Allocation of Teleporters in Major Cities to Minimize Air Traffic.

Aditya Shankar, Ho Joong Kim, Qingyu Chen, Ryan Harty, Vincent Peng

Abstract

The teleportation allocation problem highlights the usage of Dijkstra's Algorithm to optimize weighted edge removal that will produce minimum air traffic. Given a specific number of airport hubs and teleporters, we created graphs that has hubs as vertices and weighted edges between the hubs reflecting the air traffic between the hubs. Then we used the shortest path algorithm to compute optimal allocation that will minimize air traffic for the case of 10 hubs and 4 teleporters in this paper, and reached air traffic reduction of 69.55%.

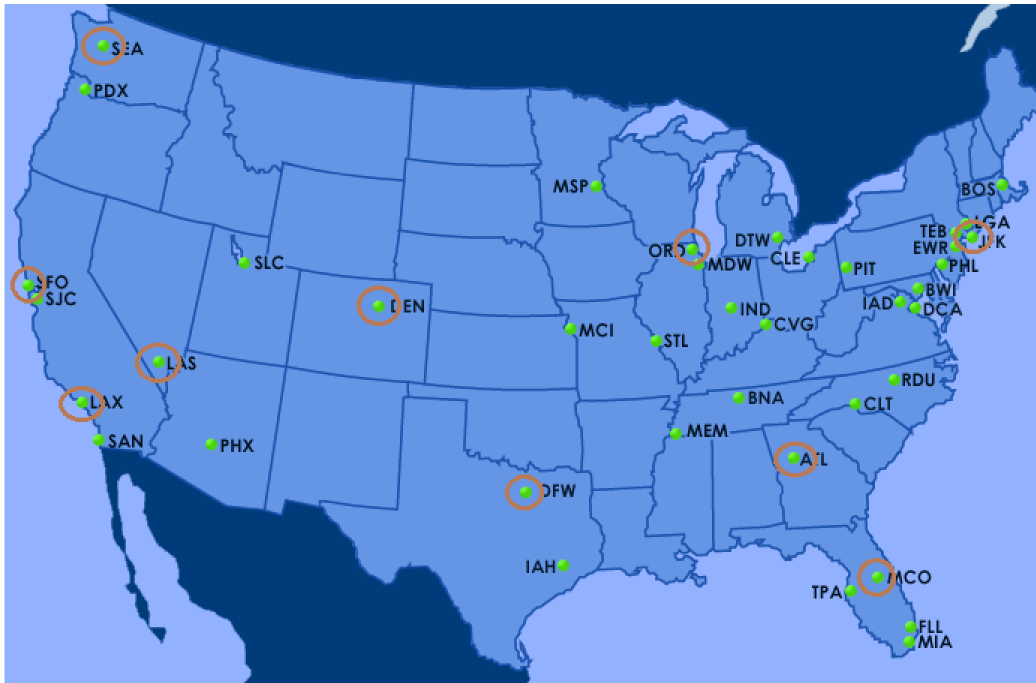
1. Introduction

The year is 20xx, and teleportation technology has finally arrived! While the field of quantum physics has improved by a significant margin to allow for this, the field of Operations Research still has a ways to go, and the assistance of students is required to tackle an important problem regarding air traffic.

Four teleporter systems have been made available by the government to reduce air traffic between 10 airports. A teleporter system is a direct connection between two airports which, when initialized, reduces the total air traffic between those two airports to zero by making it possible for people to instantaneously travel from one of these airports to the other. The problem we are given is how to assign these four teleporter systems in order to minimize the total edge weight (air traffic) between the 10 airports we have chosen. We will use Dijkstra's Algorithm to find the shortest path value for each city for each combination of four teleporter system placements, which will allow us to ascertain the optimal combination of teleporter system placements that minimize the total air traffic between the 10 airports.

Preprint submitted to Elsevier *December 16, 2019*

2. Background



Above is a map of the 10 airports we are using (the circled locations), which can be represented as a complete graph (pictured on Page 9). Installing a teleporter system between two airports has the effect of removing the "edge" between the two airports and merging the two "vertices" together. In theory, this should make the distance between the two airports 0 (though we ran into a problem with our code with this line of reasoning, which will be explained later).

There are a lot of assumptions we made when constructing this problem. Teleportation technology will probably never be implemented in our lifetimes, so we had to think about what exactly we wanted our teleporters to do. Because of the seemingly unlimited amount of options we had, the difficulty of this problem could have been anywhere from extremely easy or extremely challenging, so we found an appropriate middle ground with one teleporter system removing one edge from the graph and merging the vertices it connected. After choosing this option for our teleporters, we had to make some more assumptions regarding how they would actually operate: We had to assume that there would never be technical issues with the teleporters,

everyone who wanted to travel on a direct route a teleporter system was installed for could and would use the teleporters for travel rather than flying, and the normal route a plane took from one airport to another was a straight line, without any stops or complications.

To solve this problem, we made plentiful use of Dijkstra's Algorithm, which was a big concept we were taught in class. Dijkstra's Algorithm finds the shortest path from a single source vertex to all of the other vertices in a graph in a few steps:

1. Assign every node except for the source node a tentative distance value of infinity, and assign the source node a distance value of 0. Keep track of all the visited and unvisited nodes in the graph

2. For the current node (we start the algorithm at the source vertex), look at all of the unvisited neighbors and for each neighbor, find the distance to the current node and add it to the distance from the current node to the neighbor. If this is less than the current tentative distance of the neighbor node, update the tentative distance of the neighbor node with this new value

3. When all of the neighbors of the current node are considered, mark the current node as visited and remove it from the list of unvisited nodes

4. The unvisited node with the smallest tentative distance is now the current node. Go back to Step 2 and repeat the process

(Important note: The algorithm is normally finished when the destination node is marked visited; however, in our problem we don't have a specific destination node so we will keep going until every node has been marked visited)

For each allocation of four teleporter systems, we had to run Dijkstra's Algorithm for each vertex in order to calculate the minimum air traffic/edge weight between all of the vertices, and the minimum air traffic/edge weight among all of the different allocations of four teleporter systems gives us our solution. As Dijkstra's Algorithm is an $O(n^2)$ algorithm, we had to limit the number of airports we could look at and the teleporter systems we could use, or else our code would take too long to run. This issue is discussed in more detail further into our report.

3. Data Collection

Our data collection proved more difficult than we first imagined- after all, while distances between airports are easy enough to find online, the number of passengers traveling those routes are more difficult to secure. The Bureau of Transportation Statistics in the United States Department of Transportation was ultimately our most reliable source of passenger data between United States airports. We used their Airline Origin and Destination Survey, which is a ten percent sample of all airline tickets processed from participating carriers that discloses the origin and destination airports on a certain passenger's ticket, thus revealing their flight path and whether they traveled between two of the airports included on our graph. Moreover, our sample only included tickets from the first quarter of 2019.

Number of passengers traveling from origin airport to destination airport, 10% survey Q1 2019

		Destination									
		ATL	LAX	ORD	DFW	DEN	JFK	SFO	SEA	LAS	MCO
Origin	ATL	N/A	1604	1162	1544	1425	600	746	1799	882	1440
	LAX	3499	N/A	3565	2460	2056	2159	1723	1660	1470	773
	ORD	1586	1875	N/A	1503	1601	198	1545	1587	743	736
	DFW	2294	1684	1936	N/A	1190	328	858	1092	708	839
	DEN	2254	1257	2071	1388	N/A	208	1281	1405	716	708
	JFK	653	1668	281	421	209	N/A	1091	622	643	513
	SFO	1644	1798	2614	1440	1780	1344	N/A	1461	1001	295
	SEA	2034	1083	1487	1627	1725	650	1235	N/A	823	297
	LAS	2508	2469	1896	2366	2069	641	1117	1283	N/A	234
	MCO	5366	793	2311	2194	1292	591	426	497	291	N/A

Airport City Key

ATL	Atlanta, GA
LAX	Los Angeles, CA
ORD	Chicago, IL
DFW	Dallas-Fort Worth, TX
DEN	Denver, CO
JFK	New York City, NY
SFO	San Francisco, SF
SEA	Seattle, WA
LAS	Las Vegas, NV
MCO	Miami, FL
EWR	Newark, NJ
CLT	Charlotte, NC

The airports that we chose to included in our graph were the following: Hartsfield-Jackson Atlanta International Airport (ATL), Los Angeles International Airport (LAX), O'Hare International Airport (ORD), Dallas/Fort Worth International Airport, (DFW), Denver International Airport (DEN), John F. Kennedy International Airport (JFK), San Francisco International Airport (SFO), Seattle-Tacoma International Airport (SEA), McCarran International Airport (LAS) and Orlando International Airport (MCO). These are the twelve busiest airports as designated by Federal Aviation Administration data, and we decided on ten airports for our study to balance model run time and regional representation. For our project, we decided to place teleporters in such a way that they minimized total air traffic- with air traffic being the product of an airline route's distance in miles and its number of passengers along that route. Our data is displayed below.

Air traffic (Distance times number of passengers) between any two airports in our graph

	ATL	LAX	ORD	DFW	DEN	JFK	SFO	SEA	LAS	MCO
ATL	0	9910026	1665288	2797902	4400084	952280	5102650	8344441	5905380	2756430
LAX	9910026	0	9476480	5109552	2852493	9452690	1190098	2619565	933543	3467124
ORD	1665288	9476480	0	2758078	3257064	353502	7665037	5274984	3987529	3068329
DFW	2797902	5109552	2758078	0	1652498	1040361	3361974	4508102	3236922	2984472
DEN	4400084	2852493	3257064	1652498	0	676374	2956926	3198860	1746195	3090000
JFK	952280	9452690	353502	1040361	676374	0	6284735	3071880	2880012	1044384
SFO	5102650	1190098	7665037	3361974	2956926	6284735	0	1830584	876852	1761403
SEA	8344441	2619565	5274984	4508102	3198860	3071880	1830584	0	1823796	2025494
LAS	5905380	933543	3987529	3236922	1746195	2880012	876852	1823796	0	1068900
MCO	2756430	3467124	3068329	2984472	3090000	1044384	1761403	2025494	1068900	0

We calculated air traffic by using the passenger and distance statistics from our Bureau of Transportation Statistics data, creating a 10x10 matrix that reflected passenger numbers for the air routes in our study. Since we chose the 10 largest airports in the United States, our airports and flights formed a complete graph, and all that was left was to ensure that the passengers on a route from A to B were added to those on a route from B to A (and vice-versa), as a teleporter system between points A and B would reduce air traffic equally in each direction. At this stage, we had the input we needed for our shortest path algorithm- an adjacency matrix, symmetric on the diagonal, that weighted the edges in our graph based on their air traffic value (passengers times distance for an air route between two cities).

Obviously, the data we used was not perfect, but rather the best we could manage for a project with a short time horizon. The Bureau of Transportation Statistics data was only a ten percent sample of airline tickets from participating carriers, and when we compared the list of participating carriers to the list of largest air carriers in the United States, 1 large carrier and a few mid-size carriers were missing. This could certainly account for regional bias, which seemed evident in two of the airports we studied- JFK and MCO. JFK is well-known as one of the largest airports in the United States, but our data showed relatively low passenger numbers, a phenomenon that we found puzzling. While we originally thought this might be due to JFK processing more international flights than other airports, we realized that this would still be present in our data- if a passenger flew into the United States through JFK, they would still have to fly somewhere else from JFK, and if

this destination were one of our chosen locations, it would still show up in our study. Another source of variation in our model is the data surrounding the MCO airport- two of the five busiest airline routes included as edges in our model featured MCO despite MCO's status as only the 10th busiest airport in the United States (as per FAA data). These are the types of regional variation that can occur due to sampling preferences, and since we unsure if the 10 percent sample was truly random or not, we cannot be entirely sure that our model is unbiased in this way.

Finally, a last weakness of our data is that our air traffic values are quarterly in the United States for sample of airline carriers, so it would prove difficult to convert them to yearly, nation-wide absolutes that convey the true number of passenger-miles the optimal teleporter placement has saved. An example of this is that while the ATL-ORD air route is estimated to serve 2.7 million passengers a year as per the FAA, scaling our data up to a yearly, 100 percent airline ticket basis still only indicated about 460,000 passengers along this route. Clearly, there is another scaling factor that must be applied to account for sample bias here, and without knowing the exact sampling procedure used, it would be foolish to attempt to bring our quarterly, 10 percent sample results up to a yearly, nationwide basis. In essence, while we have confidence in the values of our data relative to other values in our data, we would need better data to find the true savings of the teleporters- the structure of our data lends to more accurate allocation than cost analysis results.

4. Method

4.1. Set Up

As our goal was to minimize air traffic, which factors in frequency of flight and distance between two airports, by placing a teleporter. We set up the problem as follows:

- Represent the n airport hubs as nodes, or vertices in a graph.
- Represent the air traffic between two airport hubs as a weighted edge in a graph.
- If a teleporter is placed between two airport hubs, it reduces the air traffic of the edge between the two hubs to zero.
- Select up to k edges that will have their air traffic reduced to zero.
- Find the best possible placement of teleporters to minimize total air traffic.

In this problem, n would be 10, and k would be 4. With graph construction fit to USA's landscape, the original complete weighted graph looks like below.

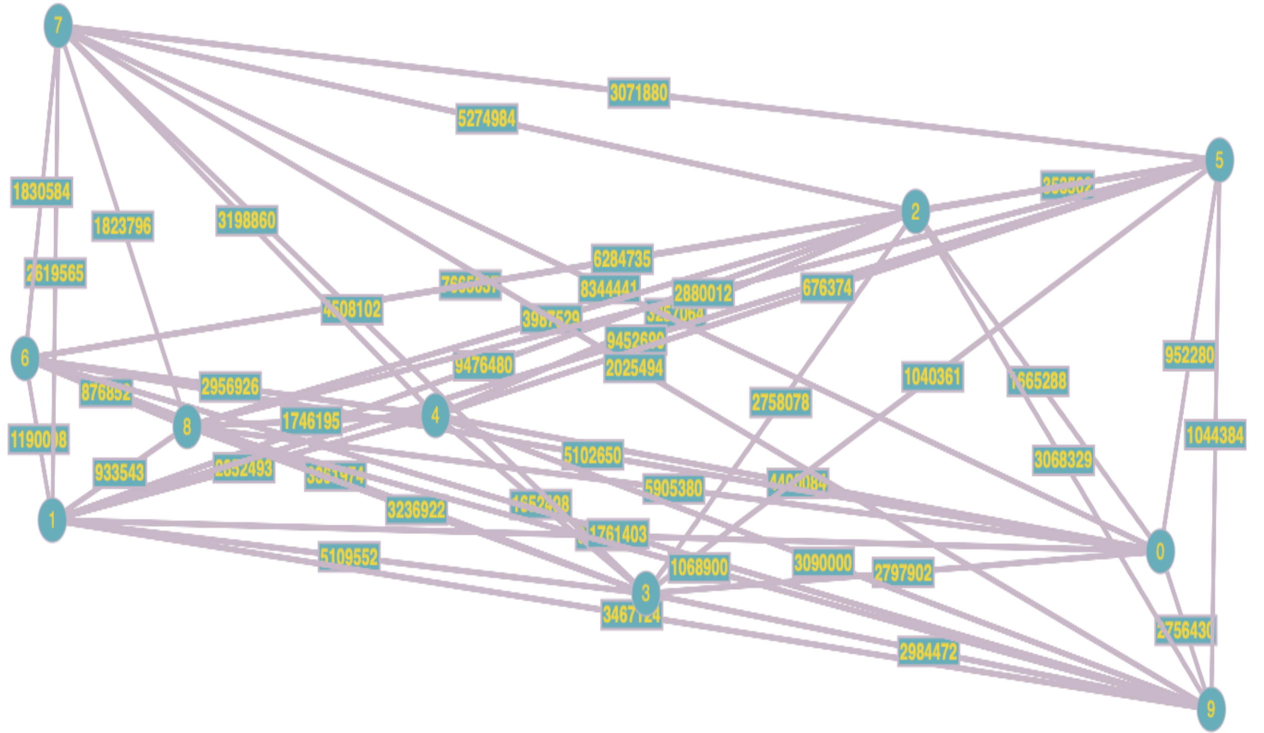


Figure 1: Original Complete Weighted Graph

We do not simply reduce the top five edges with most air traffic, but instead, using the shortest path algorithm and fast machine calculation, we create all possible instances of original graph with k weighted edges reduced to zero, and run the algorithm on each instances to deduce the optimal reduction.

4.2. Approach

We used python to code out the process. First we considered the airports as nodes that will be represented as columns and rows of adjacency matrix. Based on the real air traffic data we collected from major airport hubs in the country, we created a table with weighted number representing the amount of people traveling between two cities and multiplied that number with the distance between two cities. Then, we create an n-complete graph, with each edge between nodes weighted with the traffic number obtained from the table. From $\binom{n}{2}$ total edges, a teleporter system will erase the weight of

the edge connecting two nodes to zero, and total of k edges will have their weights reduced to zero. Thus $\binom{\binom{n}{2}}{k}$ possibilities of sample graphs that have unique k edges' weights reduced to zero will be created.
For each instances of the graphs, we would run the Dijkstra's Shortest Path Algorithm for each vertices to all other vertices. We then sum up the value output from the algorithm for all vertices. This number will represent the unique instance of graph's air traffic reduction resulting from the teleporter placement.
Then take the minimum of the summed up values of all $\binom{\binom{n}{2}}{k}$ instances. Analyze the result and determine which weighted edge reduction led to the optimal result.

4.3. Adjustments

While writing out the code, an error occurred where the summed up output of running the shortest path algorithm for each instance of a graph returned identical number amongst all instances. Struggling to find out the cause, we separately ran bits of our codes on much smaller scaled graphs, and soon found out that our adjacency matrix used for data input caused unintentional bug. While the air traffic adjacency matrix represented the same airport to its own as zero, when we reduced certain edges' air traffic to zero, the program recognized the reduced edge as infeasible air service. This caused our program to end up with same instance of graph all the time with same edges removed all the time. After fixing the issue, the bug disappeared, and unique results were displayed for each instances of the graph.

```
# Library for INT_MAX
import sys
import math
import copy
from itertools import combinations

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        cur_min = sys.maxsize

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < cur_min and sptSet[v] == False:
                cur_min = dist[v]
                min_index = v

        return min_index

    # Funtion that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
```

```python
    def dijkstra(self, src):

        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shotest path tree
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and
                   dist[v] > dist[u] + self.graph[u][v]:
                   dist[v] = dist[u] + self.graph[u][v]

        #print(sum(dist))
        return sum(dist)

# Driver program
g = Graph(10)
#12 * 12 matrix
original_graph = [[0, 9910026, 1665288, 2797902, 4400084, 952280,
    5102650, 8344441, 5905380, 2756430, 1848345, 666699],
         [9910026, 0, 9476480, 5109552, 2852493, 9452690, 1190098,
             2619565, 933543, 3467124, 4479221, 5720337],
         [1665288, 9476480, 0, 2758078, 3257064, 353502, 7665037,
             5274984, 3987529, 3068329, 1549437, 1543024],
         [2797902, 5109552, 2758078, 0, 1652498, 1040361, 3361974,
             4508102, 3236922, 2984472, 2056370, 3369872],
```

```
        [4400084, 2852493, 3257064, 1652498, 0, 676374, 2956926,
           3198860, 1746195, 3090000, 3133157, 2595240],
        [952280, 9452690, 353502, 1040361, 676374, 0, 6284735,
           3071880, 2880012, 1044384, 3822, 725481],
        [5102650, 1190098, 7665037, 3361974, 2956926, 6284735, 0,
           1830584, 876852, 1761403, 6005760, 3655740],
        [8344441, 2619565, 5274984, 4508102, 3198860, 3071880,
           1830584, 0, 1823796, 2025494, 2033355, 2541175],
        [5905380, 933543, 3987529, 3236922, 1746195, 2880012,
           876852, 1823796, 0, 1068900, 1999800, 4120360],
        [2756430, 3467124, 3068329, 2984472, 3090000, 1044384,
           1761403, 2025494, 1068900, 0, 1595361, 2171939],
        [1848345, 4479221, 1549437, 2056370, 3133157, 3822,
           6005760, 2033355, 1999800, 1595361, 0, 1068144],
        [666699, 5720337, 1543024, 3369872, 2595240, 725481,
           3655740, 2541175, 4120360, 2171939, 1068144, 0]]

g.graph = [[0, 9910026, 1665288, 2797902, 4400084, 952280,
    5102650, 8344441, 5905380, 2756430],
        [9910026, 0, 9476480, 5109552, 2852493, 9452690, 1190098,
           2619565, 933543, 3467124],
        [1665288, 9476480, 0, 2758078, 3257064, 353502, 7665037,
           5274984, 3987529, 3068329],
        [2797902, 5109552, 2758078, 0, 1652498, 1040361, 3361974,
           4508102, 3236922, 2984472],
        [4400084, 2852493, 3257064, 1652498, 0, 676374, 2956926,
           3198860, 1746195, 3090000],
        [952280, 9452690, 353502, 1040361, 676374, 0, 6284735,
           3071880, 2880012, 1044384],
        [5102650, 1190098, 7665037, 3361974, 2956926, 6284735, 0,
           1830584, 876852, 1761403],
        [8344441, 2619565, 5274984, 4508102, 3198860, 3071880,
           1830584, 0, 1823796, 2025494],
        [5905380, 933543, 3987529, 3236922, 1746195, 2880012,
           876852, 1823796, 0, 1068900],
        [2756430, 3467124, 3068329, 2984472, 3090000, 1044384,
           1761403, 2025494, 1068900, 0]]

#find all the possible
def find_edge_lists():
```

```
  result = []
  for i in range(len(g.graph)):
    rowlist = g.graph[i]
    for j in range(len(rowlist)):
      if i <= j:
        continue
      else:
        result.append([i,j])
  return result

def find_teleporter_sets(edge_list):
  all_possible_sets = []
  comb = combinations(list(range(45)), 4)
  for (i,j,k,l) in list(comb):
    all_possible_sets.append([edge_list[i], edge_list[j],
        edge_list[k], edge_list[l]])
  return all_possible_sets

#graph after putting teleporter (edge weights become 0)
def put_teleporters_one_case(teleporter_set):
  new_g = Graph(10)
  new_g.graph = copy.deepcopy(g.graph)

  for teleporter in teleporter_set:
    [a, b] = teleporter
    new_g.graph[a][b] = 1
    new_g.graph[b][a] = 1

  return new_g

def compute_total_traffic(graph):
  total_traffic = 0
  for node in range(len(graph.graph)):
    shortest_total_traffic_for_cur_node = graph.dijkstra(node)

    total_traffic += shortest_total_traffic_for_cur_node
  return total_traffic

def pick_teleporter_location(graph):
  edge_list = find_edge_lists()
```

```
  all_teleporter_sets = find_teleporter_sets(edge_list)

  curr_min_total_dist = -1
  curr_best_teleporter_set = []
  best_graph = Graph(10)

  for teleporter_set in all_teleporter_sets:
    new_graph = put_teleporters_one_case(teleporter_set)

    total_dist = compute_total_traffic(new_graph)

    if curr_min_total_dist == -1:
      curr_min_total_dist = total_dist
      curr_best_teleporter_set = teleporter_set
      best_graph = new_graph

    elif total_dist < curr_min_total_dist:
      curr_min_total_dist = total_dist
      curr_best_teleporter_set = teleporter_set
      best_graph = new_graph
  return (curr_best_teleporter_set, curr_min_total_dist, best_graph)

curr_best_teleporter_set, curr_min_total_dist, best_graph =
    pick_teleporter_location(g.graph)
print(curr_min_total_dist)
print(curr_best_teleporter_set)
print(best_graph.graph)
```

5. Result

We determined that adding a teleporter from New York to 4 other locations resulted in a graph with the least traffic. The following were the airports that we added a teleporter to from New York:

1. Dallas-Fort Worth
2. Seattle
3. Las Vegas
4. Miami

	3 Teleporters	4 Teleporters	5 Teleporters
10 Cities	Dallas Fort-Worth Seattle Las Vegas	Dallas Fort-Worth Seattle Las Vegas Miami	Dallas Fort-Worth Seattle Las Vegas Miami Atlanta
12 Cities	Dallas Fort-Worth Seattle Las Vegas	Dallas Fort-Worth Seattle Las Vegas Miami	Dallas Fort-Worth Seattle Las Vegas Miami Atlanta

Table 1: Results from running the program on different numbers of teleporters and cities

We also ran our program on 3, 4, and 5 teleporters and achieved similar results with New York being the airport hub. Additionally, we ran the program on 12 cities, adding Newark, NJ, and Charlotte, NC to our previous 10 cities. We achieved the same exact results that we did with 10 cities.

6. Conclusion

The results from our algorithm showed us that small changes in the number of teleporters or cities in the problem don't have a material effect on the structure of the results. Namely, our algorithm prioritized New York City in every scenario we tested, creating a star-shaped pattern of different cities joining to New York City in order to link cities near and far. This had far-reaching effects, as when Seattle and Miami were both linked to New York, people could travel from Seattle to Miami (and vice-versa) in a much-reduced amount of time (in addition to being able to travel to New York much faster). This star pattern appears to be the best way to connect cities given the airline data and limitations of the problem, and its results in reducing passenger miles in our model can be seen below:

New passenger miles based on 10 city, 4 teleporter solution

	ATL	LAX	ORD	DFW	DEN	JFK	SFO	SEA	LAS	MCO
ATL	0	3276126	1665288	1554390	3848234	507465	1957410	1552365	1372950	2756430
LAX	3276126	0	5304000	982128	2852493	906999	1190098	650091	933543	371142
ORD	1665288	5304000	0	2537982	3257064	353502	4791168	2268612	1947582	2248686
DFW	1554390	982128	2537982	0	1616406	0	951372	0	0	0
DEN	3848234	2852493	3257064	1616406	0	261459	2956926	1962510	1746195	1254000
JFK	507465	906999	353502	0	261459	0	1008090	0	0	0
SFO	1957410	1190098	4791168	951372	2956926	1008090	0	1116144	876852	298494
SEA	1552365	650091	2268612	0	1962510	0	1116144	0	0	0
LAS	1372950	933543	1947582	0	1746195	0	876852	0	0	0
MCO	2756430	371142	2248686	0	1254000	0	298494	0	0	0
	Previous Total Passenger Miles				Total Passenger Miles with 4 Teleporters					
	414607824					126268392				
			Reduction in Passenger Miles							
				69.55%						

This table was calculated by iterating through each entry in the following way: evaluating if there was any way to use any teleporter route to reduce the distance between two cities, then making the change in distance if possible or keeping the old distance if there was no teleporter route that reduced distance between cities. Since the table is symmetric, this only required 45 iterations that got progressively easier as an understanding of the new graph was developed, but for further applications of our project a new algorithm would be required to evaluate its effectiveness in the manner we have below. As discussed, many of the entries in this table are zero- in fact, while 4

teleporters guarantee that a minimum of 4 routes will have zero passenger miles, we achieved 10 routes with zero passenger miles using our shortest path algorithm. Overall, the reduction in passenger miles was almost seventy percent, as the edges our algorithm removed were well-trafficked and their endpoints were central enough to be used to lower other travel distances.

A great question to be asked is what real-world structure our star-shaped pattern is mimicking- surely, transportation analysts have conducted similar analysis when new means of transport emerged to make the most of their implementation. We found that our teleporter placement actually resembles the hub system used in the airline industry- except that instead of needing many large hubs to cover the United States on a geographical basis, we only need one central hub anywhere in the United States (NYC in this case) to drastically reduced airline transportation requirements.

Finally, what are the applications of this project? Well, aside from teleporters actually be invented in the near future, this could be used to approximate any new method of transportation that is sufficiently faster than those currently in use. While our assumption of a teleporter instantly transporting a person between two locations regardless of distance is unrealistic, a relaxation parameter could be applied to distance graphs instead, and it is likely that the results would be similar to ours if the technology being introduced was sufficiently advanced in comparison to the airline industry. If passenger miles were able to be reduced on a large scale, and their replacement in the form of new transportation technology was environmentally friendly, then the application of these results could go a long way towards helping stave off climate change. According to a study by Ulrike Burkhardt and Lisa Bock at the Institute of Atmospheric Physics in Germany, aviation is responsible for about five percent of harmful emissions currently contributing to global warming, and while air travel is currently considered necessary to a global society, finding a way to phase it out could benefit everyone long-term.

However, in order to apply our algorithms to larger data sets, there must be considerations of runtime and an understanding of the resulting limits inherent in our computational methods.

7. Runtime

7.1. Runtime Analysis

Here is the runtime analysis for our algorithm. We have a complete graph with n vertices. There are $\binom{n}{2}$ edges in the graph. Let the number of

teleporter systems be k. Since each teleporter system eliminates one edge in the graph, there are a total of $\binom{\binom{n}{2}}{k}$ possibilities. For each possibility, we run Dijkstra's Algorithm (with complexity $O(n^2)$) on each vertex to compute the shortest path problem for that vertex and sum the weights to get the total traffic. Then we choose the minimum total traffic solution from all the possibilities. Therefore, the total runtime complexity is $O(\binom{\binom{n}{2}}{k} \times n \times n^2) = O(n^3 \binom{n^2}{k})$.

7.2. Runtime Improvement

For each possible set of teleporters ($\binom{\binom{n}{2}}{k}$ in total), we compute the shortest path problem for each vertex. However, there are repetitive calculations involved. For example, Dijkstra's Algorithm will calculate the shortest path between Node A and every other node in the graph, if Node A is the source node. However, when we run the algorithm with Node B as the source, we will calculate the shortest path between Nodes A and B again, thus doing unnecessary computations. We could resolve this issue by creating an empty adjacency matrix that stores the values returned by Dijkstra's Algorithm for each node pair. By doing this, we could check if the shortest path algorithm has already been run between two nodes. If it has already been run, then we can skip the current iteration of Dijkstra's Algorithm, and if it has not been run, we can run the algorithm and store the result in the adjacency matrix. This would therefore reduce runtime significantly.

7.3. Further Considerations

Consider the smallest weight in a graph. It would be highly unlikely to replace that edge with a teleporter system. In our example, the edge weight between JFK and Newark is 3822, which is significantly smaller than any of the other weights. With a small number of teleporters, we can safely ignore the edges with the smallest weights when finding the combinations of teleporter sets. However, for a larger number of teleporters, we need to consider the fact that certain edges, such as those with small weights, could connect major hubs together. As we eliminate the edges coming out of these major hubs by adding teleporters, we would significantly increase the weight of the edge connecting the hubs. This weight could become large enough that we would need to add a teleporter.