K-means clustering (Facility location problem)

Arash Ziaee

January 18, 2022

Abstract

In this report, the solution of facility location problem using K-means clustering approach is described. As an example, 6800 cars locations on the street are considered which the rescuers should carry them to rescuer bases to be fixed. The problem is to find the best places for building rescuer bases. A primary solution is finding some mean locations of the cars to form definite number of clusters. In the present work, one extra parameter is considered. Every car has a weight that is used to find the optimal place for the rescuer base location. This is the parameter that changes the simple K-means clustering to facility location problem.

Introduction

Consider a definite number of the cars on the streets which should be carried by the rescuers to the rescuer bases to fix them. Solving a facility location problem means that finding the best places for building the rescuer bases when the car weights and their locations are known. It is trivial that car places are not constant but the history of previous car services are only accessible data.
In the simple K-means clustering when we are trying to find a new position for clusters positions we just calculate the mean value for cluster members but in this case to consider the weight of every member using the center of mass is a good choice. To solve this problem, 6800 car locations on the street were used. To find the best places for building rescuer bases, the weights of the cars, like distances, were considered, as the costs of carrying the cars with different weights are not equal. Solving this problem minimizes the general cost. So, the problem is a variant of K-means clustering that is explained in the next part of the report. This report includes the following topics.

1. **Methodology:** In this section the mathematical way of center of mass calculating and the algorithm of K-means clustering are explained.
2. **Implementation:** In implementation section the code is described. The explanation contains the graphical interface implementation and a general documentation of K-means calculator for both sequential and parallel parts.
3. **Results:** In this part some pictures of the application interface are given that contains related logs to show the run time and number of cycles. This determines the final clusters centers location.
4. **Benchmarks:** In this section the effect of changing number of data and number of clusters on run time are considered. The results are given by graphs and tables for both sequential and parallel mode.
5. **Conclusion:** Conclusion is the final part of the report. in this section the results are discussed.

1 Methodology

In the K-means approach the number of clusters is predefined and clustering some data has two steps that should be repeated to reach the stop conditions.The stop conditions reaches when no other change observes by repeating the algorithm.

1. In the first step, a specific number of clusters must be considered in order to their centers location be calculated. For the fist cycle there is no cluster, so it seems that must be initialized randomly, but there are some conditions that if be applied, reduces the number of cycles up to the stop conditions. Toward this end, some of the initial data choose as the clusters centers. Although it's not necessary, it helps to find the clusters centers in the area of data set. After the first step, the centers location should be calculated. In every cycle, the centers of the clusters must be updated. In the simple K-means, simple mean uses to calculate centers location. for example if we have a set of 2D data in a cluster with positions $P_(x_i, y_i)$ that x_i, y_i are components of the i-th location in 2D space we can calculate the center of cluster by:

$$x_c = \frac{1}{N}\sum_{i=0}^{N} x_i, y_c = \frac{1}{N}\sum_{i=0}^{N} y_i. \tag{1}$$

   Here, N determines the number of data in the cluster and x_c, y_c are center location components. In this problem, the simple mean of the positions replaces by the center of mass. If the weight of every position were given by m_i, the components of center of mass in a 2D space are:

$$x_c = \frac{1}{M}\sum_{i=0}^{N} m_i x_i, y_c = \frac{1}{M}\sum_{i=0}^{N} m_i y_i, \tag{2}$$

   in which the M is sum of the mass of all clusters data. $M = \sum_{i=0}^{N} m_i$

2. In the second step, every data should be assigned to it's nearest cluster center. Although the goal is minimizing the cost as a function of distance and weight, considering the weight in this step is not useful as to assign the data to one of clusters we should find the cost that is $m_i d_{i,j}$ (which $d_{i,j}$ is the distance between position i and center of cluster j) and to find the minimum cost we should check condition $m_i d_{i,j} > m_i d_{i,k}$ and we know that there is no difference if we check this condition instead: $d_{i,j} > d_{i,k}$. As a matter of fact, m_i will be omitted from the inequality .So in this step of calculations simple K-means may be adopted.

   There is another point about the K-means algorithm. Actually there are two stop conditions in this implementation. If we reach a situation that none of data in two consecutive cycle changes their cluster, then we reached the best situation for the centers of the clusters. But because there is nothing to guarantee that this situation will be reached in a finite number of cycles we should consider a limit for the number of cycles. This limit is the second stop condition.

2 Implementation

2.1 Introduce graphical interface

The Java swing adopted to develop a graphical interface for this application. I also used Maven as package manager, jxmapviewer as map provider in the UI and JMathPlot to plot the graphs. The calculational parts of the application runed on separated threads, so the graphical interface doesn't halt while the calculation parts are running. The graphical interface includes:

- A white area to show the log of the application.
- A radio button to switch between sequential and parallel mode.
- Two text boxes to enter number of clusters and number of data.
- A button to clear the log area.
- A button to create a graphical benchmark report with constant number of clusters.
- A button to create a graphical benchmark report with constant number of data.
- A button to start calculation based on the user input.
- A map to show the data and clusters.

The graphical interface can be resized manually. The map supports zoom (by scrolling) and move (by mouse) and initially shows the area of Tehran. The map is generated jxmapviewer2 liberary which shows online open street map and needs internet to show the map. The log contains number of cycles and time of running in millisecond.

2.2 K-means calculator

The code contains a K-means class that provide the calculation of clustering. Every object of this class keeps the state of the data and clusters information. This class has a public function that is named *run*(). This function returns a *KMeansReport* object that contains cycle counts and running time and follows the algorithm:

```
initData();
initClusters();
repeatCount=0;
while(true) {
    bindToClusters();
    updateClustersCenters();
    repeatCount++;
    if (repeatCount > 1000) {
        break;
    } else {
        if (!bindToClusters()) {
            break;
        }
    }
}
```

As mentioned in the methodology section, there are two conditions which can break the while loop. I should explain that the *bindToClusters*() function returns a boolean that shows if in binding process none of data points change their clusters. So after the first call of *bindToClasters*() the boolean checks that the function returns and in the case of false, the while loop breaks.

2.3 Run mode

The algorithm of calculation is the same in sequential and parallel mode. But to implement parallel mode I changed the implementation of the *bindToClusters*() and *updaeClustersCenters*() functions.

2.3.1 Sequential

***bindToClusters*() :** In the sequential mode to bind each data to the nearest cluster center I used two nested for-loops. The first for-loop iterates over data list and the inner for-loop iterates over clusters. Finally the nearest cluster centers to the data will be found and then every data will be assigned to it's nearest cluster.
***updateClustersCenters*() :** Every cluster object has a void $updateCenter()$ function that calculates it's center of mass and replaces it with its old one. So in the $updateClustersCenters()$ it needs to simply iterate over clusters and call their $updateCenter()$.

2.3.2 Parallel

***bindToClusters*() :** To implement parallel mode I decided separate data list to number of threads sub-lists and give each sub-list to one of threads to bind it's data to the nearest cluster. This is my code to do this separation:

```
for(int i=0; i¡ numberOfThreads ; i++)
    int subListStart = dataList.size() / numberOfThreads * i;
    int subListEnd = dataList.size() / numberOfThreads * (i+1);
    bindToClustersRunnables.add(
        new BindToClustersRunnable(dataList, clusters, subListStart, subListEnd, latch)
    );
    executor.execute(bindToClustersRunnables.get(i));
```

When $i = 0$ we know that $subListStart = 0$ and when $i = numberOfThreads-1$, $subListEnd = dataList.size()$. In this way I can cover all $dataList$.

Here, there is a problem: Each cluster has a list field to keep it's assigned data. When some threads are assigning data to these clusters, we should be careful about conflicts. So I added a synchronized function to cluster class to add a new data to it's data list. This is the way that I can be sure that there is no problem with multi threading.
***updateClustersCenters*() :** To update clusters centers we can easily assign each cluster to one thread to run it's $updateCenter()$ function. I have a $UpdateCentersRunnable$ class that it's constructor is $UpdateCentersRunnable(Clustercluster, CountDownLatchlatch)$. It gets a cluster and a CountDownLatch to manage main thread to wait for thread pool finishing their tasks.
I also used the CountDownLatch to manage main thread to wait for other threads in $bindToClusters()$ function.

2.4 Data generation

There is one file in the resource of the project that contains about 6800 locations of the cars in the Tehran. When user enter number of data I randomly chose some data (number that user asked) and add a random tolerance to them. This algorithm guarantees that the generated data set remain in the area of Tehran and it's not necessary to care if the number of data that user entered is grater than the length of the initial data set or not.

2.5 Report generator

To capture run time there is a class in the application. It's name is $ReportManagment$. There are two functions in this class $reportWithConstClusterNumber()\&reportWithConstDataNumber()$ which run K-means algorithm with different number of data and clusters. They run each configuration 3 times and report the mean result. These functions return a JPanel which includes

a plot of the result.

3 Result

This section just presents the application and contains some screenshots of the application. After clustering, the map shows the locations and determine their clusters by colors. The bigger circles are the centers of the clusters and size of each data point shows it's weight. The running time of clustering and the number of cycles to finish the clustering is shown in the log part.

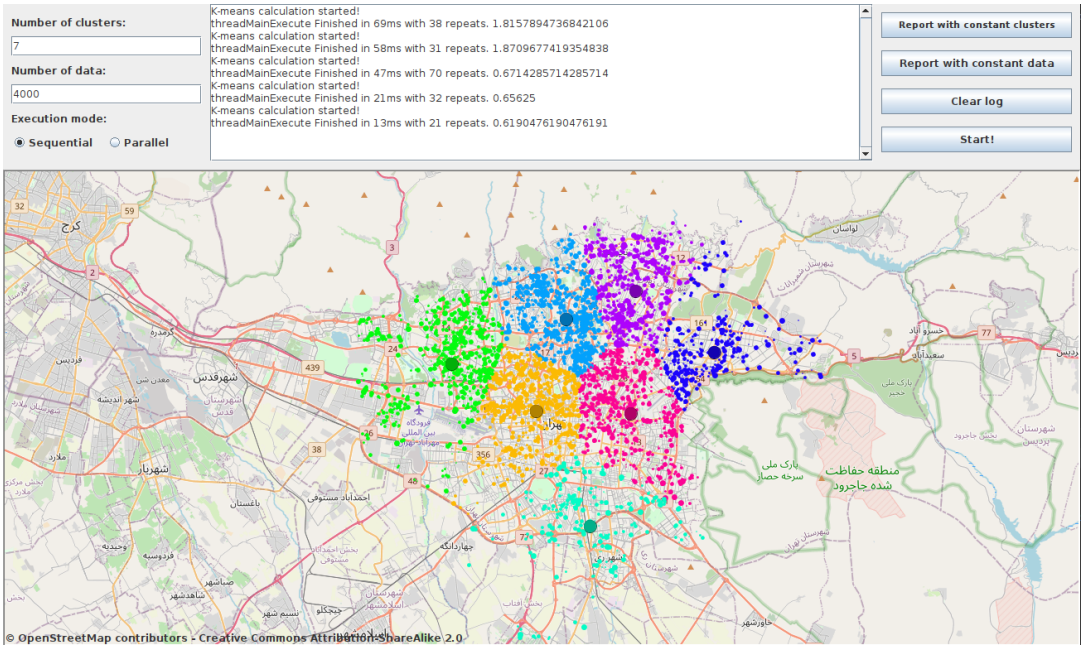

Figure 1: This is a screenshot of the graphical interface. Clustering

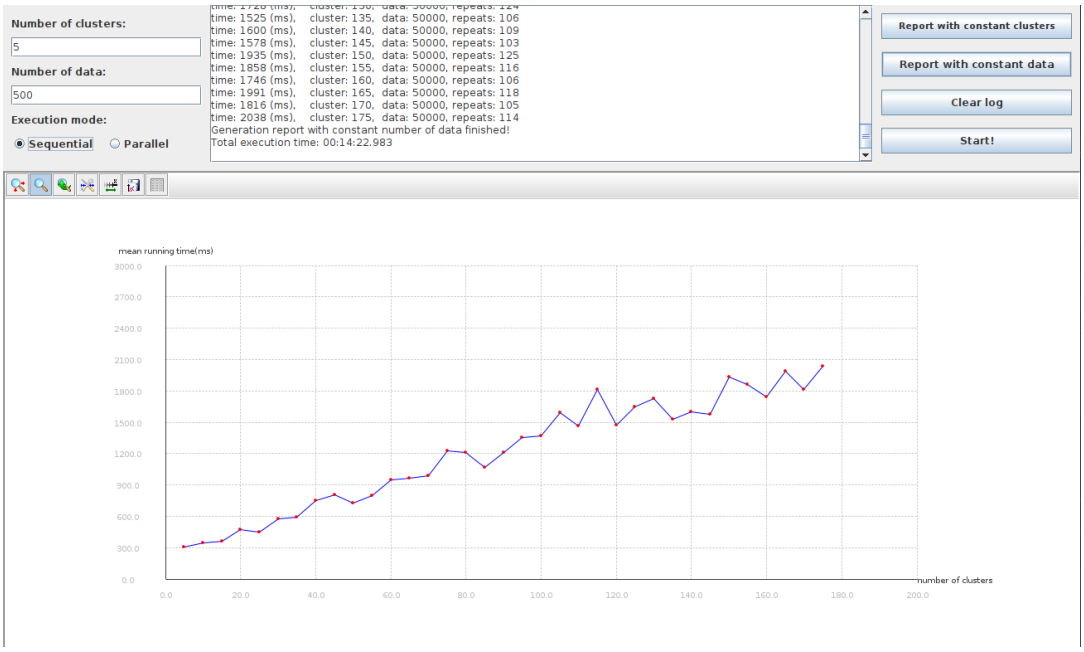

Figure 2: This is a screenshot of the graphical interface. Log of report with constant number of data.

4 Benchmarks

This section contains the result of tests in graph and table format. Each configuration is repeated 3 times and considered average run-time as final result. Each test includes 20 different configurations. All tests were performed on a machine with 16Gb RAM and 8-core (16treads) processor Intel Core i7-10870H. Parallel mode were ran with a thread-pool with 8 threads.

4.1 Test with constant number of clusters

4.1.1 Sequential

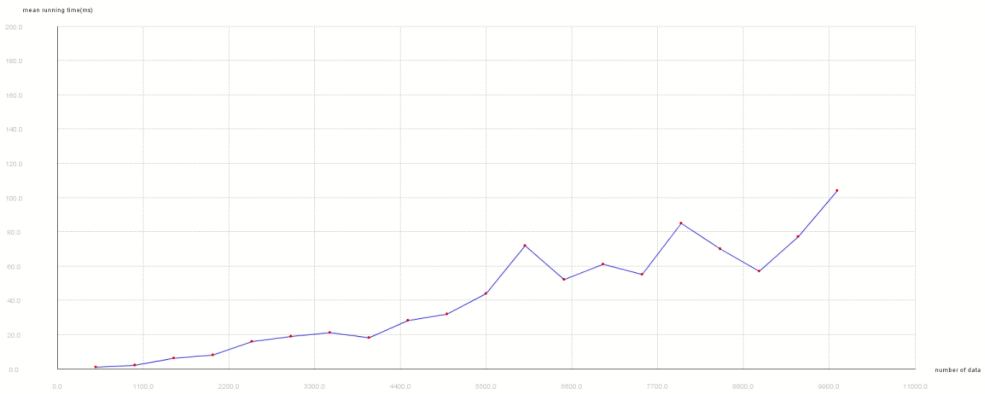

Figure 3: Time in millisecond verses number of cars with 20 clusters

Number of data	Time in millisecond
500	1
1000	2
1500	6
2000	8
2500	16
3000	19
3500	21
4000	18
4500	28
5000	32
5500	44
6000	72
6500	52
7000	61
7500	55
8000	85
8500	70
9000	57
9500	77
10000	104

4.1.2 Parallel

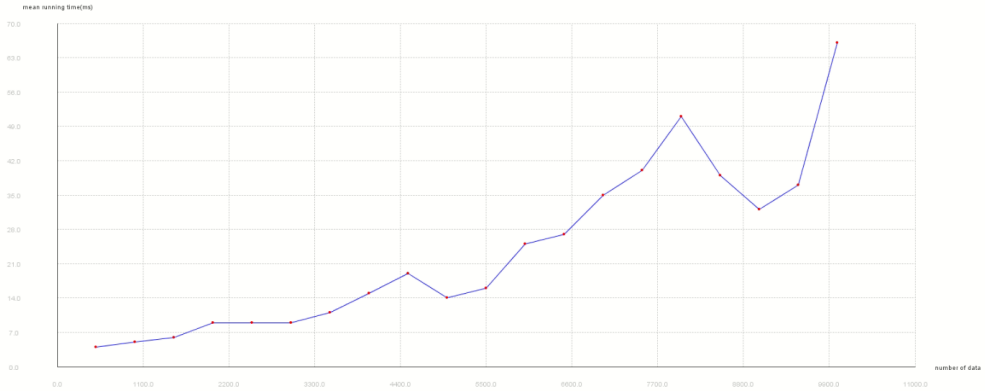

Figure 4: Time in millisecond verses number of cars with 20 clusters

Number of data	Time in millisecond
500	4
1000	5
1500	6
2000	9
2500	9
3000	9
3500	11
4000	15
4500	19
5000	14
5500	16
6000	25
6500	27
7000	35
7500	40
8000	51
8500	39
9000	32
9500	37
10000	66

4.2 Test with constant number of data

4.2.1 Sequential

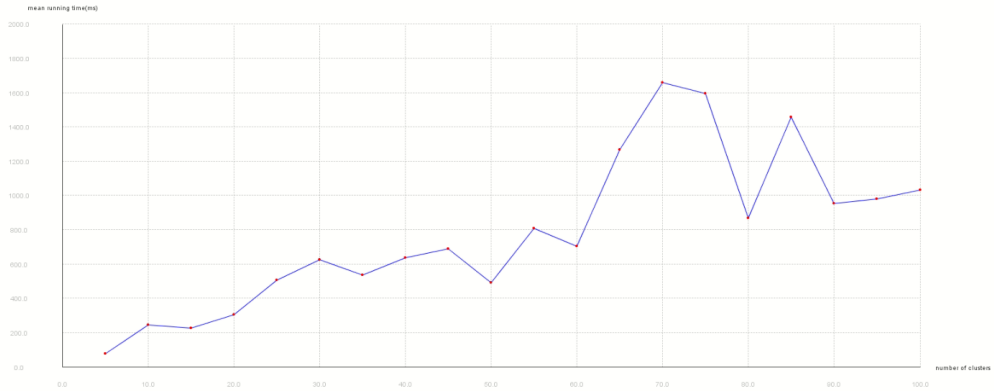

Figure 5: Time in millisecond verses number of clusters with 30000 data

Number of clusters	Time in millisecond
5	75
10	245
15	227
20	304
25	504
30	625
35	535
40	638
45	690
50	491
55	808
60	705
65	1266
70	1660
75	1597
80	868
85	1456
90	954
95	978
100	1033

4.2.2 Parallel

Figure 6: Time in millisecond verses number of clusters with 30000 data

Number of clusters	Time in millisecond
5	205
10	214
15	246
20	131
25	158
30	152
35	223
40	234
45	251
50	191
55	241
60	314
65	362
70	382
75	225
80	368
85	349
90	338
95	378
100	390

5 Conclusion

The results of the present report show many considerable points. At first the k-means algorithm is capable of running in parallel, as expected. According to the result, in parallel mode, the execution speed was about twice that of sequential mode. In performances with fewer data and clusters, the parallel mode advantage is less, as parallelism always has an over head. So when the data is less, the parallel is less advantageous than the sequential. The graphs are almost linear which indicates that the order of computations is N relative to the number of data and the number of clusters. Of course, this is obviously independent of the type of k-means algorithm, being parallel or sequential, so both diagrams for sequential and parallel performances increase linearly with the number of data and clusters.