Shawn Lau
DS210
Professor Kontothanassis
5 May 2024

Project Report
Dataset: https://users.cs.utah.edu/~lifeifei/SpatialDataset.htm

Objective: This project uses a San Francisco Road Network composed of 174,956 nodes representing road intersections, and 223,002 edges representing roads. The goal of this project is to investigate the average distances from sampled nodes to the rest of the nodes in the set.

Cleaning and Preparation: The dataset of edges contains (edge id, start node, end node, weight) in the form of a text file. There are no errors or missing values in the set. Before working on the set, the edges were extracted by reading the start node and end node via Rust's std File reader. For Dijkstra's algorithm, the weights were extracted from the text file as well. The edges are turned into a vector of Vec<(usize,usize)> representing the start and end nodes. This vector is then turned into an adjacency list of Vec<Vec<(usize)>> where the index of the outer vector represents the start node, and the inner vector represents all nodes that it is connected to (edges). The adjacency list is necessary for finding distances. The extract_edges and to_adjacency_list functions are part of a helper module in the source file.

Method: To compute distances in the set, both Breadth First Search and Dijkstra's algorithm were used. Since Breadth First Search is useful for finding distances on unweighted graphs and Dijkstra's algorithm is useful for finding shortest paths of weighted graphs, both can provide different insights into the graph. For the two processes, a random sample of 2000 points is chosen from the set. Breadth First Search and Dijkstra's algorithm are performed on these sampled points and the average distance is calculated and outputted. Essentially, this is the average of the average distance between points.

Breadth First Search: To compute BFS, a queue and a list of distances were used. The queue begins with the starting point. For each neighbor of the first element of the queue (found using adjacency list), if the neighbor has not yet been visited, it is pushed onto the queue and the distance of the neighbor is updated to the current_node +1. This ensures that the distances for neighbors are properly calculated. The main loop is as follows:

```
while !queue.is_empty() {
    let current_node: usize = queue[0];

    for neighbor: &usize in &adjacency_list[current_node] {
        if !visited[*neighbor].0 { //if neighbor not visited
            queue.push(*neighbor);
            visited[*neighbor] = (true, current_node, visited[current_node].2 +1);
        }
    }
    queue.remove(index: 0);
}
```

The BFS data queue follows the principle of first in first out. Since node neighbors are added onto the queue later, the first visit to each neighbor will always be the shortest distance. The time complexity of BFs is O(V + E), where V represents the number of nodes and E represents the number of edges. This aligns with the code.

Dijkstra's Algorithm: Dijkstra's algorithm was used with the set of edges along with their associated weights that are provided in the text file. An additional extract_weighted_edges function was used to extract edges which were then turned into an weighted adjacency list. To implement Dijkstra's algorithm, two approaches were used. On the first attempt, a list of tuples containing (node visited, shortest distance) was created. On initialization, all nodes have a distance of infinity/MAX value since these points have not yet been visited. Starting from the first node, if an unvisited neighbor has a shorter distance than what is tracked in the distance list, the value is updated. Then, the next node in the iteration is selected by choosing the next unvisited point of the shortest distance. This next node is marked as visited and the process repeats until all points have been visited.

```
while visited_count != node_count { //assuming every node can be visited

    for (neighbor,weight) in &weighted_adjacency_list[current_node] { //(node, weight)
        if !visited[*neighbor].0 && *weight + visited[current_node].1 < visited[*neighbor].1 {
            // if neighbor unvisited and distance shorter, update distance
            visited[*neighbor].1 = *weight + visited[current_node].1;
        }
    }
    current_node = next_shortest(&visited);
    visited[current_node].0 = true;
    visited_count += 1;
}
```

However, this process was highly inefficient because the next shortest distance was found by iterating through all nodes (in this case 174,956 nodes). Running the program took a significant amount of time, even for a small sample size of 10.

The second version of Dijkstra's algorithm implementation used a priority queue and a list of distances. By using a priority queue, the amount of computations is reduced and the program runs significantly faster. For the priority queue, Rusts's Binary Heap was used. However, since the language automatically opts for a Maximum heap, and floating points numbers do not support ordering and partial ordering, changes were made to support the priority queue. The new min heap was created using a struct which implements partial ordering and ordering. This was guided by code found on stack overflow (see sources). The list of distances is initialized with all MAX values, as these points have not been visited. The starting point is pushed onto the queue. While the queue is not empty, each neighbor of the current node is checked. If the total distance to this node is less than what is stored in the distance list, the distance value is updated and the neighbor is pushed onto the priority queue. By nature of the priority queue, the shortest distance nodes will always be at the top, thus being next in line. This is the basis of how Dijkstra's algorithm works, with shortest distances being the next visited node. This ensures that the shortest paths are always found. The main loop appears as follows:

```rust
let mut pqueue: BinaryHeap<Rev> = BinaryHeap::new(); //holds (dist, node)
let mut dist: Vec<f64> = vec![f64::MAX; node_count];

dist[node] = 0.0;
pqueue.push(item: Rev(0.0, node));

while !pqueue.peek().is_none() {
    let current_node: usize = pqueue.pop().unwrap().1;

    for (neighbor: &usize, weight: &f64) in &weighted_adjacency_list[current_node] {
        if dist[current_node] + *weight < dist[*neighbor] {
            dist[*neighbor] = dist[current_node] + *weight;
            pqueue.push(item: Rev(dist[current_node] + *weight, *neighbor));
        }
    }
}
```

The algorithm returns a vector of shortest distances. The time complexity of Dijkstra's algorithm is O(V+E(l o g(V))) where V represents the number of nodes and E represents the number of edges. This aligns with the code and makes sense because of the priority queue or minimum heap.

Average Distances:
The Breadth First Search and Dijkstra's functions output a list of distances from each node. The average node distance is calculated by finding the average distance of these lists (sum / visited nodes). For the sampled 2000 points, the average is then computed by finding the average of these average distances (sum average node distance / 2000).

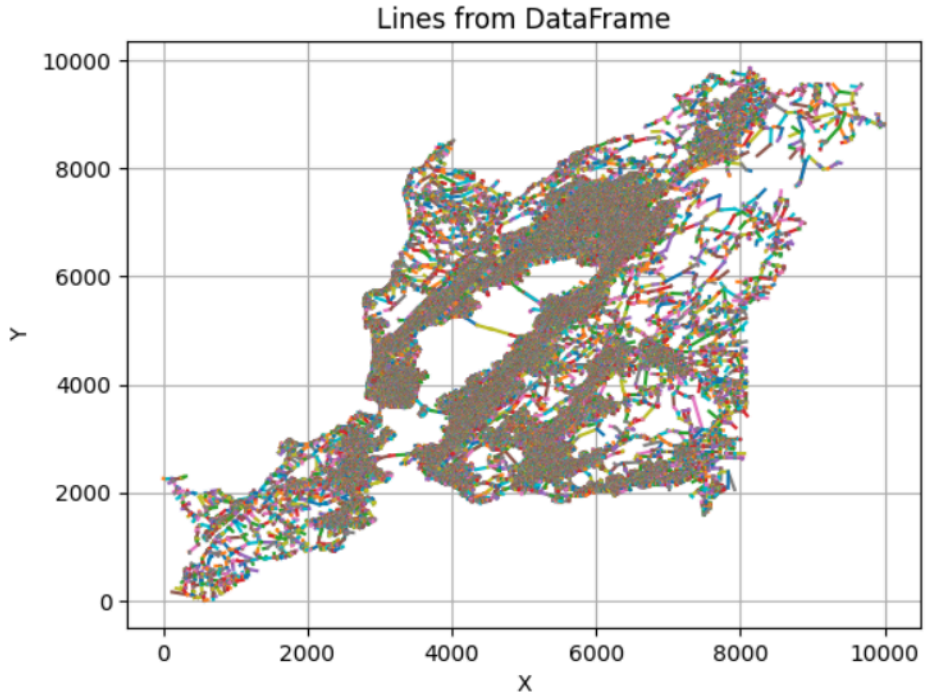
Output and Analysis:

```
Breadth First Search Average: 214.03803959018043
Dijkstra's Algorithm Average: 3652.814803313136
```

Based on the sample of 2000 random points, the average distance for Breadth First Search was about 214 and the average distance for Dijkstra's was about 3652.81 (meters). Considering the nature of the dataset, this makes sense. There are 174,956 nodes in this set, so a BFS distance of 214 is reasonable. Although other graphs, such as social networks, might be able to connect in short distances, an intersection of a road network can only have so many connecting edges. The average distance of 3652 meters converts to about 2 miles. Although this may seem long, this is considering the distances to all other points.

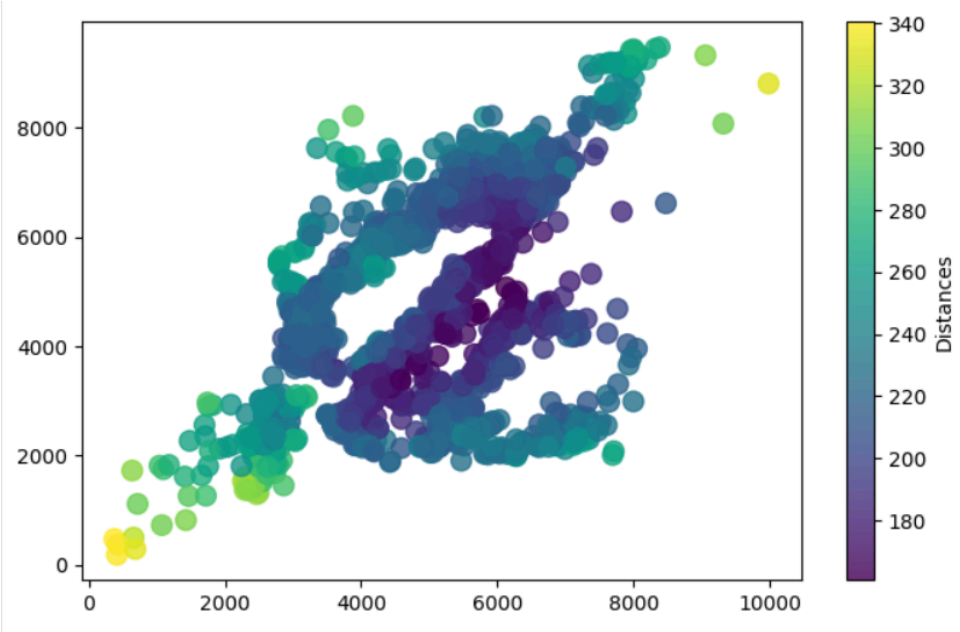
To further confirm results, I decided to graph the average distances using Python.

Road Map



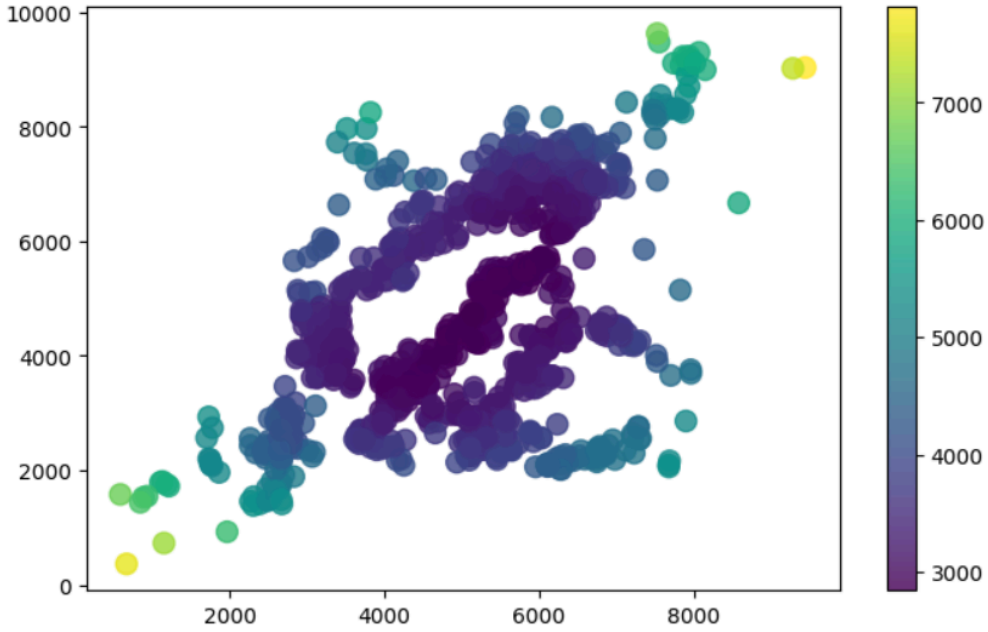
As seen in the visual, the streets are quite dense around the center of the city.

Breadth First Search Distances



The Breadth First Search Distances align with what one might expect from the graph. The nodes nearer to the center of the city have shorter average distances to the rest of the points. This makes sense, as nodes near the outer boundaries would be much further from all points in the set.

Dijkstra's Algorithm Distances



The Dijsktra's Algorithm Distances aligns with both the original road network and the Breadth First Search distances. The nodes closer to the center of the city have a much shorter average distance than the others. As the nodes begin to move away from the center, the average distance increases.

Sources:

https://www.youtube.com/watch?v=bZkzH5x0SKU&ab_channel=FelixTechTips
https://www.youtube.com/watch?v=EFg3u_E6eHU&ab_channel=SpanningTree
https://takeuforward.org/data-structure/dijkstras-algorithm-using-priority-queue-g-32/
https://doc.rust-lang.org/std/collections/struct.BinaryHeap.html
https://stackoverflow.com/questions/39949939/how-can-i-implement-a-min-heap-of-f64-with-rusts-binaryheap
https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/#:~:text=Dijkstra's%20algorithm%20uses%20a%20priority,of%20vertices%20sorted%20by%20distance.