DATA STRUCTURES AND ALGORITHMS

PROGRAMMING EXAM

1. **Write a program that computes the "diameter" of a directed graph which is defined as the maximum-length shortest path connecting any two vertices. Estimate the runtime of your algorithm.**

Ans: The approach I had used for this problem was the Dijkstra's algorithm, since it is used to obtain the shortest path distance from a source vertex to the other vertices in the graph.

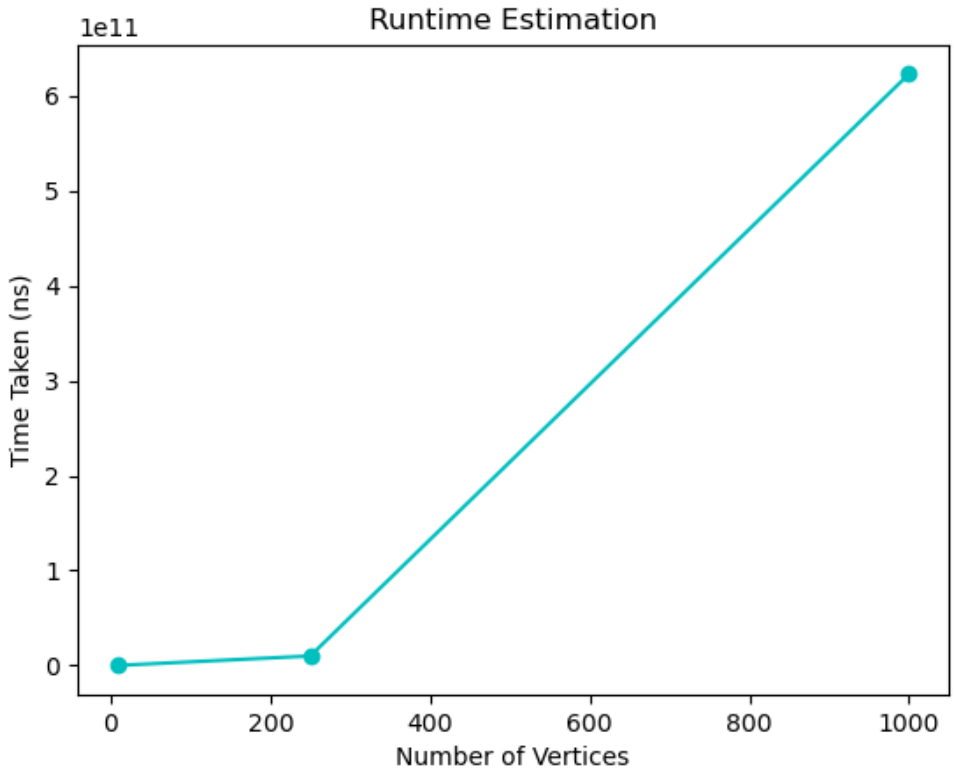
First, I encountered an error in reading the dataset given, which I realized was due to the extra space given for the 2-digit vertices (for mediumEWD.txt, 1000EWD.txt and 10000EWD.txt). To resolve this, I wrote a code that would format the dataset in such a way that a single space exists between each vertex listed in the text files. These are labelled as file2.txt, file3.txt and file4.txt.

Since the question mentions that the diameter of a graph is the maximum-length shortest path, I obtained the shortest path distances from every vertex in the graph to another, which I returned as an array to the *get_diameter()* function. I then took the maximum value from the array (using the in-built *max()* function) and obtained the diameter of the graph.

While running the program, I found that for the graph with 10,000 vertices (file4.txt), the program took a really long time to display the diameter. Due to this and the time constraint, I displayed the diameter I had obtained for the 3 text files, which are shown below:

```
Importing data from  tinyEWD.txt into the graph...

For 8 vertices, diameter is = 1.49
Time taken = 999900 ns.

Importing data from  file2.txt into the graph...

For 250 vertices, diameter is = 0.98
Time taken = 10003310400 ns.

Importing data from  file3.txt into the graph...

For 1000 vertices, diameter is = 1.11
Time taken = 623198992600 ns.
```

I used the matplotlib.pyplot library to plot the variation of runtime versus the number of vertices in the graph. The graph obtained is shown below:



The above graph shows the variation of the run-time in nanoseconds versus the number of vertices present in the graph. The underlying data structure I am using to store the vertices of the graph is an array, so the time complexity for this algorithm is O(V^2), based on the above graph.

2. **Write a program to find values of** "A" **and** "M"**, with M as small as possible, such that the hash function** (A*K) modulo M**, for transforming the** K**th letter of the alphabet into a table index produces distinct values for the keys** S E A R C H X M P L**.**

Ans: The approach I used for this problem was first, use a dictionary to store the keys along with their corresponding values, and then extract the values from the dictionary for further implementation.

I created a function *hash_table()* which creates the hash table based on the given hash function. Since the only given conditions are that M has to be minimum and the hash values should be distinct, I chose a range of values from 2 to 30. Next, since the hash values have to correspond to the letters of the alphabet, I set M to have a range from 1 to 26.

Now, this function basically loops through A and M, along with the values of the input keys, and computes the hash values for each value. I add all these values into an array and create arrays to store the corresponding A and M values. I split the list into a list of lists with each list corresponding to a particular value of A and M. I then return this list of lists back to main.

I created another function *is_distinct()* to check if the hash values are distinct. In this function, I iterated through the list of lists and check if the elements in each list of the list of lists repeat or not. I do this by declaring a set for each element in that list and compare with the original list, since a set can only hold unique elements; it does not allow duplicate values. So, if the length of the original list is equal to the set list, that means that there are no duplicates. It then returns the list, value of A and M to the main.

Initially, I had taken A from 1 to 31, but what I saw was when A was 1 and M was 20, the hashed values were very similar to the original values. For this, I set the range of A from 2 to 31.

The final result I obtained is shown below:

```
Original values are: [18, 4, 0, 17, 2, 7, 23, 12, 15, 11]

Results
A = 3, M = 20, Hash table values = [14, 12, 0, 11, 6, 1, 9, 16, 5, 13]
```