December 2, 2018

Homework 4: Ranking Components By Size
50 Points

A *component* of a graph $G = (V, E)$ is a maximal connected subgraph $G_1 = (V_1, E_1)$ of G. Any two vertices in V_1 are connected by a path and no edge has one vertex in V_1 and the other outside V_1.

A *component* of a Partition p is one of the sets in p.

Part 1: Algorithms. Invent an algorithm named **RankComponentsBySize** that operates on a Partition object p (through its API) and produces a vector v of unsigned integers such that $v[i]$ is the size of the $(1+i)^{th}$ largest component of p: $p[0]$ is the size of the largest component, $p[1]$ is the size of the second-largest component, and so on.

Also invent a process that creates a Partition object p that captures the precise component structure of an undirected graph g. Combining the process with the algorithm yields an application for a graph g: The Component Rank Sequence of g.

Part 2: Implementations. Code up the RankComponentsBySize algorithm in C++ conformant with the stub below (and also available in the file `LIB/graph/partition_util.h`).

And also install your process for capturing the component structure of a graph in the second stub below (and also available in the file `LIB/graph/graph_util.h`).

Test your implementations by compiling a copy of `LIB/graph/agraph.cpp` and executing `agraph.x` on various graphs: on small graphs that can be hand verified and on some large graphs (such as the "Kevin Bacon" actor-movie abstract graph) and some very large graphs generated at random. Compare your results with those using `LIB/area51/agraph_i.x`.

The following libraries *may not* be used: <string>, <set>, <unordered_set>, <map>, <unordered_map>, <algorithm> . Use components of `cop4531p/LIB` instead.

Part 3: Correctness. Provide an argument that your algorithm is correct.

Part 4: Run Costs. Provide an estimate of the runtime and runspace requirements of your algorithm and your component modelling process.

Part 5: Experiments.
1: Try to provide experimental evidence of the Erdös-Reńyi "critical value" for the emergence of a giant component.
2: Given your analysis of the Kevin Bacon graph, in the light of the Erdös-Reńyi result, what can you see or say about these graphs?
3: Discuss large multi-component maze graphs in the light of the Erdös-Reńyi.

Here is C++ code stub in which to code your algorithm. Note that the partition `p` and the vector `v` are passed by const reference and non-const reference, respectively.

```
template < class P >
void RankComponentsBySize (const P& p, fsu::Vector<size_t>& v) // p is a Partition object
{
  // your code goes here
}
```

(See the appendix below and the file `LIB/graph/partition_util.h` for complete context).

Here is C++ code stub in which to code your graph component model process. Note that the graph `g` is passed by const reference and the other two arguments are passed through to the call to RankComponentsBySize.

```
template < class G >
void ComponentRankSequence(const G& g , size_t maxToDisplay, std::ostream& os)
{
  fsu::Partition p (g.VrtxSize());
  // your process to model the components of g with p goes here
  RankComponentsBySize(p,maxToDisplay,os); // <-- calls your algorithm here
}
```

(See the the file `LIB/graph/graph_util.h` for complete context).

Include a test diary in your submission. And **Cite your sources!**

Appendix: Computational context for RankComponentsBySize

```
template < class P >
void RankComponentsBySize (const P& p, fsu::Vector<size_t>& v)
{
  // your code goes here
}

// below is complete code used to display the results to a stream
template < class P >
void RankComponentsBySize (const P& p, size_t maxToDisplay, std::ostream& os = std::cout)
{
  int cw = floor(log10(p.Size()));
  if (cw < 4) cw = 4;
  cw += 3;
  size_t enough, components;
  fsu::Vector<size_t> componentSize(0);
  RankComponentsBySize(p,componentSize);
  enough = components = componentSize.Size();
  if (0 < maxToDisplay && maxToDisplay < enough) enough = maxToDisplay;
  os << ``  number of components: `` << components << '\n';
  if (enough == components)
    os << ``  all components ranked by size:''  << '\n';
  else
    os << ``  top `` << enough << `` components ranked by size:'' << '\n';
  os << std::setw(cw) << ``rank''
     << std::setw(cw) << ``size'' << '\n'
     << std::setw(cw) << ``----''
     << std::setw(cw) << ``----'' << '\n';
  for (size_t i = 0; i < enough; ++i)
  {
    os << std::setw(cw) << 1 + i
       << std::setw(cw) << componentSize[i] << '\n';
    if (componentSize[i] == 1 && 1 + i < componentSize.Size())
    {
      os << std::setw(cw) << '*'
         << std::setw(cw) << 1 << ``  (the remaining `` << (components - i - 1)
         << `` components have size 1)\n'';
      break;
    }
  }
}
```

```
// below is complete code used to write the results to a file
template < class P >
bool RankComponentsBySize (const P& p, size_t maxToDisplay, const char* filename)
{
  std::ofstream os;
  os.open(filename);
  if (os.fail())
  {
    std::cerr << ‘‘ ** Error: unable to open file ‘‘ << filename << ’\n’;
    return 0;
  }
  RankComponentsBySize (p, maxToDisplay, os);
  os.close();
  return 1;
}
```