DSA Project

presented by Saad Thaplawala
and Sharjeel Chandna

Project vision and mission

The code takes a bidirectional weighted graph (telephone network) from the user where the vertices represent switching stations, the edges represent the transmission line, and the weight of edges represent the Bandwidth. By the use of graphs and a modified Dijkstra's algorithm, the code allows us to route the call with the highest bandwidth. It also caters to breadth-first and depth-first traversals of the graph.

01. Dijkstra's algorithm is a single source shortest path algorithm used to calculate the shortest path between two vertices of a graph. Modified Dijkstra algorithm traverses over all paths in O(V^2 + E) and gives highest bandwidth path to make sure there is no bottleneck when transferring data(in this case calls).

02. BFS is a graph traversal approach in which traversal starts at the source node and then layer by layer through the graph, analyzing the nodes directly related to the source node. Then, moves on to the next-level neighbor nodes.

03. DFS is a graph traversal approach in which traversal starts at the root node and then explores as far as possible along each branch before backtracking.

DATA STRUCTURES

Graphs

The graph data structure is essential for representing the network of vertices and edges with bandwidths; Allowing dynamic addition of edges on runtime, implementing traversal algorithms (DFS and BFS) and finding the highest bandwidth path between two vertices.

Linked List

Linked lists are used to represent adjacency lists in the graph, allowing for dynamic addition of edges.

Arrays

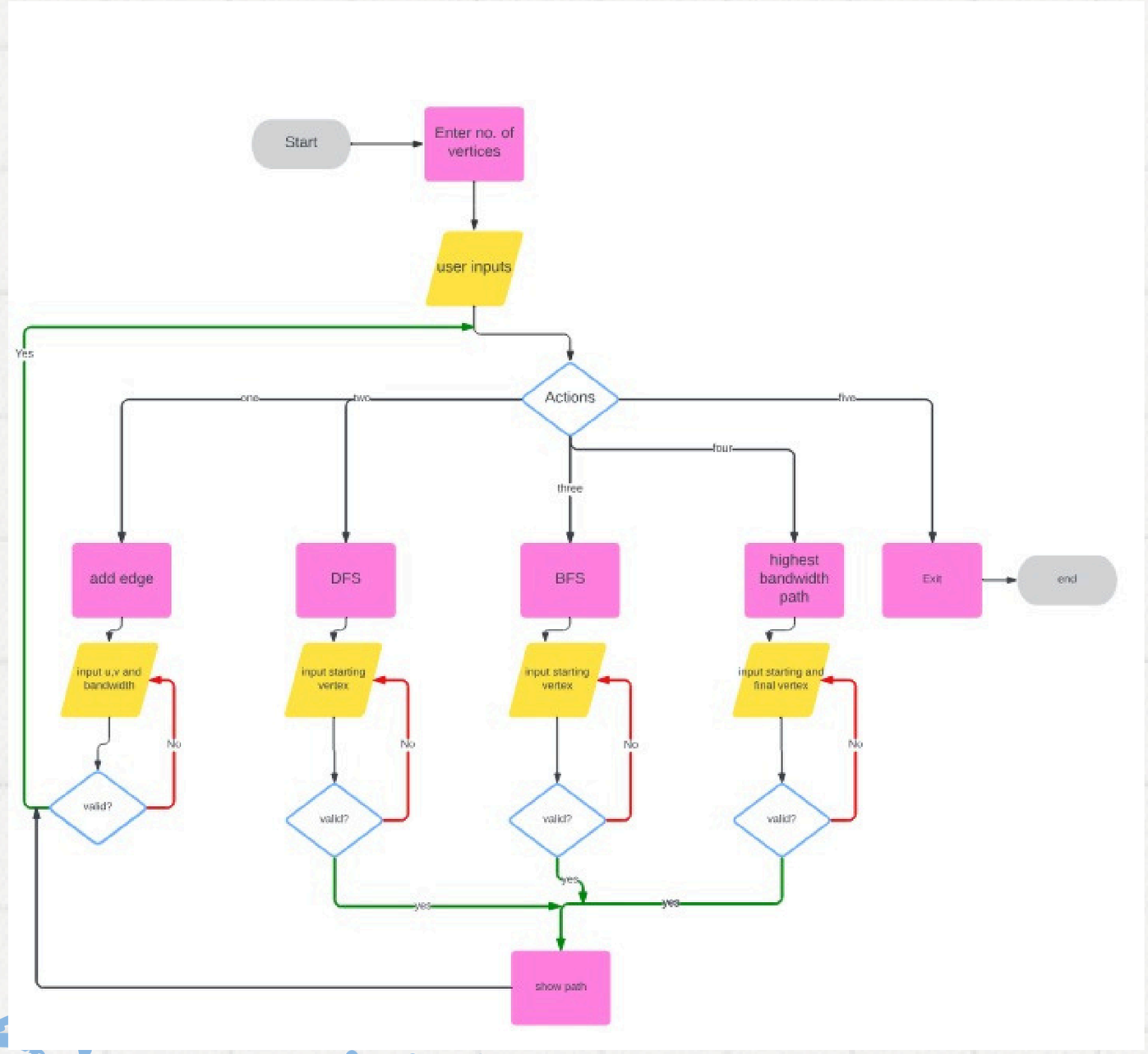
> Array in 'Graph' class: Stores an array of linked lists to represent the adjacency list of each vertex, enabling efficient edge lookups and insertions.
> Boolean Array for visited in 'DFS', 'BFS', and 'highestBandwidthPath' methods: Keeps track of visited vertices to avoid revisiting and forming cycles in the graph traversals.
>Integer Array for 'bandwidth', 'visited', and 'predecessor' in 'highestBandwidthPath' method:
-bandwidth: Stores the maximum possible bandwidth for each vertex from the start vertex.
-predecessor: Stores the predecessor of each vertex in the path for reconstructing the path.
-visited: Keeps track of whether each vertex has been visited during the algorithm.

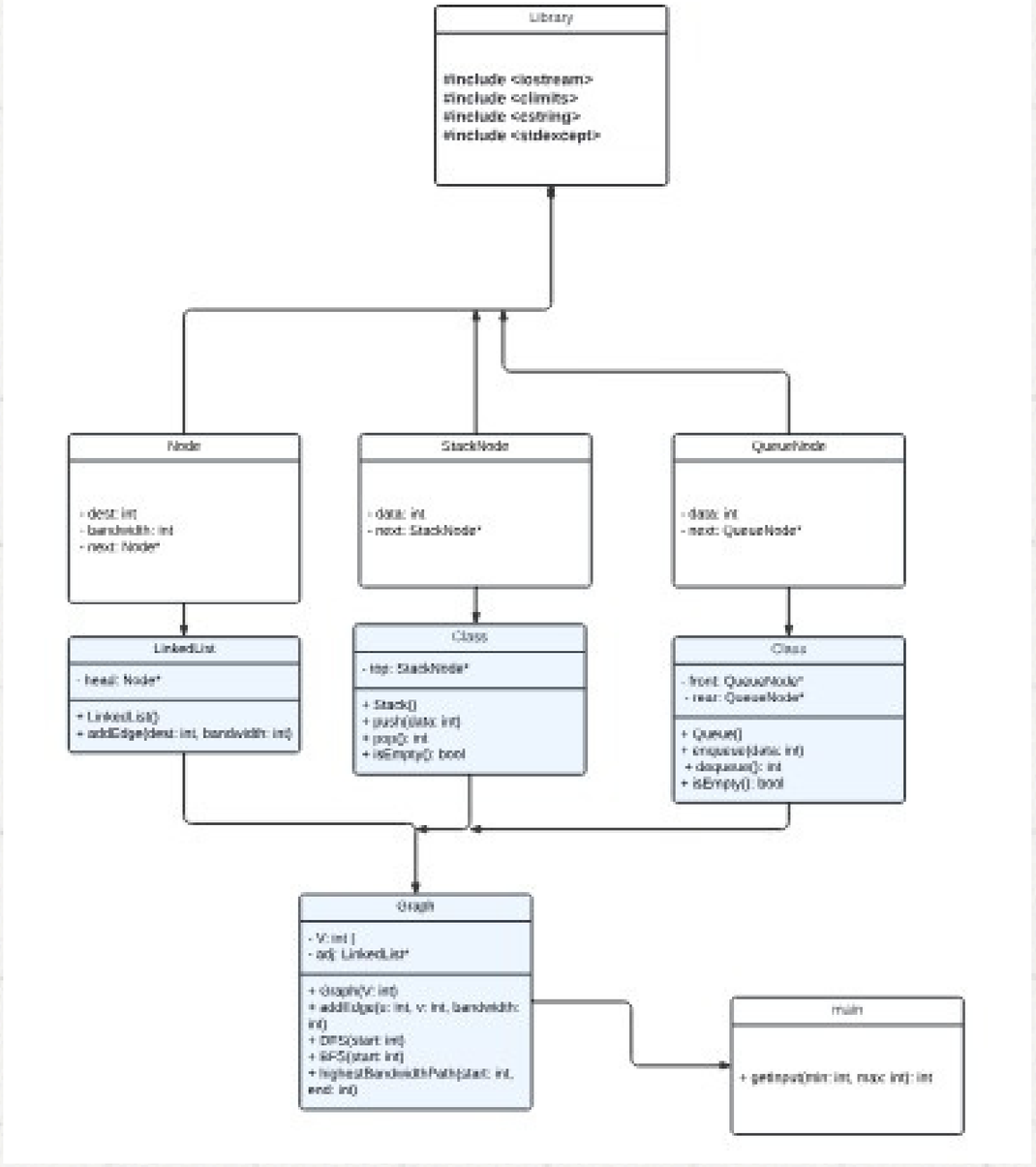
Stacks

DFS traversals make use of stacks to manage the vertices to be visited next. Stacks are suitable for LIFO (Last-In-First-Out) order, which is essential for DFS.

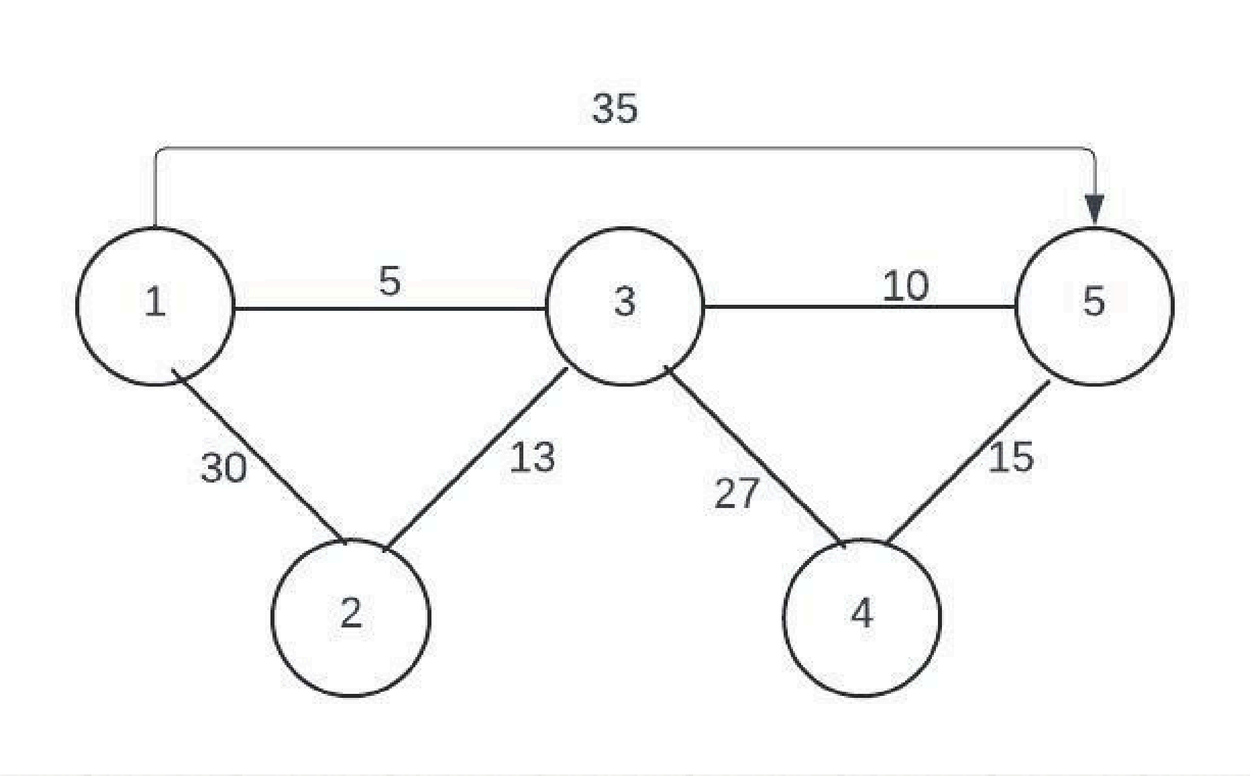
Queues

BFS traversals make use of queues to manage the vertices to be visited next. Queues are suitable for FIFO (First-In-First-Out) order, which is essential for BFS.


Start
Enter no. of vertices
user inputs
Yes
Actions
one
two
three
four
five
add edge
DFS
BFS
highest bandwidth path
Exit
end
input u,v and bandwidth
input starting vertex
input starting vertex
input starting and final vertex
No
No
No
No
valid?
valid?
valid?
valid?
yes
yes
yes
show path


Library
#include <iostream>
#include <climits>
#include <string>
#include <stdexcept>
Node
- dest: int
- bandwidth: int
- next: Node*
StackNode
- data: int
- next: StackNode*
QueueNode
- data: int
- next: QueueNode*
LinkedList
- head: Node*
+ LinkedList()
+ addEdge(dest: int, bandwidth: int)
Class
- top: StackNode*
+ Stack()
+ push(data: int)
+ pop(): int
+ isEmpty(): bool
Class
- front: QueueNode*
- rear: QueueNode*
+ Queue()
+ enqueue(data: int)
+ dequeue(): int
+ isEmpty(): bool
Graph
- V: int
- adj: LinkedList*
+ Graph(V: int)
+ addEdge(s: int, v: int, bandwidth: int)
+ DFS(start: int)
+ BFS(start: int)
+ highestBandwidthPath(start: int, end: int)
main
+ getInput(min: int, max: int): int

Test Case



DFS starting from 1:

1,2,3,4,5

BFS starting from 1:

1,5,3,2,4

Dijkstra:

1->5->4->3

Method	Big oh	Explanation
addEdge(int dest, int bandwidth)	O(1)	Adding a new node at the head of the linked list is a constant time operation.
push(int data)	O(1)	Adding a new node to the top of the stack is a constant time operation.
pop()	O(1)	Removing the top node from the stack is a constant time operation.
isEmpty()	O(1)	Checking if the stack is empty is a constant time operation.

Method	Big oh	Explanation
enqueue(int data)	O(1)	Adding a new node to the rear of the queue is a constant time operation.
dequeue()	O(1)	Removing the front node from the queue is a constant time operation.
DFS(int start)	O(V + E)	In the worst case, DFS will visit all vertices and all edges once.
BFS(int start)	O(V + E)	In the worst case, BFS will visit all vertices and all edges once.

Method	Big oh	Explanation
highestBandwidthPath(int start, int end)	O(V^2 + E)	The algorithm involves iterating over all vertices multiple times (nested loop structure for updating bandwidth) and processing all edges.
getInput(int min, int max)	O(1)	Reading input and validating it takes constant time operations.

sources:

Dijkstra:

-https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/

-https://youtu.be/mmJC615pjH4?si=K_9M1qnone2lrxi3

BFS:

https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm#:~:text=Breadth%2DFirst%20Search%20Algorithm%20or,the%20next%2Dlevel%20neighbor%20nodes.

DFS:

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

GUI(SFML):

https://youtu.be/EXY7MNqKHTc?si=0pd-xAQmQEobN7Hm

Flow Chart (Lucid Chart):

https://lucid.app/lucidchart/invitations/accept/inv_ca90d931-b3ca-4270-8b34-005da5840785

UML (Lucid Chart):

https://lucid.app/lucidchart/0074a1b6-485b-46b2-8de6-4ded8f61016e/edit?viewport_loc=-1008%2C-619%2C4942%2C2151%2CHWEp-vi-RSFO&invitationId=inv_3894b449-6ff1-43d2-8626-4e74068713ee

Test Case (Lucid Chart):

https://lucid.app/lucidchart/c3a9f527-3d32-440e-aff7-b55924e77e82/edit?viewport_loc=-11%2C-11%2C1480%2C660%2C0_0&invitationId=inv_8423b42e-9d43-48c9-9f6a-7f14e29dd1c5