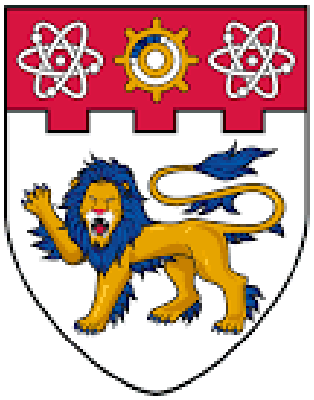


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Course: CE2001 Algorithms

Project 2: Graph Algorithms

Lab Group: SEP2

Chan Ming Kai, Alvin (U1921305J)

Krithika D/O Ramamoorthy(U1720935A)

Oui Wein Jien(U1840300H)

Palaniappan Sneha (U1722528E)

Venkat Subramanian (U1921075D)

Yu Xuan (U1922631L)

(Need to Insert Contents Page) (I do later) - alvin

Project 2

Background

For project 2, we are tasked to design an algorithm that is able to find the shortest path from one node (n) to its nearest hospital in an undirected unweighted graph (G). The algorithm is supposed to take in 2 inputs, the graph and the list of hospitals. The algorithm is then tested on a random graph and a real road network.

We used python to develop the algorithm and made use of additional packages such as networkx.

1. Part (a) and (b)

Code implementation

For this project, to find the nearest path we made use of Breadth First Search (BFS). To create the unweighted and undirected graph, we read the starting and ending nodes from the road network. Using the add_edge function, we are able to add the nodes and the edge into the graph.

To make the algorithm independent of the number of hospitals, we had to do some preprocessing on the hospital array obtained from reading the list of hospitals in the text file. We first created a visited array that has the length of the graph and those nodes that are not a hospital are initialized false. The hospital nodes are initialized to a value of -1 in the visited array. The copy of this array is then sent to the bfs_b algorithm.

The bfs_b algo takes in 3 inputs, the graph, the source node, and the visited array. In the bfs_b algorithm, we will create a queue list which will hold all the paths from the source node (this is stored as a nested list). The algorithm is repeated until the queue is empty. In each of the iterations, we will pop the first element from the queue and obtain a new source node by accessing its last element from the popped out list. Then each of the source nodes is first checked if it's a hospital. This is done so by checking its value in the visited array. If it is -1, it is deemed as a hospital. Since there will be no shortest path from a hospital, the hospital node is returned. If the source node is not a hospital, we will look at all the nodes adjacent to it. A ‘for’ loop is run to go through all the adjacent nodes of the source node. In the ‘for’ loop each of the adjacent nodes is checked if it has not been visited(the value of its index in the visited array is False), we will change the value of the node in the visited array and add the node visited to the shortest_path array which will be then appended to the queue. If the adjacent node is determined to be a hospital (The value of the node

in the visited node is -1), the node is appended to the shortest path and then the shortest path is then returned.

Time Complexity Analysis

Best case:

The best-case occurs when all the nodes are hospital nodes. Then each of these nodes is only compared once with the value of -1, therefore the time complexity of running the algorithm on one of the source nodes is O(1), This process is repeated for every node, therefore the time complexity would be O(V*1) = O(V), where V is the number of nodes in the graph. There is also a preprocessing component involved, where we need to create a visited array. If we assume that the time taken to assign -1 to each of the hospitals is C1, then the overall time complexity of the algorithm would be $O(V + C_1 h) = O(V + C_1 V) = O(V)$ since in the best case is when all the nodes are hospital nodes, which is h = V.

Average Case:

In the average case, the bfs_b search tree that we obtained from our algorithm will be a subset of the search tree when a normal Breadth-First-Search(BFS) is done. Therefore we can assume that the average time complexity of bfs_b will be a multiple of BFS's time complexity.
The time complexity of a normal BFS search on a single source node is $O(V + E)$. Since the average time complexity of bfs_b is a subset of BFS, the time complexity of bfs_b on a single node is $O(J(V + E))$, where J is an arbitrary constant.
Therefore the average time complexity to find the shortest path for all the nodes will be $O(V.J(V + E))$

$Average\ time\ complexity = O(h) + O(V.J(V + E)) = O(h) + O(V^2 + EV)$

And since the range of h is between 0 and V, the overall time complexity will be much larger than the time taken to preprocess the array. Therefore the overall time complexity will be independent of the number of hospital

$Average\ time\ complexity = O(V^2 + EV)$

Worst Case:

The worst case occurs when

1. The hospital doesn't exist in the source graph
2. The hospital is not connected to any of the nodes

In the worst case the algorithm will traverse through the entire graph, resulting in the time complexity of finding a hospital is O(V+E) .

If we assume that the number of edges is the maximum which is $\frac{V^2-V}{2}$.

$$Time\ complexity\ =\ O(V(\ V\ +\ E))\ =\ O(V(\ V\ +\ \tfrac{V^2-V}{2}))$$
$$=\ O(V^2\ +\ \tfrac{V^3-V^2}{2})$$
$$=\ O(V^3)$$

2. Part (c) and (d)

Code implementation

The code implementation for part c and part d is similar to the previously mentioned algorithm. The only addition to the previous algorithm is the additional input which is the number of nearest hospitals to be displayed. And there is an additional queue to keep track of all the paths to a hospital (queue_2). And unlike the queue in the first algorithm there is no deletion of elements in this queue. Due to these changes the number of comparisons have increased by 3 for each call of the algorithm. One of the comparisons is to check if queue_2 is an empty list. The second comparison is to check if the hospital node is already added into the queue_2 by checking its recently added node with the current node. And the third comparison is to check whether the size of queue_2 is equal to the number of hospitals the user specified.

Time Complexity

Since we are repeating the algorithm to get the top k path to k distinct hospital, the time complexity of bfs_c will be a k times of that in bfs_b.

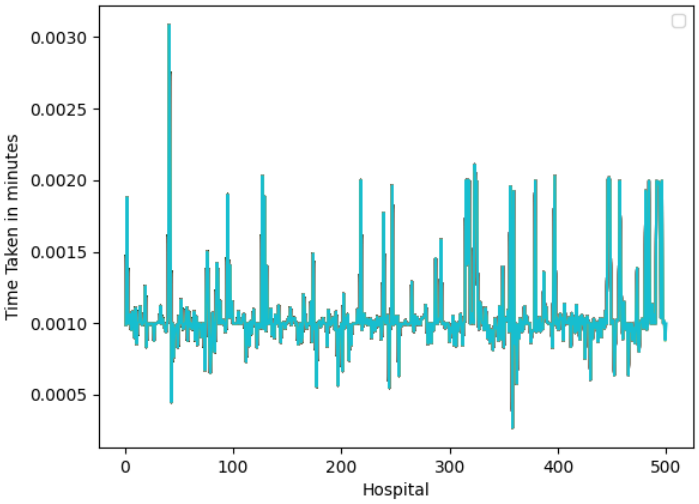
Best case : $O(k.V)$

Worst case : $O(kV^3)$

Average case : $O(kV^2\ +\ kEV)$

Proving correctness of the Algorithm

We tested the algorithm on a random graph that has 10 nodes, and we manually checked if the algorithm is able to return the shortest path from the source node to the nearest hospital. We also tested this on a larger sample, using a BFS algorithm which takes in the source node and the destination. We used a nested loop to send in all the nodes and all the hospitals. Therefore the BFS algorithm will find the shortest path from one node to all of the hospital. We then compared this result to the output from our own algorithm , all of the result obtained was similar to code that we adapted, proving the correctness of our proposed algorithm.

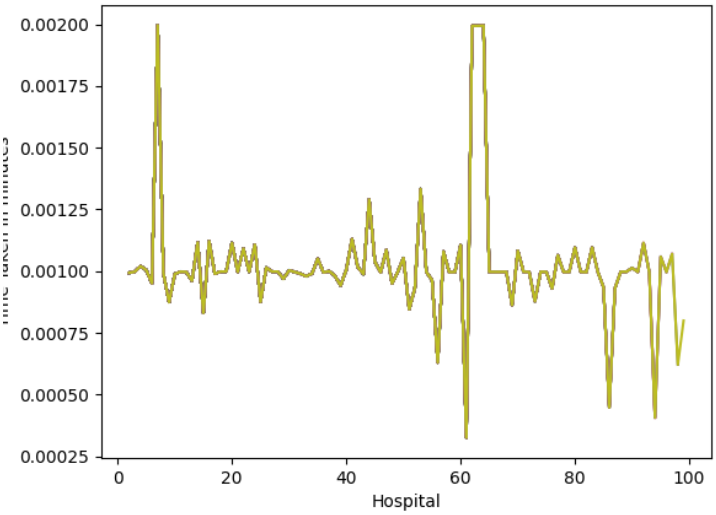


Effects of h and k on the time taken

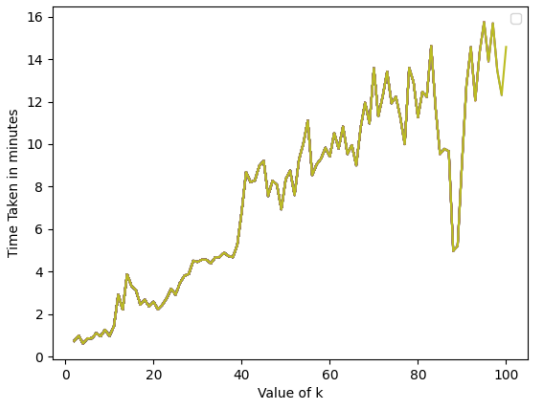
To see the effects of how the number of hospitals would affect the time complexity. We fixed the source node, and increased the number of hospitals present. If we were to remove the noise, we can see that most of the data takes approximately the same time. Therefore, even when the number of hospitals are increased, the time taken to find the shortest path from a fixed source node is relatively the same as when we used bfs_b.

2. Effects of h and k on bfs_c

To see the effect of h, on the time taken on bfs_c we kept the value of k to 2 and kept the source node constant.



From the graph we can see that, when the source and the value of k is fixed, the taken for most of the data points are the same. Implying that the time complexity for bfs_c is independent of the number of hospitals



Effect of k on the time complexity
When we increase the value of k, from o to 100 we can see that the time taken to find the k path to the shortest path, increases exponentially.

Limitations
The limitations of this algorithm is that we have to use an adjacency list (in python it's in a dictionary format) rather than an adjacency matrix since there is a space limitation. And the other possible limitation is that, if there are 2 hospitals that have the same distance (same number of edges from the source node) the hospital with a smaller node index is returned. Another limitation is that in bfs_c , if k is greater than the number of hospitals reachable from the source node it will return None.

Space limitation(max size of algo input.)

Python, the language in which we implemented the algorithm, does not have a maximum file read size. Python stores string variables in the ram. Hence, the maximum file size is determined by the memory allocated to the program.

A 32 bit-python installation would only have access to ~4GB of ram, which can further decrease based on other factors, such as physical memory available and operating system overhead.

In our algorithm, incurred preprocessing cost as we created the list of nodes, O(V).

The algorithm is has a space complexity of O(V+E) where V is the number of nodes, and E is the nodes linked to the respective list.

As a result this results in a space complexity of O(V) + O(V+E) = O(2V+E) = O(V+E)

The actual max size is dependent on a host of factors, including but not limited to, architecture of cpu, the physical memory of the machine, OS, and program memory allocation settings of the machine.

Test on real world data

Possible improvements

Fringe cases

Conclusion

References

Python Patterns - Implementing Graphs. (n.d.). Retrieved October 27, 2020, from https://www.python.org/doc/essays/graphs/

Statement of Contribution - Group 1

The main components of the project were research, coding, drafting of report, slides and presentation. Everyone contributed to each component. Venkat, Wein Jien and Alvin presented the project. Everyone contributed in overseeing and cleaning up all components of the project.

BFS(a)-(d):
Report:
Slides: Sneha, Krithika

Word Count: 79/100

(Need to update)