RCL - March 13, 2019
Jonathan Crawford – July 3, 2019

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.

This is similar to a project I did in COP4020 in which I had to count all the occurrences of different characters, 100 commas, etc. For this type of implementation the key is to use a Map because then you don't have to waste time checking for includes() before incrementing your pairs. You can just do it on the fly. For that project every time I recognized a character I put it in an std::map called hangar and incremented the amount variable. Then at the end I put the elements into a vector called launchpad and sorted them for output or "launch". This is the same scenario I need to increment the occurrences of sizes of components and to do so with only vectors would be poor runtime usage. To do this I will declare a map with both values size_t, one for the amount and one for the size. Then I will count all of the amounts of that size of component from the p.Root. Then I will store those in the provided vector for sorting and return it sorted backwards. For this project I first did the implementation using the standard library and then converted everything to fsu Library usage to meet the spec. I had a tough time declaring the hashtable internally and figuring out the sort logic at first due to the amount of things required to be included. After doing so I used merge sort because it is stable and provides nlogn runtime.

Also invent an algorithm that creates a Partition object p that captures the precise component structure of an undirected graph g. Combine the two algorithms to obtain an application for a graph g: The Component Rank Sequence of g.

In graph_util the first part of the code is given to declare a partition object, this will be how we represent the maximally connected components within g. Next we will loop to walk each vertex and it's adjacency list and call union in order to find all reachable vertices from each vertex and unite them as a component. This is almost the same logic as when we created our maze. All vertices of the graph will be visited and the components will be maximally connected within so this constitutes a precise component structure for g as stipulated above. Then once all components are found, we can call the RankComponentsBySize() function from partion_util to count and order them correctly for display.

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).

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)
void RankComponentsBySize (P& p, fsu::Vector<size_t>& v) // allows path compression
{
  fsu::HashTable<size_t, size_t, hashclass::KISS <size_t> > pMap;
  fsu::HashTable<size_t, size_t, hashclass::KISS <size_t> >::ConstIterator i;
  fsu::GreaterThan<size_t> comparator;

  for (size_t i = 0; i < p.Size(); ++i)
    pMap[p.Root(i)]++;

  for (i = pMap.Begin(); i != pMap.End(); ++i)
    v.PushBack((*i).data_);

  fsu::g_merge_sort(v.Begin(), v.End(), comparator);
}
```

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)
{
  size_t vSize = g.VrtxSize();
  fsu::Partition<size_t> p (vSize);
  for (typename G::Vertex v = 0; v < vSize; ++v)
  {
    for (typename G::AdjIterator j = g.Begin(v); j != g.End(v); ++j)
      p.Union(*j, v);
  }
  RankComponentsBySize(p,maxToDisplay,os);
}
```

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.

```
Welcome to graph analysis
 g.VrtxSize(): 2000
 g.EdgeSize(): 1000
 g bipartite?  YES:  Red = 1376 , Black = 624
  number of components: 1001
  top 10 components ranked by size:
   rank   size
  ----   ----
     1    127
     2     64
     3     28
     4     24
     5     24
     6     23
     7     19
     8     18
     9     16
    10     15
Welcome to graph analysis
 g.VrtxSize(): 615635
 g.EdgeSize(): 535450
 g bipartite?  NO
  number of components: 133990
  top 10 components ranked by size:
    rank    size
   ----    ----
      1  436326
      2      29
      3      29
      4      27
      5      25
      6      22
      7      20
      8      19
      9      18
     10      18
```

```
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph_i.x movies.txt.ug 10 > t.bacon
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x movies.txt.ug 10 > m.bacon
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>diff t.bacon m.bacon
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>

Welcome to graph analysis
 g.VrtxSize(): 119429
 g.EdgeSize(): 202927
 g bipartite?  YES:  Red = 4188 , Black = 115241
  number of components: 33
  top 10 components ranked by size:
   rank    size
   ----    ----
     1  118774
     2      67
     3      46
     4      44
     5      34
     6      29
     7      29
     8      27
     9      27
    10      26
```

After some testing with small medium and extremely large graphs generated at random I am confident in the algorithm correctness. I found the Kevin Bacon graph and implemented it as well. The following statement was a big clue as to how to do the algorithm along with the teacher's hint for using an AA and sorting with GreaterThan in the discussion board.

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.

RankComponentsBySize():
I know that this is correct because I have incremented the occurrences of each component by size in my hashtable and then sorted them in reverse order. It displays the correct output in comparison with agraph_i.x, The counting is done by recursive root because we are looking at each component in the partition. This is similar to the spanning tree logic presented in RCL2016. I used a stable but fast sort since this implementation is meant to be counted upon as a utility to be called from a library. Merge_sort does have runspace considerations because it is not in place but this is a tradeoff considering the implementation is being used in Linprog.

ComponentRankSequence():
Much like how we used the function Connect() in proj3 to create our maze, we used a partition object and then called p.Union(cell,neighbor); to provide a path through the connected components. This provided an accurate component structure in which all cells were reachable within that component. We know that no vertex V has an edge outside of its component or it would have been found in the list walk. The implication is the same here, we united the vertices in the graph, we know that by RCL2019,
Suppose $G = (V,E)$ is an undirected graph. G is called *connected* iff for every pair $x,y \in V$ of vertices there is a path in G from x to y. A *component* of G is a graph C such that

1. C is a subgraph of G,
2. C is connected, and
3. C is maximal with respect to the first two properties.

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

As noted in RCL2016, The worst case runtime for Union, Find, and Root is ≤ O(log2 n). Union and root were both used at that runtime. For the smaller function Union() was called at this runtime and a loop bounded by n, so it is running at O(n+ log2 n). Then in the larger function the adding of each element to the hashtable is O(1+n/b) bounded by the amount of elements and root() at the above listed runtime from RCL2016. The copying of the elements from the hashtable to the vector is also O(n), there is no way around something done sequentially. The sorting of the vector in reverse order is done by merge sort and is O(nlogn). The runspace required is +O(n). Bringing this all together we get O(n + nlogn + log2 n) runtime and +O(n) for runspace.

Part 5: Experiments.
1: Try to provide experimental evidence of the Erd¨os-Ren´yi "critical value" for the emergence of a giant component.

According to RCL 2018 Random Graphs this can be done by, Use rangraph and rangraph_ER to tease out the phase change behavior as the expected degree passes the value 1.0. This serves to confirm the classical result of Erdös and Rényi.

crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x ERg.75 5

Welcome to graph analysis
g.VrtxSize(): 100
g.EdgeSize(): 34
g bipartite? YES: Red = 76 , Black = 24
number of components: 66
top 5 components ranked by size:

```
  rank   size
  ----   ----
    1     5
    2     4
    3     4
    4     4
    5     4
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x ERg.9 5

Welcome to graph analysis
 g.VrtxSize(): 100
 g.EdgeSize(): 53
 g bipartite?  YES:  Red = 67 , Black = 33
  number of components: 47
  top 5 components ranked by size:
  rank   size
  ----   ----
    1    11
    2    10
    3     9
    4     7
    5     7
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x ERg1.0 5

Welcome to graph analysis
 g.VrtxSize(): 100
 g.EdgeSize(): 50
 g bipartite?  YES:  Red = 70 , Black = 30
  number of components: 50
  top 5 components ranked by size:
  rank   size
  ----   ----
    1    28
    2     5
    3     5
    4     4
    5     3
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x ERg1.1 5

Welcome to graph analysis
 g.VrtxSize(): 100
```

```
 g.EdgeSize(): 40
 g bipartite?  YES:  Red = 74 , Black = 26
  number of components: 60
  top 5 components ranked by size:
   rank   size
   ----   ----
     1     10
     2      7
     3      7
     4      5
     5      4
crawford@linprog7.cs.fsu.edu:~/cop4531/hw4>agraph.x ERg1.25 5

Welcome to graph analysis
 g.VrtxSize(): 100
 g.EdgeSize(): 60
 g bipartite?  YES:  Red = 67 , Black = 33
  number of components: 41
  top 5 components ranked by size:
   rank   size
   ----   ----
     1     21
     2     20
     3     13
     4      4
     5      3
```

The data shown where ERg(value of expected degree for graph name) shows how far off the components are spread at +-.25 for the .75 started with and the 1.25 ended with. The values of +-.1 for .9 and 1.1 show the realization of components starting to trend upward. And the value of 1.0 shows the giant component. By slowly growing and showing the tightening at 1.0 and then falling back off our mark we have proven the Erdos Renyi critical value that when G(n,p) = 1 the graph with have a giant component.

2: Given your analysis of the Kevin Bacon graph, in the light of the Erd¨os-Ren´yi result, what can you see or say about these graphs?

The bacon graph has the emergence of a giant component because the actors are so connected by the movies they have been in, that is the whole point of the project to see that once a component is made all actors within are maximally connected with such a large sample of movies to connect most actors degrees will be one and they will be inside the giant component. Because

they aren't simply relying on degrees of Bacon but on degrees from just anyone of each other. This is true because when the graph is unioned 1,2 and 2,6, then 1 can touch 6.

3: Discuss large multi-component maze graphs in the light of the Erd¨os-Ren´yi.

The large maze graphs are undoubtedly going to have a large component due to the breaking down of the walls to find a path, this is limited only by the size of the rows and columns, then there will exist pockets of smaller components that are unreachable. Although its hard to picture just by looking at the maze the components are pocketed all around the critical path through the maze. The emergence of a giant component is absolutely necessary in order to find the path from one side to the other, and the amount of components it gobbles up in the process shrinks the surrounding pockets size possibility.