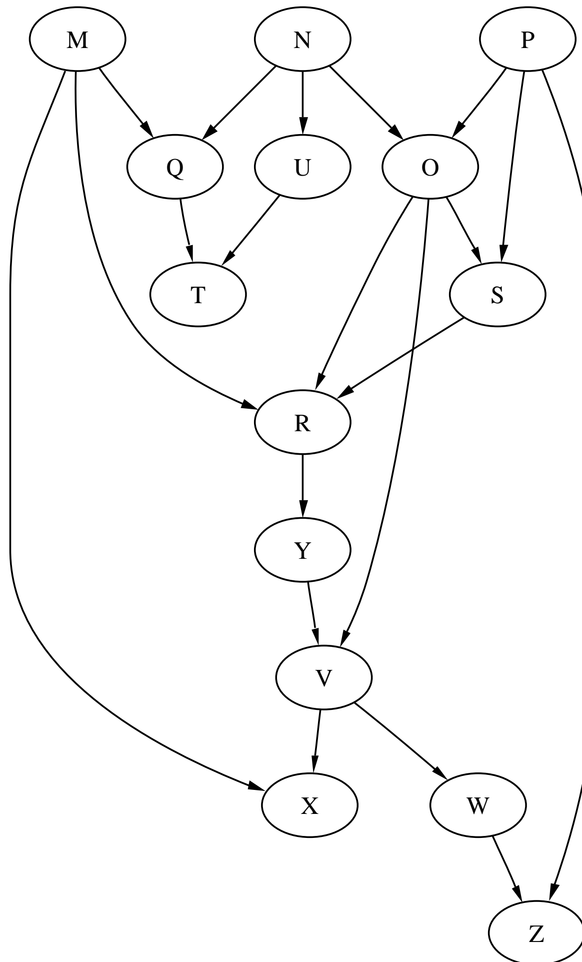
689: Generic Programming — Assignment 3

October 16, 2004

1. Consider the following problem:[1]

   Give a linear time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of paths from s to t in G. For example, in the following graph, there are exactly four paths from vertex P to vertex V: POV, PORYV, POSRYV, and PSRYV. (Your algorithm only needs to count the paths, not list them.)



[1]From T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, MIT Press, 2001, p. 552.

This appears as an exercise in CLRS in a section on the topological sort algorithm. However, actually *using* the topological sorting algorithm is not necessary or desirable, since it is not generally necessary to process the whole graph.

The algorithm for topological sort is DFS with one added operation invoked whenever DFS *finishes* a vertex. The added operation simply prefixes the vertex to a list (or it could append it to the list or output it the output stream, which would produce the result in *reverse topological order* as we did in the class). Counting paths can also be implemented by adding operations to be invoked at certain control points of DFS (not just the finish-vertex control point):

> Add a field to the vertex representation to hold an integer count, or use an auxiliary container to map from vertices to counts. Start running DFS with s as the start vertex. Then, each time DFS *discovers* a vertex, if the vertex is t then set its count to 1 and otherwise to 0. t should also be immediately marked as finished, without further processing starting from it. Subsequently, each time DFS *finishes* a vertex v, set v's count to the sum of the counts of all vertices adjacent to v. When DFS finishes vertex s, stop and return the count computed for s.

Correctness of the algorithm can be established based on the following

- *Invariant:* For every vertex marked as finished, its count field holds the number of paths from that vertex to t (the target vertex). This invariant holds at the control point in the algorithm where a vertex is marked finished.
- *Initialization:* Initially, there are no vertices marked as finished, so the invariant is vacuously true. Note also that when t is discovered, its count is set to 1, which represents the number of paths from t to t (since a single vertex is considered to be a 0-length path).
- *Maintenance:* Whenever a vertex v is marked as finished, any adjacent vertex w must already have been finished, and thus by the invariant the count on w must be the number of paths from w to t. Therefore there are exactly $\sum_{w \in \mathrm{adj}(v)} \mathrm{count}(w)$ paths from v to t, and thus updating $\mathrm{count}(v)$ with this value maintains the invariant. (Note that if v has no adjacent vertices, the sum is taken over an empty set, which by convention is 0.)
- *Termination:* At termination, the invariant implies that for all vertices marked finished, including s, the computed count is the number of paths from that vertex to t.

What is a time bound for the path-counting algorithm?

> We know (1) DFS has an $O(V + E)$ bound, (2) this algorithm does (potentially) less exploration, and (3) the total time for the extra initializations and additions to compute the counts is $O(E)$. So this path-counting algorithm also has an $O(V + E)$ bound.

The assignment

The assignment is to program this solution using the Boost graph library. Do this using some BGL algorithm that provides depth first search functionality. You will need to write a visitor that causes the algorithm to do the extra computation to accomplish the count updating at the appropriate control points of the algorithm and to terminate the algorithm when it is finished with source vertex s.

(a) Encapsulate this computation in a generic algorithm `path_count` with the following interface:

```
template <typename VertexListGraph>
typename graph_traits<VertexListGraph>::edges_size_type
path_count(VertexListGraph& G,
 typename graph_traits<VertexListGraph>::vertex_descriptor source,
 typename graph_traits<VertexListGraph>::vertex_descriptor target)
```

The value returned by `path_count` should be the number of paths from `source` to `target` in `G`.

(b) Write an acceptance test function that takes the same inputs as the algorithm and one additional input, an integer `c`, and checks whether `c` is the number of paths from `source` to `target` in `G`. Although it is best when possible to program an acceptance test by some means other than re-computing the answer by a different algorithm or implementation, it appears that in this case it is necessary to do just that. In such a case, it is safest to compute the answer by an independent algorithm, and one which is programmed as straightforwardly as possible (e.g., by avoiding potential optimizations). In this case, one can compute the answer independently and rather simply using a topological sort followed by computing the counts during a traversal of the resulting linear sequence in reverse.

(c) Write a main program that reads a graph from a file along with two vertices s and t, outputs them in a readable form on the standard output stream, applies `path_count` and outputs the result, and checks the input/output relation with the acceptance test function.

(d) Write another main program that generates a random directed graph and randomly selects vertices s and t, applies `path_count` and outputs the result, and checks the input/output relation with the acceptance test function.

What to turn in

In general, follow the directions from assignment set 1. For this assignment you will need to use the Boost Graph Library, and for that you need to install Boost. I recommend that Boost be installed on path `${HOME}/boost` — that's where my boost directory is — to ensure that I can `make` your programs without extra tweaking. Note that you do not need to actually *build* Boost. You only need to download it, unpack it to the right directory, add that directory to the include path of your compiler, and then include the necessary BGL headers.