Assignment-4 Report

Task-0: Building the graph

In this assignment, two files are given to read from to work on it. One is having information about vertices with total number of vertices on line1 and other lines have the vertices id with the word which that vertex represents.

Another file has the information about edges where the two numbers in the file represents an undirected edge representing the two vertices with those IDS.

In my task, the graph is implemented in __init__ method to be used in later tasks. There is the array inside an array created to store the graph as it represents the adjacency list of the vertices in the file. To do that, the file is read line by line and the words in the file are separated using rstrip() to store it inside the array created in the beginning.

To make sure, we don't mix up with the number of vertices in line no. 1, we will take the value out and separate other vertices information into different array.

The same process of file is done on the second file containing edge information. The edges information is stored into an array inside tuple. The adjacency list is created just using the edges array and appending the values of corresponding connected vertex to the given vertex to make sure these vertexes are connected to the given current vertex.

If my_graph is used to store the developed graph, my_graph.adj_list should return the graph.

The time taken to find those vertices is as below:

Time complexity: O(V + E),

Where, V is the number of lines in vertices_filename

E is the number of lines in edges_filename

As we go through the vertex file and put it into the array to work on it. We get the words being represented for the vertices in this file which will be useful for task-1 implementation. So, the complexity will be V as above and addition of E, again as above.

Task-1: Solving a ladder

In task-1, Given a word ladder puzzle (i.e. a start_vertex and target_vertex, which each have a word), the function solve_ladder which finds the shortest list of intermediate words which solves it (or reports that no such list exists) is written that returns a list of the intermediate words. For a given pair of start_vertex and target_vertex, there may be several equally short lists of intermediate words.

The method written returns the possible path from start vertex to the final vertex. The main algorithm being the attraction in this task is BFS as known as Breadth first search. The logic is that only one letter out of three letters in the word changes at each vertex visit. So, it is best to use BFS such that all the next words can be kept track of while going towards the last vertex.

Like the classic BFS method, there's an array having a size of the graph where all the false values are allocated for each vertex except the starting vertex as obviously, we are only on the first vertex. As we go through into the graph using BFS to check if the next vertex is the valid option to visit, the vertex being visited will get true values. Starting from the starting vertex, queue is used to keep track of current vertex. Adjacency list is used to allocate the connected vertex and another stored array will be used to allocate representing word to search through list.

Time complexity: O(V + E) ,

Where, V is the number of vertices in the graph

E is the number of edges in the graph

As we need to look through adjacency list and the possible words in the file which will both take combined time of V+E as above.

Task-2: Restricted word ladder

For the given task, the focused algorithm will be Dijkstra's algorithm but the data structure is mainly to fit it in the form of the cheapest word size according to the problem given in the assignment. Min-heap is designed to get the condition of how the words are compared in terms of size and it will be used while designing method named solve_ladder.

Min-heap is used to compare between the cheapest ladder as I have used rise() and sink() to move these distance values up and down the heap. After that, I used Dijkstra's algorithm to get the shortest path using start and target vertex. It will return the possible paths from start vertex to every other words.

Ord() is used to compare between the letters according to algorithm in the document. As min-heap takes log(V) time to rise and sink, also going through edges:

Time complexity: $O(E \times \log(V))$ time,

Where, V is the number of vertices in the graph

E is the number of edges in the graph