Given an undirected unweighted graph G with V vertices and E edges. There are H special vertices(hospitals) in the graph. Our tasks is to find the nearest, 2-nd nearest, and generalize it for k-th nearest hospitals from every vertices. In this report, we will be utilising the following three algorithms to solve the tasks:

1. Breadth-First-Search(BFS) Algorithm
2. Multi-Source BFS Algorithm
3. Self-Modified Multi-Source BFS Algorithm

1 Breath-First Search(BFS) Algorithm

Breadth-First Search (BFS) is a graph traversal algorithm. It will start from a source and proceed to explore all of its neighbouring vertices in the current depth, before moving on to the next depth. This ensures that the searching path to reach every vertex is the shortest from the source. To implement BFS, firstly push the source vertex to the queue and initialize its distance from source as 0 and distance for every other vertices as infinity in distance array `d[]`. After that, use a `while` condition to check if the queue is empty. Else, remove the front vertex of the queue, add all unvisited neighbours of that vertex into the queue and update the value of neighbour in `d[]`. Every vertex will be pushed and popped once from the queue, this will take up $\mathcal{O}(V)$ operations. For every vertex in the queue, we need to traverse through all of its connecting edges. That means we will check for $degree(v)$ edges for every vertex v. By Handshaking Lemma,

$$\sum_{v \in G} degree(v) = 2E$$

Hence, this will take $\mathcal{O}(E)$ operations, which will sum up to the time complexity $\mathcal{O}(V + E)$.
For part (a), we can easily use BFS to start searching from every hospital and for every vertex, we find which hospital is the nearest. This will run H times of BFS, which have complexity $\mathcal{O}(H(V + E))$.

1.1 Correctness of BFS Shortest Path

We need to proof BFS can find the shortest distance for every vertex from source in unweighted graph. More precisely, for every vertex v, the value $d[v]$ in the distance array will equals to the distance of source s to v(denoted as $dist(s, v)$) after running BFS.

Proof. Suppose there exist vertices v such that $dist(s, v) \neq d[v]$. Let v be such vertex with smallest $dist(s, v)$. Firstly, $v \neq s$ due to the initialization, hence $dist(s, v) \geq 1$. Clearly, we will update $d[v]$ only when we reach v by some path and the value will be the length of that path, which is not smaller than $dist(u, v)$. Also, $dist(s, v) \neq d[v]$ by assumption. So $d[v] \geq dist(s, v) + 1$. There must exist a neighbour u of v such that $dist(s, u) + 1 = dist(s, v)$, hence $dist(s, u) < dist(s, v)$. By choice of v we have $dist(s, u) = d[u]$. Since when we processing neighbours v of vertex u, we will update $d[v] = \min(d[v], d[u] + 1)$, we get $d[v] \leq d[u] + 1 = dist(s, u) + 1 = dist(s, v)$. So $dist(s, v) + 1 \leq dist(s, v)$, which is a contradiction. *Q.E.D.*

2 Multi-Source BFS Algorithm

For part (b), BFS is modified to start searching from multiple sources. Firstly, push all of the hospital vertices into the queue. Then run the exploring process in the single source BFS. By maintaining a `visited` value of every vertex, we could visit every vertex exactly once. Therefore, every vertex is inserted into the queue exactly once, hence the complexity is the same as normal BFS, which is $\mathcal{O}(V + E)$. Besides, we need to maintain a `pre[]` array. For each vertex v, store `pre[v]` as the father vertex in the BFS tree. To print the path from every vertex to the nearest hospital, repeatedly print current vertex and move to the `pre` vertex, until reach one of the hospital vertex(can store `pre[hospital]` as -1). Printing all the paths will need $\mathcal{O}(V^2)$ in the worst case, as every path will have length at most $V - 1$.

3 Self-Modified Multi-Source BFS Algorithm

For part (c) and (d), we will continue to modify the algorithm used in part (b). In part (b), the vertex will be visited exactly once. This will limit us to only being able to find the nearest hospital for every vertex. Hence, the modification of the `visited` value is needed. For every vertex, we need to maintain an array for storing this information. The array

will record that the vertex is already visited by which hospital. Let's denote this array of vectors as `visited[V][]`, where `visited[i][j]` is a pair that store the j-th hospital that reaches vertex i and the distance between them.

For implementation, we need to modify the queue to store tuple information, which is $(v, \text{hospital that reach } v)$ instead of just v. Firstly for every hospital h, push $(h,\ h)$ into the queue and insert h into the `visited[h]`, which means that the hospital h is already visited by itself. Then we started the exploring process. Firstly, taking out the first element (v, f) of the queue, check every neighbour u of v. To visit u in this step, it needs to satisfy two conditions:

1. u haven't visited by hospital f.
2. u is visited by less than k hospital.

This can be done easily by checking the vector `visited[u][]`. If these conditions met, visit u from v by pushing (u, f) into the queue and insert f into `visited[u][]`, and take element from queue to process again.

3.1 Analysis

Every vertex v in the graph will be visited at most k times due to the condition 2 when checking `visited[v][]`. Hence, there will be at most kV elements pushed or popped from the queue, which takes $\mathrm{O}(kV)$ operations. For every element (v, f) popped out from the queue, every connecting edge of v. By handshaking lemma again, the total operations of this part is at most k times of $2E$. For every neighbour of v, we will need to check whether it is visited by f, this will use $\mathcal{O}(k)$ operations for the array searching, hence leading to complexity $\mathcal{O}(k^2E)$. This can be furthermore optimized by maintaining the `visited[][]` array by using an array of self-balancing binary search trees(BST) to shrink the $\mathcal{O}(k)$ factor to $\mathcal{O}(\log k)$. If using hash sets, this will become $\mathcal{O}(1)$. Hence, the complexity of this part is $\mathcal{O}(kE)$. Overall, the total complexity will be $\mathcal{O}(kV + kE) = \mathcal{O}(k(V + E))$.

In terms of space, there are three main information that needs to be stored, which is structure of graph, queue for BFS and the `visited[]` array. For structure of graph, we can store it in $\mathcal{O}(V+E)$ space by adjacency list. Since every vertex will be pushed to the queue at most k times, the space of the queue will be at most $\mathcal{O}(kV)$. For `visited[]` array, if using arrays/vectors to store, it will need $\mathcal{O}(kV)$ spaces(due to condition 2 when checking `visited[]`). If using BST, every BST will store at most k value, in total the space will be $\mathcal{O}(kV)$ too. If using hash set, the space will be $\mathcal{O}(HV)$ where H is the size of the hash tables.

To print the path, $\mathcal{O}(kV)$ space will be used for storing the transition information. Every path will have length at most $\mathcal{O}(V)$, and there are k path for each vertex, hence the time complexity for printing path will be $\mathcal{O}(kV^2)$.

Data structure in `visited[]`	Total time complexity	Total space complexity
Arrays/Vectors	$\mathcal{O}(kV + k^2E)$	$\mathcal{O}(kV + E)$
Self-balancing BST	$\mathcal{O}(kV + Ek \log k)$	$\mathcal{O}(kV + E)$
Hast Set	$\mathcal{O}(k(V + E))$	$\mathcal{O}(HV + E)$

Table 1: Time/space complexity for using different data structure

3.2 Correctness

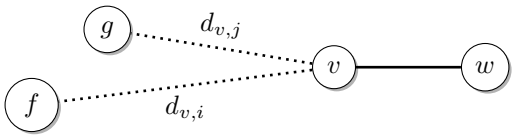
If we run the algorithm above, and remove the condition 2 in the searching process, we will get the sequence of elements be pushed into the queue, denote it as $S = \{(v, f)_i\}$.

Consider doing H times BFS from every hospital f, let the sequence of all vertices be pushed into the stack be $\{v_i\}_h$. Since the visit order of vertices that have the same distance from the source is not important, we can reorder the order of same distance and merge $\{v_i\}_h$ for every f to obtain the sequence S. Hence, these can be viewed as doing H times BFS from hospitals with one queue.

Next, we add back the condition 2 in the searching process. This is equivalent to remove the pairs (v, f) from S if v appeared not less k times in S before. Now, we need to prove that those pairs are unnecessary, more precisely, removing this pair won't affect us to find the k nearest hospitals of any vertex.

Proof. Denote $p_{v,i} = (v, f)$ as the i-th pairs correspond to v in sequence S, and denote the distance correspond to this pair(distance from v to f) is $d_{v,i}$. We can know that $d_{v,i} \le d_{v,j}$ if $i \le j$ based on the properties of BFS.

Suppose after we remove $p_{v,i}$ for some v that $i > k$, some shortest path to its k nearest hospital of some vertices is affected. That means there must exist a vertex w, using $p_{v,i} = (v, f)$ in one of the paths to reach its k nearest hospitals. More precisely, f is one of the k nearest hospitals of w, with distance $d_{v,i} + 1$. That means there must exist a hospital g such that g is one of the k nearest hospitals for v but not w, i.e. for some $j \le k$, $p_{v,j} = (v, g)$.



Since g is k nearest hospitals to v but f is not. Hence, $d_{v,j} \leq d_{v,i}$ By using this (v, g) to reach w instead of (v, f), we can reach w from g with a distance d not greater than $d_{v,j} + 1$(might smaller because there might have some path from g to w doesn't pass through v), which will also not greater than $d_{v,i} + 1$ since $d_{v,j} \leq d_{v,i}$, i.e. $d \leq d_{v,i} + 1$. Then, we have two case:

1. If the distance $d < d_{v,i} + 1$, that contradicts the fact that g is not one of the k nearest hospitals for w but f is.
2. If the distance $d = d_{v,i} + 1$, that means $d_{v,j} = d_{v,i}$, we can changed the k nearest hospitals f for w to g without affecting the distance. Since d_v is in not decreasing order(due to the properties of BFS) and $d_{v,j} = d_{v,i}$, the value of $d_{v,x}$ is same for every $j \leq x \leq i$, hence $d_{v,i} = d_{v,k}$. So, changing from f to g won't affect the **distances** of w to its k nearest hospitals (that means we can always find k-nearest hospital for w although the hospital might be different due to the same distances).

Hence, this proof that removal of the pair won't affect us to find the k nearest hospitals of any vertex, and the algorithms is correct. *Q.E.D.*

4 Empirical Time Analysis

We will now comparing the performance of the algorithms implemented with three different data structure along with a naive algorithm. The naive algorithm will start doing BFS from every hospital to find the distance to every vertex, and will sort the H hospitals based on the distance for every vertex. The total complexity of this algorithm is $\mathcal{O}(VH \log H + HE)$, which mainly comprised by the BFS part $\mathcal{O}(H(V + E))$ and the sorting part $\mathcal{O}(VH \log H)$. The California road network dataset[*link*] is the testing dataset for this analysis. We set $H = 50$ and varying the value of k. The results is shown below.

Algorithm \ k	1	2	4	8	16	32	Time Complexity
Naive search from each hospital	108.25	113.84	109.28	107.56	114.94	110.98	$\mathcal{O}(VH \log H + HE)$
Array-based `visited[]`	1.25	2.98	6.53	18.64	42.52	107.17	$\mathcal{O}(kV + k^2E)$
BST-based `visited[]`	2.78	6.42	19.26	47.65	98.99	196.77	$\mathcal{O}(kV + Ek \log k)$
Hashset-based `visited[]`	2.45	5.55	14.89	36.72	71.73	173.85	$\mathcal{O}(k(V + E))$

Table 2: Empirical Performance in Time when $H = 50$

In the table we found that our algorithms work well when k is significantly smaller than H, performance becomes similar to the naive algorithm when k is near to H. We can also observe that array-based having the best performance in this situation even the theoretical complexity seems worst. This is because of the constant factor of the time complexity is omitted in the theoretical analysis. In this case, k is quite small ($<= 32$), the complex data structure will be unnecessary in this case due to the constant factor from the complex calculation(tree traversing/hashing). These complex data structures will have advantages when k is large.

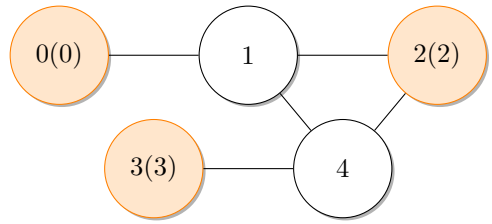
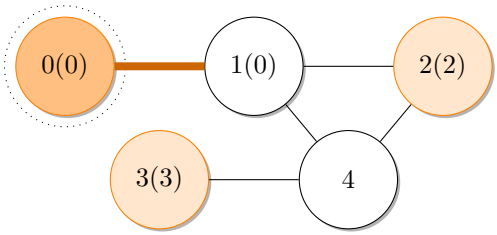
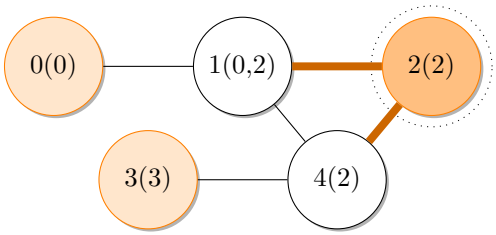
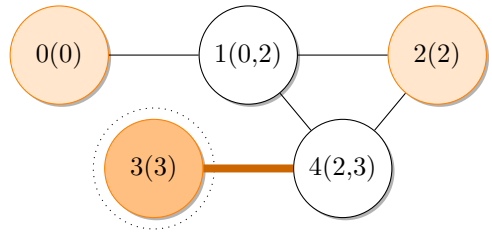
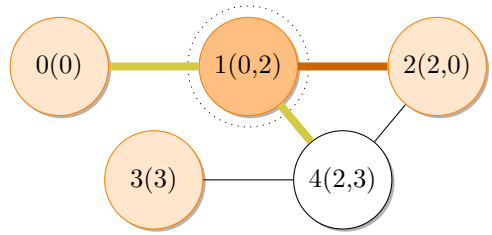
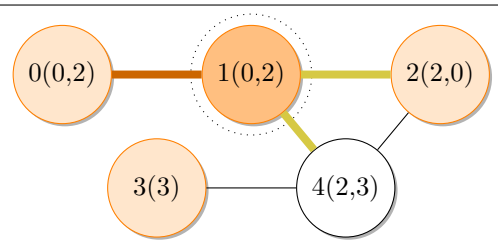
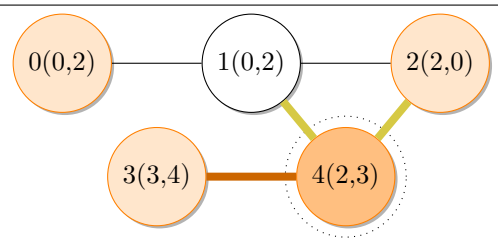
5 Conclusion

Different algorithms will have different use cases. We will need to select the correct algorithms based on different applications. For example, if k is small enough, the array-based version will be the best choice. Although theoretical time complexity gives us some sense for running time, it might not be suitable when the parameter is small. Some simple naive algorithms might perform well in this case due to the simple calculation steps.

6 References

`https://www.geeksforgeeks.org/multi-source-shortest-path-in-unweighted-graph`
`https://www.cs.mcgill.ca/~pnguyen/251F09/BFScorrect.pdf`

Table 3: Demo of searching first $k = 2$ nearest hospitals for each vertex. Vertex $0, 2, 3$ are hospitals. Distance from each hospital is omitted in this demo. Demo stop when all vertex found their $(k = 2)$-th nearest hospitals.

Graph	Processing	BFS Queue $(\text{vertex}, \text{hospital})$
	Initialize	$(0,0), (2,2), (3,3)$
	$(0,0)$	$(2,2), (3,3), (1,0)$
	$(2,2)$	$(3,3), (1,0), (1,2), (4,2)$
	$(3,3)$	$(1,0), (1,2), (4,2), (4,3)$
	$(1,0)$	$(1,2), (4,2), (4,3), (2,0)$
	$(1,2)$	$(4,2), (4,3), (2,0), (0,2)$
	$(4,2)$	$(4,3), (2,0), (0,2), (3,2)$