COSC 320 - 001
Alalysis of Algorithms
2020 Winter Term 2

Project Topic Number: #5

Shortest Flight Path

Group Lead: Jace Lai

Group Members:
Jace Lai
Jinyang Yao
Lucas Pozza
Tatiana Urazova

April 9, 2021

1 Implementation

The algorithm implementation represents each airport as a vertex and each flight as an edge in a directed graph. Each edge in the graph is assigned a weight or cost value using a cost function that combines the three requirements: minimum wait time, lowest cost, and minimum flight duration. Each vertex in the graph is stored in an adjacency list as a Hash Table. Since Dijkstra's algorithm works with weighted edges directly, no other modification to the graph is required. The algorithm uses a priority queue which stores nodes where the nodes with the lowest cost are at the front of the queue. The algorithm starts by removing the node with the shortest distance from the priority queue.

```
public ArrayList<Node> dijkstra(Graph g, Node s, Node e) {
  PriorityQueue<Node> queue = new PriorityQueue<Node>();
  s.pathLength = 0;
  queue.add(s);

  while (!queue.isEmpty()) {
    Node n = queue.poll();
    for (WeightAdjacent adj : g.get(n)) {
      queue.add(adj.node);

      if (adj.node.pathLength > n.pathLength + adj.weight) {
        adjnode.pathLength = n.pathLength + weight;
        adjnode.setPred(n);

        // Reached destination node
        if (adjnode.name.equals(e.name)) {
          // Traceback builds the path from s to e
          return adjnode.traceback(s);
        }
      }
    }
  }
  return null; // No path was found
}
```

The priority queue ensures that the node with the lowest cost is searched first. We add each adjacent node to the priority queue to explore new paths in the graph. The predecessor (parent) of the adjacent node is also set. This is used to trace the resulting path from s to e. If the current adjacent node is the destination node, then we have successfully found a path from s to e in the graph. The traceback procedure builds the path as a list of nodes from s to e. To find the path traceback uses the predecessor of each node starting from e until it reaches the starting node which has no predecessor. The resulting path is guaranteed to be the shortest path since the nodes were explored in order, with the lowest cost nodes being searched through first.

2 Results

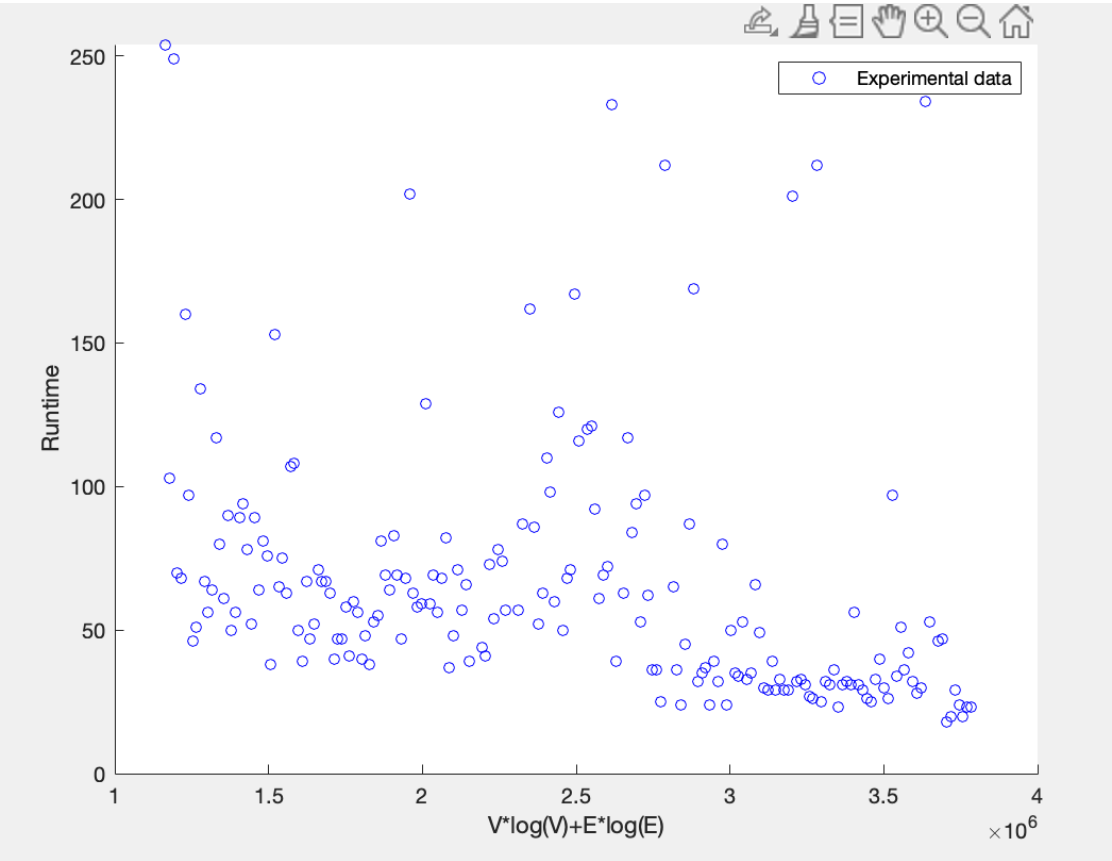

Figure 1: Algorithm runtime & $V * log(V) + E * log(E)$

According to our analysis of the algorithm in Second Milestone, the algorithm has $O(V * \log(V) + E * \log(E))$ runtime complexity. Running the algorithm on different values of V and E and recording the times elapsed, we get a plot of the runtime of our algorithm against $(V * \log(V) + E * \log(E))$.

This graph shows that the data points are not following any specific trend, so we cannot really see if our predictions of the runtime complexity were correct. This might be due to insufficient precision of our measurements. Due to how fast the algorithm ran, we had to record the time in microseconds to measure the runtime. However, the precision of the microseconds measurement could be not enough to record the trend. It is also possible that upon further experimentation and obtaining a larger dataset of airports and flights, we would be able to record experimental data that shows the growth in the runtime as the input size increases.

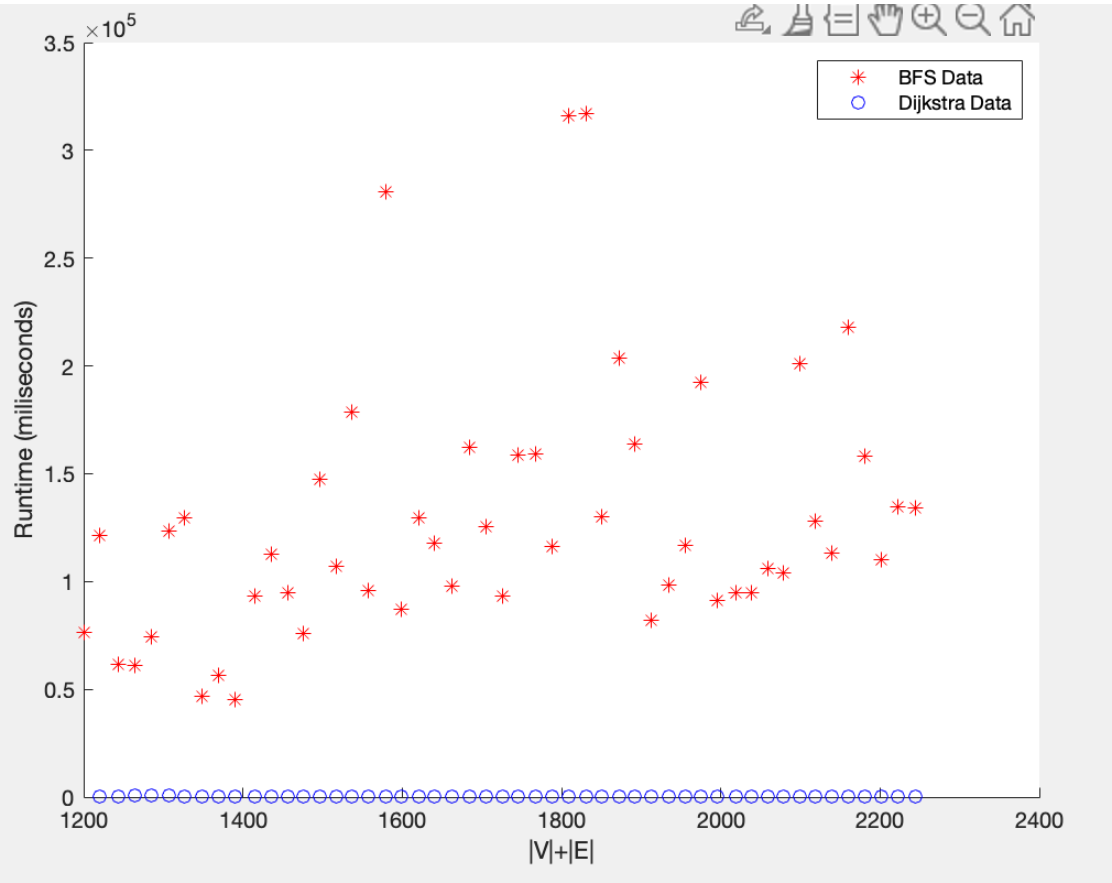

Figure 2: Comparison between Algorithm 1 (Modified BFS) and Dijkstra

Both algorithms have a similar complexity due to the fact that they traverse the entire graph. But our result indicates that Dijkstra is significantly faster than BFS, the reason is because BFS with dummy nodes need to greatly bulk the input size before doing the search, so although both algorithms are seeking for the same result and have similar asymptotic growth, Dijkstra only need to work on the original weighted graph, but BFS need to make a simulation of the weighted graph by making a unweighted graph with dummy nodes only for representing the weight. For BFS, each access of the adjacent node will impose N number of extra accesses. If we are searching in a graph $G(V, E)$, for Dijkstra the complexity of the running time is $E + V * \log V$. For with the BFS method, we add $N = M * E$ nodes in the graph, where M is the average number of nodes inserted between two original nodes. Thus we can say that to visit an adjacent node in the original graph, we need M access. For all edges in the graph the cost of visiting edges will be $E * M$. Because dummy nodes themselves are nodes, so we have $(V + M)$ nodes in total. The final running time will be $EM + (V + N) * \log(V + N)$. Although it's still asymptotically the same as Dijkstra, depending on the nature of the data set the overhead might be noticable. More particularly in our case, we have a very

large number of edges (millions) but a very small number of nodes (hundreds). This means any overhead on the edge will make a significant difference.
Matlab code used for the plots:

```
input=VDijkstra.*log(VDijkstra)+EDijkstra.*log(EDijkstra);

figure,scatter(input,timeDijkstra,25,'b','o')
P = polyfit(input,timeDijkstra,1);
yfit = P(1)*input+P(2);
hold on;
plot(input,yfit,'r-.');
legend('Experimental data',
  sprintf('O(V*log(V)+E*log(E)): y=%.2f*x+(%.2f)',P(1),P(2)));
xlabel('V*log(V)+E*log(E)')
ylabel('Runtime')
hold off
```

3 Unexpected Cases and Difficulties

The algorithm is quite inefficient since we search the entire graph. So far we have been testing the algorithm with a subset of the data so that the algorithm can finish in a reasonable amount of time. In order to test much larger datasets some pruning would need to be done to the graph before running it through the algorithm. Pruning can mean that before running the algorithm we remove all flights that are not within a couple of days within the first flight, since any connecting flight that requires such long layovers is unlikely to be the best possible option. Since our dataset includes data for all flights in a given year, this type of pruning would reduce the search space significantly.

4 Task Separation and Responsibilities

Jace Lai: Slides, presentation video voiceover and editing
Jinyang Yao: Algorithm implementation
Lucas Pozza: Algorithm implementation details and slides.
Tatiana Urazova: Results analysis and testing code