CS5234 Mini Project

Most frequent trips and hubs in sublinear time

- Naheed Anjum Arafat
- Divya Sivasankaran

Introduction

In this project we try to perform different kinds of analytics on the New York City Taxi Trip data provided by Taxi and Limousine Commission (TLC) in Smaller Memory Space (Sublinear) and/or in Limited time (Sublinear Time). The TLC trip dataset is very extensive in the sense it includes fields capturing **pick-up and drop-off dates/times**, **pick-up and drop-off locations**, **trip distances**, **itemized fares**, **rate types**, **payment types**, and **driver-reported passenger counts** for each trip made in yellow and green taxicab.

One reason for choosing this dataset for analysis is it contains Trip records from January, 2009 to June, 2016, totalling to over 1.1 billion taxi trips occupying over 267 GB on disk. Thus it undoubtedly fits the criterion of the mini-project since the resulting graph on this dataset will be impossible to fit in memory and performing faster analytics would be challenging as well.

The problems we want to focus on are as follows:-

1. Finding Most frequent trips (i.e, most popular destination pairs) over a time horizon over a stream

2. Finding Most connected hubs/networks (i.e highly connected components) over a time window and preferably in a sliding horizon Sketch.

To fulfill those tasks in sublinear time, we focus on sampling algorithms, that guarantee success with some error bounds. In the next section we describe the dataset and our approach.

Background & Approach

Most Frequent Trips

The NYC dataset consists of all the taxi trips made within New York from January 2009. Analyzing the most frequent/top k destination pairs could give us useful insights about the city's transport connectivity. One could easily conjecture that the busiest end points in the number of taxis taken between them could be a result of the lack of easier public transportation/ newly rising bottleneck in the transportation network arising from any number of reasons like increasing crowd, lack of road infrastructure, inaccessible public transportation in one or both ends, etc. Addressing these could vastly improve the city's infrastructure by addressing the weakest links. We approach this as a counting problem in a graph stream.

We use a standard Count-Min-Sketch algorithm presented in the next section. Assuming, each trip made is updated via a stream, we can construct a sketch, that estimates the frequency of the edge at any point without actually storing the entire stream/graph. The construction & algorithm are presented in the sections below.

Highly connected networks/hubs

For reasons, very similar to the ones presented for the previous problem, analysing networks/communities within the taxi trips over different time periods could give better insight to plan public transportation, infrastructure planning. Understanding

communities/clustering patterns could also provide taxi companies rearrange the taxi/fleet allocations in dispatch centers around the city in order to maximize their businesses.

We approach this problem using a sliding window model (graph) to estimate the clusters based on a sub-graph that is formed by sampling edges from this window. In [2], the authors present a sliding window based sublinear time & space algorithm for clustering. Their idea is based on the assumption that finding connected components in a sampled subgraph is a good approximation to the clusters in the original graph (Figure [1]). However, they do not give the specifics for their proof of correctness or approximation guarantees. In the next sections, we present our naive sampling algorithm based on this intuition from [2] and also provide the guarantees for the sampling parameter chosen by it.

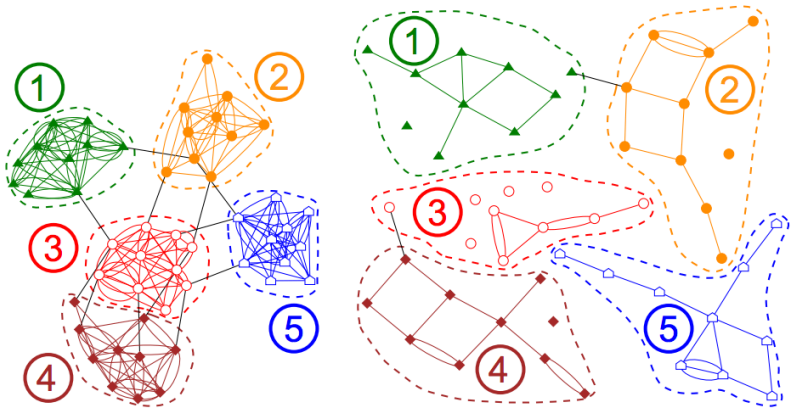

Figure 1: (a) shows original graph G, (b) shows connected components in the sampled subgraph T

We also present an improvement to this algorithm by proposing a sampling algorithm for the minimum-k-cut problem [5]. The minimum-k-cut problem is a graph problem where the goal is to identify a set of edges with minimum weight,

whose removal results in k connected components in the graph. This version proves to be better than the first algorithm for the NYC dataset. This is because the NYC dataset is very dense and nearly fully-connected graph. We will discuss this further in the next sections.

Graph Construction:

In this section we present the construction of our graph from the NYC dataset
We construct the graph as follows:

Location grids:

The pick-up and drop off locations are stored as lat-long coordinate pairs. We use a basic grid-technique to discretize the continuous locations in the map. We create a unique location ID based on a rectangular grid superimposed on the lat-long extremities of New York based on the dataset. The grid size is a tunable parameter and we currently use a grid of size of 10000, which results in 10^8 unique possible locations/nodes.

Building the graph:

Nodes - each location grid indicated by a unique LocationID.

Edges - each edge indicates a trip made between the two corresponding location grids/nodes. The edge weight increases by 1 for every trip made between the two points in the grids.
In the following sections we present Count-Min sketch first. Then we present the algorithms along with the theoretical bounds on sampling so that the algorithm succeeds with high probability.

Algorithms & Theory

Count-Min Sketch

Count-Min Sketch is a probabilistic Data Structure to store a function computed on the subset of the items (for instance, L_p-norm of the data items) in the stream and later query on the function value stored in it. See [4] for detailed description.
We have A hash functions and each of the hash functions map {1,2,...,n} to {1,2,...,B}. Thus there are B counters associated with each of the A hash functions. In our application, we are counting L_0 norm of the frequency of the edges/trips.
Analysis:
Suppose, x is an edge in the stream with true frequency n(x) and query(x) returns an estimate of n(x) computed by taking min of the estimate of n(x) returned by A hash functions which hashed x into B locations each.
Claim:
If $B = 2/\delta$ and $A = \log(1/\varepsilon)$, then $|\text{query}(x)-n(x) \leq \delta N$ with probability at least $(1 - \varepsilon)$.
Following this claim,
Space-complexity of Count-Min sketch is: $O(AB(\log m) + B(\log N))$
where AB = total number of counters and m = maximum trip frequency and N = total number of edges.
Time-complexity of query(x) is: $O(B)$

Sampling algorithm for Graph Clustering

Below we show the algorithm we have designed based on the paper by Ahmed et al. [2] for finding the most popular network/clusters in the graph. One significant difference between their work and ours is the theoretical bound for sample size that we provide here is lacking in their work. We use a sliding window model, and

we sample the edges from the window to estimate the number of connected components in the original graph.

Algorithm for Graph Clustering
  Input: Graph G
  Output: Set of Clusters C
  Sampled edge set, S = N_p edges sampled from G
  C = ϕ
  for each edge $e = (v_i, v_j) \in S$ **do**
    $C_i = FindSet(v_i)$
    $C_j = FindSet(v_j)$
    if C_i is empty **then**
      $C_i = MakeSet(v_i)$
    end if
    if C_j is empty **then**
      $C_j = MakeSet(v_j)$
    end if
    $C_i = Union(C_i, C_j)$
    Update C_i in C
  end for
  Return C

Algorithm 1

We provide the following two lemma for optimal value of N_p:-

Lemma 1. *Given there are t clusters, each pair of them separated by k cutedges, the algorithm succeeds (gives t clusters/t components) with probability at least δ if*

$$N_p \leq \frac{m \ln(\frac{1}{\delta})}{k\binom{t}{2}} \tag{1}$$

Proof. Consider a Graph $G = (V, E)$ with n vertices and m edges. Since each of the $\binom{t}{2}$ clusters are separated by k inter-cluster cut edges, there are in total $k\binom{t}{2}$ inter-cluster cut edges. Lets denote this set as Ce
The probability that a randomly picked edge X is not from Ce, $Pr(X \notin Ce) = (1 - \frac{k\binom{t}{2}}{m})$
The probability that none of the N_p randomly picked edges by the algorithm is not from Ce = Pr(success) = $(1 - \frac{k\binom{t}{2}}{m})^{N_p}$
In order to have Pr(success) $\geq \delta$ we should have $(1 - \frac{k\binom{t}{2}}{m})^{N_p} \geq \delta$. However, $(1 - k\binom{t}{2})^{N_p} \leq e^{\frac{-kt\binom{t}{2}N_p}{m}}$. Thus we have $N_p \leq \frac{m \ln(\frac{1}{\delta})}{k\binom{t}{2}}$ □

Lemma 2. *Given a graph with m edges having t connected components with min-cut size k', the algorithm succeeds with probability at least ϵ if*

$$N_p \geq \frac{m \ln(\frac{t}{1-\epsilon})}{k'}) \tag{2}$$

Proof. Suppose, $\{S_1, S_2, \ldots, S_t\}$ are the set of edges of the t components with $\{m_1, m_2, \ldots, m_t\}$ edges in graph G. Each of the $S_i's$ have a cutsets $\{C_1, C_2, \ldots, C_t\}$ of cardianlity $k_1, k_2, \ldots, k_t$.
Pr(Success) = Pr(Success at Every Cluster) [Here Success at every cluster means at least one cut-edge from every cluster] = 1 - Pr(Failure at at least 1 cluster) $\geq$ 1 - $\sum_{i=1}^{t}$ Pr(Failure at Cluster i) [By Union Bound] = 1 - $\sum_{i=1}^{t}$ Pr(None of the $e \in C_i$ is picked over N_p trial) = 1 - $\sum_{i=1}^{t}(\frac{m-k_i}{m})^{N_p} \geq 1-(\sum_{i=1}^{t} e^{-\frac{k_i N_p}{m}})$.
Assuming $k_1 = k_2 = k_3 = \cdots = k_t = k'$, Pr(Success) $\geq (1 - te^{-\frac{k' N_p}{m}})$.
We want Pr(Success) $\geq \epsilon$, Thus, $N_p \geq \frac{m \ln(\frac{t}{1-\epsilon})}{k'})$

□

The main idea behind the two lemma is we want the sample size to be small enough to not include inter-cluster cut-edges but large enough to include the min-cut edges in each cluster.

Space Complexity: $\mathcal{O}(N_p)$

Time Complexity: MakeSet: $\mathcal{O}(1)$. FindSet: $\mathcal{O}(1)$ Union: $\mathcal{O}(2N_p)$
Total: $\mathcal{O}(N_p^2)$

Improved Graph Clustering algorithm based on k-min cut

The above mentioned algorithm only works under the assumption that after sampling, the sub-graph will contain two or more connected components. But, for highly connected dense graphs, this assumption doesn't always hold. Finding a sample size that works with such graphs becomes a lot harder. So, we suggest approaching the clustering problem as a k-minimum cut problem.

The intuition for this change is as follows - we sort the edges in descending order of their edge weights. We then iterate the edges in this order, and only merge the two originating clusters if the new cluster's connectivity is higher than their previous connectivities. But, this raises the question of estimating the connectivity of a clustering (which in and of itself is often a hard problem). But, for our purposes, we can simply use the lowest degree vertex of the cluster as a representation of its connectivity. This is because, the connectivity of the cluster cannot be more than that of the lowest degree vertex. In our implementation of the Disjoint Set, we merge two sets using the degree of a vertex, by pushing the lowest degree node to the root during the union. We then use the weight of the root, to evaluate whether two clusters can be merged.

Run Time: $O(N_p^2 + N_p log\ N_p)$

Space Complexity: $O(N_p)$

In the next section we describe the main implementation decisions we made, along with the experimental results.

Improved Graph Clustering based on k-min cut

Input: Graph G
Output: Set of Clusters C
Sampled edge set, S = N_p edges sampled from G
Sort S in decreasing order of weight
C = ϕ
for each edge $e = (v_i, v_j) \in S$ **do**
  $C_i = FindSet(v_i)$
  $C_j = FindSet(v_j)$
  if C_i is empty **then**
    $C_i = MakeSet(v_i)$
  end if
  if C_j is empty **then**
    $C_j = MakeSet(v_j)$
  end if
  $\hat{C} = Union(C_i, C_j)$
  if $\hat{C}.connectivity > C_i.connectivity$ OR $\hat{C}.connectivity > C_j.connectivity$ **then**
    $C_i = \hat{C}$
    Update C_i in C
    Delete C_j from C
  end if
end for
Return C

Algorithm 2

Implementation

We implemented all the algorithms in python. We used the python package **Gephi** for all our visualizations. Below are the basic implementation details along with some key-decisions made during the implementation that affects our results. We use the built-in hash functions from python for the counters.

Graph

In the streaming version, we read each of the incoming trips/(from a window), and add edges on the stream for every trip encountered in the data by sending (EdgeID, 1) to the stream. There is no edge delete operation. We do not store a graph.

In the sliding window version, we simply store a graph with the nodes, edges as adjacency lists. The weight of the edge will represent the number of trips taken between the two nodes within the window.

Disjoint Set

For both the clustering algorithms, we use disjoint sets to estimate the number of connected components in the graph. We choose to hold a hash map for every node, to hold its current degree and root. This results in a O(1) find operation but, O(samples) worst case union-operation. However, this can be improved by using a tree model to get O(logn) Find() and Union() operation.

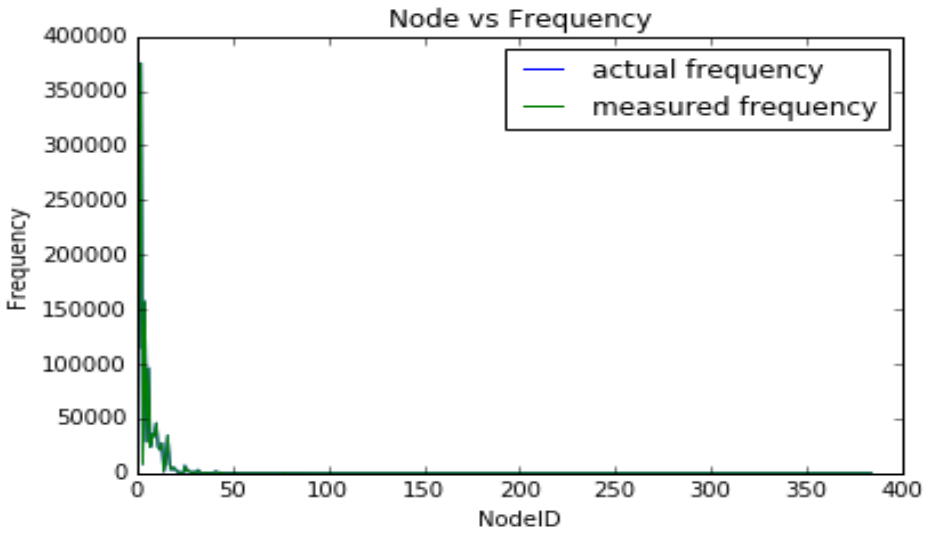

Figure 2: Location Ids against the frequency measured & ground truth

The reason to do this, is to maintain the root of each cluster to be the highest/lowest degree node in the component. This allows us to do the merge/union based on the node's degrees as against using component size as in a standard disjoint set implementation.

Experiments & Analysis

Most frequent trips

We set the parameters to be A = 100 and B = 100. Figures[2-4] represents the results obtained at the end of 1 month. Since these are pretty standard, we used a value that follows the theoretical bounds. Figure [3] shows the error plotted against the distinct location IDs observed during this time interval (1 month), covering a total of 1,441,172 trips made. As we can see, the maximum error seen is 1.0 (i.e, the frequency of a trip is only off by 1). This satisfies our initial guarantees.

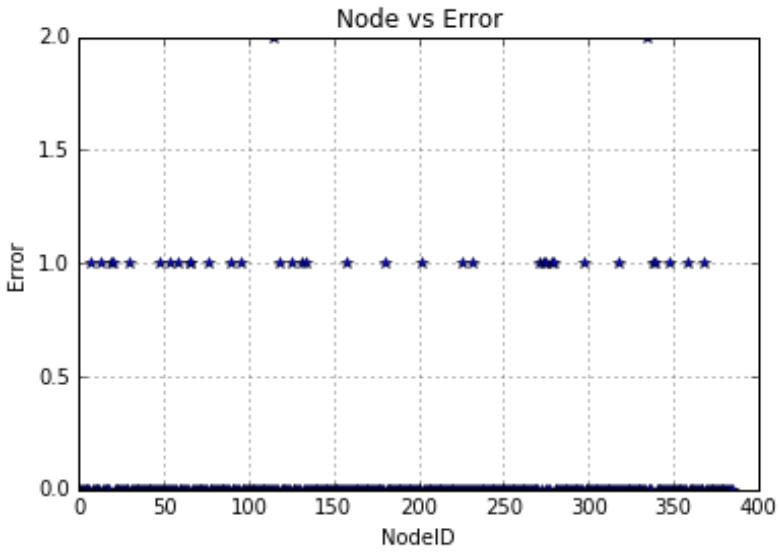

Figure 3: Location ID against the error

Figure [4b] and [4c] show the frequencies plotted against average price and distance. As expected, we notice that generally the price and the distance follow each other, excepting a few cases where the fare is a lot higher than the distances; this could be attributed to extra fares arising from traffic/other delays.

Figure [4(a)] shows a histogram of the frequencies. Looking at this we can notice that there are indeed only a handful of places where the most-frequent trips seem to be coming from. The highest bin in the histograms appear very early on, indicating that most destination pairs are actually quite evenly distributed. The fewer nodes that fall in the highly-frequent zones are the weaker links which could give tremendous insight into the city's transportation bottlenecks.

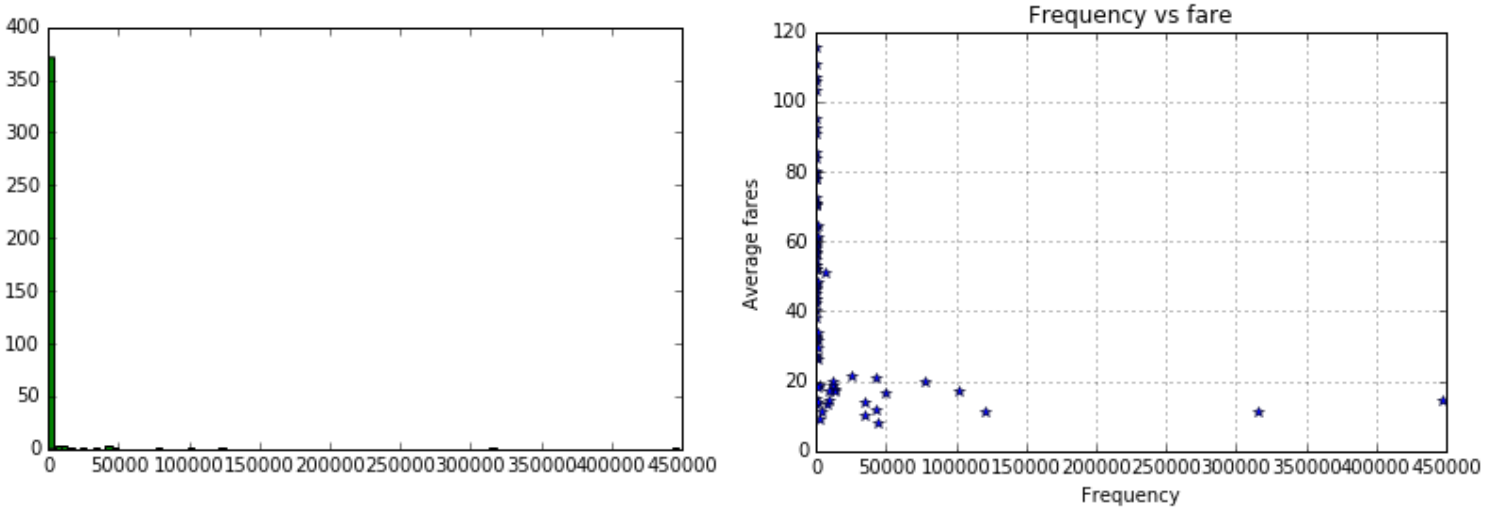

Figure 4: (a)Frequency histogram, (b) frequency plotted against the average fare, (c) frequency plotted against average distance

One of the interesting finds, was to try and find one of the most frequent destination pairs. When we traced back the locations, (even though we couldn't retrieve the exact latitude, longitude anymore, we get a location grid that's somewhere in the locality of the trip shown in figure [5]. However, we have not independently identified this with the actual data/practically verified it's significance. In any case, 1 month might be too small a time frame to vette an extremely popular taxi route.

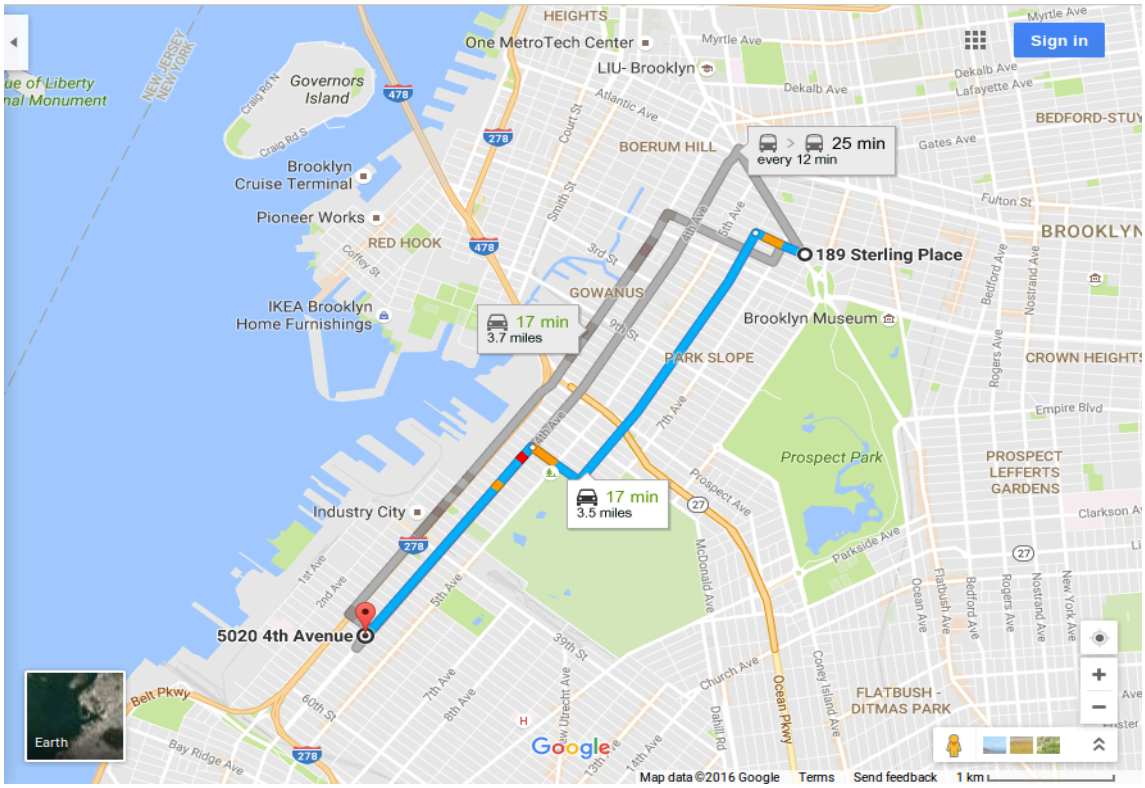

Figure 5: Most frequent trip

Highly connected networks:

Synthetic dataset

We created a synthetic dataset with 1000 nodes, 9345 edges and 10 clusters, pairwise cut size <=5 (Figure[6]). Figures [7a] & [7b] show the results obtained when the sample size is varied from 0.08m to 0.03m, where m = the number of edges. Clearly, we can see that as we reduce the sample size, we are able to find more disconnected clusters. This aligns with our claims and discussions above. The more edges we sample, the more likely it is that we will merge 2 less connected clusters.

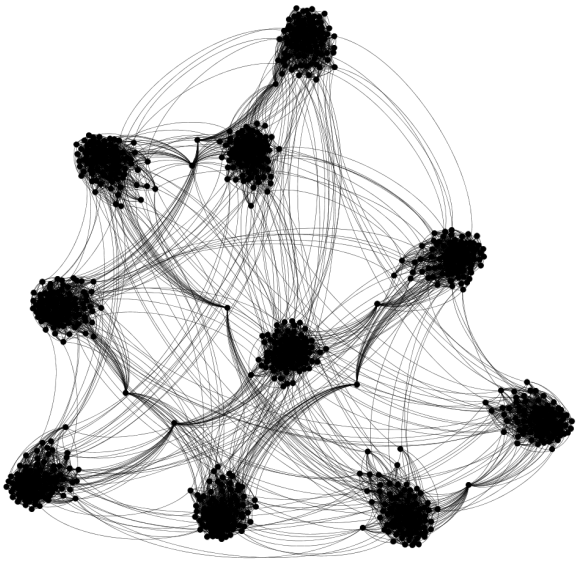

Figure 6: Synthetic data

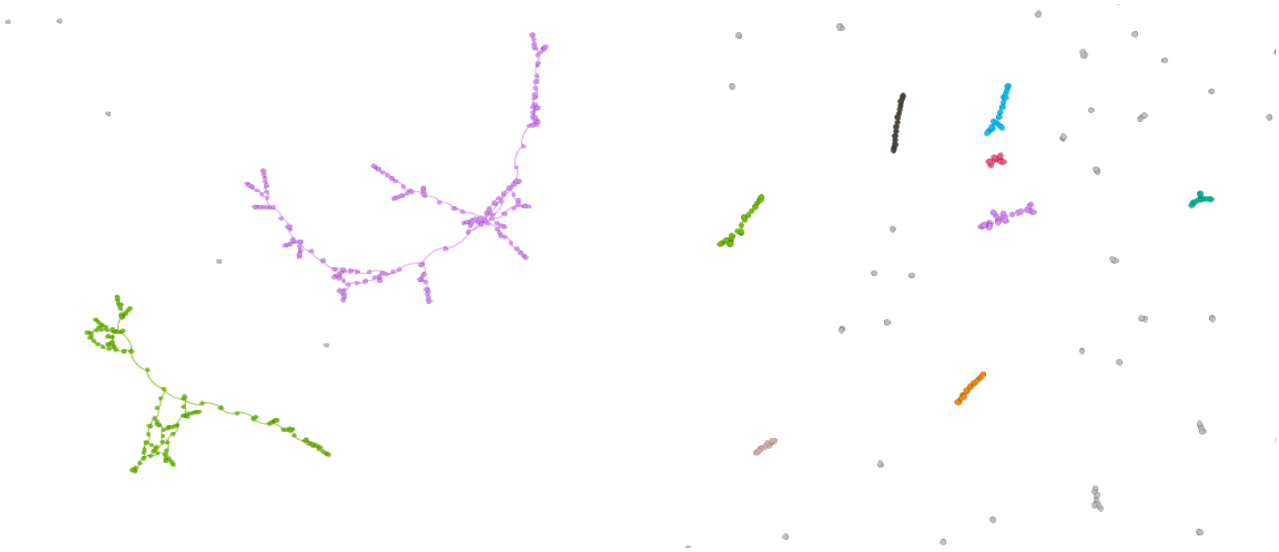

*Figure 7: Algorithm 1 results on sample size (a) 0.08*m and (b) 0.03*m*

NYC dataset - Figure [8a], represents the data obtained over a span of 24 hours on 1 January 2016. As we can see, the graph is very densely connected and even after sampling the graph to roughly 10% of the original size, we still obtain a very dense graph (Figure[8b]). The first algorithm performs very poorly for this dataset, because the difference in the connectivity of the clusters isn't very far from the inter-cluster connectivity. In real life datasets, we don't have any direct way of guessing the values for k1 and k2 (from our lemmas). So, finding the right sample size in this narrow window, even with the bounds we have shown, is very hard.

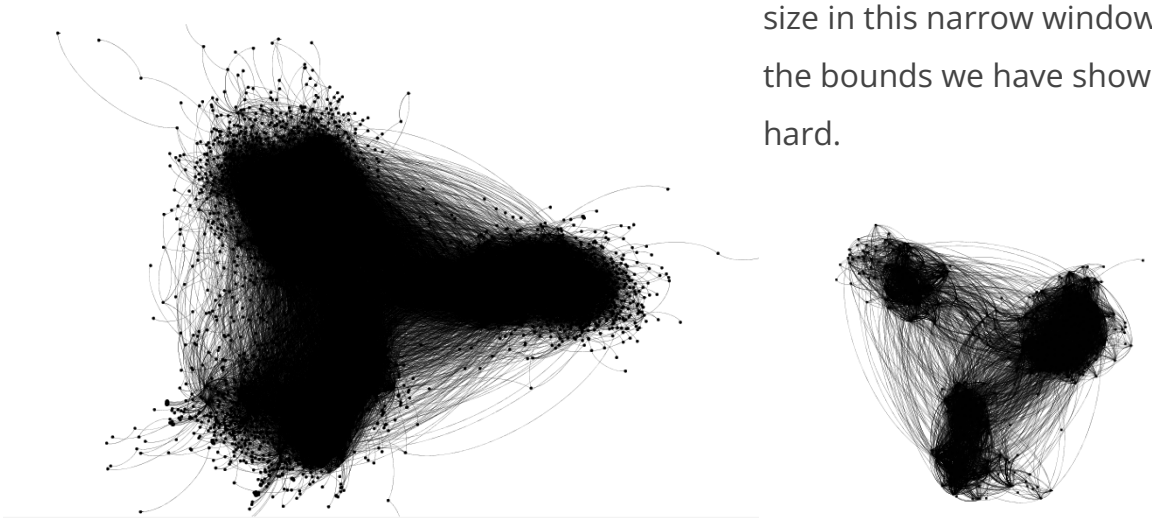

Figure 8: (a) NYC data on 01-01-2016 (b) 10% of NYC data on 01-01-2016

Algorithm 1 finds one big connected component with many fringe vertices or finds extremely disconnected components. (Figures [9a,9b])

*Figure 9: Algorithm 1 results on NYC data Fig 8 with sample size (a) 0.05*m (b) 0.005*m*

In the improved version of our algorithm, we aimed to rectify this, by updating the merging condition. We only merge clusters when their combined internal connectivity is higher than the original clusters. And we see the corresponding results in Figures [10a,10b] respectively (for the same sample set sizes), where in we are able to recover a most of the graph structure. This makes sense, because, we parse the edges in decreasing order of their degrees (i.e, we first merge the highly connected edges into components, before reaching edges with lesser weights). So, naturally when we arrive at the fringe/cut edges between two clusters, the clusters are already strongly connected thus rejecting the merge requests. In both cases, it is clear that the improved algorithm performs better in recovering the structure of the original networks/clusters.

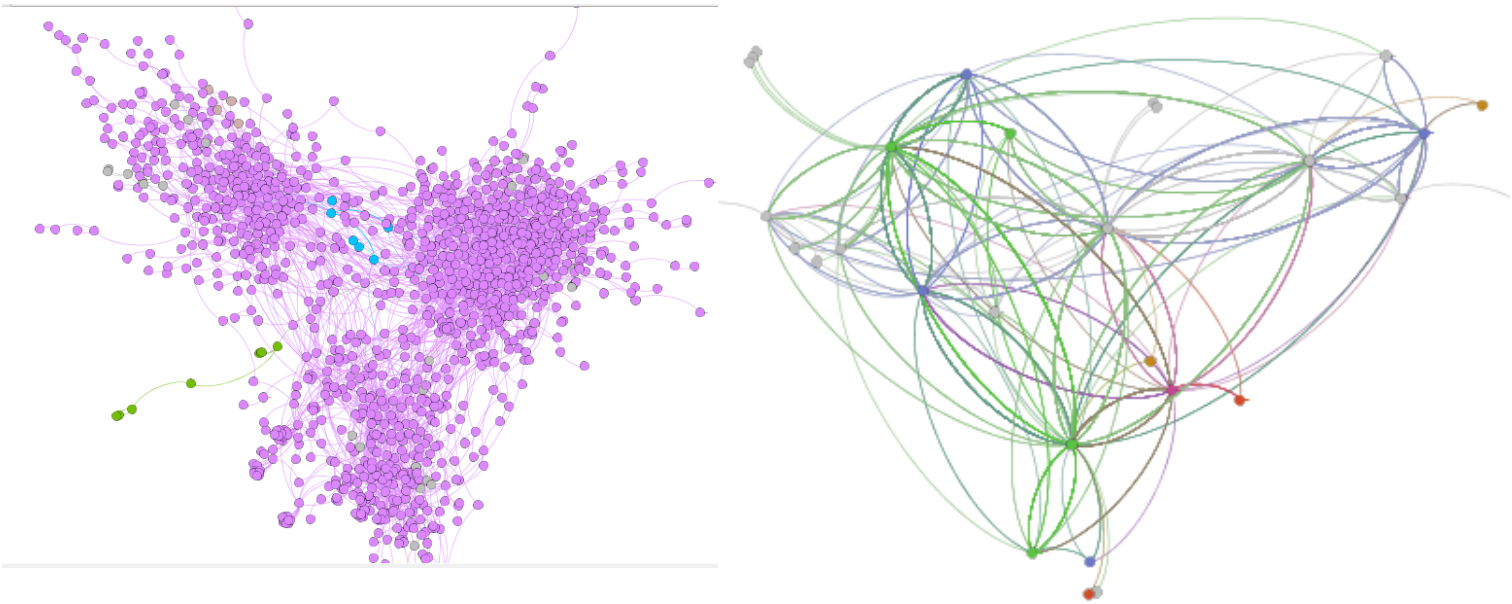

*Figure 10: Algorithm 2 results on Fig 8 data with sample size (a) 0.05*m (b) 0.005*m*

We couldn't experiment more with our algorithms to compare the results by varying the parameters against one another, or plotting the other variables like price, distance etc against these networks to see if they hold any interesting properties owing to lack of time. But, we believe these would be very interesting to analyze and learn from.

Conclusion

In conclusion, we find that even though we can sometimes provide theoretical bounds, we may not necessarily get predictably good/decent results. A lot of times, the parameters used in theoretical analysis are oversimplified, unavailable or simply insufficient to get meaningful results directly. We often have to work with the individual dataset, understand its particular properties and modify the algorithm in order to even begin to see some useful results.

Some future directions for this project include -

- Approximate k in Lemma 2.
- Approximating d_{max} in Lemma 3.
- Simultaneous approximation of k and N_p until the results converge
- Try recursively approximating algorithm 1 until we have k components, however we will have to find better bounds for the sample size.

The following github repository contains our codes:-
https://github.com/div1090/CS5234

References

[1] *NYC Taxi & Limousine Commission - Trip Record Data*. (n.d.). Retrieved November 13, 2016, from http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

[2] Eldawy, A., Khandekar, R., & Wu, K. (2012). *Clustering Streaming Graphs*. 2012 IEEE 32nd International Conference on Distributed Computing Systems. doi:10.1109/icdcs.2012.20

[3] *Graph Clustering* - cs-people.bu.edu. (n.d.). Retrieved November 13, 2016, from http://cs-people.bu.edu/evimaria/cs565/lect19-20.pdf

[4] Cormode, G. (2014). Count-Min Sketch. *Encyclopedia of Algorithms,* 1-6. doi:10.1007/978-3-642-27848-8_579-1

[5] Saran, H., & Vazirani, V. (n.d.). *Finding k-cuts within twice the optimal*. [1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science. doi:10.1109/sfcs.1991.185443