SERVER MANAGEMENT ENGINEERING METHOD

1. **Problem:**

In 2023 there's been a dire need for services that enable companies to quickly know the speeds and routes their servers provide for better use of their immediate resources. Rather than having to manually check every server they would rather set them up in a simulated environment with their respective connection information to quickly learn the fastest route between servers for stability and efficiency. For this problem our company aims to develop a simulated environment that mimics the respective server connections for companies to input their networks to and micromanage their resources. This includes ways to add servers, add connections and a way for the program to find the fastest routes between two chosen servers.

Symptoms and needs:

- The companies need a way to know the how their servers are going to work.
- The companies want to test the stability and efficiency of their servers.
- The companies want to know how their servers are going to work with a great flow of data.
- The companies need a way to manage the tests of the connections between servers without spending a lot of resources.
- The companies want to know the fastest routes between two servers to give a better service for their clients.

Problem identification:

The companies need a way to simulate the connections between servers for the purpose of knowing the fastest routes for send data

2. Required knowledge:

a. Data structures:

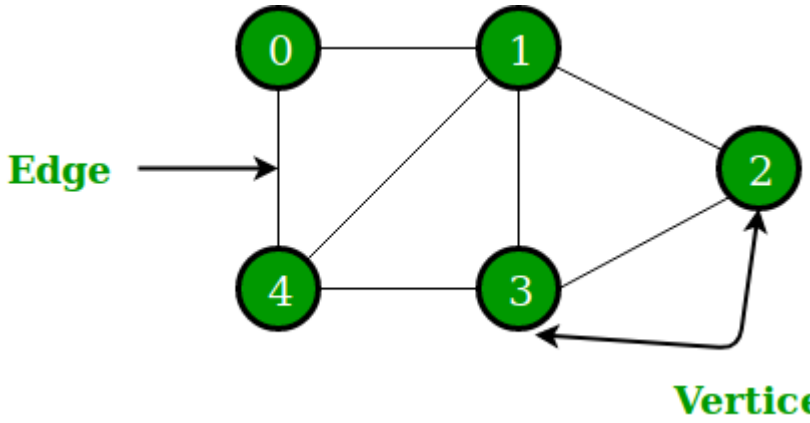
i. **Hash table:** A hash table is a structure capable of storing information depending on a given "key" value, this value is then converted into a position on an array where the information is to be stored, which then can be retrieved with the same key, in average this operation takes $\theta(1)$ time. Helps in saving edge values being vertex destination and weight.

ii. **Priority queue:** A queue that is reorganized depending on a certain attribute (priority) of its members. This kind of structure uses a heap to store and sort the data inside of it. A heap is a way of saving the elements in an array as if they were part of a graph in which on top there are the smallest elements, and on the bottom, there are the biggest ones. Helps with organizing vertex array list or even edges for simpler use in other algorithms.

iii. **Array List:** A list of objects and or attributes that are saved in sequential order. Can be used to store objects indefinitely if you can add. Helps in storing vertices or edges.

iv. **Graphs:** A graph structure is a mathematical representation of a set of objects, called vertices or nodes, and the connections between them, called edges or arcs. It is a way of visually and conceptually representing relationships and interactions between elements. Graphs can be used to model various real-world systems, such as social networks, transportation networks, and computer networks.

v. and continues until all areas are connected. If an edge adds a cycle, it's excluded.



Source: https://www.geeksforgeeks.org/applications-of-graph-data-structure/

vi. **Floyd-Warshall Algorithm:** Graph algorithm that finds the shortest path between two vertices based on the weight that can be used as positive or negative values. Helps in finding the shortest path between two servers instead of mapping everything. The downside is it doesn't save the connections.

vii. **Dijkstra Algorithm:** Algorithm that finds the shortest path between two vertices by distance or weight depending on implementation. It returns a list of vertices that compose said shortest path only problem is you will have to use another method to get the total weight of the path.

viii. **BFS Algorithm:** Method that explores graph level by level. Helps in determining current connections or even showing the whole graph. Can be used to find the shortest path between vertices in an unweighted graph.

ix. **DFS Algorithm:** Method that explores the graph by going as deep as possible and then backtracking. Helps in finding cycles in a graph.

x. **Prim Algorithm:** Helps in finding the shortest path between all vertices without cycles using each vertex as an index.

xi. **Kruskal Algorithm:** Helps in finding the shortest path between all vertices without cycles by going edge by edge from smallest to largest

3. Proposed solutions

Solutions are divided into different types being pathfinding, supportive, infrastructure:

Pathfinding:

1. BFS
2. DFS
3. Dijkstra
4. Floyd
5. Prim
6. Kruskal

Supportive:

1. Priority Queue
2. Collections Sort
3. Heap Sort

Infrastructure:

1. Hash Maps
2. ArrayList
3. Graph
4. Array
5. Hash Table

4. Discarded Solutions

Pathfinding:

DFS: Helps find cycles and explore deep through the graph but its not efficient or made to find shortest paths.

Prim: While it can find a short path depending on the root it might not find the most optimal path.

Kruskal: Same as prim but the difference is that there might be a direct path between two vertexes that gives the minimum weight but might not take it into account as it starts connecting the vertexes by smallest edges so might end up linking the smallest edge but give my route the longer accumulated path.

Supportive:

Collections Sort: This would help if we were organizing specific attributes but this program is object intensive so it can't compare to an in-built comparator for a priority queue.

Infrastructure:

Array: Having a fixed array is not helpful unless we want to limit something if not array list is always better for expansion purposes.

5. Evaluation criteria:

With the purpose to find the best solution to be implement, we will create an evaluation system with a numeric value to make easier the evaluation:

- A) Efficiency
- B) Usability
- C) Maintainable
- D)Scalability

Pathfinding:

BFS:

A) 3
B) 2
C) 3
D) 2

=10

Dijkstra

A) 4
B) 4
C) 4
D) 4

=16

Floyd

A) 5
B) 4
C) 4
D) 3

=16

Supportive:

Priority Queue

A) 3
B) 4
C) 4
D) 4

=15

Heap Sort

A) 3
B) 3
C) 3
D) 4

=13

Infrastructure:

Hash Maps

A) 5
B) 4
C) 4
D) 5

=18

Array List

A) 3
B) 5
C) 3
D) 3

=14

Graph

A) 4
B) 5
C) 4
D) 5

=18

Hash Table

A) 4
B) 4
C) 4
D) 3

=15

We have determined after examining evaluation criteria and for implementation the most appropriate functions:

- For **pathfinding** we will use Dijkstra for the path and Floyd for the path weight. This helps focus on each of its strengths rather than having to search through arrays or crosscheck adjacent vertices.
- For **support** we will use the priority queue not only for its efficiency and ease of use but because of its versatility in comparators that help in comparing objects especially when working with generics.
- For Infrastructure we have decided to pick Graph and HashMap function for its implementation as this combination will increase efficiency, but we will also include array list for ease of usability.

6. **PREPARATION OF REPORTS AND SPECIFICATIONS:**