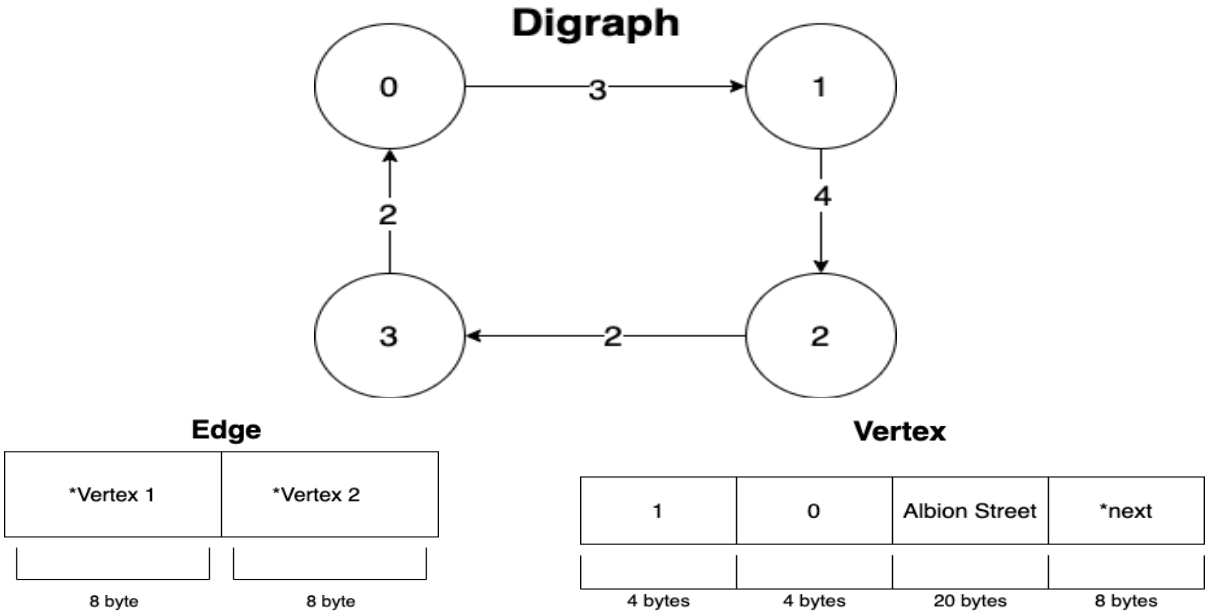
ASSIGNMENT 4

Digraph

The nodes and edges form a digraph or a directed graph in which he edges connect vertices in a specific direction. Top-view and in-memory representation of a part of a graph are shown below.



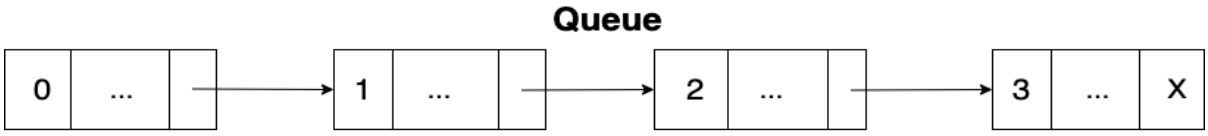
Data Structures

In this assignment, I have used the following data structure to form a perfect bus network.

- Queue: in Kosaraju and BFS algorithms
- Heaps: in Dijkstra's algorithm
- Adjacency lists: in-memory representation of the digraph

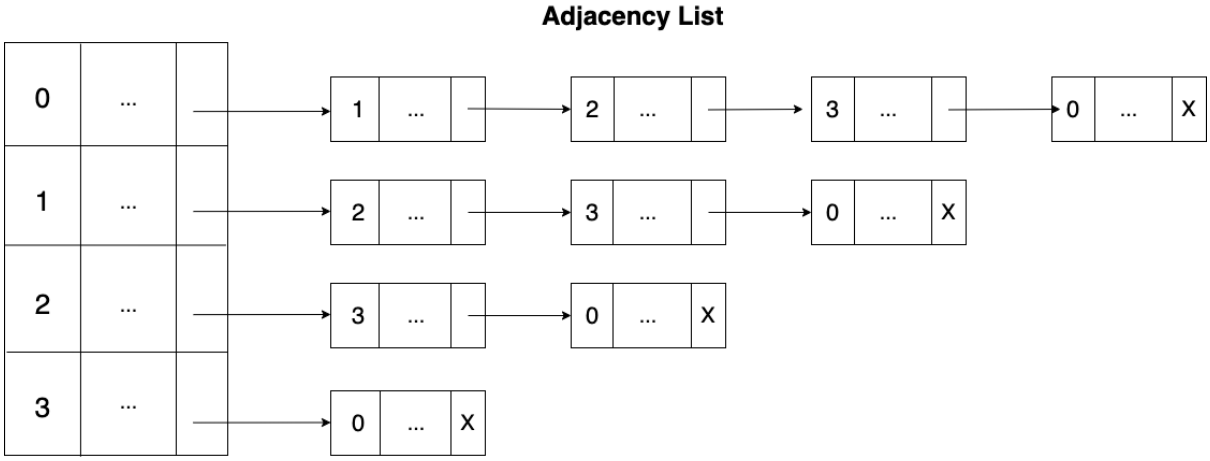
Queue

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.
Memory representation of each node and their queue formation is shown below.



Adjacency List

In this assignment, the digraph is represented as an adjacency list. As the space requirement of the adjacency list is V+E which is less than V2 of that of the adjacency matrix.
Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex iii, store an array of the vertices adjacent to it. We typically have an array of |V| adjacency lists, one adjacency list per vertex. Vertex numbers in an adjacency list are not required to appear in any particular order, though it is often convenient to list them in increasing order.
We can get to each vertex's adjacency list in constant time because we just have to index into an array. To find out whether an edge (i,j) is present in the graph, we go to i's adjacency list in constant time and then look for j in i's adjacency list. Θ(d) is the worst case where d is the degree of vertex i because that's how long i's adjacency list is. The degree of vertex i could be as high as |V|-1 (if i is adjacent to all the other |V|-1 vertices) or as low as 0 (if i is isolated, with no incident edges).
Corresponding adjacency list of the Digraph is shown below.

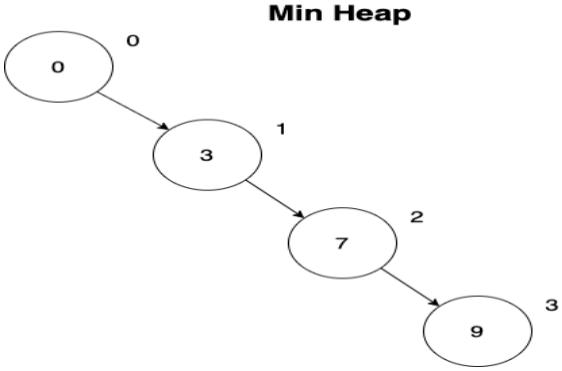


Heap

A Binary Heap is a Binary Tree with the following properties.

- It's a complete tree (All levels are filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
- A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min-Heap.

The min-heap is created with source vertices and the weight of the edges between them as the keys. We first add the source and then traverse through the graph to reach every possible vertex and store the vertex and min distance to reach them as the key. Finally, when the heap is created we start removing the root of the min-heap, which will always be the minimum value in that heap. Repeat the same process until we reach the destination node. The path will always be the shortest in this scenario.
The corresponding min-heap of the digraph is shown below.



Algorithms

In this assignment we are majorly using three algorithms namely

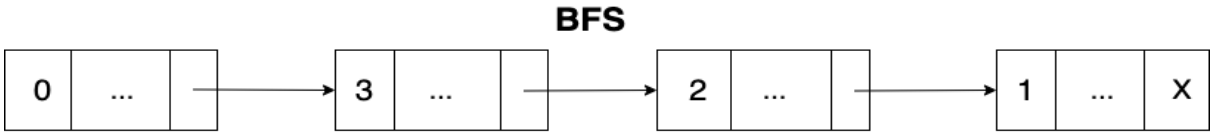
- Breadth-first search (BFS) algorithm: for traversal
- Kosaraju' algorithm: for strongly connected components
- Dijkstra's algorithm: for shortest path

Breadth-First Search (BFS)

Breadth-First Search (BFS) algorithm traverses a graph in a breadth-ward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

1. Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
2. If no adjacent vertex is found, remove the first vertex from the queue.
3. Repeat 1 and 2 until the queue is empty.

Traversing the above digraph through BFS forms the following queue:



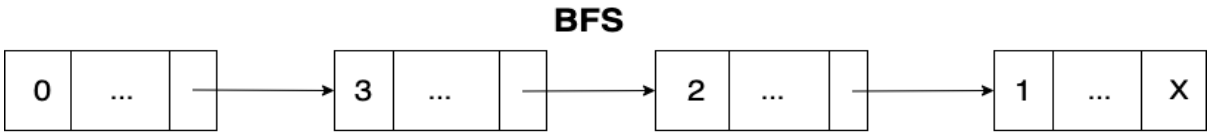
Kosaraju's Algorithm

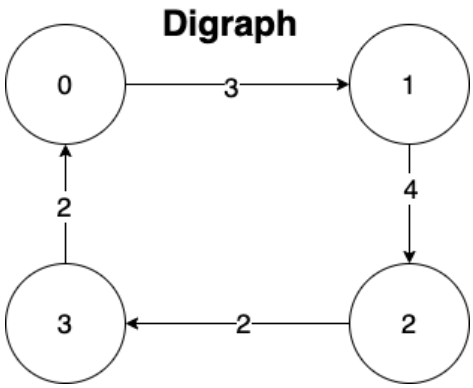
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the given graph.

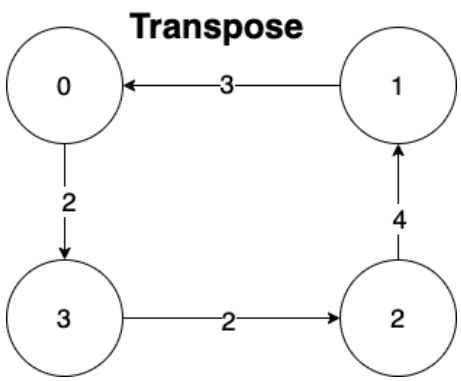
We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm.

Kosaraju's algorithm has three major steps, as follows:

1. Create an empty queue and do BFS traversal of a graph. In BFS traversal, after calling recursive BFS for adjacent vertices of a vertex, add the vertex to the queue.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one remove a vertex from the queue while it is not empty. Let the removed vertex be 'v'. Take v the as source and do BFS. The BFS starting from v prints strongly connected component of v.

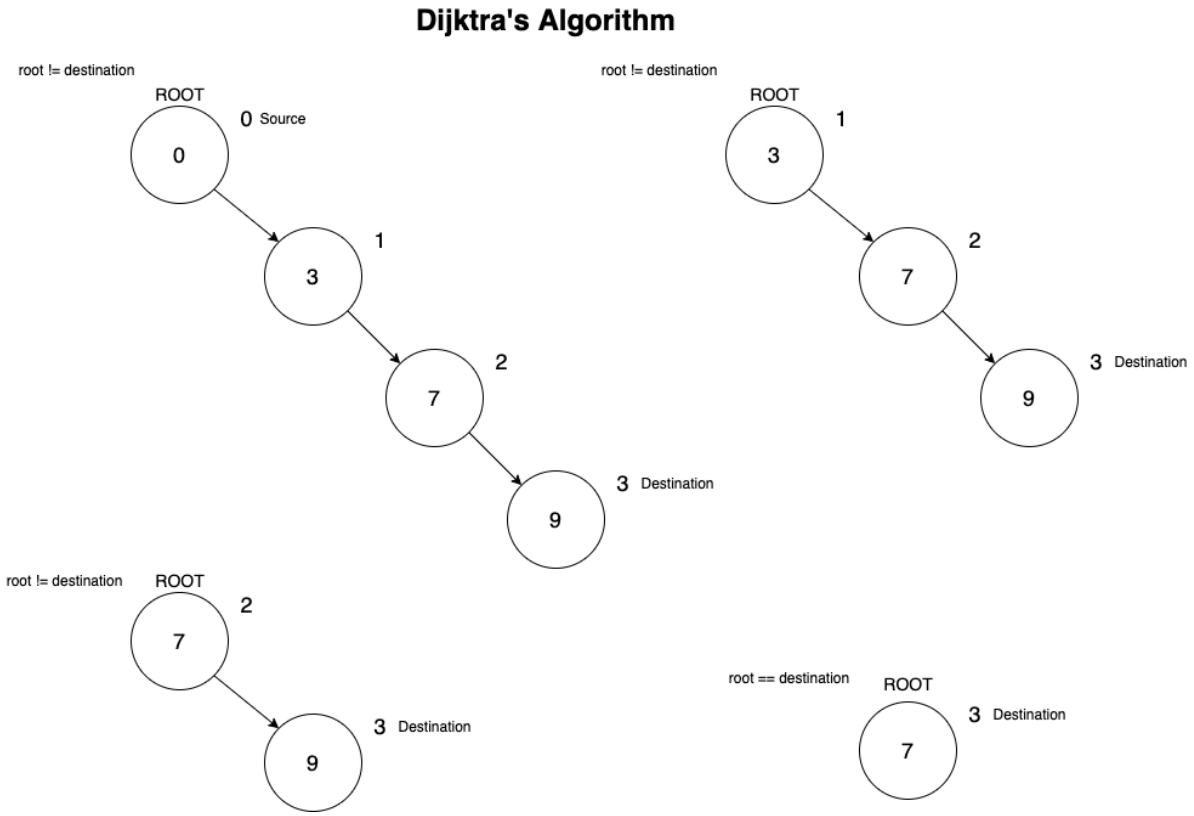






Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from a minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph. Dijkstra used this property in the opposite direction i.e. we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest sub-path to those neighbours. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the result is the best solution for the whole problem.
In this assignment, minimum priority queue i.e. min-heap is used to efficiently receive the vertex with least path distance.
On every iteration, the root node is removed and checked if the root is the destination node. Heap is adjusted again to maintain the min-heap structure, i.e. children are promoted to the root node.
Repeat the process until the root node is the destination.
At last, when the destination is the root node, we can say that the key of the node is the shortest path in the digraph from source to destination.



Time Complexity Analysis

1. *int InsertEdge(Graph g, Edge *e)*
   This function inserts the edge in the given graph. It takes constant time to do so as the graph is represented in the adjacency list fashion.
   ⇨ Hence Time Complexity: O(1)

2. *Graph CreateGraph(Graph g, const char *busStops, const char *Distance, int transpose)*
   This function creates a graph by reading the given text files on the fly. It also can create transpose of the graph if the flag is passed as 1 [one]. Let's assume there are N lines in the test files
   ⇨ O(N) + O(N) = O(N)
   ⇨ Hence Time Complexity: O(1)

3. *void ResetGraph(Graph g)*
   This function resets all the vertices of the graph to default and returns the head node of the graph. Let's assume there are N vertices in the given graph. This function iterates over each node to reset it.
   ⇨ Hence Time Complexity: O(N)

4. *void BFS(VertexNode *node, int print)*
   This function traverses each vertex and edge of the given graph in breadth-first fashion, i.e. first iterates over all the adjacent nodes of the current node. Let's assume there are N vertices and M edges
   ⇨ Hence Time Complexity: O(N+M)

5. int distance(int busCode1, int busCode2)
   This function returns the distance between the two given vertices by reading the text file on the fly. Let's assume there are N lines in the given text file
   ⇨ Hence Time Complexity: O(N).

6. *int StronglyConnectivity(const char *busStops, const char *BusNames, const char *BusRoutes, const char *Distance)*
   This function returns 1 if the given graph is strongly connected or else 0, not strongly connected. The Breadth-first approach is used twice on the graph and its transpose and each vertex are checked iteratively. Let's assume there are N vertices and M edges
   ⇨ O(N) + O(M+N) + O(N) + O(M+N)
   ⇨ Hence Time Complexity: O(N+M)

7. *void maximalStronglyComponents(const char *busStops, const char *BusNames, const char *BusRoutes, const char *Distance)*
   This function returns all the strongly connected components of the graph using Kosaraju's algorithm. Let's assume there are N vertices and M edges
   ⇨ Hence Time Complexity: O(N+M)

8. *void reachableStops(const char *sourceStop, const char *busStops, const char *BusNames, const char *BusRoutes, const char *Distance)*
   This function returns all the reachable stops from the given source stop using BFS. Let's assume there are N vertices and M edges.
   ⇨ Hence Time Complexity: O(M+N)

9. void TravelRoute(const char *sourceStop, const char *destinationStop, const char *busStops, const char *BusNames, const char *BusRoutes, const char *Distance)
   This function returns the shortest travel route from the given source and destination stops using Dijkstra's algorithm and priority queue as the heaps. Let's assume there are N vertices and M edges
   ⇨ O(logN) + O(1) + O(M+N) * O(logN)
   ⇨ Hence Time Complexity: O(logN(M+N))