1.Introduction

1.1 Objectives
We are given an undirected, unweighted graph which represents a city's road network with V nodes being buildings among which there are H hospitals, and E edges being roads. Our objective for the project is to propose algorithms that find, for each node, the path to the nearest hospital or the distances to the k nearest hospitals.

1.2 Overview
In this report, we will discuss the design and implementation of our modified BFS algorithm and how its performance varies asymptotically (time and space complexity wise) as the value of k and H are varied.

2.Algorithms and analysis

2.1 Design and implementation

Single source BFS starts at any node on the graph and traverses all neighbouring nodes before moving on to their neighbors, i.e. it checks all the nodes at a distance of d before moving on to the nodes at a distance of d+1 from the root node. There is no specific order which is followed when exploring nodes at a particular depth. The algorithm is implemented so as to ensure that the every node is visited only once and every edge twice, once when going forward to explore the nodes and the second time while backtracking to check if the node has already been marked. However, implementing single source BFS with multiple hospitals would lead to the same nodes being traversed multiple times when we start checking distances from each node. To solve this problem and reduce time complexity, we will use multi-source BFS to have multiple instances of BFS run concurrently and break early.

We will be using an adjacency dictionary (**pointerDict**) to store the graph. Output will be in the form of a distance dictionary (**distDict** or **topKDist**) of size V or VH depending on the problem. We also use another dictionary (**checked**) of the same dimensions to store whether the distance of a particular node from a particular hospital has been checked already.
<u>Although this algorithm could be implemented with lists with the same time complexity, we used dictionaries instead because the real road dataset did not have clean node indexes.</u>

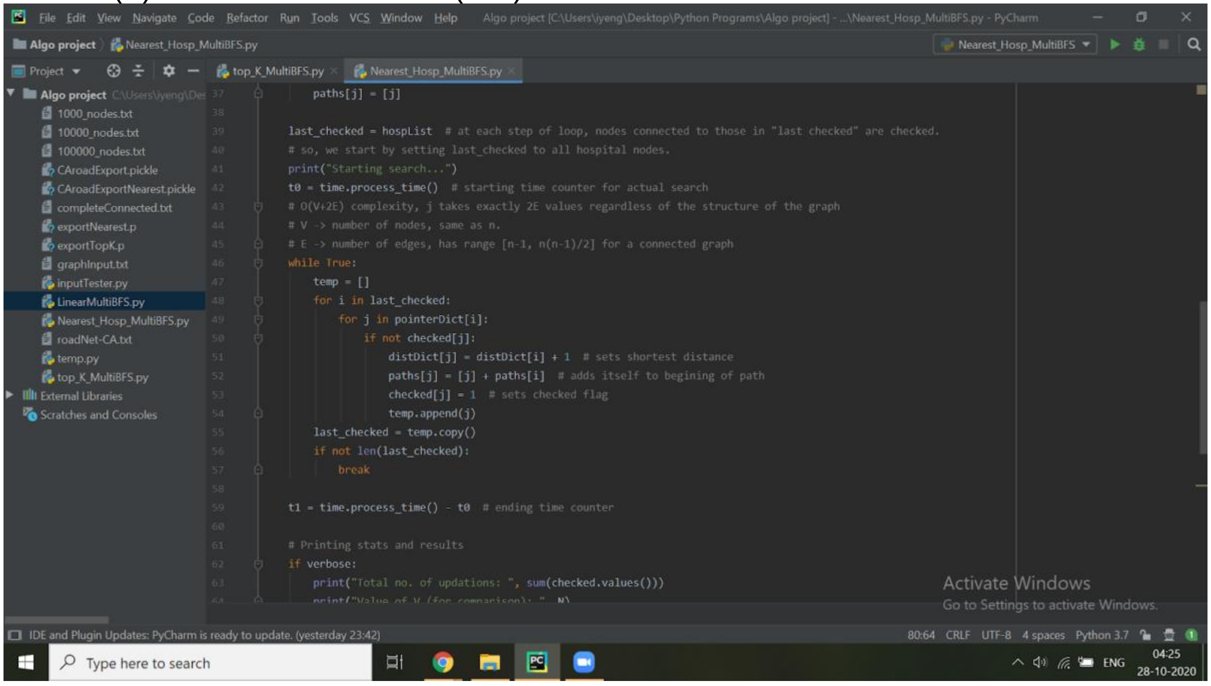
2.2 Shortest path to nearest hospital (independent on number of hospitals)

2.2.1 Time Complexity analysis *(Constant regardless of graph structure)*
To analyse the time complexity of our algorithm, we will look at the number of updates to the distances from hospital nodes, as well as the number of edges traversed. Initially all the indexes of the hospitals are stored in a list, namely **hospList (Figure 1)**, and their distances are set to 0. Next, at the **i**th "step" of the multi-source BFS, the neighbors of the nodes checked in the **(i-1)**th step *that have not been checked yet* are visited, with their distance set to i.

Looking at the code below, we can see that regardless of how the graph is traversed, j can take exactly as many values as are there in pointerDict, which equals **exactly 2E**. This value represents the number of **edge traversals**. However, the code block only executes if the node hasn't been checked already, so the code block executes **exactly V times**, relating to the number of **node traversals**. Assuming the code block is constant time c1 and the if statement is c2, then the time complexity is **O(V.c1 + 2E. c2)**. This can be estimated to be **O(V + 2E)**. Since the same steps execute regardless of graph structure, <u>this is both best and worst case</u>.

2.2.2 Space Complexity analysis

We have that pointerDict has exactly 2E items, distDict and checked are both dictionaries of length V, and "last_checked" is only a fraction of V, so space complexity for this algorithm is O(V + 2E). This excludes the space complexity of storing paths, which depends on the graph and is O(V) in the best case and O(V^2) in the worst case.



```python
            paths[j] = [j]

    last_checked = hospList  # at each step of loop, nodes connected to those in "last checked" are checked.
    # so, we start by setting last_checked to all hospital nodes.
    print("Starting search...")
    t0 = time.process_time()  # starting time counter for actual search
    # O(V+2E) complexity, j takes exactly 2E values regardless of the structure of the graph
    # V -> number of nodes, same as n.
    # E -> number of edges, has range [n-1, n(n-1)/2] for a connected graph
    while True:
        temp = []
        for i in last_checked:
            for j in pointerDict[i]:
                if not checked[j]:
                    distDict[j] = distDict[i] + 1  # sets shortest distance
                    paths[j] = [j] + paths[i]  # adds itself to begining of path
                    checked[j] = 1  # sets checked flag
                    temp.append(j)
        last_checked = temp.copy()
        if not len(last_checked):
            break

    t1 = time.process_time() - t0  # ending time counter

    # Printing stats and results
    if verbose:
        print("Total no. of updations: ", sum(checked.values()))
        print("Value of V (for comparison): ", N)
```

Figure 1 : algorithm for the nearest hospitals

2.3 Distance to nearest k hospitals

(This algorithm represents our solution for the nearest 2 problem as well)
The algorithm shown above was modified to be generalized for the case of top-k hospitals. The general idea is the same. However, instead of stopping checking a node when its nearest hospital is found, we instead stop checking a node after it's nearest k hospitals are found. Hence, each node is visited exactly k times and this is an improvement over the more naive multi-source BFS approach that would have had H visits per node instead.

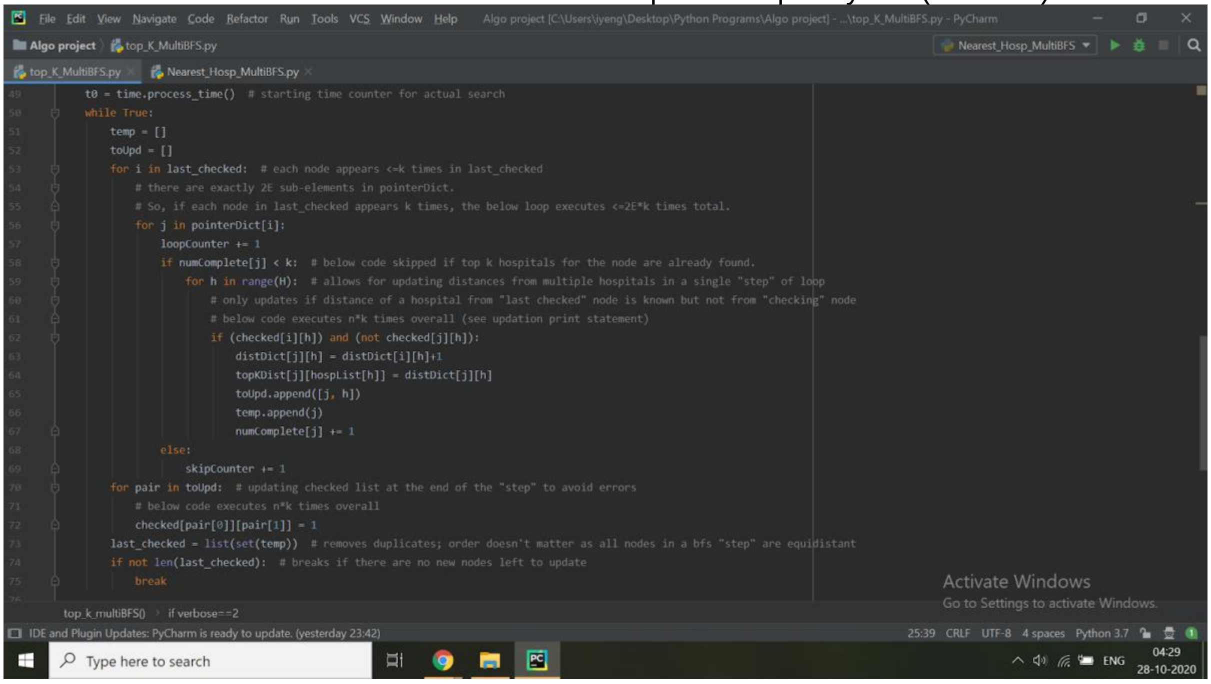
2.3.1 Time Complexity analysis *(Constant regardless of graph structure)*

Once again, we will be using both number of node visits as well as edge traversals for obtaining the theoretical time complexity. Much like before, initially all the indexes of the hospitals are stored in hospList, and their distances *to themselves* are set to 0. Next, at the ith "step" of the multi-source BFS, the neighbors of the nodes checked in the (i-1)th step *that have less than k hospitals found so far* are updated only for the values where *the distance of a hospital to the last_checked node is known but to the current hospital is not known*.

Hence, each time a node is visited, distances to *at least one* hospital are updated, and hence **each node is visited at most k times**. So, the code block represented by constant time c1 is executed at most Vk times. However, due to the implementation of the checking segment, the if statement represented by c2 is executed **at most 2EHk times**.
So, the overall time complexity is **O(V.k.c1 + 2E.H.k.c2)** which can be estimated as **O(k(V + 2EH))**. Practically speaking, since **c1 >> c2** and assuming that **V, 2E >> H** with graph density remaining approximately constant (i.e. **E∝V**), we can say that **asymptotic time complexity is of order O(V.k)**. This is further proved by our empirical analysis on a real road dataset, where change in H only causes a small constant time difference.

2.3.2 Space Complexity analysis

Similar to nearest hospital case in 2.3.2, however in this case our distDict and checked dictionaries both have dimensions VH so overall space complexity is O(2E + VH).



```python
        t0 = time.process_time()  # starting time counter for actual search
        while True:
            temp = []
            toUpd = []
            for i in last_checked:  # each node appears <=k times in last_checked
                # there are exactly 2E sub-elements in pointerDict.
                # So, if each node in last_checked appears k times, the below loop executes <=2E*k times total.
                for j in pointerDict[i]:
                    loopCounter += 1
                    if numComplete[j] < k:  # below code skipped if top k hospitals for the node are already found.
                        for h in range(H):  # allows for updating distances from multiple hospitals in a single "step" of loop
                            # only updates if distance of a hospital from "last checked" node is known but not from "checking" node
                            # below code executes n*k times overall (see updation print statement)
                            if (checked[i][h]) and (not checked[j][h]):
                                distDict[j][h] = distDict[i][h]+1
                                topKDist[j][hospList[h]] = distDict[j][h]
                                toUpd.append([j, h])
                                temp.append(j)
                                numComplete[j] += 1
                    else:
                        skipCounter += 1
            for pair in toUpd:  # updating checked list at the end of the "step" to avoid errors
                # below code executes n*k times overall
                checked[pair[0]][pair[1]] = 1
            last_checked = list(set(temp))  # removes duplicates; order doesn't matter as all nodes in a bfs "step" are equidistant
            if not len(last_checked):  # breaks if there are no new nodes left to update
                break
```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
Algo project [C:\Users\iyeng\Desktop\Python Programs\Algo project] - ...\top_K_MultiBFS.py - PyCharm
Algo project
top_K_MultiBFS.py
Nearest_Hosp_MultiBFS.py
Nearest_Hosp_MultiBFS
top_k_multiBFS()
if verbose==2
IDE and Plugin Updates: PyCharm is ready to update. (yesterday 23:42)
Activate Windows
Go to Settings to activate Windows.
25:39 CRLF UTF-8 4 spaces Python 3.7
Type here to search
ENG
04:29
28-10-2020

Figure 2 : algorithm for k nearest hospitals, k>1

<u>2.4 Empirical analysis</u>

V = 1,965,206 (Real Road Dataset,roadNet-CA)

<u>Algorithm</u>		**<u>Runtime in seconds (s)</u>**		
		<u>H=5</u>	**<u>H=10</u>**	**<u>H=20</u>**
<u>Nearest Hospital Multi BFS</u>		5.23	4.92	4.36
<u>Top k multiBFS</u>	k=2	16.80	19.11	24.14
	k=3	24.47	27.69	36.98
	kr=4	33.62	38.05	50.58
	k=5	42.31	50.12	64.70

Table 1: Empirical analysis of the algorithms for real road dataset

V = 100,000 (Randomly Generated Network)

Algorithm		**Runtime in seconds (s)**		
		H=5	**H=10**	**H=20**
Nearest Hospital Multi BFS		0.30	0.31	0.31
Top k multiBFS	k=2	1.19	1.25	1.44
	k=3	1.67	1.81	2.06
	k=4	2.28	2.28	2.56
	k=5	2.69	2.69	3.00

Table 2: Empirical analysis of the algorithms for randomly generated network

2.5 Conclusion

The tables above show the runtime of the two different algorithms we proposed - for the nearest hospital, and for the top k nearest hospitals as the value of k is changed, when the number of hospital nodes (H) is 5, 10 and 20. Table 1 shows a Real Road network, with approx. 2 million vertices, and table 2 shows a Randomly generated graph with 100,000 vertices. We can clearly see that as the number of nodes is reduced to around 1/20th, the time taken also reduces to 1/20th in both the algorithms, which means that both algorithms are linear with V. We notice that for the top k nearest hospitals in both algorithms, the time varies linearly with k. As we vary H, there is a constant time difference added, which is due to the small 2E.H.k.c2 component theoretically explained in *Section 2.3.1*. We can also see that the time complexity of the nearest hospital algorithm is about constant with H. In fact, it performs slightly better with a higher H, as on average the paths become shorter so updating and returning the paths becomes faster.

3.1 References

For generating Random graphs: https://networkx.org/documentation/stable/index.html
All ideas and code for the actual algorithms were thought of and implemented by ourselves, no external resources or websites were used