**Graph Applications Research: Lab Report**

Brandon Michelsen

**Introduction:**

This assignment served as a beginning exploration of graph libraries and interfaces for us students. It gave us the chance to experiment with creating graphs, representing graphs, and running algorithms on graphs. Overall, it provided it us with a very good introduction to how graphs can be used in practice.

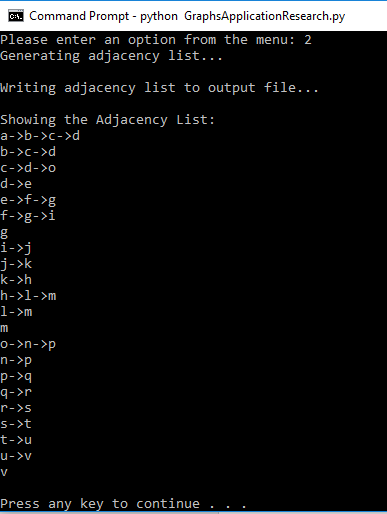
For this assignment, I decided to work with the NetworkX library for Python. NetworkX is a module built for Python that allows for the creation, manipulation, and study of complex graphs and networks. It provides the functions and algorithms needed for generating graphs structures and representations as well as traversal and study of graphs. The version of NetworkX that I implemented in my program is version 2.2. I also implemented the drawing functionality that NetworkX provides – that is, being able to output a graphical view of the graph. To use this functionality, my program also makes use of Matplotlib for Python. Specifically, it uses Matplotlib version 3.0.2. The version of Python being used version 3.7.

The program is structured to output several files in relation to several of the functions of the program. These files include: the output of each of the three algorithms, the output of the two graph representation functions, and the output graphical representation of the graph. The algorithms outputs and the graph representation outputs are each stored as a text file in a folder named “Output Text Files” in the code directory. The graphical representation of the graph is stored as a PNG file in the code root directory. In addition to this, text files representing graph data are used to build a graph (text files are read in where each line represents a weighted edge between two nodes). Example graph files are given in the “Input Graph Files” folder in the code directory. The graphs built by this program are weighted, undirected graphs.

**Graph Representations:**

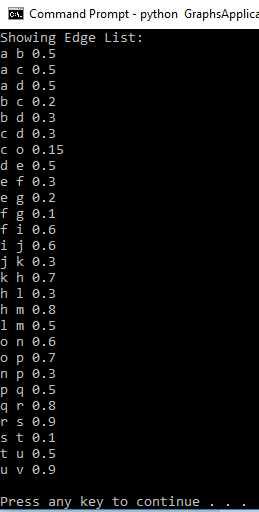
The first step of implementing the graph was to ensure that the graph could properly by loaded into the NetworkX structure. To ensure that the graph data was loaded correctly, we need a way to display it. This is where graph representations come in. A graph representation is a way to show the graph structure without using a graphical representation. NetworkX provides several different ways to represent a graph. The two methods that I chose were the adjacency list representation and the edge list representation.

The adjacency list representation shows the vertices in the graph and what vertices are connected to each other. On the left-hand side of the adjacency list is the list of vertices in the graph. Moving toward the right of each vertex is a list of vertices attached to that vertex. Usually, this representation would provide a way to show the weights for edges. However, the adjacency list representation that NetworkX uses does not show the weights. An example of the adjacency list output is given below:



As can be seen, the adjacency list shows a list of all the vertices in the graph to the left, and displays the vertices connected to a given vertex in a list to the right of that vertex.

The second representation that I used – the edge list representation – shows all the edges in the graph. This gives us all the information we need about the graph, which includes: the edges in the graph, the nodes in the graph, and the weights for each edge. An example of the edge list output is given below:



As can be seen, each line in the representation displays an edge between two nodes along with the weight for that edge at the end of the line. This gathers all the information we need to know about the graph.

**Graph Algorithms**

This section describes the three algorithms I chose to implement in my program. The three algorithms I chose are the A\* Shortest Path algorithm, the PageRank Link Analysis algorithm, and the Triangles Clustering Algorithm. Each of these three algorithms is discussed in detail below, including a description of how the algorithm works, an explanation of the inputs for the algorithm, a description of the results from each algorithm, and an explanation of where each algorithm can be used in a practical application.

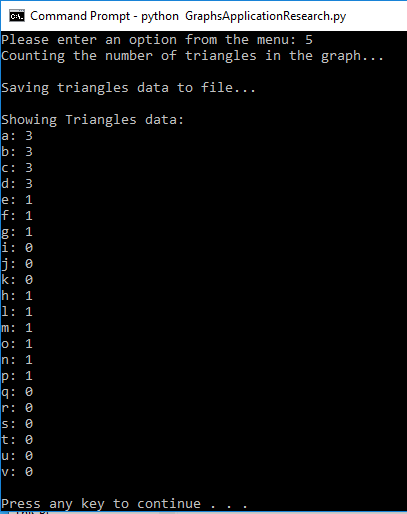
**Triangles:**

The first algorithm I would like to discuss is the Triangle Counting Algorithm. Triangle counting is used in cluster analysis in a graph structure. A triangle, in our case, is a set of three vertices where each vertex has a relationship to the each other vertex in the series.

The algorithm for counting triangles is quite simple. In its simplest form, the algorithm takes a node in the graph and tracks how many triangles that node is a part of. A simple algorithm to solve this is as follows: given a vertex *v*, for every pair of vertices *u*, *w* in the set of *v*’s neighbors, if the triplet *u*, *v*, *w* forms a triangle, increment the running total of triangles that contain *v*. This is one of the simpler algorithms, and more complex ones exist with more efficient runtimes. The algorithm used by the NetworkX library actually counts the number of triangles every vertex in the graph is a part of, which makes use of the following change to the algorithm given above: for every vertex *v* in graph *G*, for every pair of vertices *u*, *w* in *v*’s neighbors, if the triplet *u*, *v*, *w* forms a triangle, increment the number of triangles that contain vertex *v*.

For the inputs that the version of the algorithm NetworkX uses, they involve the graph, *G*, to count the triangles in as well as an optional input telling the algorithm what specific vertices to count the triangles for (the default is the whole graph). The output given by the algorithm is a Python dictionary, containing each vertex in the graph as keys and the number of triangles each vertex is part of as the values for each key.

An example of the Triangles function for NetworkX is given below (using a graph structured from Graph2.txt in the program):



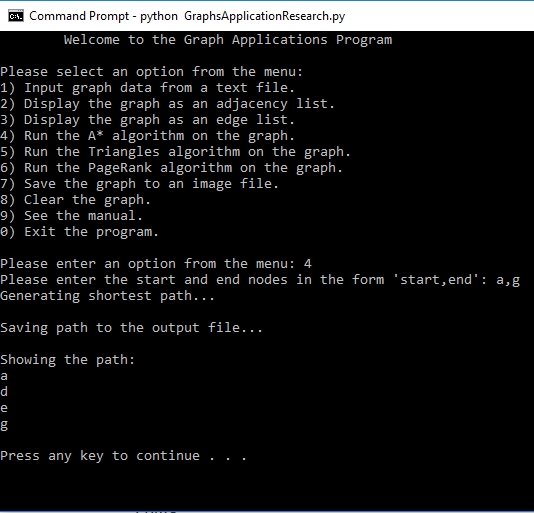
As can be seen in the image above, there are several triangles created by the given graph. The algorithm tracks the number of triangles in the graph, giving the number of triangles that each vertex is a part of as an output. Looking at the results of the algorithm, we can see that there are several triangles that contain the first few vertices in the graph, showing that that area is where the most clustering is occurring. The rest of the graph either has no triangles or only one triangle between a set of three nodes.

As mentioned earlier, triangle counting is used the most in cluster analysis as it relates to graph theory. Specifically, a triangle count is used to calculate the clustering coefficient of a given vertex, which is the measure of the tendency of the node clusters to the rest of the nodes in the graph. This is extremely useful in social network analysis, where tightly knit groups of people tend to crop up. This can help determine the cohesiveness of communities and the trust between people in those communities.

**A\*:**

The A\* algorithm is a very useful graph traversal algorithm. It is perhaps the most widely used graph traversal algorithm in modern software. The A\* algorithm is a shortest-path algorithm, meaning it searches for the shortest path between two vertices in a given graph. Alternatively, the algorithm can be run in such a way that it finds the shortest path between all vertices in the graph, the output of which can be directly addressed later to decrease algorithm runtime. The algorithm is relatively straightforward. Given a graph, a destination vertex, and a starting vertex, the algorithm will traverse the vertices until the destination node is reached. This traversal is somewhat similar to Dijkstra’s algorithm – another graph traversal algorithm – in that we use the weight of an edge to determine the shortest path. However, A\* differs from Dijkstra’s algorithm by its use of heuristics. Simply put, a heuristic is an aspect of the problem that influences a program’s decision making. In the case of A\*, the heuristic is represented by the distance from a given vertex to the destination vertex. A\* operates using the following algorithm (here, the cost is the cost of the vertex given by Dijkstra’s algorithm – the distance from the starting vertex – and estimated cost is the cost plus the heuristic): initialize vertices (predecessors are null, vertex cost is infinity, estimated cost is infinity); set source vertex costs (cost is 0, estimated cost is cost plus heuristic); initialize the open and closed lists (closed list is empty, open list is a priority queue of nodes based on the estimated cost – begins with the source node); while the open list is not empty, get the current vertex from open list, add the current vertex to the closed list; for each edge in the current vertex – if the source cost plus the edge weight plus the heuristic is less than the destination cost, reset destination cost to source cost plus the edge weight, reset the destination estimated cost to the cost plus the heuristic, set the predecessor of the destination to be the source vertex, and add the destination to the open list; repeat the while loop.

The inputs to the A\* algorithm are relatively simple: a graph is given along with a starting vertex and a destination vertex. In the implementation of A\* that NetworkX uses, the algorithm outputs a list of vertices in the shortest path between the starting vertex and the ending vertex. An example output of the A\* algorithm is given below (the graph was created using the Graph2.txt file):



As can be seen in the image above, we give the algorithm a starting node (in this case ‘a’) and an ending node (‘g’). The algorithm then calculates the shortest path between these two nodes and returns a list of vertices representing the found path (‘a’, ‘d’, ‘e’, ‘f’).

There are many applications for the A\* algorithm. One of the biggest ones has to do with artificial intelligence in games. For example, if you needed an enemy in a game to find the best path to your player’s character while also avoiding objects, then the A\* algorithm can be used. Another example is with path finding on mapping applications. By representing the map as a graph with vertices representing nodes and edges representing paths that can be traveled, A\* can be used to find the shortest path between two locations.

**PageRank:**

Finally, I would like to discuss the PageRank algorithm – which is the final algorithm I used in this program. The algorithm was developed by Google founders Larry Page and Sergei Brin in their 1998 paper “The PageRank Citation Ranking: Bringing Order to the Web.” The basic concept of PageRank is that it is a voting system, where each webpage on the internet has a say in what web pages are of “importance” – which helps determine what pages show up first as search results. This can be applied to graphs if we imagine each web page as a vertex in the graph and edges as the links between each web page. Normally, the algorithm is meant to be run with directed edges, since each link on a webpage only links one way – going out to the destination web page. However, NetworkX can use undirected graphs by converting each undirected edge to a set of two directed edges, one going out from each node in the set.

As stated previously, PageRank is essentially a voting system. Each web page has a “vote” that they can cast to determine the important web pages on the internet. The “vote” is distributed out amongst any number of web pages that the given web page links to (the links on the web page represent that page’s vote). The PageRank of a site is given by the following probability distribution:

Where:

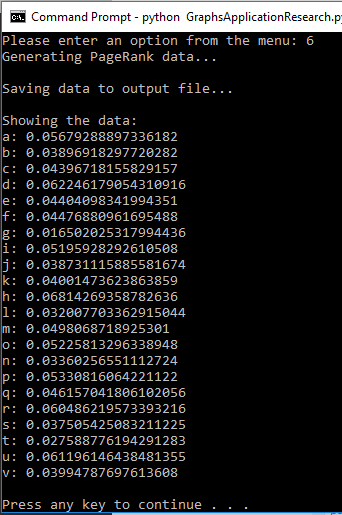
PR(V) = The PageRank of the given vertex

d = 0.85 (the damping factor used in the equation to limit the amount of influence any one vertex has)

PR(Tn) = The PageRank of the vertices linking to the V

Now, when starting out with the just the equation, none of the vertices have a PageRank. Since the PageRank of a vertex depends on the PageRank of the vertices connecting to it, how can we calculate the PageRank of that vertex without knowing the PageRank of the other vertices? Page and Brin give the answer in their paper, by saying that the calculation can be run through an iterative algorithm. Essentially, this means we can guess the beginning values for the incoming PageRanks and then run many iterative calculations using the equation above. Each iteration brings us closer to the true PageRank value for our vertex.

The input for the algorithm, as given by the NetworkX implementation, is the graph that we will be running the PageRank algorithm on. The output is a Python dictionary where the keys are the vertices in the graph and the values are the PageRank values for each vertex. An example output of the algorithm is given below (the graph was created using the Graph2.txt file):



As can be seen in the algorithm above, the algorithm returns a ranking for each node in the graph. Looking through the values, we can find out which vertex in the graph has a higher “importance” by finding the nodes with the higher-ranking value. In the case of our graph, the vertex with the highest ranking appears to be vertex ‘h.’

As stated earlier, the PageRank algorithm was developed by Google founders Larry Page and Sergei Brin for the Google search engine. This is where the most use for the PageRank algorithm comes from. However, there are other use cases, such as fraud detection.

**Conclusion**

In closing, graphs are a very widely used data structure in modern software engineering. They have a wealth of use cases and a host of supporting algorithms that make them a very important tool in a programmer’s toolbox. There are several libraries that can be used to implement a graph data structure. NetworkX is a very powerful version of these libraries for Python. Making use of a graph library to implement a graph data structure can help to create a very powerful program.

References

Neo4j. (n.d.). 4.1. The PageRank algorithm - Chapter 4. Centrality algorithms. Retrieved from https://neo4j.com/docs/graph-algorithms/current/algorithms/page-rank/

Neo4j. (n.d.). 5.5. The Triangle Counting / Clustering Coefficient algorithm - Chapter 5. Community detection algorithms. Retrieved from https://neo4j.com/docs/graph-algorithms/current/algorithms/triangle-counting-clustering-coefficient/

NetworkX Developers. (n.d.). Reference — NetworkX 1.10 documentation. Retrieved from https://networkx.github.io/documentation/networkx-1.10/reference/index.html

Page, L., Brin, S., Motwani, R., & Winograd, T. (1998, January 29). The PageRank Citation Ranking: Bringing Order to the Web. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.5427

Patel, A. (2018, November 24). Introduction to A\*. Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Piira, K. (2016, March 21). PageRank Explained. Retrieved from https://www.youtube.com/watch?v=MG0fIXfrT9A

Rogers, I. (n.d.). Pagerank Explained Correctly with Examples. Retrieved from https://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm

Roughgarden, T. (2014, March 31). Counting Triangles. Retrieved from http://theory.stanford.edu/~tim/s14/l/l1.pdf

Slack, K. (2018). Dijkstra's Algorithm. Retrieved from https://synchronic.uat.edu/courses/1893/pages/dijkstras-algorithm?module\_item\_id=162338

Swift, N. (2017, February 27). Easy A\* (star) Pathfinding. Medium. Retrieved from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2