MTech Thesis

Student Name: Yesha Mittal

IIIT-D-MTech-CS-IS-11-001
July 24, 2016

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Dr. Vinayak Naik (Advisor)
Dr. Viswanath Gunturi (Advisor)
Dr. Sandip Chakraborty (External Advisor)
Dr.Chetan Arora (Internal Advisor)

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Information Security

Keywords: traffic, taxi, clustering, k-medoid, Dijkstra, CLARANS, Quadtree

Certificate

This is to certify that the thesis titled **"Finding optimal locations for taxi stands on city map"** submitted by **Yesha Mittal** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her under our guidance and supervision at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Vinayak Naik and Dr. Viswanath Gunturi
Indraprastha Institute of Information Technology, New Delhi

Abstract

We present an algorithm which suggests suitable locations for taxi stands on a city map. This is done using pickup locations information gathered from GPS devices installed on taxis. We aim at finding locations where taxi drivers are more likely to get a passenger or locations from where passengers can be reached by travelling minimum possible distance. To do the above mentioned tasks, we have modified CLARANS algorithm. The two major changes suggested by us include 1) Selecting initial seeds by dividing space using Quadtree approach 2) Swapping medoids with non-medoids in 'nearby' areas. We have tested our method using trajectory data generated over 90 days in Singapore and have compared our solution with the optimal solution obtained using PAM algorithm. We also show that not much variation occurs in clusters' quality if we increase the size of 'nearby' areas but running time increases to almost double the value.

Acknowledgments

I take this opportunity to express my profound gratitude and deep regards to my guides **Dr. Vinayak Naik and Dr. Viswanath Gunturi** for their exemplary guidance, monitoring and constant encouragement throughout the project. The door to both the professors' office was always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently steered me in the right the direction whenever they thought I needed it.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement.

This accomplishment would not have been possible without all of them. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the growth of population, the demand and need of smart cities is increasing. The aim of smart cities is to make optimal use of available resources, of course, by maintaining balance between environmental, social and economic costs. The number of Multinational corporations are also increasing and so is the number of people working for them. Gone are those days when industries were located on the outskirts of a city and people working in them lived around them. Now these big companies are scattered all around the city and not just the MNCs, the basic amenities like the hospitals, airports, schools etc. span across a wide geographical area. This has lead to a wide increase in mobility of people in different directions. This is supported by availability of a large number of taxis and other public transportation.

Our work here investigates the mobility pattern of people traveling through taxis in a city. Basically, we have the data about the pickup and drop locations of people using taxi services in Singapore. This allows us to know the demand of taxi services of all the areas of Singapore and also how it varies at peak hours of the day. Using this information about demand of taxi services, we aim to make taxi stands in the city. The number of taxi stands to be made is constant and is specified by any taxi service company using our work. This number can depend on various factors like land area of the city, cost of installing each taxi stand, total demand etc. So, given an input about the number of taxi stands, the main factors we incorporate in installing taxi stands are: 1) Demand of different areas: Higher is the demand of an area, more is the probability of installing a taxi stand in it 2) Demand can vary over space and time.

So, for the purpose of knowing the demands of different areas, we started by exploring the data. The dataset was quite big and the pickup locations were spread across the whole Singapore. Also, the demand of taxi services varied with time. This led us to investigate the spatio-temporal behavior of the pickup points. As there existed change in number of pickup points in different areas over space and time, we switched to different partitional methods of Clustering to get the appropriate locations of taxi stands at different times of day. This is covered in detail in Chapter 5.

We looked at various clustering techniques. We started from PAM but as it is not suited for big datasets, we switched to CLARA and then CLARANS. Finally, what we have developed in this work is the modified version of CLARANS. The modification is particular to our problem statement or any work similar to ours. The two major changes that we have made are: 1) The way CLARANS chooses its initial seeds 2) The way in which medoids are swapped with non-medoids. The details are covered in Chapter 6. The result of the clustering algorithm is the set of locations suitable for installing taxi stands in Singapore.

Or results in chapter 7 show a trade-off between quality and runtime between solution obtained

from our algorithm and the one obtained using PAM. We also show that not much variation occurs in clusters' quality if we increase the size of 'nearby' areas but running time increases to almost double the value.

The rest of the report is organized as follows: Chapter 2 explains our Problem Statement in detail, chapter 3 gives a brief overview about the similar work done in this field, chapter 4 looks deep into the clustering algorithms and other techniques useful to understand our algorithm, chapter 5 gives proof of spatio-temporal variation, chapter 6 describes our clustering algorithm, chapter 7 shows different pre-processing techniques and the results obtained through various experiments and finally chapter 8 gives the conclusion and future works.

Chapter 2

Problem Statement

This chapter aims at giving details of all the attributes we receive as the input, the output we want to generate and the objective function driving the output.

2.1 Input

- **Dataset containing pickup locations:** In abstract terms, the input is the demand of taxi services in a city. The data which we receive is in the form of separate .csv files for each day of each month and contains information about the locations where passengers take cabs in Singapore and the places where their trips end. These places are marked as latitude, longitude of start and end points of their trips. A subset of fields from the dataset are shown in table 2.1. The entries in each file start from 12:00 AM in the night and end at 11:59:59 PM. The time attribute helps us to find different patterns in demand across a day. For example some areas might show higher demand at particular time durations of the day. We extract all the start latitude and longitude from the .csv file which gives the locations of all the pickup points in Singapore.

Start(date, time)	End(date, time)	Start lat	Start lon	End lat	End lon

Table 2.1: Format of trip file

- **No. of taxi stands:** This refers to the total number of taxi stands a taxi service company wishes to install in a city. As this is a constant and depends on the whims of the service company, this is taken as an input.

2.2 Output

In a given city, the output is the set of locations suitable for installing taxi stands. The criteria for 'suitability' is explained in section 2.3.

2.3 Objective

Our main objective is to find 'k' locations of taxi stands in a given city such that the average distance between pickup points and taxi stands can be minimized. These details will be explained

in chapter 6.

2.4 Example

Here, we will give an example of a city with pickup points in it. This example will be traced through while we explain the algorithm in chapter 6 to finally get the set of locations for taxi stands in the city.

Suppose there exists a small city in which we are just given in input the locations from where taxis are taken. As the input is in the form of latitude and longitude of the pickup points, let's assume the locations look somewhat like the one shown in figure 2.1.

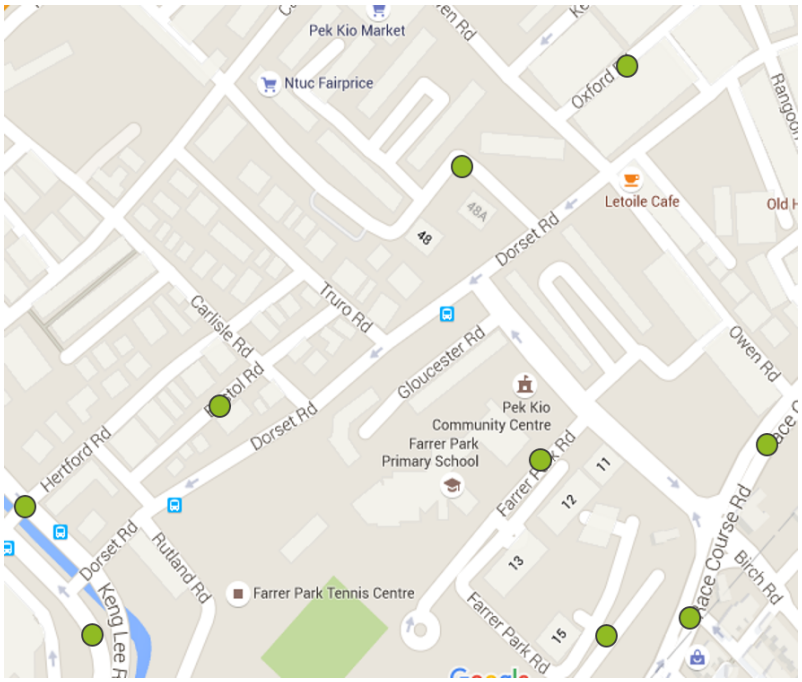

Figure 2.1: An example showing pickup points in a city. All the green dots denote the pickup locations

This example will be used throughout our algorithm in 6 to see how it works over a city.

Chapter 3

Related Work

3.1 Location Recommendation for taxi drivers or passengers

Yuan et al. [17] proposed a recommendation system for taxi drivers and people waiting for taxis. Their work is based on the knowledge of passenger's mobility patterns and taxi drivers' pickup behaviors. The recommender provides taxi drivers with some locations, along with routes to these locations, where they can find passengers quickly and maximize their profit. It also recommends people with locations, within walking distance, where vacant taxis can be found in minimum waiting time. Their main contribution is in devising a probabilistic model to formulate time-dependent taxi behaviours based on which they build the recommendation solutions for both taxi drivers and passengers.They show that their method can provide high-profit locations to taxi drivers and precision of recommendation reaches upto 67%.

Ge et al. [3] provide a case study focused on 'extracting energy-efficient transportation patterns from location traces'. They develop a mobile recommender system which gives sequence of pickup locations and parking spots to taxi drivers. They start by saying that the potential pickup locations can be determined by clustering pickup locations over certain time and using their centroids as pickup locations. As this has a high computational cost with increasing number of pickup points, they provide Potential Travel Distance (PTD) function to evaluate different candidate routes. The function generates a smaller subset of candidate routes due to its monotone property. They also developed a route recommendation system which uses monotone property of PTD (LCP). They also proposed the SkyRoute algorithm to compute sky-line for candidate routes. Finally they show that LCP and *SkyRoute* out-perform other brute-force methods.

Another work in the field of passenger-finding was done by LI et al.[8]. They tried to investigate both the efficient and in-efficient strategies of passenger finding using the taxi-GPS dataset. Basically they tried to find out, what are the key factors that determine a taxi driver's performance and and how these factors can be used to help ordinary drivers to improve their business profit. So, to solve this problem, the main challenge they faced was to convert the 'driving strategies' into machine-understandable form. To do so, they converted passenger-finding strategies as triplets (Time, Location, Strategy) to differentiate between good and ordinary performance based on taxi driver's behaviour. They then find all the combinations of the three attributes and count the number of times a taxi falls in each pattern. L1-Norm SVM is then adopted as feature selection tool to differentiate between ordinary and top performance taxis. The taxi predictor they built showed a prediction accuracy of 85.3%

Similar study was done by Gao et al. [2] which differentiated between top and ordinary drivers

based on their incomes. They used two novel encoding schemes Choice-of-Location graph Move/Wait Strategy tree to find out driver's behaviors when their taxis are vacant and while finding operating locations. They try to investigate taxi drivers' mobility intelligence using data collected from GPS devices installed on taxis. Through their visualization they try to find out 'preferred drivers' locations' and their move/wait strategies. Their study gives the following findings: High income drivers are more intelligent and have more customers. They tend to have more favoured regions while low-income drivers are more concentrated in smaller areas. They actively search for customers and stick to their move/wait strategies during vacant trips.

Zheng et al. [19] devised a method to give an estimate of waiting time for vacant taxis at a given time and place and then based on this they develop an approach to recommend passengers about suitable locations where they should wait for a taxi. To address their problem they use the non-homogeneous Poisson process (NHPP) to know the behavior of vacant taxis. They estimate the waiting time at different times on road segments using the statistics of parking time of vacant taxis on the roads and the number of vacant taxis leaving the road in history. Their major contributions in this field include: employing NHPP to model the behavior of vacant taxis to derive the probability distribution of waiting time, design recommender for people who want to take taxis based on waiting time and they also developed a mobile application to help the users find an appropriate place to wait for their taxi.

Phithakkitnukoon et al. [14] proposed a predictive model to find the number of vacant taxis in a given area based on time, week-day and weather conditions. They used naiive Bayesian classifier to get probability distributions from historical data. Using 150 taxis in Lisbon, Portugal, their model is able to predict accurately with error of 0.8 taxis per 1x1 km^2 area.

Liu et al. [10] discuss the issue that when a driver is new to the city or is at a road unknown to him what measures does he take to learn about that road? They argue that in such scenarios, the driver learns not just from his own experience but also through interactions with other drivers. In their work, they call this problem as Socialized Information Learning (SIL) and propose models to learn about how taxi drivers gather information in an uncertain area using their social network. Their main contributions are proposing an **Individual Information Model** to describe information collection of taxi drivers, then they introduce the **Socialized Information Model** and then finally devise **Dynamic Socialized Information Model** to gain knowledge using spatio-temporal characteristics in dynamic scenarios.

As most of the proposed systems were concerned with individual customer satisfaction, Seow et al. [16] presented a novel multiagent approach for automating taxi dispatch system. They argued that most of the operators try to increase customer satisfaction in a local manner but the proposed system works more globally. They further add that increasing individual customer satisfaction involves assigning nearest taxi to the customer based on who cam first but this does not consider the effects it might have on other waiting customers. So, their main motive is to increase 'group average customer satisfaction' rather than of an individual. Thus, they try to answer the question *"Would (group)total customer waiting time shorten if two taxis are allowed to exchange their currently assigned bookings?"*. So, they propose a multiagent approach to answer this question by deploying collaborative taxi agents that can communicate among themselves, on behalf of taxi drivers, to decide assignments for themselves from different taxi requests generated within given time frame. The results showed that their could reduce empty cruising time and customer waiting time by 26.3% and 33.1% respectively; and upto 41.2% and 41.8% reduction using a simple negotiation speedup heuristic.

As different approaches aim to make taxi recommendation systems, a considerable amount of work goes in identifying the traffic patterns of the city. Liu et al. [9] suggested a method to detect 'crowdedeness spots' in a city. They proposed a novel density based approach called

mobility-based clustering for the same. They argue that the traditional clustering algorithms are insufficient to capture real clustering characteristics of moving vehicles. Thus, *mobility-based clustering* is based on the observation that vehicles generally have high mobility (speed) and areas with high speed can be termed as less crowded than others and vice versa. Their major contributions are: proposing novel mobility-based model to determine the crowdedness of an area using the mobility and object dynamism, investigating several factors that have impact on vehicle crowdedness measurements, categorizations of different spots using spot mobility and crowdedness dynamism and finally they show that 'top crowdedness spots are locality consistent over time, while more crowdedness spots present more locality variations'.

3.2 Clustering techniques

The other set of study we did for our work was regarding different clustering techniques. We began our study from k-means clustering algorithm [11] (mentioned in section 4.2.1). As the algorithm is not good in case of outliers, we shifted to *k-medoids* algorithm. The most common realization of *k-medoids* algorithm is the *PAM* algorithm. It was developed by Kaufman and Rousseeuw [7]. In a dataset of N objects, PAM starts by selecting k arbitrary objects. Then it swaps one of the medoids O_m with the non-medoid O_p till the time quality of clustering is improving. The details of PAM algorithm are mentioned in section 4.2.3.
As PAM algorithm is computationally infeasible for large datasets, we then switched to CLARA (Clustering LARge Applications) which relies on sampling. It draws a sample from the dataset, applies PAM on it and gives the medoids for the sample. The point to note is that the more randomly generated the sample is, higher would be the approximation of medoids for the whole dataset. Experiments reported in [7] show that CLARA is more efficient than PAM for larger values of n.

After CLARA, we then moved to CLARANS algorithm stated in [12]. The main motivation for CLARANS algorithm was the graph abstraction. They stated that the procedure of finding k medoids can be visualized as searching through a graph. They said that the graph can be represented by $G_{n,k}$ and every node of this graph is a set of of k objects $\{O_{m_1,\ldots,m_k}\}$ where $O_{m_1,\ldots,m_k}$ are the selected medoids.
Two nodes of the graph are considered neighbors if they differ in exactly one object i.e. two nodes $S_1 = \{O_{m_1,\ldots,m_k}\}$ and $S_2 = \{O_{w_1,\ldots,w_k}\}$ are neighbors only if their intersection cardinality is $k-1$. Thus, each node has $k(n-k)$ neighbors.
Through their work they were able to show that for small data sets, CLARANS is faster than PAM but is much faster for larger datasets. Also, CLARANS is not restricted to search in a specific subgraph as in the case of CLARA and in the same amount of run-time, produces much better quality clustering than CLARA. Further details of CLARANS algorithm are given in section 4.2.4.

Estivill-Castro et al. suggested in their work [1] some hill climbing heuristics for improvements in CLARANS algorithm. They proposed a novel idea of stopping early of the heuristic search which largely saves computational time and while the quality of clustering remains unchanged. They argued that the clustering problem translates into a 'combinatorial optimization problem' where one needs to find the set of representatives that minimizes MP defined by

$$MP(C) = \sum_{i=1}^{n} d(s_i, rep[s_i, C]), \tag{3.1}$$

for all subsets C of S with $||C|| = k$ where MP(C) represents quality of clustering. Just like

as mentioned in [12], the authors argue that the search space to find minimum value of MP(C) can be visualized as a graph whose set of nodes is all the subsets $C \subseteq S$ where $||C|| = k$. Local Search [6] is this graph is computationally feasible. The $Local - Search$ heuristics for MP start by randomly selecting a solution C_o and then iteratively the heuristic explores a neighboring node C_{t+1} of the current node C_t. Typically, $MP(C_t) > MP(C_{t+1})$ and the search stops when no neighbor with lower cost is possible. This results in a local optima. Their studies revealed that LOCAL HILL CLIMBING is the best local search heuristic. They also showed that convergence to a local optima for GLOBAL HILL CLIMBING is $O(N_G n^2)$ time (in case of PAM), for RANDOMIZED HILL CLIMBING it requires $O(N_R \beta n)$ time, as in the case of CLARANS, and for LOCAL HILL CLIMBING, its $O(N_L n)$ time where N_G, N_R and N_L are the number of improvement cycles.
Selman et al. also explain hill-climbing search strategies in [15].

Other than the above mentioned papers on clustering techniques, we also went through some papers which improve the efficiency of k-medoid algorithm. For example Zhang et al. [18] presented a new algorithm that utilizes TIN of medoids to facilitate local computation while searching for optimal set of medoids. They finally showed through their experiments that their algorithm is faster than the basic PAM algorithm and is more efficient than CLARA in case of larger number clusters. Another faster approach for k-medoid algorithm was suggested by Park et al. [13]. They stated that their algorithm tested various methods of selecting initial seeds while still running K-means algorithm. Their algorithm calculated the distance matrix only once and used it to find new medoids at each iterative step. Their experiments showed that the proposed algorithm took much lesser time as compared to other partitioning around medoids methods.

As we have worked on a graph of road network, other than partitioning methods of clustering, we also looked at chameleon algorithm, suggested by Karypis et al., which is a hierarchical clustering algorithm. It is a two-phase algorithm in which in phase 1, it uses a graph-partitioning algorithm to form large number of small sub-clusters while in second phase it uses agglomerative hierarchical clustering algorithm to form approapriate clusters using the sub-clusters. In phase 2, while merging the sub-clusters, chameleon uses interconnectivity and closeness of the clusters. Experiments have shown that chameleon can discover arbitrarily shaped clusters of better quality as compared to algorithms like BIRCH and DBSCAN.

3.3 Research Contributions

The work done by us makes the following contributions:

- We present an approach to find optimal set of locations for taxi stands in a given city. The approach presented by us works more on a global level(for the whole city), rather than focusing on individual profit as opposed from most of the works presented in section 3.1.
- We use partitioning methods of clustering to find the set of locations while other works focus more on probabilistic models or other techniques.
- We have modified the existing CLARANS algorithm in two ways:
  - Using Quadtree approach to partition space and selecting initial seeds.
  - Swapping medoids with non-medoids in 'nearby' areas.

  The solution is then compared with the optimal solution obtained through PAM.

Chapter 4

Background

The following section explains some of the terms and techniques that are necessary with regard to the work done in this thesis.

4.1 Open Street Maps

Open Street Maps is a collaborative project to enable people to use and enhance a free map of the world. The map data is crowd-sourced and is available under an open license. Increase in the availability of portable satellite navigation devices is one major reason for the establishment and growth of OSM. The new restrictions imposed on Google maps which charges a user 4 per 1,000 map loads limits the availability of map information across the world. This is another driving factor promoting the usage of a freely available map of the world. Crowd-sourced OSM has outdone Google Maps at various instances, one of them being provision of detailed maps of areas in and around Sochi during the 2014 Winter Olympics. The data generated by the OSM project is now used in many applications like its usage by MapQuest Open, Foursquare (replaced Google Maps), Apple, etc. Though the quality of collected data varies worldwide, the current number of registered users nearing 1.5 million and still growing is the reason for the growth of Open Street Maps. The project was launched 9 years ago by Steve Coast and has been awarded by the Free Software Foundation Europe (FSFE) for its work on Open and Emerging Standards on occasion of the Document Freedom Day.

4.2 Clustering

"Cluster analysis or clustering is the process of partitioning a set of data objects into subsets. Each subset is a cluster, such that objects in a cluster are similar to one another, yet dissimilar to objects in other clusters" [5]. These clusters are then used to analyze the given data distribution and apply various other data mining techniques.

Many clustering algorithms exist in literature but the major clustering methods can be classified as follows:

Partitioning methods: For a given set of n data objects, partitioning methods aim to create k subsets of data such that $k \leq$n and each subset must contain at least one object.

Hierarchical methods: These methods create a hierarchy of given set of data objects. The method can be further divided on the basis of hierarchical decomposition formed as *agglomerative*

or *divisive*.

Density-based methods: These techniques can only be applied on spherical shaped clusters and work on the basis of distance between objects.

As our work focuses on finding optimal set of taxi hubs in the city of size k, here we will discuss only partitioning methods of clustering in detail. The most common types of partitioning methods are k-means and k-medoids.

4.2.1 *k*-Means

For a data set D of n objects, this partitioning technique aims at distributing the objects in k clusters, $C_1...C_k$ such that $C_i \subset D$ and $C_i \cap C_j = \phi$ for $(i \geq 1, k \geq j)$. The quality of a cluster is determined with the help of an objective function which gives an estimate of how similar objects within same cluster are and different from objects in other clusters.

As the name itself suggests, this partitioning technique uses mean of the objects of a cluster c_i, also known as *centroid* of C_i, to represent it. To assign an object **p** to any cluster C_i, the difference between the centroid of C_i and **p** is calculated. The difference is measured by *dist(**p**,c_i)*, where dist(a,b) can be Euclidean distance or any other distance measure between points **a** and **b**. The objective function defining the quality of a cluster is the sum of squared distances between all objects in C_i from its centroid c_i as

$$E = \sum_{i=1}^{k} \sum_{p \epsilon C_i} dist(p, C_i)^2, \tag{4.1}$$

where E is the sum of squared errors of all objects p in dataset D. The objective function tries to make the clusters separate and compact.

The k-means procedure is as shown in Algorithm 4.1

Algorithm 4.1 k-means

Input:

- D: Dataset of n objects
- Integer $k \leftarrow \#clusters$

Output: A set of k clusters

1: **procedure** k-MEANS
2: Initialize: k cluster centroids $c_1...c_k \ \epsilon \ D$ randomly
3: Repeat till convergence occurs:{
4: **for** each i **do**
5: $m_i = \underset{j}{\operatorname{argmin}} \|x_i - c_j\|^2$
6: **end for**
7: **for** each j **do**
8: $c_j = \frac{\sum_{i=1}^{n} 1\{m_i = j\} x_i}{\sum_{i=1}^{n} 1\{m_i = j\}}$
9: **end for**
10: }
11: **end procedure**

The time complexity of k-means algorithm $O(nkt)$ where n is the number of objects in D, k is number of clusters to be formed and t is number of iterations.

4.2.2 *k*-Medoids

The main short-coming of k-means algorithm is that it is sensitive to outliers. This is because the distance of outliers from the centroid is much more with respect to other data objects and thus they can change the mean value of the cluster dramatically. k-medoids algorithm provides the solution to this problem.

***k*-medoids** algorithm, instead of picking mean of the data objects as cluster representative, picks actual objects from the clusters as their representatives. The partitioning method then assigns each object p to the cluster representative by minimizing the sum of dissimilarities between them. The objective function can be defined as

$$E = \sum_{i=1}^{k} \sum_{p \epsilon C_i} dist(p, o_i)^2, \tag{4.2}$$

where E is the sum of squared errors of all objects p in dataset D and o_i is the representative object of cluster C_i also known as *medoid* of C_i.

The **Partitioning Around Medoids (PAM)** algorithm is the most common realization of k-medoids algorithm. It is explained in details in section PAM.

4.2.3 PAM: Partitioning Around Medoids

PAM is a common realization of k-medoids algorithm. It is based on an iterative and greedy approach. Just like k-means, the algorithm starts with randomly selecting initial representative objects from the dataset. Then, iteratively, the representative object is replaced with a non-representative one, till the time cost (eq. 4.2) is decreasing. All the possible replacements are tried out in this way. Let us see this by an example.

Consider a dataset D, with n objects. Let $o_1, ...o_k$ be the initial set of medoids. For a given set of medoids, all the 'neighboring' sets of medoids are computed. A neighbor set is the set which has only one different object. Like, for a set of three medoids $\{o_1, o_2, o_3\}$, a neighbor set would be o_1, o_2, o_{random}. Thus, one can say that, two sets S_1 and S_2, with k medoids each, are neighbors if their intersection cardinality is $k-1$ [12]. Thus, each set of medoids has $k*(n-k)$ sets of neighbors possible. Then, to select the set of medoids with the least cost, distance from every object **p** to the current set of medoids is calculated. The current set is then replaced with any other neighbor set and the cost is recomputed. The procedure is repeated till the time cost is decreasing. The algorithm for the procedure is given in algorithms 4.2.

So, we see that PAM is more robust than k-means in presence of outliers but each of its iteration has a complexity of $O(k*(n-k)^2)$ which is a very costly computation for large values on n and k. Thus, there exists need of an algorithm which can handle large datasets. Following this motivation, CLARANS algorithm was developed which can be used as a trade-off between cost and quality.

Algorithm 4.2 PAM

Input:

- D: Dataset of n objects
- Integer $k \leftarrow \#clusters$

Output: A set of k medoids

1: **procedure** PAM
2: Initialize: k cluster centroids $c_1 ... c_k \ \epsilon \ D$ randomly as set L
3: L = pam(D,L)
4: **end procedure**

Algorithm 4.3 pam: Gives local minima in terms of cost for a given set of medoids

1: **function** PAM(D, L)
2: θ = computeCost(D, L)
3: Nbrs = getNeighbors(D, L)
4: **for** Each medoid list L' in Nbrs **do**
5: γ = computeCost(D, L)
6: **if** $\gamma < \theta$ **then**
7: **return** pam(D,L')
8: **end if**
9: **end for**
10: **return** L
11: **end function**

Algorithm 4.4 Computed the cost of associating each object p in D to a given set of medoids L

1: **function** COMPUTECOST(D, L)
2: $\theta = 0$
3: **for** Each object p in D **do**
4: $dist \leftarrow \min_m$ distance(p,m)[1] (for all medoid points m)
5: $\theta += D$
6: **end for**
7: **return** θ
8: **end function**

Algorithm 4.5 Computes all the neighboring sets of a given set of medoids

1: **function** GETNEIGHBORS(D, L)
2: Neighbors = {{}}
3: **for** Each medoid m in L **do**
4: **for** Each object p in $D - L$ **do**
5: $L' = L - \{m\} \cup \{p\}$
6: $Neighbors \cup L'$
7: **end for**
8: **end for**
9: **return** Neighbors
10: **end function**

4.2.4 CLARANS

Clustering Large Applications based upon RANdomized Search is a randomized algorithm. It first randomly selects the k initial medoids from dataset D and computes cost of the configuration using equation 4.2. Then, from all the possible 'neighbors' (refer to section 4.2.3), it randomly selects any one neighbor and swaps it with the current list of medoids. If the cost decreases, it makes the replacement and repeats the procedure $'maxNeighbor'$ times [12]. CLARANS repeats the whole randomized process $'numlocal'$ times [12] and gives the best local optimal set of medoids possible. The two parameters of CLARANS, *maxneighbor* and *numlocal*, are the maximum number of allowed neighbors and the number of local optima obtained respectively. The algorithm for CLARANS is given in algorithm 4.6.

Algorithm 4.6 CLARANS

Input:

- D: Dataset of n objects
- Integer $k \leftarrow \#clusters$
- Integer $numlocal \leftarrow$ #(local minima obtained)
- Integer $maxneighbor \leftarrow$ #(maximum number of neighbors tested)

Output: A set of k medoids

1: **procedure** CLARANS
2: L = clarans($D, k, maxneighbor, numlocal$)
3: **end procedure**

Algorithm 4.7 clarans: Gives the best local optimal set of medoids possible

```
1: function CLARANS(D, k, maxneighbor, numlocal)
2:     mincost ← ∞
3:     medoids = {}
4:     Nbrs = getNbeighbors(D, L)
5:     while i < numlocal do
6:         Initialize: k cluster centroids c_1...c_k ϵ D randomly as set L
7:         L' ← findLocalMinima(D, L)
8:         θ = computeCost(D, L')
9:         if θ < mincost then
10:            mincost ← θ
11:            medoids ← L'
12:        end if
13:        i++
14:    end while
15:    return medoids
16: end function
```

Note: functions *computeCost*(D,L) and *getNeighbors*(D,L) are same as in Algorithms 4.4 and 4.5 respectively.

Thus, CLARANS algorithm gives the benefit of randomly selecting the 'neighbor' medoid sets

Algorithm 4.8 Gives the set of medoids with least possible cost as in eq 4.2

```
1: function FINDLOCALOTIMA(D, L, maxneighbor)
2:     θ = computeCost(D, L)
3:     Nbrs = getNeighbors(D, L)
4:     while j < maxneighbor do
5:         L' ← Randomly choose a list from Nbrs
6:         γ = computeCost(D, L')
7:         if γ < θ then
8:             return findLocalMinima(D, L', maxneighbor)
9:         end if
10:        j++
11:    end while
12:    return L
13: end function
```

and also limits the number of neighbors tested using the parameter *maxneighbor*. If *maxeighbor* is increased to the value $k*(n-k)$, and *numlocal* is set as 1, CLARANS will execute like PAM.

4.3 Other Algorithms and Definitions

4.3.1 QuadTree

Quadtrees are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes. Some common features of quadtrees include:

- They decompose space into adaptable cells.
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits again in four quadrants.

4.3.2 Dijkstra's Algorithm

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing.

The algorithm aims to solve the following problem: Given a graph with n nodes and a source node in the graph, find shortest paths from source to all nodes in the given graph. Thus, it is said to solve the single-source shortest path problem (SSSP). It does so by generating a shortest path tree with the given source node as the root.

Dijkstras original algorithm does not use a min-priority queue in its implementation and runs in time $O(n^2)$. On the other hand, if a min-priority queue is implemented as a binary heap (if the graph is sparse), the running time of the algorithm is O((n + m) log(n)) (where m is the number of edges in the graph) or as a Fibonacci heap the running time of the algorithm is O(n log(n)).This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

4.3.3 silhouette Coefficient

"The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation)" [**silhouette**]. The value of silhouette ranges from -1 to 1. higher the value of silhouette, better is the clustering which means that the object is well-matched with its own cluster and differs from the neoghboring clusters. If many objects have negative or low values, it means that the there are either too many or too few clusters present.

Suppose there is a dataset D with k clusters. Let a_i be the average dissimilarity of i with the data within its own cluster. Let b_i be the lowest average of datum i to any other cluster to which i does not belong. So, silhouette is defined as:

$$s_i = (b_i - a_i)/max\{a_i, b_i\}, \tag{4.3}$$

So, it is clear from above that,

$$-1 \leq s_i \leq 1, \tag{4.4}$$

Average s_i for whole data gives a measure of how appropriately the whole data is clustered.

Chapter 5

Exploration

This chapter looks deep into the input data to see its variation across space and time. As mentioned earlier in section 2.1 the data is in the form of separate .csv files containing information about the pickup locations in the form of latitude and longitude.

When we first started working with the cab data, we found that the size of the data was very large, even for a single day. Each file had around $4 * 10^5$ entries where each entry corresponds to a new trip. So, to have a visual understanding of the start locations of the trips, we started plotting data of different days over Singapore map. A snapshot of data of one of the days is shown in figure 5.1 plotted using Google Fusion Table API[4]

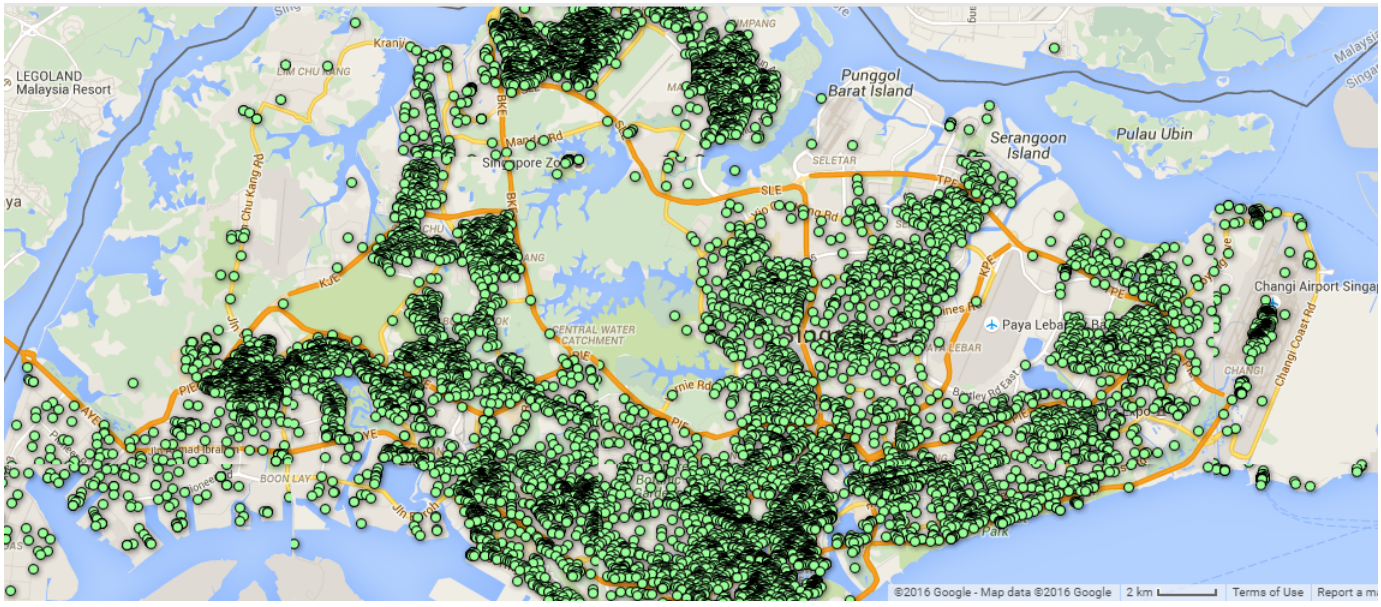

Figure 5.1: Start trip locations (green dots) of a single day plotted over Singapore map

To proceed further, we started analyzing the data on an hourly basis rather than taking data of the whole day. This had the advantage that the number of trips of an hour were much less than than of the whole day and helped us finding temporal variations in data. Some of the observations are listed in sections 5.1 and 5.2

5.1 Spatial Variation

To know about spatial orientation of data, we first plotted all the start points of the trips of each hour of a day onto the map of Singapore. We saw that many trips had overlapping start locations and the number of pickup points of areas varied from very high to very sparse low over space. This clearly was a signal that the number of start points vary over space. Like for example, figure 5.2 shows the pickup locations of a day in Singapore from 8:00-9:00AM. Figure 5.3 and 5.5 show in detail how the total number of pickup locations vary over space.

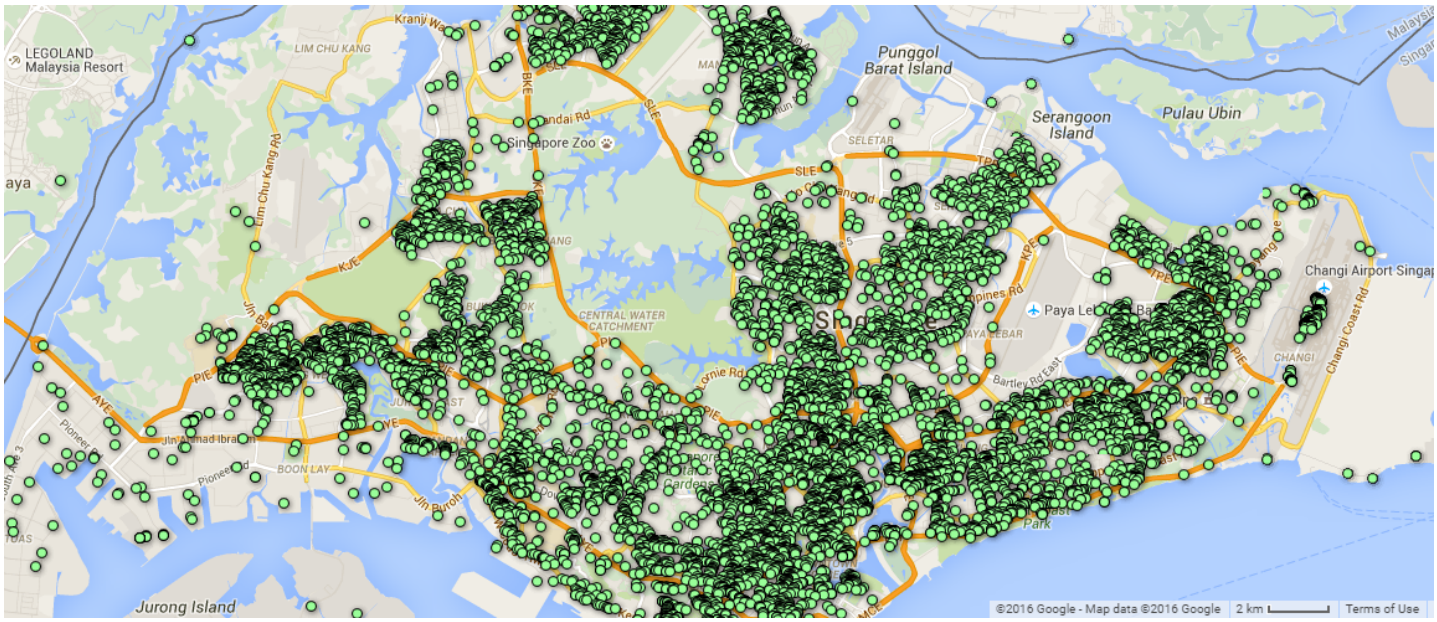

Figure 5.2: Start trip locations of a day from 8:00-9:00 AM

5.2 Temporal Variation

As we plotted the pickup points over a day for every hour, we realized that the data showed temporal variations as well. As a proof of this fact, we analyzed 3 areas (of almost equal areas) over a day (chosen at random) and found that the number of pickup points in the areas kept changing over time. Figure 5.4 shows the two observed areas. Figure 5.5 shows the variation of number of pickup points over time.

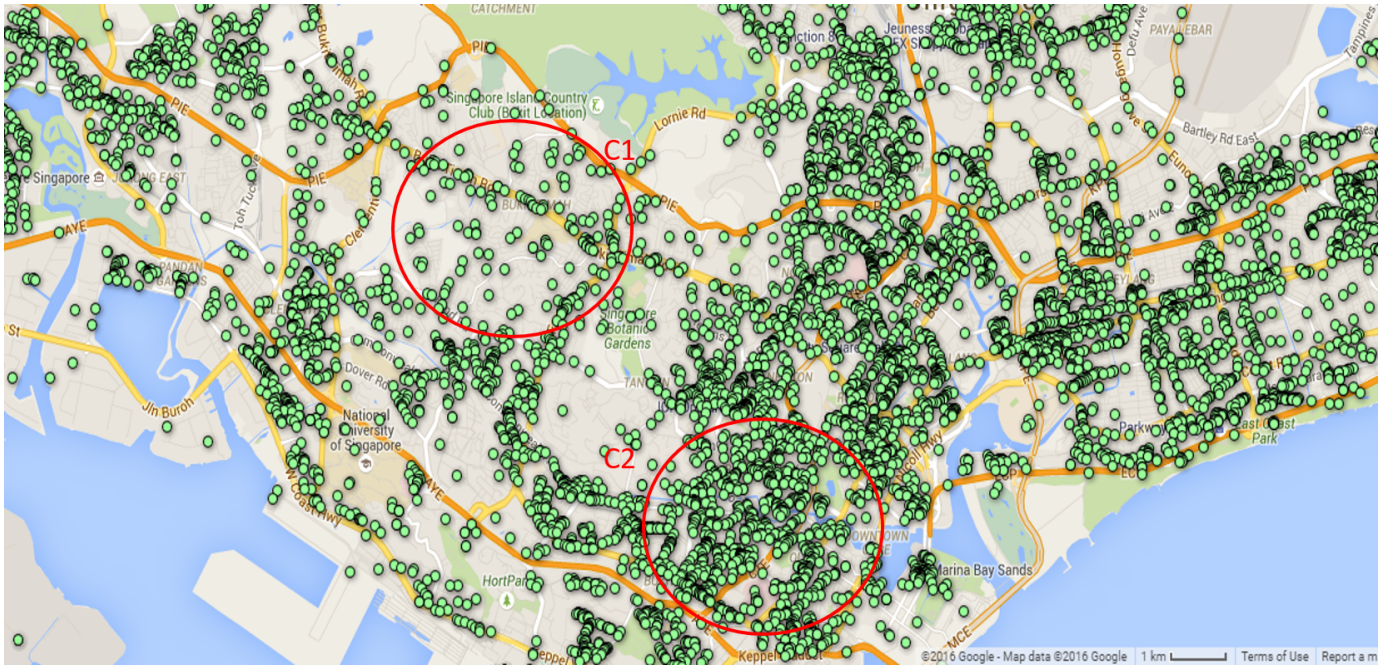

Figure 5.3: Number of pickup points vary over space. Circle C1 has a lower density of number of pickup points with respect to that of circle C2

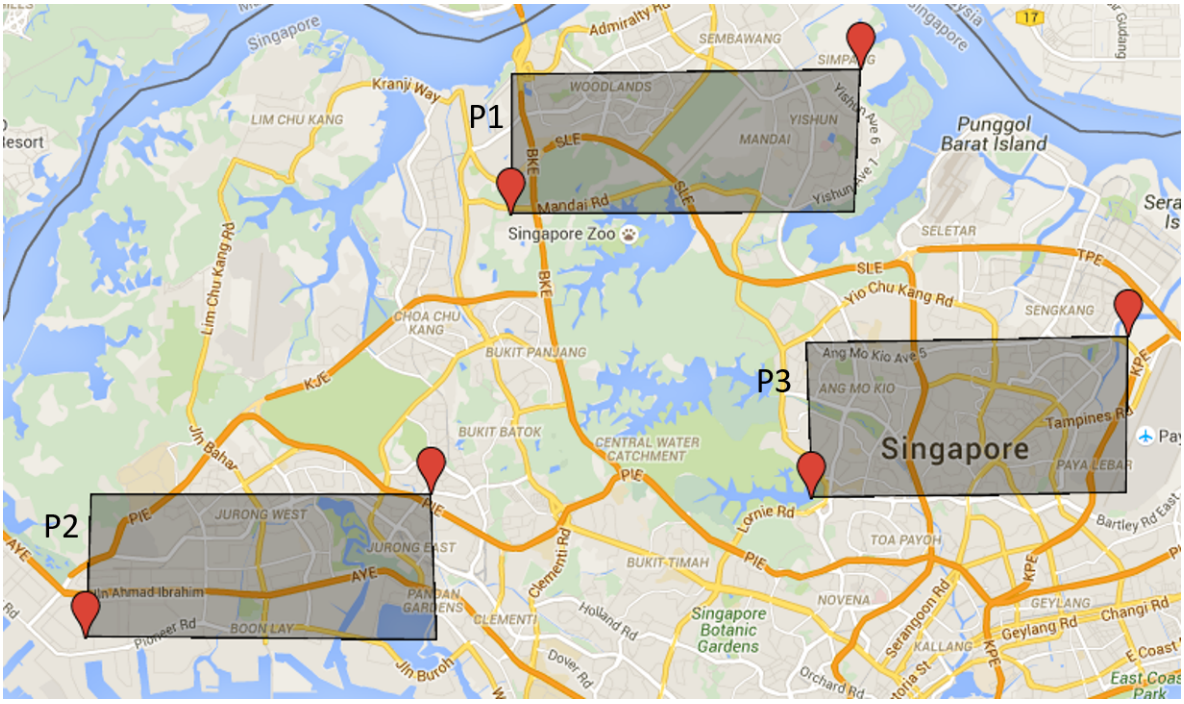

Figure 5.4: Randomly chosen areas to see variation in pickup points over space and time

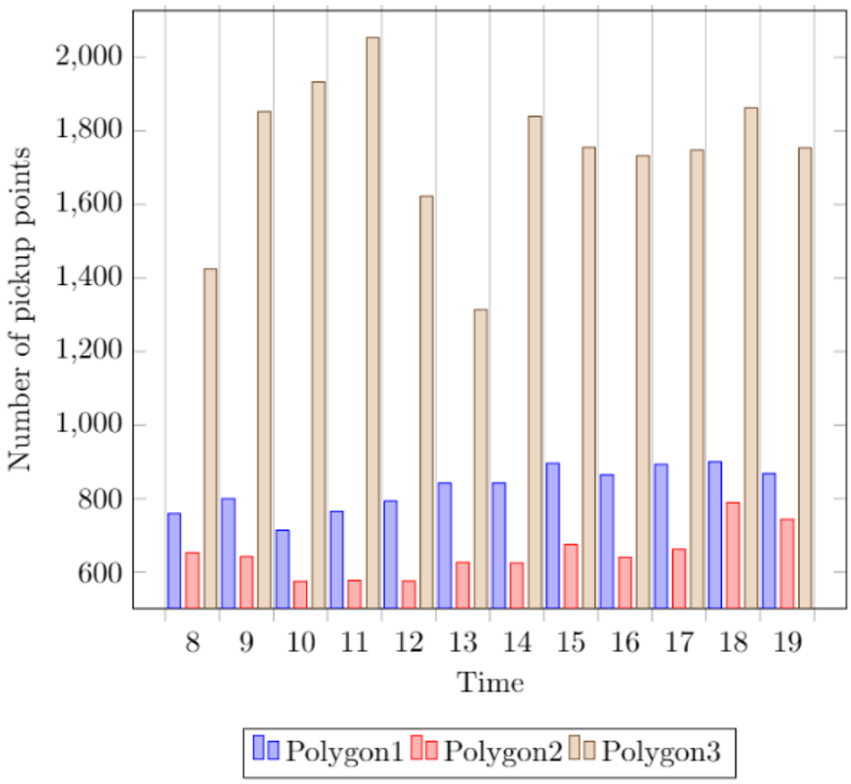

Figure 5.5: Graph showing change in the number of pickup points over time and space. Spatial variation: Polygon 3 has higher number of pickup points than the other 2 polygons. Temporal variation: The number of pickup points change over time and a drastic change can be observed in polygon 3

5.3 Conclusion

So, we see that the data is both spatially and temporally spread across the country. The following conclusions can be formed from these observations.

- As the number of pickup locations varies or, in other words, as taxi demand varies over space, one cannot uniformly distribute the taxi hubs across the city. A recommendation system is required which takes into account the demand of that area.
- As demand of taxis in a fixed area keeps changing over time, the location of taxi hubs cannot stay constant. Different areas should have different taxi hubs at different times of the day.

5.4 Proposed arguments

The conclusions mentioned in section 5.3 helped us propose the following:

- The non-uniform distribution of data suggests that some kind of partitioning technique based clustering is required (refer to section 4.2) to find the set of locations appropriate for taxi stands. This is based on the fact that, given the number of taxi stands one aims to make in a city, partitioning method will be able to find the dominant (higher the number

of pickup points in a region, more dominant that area is) areas and will come up with the locations that could serve as the taxi stands. Each of this taxi stand ts will be such that the distance of a pickup location p, lying in its own cluster, will be less than the distance of p from some other taxi stand ts'. This facilitates a taxi driver standing at the taxi stand to reach to a pickup point (in its own cluster) by traveling minimum possible distance.

- As a taxi runs on the road network of any city, it is needed that the distance between any taxi stand and pickup point is calculated along the road and not just as the Euclidean distance between them. The distance between two points in a connected component (road network) can be calculated using Dijkstra's algorithm. This means that we require a graph of the underlying road network and all the pickup points must exist on it. In this way, we will be able to calculate the distance of any pickup point from any taxi stand also lying on the graph. Thus, three things will be required for the same: pickup locations, road network of Singapore(graph) and a technique to map all the pickup locations onto the graph (all these procedures are described in chapter 6).

Chapter 6

Proposed Solution

This chapter covers the techniques we use to find the appropriate location of taxi stands in Singapore. Figure 6.1 gives a brief overview of the sequence in which the techniques are used. The underlying sections from 6.1 to 6.2 explain how each of these techniques work.

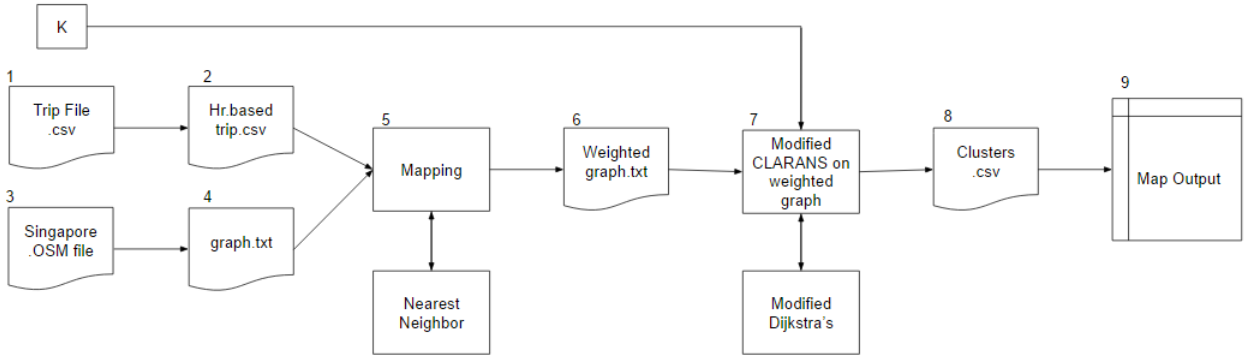

Figure 6.1: Flow chart depicting the sequence in which different techniques have been used

6.1 Inputs

We saw in section 2.1, that two inputs, the number of taxi stands and the pickup locations are required to make taxi stands in a city. Also, we stated in section 5.4, that to find the distance between taxi stands and pickup locations we need a road network. So, other than the two already stated inputs, we also need to have a road network of Singapore which becomes our third input.

This section gives a brief explanation of all the three inputs.

6.1.1 Open Street Maps

Section 4.1 gave an outline of Open Street Maps and their need. Let us look deeply into the structure of OSM files.

OpenStreetMap is a free, editable map of the whole world. Unlike proprietary datasets like Google Map Maker, the OpenStreetMap license allows free access to the full map dataset. This

database is available in form of a .osm file, which is basically an XML file. There are four types of elements in an OSM XML file namely:

- Nodes: A node represents a specific point on the earth's surface defined by its latitude and longitude. It also has a unique node id.e.g. A node could represent your society water tank.
- Ways: A way comprises an ordered set of nodes, which together form a path or a boundary of a closed area essentially linear features like roads and rivers. Different tags define the properties of a way.
- Relations: A relation models the relationship between two or more data elements, may be nodes, ways or some other relations.
- Tags: Tags are map features associated with each of the basic data structures mentioned above that describes a geographic attribute of that feature. It consists of a key-value pair,where the key describes a broad class of features and the value is the specific feature that the key classifies. e.g. highway = residential.

Implementation Challenges

Open Street Maps would store even a park bench or a water well as a node in the XML file. If all such minute details are considered in these maps, one can imagine how intricate their structure would be. ((For instance Singapore has a land area of 719.1 km^2 and the OSM XML file for Singapore has approximately $6.5 * 10^5$ nodes.)

The underlying graph can be retrieved using the techniques mentioned in section 7.1.2. Once it is retrieved, for the usual graph algorithms to run efficiently on such huge graphs, lot of structural properties have to be taken into account in order to develop efficient heuristics.

6.1.2 Trip files

The trip files, as also mentioned in chapter 5, are the .csv files that contain the information about pickup and drop locations of people using taxi services. Table 2.1 gives the format of trip files. These files keep a log of all the trips (in terms of start and drop locations of a taxi service) of the city for all the days of every month.

The information extracted from these files form the basis of our experiments and work. The process of information extraction from trip files is explained in section 7.1.1.

6.1.3 k: Number of taxi stands

As the name itself suggests, number of taxi stands, k, is an integer which refers to the total number of taxi stands that need to be installed in a city. As it depends on the choice of any taxi service company, wishing to install taxi stands, k is taken as an input.

We pre-process the above mentioned inputs (refer to section 7.1) to get the output shown in figure 6.2. We now explain the modified CLARANS algorithm and how it works on this graph.

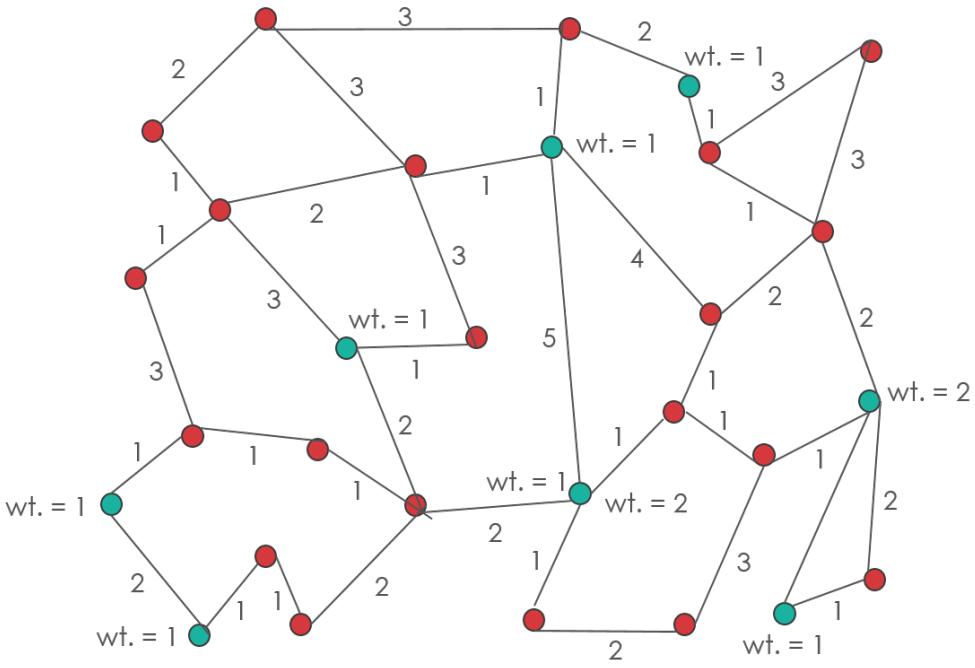

Figure 6.2: All the blue ones are the vertices of the graph which were nearest to one or the other pickup locations mentioned in section 2.1, figure 2.1 and thus have *weight* > 0. These points serve as the pickup points in our algorithm

6.2 Clustering Algorithm: Modified CLARANS

This section gives details of the final clustering algorithm. For the purpose of our work, we are implementing *modified CLARANS algorithm*.

Recall that in section 4.2.4, we saw that CLARANS works by first initializing random seeds and then these are swapped by their *neighbors*, *maxneighbor* number of times. The whole procedure is repeated *numlocaltimes* and the best (the one with minimum cost) set of medoids is returned.

Following are our observations from the same:

- As the initial seeds are being selected randomly, it might happen that, some or all of these seeds, can be amongst the $faraway$ points in the dataset D (the probability of selecting every point as the initial seed is the same and it is $1/n$, where n is the total number of points in D). In other words, the initial seeds might be the ones due to which, cost of the configuration might come very high and thus a large number of swaps will be required to reach to the near optimal solution.
- The initial set of seeds can be swapped with any of the *neighbor* with equal probability. For example, for a set of k medoids, there are $k*(n-k)$ possible neighbors and probability of selecting any of these for swapping is same which is equal to $1/(k*(n-k))$.

Based on the above two observations, we made two modifications in the way CLARANS algorithm works to suit to our problem statement (find locations for taxi stands in the city).

- **Seed Initialization:** As spatial variation exists (refer to chapter 5) in the number of

pickup locations, due advantage should be given to dense areas for selecting initial seeds or in our case initial set of locations. In other words, the areas with more number of pickup points should have a priority for initial locations of taxi stands and also if the number of taxi stands to be made are large, these areas should have more taxi stands than others. The procedure to do so is explained in section 6.2.1.

- **Swapping medoids with non-medoids:** While swapping the initial set of locations with *neighbor* locations, one does not need to check from the set of all possible *neighbors*. As the initial seeds are being taken based on dense areas, an initial location can be swapped with a *nearby* (explained later) location to check for a lower cost. Details for the same are written in section 6.2.2.

6.2.1 Seed Initialization:

As mentioned before, we want our initial locations of taxi stands to be chosen based on number of pickup points in a region. We have developed an algorithm for the same. The flow chart for the same is given in figure 6.3. The seed initialization algorithm is a divide and conquer algorithm which uses QuadTree to divide a given land area. Let us understand this with the help of our example city. The step-by-step process is given below.

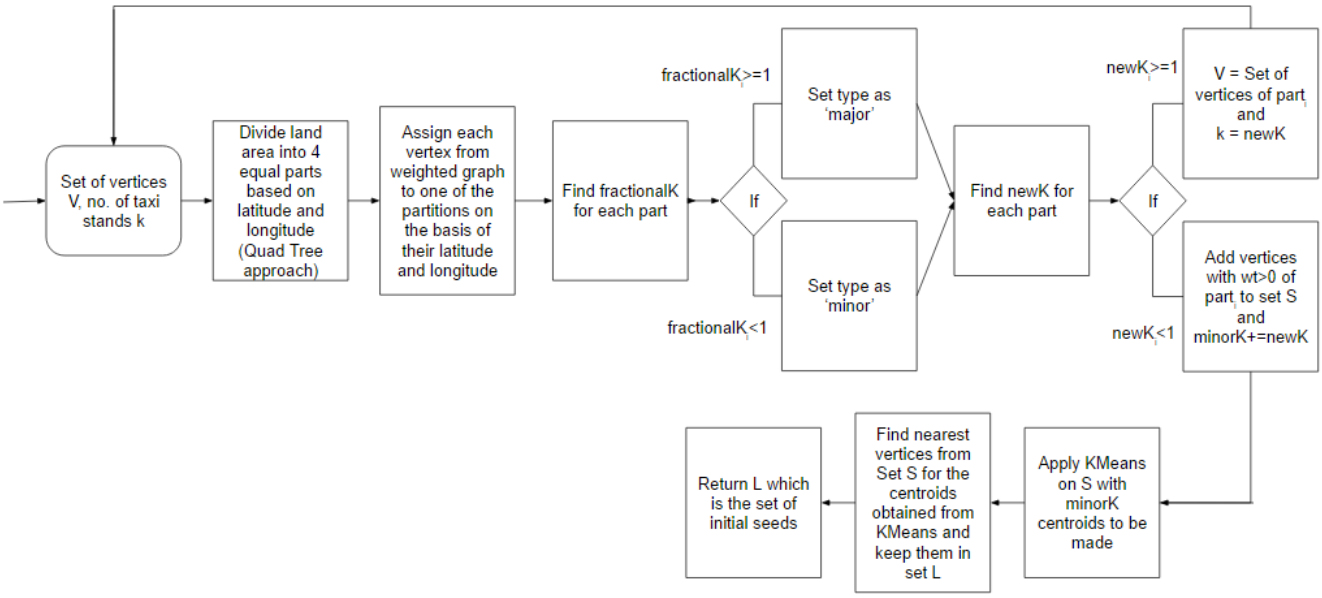

Figure 6.3: Flow chart depicting process of seed initialization

1. The city is divided into 4 equal parts (based on latitude and longitude) as shown in figure 6.4.

2. Each part gets a fraction of total taxi stands (k) to be made. The distribution of k is such that the part with more number of pickup points gets priority. Thus, we first find out what fraction of total taxi stands that each partition should get. So, $fractionalK_i$ = (number of pickup points in $part_i$/total number of pickup points)*k. For example, if $k = 3$, each part will get $fractionalK_i$ as 0.125, 0, 0.75 and 0.125 (figure 6.5).

3. All the parts with $fractionalK_i < 1$ are called as *minor* partitions while others are called *major* partitions (figure 6.6)

4. Again distribute k over partitions greedily based on their $fractionalK_i$. Let's call it $newK$. Note that $fractionalK$ is required only to determine the type of a partition. Once it is done, we want to have an integer value with each partition. There are two steps for this: For step1, each part gets a value $newK$ such that for each part i, $newK_i$ = (int)$fractionalK_i$. So, each part gets $newK_i$ as 1,0,0,1, in this step. Step 2 occurs only if $k > \sum_{i=1}^{4} newK_i$. So, for our example as $\sum_{i=1}^{4} newK_i$ is 2 and k=3, round 2 does happen. Round2 sorts ($fractionalK_i$ - $newK_i$) value of each partition in decreasing order and increases their $newK_i$ till the time $k > \sum_{i=1}^{4} newK_i$. So, here ($fractionalK_i$ - $newK_i$) value for each part0, part1, part2 and part3 is 0.125, 0, 0.75 and 0.125. These are then sorted in decreasing order as 0.75, 0.125, 0.125 and 0. Going by the algorithm, for part2, $newK_2$ will be changed to 1. As k is now 0, each of the parts now get new_K as 1,0,1,1.See figure 6.7

5. All the 'major' partitions are again divided using QuadTree approach with each of them having k=1 ($newK_1$ and $newK_3$ both are 1) as shown in figure 6.8.

6. For all the *minor* partitions, $KMeans$ is applied on all of their pickup points. The number of centroids to be made is the sum of $newK$ of minor partitions i.e. #centroids=($newK_1$ + $newK_2$). For the centroids found using KMeans, their nearest pickup points, from the graph, are initialized as the initial seeds for our modified CLARANS algorithm. See figure 6.9 for details.

7. Figures 6.10 to 6.13 show further division of part0 and part3 to finally get other two initial seeds (total 3 taxi stand locations have to be initialized). So, after the whole process, we finally get the initial seeds for our modified CLARANS algorithm. Note that, the QuadTree structure is only used to divide search space and initialize seeds. Once it is done, and we have the locations of our initial seeds, no more division exists. We switch back to our weighted graph with these locations serving as the seeds as shown in figure 6.14

The pseudo-code for the whole process of seed initialization is given in 6.9

Algorithm 6.9 Initialize Seeds

Input:

- D: Vertex Set V of graph G
- Integer $k \leftarrow$ #taxi stands

Output: A set of k initial locations

1: **procedure** INITIAL SEEDS
2: L = getInitialSeeds(V, k)
3: **end procedure**

Algorithm 6.10 Gives the set of initial locations of taxi stands

```
1: function getInitialSeeds(V, k)
2:     List initialSeeds = {}
3:     List S = {}
4:     numCentroids = 0
5:     List parts ← getParts(V) //using QuadTree approach
6:     double[] fractionalK = getFractionalK(k)
7:     for Each value i in fractionalK do
8:         if fractionalK[i] ≥ 1 then
9:             part[i].type = 'major'
10:        else
11:            part[i].type = 'minor'
12:        end if
13:    end for
14:    int[] newK = getNewK(fractionalK, k)
15:    for Each part j in parts do
16:        if part[j].type=='major' then
17:            initialSeeds.add(getInitialSeeds(part[j].vertices, part[j].newK))
18:        else
19:            S.add(part[j].vertices)
20:            numCentroids = numCentroids + part[j].newK
21:        end if
22:    end for
23:    initialSeeds.add(KMeans(S, numCentroids))
24:    return initialSeeds
25: end function
```

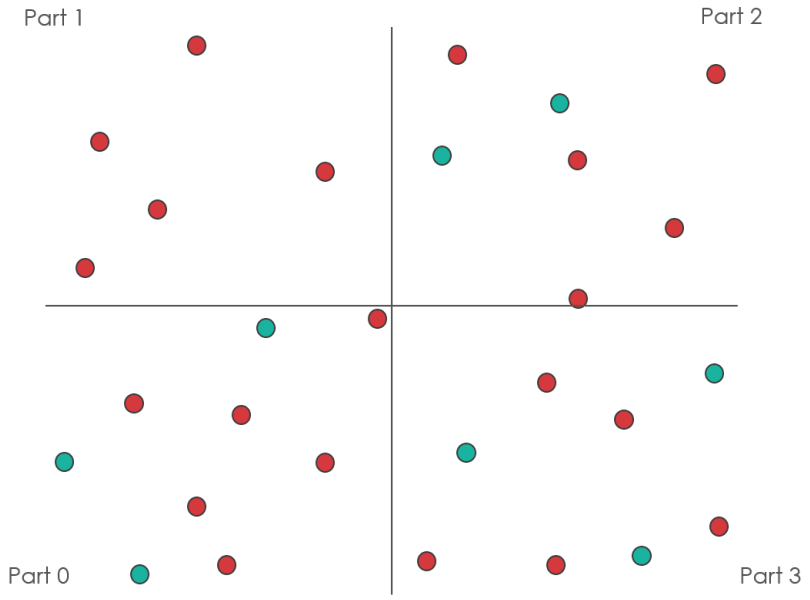

Figure 6.4: The vertices of weighted graph can be visualized as a set of locations in space. Quad Tree distributes all the points into 4 parts depending on their latitude and longitude

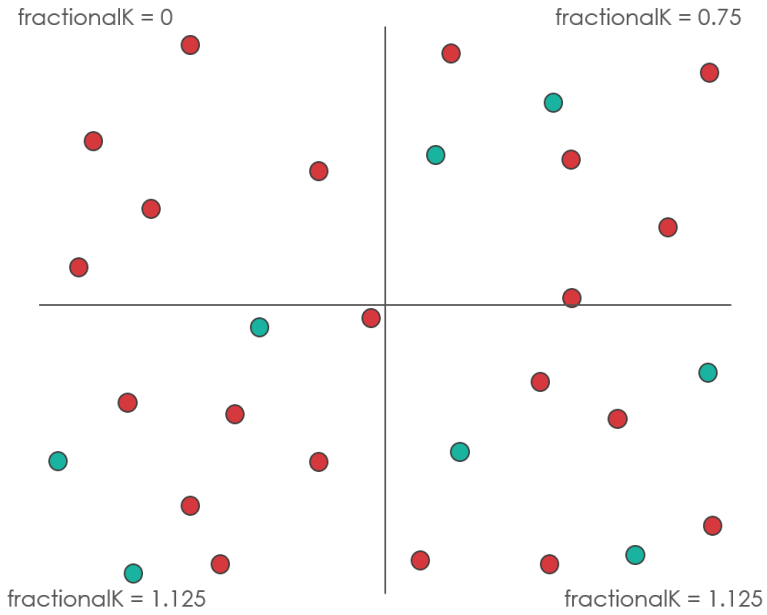

Figure 6.5: fractionalK for a $part_i$ = (number of pickup points in $part_i$/total number of pickup points)*k. For part0, fractionalK = (3/8)*3 as part0 has 3 pickup points with total number of pickup points in the graph being 8 and total number of taxi stands to be made are 3. Similarly fractionalK for other parts is calculated.

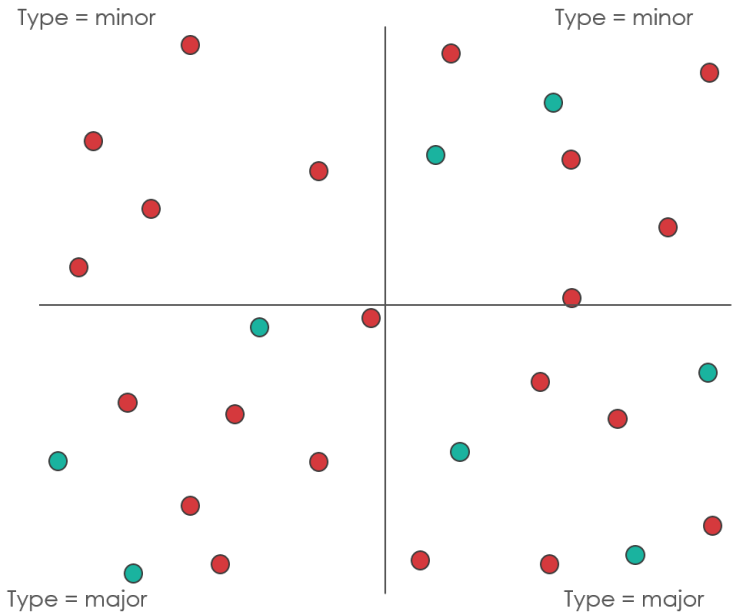

Figure 6.6: As fractionalK for part1 and part2 are less than 1, they are *minor* partitions and on the other hand part0 and part3 are *major* partitions

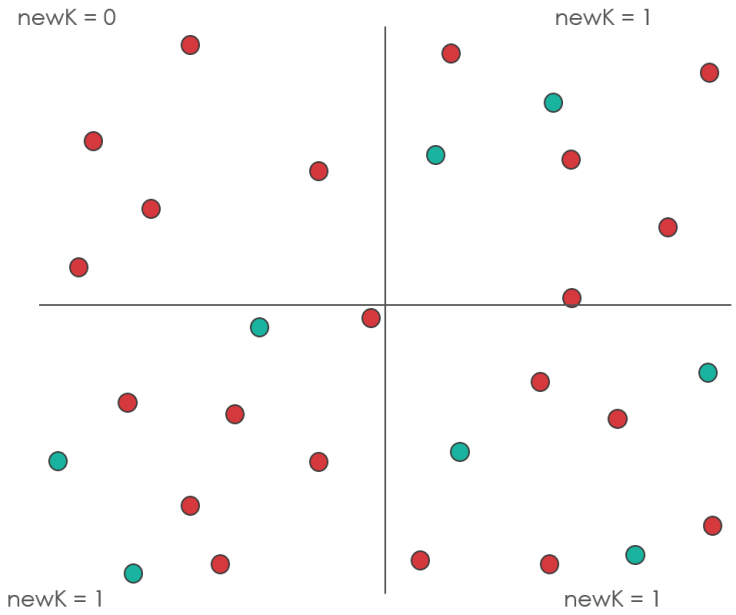

Figure 6.7: The 2 step process assigns a newK to each partition

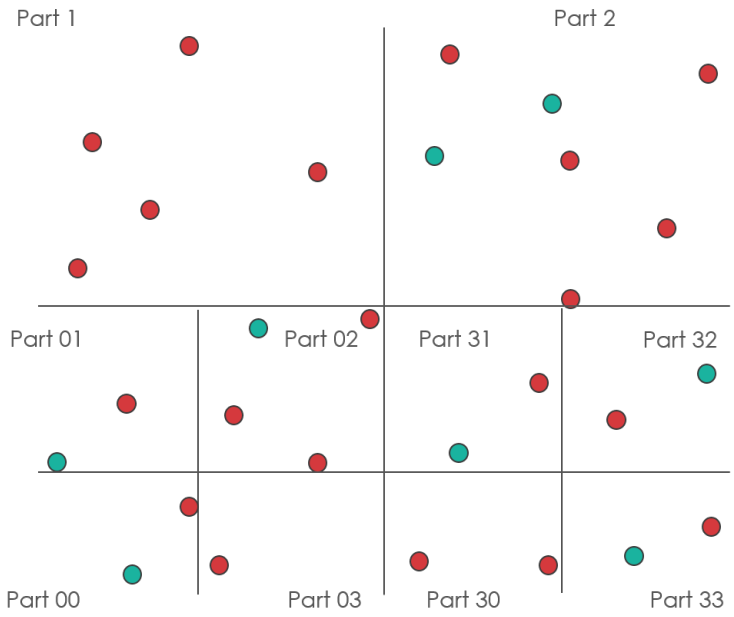

Figure 6.8: As part0 and part3 are major, they undergo further division till each of their sub-parts become 'minor' partitions

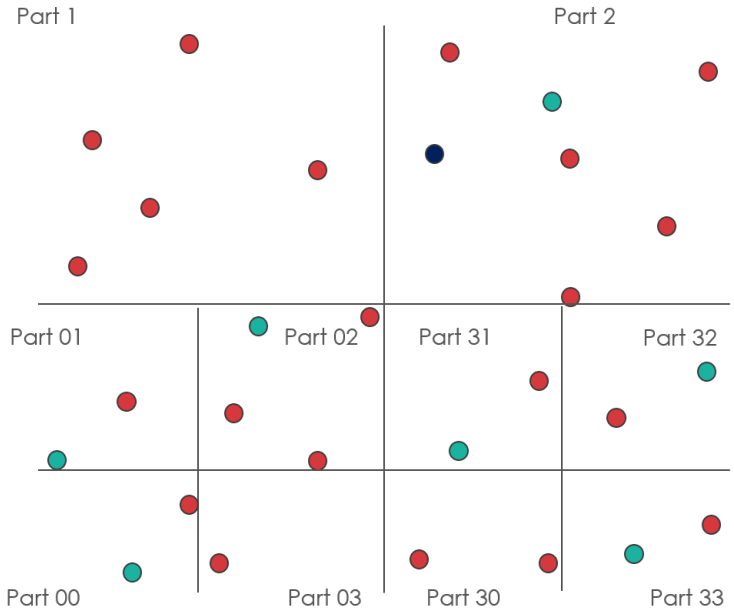

Figure 6.9: KMeans is applied on pickup locations of part1 and part2. As sum of their newK = 1, only 1 centroid has to be found out. For the found centroid, nearest pickup point is initialized as seed. Black dot in the figure is now one of the initial seeds

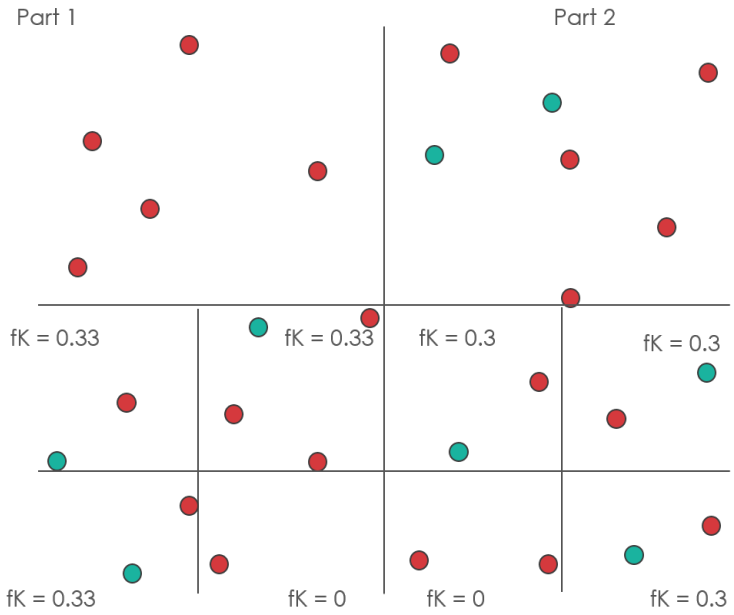

Figure 6.10: Part0 and part1 are again divided to get fractionalK values.

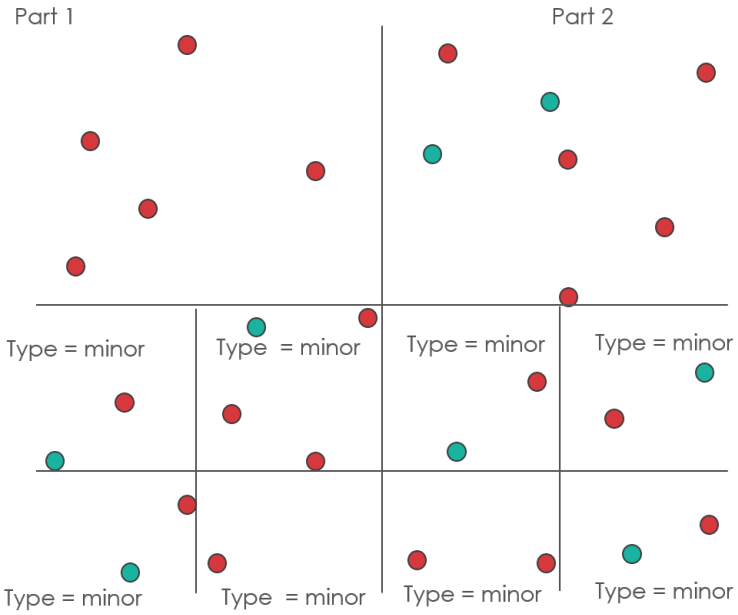

Figure 6.11: Minor and major types are assigned to each part based on their fractionalK values. As for all the sub-parts of part0 and part3, fractionalK is less than 1, all of them are of type minor.

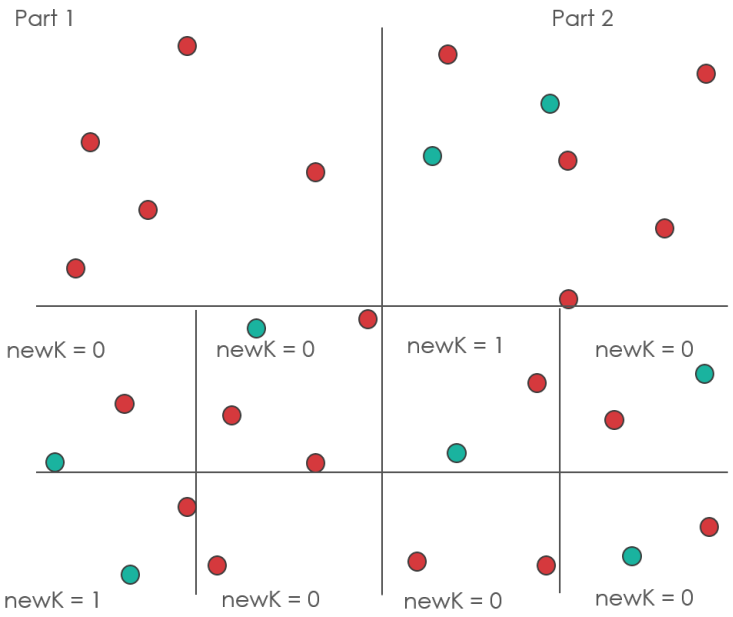

Figure 6.12: newK is assigned to each sub-part of part0 and part3 according to our two step greedy algorithm

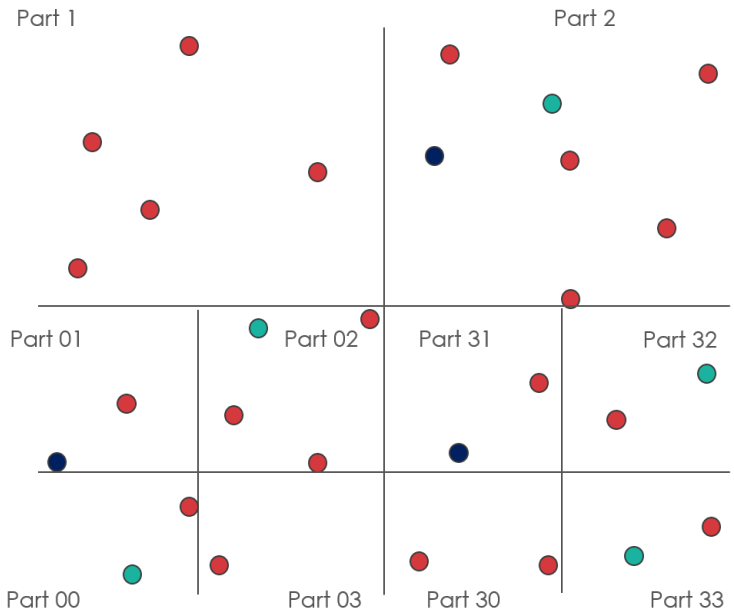

Figure 6.13: KMeans is applied on pickup points of part0 and part3 (individually). As sum of the newK of each sub-part of part0 is 1, 1 centroid is found for part0 and similarly for part3. Nearest pickup point for the found centroid is initialized as the seed. Black nodes in part0 and part3 are the initial seeds.

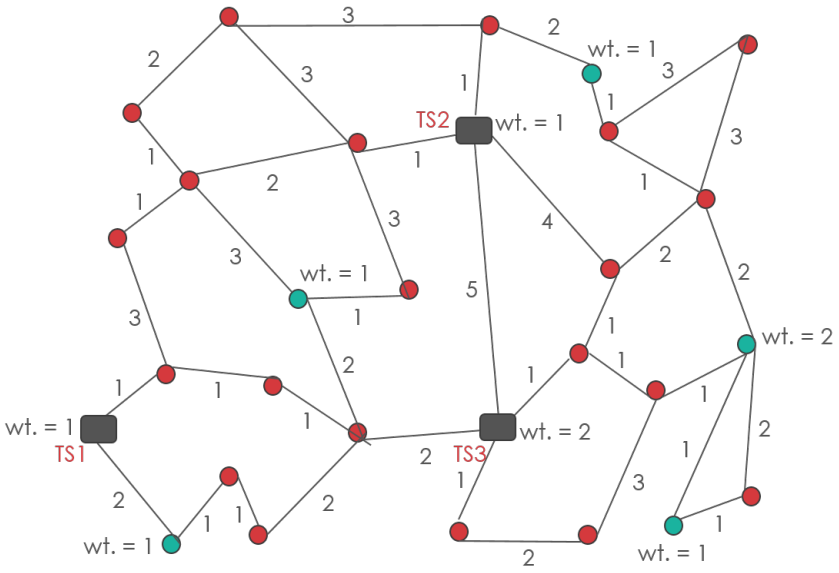

Figure 6.14: The black nodes in the graph are the initialized seeds

6.2.2 Swapping medoids with non-medoids

The other change which we have done in the working of CLARANS algorithm is the way a medoid is swapped with other non-medoids. In the traditional way, a medoid m from a set of medoids M of size k, will be chosen randomly and then it will be swapped by any one of the non-medoids from the dataset. This means that if a dataset D is of size n then m can be swapped with any one of the points in D with equal probability of $1/*(n-k)$. For the purpose of our work, as we have already taken the initial location of taxi stands according to their demands in different areas, we can argue that there is no need to swap a taxi location of an area with some random other point on the graph. Every taxi location should be swapped with a 'nearby' point to check for a lower cost (total distance to be traveled to reach corresponding taxi stand from a set of pickup points around it). Let's see this on our example city. The initial taxi locations obtained from section 6.2.1 are the black rectangles as shown in figure 6.15. One can say that in general, the taxi demands for areas 'nearby' TS1, will be taken care by TS1 rather than TS2 or TS3. Thus, while swapping taxi location of TS1 with some other location on the road network, one can just check the locations around TS1 lying in radius r. All the points lying in radius r around a taxi stand are considered as 'nearby'. Figure 6.16 explains this in more detail. Radius r is chosen based on experiments and it is obvious that larger the value of r, the more area is being taken into consideration for swapping a taxi location TS. If r is taken as ∞, any point from the road network (other than the ones which are already the taxi stands), can be chosen to swap existing taxi stand, just as traditional CLARANS works.

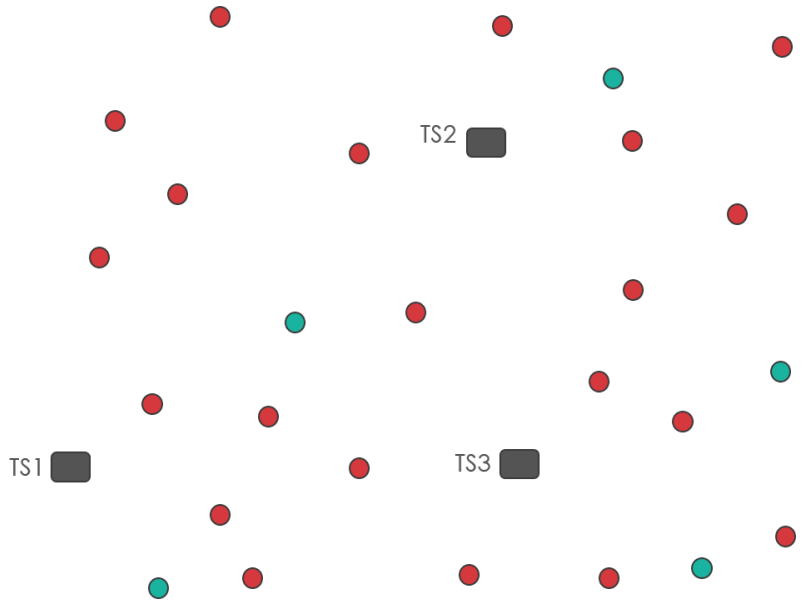

Figure 6.15: TS1, TS2, TS3 are the initial locations of taxi stands

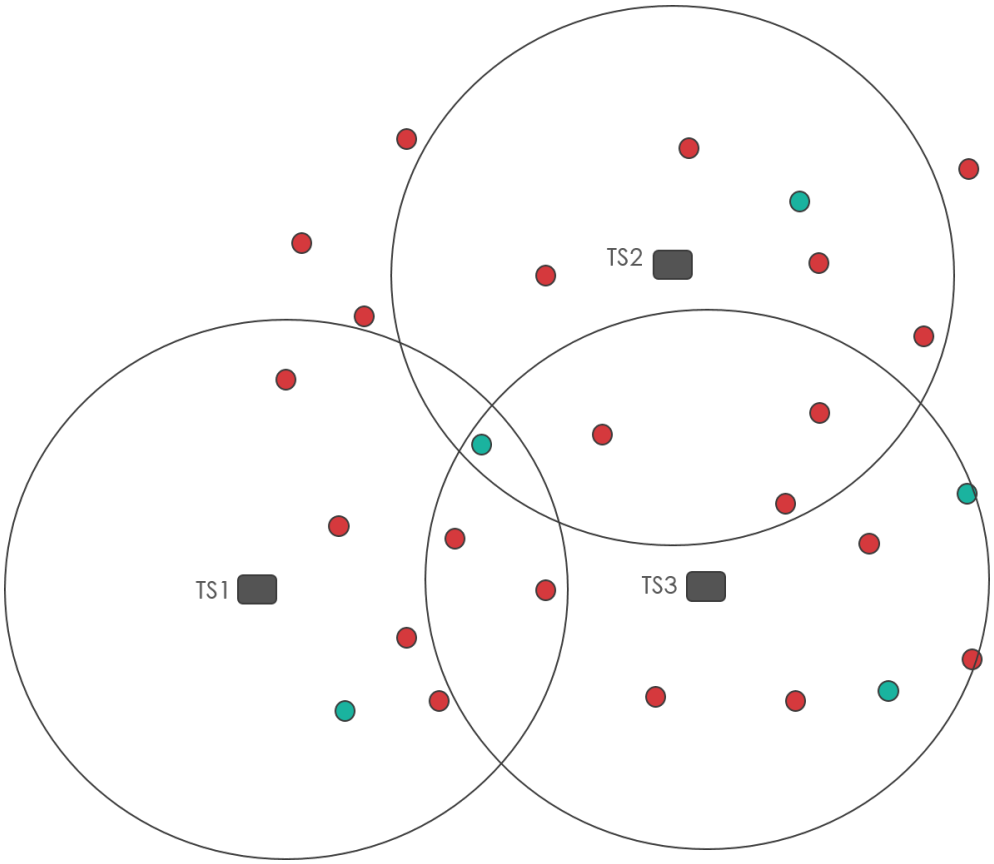

Figure 6.16: For each taxi stand, only the points around it in radius r, are used for swapping. We ensure that radius r is not very small and that different areas are overlapping

6.2.3 Modified CLARANS

The overall procedure of modified CLARANS is similar to traditional CLARANS algorithm other than the two suggested changes mentioned in sections 6.2.1 and 6.2.2. The whole procedure is given below.

1. In input we have the weighted graph G (with seeds initialized in it (figure 6.14)), *numlocal* and *maxneighbor*. *numlocal* denotes number of times we want to run modified CLARANS algorithm while *maxneighbo*r is the number of times medoids will be swapped with non-medoids.
2. Initialize variable i and j with 0. When i becomes equal to *numlocal* and j becomes equal to *maxneighbor*, algorithm will terminate.
3. Randomly select any one of the seed locations from the graph. Let this be m.
4. For every pickup point, calculate its distance from its nearest seed and then multiply it with the weight of the pickup point. Add all such distances and let it be cost θ.
5. Select any non-medoid vertex from the graph based on its weight. Higher the wt., higher is the probability of getting selected.
6. Repeat step 5 till you select a vertex vt which lies inside radius 'r' of m .
7. Move taxi stand to vt and recalculate the distance of all pickup points from new set of taxi locations in the same way as mentioned in step 4. Let the new cost be γ.
8. If $\gamma < \theta$, set j=0 and start from step 3 with new set of taxi locations.
9. Otherwise, do not make the corresponding swap, increment j by 1 and move to step 3.
10. When j becomes *maxneighbor*, increment i by 1.

Now let us understand the algorithm with the help of our example city. Let us assume, *numlocal* = 1 and *maxneighbor*=3. i and j are initialized to 0. Figures 6.17 to 6.41, explain the whole algorithm in detail.

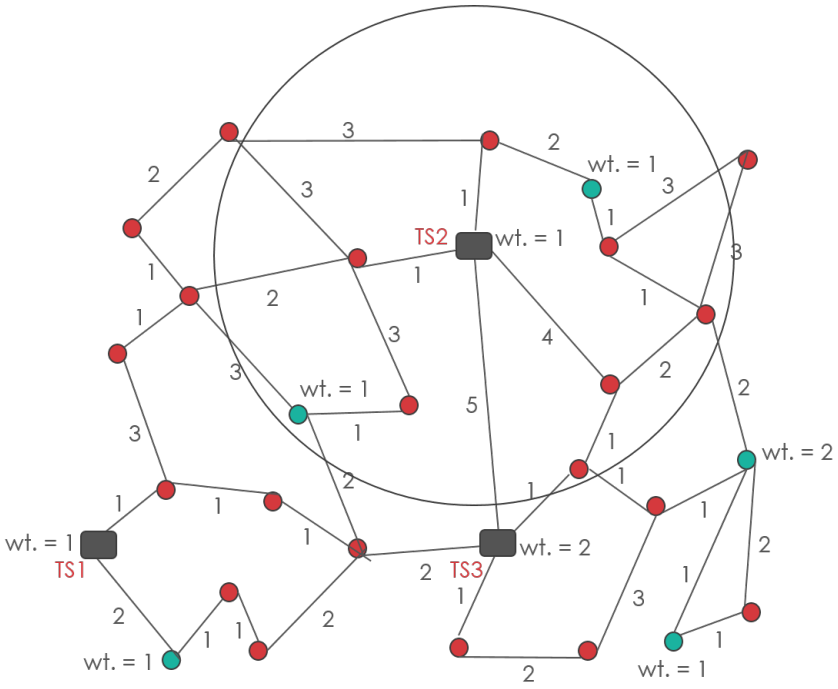

Figure 6.17: Let TS2 be the randomly chosen seed location as mentioned in step 3. Distance of all the pickup locations is then calculated from the set of taxi locations (step 4). The cost θ comes out to be 20 units. See section 6.2.4 for details of distance calculation.

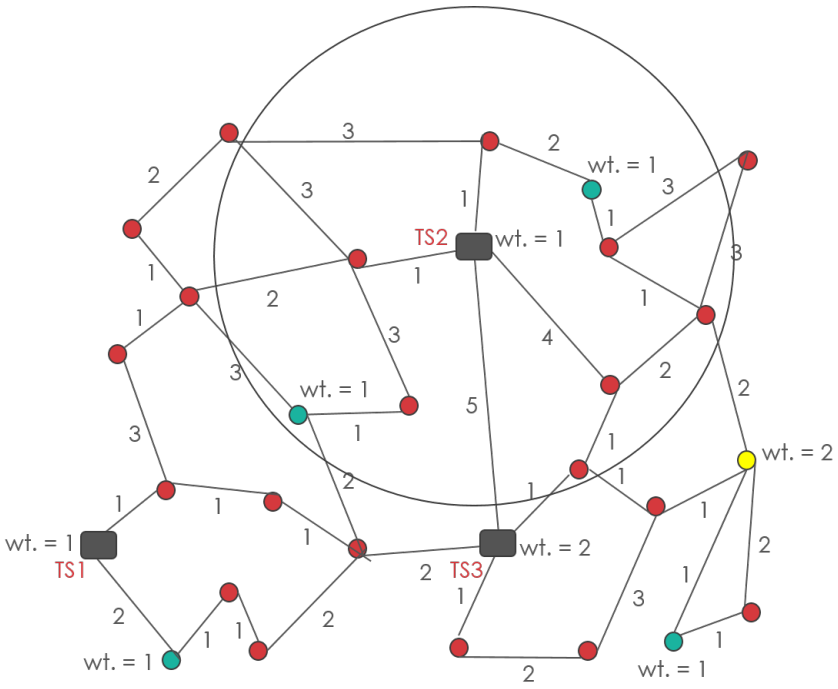

Figure 6.18: Yellow node is chosen randomly from the non-medoids to be swapped with TS2 but as it lies outside radius 'r', it is rejected (step 5)

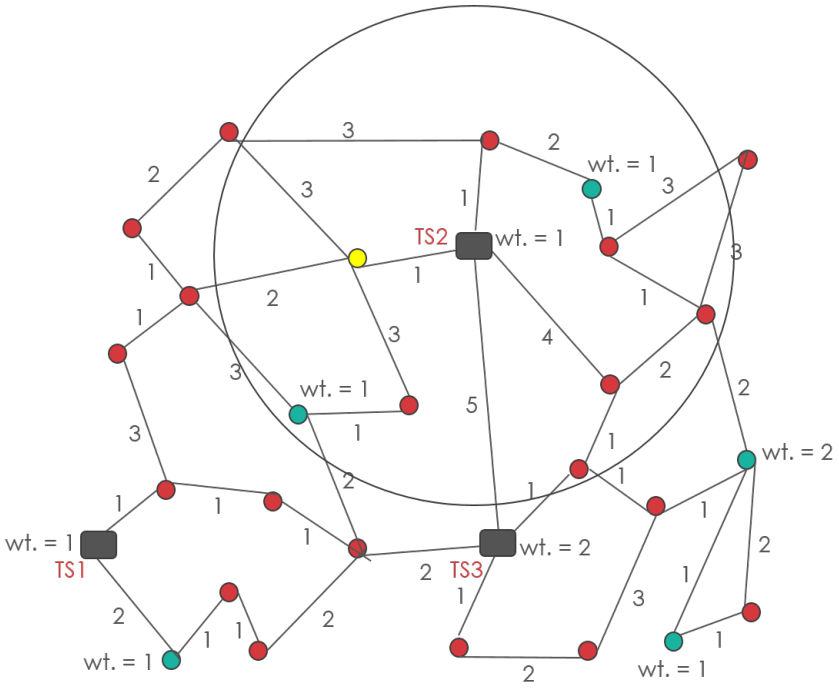

Figure 6.19: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

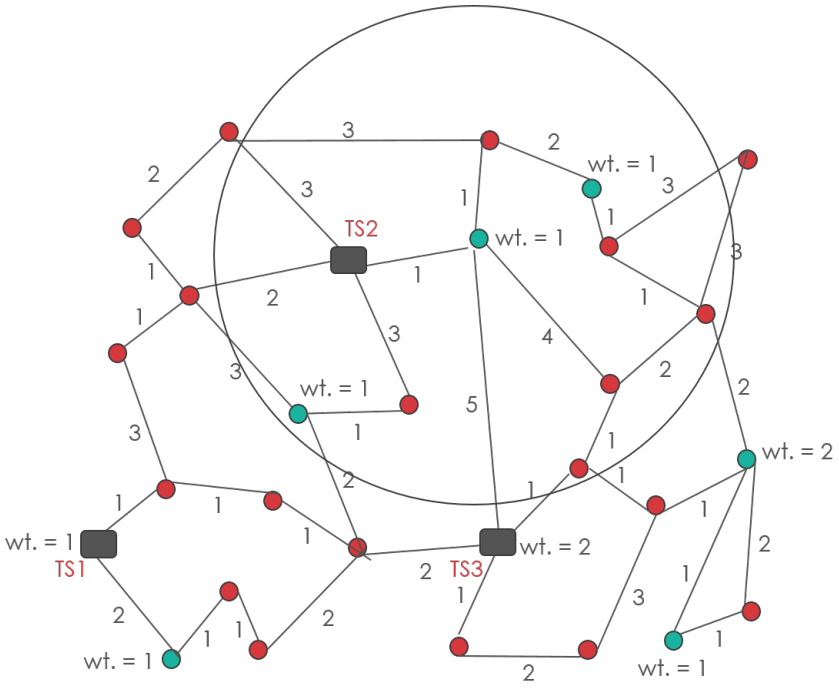

Figure 6.20: TS2 is moved to yellow vertex location of figure 6.19 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 21 units which is greater than the original cost θ. So, the swap is rejected, j is incremented by 1 (now j=1) and we move back to step 3 as shown in figure 6.21

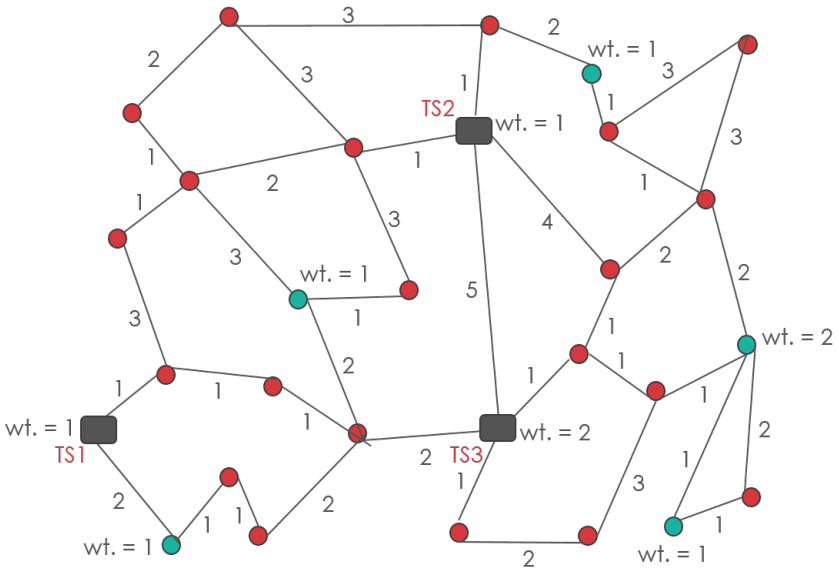

Figure 6.21: After removing the swap, we are back to our previous taxi locations

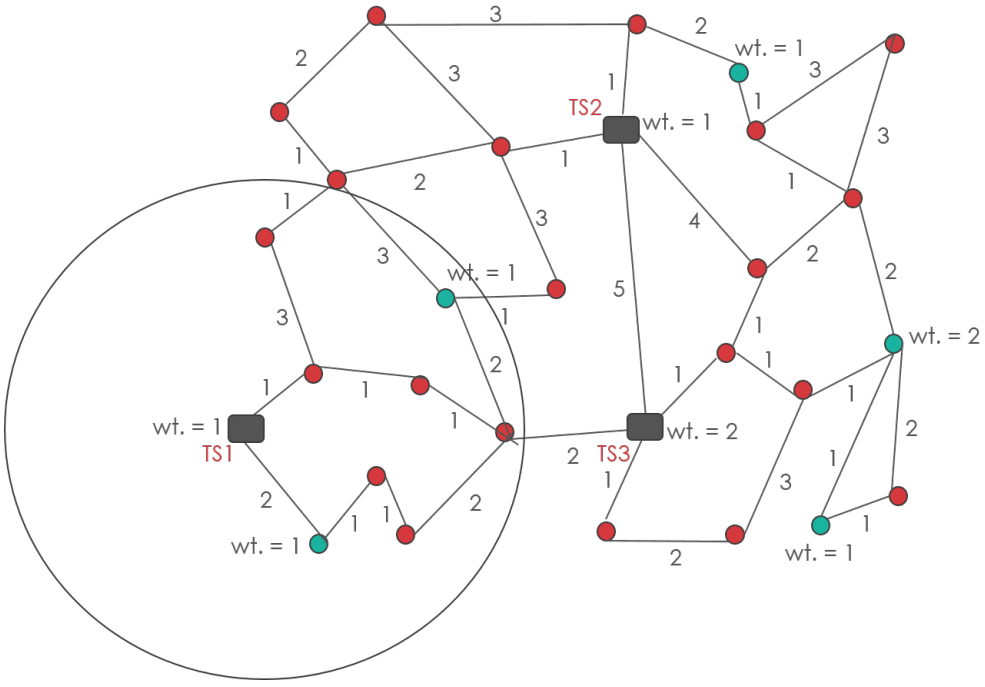

Figure 6.22: Moving to step 3, we again randomly choose one of the taxi stands. Let it be TS1 this time. Distance of all the pickup locations is calculated from the set of taxi locations (step 4). The cost θ comes out to be 20 units.

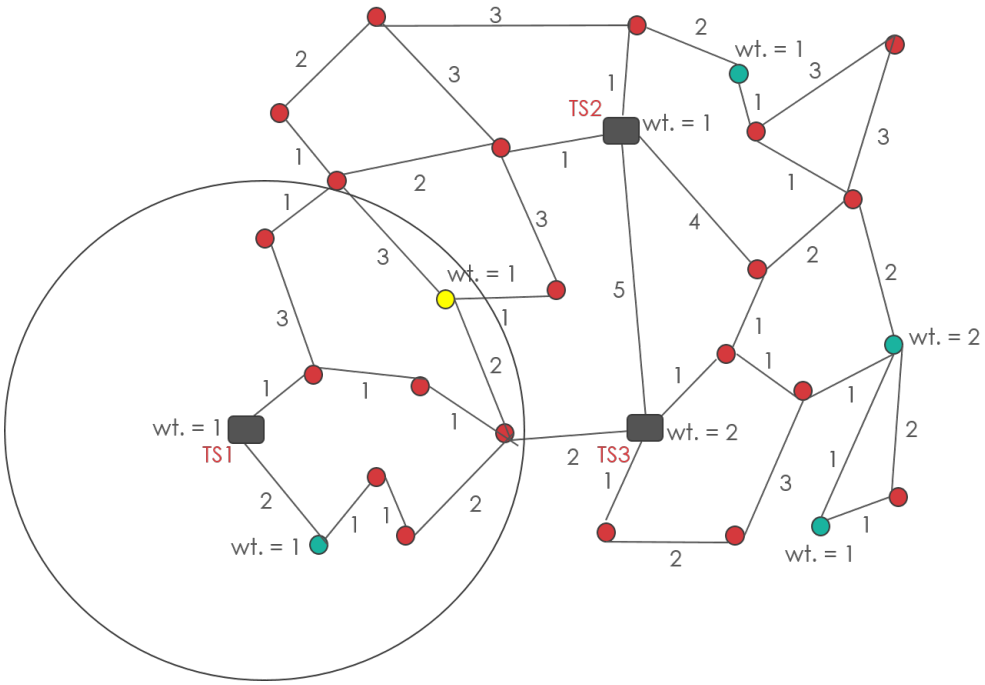

Figure 6.23: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

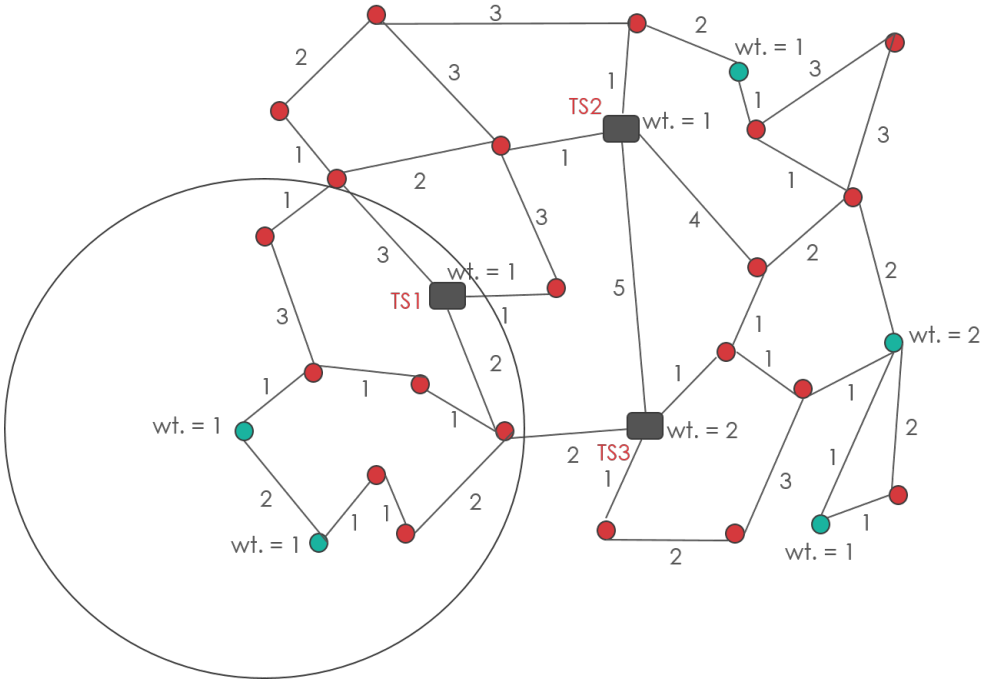

Figure 6.24: TS1 is moved to yellow vertex location of figure 6.23 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 24 units which is greater than the original cost θ. So, the swap is rejected, j is incremented by 1 (now j=2) and we move back to step 3 as shown in figure 6.25

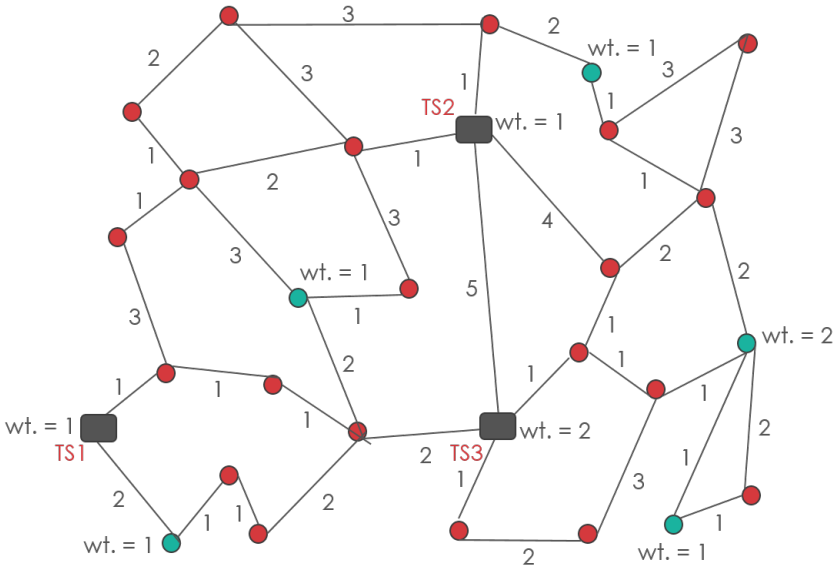

Figure 6.25: After removing the swap, we are back to our previous taxi locations

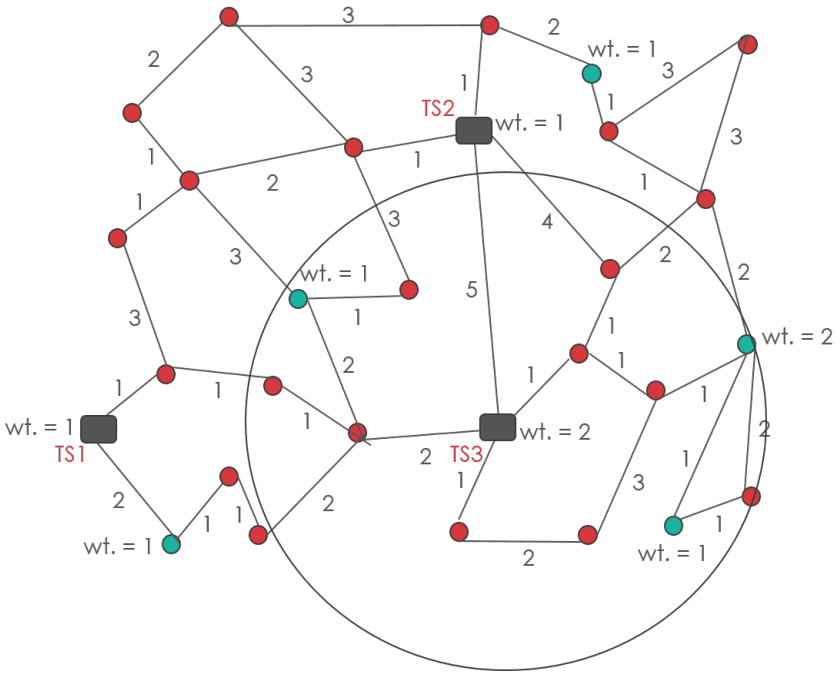

Figure 6.26: Moving to step 3, we again randomly choose one of the taxi stands. Let it be TS3 this time. Distance of all the pickup locations is calculated from the set of taxi locations (step 4). The cost θ comes out to be 20 units.

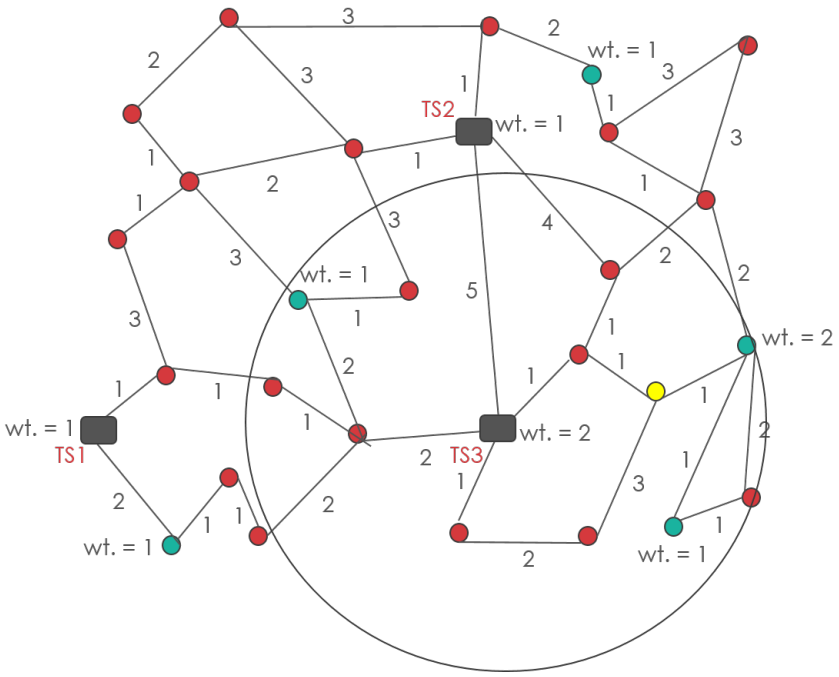

Figure 6.27: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

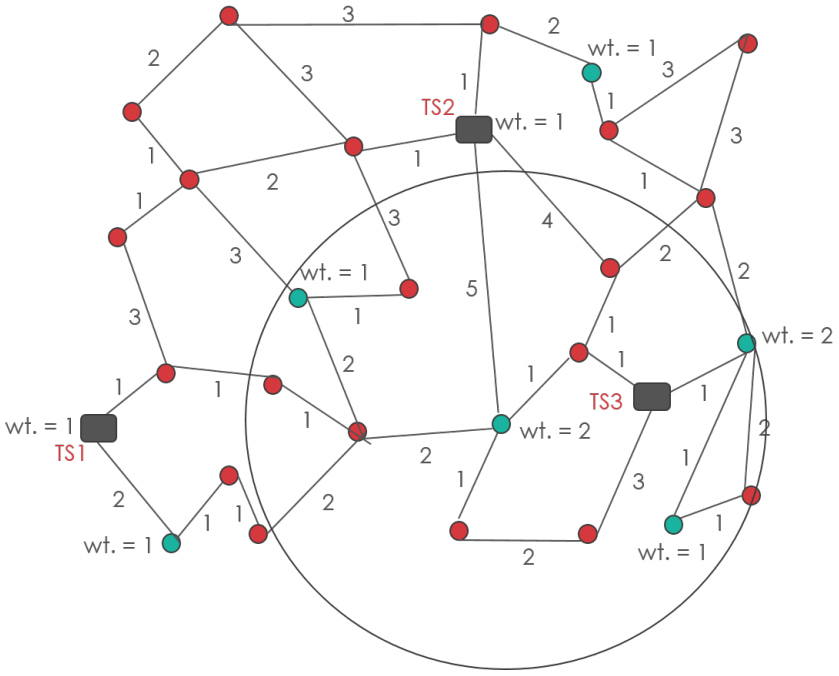

Figure 6.28: TS3 is moved to yellow vertex location of figure 6.27 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 19 units which is less than cost θ. So, the swap is accepted, j is set to 0 and we move back to step 3 with the new set of taxi locations as shown in figure 6.29

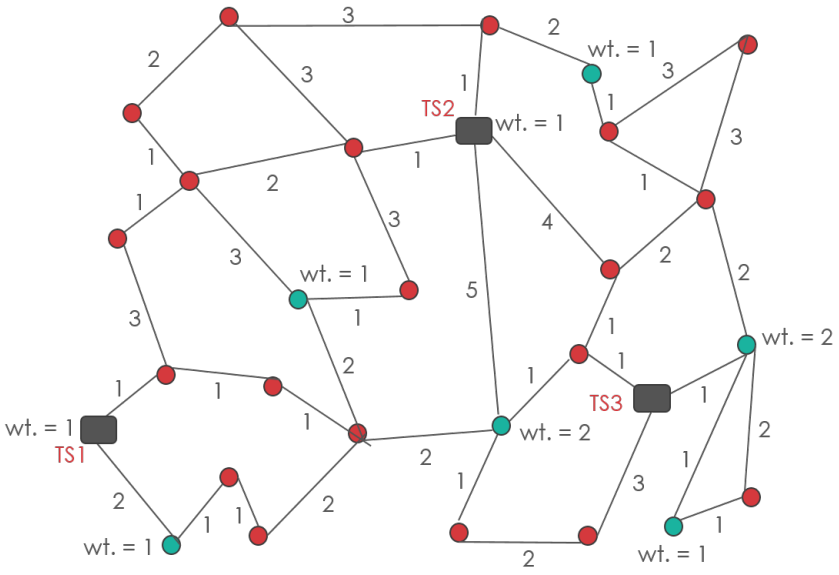

Figure 6.29: The new set of taxi locations in which TS3 has moved to a new point from the original initialized location

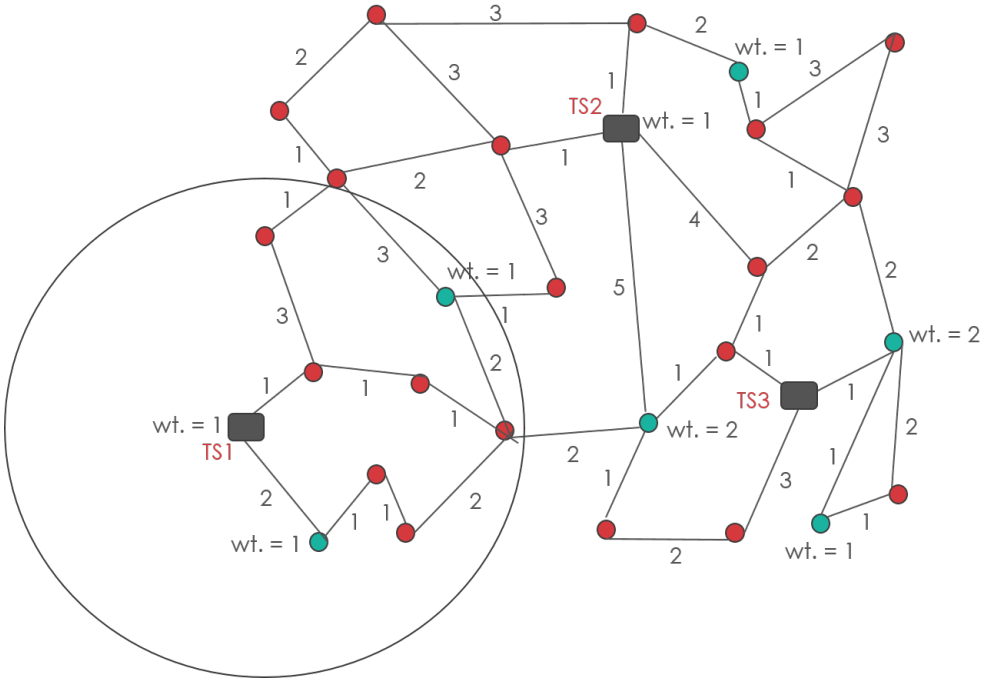

Figure 6.30: Moving to step 3, we again randomly choose one of the taxi stands. Let it be TS1 this time. Distance of all the pickup locations is calculated from the set of taxi locations (step 4). The cost θ comes out to be 19 units.

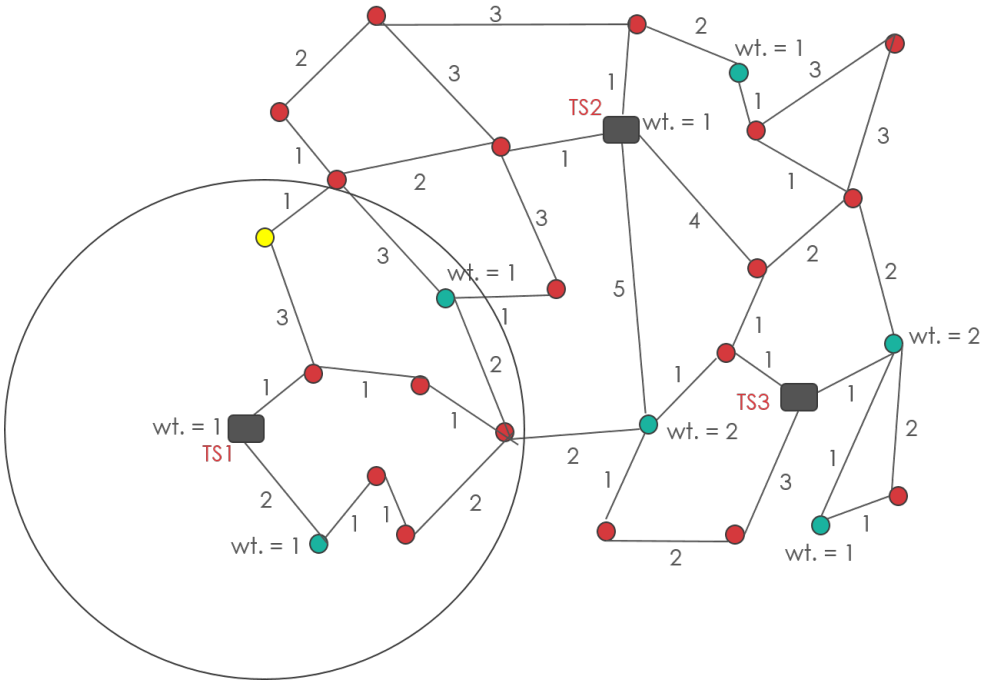

Figure 6.31: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

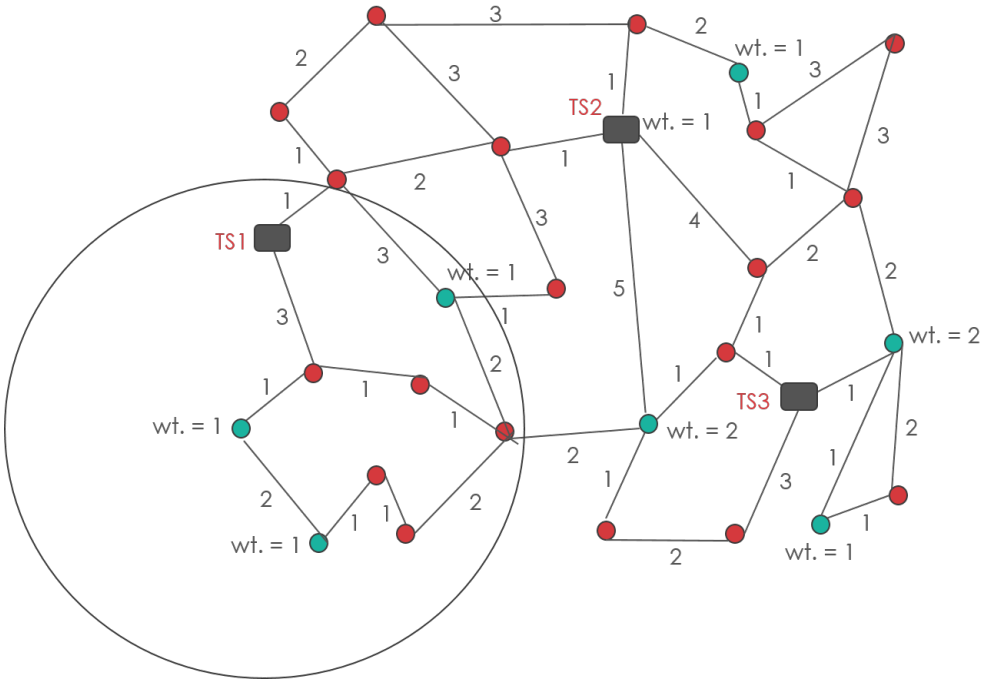

Figure 6.32: TS1 is moved to yellow vertex location of figure 6.31 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 28 units. So, the swap is rejected, j is incremented by 1 (now j=1) and we move back to step 3 as shown in figure 6.33

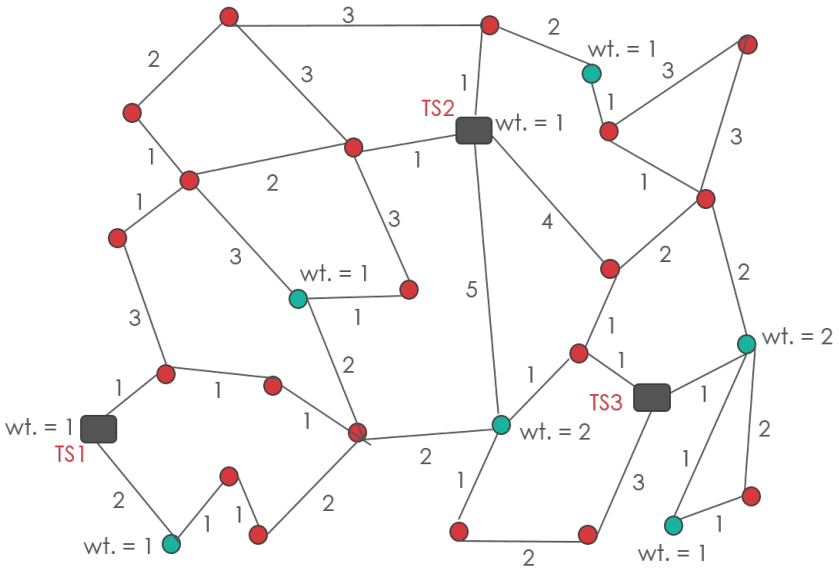

Figure 6.33: After removing the swap, we are back to our previous taxi locations

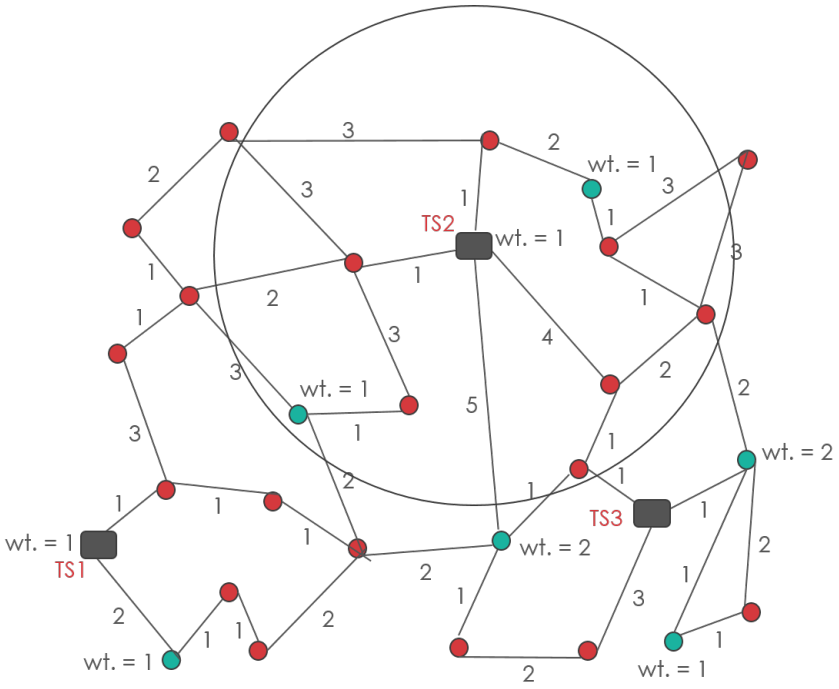

Figure 6.34: Moving to step 3, we again randomly choose one of the taxi stands. Let it be TS2 this time. Distance of all the pickup locations is calculated from the set of taxi locations (step 4). The cost θ comes out to be 19 units.

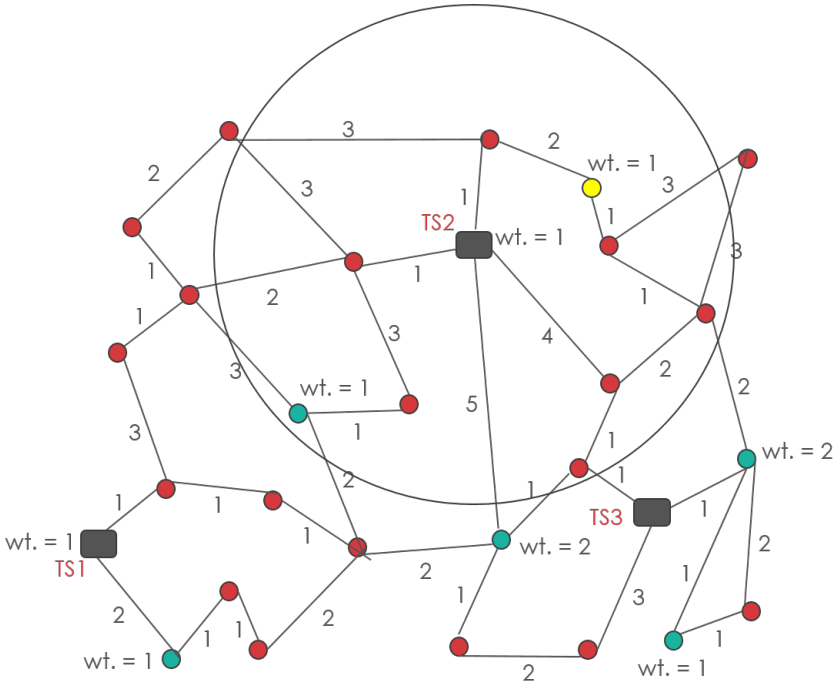

Figure 6.35: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

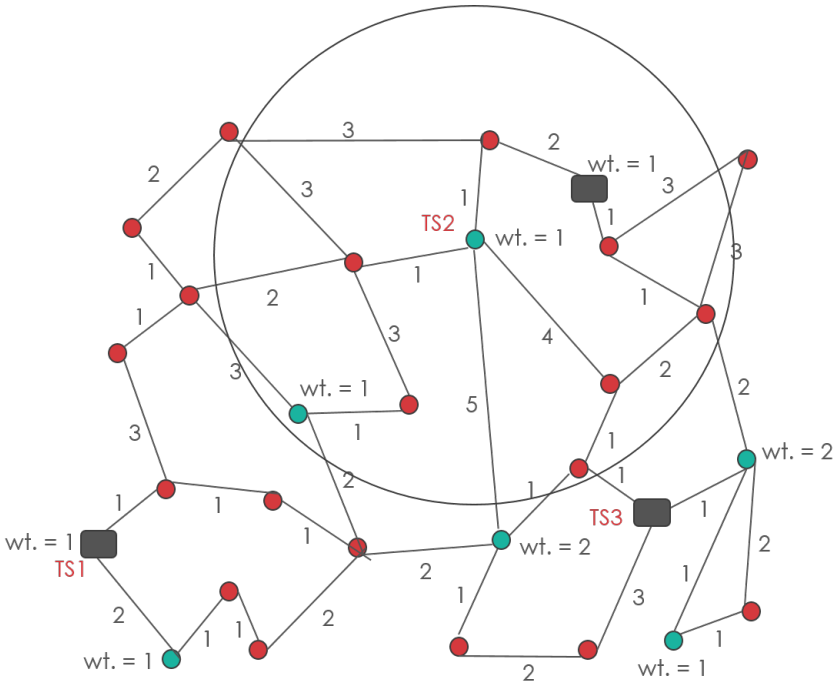

Figure 6.36: TS2 is moved to yellow vertex location of figure 6.35 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 19 units which is equal to cost θ. So, the swap is rejected, j is incremented by 1 (now j=2) and we move back to step 3 as shown in figure 6.37

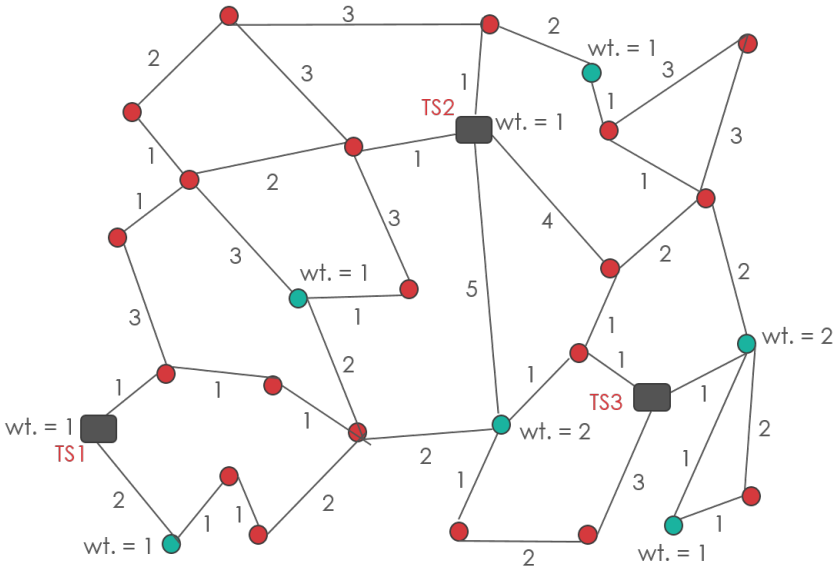

Figure 6.37: After removing the swap, we are back to our previous taxi locations

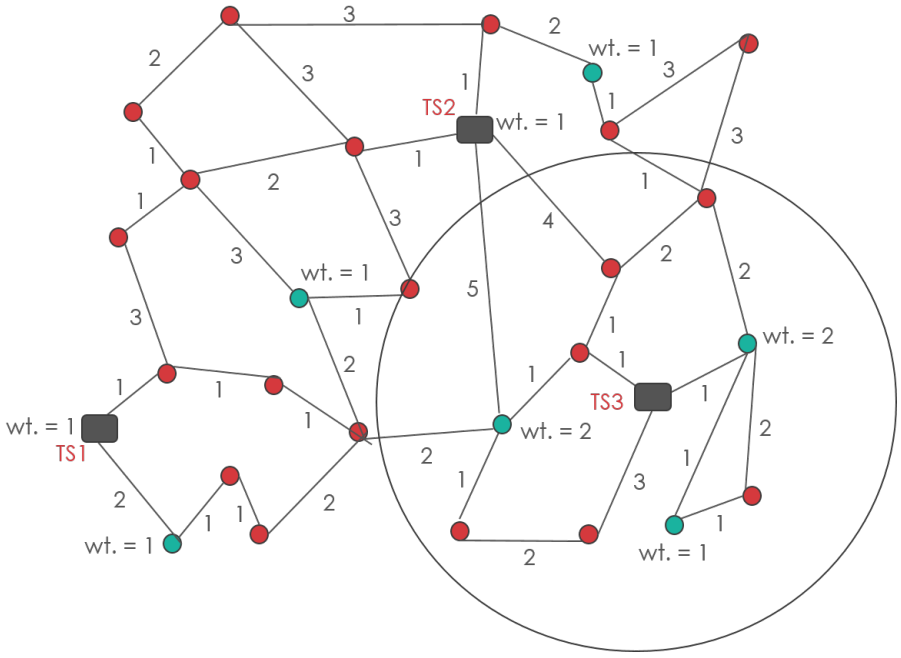

Figure 6.38: Moving to step 3, we again randomly choose one of the taxi stands. Let it be TS3 this time. Distance of all the pickup locations is calculated from the set of taxi locations (step 4). The cost θ comes out to be 19 units.

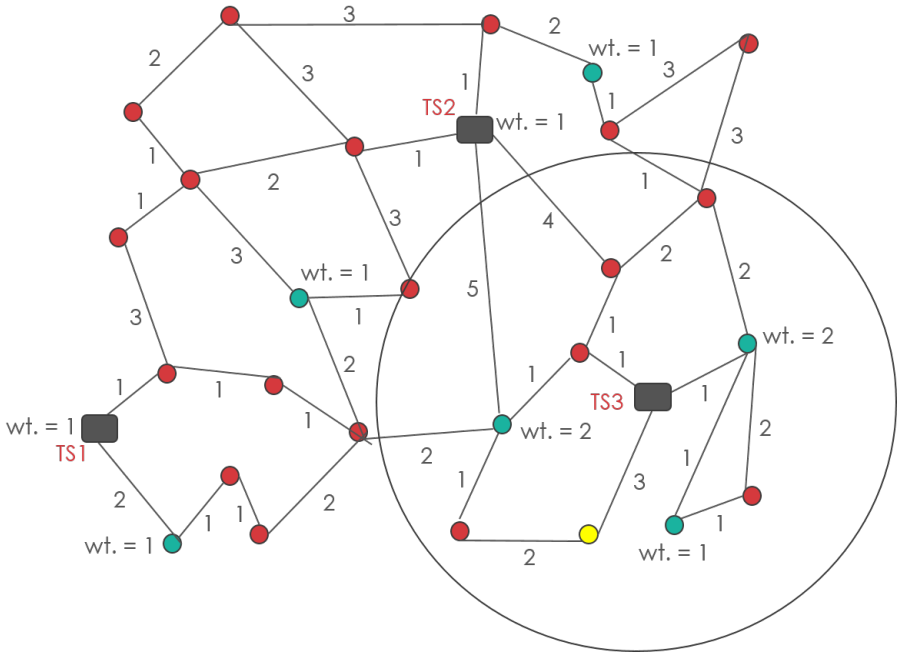

Figure 6.39: The new non-medoid randomly chosen is the yellow vertex vt. As it lies inside radius r, it is an acceptable vertex (step 6)

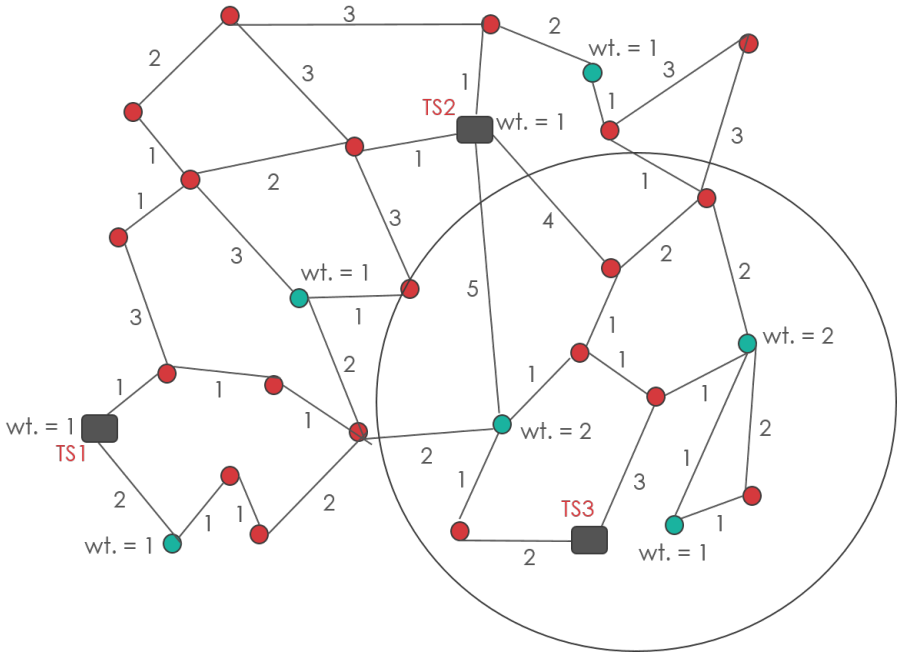

Figure 6.40: TS3 is moved to yellow vertex location of figure 6.39 and distance of all pickup locations is re-calculated from new set of taxi locations (step 7). The new cost γ comes out to be 24 units. So, the swap is rejected, j is incremented by 1 (now j=3). As now j=maxneighbor, increment i by 1 (step 10). Also, as now i=numlocal=1, the algorithm terminates.

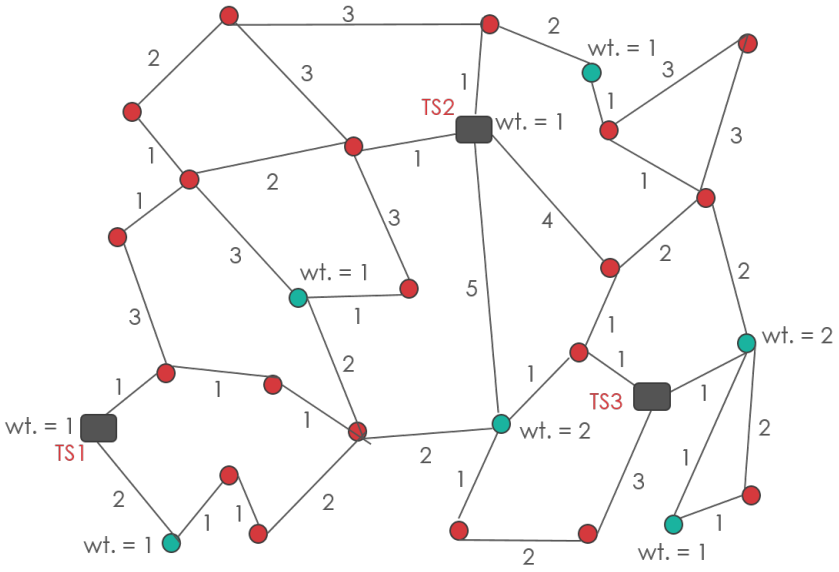

Figure 6.41: Final set of taxi locations TS1, TS2 and TS3 is returned

The pseudo-code of modified CLARANS is given in Algorithm 6.11

Algorithm 6.11 Modified CLARANS

Input:

- L: Initial set of taxi locations
- V: Vertex Set of graph G
- Integer *numlocal* ← #(local minima obtained)
- Integer *maxneighbor* ← #(maximum number of neighbors tested)

Output: A set of k initial locations

```
1: procedure MODIFIED CLARANS
2:     i=0, j=0 //Initialization
3:     while i < numlocal do
4:         while j < maxneighbor do
5:             Randomly select a seed m from L
6:             Calculate distance from each pickup vertex in V(wt > 0) to its nearest seed in L
   and multiply it with weight of pickup point. Add all such distances and let it be cost θ
7:             Select a vertex vt from V such that the vertex with higher wt. gets more priority
8:             if vt is 'nearby' to m then
9:                 Swap m with vt in L
10:                Calculate cost γ with new L
11:                if γ < θ then
12:                    Set j=0 and start from step 4 with new L
13:                else
14:                    Increment j by 1 and move to step 5
15:                end if
16:            else
17:                Move to step 7
18:            end if
19:        end while
20:        Increment i by 1
21:    end while
22: end procedure
```

6.2.4 Cost computation using Modified Dijkstra Algorithm

We saw in section 6.2.3 that we need to compute the distance of all the pickup locations from their nearest taxi stands to calculate the total cost θ. This means that we need to find the distance from a pickup location to every taxi stand and then add the minimum distance to θ. The pseudo-code for the same is given in Algorithm 6.12

The distance of each pickup point to a taxi stand is computed using Dijkstra's algorithm. So, we see that to calculate the minimum distance from a pickup location to a taxi stand, for a set of k taxi stands, k calls of distance function are required. So, we modified the Dijkstra algorithm in a way that instead of k times, we now only require one call of distance function.

The basics of traditional Dijkstra algorithm are given in section 4.3.2. Basically, it takes a node

Algorithm 6.12 Compute total costθ

Input:

- G: Weighted graph G
- L: Set of taxi locations

```
1: function computeCost(G)
2:     θ = 0
3:     for Each pickup location p in G do
4:         dist ← min distance(p,ts)  (for all taxi stands ts)
5:         θ+ = dist
                   m
6:     end for
7:     return θ
8: end function
```

as input in the graph as source node and returns the distance from it to all other nodes or if a destination node is also given, the algorithm stops as soon as the destination is reached. The complexity is $O(n^2)$ but can be minimized to $O(n * logn)$ when priority Queue is used to get the vertex at minimum distance.

For our case, we need to find the minimum distance from a pickup location (source node) to any medoid in list L. So, the modification that we have done is instead of having one destination, we use an array of destinations. Dijkstra algorithm uses this array and whenever any one destination of the array is found, the algorithm stops giving us the distance between source vt and the destination. The found destination will always be of the nearest vertex among all the medoids to vt. This is because, Dijkstra algorithm moves step by step by looking at each of its neighbors. The nearest of all the vertices (seeds) will be found first. Figure 6.42 depicts the scenario.

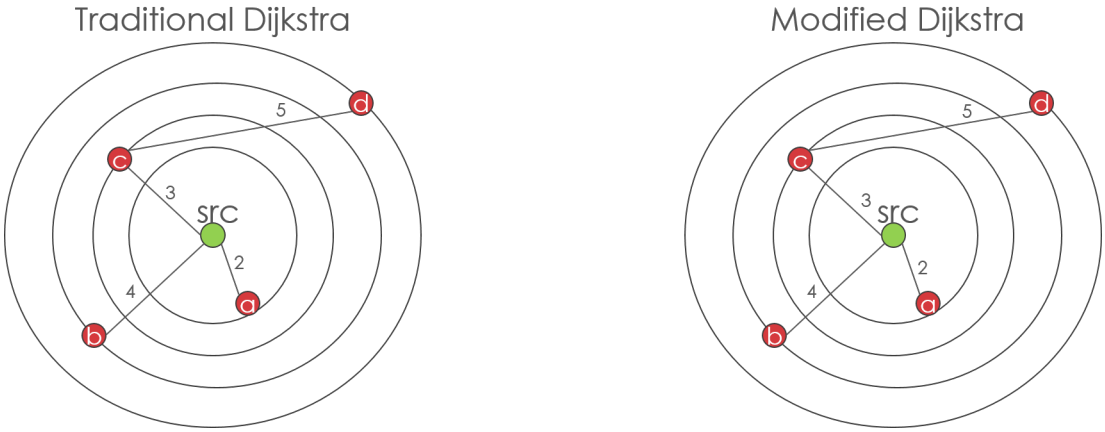

Figure 6.42: Let the red nodes: 1,2,3 and 4 be taxi stands and we want to find the minimum distance between any taxi stand and the source src. If we use traditional Dijkstra algorithm, we need 4 calls of distance function. Distance d = minimum{distance(src,a), distance(src,b), distance(src,c), distance(src,d}. So, d = minimum{2,4,3,8} = 2 units. Here distance(x,y) is the distance between x and y using Dijkstra algorithm.
In modified dijkstra algorithm, distance d = distance(src, [1,2,3,4]). It keeps looking for any of targets in the array and terminates as soon as any one of them is found which means as soon as 'a' is found the algorithm terminates. So, we need only one call of distance function.

The algorithm for modified Dijkstra is given in Algorithm 6.13

Algorithm 6.13 Modified Dijkstra Algorithm

Input:

- G: weighted graph
- vt: Source vertex
- dst[]: Array of destinations

Output: Value of distance between *vt* and nearest destination from dst[]

```
1:  procedure MODIFIED DIJKSTRA
2:      dist[vt] ← 0
3:      Create an empty priority Queue //Heap in our case
4:      Q.insert(vt, dist[v]) //with priority
5:      min = Q.extractMin()
6:      set.put(vt)
7:      while !(dst.contains(min)) do
8:          for each neighbor n of min do
9:              if !(set.contains(nbr)) then
10:                 θ = dist[min] + edgeLength[min,n]
11:                 if θ < dist[n] then
12:                     dist[n] ← θ
13:                     prev[n] ← min
14:                     Q.insert(n, dist[n])
15:                 end if
16:             end if
17:         end for
18:         min = Q.extractMin()
19:         set.put(min)
20:     end whilereturn dist[min]
21: end procedure
```

Chapter 7

Experimental Evaluation

In this chapter, we will first explain the pre-processing steps (section 7.1) required to obtain the output shown in figure 6.2. Then we will do the quality and run-time analysis of the clusters formed by running modified CLARANS algorithm over taxi data of Singapore in section 7.2. The quality analysis will be done using silhouette coefficient. As mentioned in section 4.3.3, it gives a measure about how similar an object is to its own cluster compared to other clusters. In section 7.3, we will do a comparative study between PAM and modified CLARANS to see trade-off between run-time and quality and will finally show the output of our algorithm in section 7.4.

7.1 Pre-processing

This section explains the pre-processing that needs to be done on the received inputs to apply clustering algorithms (section 6.2) onto them.

7.1.1 Pre-processing on trip file

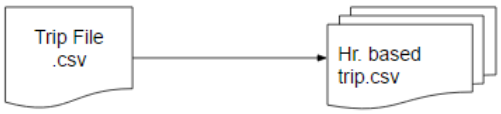

Figure 7.1: From every trip file, a set of files is generated which contain information about pickup locations for each hour in a city

A trip file contains entries of all the taxi trips of a day in a city. So, we take a trip file and extract pickup location (latitude, longitude) from it for each hour. This information gives us an insight on areas where people take more taxis and how the number of pickup points in these areas change over time. We store this information in the form *.csv* files for each hour starting from 8:00 AM in the morning to 9:00 PM in the night (13 such files). Each file contains about 17000-18000 pickup points. The format of all such files is shown in table 7.1 where *startlatitude* and *startlongitude* are the latitude and longitude of the location from where a taxi trip starts.

Start latitude	Start longitude

Table 7.1: Format of hr. based trip file

7.1.2 Pre-processing on OSM file

For our example stated in section 2.1, the nodes extracted from the OSM file might look something like as shown in figure 7.2 when plotted on the map. (The number of nodes are much more in reality in an XML file and will cover the entire road network. For simplicity we have taken only small number of them in the example).

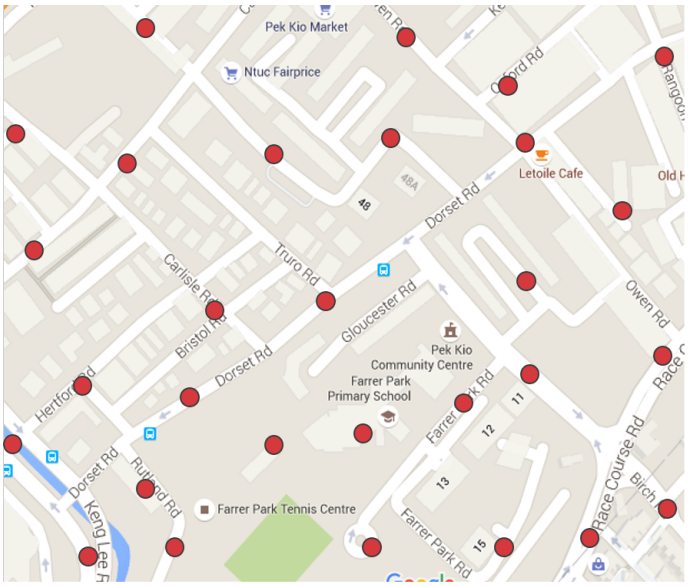

Figure 7.2: Nodes extracted from an OSM XML file when plotted on a map

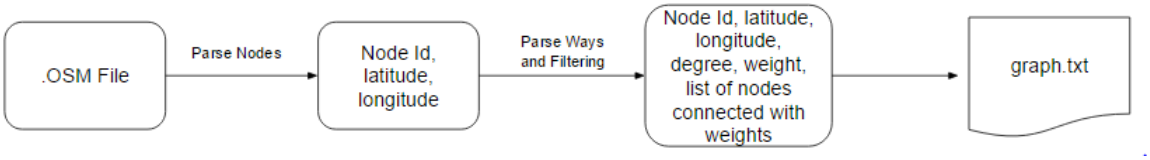

Figure 7.3: Flow of pre-processing steps of OSM file

1. **Parsing Nodes:** We linearly scan the OSM file and extract all nodes with their latitude and longitude information. This is usually a fast step and takes approximately 10 seconds per million nodes.

2. **Parsing Ways:** This is one of the crucial steps. For every way we parse, we directly store it as number of edges. As every way is a set of nodes, each of it (road) can be approximated as a set of small straight lines joined together(Figure 7.4). Thus, every two consecutive nodes on a way can be considered as an edge with distance between them as the Euclidean distance. This takes majority of the pre-processing time.

3. **Filtering Nodes:** The existence of zero-degree nodes is insignificant to the aim of computing distance between two nodes. So, it is better to filter them. Also, this step includes removal of various connected components. Since, we need to apply the Dijkstra's algorithm on the graph, so the graph must be connected. Hence we only keep the largest connected component and remove others. We store it as a text file (graph.txt).

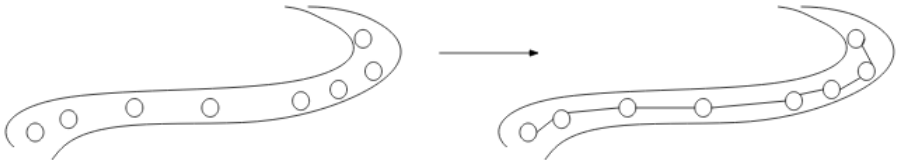

Figure 7.4: Each of the circles on the way are the set of nodes of which this way is made of. Each of these have a latitude and longitude and are extracted from the OSM XML file in step 1 of parsing. Thus, the length of the way can be approximated as the length of all the small straight lines as shown. Each edge between two consecutive nodes has a length equal to Euclidean distance between them.

So, at the end of processing OSM XML file, we have the road network of Singapore in the form of a graph. This is an edge-weighted graph where weight of each edge is the Euclidean distance between the two consecutive nodes. It contains IDs of all the nodes in XML file. Each node consists of its latitude, longitude, degree (number of nodes to which it is directly connected) weight initialized with zero (explained in section 7.1.3) and set of nodeIDs which are directly connected along with their corresponding distances.

Say for our example city, the graph formed is as shown in figure 7.5. For ease of understanding, the edge weights are taken as integer numbers.

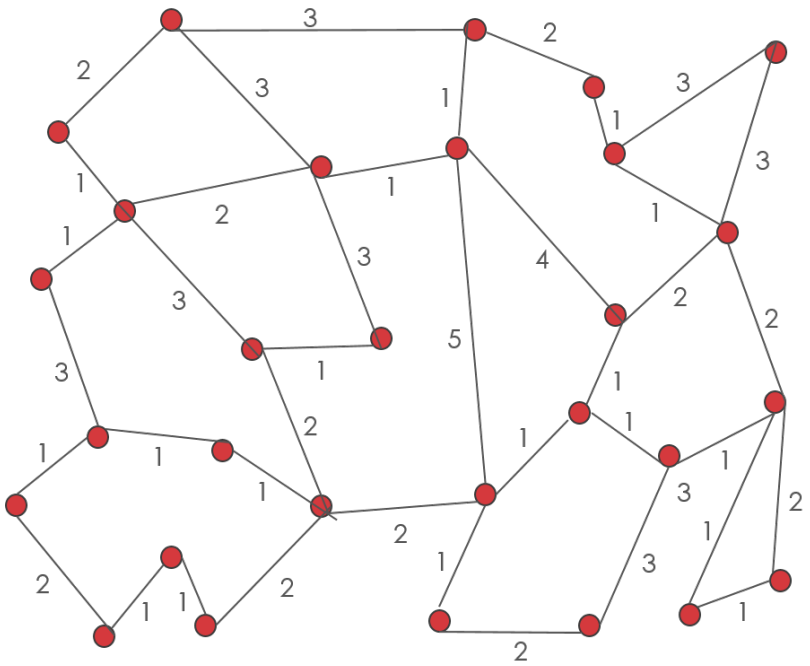

Figure 7.5: Graph formed by pre-processing of OSM XML file

7.1.3 Populating trip start locations on graph

The pre-processing of trip file and OSM XML file gave us two files, *hr.basedtrip.csv* and *graph.txt*, respectively. As mentioned in section 5.4, the locations in *hr.basedtrip.csv* file need to be mapped onto the graph. The whole procedure is listed as follows:

1. **Grid Structure:** We first make a grid structure to cover the whole land area of Singapore. So, the size if this structure depends on the minimum and maximum latitude and longitude of Singapore i.e. if the size of each block of grid is θ units, the number of rows in grid would be $(maximumlatitude - minimumlatitude)/\theta$ and the number of columns would be $(maximumlongitude - minimumlongitude)/\theta$.

2. **Insert nodes from graph:** Insert all the nodes obtained from OSM file (vertices of graph) into the grid based on their latitude and longitude.

3. **Get nearest nodeID** For a pickup location in the *hr.basedtrip.csv*, find the corresponding nearest node in the grid. This is done as follows:

   (a) Find the block in which pickup point p lies, based on its latitude and longitude.

   (b) Find the distance of point p, from all the nodes lying in its corresponding block as well as in eight blocks surrounding it (total 9 blocks are checked) and note the ID of the nearest node as well as its distance from p.

   (c) Say, the ID of found node is nn_1 and its distance from p is d_1. Distance between nn_1 and p is the Euclidean distance. If$(d_1 < \theta)$, then we have found the vertex in graph which is nearest to the pickup location p.

   (d) Otherwise, instead of looking at 9 blocks, compute the distance of p from all the nodes in 25 blocks surrounding it, including the one in which it lies. Again note the ID of nearest node and its distance from p.

   (e) Say, now the ID of found node is nn_2 and its distance from p is d_2. If$(d_2 < 2 * \theta)$, then we have found the node in graph which is nearest to the pickup location p.

   (f) The mth repetition of the procedure contains $(1+2^m)^2$ blocks to be checked for nodes with threshold for accepting the node ID nn_m as $distance(p, nn_m) < m * \theta$

   (g) The procedure is terminated after a certain number of blocks have been checked (49 in our case), considering p, as an outlier.

4. For the nearest node Id found, increase its *weight* by one. Recall that every node ID/vertex in the graph formed in section 7.1.1 had a weight attribute with it initialized with zero.

5. Repeat step 3 for every point in *hr.basedtrip.csv* and save the new edge and vertex weighted graph as *weightedgraph.txt*

So, at the end of step 5, we have an edge and vertex weighted graph. Note that each vertex in this graph has a weight equal to number of pickup points for which it was nearest. Thus, some vertices in this graph might have a very high weight (for a node in an area with high density of pickup points) and others can have a weight as low as 0. An instance of such a graph for our example city is shown in figure 7.6

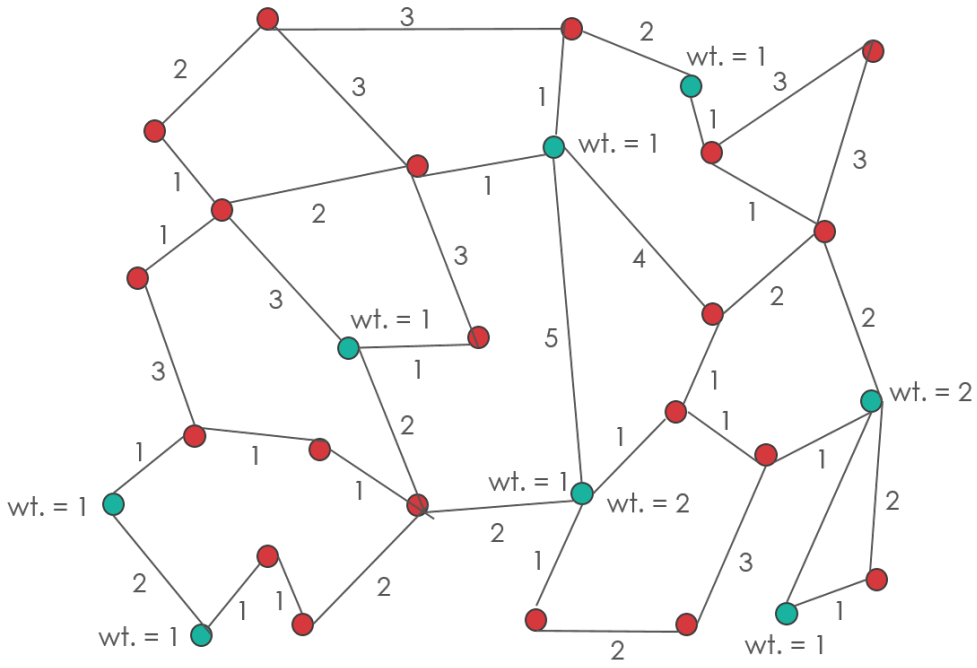

Figure 7.6: This graph is the input to our algorithm in section 6.2 as shown in figure 6.2

7.2 Experiments

For the purpose of our experiments, we varied 4 parameters, namely *numlocal*, *maxneighbor*, k and *radius* and checked how these affected the quality of clusters and run-time of algorithm. So, from our total dataset of 630 .csv files (Total data is of 3 months i.e. 90 files and for each day, as the trip entries in each file are made on the basis of time, we have made 7 files for each day starting from 8:00 AM in the morning to 9:00 PM in the night for alternate hours. Thus, 630 = 90*7 files are there), we chose 5 random files which are mentioned in table 7.2. For each of these files we varied the above mentioned parameters.

Dataset	No. of pickup locations
1	1385
2	1420
3	1368
4	1389
5	1378

Table 7.2: Datasets represent the weighted graphs stored as text files. We chose 5 random such graphs for our experiments

Experiment 1: Vary numlocal

The different values chosen for varying *numlocal* were [1,2,4]. While varying *numlocal*, all other values were kept constant with *maxneighbor* = 20, $k = 5$ and *radius* = 17 km. Modified CLARANS was executed on all the 5 chosen files for all the values of *numlocal*. Thus, 5*3 executions were there each giving a set of clusters. Average total time and average silhouette

coefficient were calculated for files having same value of *numlocal*. For example, for *numlocal* = 1, if run-time of files 1 to 5 are r1 to r5 respectively, and if silhouette coefficient value for the clusters made are s1 to s5 respectively, average runtime = $(\sum_{i=1}^{5} r_i)/5$ and average silhouette coefficient = $(\sum_{i=1}^{5} s_i)/5$. Three such average run-times and silhouette coefficient values are obtained for three different values of *numlocal*.
The values of silhouette coefficient ($s(i)$) and runtime are given in tables 7.1 and 7.2 for all the five input files and their variation with *numlocal* is shown in figure 7.1. It is clear from figure 7.1 that both run-time and quality of clustering (average $s(i)$) increase as *numlocal* increases. Also, as $s(i)$ value varies from -1 to 1 with value zero being an average quality clustering, our values of $s(i)$, tend to reach 0.5, which can be considered a good clustering (taking into fact that number of clusters to be formed (k) is an input which is constant).

Run-time (in min)						
numlocal	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average run-time
1	26.26	53.4	31	63	89	52.5
2	88	113	86.36	91.24	71.6	90.02
4	229.3	159.17	208.21	185.45	198.2	217.09

Table 7.3: Values of runtime for all the 5 input files with different values of numlocal

silhouette Coefficient						
numlocal	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average silhouette
1	0.370745572	0.392551248	0.373171611	0.379098699	0.449267644	0.392966955
2	0.352099626	0.416080488	0.406294808	0.461146623	0.417903263	0.410704962
4	0.498336519	0.492949187	0.442737506	0.39549697	0.513042474	0.468512531

Table 7.4: Values of silhouette coefficient for all the 5 input files with different values of numlocal

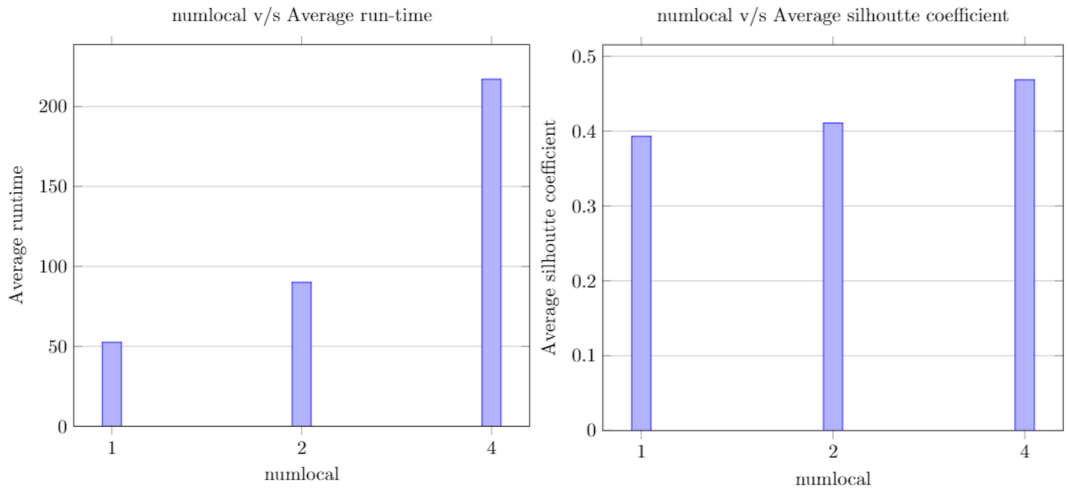

Figure 7.7: Graph showing variation of average run-time and average silhouette coefficient with numlocal. Other parameters were kept constant at values maxneighbor = 20, k=5, radius = 17 km.

Experiment 2: Vary maxneighbor

The values chosen for *maxneighbor* were [5,10,20,25,50] with other values being constant as $numlocal = 1$, $k = 5$. As 5 different values of *maxneighbor* are used, 25 executions of modified CLARANS were there, each giving a set of clusters. The procedure of calculating average run-time and average silhouette coefficient is the same as mentioned in experiment 1, giving 5 different values for both of them.
The values of average silhouette coefficient and average runtime are given in tables 7.3 and 7.4 for all the five input files and their variation with *maxneighbor* is shown in figure 7.2. The trends obtained for both run-time and silhouette coefficient are similar to that obtained in experiment and suggest that both of them increase with increasing value of *maxneighbor*.

Run-time (in min)						
maxneighbor	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average run-time
5	6.56	11.22	12.46	10.19	6.9	9.52
10	8.20	13.14	14.28	12.9	23.19	14.3
20	32.2	50.18	34.49	56.13	26.30	39.97
25	28.42	71	25.3	56.57	66.16	67.59
50	141.24	79.59	104.25	67.2	56.11	89.88

Table 7.5: Values of runtime for all the 5 input files with different values of maxneighbor

silhouette Coefficient						
maxneighbor	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average silhouette
5	0.442532739	0.302967431	0.329585757	0.454942978	0.365100384	0.379025858
10	0.410455909	0.373873306	0.347698799	0.382482575	0.485688576	0.400039833
20	0.496328858	0.385302447	0.524443191	0.446724934	0.357798259	0.442119538
25	0.420555457	0.415559924	0.417756858	0.301224943	0.453425727	0.401704582
50	0.498356303	0.399360663	0.513097795	0.46101122	0.463977398	0.467160676

Table 7.6: Values of Silhouette coefficient for all the 5 input files with different values of maxneighbor

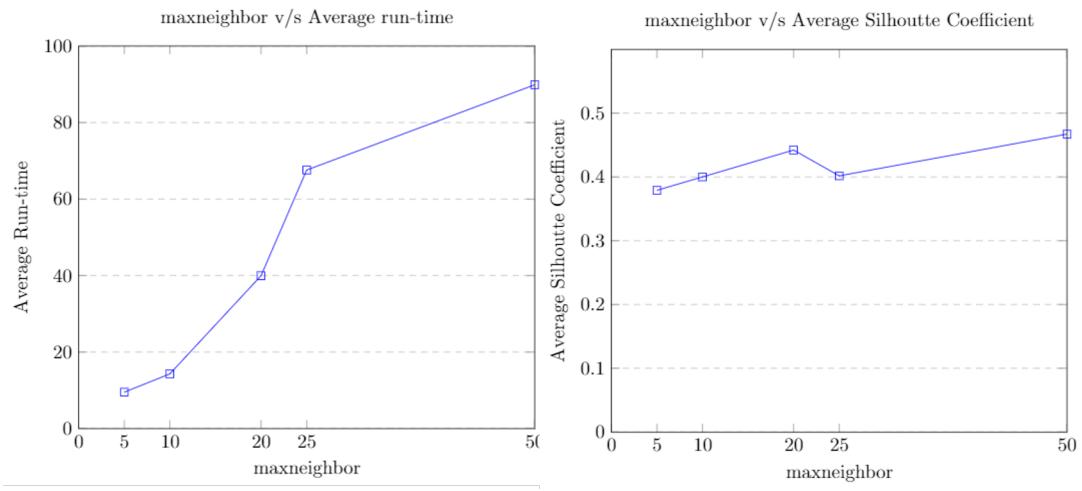

Figure 7.8: Graph showing variation of average run-time and average Silhouette coefficient with maxneighbor. Other parameters were kept constant at values numlocal = 1, k=5, radius = 17 km.

Experiment 3: Vary number of taxi stands: k

The values of k chosen were [3,5,7,10] with other values being constant as $numlocal = 1$, $maxneighbor = 20$ and $radius = 17km$. . For 4 different values of k and 5 input datasets, 20 outputs were generated. The values of runtime and silhouette coefficient are as shown in tales 7.7 and 7.8 respectively. The graph showing variation of average run-time and Silhouette coefficient with k are shown in figure 7.9. It is clear from the graph that runtime decreases as number of taxi stands (k) increase and silhouette coefficient first increases and then decreases.

Run-time (in min)						
k	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average run-time
3	31.33	43.16	24.29	99.28	70.3	53.672
5	32.2	50.18	34.49	56.13	26.30	39.97
7	23.2	14.2	14.41	15.28	17.57	16.932
10	12.29	14.2	9.4	9.46	11.58	11.386

Table 7.7: Values of runtime for all the 5 input files with different values of k

Silhouette Coefficient						
k	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average Silhouette
3	0.431241644	0.464150928	0.422323851	0.462841773	0.425266345	0.441164908
5	0.496328858	0.385302447	0.524443191	0.446724934	0.357798259	0.4421195378
7	0.322503018	0.420066719	0.408855406	0.777421806	0.387550084	0.463279407
10	0.359032185	0.314989496	0.346681189	0.361564576	0.348785393	0.346210568

Table 7.8: Values of Silhouette coefficient for all the 5 input files with different values of k

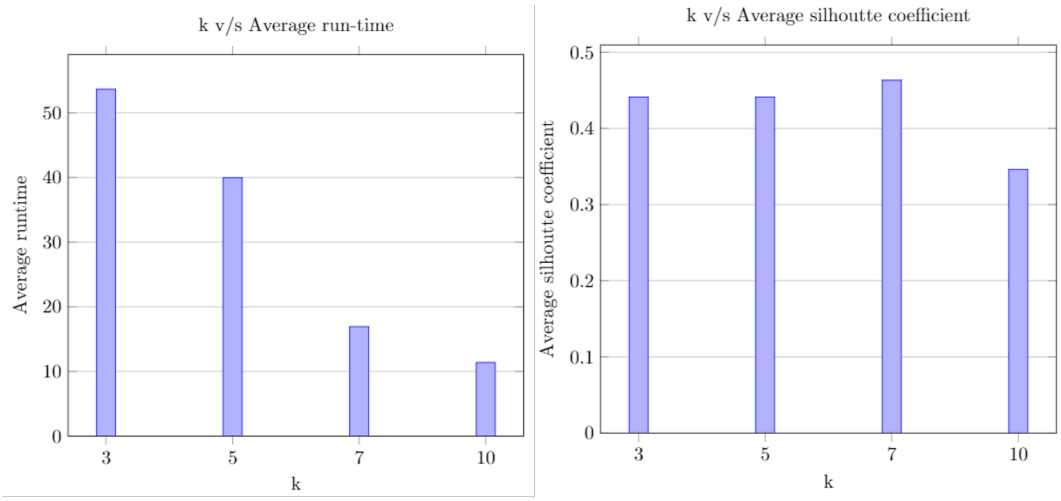

Figure 7.9: Graph showing variation of average run-time and average Silhouette coefficient with k. Other parameters were kept constant at values numlocal = 1, maxneighbor = 20 and radius = 17 km.

Experiment 4: Vary radius r

The values of *radius* chosen were [5,10,17,25] with other values being constant as $numlocal = 1$, $maxneighbor = 20$ and $k = 5$. For 4 different values of *radius* and 5 input datasets, 20 outputs were generated. The values of runtime and silhouette coefficient are as shown in tales 7.9 and 7.10 respectively. The graphs showing variation of average run-time and average silhouette coefficient with k are shown in figure 7.9. It is clear from the graph that even though runtime increases with increasin radius, not much of a variation is seen in silhouette coefficient denoting the quality of clusters formed. In one case like for radius=10 km, quality even decreased with increase in radius from 5km to 10 km.

Run-time (in min)						
radius	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average run-time
5	18.3	23.4	23.34	22.4	22.26	21.94
10	18	46.9	35.1	65.23	25.1	38.066
17	32.2	50.18	34.49	56.13	26.30	39.97
25	47.1	49.2	59.5	47.24	26.8	45.968

Table 7.9: Values of runtime for all the 5 input files with different values of radius

silhouette Coefficient						
radius	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Average silhouette
5	0.393336689	0.375622767	0.444860714	0.499635653	0.396592959	0.422009756
10	0.321835154	0.514709007	0.310978056	0.4435837	0.397482613	0.397717706
17	0.496328858	0.385302447	0.524443191	0.446724934	0.357798259	0.442119538
25	0.476549564	0.477516266	0.510429865	0.414955213	0.334148582	0.442719898

Table 7.10: Values of silhouette coefficient for all the 5 input files with different values of radius

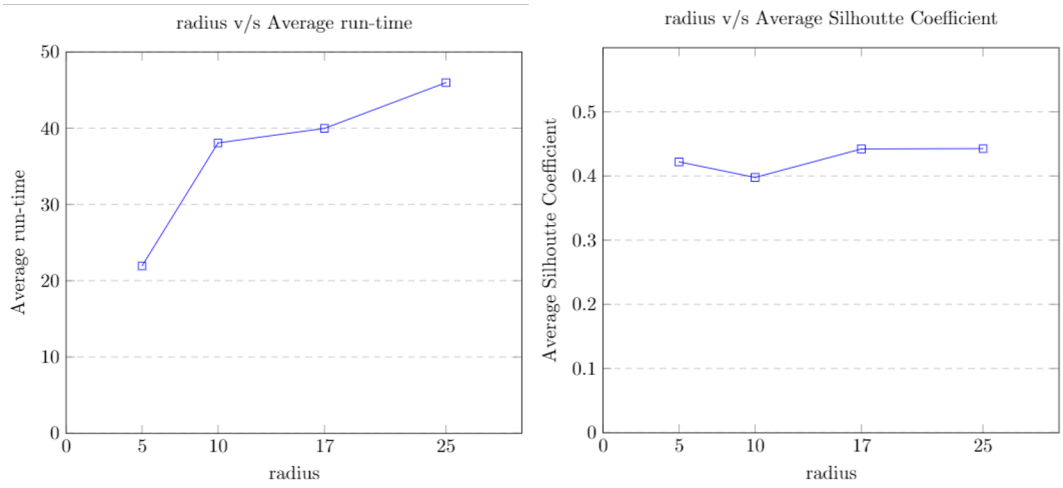

Figure 7.10: Graph showing variation of average run-time and average silhouette coefficient with radius. Other parameters were kept constant at values numlocal = 1, maxneighbor = 20 and k = 5

7.3 Comparison with PAM

We already know by now that PAM can give us a local minima (in terms of cost) for a given configuration but due to its high computation cost, it is not considered suitable for big datasets. So, we tried comparing our solution with the one obtained using PAM. After 10 days (still running) it reached to a cost value of 576.4561347692884. On the other hand CLARANS with $numlocal = 4$, $radius = 17$km and $maxneighbor = 20$, gave a cost value of 626.6786462261939 in 3hrs and 18 minutes. So, we see that with a little trade-off in quality, we are saving a lot in running time.

7.4 Output

Here we show the final results obtained by running modified CLARANS algorithm on Singapore. One of the snapshots is shown in figure 7.11. Here, we have made five clusters which are represented by five different colors. Each of the green markers represent the recommended location of taxi stand. As the points are weighted, it is not necessary that the taxi stand (green marker) will lie somewhere in the center of the cluster. Instead, it lies on the side of the cluster where weight of points is more. The scenario is clearly depicted in the cluster marked yellow. The area marked yellow is around the Changi airport and the taxi stand lies on the junction of the road going to the airport.

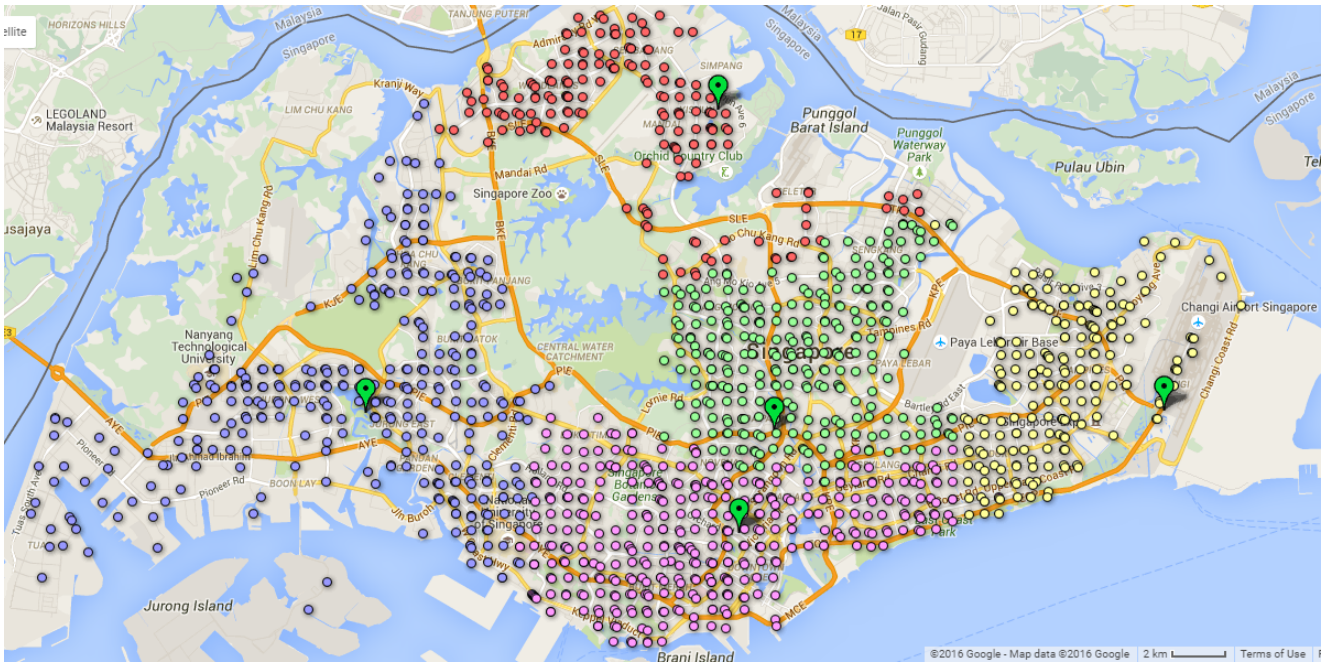

Figure 7.11: Snapshot showing clusters obtained with recommended taxi stands marked as green markers. Each point shown is weighted with weight of the point equal to number of pickup locations associated with it. Different points thus have different weights

Chapter 8

Conclusion and Future Work

We suggested an algorithm to find locations for taxi stands on a city map. To do this task, we modified CLARANS algorithm and the two major suggested by us were: 1) Process of selecting initial seeds 2) Swapping medoids with non-medoids in 'nearby' areas. We show a trade-off between quality and runtime between solution obtained from our algorithm and the one obtained using PAM. We also show that not much variation occurs in clusters' quality if we increase the size of 'nearby' areas but running time increases to almost double the value.

For future work, we aim to include all the 'end points' of the taxi trips and not just pick-up points. Also, we intend to do a case study to find different mobility and traffic patterns. The differences can exist at different hours of the day as some areas are more crowded than others. We aim to find such areas. Also, we will look at dominant areas on weekends and week-days and will try to analyze the reasons for the above mentioned trends.

Bibliography

[1] Vladimir Estivill-Castrol and Alan T. Murray. "Discovering associations in spatial data — An efficient medoid based approach". In: *Second Pacific-Asia Conference, PAKDD-98 Melbourne, Australia, April 15–17, 1998 Proceedings* (1998), pp. 110–121.

[2] Yuan Gao et al. "Visualization of Taxi Drivers' Income and Mobility Intelligence". In: *8th International Symposium, ISVC 2012, Rethymnon, Crete, Greece* 7 (2012), pp. 275–284. DOI: 10.1007/978-3-642-33191-6_27.

[3] Yong Ge et al. "An Energy-Efficient Mobile Recommender System". In: *KDD '10 Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining* (2010), pp. 899–908. DOI: 10.1145/1835804.1835918.

[4] Google. *Google fusion Table*. 1999. URL: https://support.google.com/fusiontables/answer/184641?hl=en (visited on 06/20/2016).

[5] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2011. ISBN: 9780123814807.

[6] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. "How Easy Is Local Search?" In: *Journal of Computer and System Sciences* (1988), pp. 79–100. DOI: 10.1016/0022-0000(88)90046-3.

[7] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[8] Bin Li et al. "Hunting or waiting? Discovering passenger-finding strategies from a large-scale real-world taxi dataset". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on* (2011), pp. 63–68. DOI: 10.1109/PERCOMW.2011.5766967.

[9] Siyuan Liu et al. "Detecting Crowdedness Spot in City Transportation". In: *IEEE Transactions on Vehicular Technology* 62.4 (2012), pp. 1527–1539. DOI: 10.1109/TVT.2012.2231973.

[10] Siyuan Liu et al. "Modeling Social Information Learning among Taxi Drivers". In: *The 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2013)* 7819 (2013), pp. 73–84.

[11] J. MACQUEEN. "Some Methods for classification and Analysis of Multivariate Observations". In: *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press* (1967), pp. 281–297.

[12] R.T. Ng and J. Han. "CLARANS: a method for clustering objects for spaial data mining". In: *IEEE Transactions on Knowledge and Data Engineering* 14.5 (2002), pp. 1003–1016. DOI: 10.1109/TKDE.2002.1033770.

[13] Hae-Sang Park and Chi-Hyuck Jun. "A simple and fast algorithm for K-medoids clustering". In: *Expert Systems with Applications: An International Journal* 36.2 (2009), pp. 3336–3341. DOI: 0.1016/j.eswa.2008.01.039.

[14] Santi Phithakkitnukoon et al. "Taxi-Aware Map: Identifying and Predicting Vacant Taxis in the City". In: *First International Joint Conference, AmI 2010, Malaga, Spain, November 10-12, 2010. Proceedings* (2010), pp. 86–95. DOI: 10.1007/978-3-642-16917-5_9.

[15] Bart Selman and Carla P Gomes. *Hill-Climbing Search*. URL: http://www.cs.cornell.edu/gomes/selman-gomes-encycl-hillclimbing.pdf.

[16] Kiam Tian Seow, Nam Hai Dang, and Der-Horng Lee. "A Collaborative Multiagent Taxi-Dispatch System". In: *IEEE Transactions on Automation Science and Engineering* 7 (2010), pp. 607–616. DOI: 10.1109/TASE.2009.2028577.

[17] Jing Yuan et al. "Where to find my next passenger?" In: *UbiComp '11 Proceedings of the 13th international conference on Ubiquitous computing* (2011), pp. 109–118. DOI: 110.1145/2030112.2030128.

[18] Qiaoping Zhang and Isabelle Couloigner. "A New and Efficient K-Medoid Algorithm for Spatial Clustering". In: *International Conference, Singapore, May 9-12, 2005, Proceedings, Part III* (2005), pp. 181–189. DOI: 10.1007/11424857_20.

[19] Xudon Zheng, Xiao Liang, and Ke xu. "Where to wait for a taxi?" In: *UrbComp '12 Proceedings of the ACM SIGKDD International Workshop on Urban Computing* (2012), pp. 149–156. DOI: 10.1145/2346496.2346520.