Design Document

Team Graph_Algos

Sean Garvey (sjg2174)
Minh Truong (mt3077)
Sammy Tbeile (st2918)

Team Graph_Algos set out to create a graph algorithm library that has a simple and easy to use interface. We found motivation for our library by searching for popular graph algorithm libraries and examining them, like Boost, only to find that their interfaces are notoriously complex and difficult to use. The primary challenge in using these libraries was that there are a number of parameters which are required for each of the algorithm function calls. Not only are there so many parameters to keep track of, the parameters are often specialized data types, created by the library -- meaning the user needs to learn about all of these data structures prior to using the algorithms.

During our formative discussions about what our library should and shouldn't be, we decided to make our interfaces minimalistic, such that our algorithm functions require only those parameters absolutely necessary. Johnson's, Tarjan's, and Prim's need only a Graph container passed as arguments to their functions, since they require no additional parameters to run. Bellman-Ford and Dijkstra's are single pair shortest path algorithms, so they require starting and ending vertices in order to run, so those were the only two additional parameters required for those interfaces.

Similarly, the Graph container takes only one argument on construction, the user representation of a graph. Now initially, we were considering two Graph containers, one which represented an Adjacency Matrix and another which represented an Adjacency List. As we progressed with our project, we discovered that certain algorithms are better optimized for certain graph representations, and with memory being cheap these days, we decided to implement our algorithms in only the more optimized format, and merge the two Adjacency Matrix and Adjacency List containers into a Graph container, and represent the graph internally in both formats. As a result, the user no longer need be concerned about which adjacency matrix or list container to call when constructing their graph, a bonus in terms of keeping to a minimalistic interface. We also floated around the idea of allowing the user to define the vertex type of the internal graph data structure. After considering the effect of bloated data structures on algorithm performance, we decided again to pursue a minimalistic design and restrict the allowable vertex type to doubles.

Again, as we made more progress on the project, we encountered new ideas for what our library should be. In applying our library to a set of test data, we discovered that it was a tedious process to create a representation of a graph. Our dataset contained bad, missing, duplicate, and invalid data. Programming to catch these errors was time consuming -- something that was against our ease of use principles. “I just want to add this damn edge to a graph” was ultimately what we were thinking the entire time. So, we created a helper class, which constructs a matrix, or list (the user specifies this as the sole parameter to the constructor of the class), and has a single function: add_edge(starting_vertex, ending_vertex, weight) -- an edge in its most primitive form. This also makes it convenient for those users inputting row, or indexed data, such as reading from a CSV or database. Not only did we find it complicated to construct a graph, we found it difficult to identify a particular vertex, in human terms, since we associate vertices with indexes. To resolve this issue, we construct a map, to associate vertices with indexes, and an associative vector, to associate indexes with vertices, alongside the graph representation being built by the graph helper. This way, when the user wants to input starting and ending vertices into Dijkstra’s or Bellman-Ford, they can fetch the corresponding index via the map, and when the user wants to interpret the output of the graph, a series of index values, they can get the associated vertices through the associative vector. Theses data structures are returned with the graph representation when the user calls the get_tuple function on the graph helper object -- returning the data structures in a tuple ensures that the user does not mistakenly add an edge to the graph helper object, in between calls to fetch the representation, map, and associative vector, which would in turn lead to inconsistent data structures.