John Basil

I. Description and explanation of the graph implementation.

The graph data structure implemented for this project is an adjacency list. This adjacency list is implemented as an object class named graph with four private member variables and two non-trivial public member functions (excluding member variable getters used for testing purposes, only).

The four member variables are defined as follows: an unordered map named inf_map mapping an integer key representing a vertex to an unordered integer set value representing the set of vertices connected to the key vertex by an inflowing edge; an unordered map named out_map mapping an integer key representing a vertex to an integer value representing the outdegree corresponding to the key vertex; an unordered map named key_map mapping a string key representing a vertex to an integer value, that becomes the key of the inf_map and out_map, representing the same vertex; an integer named size storing the size of the graph that is reflected by the size of inf_map, out_map and key_map.

The design of the member variables depends on the key used to represent each vertex in the graph. The key must be unique in order to represent a unique vertex in the set of vertices stored in the graph. A vertex is initially identified by a string that is passed in as an argument to the insert function. Strings represented as keys must be hashed in order to account for clashing where two strings with different character compositions can produce the same value when the character ASCII values are added together. Hashing can be costly and is unnecessary where a unique key is provided in the form of the graph size that is only updated each time a unique vertex is inserted, as is the case with this graph.

As such, the graph size is used as a unique key identifier for each unique vertex that is updated on each insertion by invocation to the graph insert function. In order to identify the vertex name from its key, the key_map is updated whenever a unique vertex is inserted. Before a vertex is inserted into the inf_map and out_map, the key_map checks if the vertex integer identifier exists by mapping the string name to the integer value. This check is performed by attempting to insert the vertex name into the key_map. The insert function of the unordered_map key_map variable (not to be confused with the insert function of the graph) returns, both, an iterator containing the integer value corresponding to the string name, as well as a boolean indicating whether the insertion was successful. If insertion is unsuccessful (i.e. the vertex is already in the graph), the iterator is positioned to where the existing string name is stored in the key_map container and the corresponding integer identifier can be accessed.

The graph insert function returns a boolean if the specified ‘to’ vertex inserts, into its list of inflowing vertices, the specified ‘from’ vertex (i.e. if ‘from’ was not previously inserted) which is useful in many contexts for verifying the result of insertion but, in this case of this project, is only used for testing purposes. Vertices that are inserted into a graph instance can be ranked according to relevance by invoking the rank function. The rank function selects, from the inf_map key, a vertex, called a base. For each base, the rank function selects, from the inf_map value, each inflow to the base. The select inflow is mapped to the corresponding outdegree computed by the insert function. The rank for the base from the select inflow is then computed by dividing the rank for the inflow by the inflow outdegree. The composite rank for the base vertex from all inflows is computed by summing all of the ranks for the base from each select inflow. This composite rank is then stored in an array of ranks corresponding to each base vertex.

A composite rank for each base vertex from all inflows is computed by iterating through the inf_map and selecting each key as base vertex and each value as the set of inflows, then iterating through the set of inflows and summing the computations. The ranks of the base vertices and inflow vertices are maintained in separate arrays. On the initial power iteration, the ranks of the inflow vertices are set as equal proportions of the graph size.

The ranks of each array are stored and accessed by indices corresponding to integer identifiers passed by either the base or inflow vertex iterators. When rank computations for all base vertices have been stored, the power iteration is complete, and the array of base ranks is copied to the array of inflow ranks and the base ranks are recomputed on the next power iteration. When power iterations are complete, each element of the array of the most recently-computed ranks is identified by the vertex integer identifier using the key_map member variable of the graph instance, and mapped to a vertex name. These name and rank pairs are inserted into an ordered map, which by default alphabetizes the set of pairs. This ordered map of name and rank pairs is the return value of the rank function.

A reasonable concern regarding the design choice to use graph size as a unique key identifier extends beyond the scope of this project where a remove or delete function is implemented for the graph. Eliminating a vertex with a previously-assigned key from anywhere except the last inserted vertex could cause the next-inserted vertex to clash with the previously-inserted vertex due to the graph size indicating the index of the last element of the graph rather than the next empty position. To address this concern, one could store the positions of removals in a private member container (e.g. a stack or queue) and when inserting, if this container is not empty, assign the vertex to be inserted a key from some position (e.g. the front) of this container rather than from size, and remove (e.g. pop or dequeue) the assigned key before the next insertion, repeating for each insertion until the container of removals is empty, at which point size is again used as key.

Being that insertions are size-based and therefore sequential, buffered by the boolean result of key_map insert which guarantees key uniqueness, indices of vectors could be used as keys; however this approach is less semantic and no more efficient than hash-backed unordered map ops and loses viability where keys are anything but ordered, unique 0-based natural numbers.

II. Worst-case computational complexity for functions of the graph implementation.

The following identifiers are represented in the space complexity analysis. V represents the variable quantity of computer memory units required to store all vertices; E represents the variable quantity of computer memory units required to store all edges; K represents the static quantity of computer memory units required to operate static features of a given function such as loop counters and list pointers. The default constructor is trivial and features O (K) space complexity.

The following identifiers are represented in the time complexity analysis: $\overline{V}$ represents the cardinality, or the number of unique instances, of a vertex in the set of vertices; outdegree($\overline{V}$) represents the cardinality, or the number of unique instances, of an adjacent vertex to a particular vertex; P represents the number of power iterations specified for the ranking feature. N represents the number of insertions. The default constructor is trivial and features O (1) time complexity. Note that insert and dereference ops for hash-backed unordered map/set is constant time for unique keys that never collide.

A. bool insert(const string& from, const string& to)

Space complexity: O (K)
Time complexity: O ($\overline{V}$)

B. map <string, float> rank(const int power_iterations)

Space complexity: O (V + E + K)
Time complexity: O (P * $\overline{V}$ * indegree($\overline{V}$))

III. Worst-case computational complexity for main().

(Using the conventions defined in the preceding section)

Space complexity: $O(V + E + K)$

Time complexity: $O(N * \overline{V} + P * \overline{V} * \text{indegree}(\overline{V}))$