Application for Facility Location Problem in Waste Management

Technical documentation

Albert Khaidarov

UP FAMNIT

August 2021

Table of content

1. Introduction

The purpose of our application is to determine the optimal locations for processing some kind of waste (e.g. discarded plastic, glass, paper). This problem also known as Facility Location Problem, which is one of fundamental problems, studied in operational research and theoretical computer science. This kind of facility location problem is NP-hard, so for solving this we need to design approximation algorithm, in this case it is one of the most popular clustering algorithm – K-means.

We have a certain area, and set of locations with capacities, which are garbage collection facilities. Each capacity is the annual amount of waste accumulated in tonnes, and we need to find an optimal solution for allocating processing plants for all these points, so that it will decrease the cost for transportation, fuel consumption, labor time, etc. In this case, we need to consider not only the distances, but also capacities of each accumulation sites. That means we need to minimize the distances between accumulation sites and processing plants, considering their capacities.

2. Problem definition

There are plenty of algorithms using for clustering, and one of the best known is K-means. It is a Centroid-based clustering, central vector represents the number of clusters, which is not necessary to be a member of the data set. We find k cluster centers, and assign the objects to their nearest centroid with the minimum squared distance. After constructing is done, each customer is assigned to exactly one of K clusters, and K is our number of processing plants. But this classical approach cannot be applied for our problem, since for optimization we need to consider not only the distances between points, but also we need to take in account their capacities. To solve this I used a formula for "center of mass", which is quite known in physics. *The center of mass is the unique point at the center of a distribution of mass in space that has the property that the weighted position vectors relative to this point sum to zero. In analogy to statistics, the center of mass is the mean location of a distribution of mass in space.*

In the case of a system of particles P_i, $i = 1, \ldots, n$, each with mass m_i that are located in space with coordinates $\mathbf{r}_i$, $i = 1, \ldots, n$, the coordinates **R** of the center of mass satisfy the condition:

$$\sum_{i=1}^{n} m_i(r_i - \mathrm{R}) = 0$$

Solving this equation for **R** yields the formula:

$$R = \frac{1}{M} \sum_{i=1}^{n} m_i \, r_i$$

where $M = \sum_{i=1}^{n} m_i$ is the total mass of all of the particles.

The algorithm has the following steps:

1. Choose the number of clusters.
2. Then we randomly generate centroids of each cluster.
3. Assign each point to the cluster based on its capacity and distance to centroid using the formula center of mass.
4. Then we update the new centroids of the respective clusters by calculating the means of cluster's points.
5. And repeat 3rd and 4th steps until convergence criterion is met.

Since our algorithm involves randomness, we violating the fifth step and running for a given number of iteration for testing purposes.

3. Design

For the purpose of parallelization, we must decide how we can distribute the workload across the specified number of computational threads. Distribution in our K-Means Algorithm should be almost or absolutely equal, since we have a fixed number of workers throughout the execution. Other requirement in designing of our requirement is in synchronization. Assigning points to clusters and updating their centroids must be done in consecutive way. Generally, K-Means Algorithm ideally fit for parallelization of processes, and we can divide our workload for both computation steps:

- On assigning points to clusters phase, because each point does not change its location and capacity (it is fixed annual amount of waste). We can equally distribute all our sites for tasks, such that if we have 400 points and 4 threads, we can assign 100 sites for each thread.
- On the phase of recomputing centroids of clusters. Here we can distribute our workload for each cluster, since in this part we doing summation of centers of mass.

Besides this, we need to take into account the processes that accompany parallelization, such as initialization and synchronization. Hence, if we making parallelization with the big number of processes, these operations can make our execution time slower. So we need to be careful about how we parallelize our tasks.

4. Implementation of the algorithm

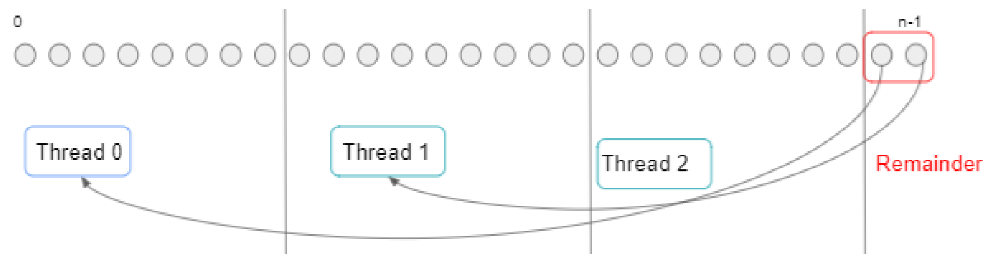
Our algorithm has two main methods: bindToClusters and updateCenterOfMass.

The steps are following: firstly we are initializing coordinates of clusters by choosing random centroids. Then we executing the method *bindToClusters*, which is needed for assigning all our points to the clusters. We returning an array with corresponding indexes, and this array will contain the index of assigned cluster from 0 to N-1 (where N is the number of clusters). And "Double" is returning the minimal distance between the objects. After we updating centers of mass with method *updateCenterOfMass*, in this stage our clusters are moving, and hence, changing their coordinates. Last two methods are iterating until convergence is met, or number of cycles is finished.

4.1. Parallelization

In our parallel or multithreaded program we distribute our workload to two or more processors, that running simultaneously, and we dividing our tasks between them. After processing of each chunk within each task is done, we merge these chunks together as in "divide and conquer" principle. In this mode we need to know which instructions we need to divide and which instructions is need to be dedicated for each of processors.

For parallelizing our program we using threads. Firstly, we splitting our set of sites into nearly equal parts and process them separately. Generally, we can divide the workload by clusters, but in this case, our algorithm will work slower because we are increasing resource costs for additional tasks. So in our parallel implementation we dividing by sites, because, as I already mentioned, each our point can be processed independently. That's why we can parallelize our program to the maximum, in theory we can even assign each point to one thread. But of course, in real world we have the hardware constraints, in particular, we need to consider how many cores we have. That's why with limited number of processors we need to have equal or almost equal distribution of tasks between threads. Therefore, we dividing all our sites to number of threads, and in case if there are some sites are left (reminder), we assign each point among threads (Picture 1).



Picture 1. Dividing sites to threads in the parallel mode

As was mentioned earlier, one of the main challenges in the parallel implementation of our algorithm lies in synchronization between our two main methods (bindToClusters and updateCenterOfMass). To manage this was used *CountDownLatch* class, which allows one or more threads to wait until a set of operations being performed in other threads completes. We are initializing it with the number of tasks, and when one of threads finishing its task, this counter decreases by 1 and thread will sleep until other threads will come to this barrier. When counter will become 0, we can proceed with the next step and all threads will be assigned with the next set of tasks.

We are initializing each thread, and creating new object called *BindToClusterThread*. We calling method *subList,* which returning a list of *n* points assigned to each thread.

After all tasks is done we merging results into one big array called *pointsOfClusters*.

Parallelization for the next step is implemented in a similar way. We taking array *pointsOfClusters,* updating the centers of mass, and returning recomputed array of clusters coordinates.

```
countDownLatch = new CountDownLatch(numThread);
clusterCoords();
runThreads();
joinThreads();
int cyclesPassed = 1
while $cyclesPassed \neq numberOfCycles$ do
    bindToClusters();
    countDownLatch.await();
    updateCenterOfMass();
    countDownLatch.await();
    cyclesPassed+=1
end
```

Algorithm 1. Parallelization of K-mean clustering for Facility Location Problem

5. Technical remarks

One of the requirements of our application is to display map with GPS-coordinates. For integrating a map were chosen library called OpenLayers, which is JavaScript library for displaying map data in web browsers. To run this on a local server I used NodeJS, and through this server we can visualize our map and clusterization part is running on Java Virtual Machine. For the algorithm itself, were used JavaFX library, which is a software platform for creating and delivering web-based desktop applications. And WebView is a mini-browser, also called an embedded browser in a JavaFX application.

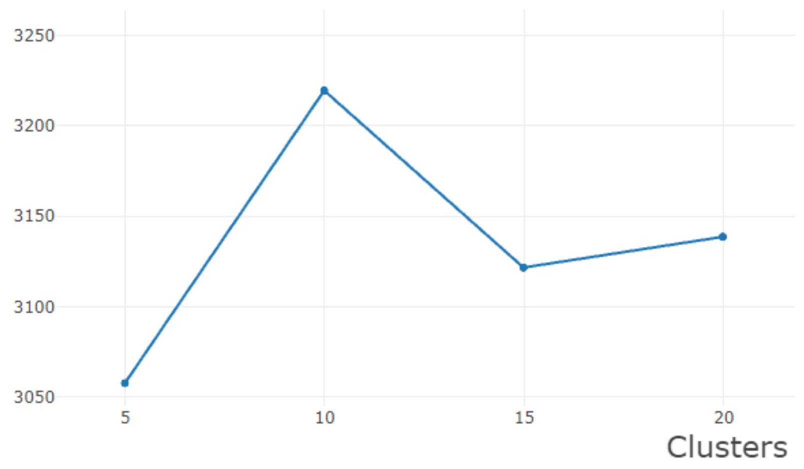
All tests were performed on a machine with 8Gb RAM and 4-core (8 treads) processor Intel Core i5-7400U.

After performing the calculations, the result is passed to the JavaFX and then to the WebView. The graphical user interface supports the full functionality of modern map applications, and allows us to perform all the needed actions such as zooming, moving, and so on. The test for each set of parameters was performed 3 times, and as a result, the average of them was taken into account. The number of cycles was also set to 1000. The best achieved results were with 4 threads, so measurements represent tests with '4' as number of threads.

6. Benchmarks

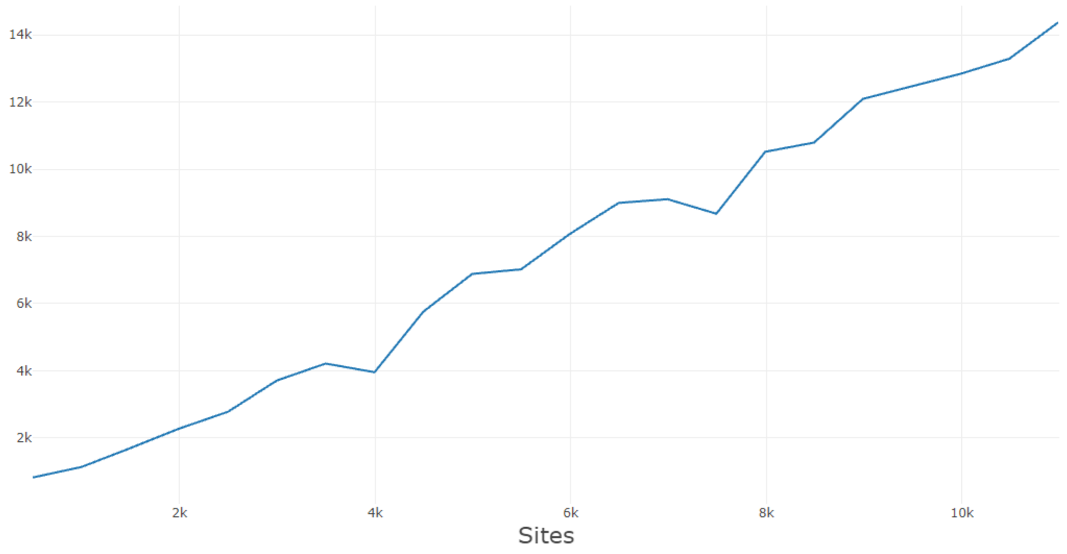
1. *Tests with fixed number of sites (30000 sites).*

Number of clusters	5	10	15	20
Time in milliseconds	3057	3219	3121	3138



2. *Tests with 20 clusters*

Number of sites	Time in milliseconds
500	793
1000	1106
1500	1669
2000	2251
2500	2754
3000	3686
3500	4191
4000	3932
4500	5741
5000	6865
5500	7001
6000	8059
6500	8983
7000	9093
7500	8661
8000	10508
8500	10782
9000	12084
9500	12461
10000	12833
10500	13285
11000	14366



7. Results

Looking at tables and graphs from the previous section we can conclude that the growing number of clusters does not have a strong impact to the time consumption of our algorithm. This can be easily explained by the fact that the most expensive operation is calculating the distance to the clusters' centers and not calculating means of clusters.

Tests with fixed number of clusters showed that we have linear growing of function with an increase in the number of sites.

8. Conclusion

From the described results we can conclude that our implemented program has satisfied all the initial requirements. As was already mentioned in this report, the K-Means algorithm is ideal for parallelization algorithms as it allows us to scale our tasks by sites. Of course, we need to keep in mind that we cannot have unlimited number of threads because of hardware limitations. Looking at the benchmarks we can see, that even with relatively slow hardware, we can sufficiently solve Facility Location Problem tasks for dozens of clusters and thousands of sites.