    nodes(Graph,Nodes),
    colour_nodes(Nodes,Colours),
    check_colours(Graph,Colours).

% colour_nodes(+Nodes,-Colours)
% to be completed...

% check_colours(+Graph,-Colours).
check_colours([],_).
check_colours([N1-N2|Es],Colours) :-
    member(N1=C1,Colours),
    member(N2=C2,Colours),
    C1 \= C2, % equivalent to not(C1=C2)
    check_colours(Es,Colours).
```

Copy and paste the above Prolog code into your source file, and write the `colour_nodes/2` predicate. This predicate is given, in `Nodes`, a list containing all of the graph nodes. The `colour_nodes/2` predicate should bind `Colours` to a list containing an element of the form `Node=Colour` for each node of the graph. Backtracking should provide every possible assignment of colours to nodes—including colour assignments that violate the requirement that adjacent nodes have different colours. You are writing the generator in this generate-and-test implementation—in this case `check_colours/2` is the test predicate.

Include a few simple tests for the `check_colours` and `colour_nodes` predicates, with at least one negative test for `check_colours`.

4.1 Initial colour sets, and constraint annotation

The above approach to graph colouring does a poor job of tracking constraints. Whenever a colour is applied to a node, a constraint can easily be recorded to avoid that colour being used in neighbouring nodes. We will implement a predicate that tracks the set of colours that a node can potentially take on, and updates this set when this node's neighbours have colours assigned to them.

[Aside: as discussed in lectures, SWI-Prolog provides support for Constraint Logic Programming—indeed most large-scale Prolog distributions do. In this assessed exercise, you should maintain the constraints directly. An efficient and complete solution to the colouring problem can be achieved in about the same amount of code as the `colour_graph_simple/2` predicate above. Feel free to verify this...]

Encode the Australian states into a colouring graph. The map is shown in figure 2, and should be encoded into the predicate `graph(example2,-List)`. Note that you will need to surround node names with single quotes in order to form atoms that begin with a capital letter. The node names should be precisely as labelled on the figure. Note that the Australian Capital Territory (`ACT`) is contained within New South Wales (`NSW`). You do not need to consider any off-shore region smaller than Tasmania (`TAS`), but you should encode the surrounding `water` as a node.

Copy and paste the code below into your source file. Note that elements of `ColourAssignments` are of the form `'NSW'=red`, while elements of `GivenColourSets` are of the form `'NSW'=[red]`. The predicate `flatten_colour_lists` below can do bi-directional conversion of these forms (if possible).

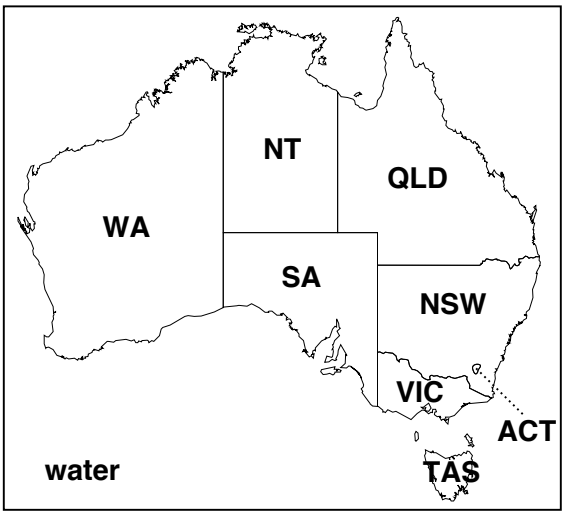

Figure 2: A map of the states of Australia

```
%% colour_graph(+Graph,+GivenColourSets,-ColourAssignments)
colour_graph(Graph,GivenColourSets,ColourAssignments) :-
    % determine the graph nodes
    nodes(Graph,Nodes),
    % initialise the nodes' initial colour sets
    initial_colour_sets(Nodes,GivenColourSets,ColourSets),
    % label each of the graph nodes with a colour
    colour_nodes(Graph,Nodes,ColourSets,ColoursFinished),
    % reorganise the terms in the colour assignments slightly
    flatten_colour_lists(ColoursFinished,ColourAssignments).

flatten_colour_lists([],[]).
flatten_colour_lists([N=[C]|RestIn],[N=C|RestOut]) :-
    flatten_colour_lists(RestIn,RestOut).

initial_colour_sets([],_,[]).
initial_colour_sets([N|Ns],GivenColourSets,[N=Value|ColourSets]) :-
    get_with_default(N,GivenColourSets,[red,green,blue,yellow],Value),
    initial_colour_sets(Ns,GivenColourSets,ColourSets).

colour_nodes(_,[],ColourSets,ColourSets).
colour_nodes(Graph,[Node|Ns],ColourSetsIn,ColourSetsOut) :-
    choose_a_node_colour(Node,Colour,ColourSetsIn,ColourSets2),
    remove_colour_from_connected_sets(Graph,Node,Colour,
                                      ColourSets2,ColourSets3),
    colour_nodes(Graph,Ns,ColourSets3,ColourSetsOut).
```

You need to write the predicate `choose_a_node_colour(+Node, -Colour, +ColourSetsIn, -ColourSetsOut)`. Unlike the `colour_nodes` predicate that you implemented earlier, this new predicate is instructed which `Node` to look up in the provided colour sets (`ColourSetsIn`). `Colour` will be unified only with elements in the given node's colour set. On backtracking, each available colour in the node's colour set should be selected. By selecting a colour for `Node`, you are effectively changing its colour set to a singleton element. This change must be reflected in `ColourSetsOut`.

Having selected a colour for a node `Node`, the colour is removed from all of the colour sets of the nodes

directly connected to that `Node` by `remove_colour_from_connected_sets/5`. Copy and paste the following code into your source file, with a comment as to where cuts can be employed, and why.

```
remove_colour_from_connected_sets([],_,_,ColourSets,ColourSets).
remove_colour_from_connected_sets([N-ON|Es],N,Colour,CSs,NewCSs) :-
    remove_colour_from_specific_set(Colour,ON,CSs,CSs2),
    remove_colour_from_connected_sets(Es,N,Colour,CSs2,NewCSs).
remove_colour_from_connected_sets([ON-N|Es],N,Colour,CSs,NewCSs) :-
    remove_colour_from_specific_set(Colour,ON,CSs,CSs2),
    remove_colour_from_connected_sets(Es,N,Colour,CSs2,NewCSs).
remove_colour_from_connected_sets([ON1-ON2|Es],N,Colour,CSs,NewCSs) :-
    N \= ON1, N \= ON2,
    remove_colour_from_connected_sets(Es,N,Colour,CSs,NewCSs).
```

The `remove_colour_from_specific_set(+Colour,+Node,+CS1,-CS2)`[3] predicate finds the colour set corresponding to node `Node` within `CS1`, removes `Colour` from it, and binds the result with `CS2`. One of the clauses of this predicate is reproduced below, for you to copy and paste into your source file. Complete the implementation of this predicate.

```
remove_colour_from_specific_set(Colour,TargetNode,
                [OtherNode=CS|ColourSets],
                [OtherNode=CS|NewColourSets]) :-
    TargetNode \= OtherNode,
    remove_colour_from_specific_set(Colour,TargetNode,
                ColourSets,NewColourSets).
```

Finally, create a graph with at least five nodes, encoded into `graph(example3,-List)` in the usual way. Demonstrate the `colour_graph` predicate successfully colouring it. Also demonstrate a situation in which the initial colour sets preclude a four colour solution being found. It may be more efficient to defer this step until after you achieve your code passing the tests provided in the next section.

5 Test Cases

Append the following test clauses to your file.

```
% test that an arbitary valid (and invalid) colouring is (and is not) proved.
test(t1):-t_exists(example1,t_s1,[a=red,b=green,c=blue,d=green,e=red]).
test(t2):-not(t_exists(example1,t_s1,[a=red,b=green,c=blue,d=green,e=blue])).
test(t3):-t_exists(example1,t_c1,[a=red,b=green,c=blue,d=green,e=red]).
test(t4):-not(t_exists(example1,t_c1,[a=red,b=green,c=blue,d=green,e=blue])).

% test that right numbers of solutions are discovered
test(s1n):-graph(example1,G),numsol( colour_graph_simple(G,_), 96).
test(c1n):-graph(example1,G),numsol( colour_graph(G,[],_), 96).
test(c2n):-graph(example2,G),numsol( colour_graph(G,[],_), 216).

% test one complete solution for each given example
test(c2s):-graph(example2,G),findall(CSorted,(colour_graph(G,
    [water=[blue],'NSW'=[green],'VIC'=[red],'TAS'=[green]],C),
       msort_colours(C,CSorted)),L),perm(L,
    [['ACT'=red,'NSW'=green,'NT'=green,'QLD'=red,'SA'=yellow,'TAS'=green,
```

[3] I note that in my own code, I would not tend to usually employ predicate names that are this long!

```
      'VIC'=red,'WA'=red,water=blue],['ACT'=blue,'NSW'=green,'NT'=green,
      'QLD'=red,'SA'=yellow,'TAS'=green,'VIC'=red,'WA'=red,water=blue],
      ['ACT'=yellow,'NSW'=green,'NT'=green,'QLD'=red,'SA'=yellow,
      'TAS'=green,'VIC'=red,'WA'=red,water=blue]]).

test(c1s):-graph(example1,G),findall(CSorted,(colour_graph(G,
    [a=[red],b=[blue]],C),msort_colours(C,CSorted)),L), perm(L,
      [[a=red,b=blue,c=green,d=blue,e=red],[a=red,b=blue,c=green,d=blue,
        e=yellow],[a=red,b=blue,c=green,d=yellow,e=red],[a=red,b=blue,
        c=green,d=yellow,e=blue],[a=red,b=blue,c=yellow,d=green,e=red],
        [a=red,b=blue,c=yellow,d=green,e=blue],[a=red,b=blue,c=yellow,
        d=blue,e=red],[a=red,b=blue,c=yellow,d=blue,e=green]]).

t_exists(GraphName,Predicate,ValidColouring):-
    graph(GraphName,Graph),
    call(Predicate,Graph,ColouringOut),
    msort_colours(ColouringOut,ValidColouring).
t_s1(Graph,ColouringOut):-colour_graph_simple(Graph,ColouringOut).
t_c1(Graph,ColouringOut):-colour_graph(Graph,[],ColouringOut).
t_c2(Graph,ColouringOut):-colour_graph(Graph,[water=[blue]],ColouringOut).

% Helper predicates from lectures
numsol(Predicate,NumberOfSolutions) :-
    findall(dummy,Predicate,AnsList),
    length(AnsList,NumberOfSolutions).
perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

run_tests_quiet :- foreach( clause(test(TestID),_), test(TestID) ),!.
run_tests_quiet :- run_tests,fail.

run_tests :- foreach( clause(test(TestID),_), run_test(TestID) ).
run_test(TestID) :-
    ( test(TestID) -> Result = succeeded; Result = failed ),
    format("Test ~w: ~w.~n",[TestID,Result]).

:-run_tests_quiet.
```

Prolog will run the above tests whenever the file is reloaded, and if any tests fail, will re-run the tests as well as outputting which pass and which fail on the terminal. Note that you can query specific `test/1` predicates in the Prolog interpreter.

5.1 Colouring an SVG file (entirely optional!)

This subsection is for your interest only, and can be skipped with no impact on your assessment whatsoever. The assessed exercise instructions continue at section 6.

From `http://www.cl.cam.ac.uk/~dme26/pl/svg-server.pl` you can download the source file `svg-server.pl`. This file provides predicates that apply your graph colouring routines to a Scalable Vector Graphics (SVG)[4] file and serve it up to your web browser of choice (not including Microsoft Internet Explorer, unless you install SVG plug-ins) care of a multi-threaded, Prolog-driven HTTP server (!).

Since `svg-server.pl` is not part of the tick exercise, it will be re-released whenever things need fixing (i.e. mostly when you tell me things are broken), and should provide sufficient information when you load

[4] See `http://www.w3.org/Graphics/SVG/` and `http://en.wikipedia.org/wiki/SVG`

the module as to its intended use. It employs Prolog language features not discussed in the course, or covered only briefly, but if they interest you, please do feel free to ask for more details.

6 Deliverables and Deadlines

You should submit a single Prolog source file named `CRSID-prolog09.pl` (replace CRSID with your CRSID). This file should contain all of the clauses discussed above, along with appropriate tests. The file should compile and load in SWI-Prolog without errors, warnings about singleton variables, or failed clauses. 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 15-Jan-2010 12:00 noon
Submission deadline for your tick (by email)	Fri 22-Jan-2010 12:00 noon
Viva sign-up sheets taken down	Fri 22-Jan-2010 12:00 noon
Viva Examinations	Thu 4-Feb-2010 13:00 to 16:00
Viva Examinations	Fri 5-Feb-2010 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 Friday 22-Jan-2010 12:00 noon;
4. Submit your tick by email before Friday 22-Jan-2010 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