

Midterm 2: Pratik Mistry – DSA Spring 2020 (pdm79)

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. Feel free to use the following datasets:

<https://algs4.cs.princeton.edu/44sp/tinyEWD.txt>

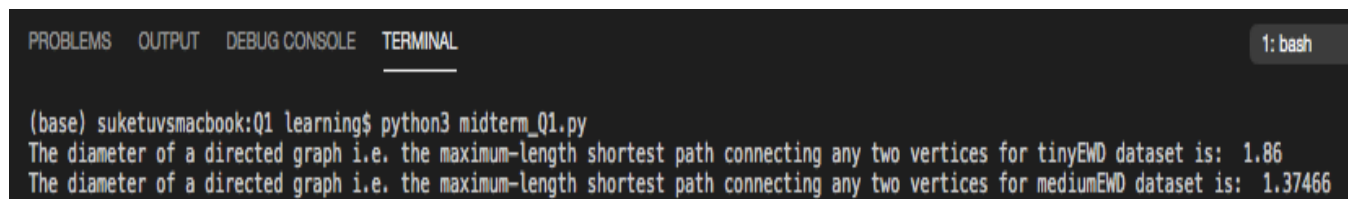
<https://algs4.cs.princeton.edu/44sp/mediumEWD.txt>

<https://algs4.cs.princeton.edu/44sp/1000EWD.txt>

<https://algs4.cs.princeton.edu/44sp/10000EWD.txt>

Solution 1:

- The dataset to be used is given in the URLs above.
- I have used standard implementation of Dijkstra's algorithm for computing the shortest path in the graph for given source vertex
- For computing the "diameter" of a directed graph i.e. the maximum-length shortest path connecting any two vertices, we execute Dijkstra's algorithm for each vertex of the graph as source vertex and find the shortest paths.
- Now, in each iteration we find the maximum-length shortest path and the "diameter" of a directed graph is the maximum of maximum-length shortest path from all iterations
- Now, the runtime complexity of Dijkstra's algorithm is $O(N) = E \cdot \log(V)$
- Since we are finding the diameter of digraph by running Dijkstra's algorithm on all the vertices of the graph, the total time complexity of the algorithm is $O(N) = V \cdot E \cdot \log(V)$
- I have executed program for only first two datasets i.e. tinyEWD.txt and mediumEWD.txt, below is the screenshot of the output.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash
(base) suketuvmacbook:Q1 learning$ python3 midterm_Q1.py
The diameter of a directed graph i.e. the maximum-length shortest path connecting any two vertices for tinyEWD dataset is: 1.86
The diameter of a directed graph i.e. the maximum-length shortest path connecting any two vertices for mediumEWD dataset is: 1.37466
```

- Thus diameter of directed graph for tinyEWD.txt dataset is **1.86** and for mediumEWD.txt dataset is **1.37466**
- The reason why I haven't executed program for 1000EWD.txt and 10000EWD.txt datasets is because my machine takes lot of time due to low configurations.
- But, I have provided functionality to execute the program for whichever dataset you would like to execute.

2. Compute the

(i) number of connected components

(ii) size of the largest component,

(iii) number of components of size less than 10 for data provided at:

<https://algs4.cs.princeton.edu/41graph/movies.txt> .

Estimate the runtime of your algorithm.

Solution 2:

- There are three main steps for finding the number of connected components:
 - Build symbol table for the dataset
 - Build undirected graph using symbol table
 - Run connected components algorithm
- For the first phase, we build symbol table for the dataset given where each movie/cast name will be inserted into symbol table list if it does not exist and each movie/cast name will have an index position as a key
- Thus when I search for the key for the given movie/actor name, symbol table API returns the respective index position of the symbol table list
- In the second phase, we read the dataset again i.e. read each movie and its casts name and get the index position in the Symbol Table list using symbol table API
- Using the indexes as vertices of undirected graph, we build the undirected graph where each vertex is index of either movie or cast name and each edge is movie and each cast of it
- Once undirected graph is created, we run the connected components algorithm to find all the connected components in the undirected graph created
- Connected Components algorithm is basically running Depth First Search (DFS) on each vertex of the graph and finding all the connected components
- Thus, all the vertices connected with current vertex on which DFS is called will have same component ID
- For performing DFS for the given dataset, recursive solution does not work as it gives Maximum Recursive Limit error i.e. machine's stack gets filled up entirely while program is still running thus resulting in Segmentation Fault issues
- To counter this problem I used *LIST* in Python as a Stack where operations like *list.append()* adds value in the top of stack and *list.pop()* removes/pops out least recently added value i.e. top of the stack value.
- Thus, it works as when vertex is visited it is added on stack. Then remove top vertex and add all its adjacent vertices (not visited) into the stack. Again, we remove the top vertex and further perform DFS for adjacent vertices. The idea works same as that of DFS done recursively where machine uses built-in stack memory store the nodes visited.
- I have also provided 3 API's in Connected Component class to get the total number of connected components, size of the largest components, number of components less than 10. The corresponding outputs are **33**, **118774** and **5** as seen in below screen shot.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

(base) suketuvs-macbook:Q2 learning$ python3 midterm_Q2.py
Total number of components are: 33
Size of largest components is: 118774
Number of components less than 10 for data provided are: 5
```

- Now discussing about the time complexity, consider there total N movies and M casts in the entire dataset
- Now, in the worst case each movie (N) will have all cast (M) in it and thus time required to build the symbol table will be $O(N*M)$
- Thus, we then build the undirected graph using symbol table and dataset given, the total time complexity to build graph will be $O(N*M)$
- Further, as we used DFS to compute the connected components, the total time complexity of DFS is $O(E + V)$ where E is Edges and V is Vertices
- Now, since vertices of graphs are all the movie names and casts, the total vertices V is $N+M$
- Also, since in the worst case we discussed each movie will have all casts the total edges will be $N*M$
- Thus, complexity of DFS is $O(V + E) = O((N+M) + (N*M))$
- Since, the complexity of DFS to compute the connected component dominates the total runtime of the algorithm, we can say that the overall complexity is the complexity of DFS discussed above i.e.

$$O(V + E) = O((N+M) + (N*M))$$

Where $V = N+M$ = total vertices and $E = N*M$ = total edges