Assignment 5: Social Network

Lee Kai Yang
(23205838 - 100%)
kai.y.lee@ucdconnect.ie

May 2, 2024

Video Link GitHub Link

Abstract - This project focuses on developing a database access layer for a social network application, enabling users to execute queries efficiently, including finding mutual friends and determining the shortest path between users. Utilizing graph theory and algorithms, the system represents social connections as an undirected, unweighted graph stored in memory. Through meticulous design decisions, such as employing adjacency lists and selecting optimal algorithms like Dijkstra and BFS for pathfinding, the project achieves responsiveness within the desired latency range of 100ms to 1000ms. Performance evaluations, conducted on a subset of the Facebook WOSN Links dataset, demonstrate the effectiveness of the implemented methods. This work not only provides insights into the underlying mechanisms of social networks but also underscores the importance of benchmarking and algorithm selection in optimizing system performance.

1 Background

Social networks has and only been something abstract for a long time up until the Internet boom along with the rise of online social medias such as MySpace, Friendster and most notably Facebook [1], [2] where software engineers had to find a way to describe relationship between one user and another to form a social circle whereby users interact with each other on the platform through this "network" where one can be friends with another and one can meet new friends through mutual friends. One intuitive way to represent these social networks in computer memory is through the use of graphs where every user is a vertex in a huge network and their relationship is described by the edges between them [3]. By doing so, it is possible to store these relationship as a state in computer memory and identifying mutual friends or looking for friends of friends can be done so by finding the intersection of two graphs and graph traversal.

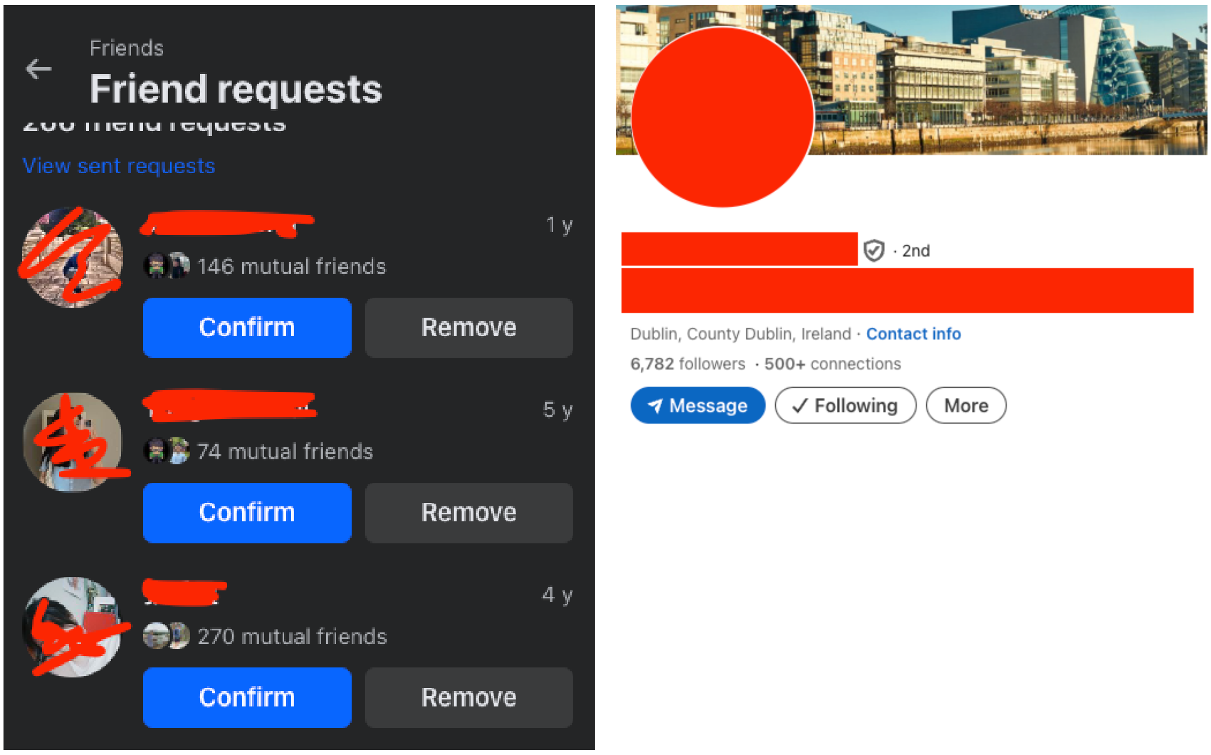
2 Introduction

The goal of this project is to create a database access layer in which users can make two type of queries in a responsive manner:

1. Find mutual friends between two users
2. Determine how far away one user from another user

As for the measure of "responsive", it should be between 100ms to 1000ms according to an experiment conducted by the performance team at Sentry [4].

The idea for these queries is to accomodate features as shown in Figure 1 below for a social network application.



Friends

Friend requests

View sent requests

[redacted] 1 y

146 mutual friends

Confirm Remove

[redacted] 5 y

74 mutual friends

Confirm Remove

[redacted] 4 y

270 mutual friends

Confirm Remove

[redacted]

[redacted] · 2nd

[redacted]

Dublin, County Dublin, Ireland · Contact info

6,782 followers · 500+ connections

Message Following More

Figure 1: Social network features

On the left in the figure above, it shows that when a user sends a friend request to another user, it is common for the application to show how many mutual friends they both share and who they are. On the right, the word "2nd" represents that this user has a second-order distance from the current user or in simpler terms the user is a "friend of friend".

However the challenge here is to find an algorithm to accomplish this with a good scaling factor since most social media platforms nowadays have massive amount of users and traversing through the entire graph is simply impossible.

2.1 Social graphs

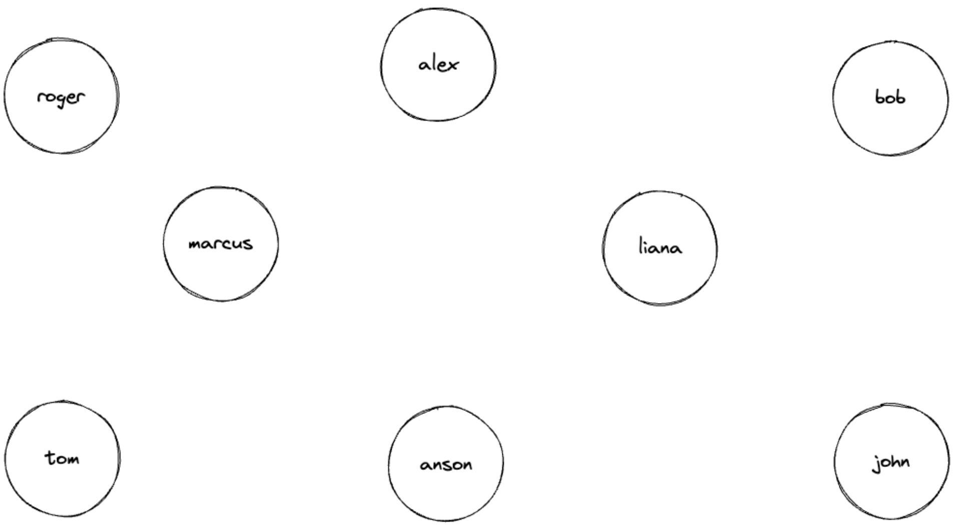


Figure 2: Unconnected Social Graph

Figure 2 shows a graph where the nodes are all unconnected, this is the initial state for a social network where everyone has an account registered but they have not make friends with anyone else.

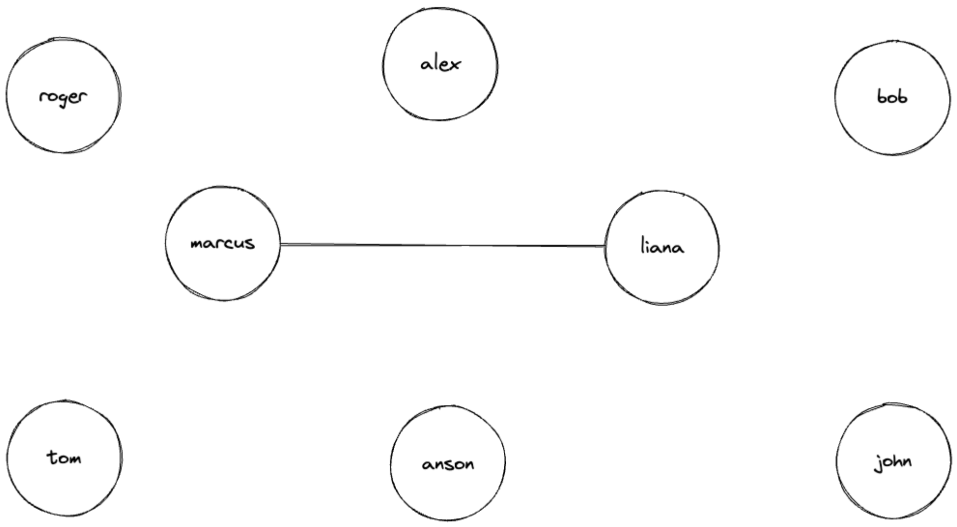


Figure 3: Social Graph (one connected)

Figure 3 shows that "marcus" and "liana" are now friends since there is an edge connected them two and note that this edge is not directed because the relationship exist both ways, "marcus" is a friend of "liana" and at the same time "liana" is a friend of "marcus".

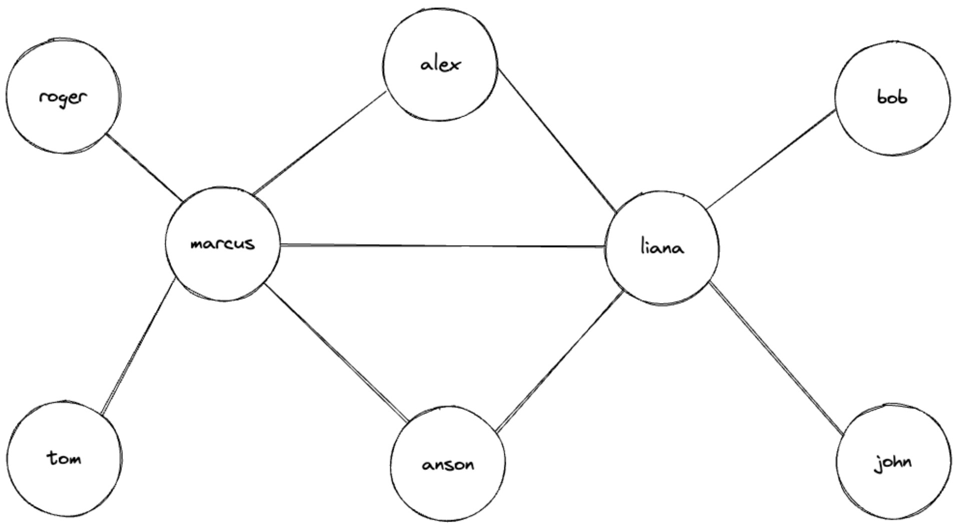


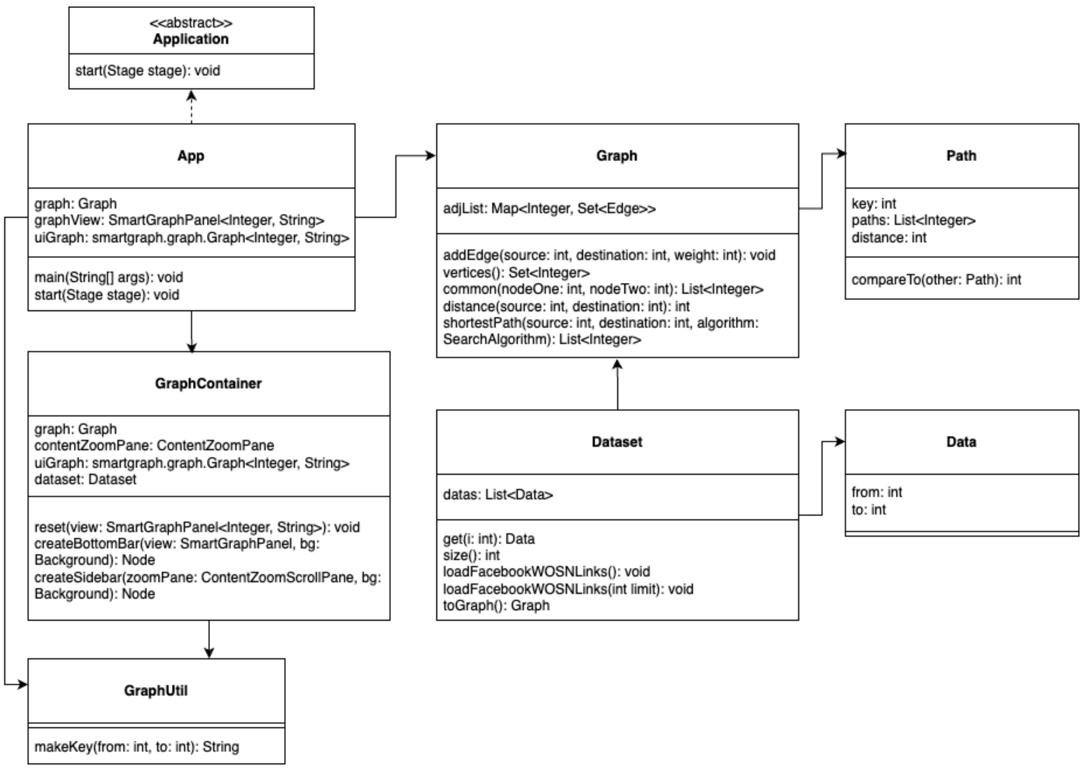
Figure 4: Social Graph (fully connected)

Figure 4 shows a fully connected graph where everyone has at least one friend or more. In this case, "alex" and "anson" are the mutual friends between "marcus" and "liana". Furthermore, "liana" has a 2nd-order relationship with "roger" or also known as "friend of friend". Also, it is good to point out that the weights for all the edges are equal.

3 Design decisions

As can be seen from the explanation above, social networks can be represented as an undirected unweighted graph. Hence, we can store them efficiently in memory as adjacency lists using a HashMap. Details of how the code is implemented is detailed below.

3.1 Classes overview



<<abstract>>
Application
start(Stage stage): void
App
graph: Graph
graphView: SmartGraphPanel<Integer, String>
uiGraph: smartgraph.graph.Graph<Integer, String>
main(String[] args): void
start(Stage stage): void
GraphContainer
graph: Graph
contentZoomPane: ContentZoomPane
uiGraph: smartgraph.graph.Graph<Integer, String>
dataset: Dataset
reset(view: SmartGraphPanel<Integer, String>): void
createBottomBar(view: SmartGraphPanel, bg: Background): Node
createSidebar(zoomPane: ContentZoomScrollPane, bg: Background): Node
GraphUtil
makeKey(from: int, to: int): String
Graph
adjList: Map<Integer, Set<Edge>>
addEdge(source: int, destination: int, weight: int): void
vertices(): Set<Integer>
common(nodeOne: int, nodeTwo: int): List<Integer>
distance(source: int, destination: int): int
shortestPath(source: int, destination: int, algorithm: SearchAlgorithm): List<Integer>
Path
key: int
paths: List<Integer>
distance: int
compareTo(other: Path): int
Dataset
datas: List<Data>
get(i: int): Data
size(): int
loadFacebookWOSNLinks(): void
loadFacebookWOSNLinks(int limit): void
toGraph(): Graph
Data
from: int
to: int

Figure 5: UML Diagram

As can be seen from Figure 5, there are a few classes in the application:

- **Graph** - A data structure to represent interconnected, undirected graphs.
- **Dataset** - A helper class that is purposed for loading external datasets.
- **App** - The entrypoint for the GUI visualiser.
- **GraphContainer** - A wrapper container UI component for the graph in GUI.
- **GraphUtil** - A utility class providing helper methods.

3.2 Graph

This class is where all the main logic resides, it maintains the graph in a HashMap as an adjacency list. Hence, the important operations such as `common()` and `shortestPath()` are found in this class.

Since the graph is stored as an adjacency list, the `common()` method can be implemented by simply finding the intersection between the edges of the two vertices. However for the `shortestPath()` method, there are plenty of path traversal algorithm that we can use such as Bellman-Ford, Floyd-Warshall, Dijkstra, etc. Considering the performance requirements, I decided to implement Dijkstra algorithm and a simple breadth first search (BFS) since these algorithms does not need to pre-compute the graph and is much more dynamic to change as in we don't have to recompute distances for the entire graph every time a new vertice or edge are added. Hence when calling the `shortestPath()` method, it is required to specify which search algorithm should be used by passing in the `SearchAlgorithm` enum which is defined as:

```
public enum SearchAlgorithm {
    Dijkstra,
    BreadthFirstSearch
}
```

To find the shortest path between two vertices, one can use either specify `SearchAlgorithm.Dijkstra` or `SearchAlgorithm.BreadthFirstSearch`:

```
graph.shortestPath(from, to, SearchAlgorithm.Dijkstra);
graph.shortestPath(from, to, SearchAlgorithm.BreadthFirstSearch);
```

3.3 Dataset

This class is used to help simplify the process of loading external datasets and transforming them into the `Graph` class. In this project we used the public dataset Facebook WOSN Links [5] which can be loaded using the method `loadFacbookWOSNLinks()`. An example of using the class is as follows:

```
Dataset dataset = new Dataset();

try {
    dataset.loadFacebookWOSNLinks();
    System.out.printf("Loaded %d data points\n", dataset.size());
} catch (Exception e) {
    System.err.println("Failed to load data: " + e);
}

Graph graph = dataset.toGraph();
```

3.4 App

This class along with **GraphContainer** and **GraphUtil** are all related to the GUI made to showcase and visualise the operations for the **Graph** class using a subset of the Facebook WOSN Links dataset.

4 Test cases

The test cases for the aforementioned classes can be found in `src/test/java/graph` and can be ran by executing `./gradlew test` in the terminal.

5 Examples

5.1 Friends Explorer

To visualise the graph I build a visualizer using JavaFX. What it does is simply loading a subset of the Facebook WOSN Links dataset, build a graph and visualise them, at the same time allow users to perform two operations:

1. Find the common vertices (mutual friends) between two nodes.
2. Find the shortest path between two nodes.

Note that we only load 60 data points otherwise the graph is too big to even see visually. To run the visualizer, run `./gradlew run -Plaunch=graph.example.App` in the terminal.

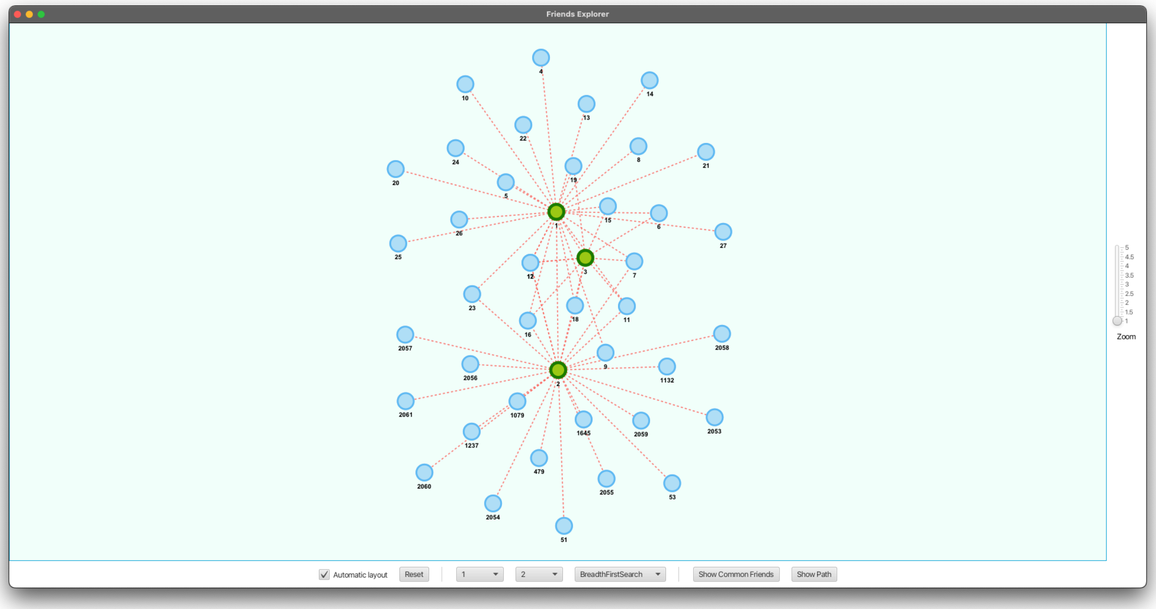


Figure 6: Friends Explorer (Start screen)

Figure 6 shows the initial screen when the visualizer is opened. As can be seen, the side bar on the right shows the current zoom level, a user can scroll their mouse wheel to control the zoom. Moreover, there is a bottom bar where users can select the **from** and **to** vertices and perform the aforementioned operations.

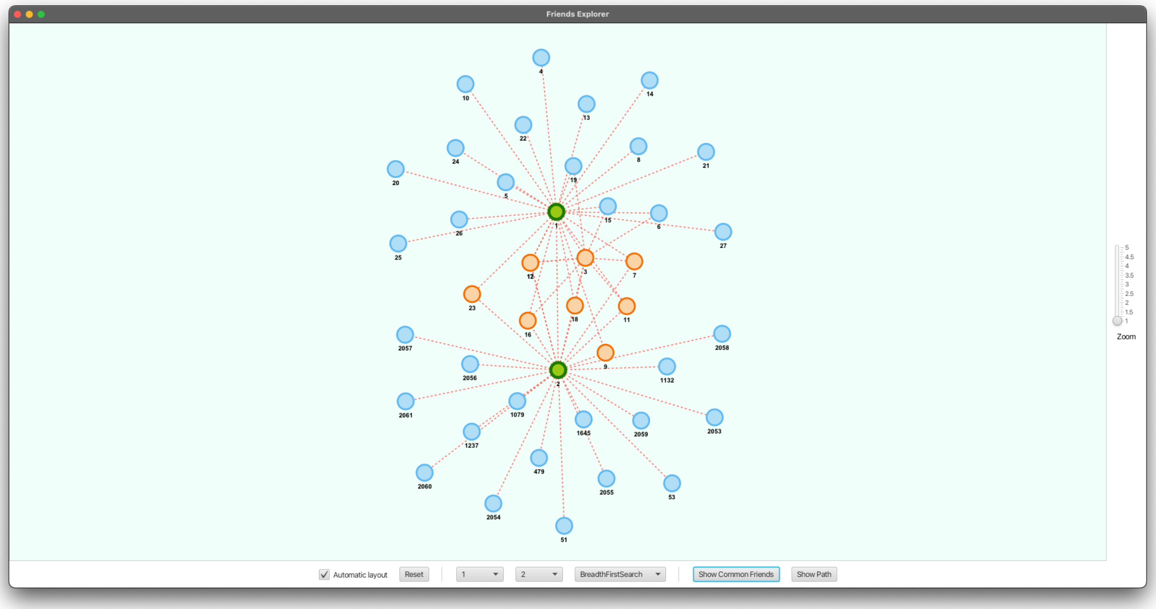


Figure 7: Friends Explorer (Common friends)

Figure 7 shows the visualiser when a user asks for the common friends between “1” and “2”, the common friends are those vertices highlighted in orange.

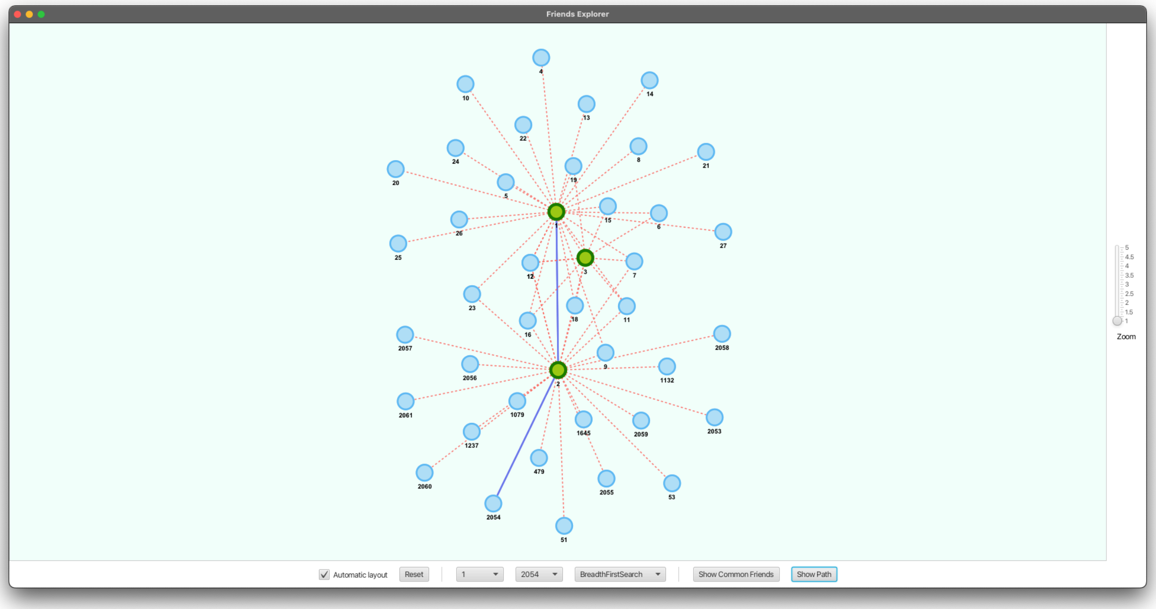


Figure 8: Friends Explorer (Shortest path)

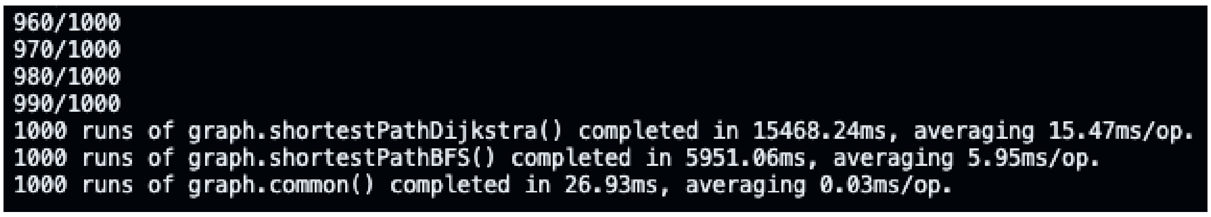
Figure 8 shows the visualiser when a user asks for the shortest path between “1” and “2054”, the path is highlighted in blue.

For a more detailed demonstration of the visualiser refer to the video linked at the beginning of the paper.

5.2 Benchmark

The goal of this project is to find a way to provide the two operations aforementioned, finding mutual friends and shortest path between two vertices in a responsive manner where the latency is ideally between 100ms - 1000ms.

To find out the performance of the graph database I built, I wrote a benchmark program which load the entire Facebook WOSN Links dataset which has 817035 data points and measure the time taken for `graph.common()`, `graph.shortestPath() (Dijkstra)` and `graph.shortestPath() (BFS)` across 1000 runs. The benchmark can be ran by the command `./gradlew run -Plaunch=graph.example.Benchmark`.



```
960/1000
970/1000
980/1000
990/1000
1000 runs of graph.shortestPathDijkstra() completed in 15468.24ms, averaging 15.47ms/op.
1000 runs of graph.shortestPathBFS() completed in 5951.06ms, averaging 5.95ms/op.
1000 runs of graph.common() completed in 26.93ms, averaging 0.03ms/op.
```

Figure 9: Benchmark results

Figure 9 shows the output of the benchmark program and as can be seen the `graph.common()` is indeed very fast which is expected since it is merely finding intersections between two vertices which has a time complexity of `O(m+n)` where `m` is the number of edges for node one and `n` is the number of edges for node two.

Comparing the latencies for the two shortest path algorithms, we see that in our case BFS is much faster than Dijkstra by almost 3 times, this is due to the fact that the BFS algorithm has a time complexity of `O(V + E)` but the Dijkstra is `O(V + E log V)`. The reason that Dijkstra has an extra log term in its time complexity is because it uses a priority queue to keep track of the current shortest path but BFS does not. However, Dijkstra is still a solid algorithm for cases where the graph is weighted and with added heuristics it can be optimised to go even faster which essentially is an A* algorithm but in our case because our graph is unweighted, there is no benefit in using Dijkstra, BFS is superior in our case.

That said, all three operations are well under the 1000ms limit which comforms to our initial performance goals. Note that in our benchmark we do not consider network latencies, it is only measuring the time elapsed for each function call and the results in a real production environment will likely differ a lot with the added HTTP overhead, etc.

6 Conclusion

In conclusion, I am satisfied with what this project had achieved and it helped me "demystified" the core ideas behind modern days social media platforms. Also, by completing this project I learned that it is important to test and benchmark the algorithm before making a conclusion which one is better. In this case, Dijkstra's algorithm is the most well-know and dominant path finding algorithm that is being used in Google Maps or any other GPS system but it does not necessarily mean that it will be the best algorithm for our application, after performing benchmark we found that a simple BFS outperforms Dijkstra's algorithm in this application.

Bibliography

[1] Tim Gulden, "The Internet: Boom, Bust and Beyond." Oct. 01, 2002.

[2] Manish Dhingra and Rakesh K. Mudgal, "Historical Evolution of Social Media: An Overview." Mar. 15, 2019. doi: 10.2139/ssrn.3395665.

[3] Dmitri Goldenberg, "Social Network Analysis: From Graph Theory to Applications with Python." May 01, 2019. doi: 10.13140/RG.2.2.36809.77925.

[4] Lazar Nikolov, "What's the difference between API Latency and API Response Time?." [Online]. Available: https://blog.sentry.io/whats-the-difference-between-api-latency-and-api-response-time/#:~:text=Just%20to%20have%20a%20number,to%20abandon%20the%20application%2Fwebsite.

[5] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi, "On the Evolution of User Interaction in Facebook." Aug. 01, 2009. [Online]. Available: https://socialnetworks.mpi-sws.org/data-wosn2009.html