CSCE 622: Generic Programming -- Assignment 3

Peihong Guo UIN: 421003404

Introduction

In this assignment a path counting algorithm is implemented using the depth first search functionalities provided in the Boost graph library. By adding a proper visitor to perform path counting operations at several control points in the depth first search process, this algorithm is capable of finding the total number of different paths from a source vertex `s` to a target vertex `t` in linear time. Augmented with proper termination criteria, this algorithm only traverses the relevant part of the entire graph, which leads to superior performance compared to a topological sort based algorithm.

Implementation Details

Graph Definition

The graph used in this assignment is defined using the `adjacency_list` structure with `NamedVertex` as vertex and default edge.

```
struct NamedVertex {
  NamedVertex() {}
  NamedVertex(const string& name) : name(name) {}
  string name;
};

using Graph = adjacency_list<listS, vecS, directedS, NamedVertex>;
```

Visitor Design

The path count algorithm is based on customized depth first search starting from the source vertex. An integer value is attached to each vertex to keep track of the path counts from a vertex to the target vertex. Two kinds of events are interested in this algorithm: `discover_vertex` and `finish_vertex`.

In `discover_vertex`, the path count of a vertex is set to 1 if the vertex is the target vertex, otherwise the path count is set to 0.

In `finish_vertex`, the path count of a vertex is set to the sum of path counts of all its descendants.

The algorithm terminates once the source vertex is finished.

The visitor is designed based on the above observation:

1. The visitor inherits from `default_dfs_visitor` to provide default behavior for the events other than the `discover_vertex` and `finish_vertex`.
2. A `map<Vertex, int>` object is included as a member of the visitor to store the path counts at each vertex. A `map<Vertex, vector<list<string>>>` object is also added to the visitor to store the found paths.
3. The final path count is made available through a reference to external path count variable.

```
template <typename Graph>
PathCounter {
public:
  using VertexDescriptor = typename graph_traits<VertexListGraph>::vertex_descriptor;
  using EdgeSizeType = typename graph_traits<VertexListGraph>::edges_size_type;
  using path_t = list<string>;

  PathCounter(VertexDescriptor s, VertexDescriptor t, EdgeSizeType& path_count)
    : s(s), t(t), path_count(path_count) {}

  template <typename Vertex, typename Graph>
  void discover_vertex(Vertex u, const Graph& g) {
    ...
  }

  template <typename Vertex, typename Graph>
  void finish_vertex(Vertex u, const Graph& g) {
    ...
  }

private:
  VertexDescriptor s, t;
  map<VertexDescriptor, int> path_counter;
  map<VertexDescriptor, vector<path_t>> paths;
  EdgeSizeType& path_count;
}
```

`discover_vertex` is a simple function that marks the path count of a vertex based on whether it is the target vertex or not:

```
template <class Vertex, class Graph>
void discover_vertex(Vertex u, const Graph& g) {
  if(u==t) {
    path_counter[u] = 1;
    paths[u] = vector<path_t>(1, path_t(1, g[u].name));
  } else {
    path_counter[u] = 0;
    paths[u] = vector<path_t>();
  }
}
```

Note that the found paths are also updated in a similar way as the path count.

`finish_vertex` needs to sum up the path counts of a vertex's descendants and use that as its new path count:

```
template <class Vertex, class Graph>
void finish_vertex(Vertex u, const Graph& g) {
  // Update the path count
  auto vp = adjacent_vertices(u, g);
  path_counter[u] += std::accumulate(vp.first, vp.second, 0,
                                     [&](size_t acc, Vertex v) {
                                       return acc + path_counter[v];
                                     });

  // Store the found paths
  auto& paths_u = paths[u];
  for(auto vp = adjacent_vertices(u, g); vp.first != vp.second; ++vp.first) {
    const auto& paths_v = paths[*vp.first];
    for(auto p : paths_v) {
      path_t this_path(1, g[u].name);
      this_path.insert(this_path.end(), p.begin(), p.end());
      paths_u.push_back(this_path);
    }
  }

  // Store final result
  if( u == s ) path_count = path_counter[u];
}
```

The found paths are also updated based on the found paths stored in the descendants of this vertex.

Algorithm Details

With the visitor described in previous section, the path count algorithm is simply a depth first search in the graph with early termination. The termination condition for each traversal path is whether the target vertex is reached or not, which can be implemented using a lambda function:

```
auto terminator = [&](vertex_desc u, const VertexListGraph&) { return u == target; };
```

Putting everything together, the final algorithm is implemented as below:

```
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)
{
  typedef typename graph_traits<VertexListGraph>::vertex_descriptor vertex_desc;

  // The color map used in the DFS traversal
  std::map<vertex_desc, default_color_type> vertex2color;
  boost::associative_property_map<std::map<vertex_desc, default_color_type>> color_map(vertex2color);

  // Initialize the graph, set all nodes to white
  // This is necessary because this algorithm calls depth_first_visit directly
  // instead of depth_first_search
  for(auto vp=vertices(G); vp.first!=vp.second; ++vp.first) {
    vertex2color.insert(std::make_pair(*vp.first, white_color));
  }

  // Final path count
  typename graph_traits<VertexListGraph>::edges_size_type path_count;
```

```
  // Execute DFS traversal
  depth_first_visit(G, source,
                    PathCounter<VertexListGraph>(source, target, path_count, std::clog),
                    color_map,
                    [&](vertex_desc u, const VertexListGraph&) { return u == target; });
  std::cout << "Path counting finished." << std::endl;
  return path_count;
}
```

Graph File Format

The graph files used in the assignment follows the format below

```
n m
v1_name v2_name ... vertex_n_name
e1
e2
...
em
```

where `n` is the number of vertices and `m` is the number of edges. Each edge `ek` is a pair of vertex names. For example, `a b` represents an edge from vertex `a` to vertex `b`.

Random Graph Generation

To generate random directed acyclic graph (DAG), it is important to make sure the edges do not form loops in the graph. This can be achieved by assigning ranks to the vertices and only allow edges from higher rank vertices to lower rank vertices.

Experiment Result

Example Graph in the Assignment

The algorithm is able to produce correct output for the example graph in the assignment:

```
vertices(g) = 0:M 1:N 2:O 3:P 4:Q 5:R 6:S 7:T 8:U 9:V 10:W 11:X 12:Y 13:Z
edges(g) = (M -> Q) (M -> R) (M -> X) (N -> O) (N -> Q) (N -> U) (O -> R)
(O -> S) (O -> V) (P -> O) (P -> S) (P -> Z) (Q -> T) (R -> Y) (S -> R)
(U -> T) (V -> X) (V -> W) (W -> Z) (Y -> V)
Finding paths from P to V
start DFS from P
discover P
discover O
discover R
discover Y
discover V
finish V: path counts[V] = 1
finish Y: path counts[Y] = 1
finish R: path counts[R] = 1
discover S
finish S: path counts[S] = 1
finish O: path counts[O] = 3
discover Z
finish Z: path counts[Z] = 0
finish P: path counts[P] = 4
```

```
Found paths:
P -> O -> R -> Y -> V
P -> O -> S -> R -> Y -> V
P -> O -> V
P -> S -> R -> Y -> V
Path counting finished.
Path count = 4
```

Random Graph

Below is an example of random graph experiment.

```
peihongguo@linux2:~/Documents/Codes/CSCE622/homework3/build$ ./path_count_random_graph 8 15 1
vertices(g) = 0:vertex 0 1:vertex 1 2:vertex 2 3:vertex 3 4:vertex 4 5:vertex 5
6:vertex 6 7:vertex 7
edges(g) = (vertex 1 -> vertex 0) (vertex 2 -> vertex 0) (vertex 3 -> vertex 2)
(vertex 3 -> vertex 0) (vertex 3 -> vertex 1) (vertex 4 -> vertex 1)
(vertex 5 -> vertex 4) (vertex 6 -> vertex 3) (vertex 6 -> vertex 5)
(vertex 6 -> vertex 0) (vertex 6 -> vertex 2) (vertex 6 -> vertex 1)
(vertex 7 -> vertex 4) (vertex 7 -> vertex 6) (vertex 7 -> vertex 3)
graph generated.
Finding paths from vertex 6 to vertex 1
start DFS from vertex 6
discover vertex 6
discover vertex 3
discover vertex 2
discover vertex 0
finish vertex 0: path counts[vertex 0] = 0
finish vertex 2: path counts[vertex 2] = 0
discover vertex 1
finish vertex 1: path counts[vertex 1] = 1
finish vertex 3: path counts[vertex 3] = 1
discover vertex 5
discover vertex 4
finish vertex 4: path counts[vertex 4] = 1
finish vertex 5: path counts[vertex 5] = 1
finish vertex 6: path counts[vertex 6] = 3
Found paths:
vertex 6 -> vertex 3 -> vertex 1
vertex 6 -> vertex 5 -> vertex 4 -> vertex 1
vertex 6 -> vertex 1
Path counting finished.
Path count = 3
```