Course Project: Graph Representation and Traversal

Functional Programming

due: 2024-05-14

A *graph* is a data structure made up of *vertices* (or "nodes") representing objects and *edges* representing relations between objects. There are several varieties of graph structures. For concreteness, in the following we will be discussing graphs in which,

- edges are directed, so an edge from x to y is distinct from an edge from y to x,
- a given ordered pair of vertices may be connected by any number of edges,
- a vertex may be connected by edges to itself.

There are several possible ways to represent the data structure of a graph. Perhaps one of the simplest is as a list of edges, where each edge has a source vertex, a target vertex, and a collection of attribute-value pairs. There is a standard syntax for representing graphs in this way called *DOT*. It is the graph description language for a program called *Graphviz*, which can render graph descriptions as diagrams. The specification of the DOT language can be found at `https://graphviz.org/doc/info/lang.html`.

For this project you will use Idris to implement a data structure to represent graphs, functions to read and write them from and to strings and files, and an algorithm to efficiently traverse them.

These tasks are intentionally less precisely specified than those in the labs and homeworks. You will need to use good judgement and programming practice to decompose each task into parts (types, functions, interfaces, etc.) that work together in a logical, modular, efficient, and elegant way to solve the problem at hand.

This leaves you with quite a lot of freedom, and you may take any approach you like provided that it conforms to the specifications and the spirit of the assignment. For example, it would violate the spirit of the assignment if you were to find an already implemented solution online, import it, and simply call the relevant functions. Thus, you must obtain permission from the instructor if you wish to use modules other than those found in the standard library.

Your solution to each of the following tasks should include a comment in your source file specifying the task number and briefly explaining the structure of your solution. If there are some parts of your program that are likely to be confusing, are unimplemented, or that contain known bugs, you should explain them in comments as well. In short, use good coding practice so that readers of your program (including your future self) can understand as easily as possible how it works. In any event, your submitted source file(s) should load without syntax or type errors.

Submit your project by pushing it to your course git repository in a directory called `project`. Your submission should include a plain-text `README` file containing any information needed to load and run your program as intended. Your solutions will be evaluated based on both correctness and programming style. While there is no expectation of optimality in terms of concision and efficiency, your program should be written clearly, making use of concepts such as interfaces and higher-order functions where appropriate, and should not duplicate functionality needlessly.

Task 1
Your first task is to write a function to read a graph description in a simplified version of the DOT language and construct the corresponding graph object if the description is well-formed:

```
read_graph  :  String -> Maybe Graph
```

You may represent a `Graph` any way you wish, but using a `List` of edges will probably be easiest at this stage. The simplified DOT language that you will need to support is for representing directed graphs using only edge specifiers. It is summarized here:

- The description should begin with the string literal `digraph` followed by an open brace `{` and end with a close brace `}`.
- Between the braces there should be a semicolon-separated sequence of edge specifiers.
- Each edge specifier should consist of an alphanumeric source node name and target node name, separated by an ASCII arrow `->` and followed by an optional attribute list.
- Each attribute list should be enclosed between brackets `[` and `]` and contain a comma-separated sequence of key–value pairs, themselves separated by an equals sign `=`.
- Whitespace, including spaces, tabs and linebreaks, is permitted but not required before or after any of the above-mentioned elements.

Here is a small example of a graph description in this simplified DOT language:

```
digraph
{
  a -> b [label = f , cost = 6] ;
  b -> c [label = g , cost = 7] ;
  c -> d [label = h , cost = 8]
}
```

You can see how Graphviz renders this graph by pasting it into the web site `https://edotor.net/`.

Your graph description reader does not need to be able to recognize and reject invalid graph descriptions in the simplified DOT language; however, it should be able to recognize and accept valid graph descriptions.

Next, write a function that reads a simple graph description from a `.dot` file:

```
read_dot_file  :  (path : String) -> IO (Either FileError (Maybe Graph))
```

Task 2
Your next task is to write a function to take a graph in whatever representation you have chosen and write a description of it in the simplified DOT language:

```
write_graph  :  Graph -> String
```

The graph descriptions that your writer emits should be valid Graphviz inputs. At this point you should test your reader and writer to ensure that if you read a valid graph description and then write the graph that you build from it, the result is itself a valid description of the same graph. Note that it need not be the *same* description of this graph, as, for example, the text spacing and order of edges may differ.

Next, write a function that writes a simple graph description to a `.dot` file:

```
write_dot_file  :  Graph -> (path : String) -> IO (Either FileError Unit)
```

Task 3
Your next task is to write a function that takes a graph and a designated "start" node and computes a subgraph of the input graph that includes just those edges that give a shortest path from the start node to each node in the input graph that is reachable from the start node.

```
shortest_paths  :  (start : Node) -> Graph -> Graph
```

This requires a bit of explanation.

A *subgraph* of a given graph consists of a subset of its nodes together with a subset of its edges, subject to the constraint that if an edge is included then so are its boundary nodes. When we represent graphs as edge lists this constraint is satisfied automatically.

A *path* in a graph is a finite sequence of consecutive edges, where two edges are *consecutive* if the target node of the first is the source node of the second. For example, in the graph from task 1 we have a path from `a` to `d` given by `f ; g ; h`.

In order to have a notion of "shortest path" we need a notion of "edge length". This is not necessarily intended to represent a geometric quantity, so we instead call it a *cost*, which may represent distance, time, currency, etc.. The cost of a path is the sum of the costs of its constituent edges. For example, the cost of the path `f ; g ; h` above is 21.

For what follows it is important that costs be non-negative. This is because we will need the cost of a path to be no greater than that of any path that extends it. For example, it should be the case that the cost of the path `f ; g ; h` is no greater than the cost of the path `f ; g ; h ; i`, regardless of what cost of the edge `i` is. In this task you may assume that the `cost` attribute of an edge specifier in a graph description is given as a natural number.

If an edge specifier does not include a `cost` attribute, or if the `cost` attribute of an edge cannot be parsed as a natural number, then we should consider the cost of that edge to be *undefined.* Only edges with well-defined costs should appear in the result of the `shortest_paths` function.

There is simple algorithm for computing the `shortest_paths` function due to Dijkstra. Using the input graph, an "explored" subgraph of it, and a dictionary of node costs for the explored subgraph, we do the following.

- Determine an edge of the input graph whose source node is explored (and thus has known cost), whose target node is unexplored (and thus has unknown cost), and such that the cost of the source node plus the cost of the edge is minimal for edges from an explored node to an unexplored node.
- Add this edge to the explored subgraph and its target node to the cost dictionary, assigning it a cost of the sum of those of the edge and its source node.

The process begins with the explored subgraph containing just the start node and the node cost dictionary assigning the start node a cost of 0. The process ends when we can no longer choose an edge with explored source node, unexplored target node, and well-defined cost. At this point we just return the explored subgraph.

The algorithm works because if at some stage we choose an edge `e : x -> y`, it is because the cost of the node `x` plus the cost of the edge `e` is minimal for all edges from an explored node to an unexplored node. Thus the cost of getting to `y` using `e` can be no greater than the cost of getting to `y` any other way, and we have found a shortest path from the start node to `y`. This process terminates because the graph is finite and at each stage we reduce the size of the unexplored portion by one node.