March 13, 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.

```
Algorithm I: RankComponentsBySize
U = universe
p = Partition object on U
v = vector to store results
{
  step 1: count the items in each set (``tree'') of the partition
    associative_array treeSize[]
    for each element x of the universe
      treeSize[p.Root(x)] += 1
  step 2: put the sizes in the vector v
  step 3: sort v large-to-small
  return v
}
```

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.

```
Algorithm II: Construct Partition model of graph
g = graph
U = universe = {vertices of g}
p = Partition object on U initialized to singletons
{
  for each edge e = [x,y] in g
    p.Union(x,y)
  return p
}
```

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`).

```
template < class P >
// void RankComponentsBySize (const P& p, fsu::Vector<size_t>& v) // p is a Partition object
void RankComponentsBySize (P& p, fsu::Vector<size_t>& v) // allows path compression
{
  size_t components = p.Components();
  fsu::HashTable<size_t,size_t> treesize(p.Components());
  for (size_t i = 0; i < p.Size(); ++i)
  {
    ++treesize[p.Root(static_cast<typename P::IntType>(i))];
  }
  assert(components == treesize.Size());
  v.SetSize(components);
  size_t i = 0;
  typename fsu::HashTable<size_t,size_t>::Iterator j;
  for (j = treesize.Begin(); j != treesize.End(); ++j)
  {
    v[i++] = (*j).data_;
  }
  fsu::GreaterThan<size_t> gt;
  fsu::g_heap_sort(v.Begin(),v.End(),gt);
}
```

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`).

```
template < class G >
void ComponentRankSequence(const G& g , size_t maxToDisplay, std::ostream& os)
{
  fsu::Partition<size_t> p(g.VrtxSize());          // uses partition2.h
  typename G::AdjIterator i;
  for (size_t v = 0; v < g.VrtxSize(); ++v)
  {
    for (i = g.Begin(v); i != g.End(v); ++i)
      // p.Union(v,*i);  // OK, but Union is called twice for each edge
      if (v < *i) p.Union(v,*i); // blocks redundant Union calls
  }
  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`.

Note that these implementations are installed in the suggested contexts `partition_util.h` and `graph_util.h`. The solution for Homework 3, `IsBipartite`, is also installed in `graph_util.h` so

that these two files are now fully implemented, and the graph analysis program `agraph.cpp` can now be built to an executable `agraph.x`.

In addition, the test harness `fpartion2+.cpp` now builds to the executable `fpartition2+.x` which gives direct access to test the partition version.

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

Algorithm I has three steps. Step 1: determines the sizes of the sets in the partition p by counting the elements that ascend to each root in the tree model. Step 2: places the set sizes in a vector. Step 3: sorts the vector large-to-small. The vector satisfies the required conclusion - a large-to-small ranked listing of the component sizes. □

Algorithm II builds a partition by calling Union on the vertices of each edge in the graph. If vertices v and w are vertices in the same component of the graph G then there is a path $v = x_0, x_1, ..., x_k = w$ such that x_{i-1} and x_i are the vertices of an edge $e = [x_{i-1}, x_i]$ in G. By construction, we have called Union(x_{i-1}, x_i) for each i, so $v = x_0, x_1, ..., x_k = w$ are in the same component.

Conversely, if v and w are in the same component of the partition then a sequence of Union operations on pairs x_{i-1}, x_i, startng at $x_0 = v$ and ending at $x_k = w$, must have occured, so there is a path in G connecting v and w. □

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

Algorithm I:

The const version of p.Root(x) has runtime $\mathcal{O}(\log_2(n))$ and the non-const version is faster, approaching $\mathcal{O}(\log^*(n))$. (See Theorems 1 and 2 of the Disjoint Sets Union/Find Notes.) The insert and access time in an associative array is amortized constant [if we use an unordered map such as a HashTable implementation] and $\mathcal{O}(\log_2(n))$ [if we use ordered map such as a balanced BST implementation]. Therefore step 1, the first (counting) loop, runs in time at most $\mathcal{O}(n \log^*(n))$ when using unordered map and non-const Patrtition methods.

Step 2 of the algorithm is a copy loop with runtime $\Theta(m)$ where m is the number of components. Clearly $m \leq n$ so step 2 is dominated by step 1.

The sort algorithm in step 3 has runtime $\mathcal{O}(m \log_2(m))$. In many situations m is much smaller than n, although there is no guarantee this is the case in general. In situations where there are only a few components, such as a random graph with expected vertex degree large than 2, the runtime will be dominated by the step 1.

Note that step 1 is also where we can improve runtime by using hashmap and allowing the calls to Root to be the non-const [path-compression] version.

We can assert that the runtime is bounded above in all cases by $\mathcal{O}(n \log^*(n) + m \log_2(m))$. □

The extra space required by the algorithm is the associative array `treesize` used to determine the sizes of the components. The size of `treesize` is the number m of components, so we can conclude the extra space requirement is $+\Theta(m)$. □

Algorithm II:

Let n be the number of vertices in the graph G. Algorithm II consists of a straightforward traversal of G with a call to Union embedded in the inner loop. The traversal itself requires $\Theta(n + |E|)$ steps and the Union call has runtime $\mathcal{O}(\log^* n)$. Therefore the runtime is $\mathcal{O}((n + |E|) \times \log^* n)$. No extra space is required. □

Part 5: Experiments.

5.1. Try to provide experimental evidence of the Erdös-Reńyi "critical value" for the emergence of a giant component.

These experimental results show the sizes of the largest and runner-up components for randomly generated graphs:

vertices	edges	[d]	largest	second	ratio
100,000	45,000	0.90	268	207	1.29
			282	226	1.24
			332	175	1.90
			323	273	1.84
			303	187	1.62
100,000	47,500	0.95	932	343	2.71
			724	537	1.35
			710	421	1.69
			365	316	1.16
			275	273	1.01
100,000	52,500	1.05	11278	537	21.00
			10614	327	32.46
			8099	372	21.77
			10999	481	22.87
			11681	301	38.81
100,000	55,000	1.10	19027	268	70.00
			19858	190	104.52
			15980	180	88.78
			17862	355	50.32
			18506	134	1381.09

The results show a 20-fold jump in the ratio of the size of the largest component to the size of the next largest component as [d] passes from 0.95 to 1.05.

5.2. 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?

The Kevin Bacon graph has this component signature:

```
Welcome to graph analysis
 Graph g:      movies.txt.ug
 g.VrtxSize(): 119429
 g.EdgeSize(): 202927
 g bipartite?  YES:  Red = 4188 , Black = 115241
  number of components: 33
  all 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
      11      25
      12      23
      13      23
      14      21
      15      20
      16      19
      17      18
      18      18
      19      17
      20      16
      21      14
      22      13
      23      12
      24      12
      25      11
      26      11
      27      11
      28      10
      29       9
      30       8
      31       7
      32       6
      33       2
```

which shows one major component that is 1773 times the size of the next smaller component. The 32 minor components represent tiny movie-actor universes that are disconnected from the main Kevin

Bacon universe. The average degree of a vertex is [d] = 3.4, so the "giant component" is predicted by Erdös-Reńyi.

Note however that this component "tail" is slightly thicker than one would expect for a random graph, which typically looks like

```
 Graph g:      rangraph.119429.202927
 g.VrtxSize(): 119429
 g.EdgeSize(): 202927
 g bipartite?  NO
  number of components: 4274
  top 10 components ranked by size:
    rank    size
    ----    ----
       1  114863
       2       9
       3       4
       4       4
       5       4
       6       3
       7       3
       8       3
       9       3
      10       3
```

Actual naturally occuring graphs tend to have this "thick tail" property, and why is unknown.

5.3. Discuss random maze graphs in the context of Erdös-Reńyi. What can you see or say about these graphs?

Here is analysis of a random maze graph with 49 components:

```
Welcome to graph analysis
 Graph g:      maze100x200.49.ug
 g.VrtxSize(): 20000
 g.EdgeSize(): 19951
 g bipartite?  YES:  Red = 10022 , Black = 9978
  number of components: 49
  top 10 components ranked by size:
   rank   size
   ----   ----
      1  19945
      2      4
      3      2
      4      2
      5      2
      6      2
      7      1
      *      1  (the remaining 42 components have size 1)
```

There is one major component (where all cells are mutually reachable), 5 very small closed regions consisting of 4,2,2,2, and 2 cells, and 42 closed boxes. Note that [d] = 1.995 for this maze graph ... approximately double the Erdös-Reńyi threshold. This is a typical result for randomly generated mazes. This result is fairly typical for random mazes when the process is terminated when start and goal are first connected:

Research-level Question: Can you find a formula for the expected degree of a random maze graph? (Note in the above that [d] is approximately 2.00 which is also the col/row ratio.)