99 questions/80 to 89

From HaskellWiki

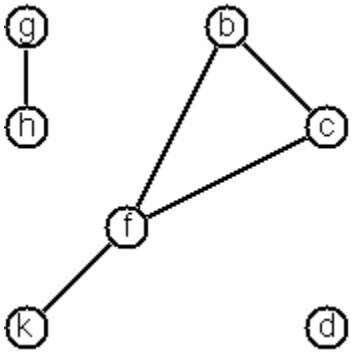
< 99 questions

This is part of Ninety-Nine Haskell Problems, based on Ninety-Nine Prolog Problems (https://prof.ti.bfh.ch/hew1/informatik3/prolog/p-99/) .

If you want to work on one of these, put your name in the block so we know someone's working on it. Then, change n in your block to the appropriate problem number, and fill in the <Problem description>,<example in lisp>,<example in Haskell>,<solution in haskell> and <description of implementation> fields.

1 Graphs

A graph is defined as a set of nodes and a set of edges, where each edge is a pair of nodes.



There are several ways to represent graphs in Prolog. One method is to represent each edge separately as one clause (fact). In this form, the graph depicted below is represented as the following predicate:

```
edge(h,g).
edge(k,f).
edge(f,b).
...
```

We call this *edge-clause* form. Obviously, isolated nodes cannot be represented. Another method is to represent the whole graph as one data object. According to the definition of the graph as a pair of two sets (nodes and edges), we may use the following Prolog term to represent the example graph:

```
graph([b,c,d,f,g,h,k],[e(b,c),e(b,f),e(c,f),e(f,k),e(g,h)])
```

We call this *graph-term* form. Note, that the lists are kept sorted, they are really *sets*, without duplicated elements. Each edge appears only once in the edge list; i.e. an edge from a node x to another node y is represented as e(x,y), the term e(y,x) is not present. **The graph-term form is our default representation.** In SWI-Prolog there are predefined predicates to work with sets.

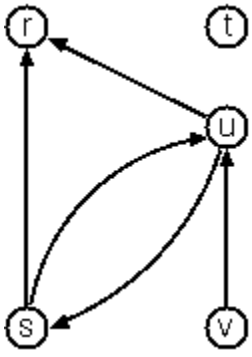
A third representation method is to associate with each node the set of nodes that are adjacent to that node. We call this the *adjacency-list* form. In our example:

```
[n(b,[c,f]), n(c,[b,f]), n(d,[]), n(f,[b,c,k]), ...]
```

The representations we introduced so far are Prolog terms and therefore well suited for automated processing, but their syntax is not very user-friendly. Typing the terms by hand is cumbersome and error-prone. We can define a more compact and "human-friendly" notation as follows: A graph is represented by a list of atoms and terms of the type X-Y (i.e. functor '-' and arity 2). The atoms stand for isolated nodes, the X-Y terms describe edges. If an X appears as an endpoint of an edge, it is automatically defined as a node. Our example could be written as:

```
[b-c, f-c, g-h, d, f-b, k-f, h-g]
```

We call this the *human-friendly* form. As the example shows, the list does not have to be sorted and may even contain the same edge multiple times. Notice the isolated node d. (Actually, isolated nodes do not even have to be atoms in the Prolog sense, they can be compound terms, as in `d(3.75,blue)` instead of `d` in the example).



When the edges are directed we call them arcs. These are represented by ordered pairs. Such a graph is called **directed graph**. To represent a directed graph, the forms discussed above are slightly modified. The example graph above is represented as follows:

Arc-clause form

```
arc(s,u).
arc(u,r).
...
```

Graph-term form

```
digraph([r,s,t,u,v],[a(s,r),a(s,u),a(u,r),a(u,s),a(v,u)])
```

Adjacency-list form

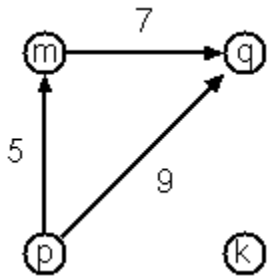
```
[n(r,[]),n(s,[r,u]),n(t,[]),n(u,[r]),n(v,[u])]
```

Note that the adjacency-list does not have the information on whether it is a graph or a digraph.

Human-friendly form

```
[s > r, t, u > r, s > u, u > s, v > u]
```

Finally, graphs and digraphs may have additional information attached to nodes and edges (arcs). For the nodes, this is no problem, as we can easily replace the single character identifiers with arbitrary compound terms, such as `city('London',4711)`. On the other hand, for edges we have to extend our notation. Graphs with additional information attached to edges are called **labelled graphs**.



Arc-clause form

```
arc(m,q,7).
arc(p,q,9).
arc(p,m,5).
```

Graph-term form

```
digraph([k,m,p,q],[a(m,p,7),a(p,m,5),a(p,q,9)])
```

Adjacency-list form

```
[n(k,[]),n(m,[q/7]),n(p,[m/5,q/9]),n(q,[])]
```

Notice how the edge information has been packed into a term with functor '/' and arity 2, together with the corresponding node.

Human-friendly form

```
[p>q/9, m>q/7, k, p>m/5]
```

The notation for labelled graphs can also be used for so-called **multi-graphs**, where more than one edge (or arc) are allowed between two given nodes.

2 Problem 80

(***) Conversions

Write predicates to convert between the different graph representations. With these predicates, all representations are equivalent; i.e. for the following problems you can always pick freely the most convenient form. The reason this problem is rated (***) is not because it's particularly difficult, but because it's a lot of work to deal with all the special cases.

Example:

```
<example in lisp>
```

Example in Haskell:

```
graphToAdj Graph ['b','c','d','f','g','h','k'] [('b','c'),('b','f'),('c','f'),('f','k'),('g',
Adj [('b', "cf"), ('c', "bf"), ('d', ""), ('f', "bck"), ('g', "h"), ('h', "g"), ('k', "f")]
```

Solutions

3 Problem 81

(**) Path from one node to another one

Write a function that, given two nodes a and b in a graph, returns all the acyclic paths from a to b.

Example:

```
<example in lisp>
```

Example in Haskell:

```
paths 1 4 [(1,2),(2,3),(1,3),(3,4),(4,2),(5,6)]
[[1,2,3,4],[1,3,4]]
paths 2 6 [(1,2),(2,3),(1,3),(3,4),(4,2),(5,6)]
[]
```

Solutions

4 Problem 82

(*) Cycle from a given node

Write a predicate cycle(G,A,P) to find a closed path (cycle) P starting at a given node A in the graph G. The predicate should return all cycles via backtracking.

Example:

```
<example in lisp>
```

Example in Haskell:

```
graph> cycle 2 [(1,2),(2,3),(1,3),(3,4),(4,2),(5,6)]
[[2,3,4,2]]
graph> cycle 1 [(1,2),(2,3),(1,3),(3,4),(4,2),(5,6)]
[]
```

Solutions

5 Problem 83

(**) Construct all spanning trees

Write a predicate s_tree(Graph,Tree) to construct (by backtracking) all spanning trees of a given graph. With this predicate, find out how many spanning trees there are for the graph depicted to the left. The data of this example graph can be found in the file p83.dat. When you have a correct solution for the s_tree/2 predicate, use it to define two other useful predicates: is_tree(Graph) and is_connected(Graph). Both are five-minutes tasks!

Example:

```
<example in lisp>
```

Example in Haskell:

```
length $ spantree k4
16
```

Solutions

6 Problem 84

(**) Construct the minimal spanning tree

Write a predicate ms_tree(Graph,Tree,Sum) to construct the minimal spanning tree of a given labelled graph. Hint: Use the algorithm of Prim. A small modification of the solution of P83 does the trick. The data of the example graph to the right can be found in the file p84.dat.

Example:

```
<example in lisp>
```

Example in Haskell:

```
<example in Haskell>
```

Solutions

7 Problem 85

(**) Graph isomorphism

Two graphs G1(N1,E1) and G2(N2,E2) are isomorphic if there is a bijection f: N1 -> N2 such that for any nodes X,Y of N1, X and Y are adjacent if and only if f(X) and f(Y) are adjacent.

Write a predicate that determines whether two graphs are isomorphic. Hint: Use an open-ended list to represent the function f.

Example:

```
<example in lisp>
```

Example in Haskell:

```
iso graphG1 graphH1
True
```

Solutions

8 Problem 86

(**) Node degree and graph coloration

a) Write a predicate degree(Graph,Node,Deg) that determines the degree of a given node.

b) Write a predicate that generates a list of all nodes of a graph sorted according to decreasing degree.

c) Use Welch-Powell's algorithm to paint the nodes of a graph in such a way that adjacent nodes have different colors.

Example:

```
<example in lisp>
```

Example in Haskell:

```
chromatic $ kcolor petersen
3
```

Solutions

9 Problem 87

(**) Depth-first order graph traversal (alternative solution)

Write a predicate that generates a depth-first order graph traversal sequence. The starting point should be specified, and the output should be a list of nodes that are reachable from this starting point (in depth-first order).

Example:

```
<example in lisp>
```

Example in Haskell:

```
<example in Haskell>
```

Solutions

10 Problem 88

(**) Connected components (alternative solution)

Write a predicate that splits a graph into its connected components.

Example:

```
<example in lisp>
```

Example in Haskell:

```
<example in Haskell>
```

Solutions

11 Problem 89

(**) Bipartite graphs

Write a predicate that finds out whether a given graph is bipartite.

Example:

```
<example in lisp>
```

Example in Haskell:

```
<example in Haskell>
```

Solutions

Category:

- Tutorials