**Introduction:**

The objective of this assignment is to implement the adjacency list and adjacency matrix implementations of the graph ADT. These graphs are used to implement both the breadth-first search and depth-first search algorithms. After implementing these algorithms I compared the runtime of both implementations with increasing vertices as well as increasing and varying edges.

**Theoretical Analysis:**

Both Implementations of the graph ADT had the following functions: addV, addE, removeV, removeE, checkE, getAllE, BFS, and DFS. V is for vertices and E is for edge.

The Adjacency list was implemented using a vector holding nodes. These nodes held a value, index, and a pointer to the next node. These represented directed edges to other vertices while the indices in the vector represented the vertices. The addV function would simply add another index to the vector and give it a node holding its own index and value passed into addV this runs in O(1) time due to only being constant operations. The addE function would create a new node in the linked list of the beginning index passed in and would have the new node hold the index and value of the ending passed in. This takes O(m) time where m is the number of edges currently stored at the beginning index. The removeV removes the desired index from the graph. This takes O(m) time where m is the number of edges associated with the index. The remove function removes the edge starting at the beginning index and ending at the end index both passed in. This takes O(m) time because it will search through the linked list at the beginning index and remove the edge with the associated characteristics. The checkE function checks to see if an edge with a specific start and end index exists. This takes O(m) time because it will search through the linked list at the start index and return true if it exists and false otherwise. The getAllE function prints out all Edges at a specific index. This takes O(m) time because it will go through the linked list at the noted index. For all cases noted, m is the number of edges associated with an index.

The Adjacency Matrix was implemented using a vector holding integer vectors. Each integer was always either 0 if the edge associated with that row, column combination didn't exists and 1 if it did. The addV function would add a new vector of length n+1 where n was the previous length and add a 0 to the end of all the previous vectors. This made sure that the vector was always n x n. This would take O(n) time due to having to add a new value to each vector. The addE function would simply change the value at the row, column combination of the desired edge to 1. This takes O(1) time. The removeV function would remove the desired vector from the overall vector and would remove the associated item from each of the other vectors. This take O(n) time due to having to go through each vector. The removeE function simply changes the value at the desired index to 0. This take O(1) time. The checkE function sees if the associated index is 1 and return true if it is and false otherwise. This take O(1) time. The getAllE function checks the vector associated with the passed starting index and prints all edges where the index is 1. This takes O(n) time where n is the length of the vector.

The DFS algorithm goes down each path associated with a start index and returns a vector of the resulting tree. This take O(n+m) time for the adjacency list and O(n^2) time for the matrix where n is the number of vertices and m is the number of edges.

The BFS algorithm visits each connection of the start index and then continues away, layer by layer. This take O(n+m) time for the adjacency list and O(n^2) time for the matrix where n is the number of vertices and m is the number of edges.

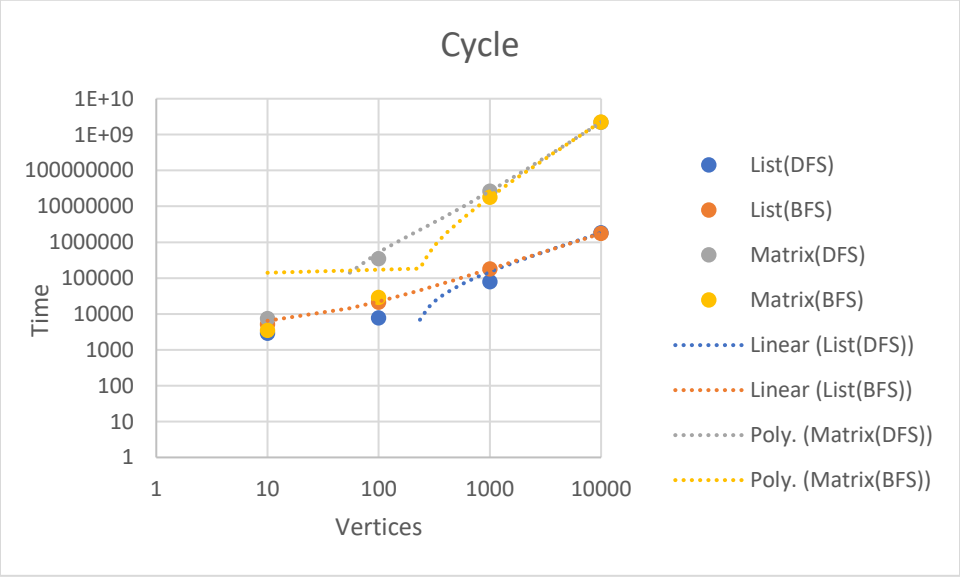
**Experimental Setup:**

To perform the experiments if this assignment I used a Dell Inspiron 5577 with an Intel Core i7-7700HQ CPU @2.80 GHz. This laptop has 8 GB of RAM and a 64-bit operating system. For the compiler I used Cygwin v3.0.7 and CMake v3.14.5 in combination with the CLion 2019.2.1 IDE. For this program I used C++14.

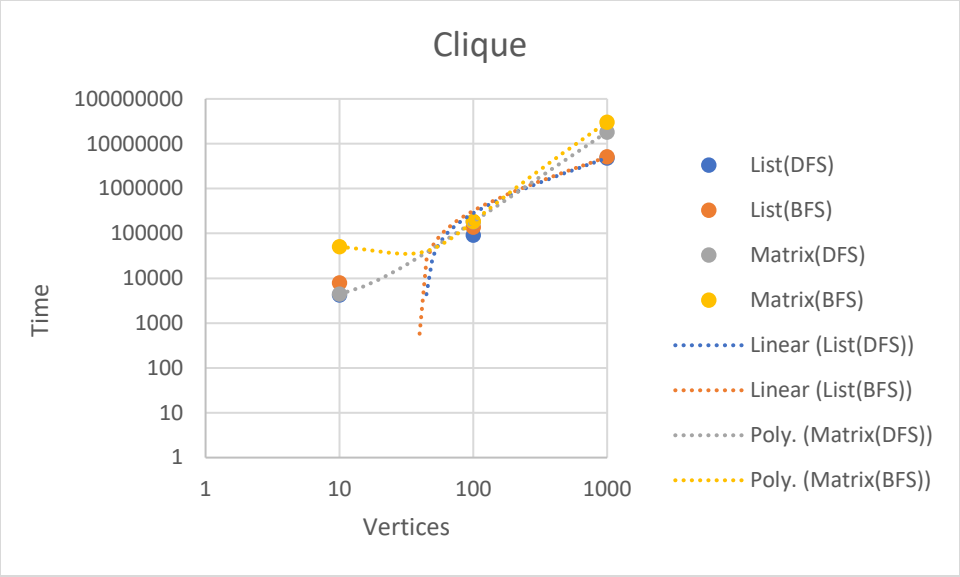
To measure the amount of time to sort n elements I used the high_resolution_clock from the C++ chrono library. I set it to nanoseconds in order to get accurate measurements on the lower test cases. I chose nanoseconds because milliseconds and coarser measurements did not give accurate results below 1000 elements.

I ran each  implementation at 10, 100, and 1000 vertices. Whichever ones could be run with 10,000 vertices I recorded, but not all would complete when run.

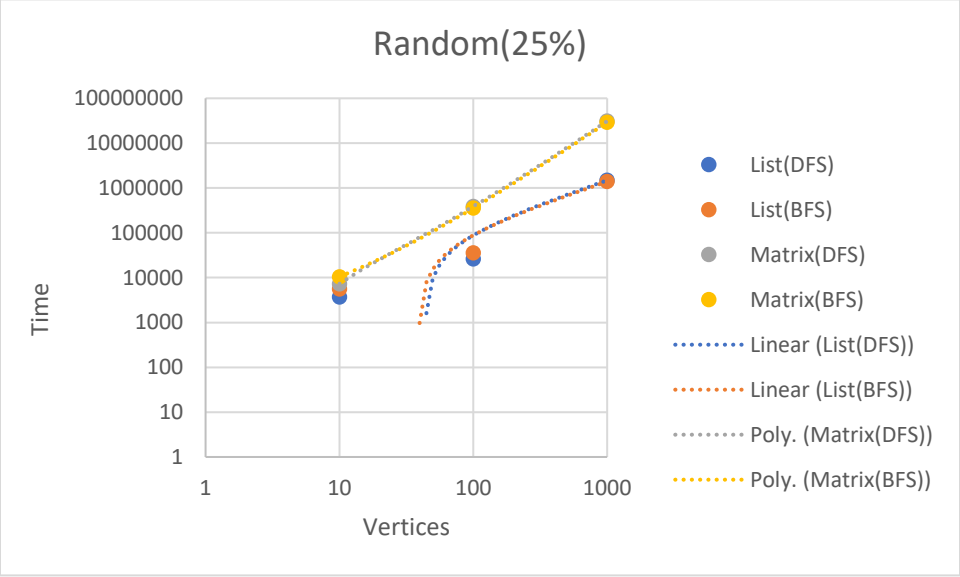
**Experimental Results:**



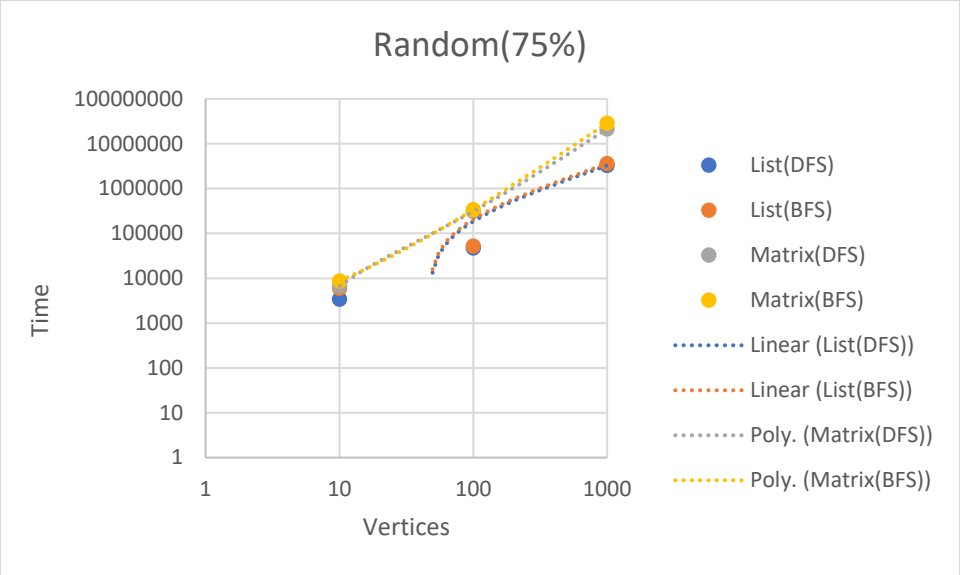

This graph shows the runtime of both implementations and search algorithms as the number of vertices increases. In this case the only edges were connecting each vertex to the next and the last to the first. The results show how the matrix implementation is much slower than the list implementation, especially at higher numbers of vertices.




This graph shows the runtime of both implementations and search algorithms as the number of vertices increases. In this case the edges connected every vertex to every other vertex. The results show how the matrix implementation is still slower but not significantly so when the number of edges  increase.




This graph shows the runtime of both implementations and search algorithms as the number of vertices increases. In this case each vertex has a 25% chance of connecting to any other vertex.The results show how the matrix implementation is much slower than the list implementation, especially at higher numbers of vertices.




This graph shows the runtime of both implementations and search algorithms as the number of vertices increases. In this case each vertex has a 75% chance of connecting to any other vertex.The results show how the matrix implementation is slower but not much so when the number of edges is higher.