Group Maker

Bhagyesh Patel and Zain Ul-Abdin

May 2020

1 Introduction

In recent times, we have seen a trend in social media. During quarantine, people have been trying to host massive Zoom or other video platform calls. From what we have seen, over 500 people join these calls. There is no way anyone can hear each other with everyone talking at once. Also, it is not optimal to talk to someone over 500 miles away, as it is not likely that they will physically meet up. So, our solution is to make a grouping of people based off the location and age. Ideally, users can meet people that are close to their age range and location.

Figure 1: How some feel during quarantine

2 What data was needed and how its handled

There are 4 main data components: age, location, gender, and snap username. First, the main factor that the groups are based off of is location. This is so that users are not talking to someone a great distance away. We use zip codes in order to achieve this goal, as it is less specific than an exact address while still allowing for a specific enough location to be used. Also, it is more general and easier to compute distances with uniform data. Next, we take age into account. We don't want someone who is 25 years old talking to someone that is 14, so we filter by age ranges of 14-17,18-20, and 21-25. The first group is those in high school, ages 14-17. We didn't include 18 as they will be adults or go to college, so they will not be in high school for long. Then, there are young adults with the cutoff age at 20 so people can hang out and not be isolated or pressured if some 21 year old users go to bars with the rest of the group and the underage user is left alone. Lastly, we have the people as adults with responsibilities, the 21-25 age range. They are older and have responsibilities, but they are still looking for fun. Moving on, we now have gender. Gender data allows us to diversify, as it is optimal to have different genders meeting and conversing. Lastly we have the usernames to add people to the group chats.

3 How does the code work

3.1 The set up

Users input the requested data, which is sent to a CSV file. We then have this data deserialized into a Plain Old Java Object (POJO). This is just a basic data model. Next, we have the zip code data file deserialized, which is also sent to a CSV file. It looks like this:

```
ZipCode|    City      |State|Latitude |Longitude |Classification|Pop.
"00501"|"HOLTSVILLE"|"NY" |"40.8179"|"-73.0453"|       "U"      |"0"
```

Listing 1: ZipCode Data example

One caveat is that it is not clear what the "Classification" column means, so we will ignore it when looking at the data. Next we take the zip codes of the users and make a 2-D matrix, which looks like this:

```
        0         04734       04762       08619     11707
      04734         0           0           0         0
      04762         0           0           0         0
      08619         0           0           0         0
      11707         0           0           0         0
      12721         0           0           0         0
      13043         0           0           0         0
      15638         0           0           0         0
      22181         0           0           0         0
```

Listing 2: Zip Code Distance Table example

Next, in the inner section of the table, data is filled up with the distance between the 2 zip codes. This is done by calculating the distance via the longitude and the latitude of the zip codes given in the CSV file. Once the table is populated, we can refer back to this table when looking for the distance between any 2 zip codes. This allows us to save time when calculating distance, which is explained later in the next section.

3.2 The grouping algorithm

Once the distance data is set up, the next goal is to group people up based off of distance. This is done by a cluster algorithm known as k-Medoids[2]. The basic idea is that given a group of N nodes, choose k nodes to be medoids. In essence, for our purposes we pick random zip codes to be the center of groups. These k medoids will be center points in the graph. Then, calculate the distance between each individual zip code and the medoids. For example:

```
        null      65052      11933      16644      53961
       00694       2082       1598       1697       2205
       01001       1092         80        317        882
       01923       1181        148        408        959

Medoids:65052,11933,16644,53961
Zip Codes:00694,01001,01923
Distance between Medoids and Zip Codes: 65052,11933,...,959
```

Listing 3: Medoid Table example

Then, we do an analysis sweep over this Medoid-distance table and calculate the cost. We look at each zip code and find the medoid that it is closest to/has the smallest distance between. We then take that zip code, add that to that medoids groups, and then add that distance to the cost. So for the table above, the cost is 1598+80+148 = 1826. And our other groups are the following(Zip = {group members}), 65052 ={}, 11933 ={00694,01001,01923}, 16644 ={}, 53961 ={}. We then repeat this process, comparing the cost of the previous random groupings of the nodes. This is done up to a max limit. After running tests, the more iterations allow for lower cost but there are diminishing returns. There a lot of new costs between iterations 0-1000, but when in the range of 30,000-1,000,000 iterations, there may be 1-5 new lower costs. These newer costs are pretty significant, with at times having a 1000 or 2000 difference from earlier costs. Thus, it is not worth the time recalculating for a better group, as larger cost means further apart.

```
Iterations  Try 1         Try 2       Try 3          avg
1000000     116747         -            -          116747
10000       133472        126410     126216    128699.3333
1000        126592        136244     137783    133539.6667
100         132471        132298     149016    137928.3333
10          151625        164501     148414    154846.6667

Cost is inside of the values in the table.
```

Listing 4: Example of Diminishing Returns

For a simple explanation for this algorithm check out this video. Here is a visualization of 1 iteration of the algorithm:

3.3 Group based off of other criteria: age and gender

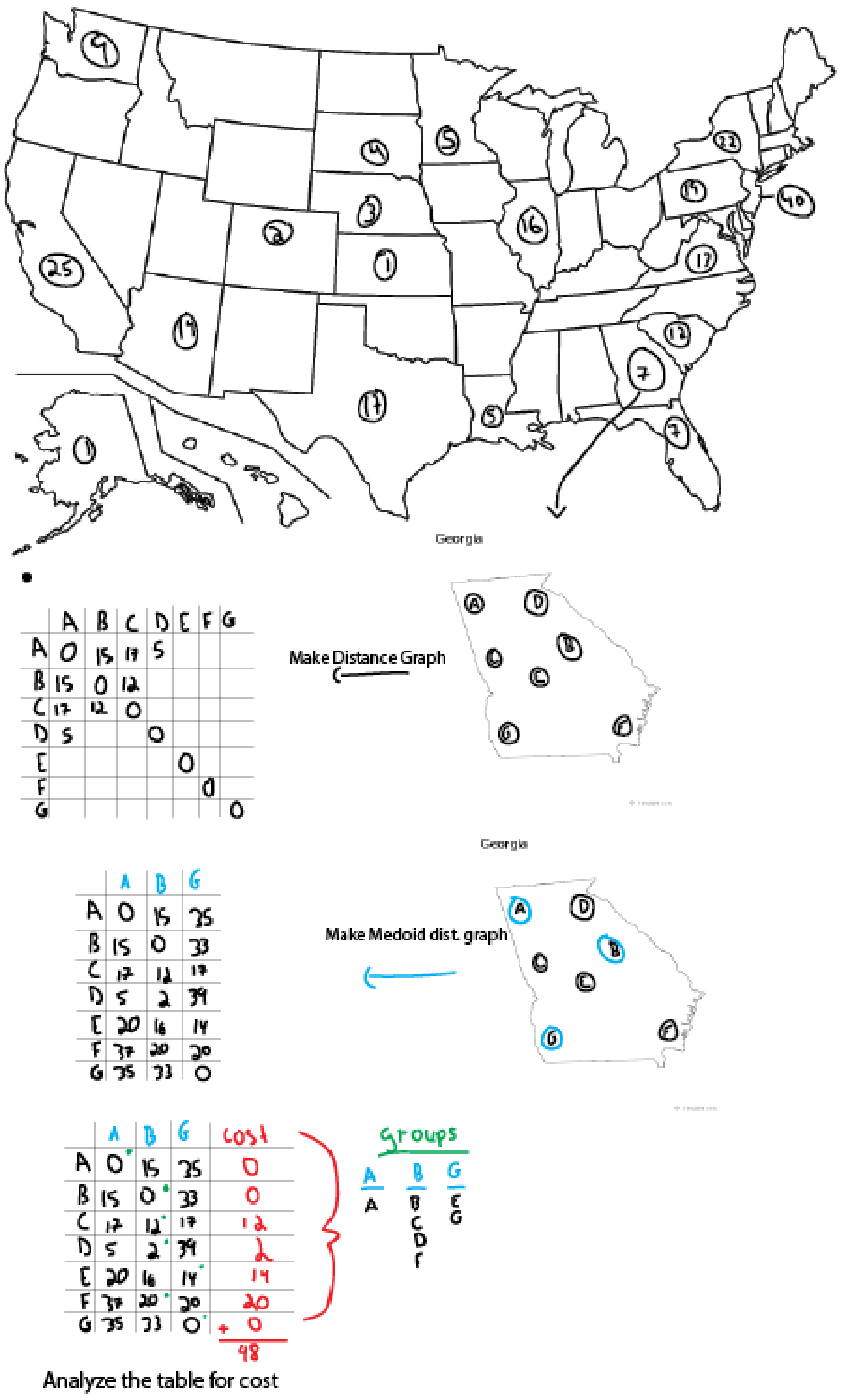
Once the groups have been determined based on location density, we can further separate these groups using any supplementary data we have, such as by age range. By looping through each entry in our resulting map (containing a map with a key pointing to list of people), we can easily use Lambda Expressions to filter a Stream of each list of people in order to separate them based on any criteria, provided the collection of the data.

4 Analysis of the Algorithm and efficiency

There were 2 main sections that were analyzed when it came to this algorithm. One was how the amount of groups affects the algorithm's speed. The other was how the amount of users affects the time to run the algorithm.

4.1 Amount of Groups: k value

We looked at how different group amounts have affected the cost and how it is affected when we increase the iteration count. For reference, k is used to represent the number of groups/medoids. Based off of testing multiple k values, we saw that if our k value is lower, than we have larger groups. In other words, we have a few groups that span a larger portion of the US. However, using a small value for k will increase the cost and will not be ideal for the users, as it spans a large distance. So if we have our k value higher, then we will have more groups that are more localized. Keep in mind that if we have a small k values, then we fit a larger amount of people in that group. Look at the image below:



	A	B	C	D	E	F	G
A	0	15	17	5			
B	15	0	12				
C	17	12	0				
D	5			0			
E					0		
F						0	
G							0

	A	B	G
A	0	15	35
B	15	0	33
C	17	12	17
D	5	2	39
E	20	16	14
F	37	20	20
G	35	33	0

	A	B	G	cost
A	0	15	35	0
B	15	0	33	0
C	17	12	17	12
D	5	2	39	2
E	20	16	14	14
F	37	20	20	20
G	35	33	0	+ 0
				48

A	B	G
A	B	E
	C	G
	D	
	F	

Figure 2: The k-Medoid applied to nodes in Georgia, 1 iteration

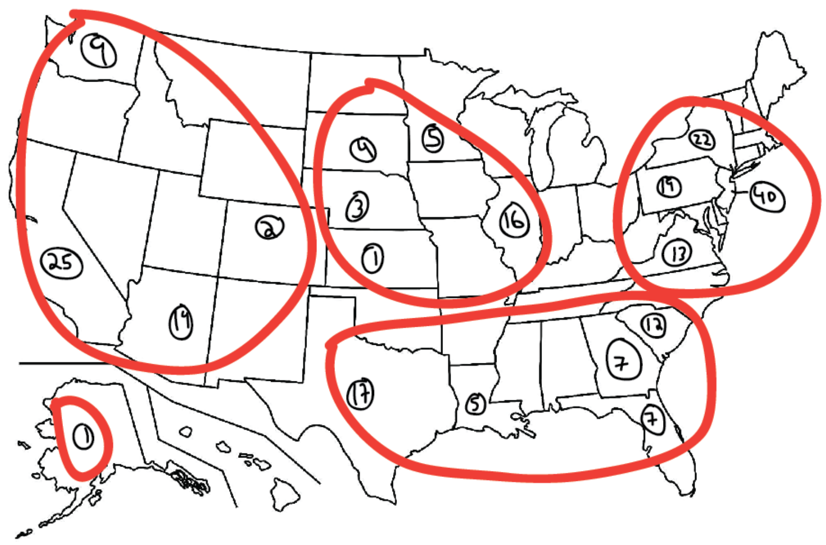


Figure 3: When the k value is small we have groups spanning larger areas.

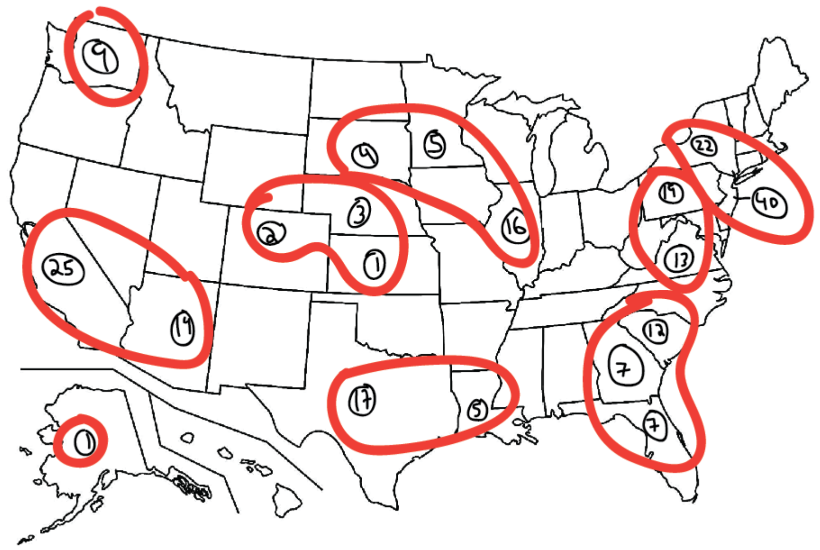


Figure 4: When the k value is larger we have groups spanning smaller areas.

As seen in figure 4, the clusters are more localized, meaning that users are more likely to be grouped with others in their local area. For the tests, we had 500 random individuals and iterated 10,000 times that would each have the following k values: 20, 30, 40, 50, 60, 100. In the following graph, the data shows that the increase in max amount of people per group also increases the cost, meaning a larger max amount of people per group also means smaller k values. Thus, smaller k values will inherently cost more.

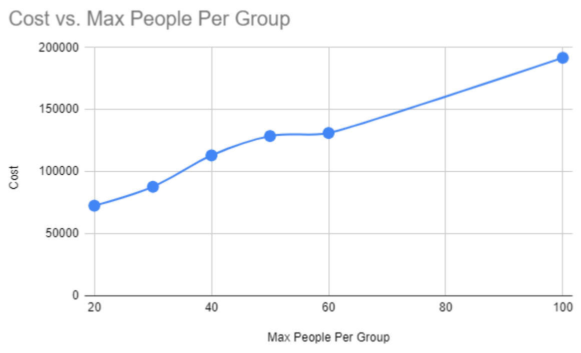


Figure 5: When the k value is larger we have groups spanning smaller areas.

We can use the max group size (L) to get k values. We set the value of k to be based on the amount of people needing to be clustered over L. If K*L <the amount of people, we increase k by one, making sure to round up if necessary.

```
            int l = 20;
            int k = (int) Math.ceil(lessThan.size() / l);
            if ((k * l < lessThan.size())) {
                k += 1;
            }
```

Listing 5: Calculate The k value Based off of L

4.2 The Big-O

We also tested for the big-O values. For those who do not know what big-O is, it is how computer scientists analyze how efficient their algorithms are. It is similar to limits as the input approaches infinity how long does the algorithm take to finish, the more constant the graph seems the better. To calculate the big-O values, we timed the execution of the algorithm from start to finish. We had the amount of people to cluster go from 100 to 3000. This was done with a max iteration of 100 and max group size of 20. We saw there was a exponential rise in the time it took to compute the data. The time on average increased at a rate of 1.2 times the previous amount, making this an O($n^{1.2}$). See the chart below.

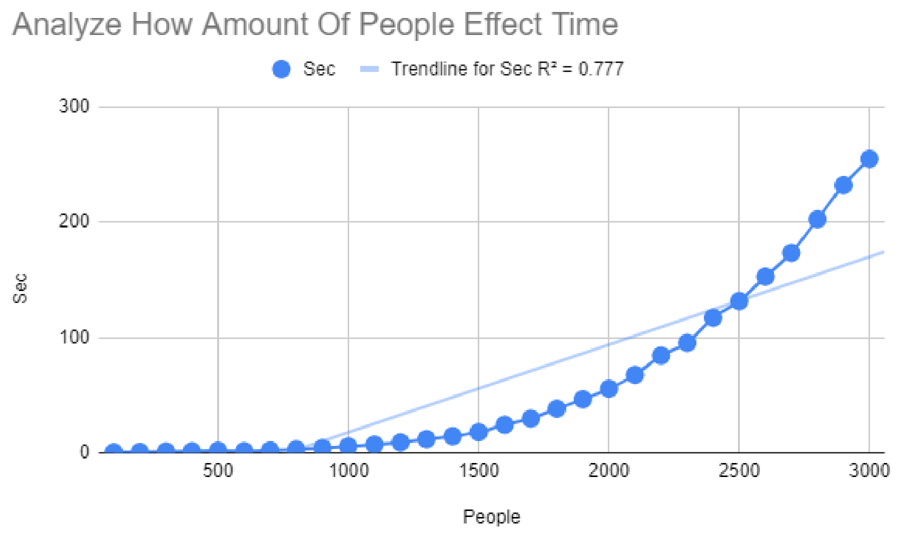


Figure 6: How the algorithm time changes based on N amount of people

5 Further Improvements

5.1 Data Types, Implemented as of (5/17/20)

This algorithm works as intended, though there is room for improvement. One of the major components to improve is how the table for distance and medoid-to-node distance is calculated. Firstly, the node distance table has the X and Y axis in integer format, while all other tables, notably the medoid to node distance table, has the X and Y axis in string format. The issue with this arises whenever we have to do a look up, as we have to cast the string to a int or vice versa. Since we have the node distance table in integer format, we have values such as the zip code "03755" show up as "3755" on the axis. This is due to the integer data type dropping the leading 0s. This issue is not shown in the figures above. Thus, there is constant casting, such as that seen in the example below, which shows code that picks the medoids randomly. [1]

```java
int random = rg.nextInt(distance[0].length);
// pad 0s infront
String temp = String.format("%05d", distance[0][random]);
// have no duplicate values/zips
if (!contains(medoids, temp)) {
```

Listing 6: Casting integer zip codes to strings

[1]Implementation is now active, see Listing 6

5.2 Distance Table fixing, Implemented as of (5/17/20)

Currently, after calculating the distance table, we then use it to look up distance given 2 zip codes. However, the issue is that the X and Y axis, which are labeled as zip codes, are not put there for O(1) look up, meaning when the algorithm looks for the distance between zip codes, such as "08863" and "07030", it scans the X axis for "08863" and the Y axis for"07030". This is because the first zip code is put in the 0th index, not the actual location on the axis. This situation means that looking up any distance could be O(n) + O(n). To combat this, we plan to make the table the data type of a Hash Map. In this Hash Map, we have the key be a zip code, and the value is another hash map where the 2nd key is another zip code and its value is the distance between the 2 zip codes. This has yet to be implemented, though soon will be. [2]

```
HashMap<String ,HashMap<String,Integer>> table = ...;
//  Key1     | Value 1
//  Zip Code| Key2       | Value 2
//           | ZipCode2  | distance between the 2 zip codes
// 85369     | 97358     | 936
// 85369     | 97355     | 926
// ...
```

Listing 7: What The Distance Hash Map Looks Like

5.2.1 Distance Table Fix Out Comes

Recently, there has been a update in the algorithm. As stated before, searches in the distance table were inefficient, as the index values did not correlate to that of the zip codes. Thus, this issue was fixed. Now, the table works as stated in Listing 7. Also, there was a bug fix where duplicated values were removed from groupings. There was another bug where people were not added to groupings. Nonetheless, these issues were fixed. In regards to how these fixes have affected the algorithm, the algorithm is now significantly faster. For example, 3,000 users over 100 iterations used to take 254.8 seconds(roughly 4 minutes and 15 sec). Now, with the new data table structure, the same 3,000 users over 100 iterations takes 27.6 seconds. On average, the improvement was about 70% (for 100 $\leq$ amount of users (N) $\leq$ 3000). This percentage is then amplified as the N-value increase, so with values like 100,000 users, it is estimated with the new data structure implementation of this algorithm the time taken had a 93.2927% decrease (see here for a rough estimate of a best fit graph). However, the big-O is still O($n^{1.2}$). This improvement can be seen in the below graph (iteration set to be 100).

[2]Implementation is now active, see Figure 7 for time comparison

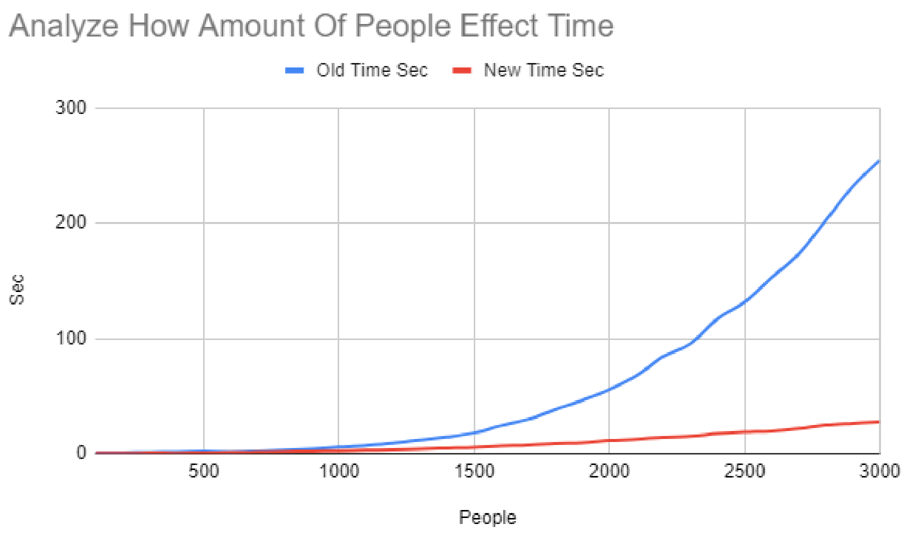


Figure 7: Amount of people vs Time improvement

5.3 Sub-set, Implemented as of (5/29/20)

Another improvement would be having this algorithm move from the current state, which is based off a global set, to a CLARA like implementation, which means having the data broken into subsets and then having the algorithm applied on those subsets[1]. There are two options for the subsets. The first is to break up the data based off of regions. This may not be optimal, as users could live near the borders of the regions, meaning they could be grouped with other individuals that are further away. The other option is to have the medoids picked based off of the density of where the users zip codes lay. For instance, if there are more NYC users, we pick a medoid there rather than having the algorithm pick it randomly. This would give us density optimization for the algorithm and would ideally decrease the amount of iterations needed, as the medoids are picked from a mathematical calculation rather than have it be completely random. We can further improve it by averaging the X and Y coordinates, which provides a rough estimate of a center, and then find the closest node to that center and make that the medoid[3]. However, it is not clear how this would affect the idea of having the ability to have k groups. Would the algorithm no longer need a k value as it has the subsets already set and the size of those subsets would now equal k? Would it be possible to have the k value be larger than the subset size for certain subsets only? It is unclear, and future testing needs to be performed.

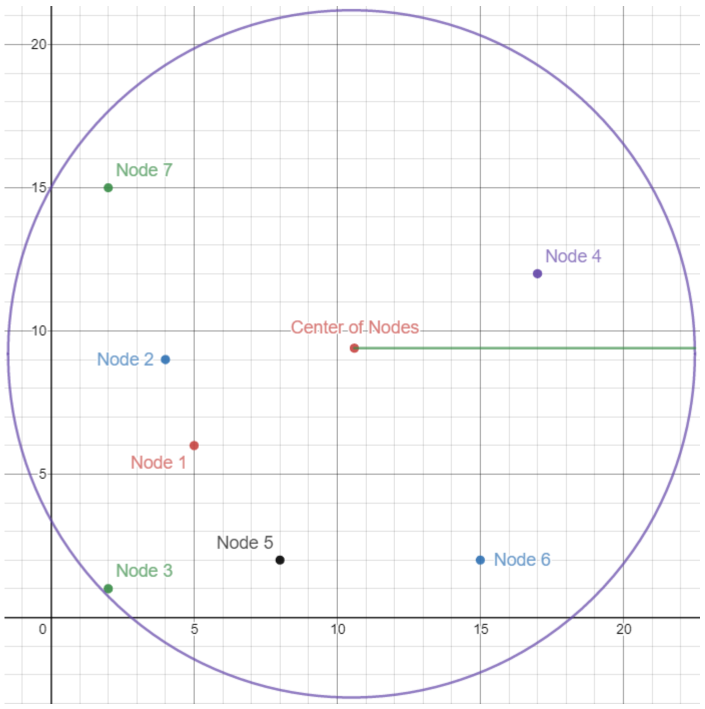


Figure 8: How we can better pick medoids in sub-sets of data

5.3.1 Defining Subset

After all these improvements, still one question remained, which concerned how we wanted to have the data be separated into subsets. It was decided that the subsets were to be based off of density of populations. We had used data collected from the 2006-2010 American Community Survey of the US Census Bureau, which was visualized by Ian Offord and Université de Strasbourg[4]. This can be seen in figure 9, below. There is still one issue, which is how do we check all users and put them in their according subset? There are 12 groups and N-amount of users. Well, we do not want to check all 12 groups, so it was decided that we would check in the order of most dense to lease dense in the hopes that we do not have to go through all 12 grouping checks, only in the worst case will the algorithm do so. This was inspired based off of Huffman trees and how popular characters have shorter paths and less popular characters have longer paths.

5.3.2 Analysis on Subset Implementation

After the algorithm was adjusted for subsets there were analyses done on different groups of people. It was concluded that subsets are only beneficial when there are a large amount of people. The rough estimate is if there are less than

7,000 people we should **not** use subset. If the amount of people is between 7,001 and 12,000 its **experimental**, meaning both algorithms should be tried. And if there are over 12,001 users its **best** to use subsets.

--

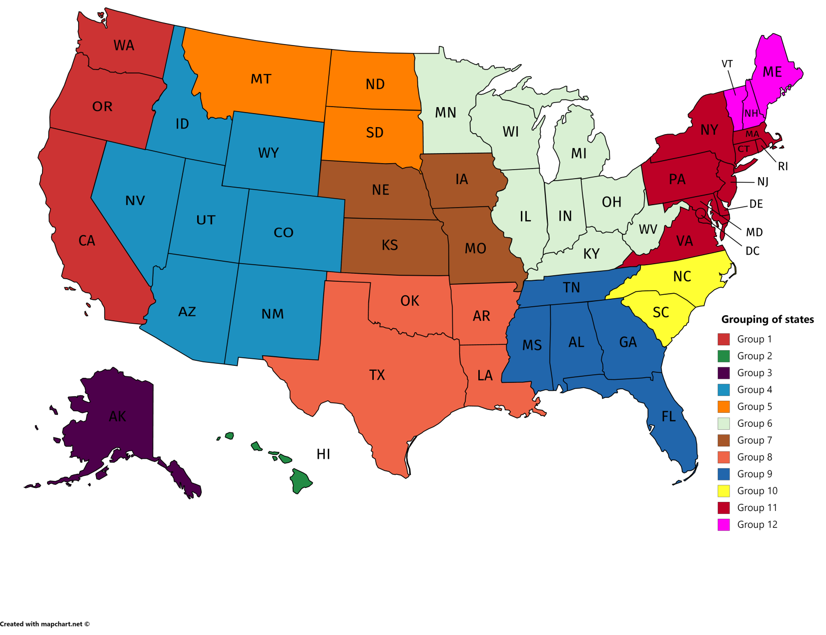


Figure 9: Grouping of all states based off of density

6 Conclusion

This algorithm has the potential to connect many people together. As seen previously, the algorithm has been improved upon and will still be in the future. This algorithm can be applied to other problems as well, such as finding the nearest gas stations, stores, or other points of interest. With the idea of sub setting data, we believe that this will improve greatly the accuracy of finding people in your area as it will increase the efficiency in the grouping of the data. Link to all data collected on the algorithm

7 About the Contributors

Bhagyesh Patel was the main developer and author of this paper. He goes to Stevens Institute of Tech. and will graduate in 2022 with a Master in CS.
Zain Ul-Abdin was the main researcher and assisted as a developer. He goes to Rutgers University and will graduate in 2021 with a Bachelors in CS.
Tanner Chiamprasert was the main editor of this paper. He goes to Stevens Institute of Tech. and will graduate in 2022 with a Bachelors in Civil Engineering and minor in CS.

References

[1] Yash Dagli. Partitional clustering using clarans method with python example.

[2] Gabriel Gama. K-medoids in r: Algorithm and practical examples.

[3] Murray H. Protter. *College Calculus With Analytic Geometry*. Addison-Wesley, Reading, Massachusetts, 1977.

[4] Université de Strasbourg. ”population density, 2010”, 2020. Data retrieved from US Census Bureau, http://ecpmlangues.u-strasbg.fr/civilization/geography/US-census-maps-demographics.html.