Project 2: Graph Algorithms

CZ2001 Algorithm

Lab/Group: SSP3 Group 1

Ang Shu Hui	U1922145K
Bachhas Nikita	U1921630F
Kundu Koushani	U1922997B
Leonardo Irvin Pratama	U1920301J

Nanyang Technological University

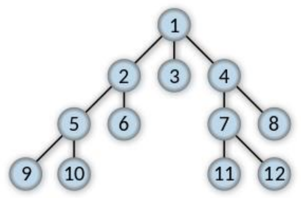
Singapore, October 2020

Problem Introduction

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices/nodes, and the links that connect the vertices are called edges. It is an abstract data type that is meant to implement both the directed and undirected graphs.

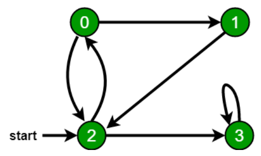
There are two types of graph traversal algorithms – Breadth First Search (BFS) and Depth First Search (DFS). Here, we will learn more in depth about BFS and how it can be implemented.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.



The diagram on the right shows the order in which the nodes are expanded.

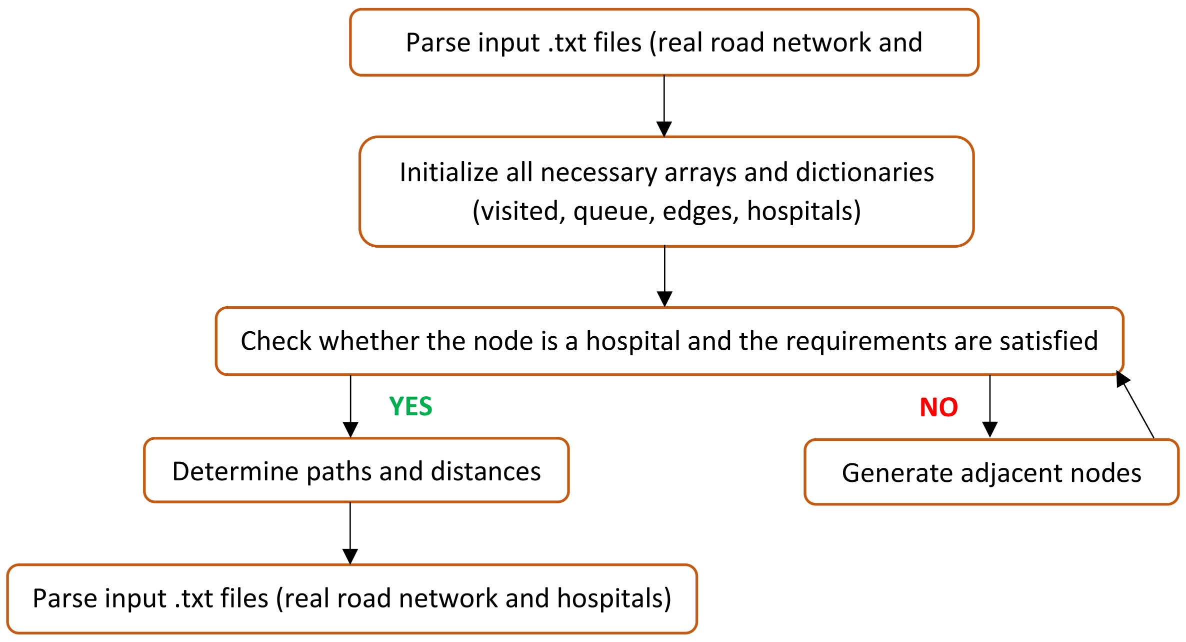
An example of a Breadth First Search can be illustrated using the graph below.



In the graph on the left, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

A Breadth First Traversal of this graph is 2, 0, 3, 1.

Algorithm Flowchart


Parse input .txt files (real road network and
Initialize all necessary arrays and dictionaries (visited, queue, edges, hospitals)
Check whether the node is a hospital and the requirements are satisfied
YES
NO
Determine paths and distances
Generate adjacent nodes
Parse input .txt files (real road network and hospitals)

Algorithm Components and Time Complexity

a. *Pre-processing Stage*

1. **edges_dict**

   Creates an edge dictionary where the keys consist of all the nodes and the values of a key 'i' is an array of nodes that are adjacent to node 'i'. This way, each node does not have to iterate through the edge list to find its adjacent nodes. We are using the *adjacency list* concept for this.

   e.g. If node 'a' is directly connected to node 'b', 'c', and 'd'. edges_dict['a'] = ['b', 'c', 'd'].

The time complexity of creating the edge dictionary would be O(e), whereby e is the number of edges. For every iteration in the for loop, two primitive operations are carried out. As a result, a total of 1+2e primitive operations are carried out in our code to create the edge dictionary.

$$Time\ complexity = 1 + 2e = O(e)$$

2. **hospital_flag (used in part(b), (c), and (d))**
   Creates an array whose length is equal to the number of nodes. It consists of Boolean values on whether a node is a hospital. This will reduce the time complexity because now, it does not depend on the number of hospitals.

```python
# Mark each node as a hospital (TRUE) or not hospital (FALSE)
hospital_flag = [False] * nodes_count
for flag in hospital_list:
    hospital_flag[flag] = True
# hospital_flag
```

   e.g. If node 'm' is a hospital, hospital_flag[m] = TRUE. If node 'n' is not a hospital, hospital_flag[n] = FALSE.

   The algorithm will traverse through the array hospital_list and mark the corresponding flag in the new array as TRUE or FALSE. The total primitive operations are 1+h, whereby h is the number of hospitals, and the time complexity would be O(h).

$$Time\ complexity = 1 + h = O(h)$$

b. *Searching Stage*
   There are several sub-algorithms that we use on the searching implementation.
   1. **Visited array**
      This array contains all nodes we have visited/analysed. Every time we encounter a node, we would check whether we have visited the node. If we have not, we would append the node to the array. If the node is in the array (indicates that we have visited the node), we do not need to revisit the node and proceed with the next node. This way, every node would not be visited more than once.
   2. **Queue array**
      This array contains all upcoming nodes to be analysed. Every time we visit a node, we append all its adjacent nodes to this array. Later, we would pop the first node in the array to become the following node. This way, we implement a First-In-First-Out method, which is the characteristic of the Breadth First Search method.
   3. **Hospital check**

```python
if hospital_flag[node]:
```

      Initially, we mark every node as either a hospital or non-hospital using a Boolean value (TRUE/FALSE). We store all the values in an array called 'hospital_flag' (mentioned in the pre-processing stage). We would use this array to check if a particular node is a hospital. We would either append the node to the result array (part c, d) or return the node as the result (part b). This is much more efficient than the algorithm we implemented on part (a), where we would check if a node is included in hospital_list, which might increase the time complexity by n times, where n is the number of hospitals exist.

```python
if node in hospital_list:
```

   4. **Number of found hospital check (only for part(c) and part(d))**
      Every time we find a hospital, we would then append the hospital node to the result array and check whether we have found enough hospitals. Once the number of results is equal to the target we have, the searching process will stop. We now proceed to the post-processing part of writing all the outputs to the .txt file.

```python
if len(nearest_hospital) == k:
    return 0
```

   5. **Time complexity analysis for each of the sub-problem (on each node)**
      a. *Algorithm in part (a)*

Best case analysis: This happens when the node itself is a hospital. The node is the first element in the hospital_list array. The time complexity is O(1) if the conditions for best case are satisfied since the best case is independent of any critical variables.

Worst case analysis: The time complexity is O(hn+m), whereby m represents the number of edges. Each node will be queued and dequeued once, and each edge will be processed to the visit_dict. For every node, the algorithm will traverse through the hospital_list array to check if the node is the hospital. For worst case analysis, the nearest hospital is the last element in the hospital_list array. This results in the time complexity of O(hn+m).

b. *Algorithm in part (b)*

The hospital_flag array is created during the pre-processing stage in part(b). This allows the time complexity of the algorithm to be independent of the number of hospitals. This change will improve the worst-case time complexity to O(n+m). Meanwhile, the best-case time complexity remains at O(1).

c. *Algorithms in part (c) and part (d)*

The difference in the algorithms in part (c) and part(d) is the introduction of the variable k, which represents the number of the nearest hospitals to be searched. In part(c), k would be fixed to 2. The time complexity for the algorithm in part (c) does not differ much from that in part (b) because the increase of 1 for k is not significant. However, for the algorithm in part(d), since the user can input the value of k, the worst-case time complexity would become O(k(h+m)) as the value of k can increase to a large number and cause the execution time to be longer.

c. *Post-processing Stage*

1. **visit_dict**

It is a dictionary which store the parent node of each visited child node. To the restriction that each node can only be visited once, the parent node of the node 'X' is the first node that branches to the node 'X'. Also, the initial node will never have a parent node as it is already the top-most node. This dictionary is used to track the path from the initial node to the nearest-found hospital.

Since this dictionary stores the parent node of each visited child node, the best-case time complexity would be O(1). This happens when the first node is the parent node and the second node is the child node. Only 1 primitive operation is carried out.

The worst-case time complexity would be O(n-h), whereby it is assumed that the parent node branches from the first node to the last node in the graph without meeting any hospitals.

2. **count**

It is a counter variable which tracks the distance between the initial node to the nearest hospital. While crawling up the parent nodes using the visit_dict dictionary, the count variable would increment by one for every parent node found. When we reach the initial node, which has no parent nodes, we stop and output the distance indicated by the count variable. Then, we reset the count variable to zero.

d. *General Flow*

1. We start off the algorithm by parsing the edges input .txt file (on real road networks) and the hospital input .txt file (which we randomly generated). Specifically, we are using "regular expression" syntax to slice the .txt files into the format we need. Here, we now have the edges_dict and hospital_flag {**a(1) and a(2)**}.
2. We initialize all the necessary variables, dictionaries, and arrays to be used in the searching stage {**b(1), c(1), and c(2)**}.

```
visited = [] # List to keep track of visited nodes.
count = 0
visit_dict = {edge_list: [] for edge_list in range(nodes_count)} # store the parent node of every child node
nearest = bfs(visited, edges_dict, unit)
```

3. Moving to the bfs function, we append the initial node to the visited and queue arrays. This indicates that the node is next to be visited and marked as visited.

```
visited.append(node)
queue.append(node)
```

4. We first check if the initial node is a hospital itself by calling the hospital check {**b(3)**}.

5. From now on, the queue array {**b(2)**} will not be empty until all the nodes have been traversed. We will be analysing the first node of the array which indicates the node with highest priority.

```
while queue:
    s = queue.pop(0)
    # print (s, end = " ")
```

6. For each node we have in step 5, we check if its adjacent nodes have been visited before by calling the visited array {**b(1)**}. We would only process the node if it has not been visited.
7. If the node has not been visited before, we would append the parent node and the child node to the visit_dict dictionary {**c(1)**}, then we proceed to check if the node is a hospital. If it is,

```
if neighbour not in visited:
    visit_dict[neighbour].append(s)
```

   a. For part (a) and (b), we would immediately stop the searching process and return the hospital node.

```
if hospital_flag[neighbour]:
    # print("Hospital is found!")
    return neighbour
```

```
if hospital_flag[neighbour]:
    nearest_hospital.append(neighbour)
    if len(nearest_hospital) == k:
```

   b. For part (c) and (d), we would add the hospital node to the result array (nearest_hospital) and check whether we have reached the required number of hospitals (2 for part (c) and user input for part (d)).
8. If the node turns out to be not a hospital or the minimum number of hospitals is not reached, we append the node to the visited and queue array {**b(1) and b(2)**} for its child nodes to be analysed next.
9. Then, we would loop back to step 5.
10. After we have found all the hospital nodes or all the nodes have been visited but there are not enough hospitals, we move on to the post-processing stage, which is to write back the output to a .txt file. We now check the result array (nearest_hospital),
    a. If we have found a hospital, the algorithm would return 1 to flag and it immediately writes "It is a hospital."
    b. If we do not find any hospitals, it will write "It is not connected to any hospitals."
    c. If the number of hospitals does not satisfy the requirements, it will print the path on step 11 and write "It is only connected to k hospital."
    d. If we have found a sufficient number of hospitals, it will print the output on step 11.
11. Traversing through the visit_dict {**c(1)**}, we can find the path from the initial node to the hospital. Note that this is only applicable for part (b) as part (c) and (d) only require the distance. We also increment the count variable {**c(2)**} to determine the distance from our initial node to the hospital.

```
while visit_dict[point] != []:
    point = visit_dict[point][0]
    count += 1
```

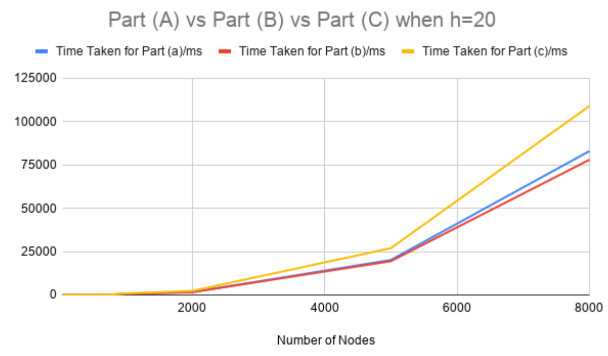
12. Finally, an output .txt file would be populated with the outputs. We are done. *Note that the execution time for the real road networks might take days because the I/O functions itself (to write to the .txt file) are inevitably time-consuming but cannot be omitted from our process.*

<u>Empirical Analysis</u>

1. *Effects of n (number of nodes)*

Number of hospitals kept constant at h=20:

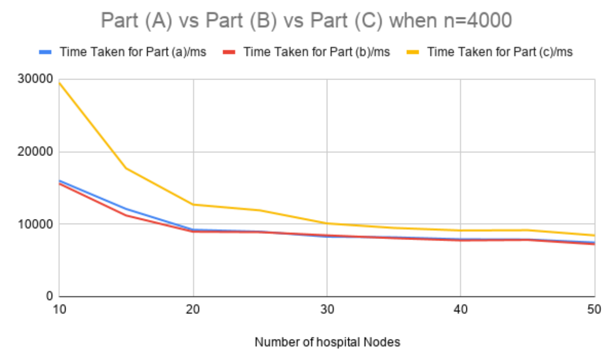
It can be seen in the graph that the time complexity to carry out all three parts is relatively the same when the total number of nodes is the smallest. However, the time taken for part(c) rapidly increases, rising much faster than that of part (a) and part (b). This supports the theory as the time complexity for part (c) is O(kh+km) in the worst case scenario, while the time for part(c) is rising faster, where we have to find the nearest 2 hospitals, than for part (a) and part (b). Comparatively, as n increases, time taken for part(a) rises faster than the time taken for part(b) as n increases. This supports the theory as the worst



case time complexity for part (a) is O(hn+m) while the worst case time complexity for part (b) is O(n+m). Hence, regardless of h, time taken for part (a) will always be higher than time taken for part (b)

2. *Effects of h (number of hospitals)*

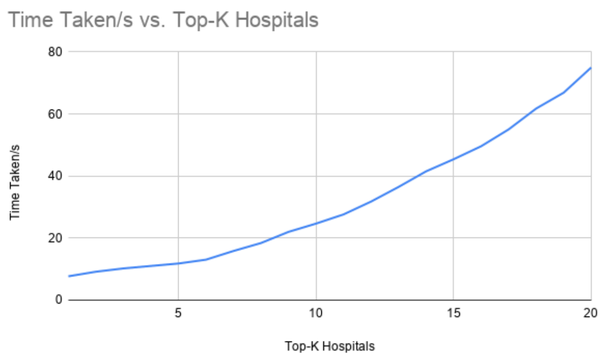
Number of nodes kept constant at n=4000:



It can be seen in the graph that the time taken for part (b) decreases at a faster rate than that for part (a). This supports the theory as the worst-case time complexity for part (a) is O(hn+m) while the worst case time complexity for part (b) is O(n+m). Since the worst-case time complexity is independent of h, only the number of nodes and number of edges between a particular node and a particular hospital will affect part (b)'s time taken. Time taken for part (c) decreases at much slower rate than part (a) and part (b), which also supports theory, as the worst case time complexity for part (c) is O(k(h+m)), which means that the time taken to carry out part (c) is affected by k, which in this part (c) is 2. As the number of hospital nodes increase, the time taken for all three parts is relatively the same.

3. *Effects of k (number of hospitals to be searched)*

Number of nodes kept constant at n=4000 and number of hospitals kept constant at h=50



It can be seen in the graph that the time taken to carry out part (d) increases exponentially as the number of k increases. This supports theory as well as the worst case time complexity for part (d) is O(kh+km). As k rises, the time taken to carry out part(d) will significantly increase each time.

Conclusion

In conclusion, BFS is used to find the single source shortest path in an unweighted graph, because we reach a vertex with minimum number of edges from a source vertex. Thus, BFS is more suitable for searching vertices which are closer to the given source (i.e, finding the shortest paths between nodes in this case).

DFS, on the other hand, might traverse through more edges to reach a destination vertex from a source. Therefore, DFS is more suitable when there are solutions away from source (for example, for games or puzzles).

References

(2020). Breadth-first search. Retrieved from https://en.wikipedia.org/wiki/Breadth-first_search

(2020). How-to-implement-a-breadth-first-search-in-python. Retrieved from https://www.educative.io/edpresso/how-to-implement-a-breadth-first-search-in-python

Anvesh. (2020). Breadth First Search or BFS for a Graph. Retrieved from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://web.stanford.edu/class/msande235/erdos-renyi.pdf

Statement of Contribution

Name	Contribution
Ang Shu Hui	- Report time complexity - Presentation time complexity
Bachhas Nikita	- Report empirical analysis, visualization - Presentation empirical analysis
Kundu Koushani	- Report introduction, conclusion - Presentation introduction, conclusion
Leonardo Irvin Pratama	- Algorithm code on Jupyter Notebook - Report algorithm flowchart, components - Presentation algorithm flow