Dynamic Community Detection using GPU

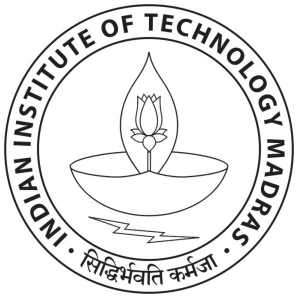
A Project Report

submitted by

SRIVATSAN R

in partial fulfilment of requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May 2018

THESIS CERTIFICATE

This is to certify that the project titled **Dynamic Community Detection using GPU**, submitted by **Srivatsan R (EE14B058)**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this project, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Rupesh Nasre
Research Guide
Assistant Professor
Dept. of Computer Science & Engineering
IIT-Madras, 600 036

Dr. Krishna Jagannathan
Research Co-Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 9th May 2018

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the people who have helped and supported me both in this work and on my path to this point.

I am extremely grateful to **Dr. Rupesh Nasre** who has provided me with expert guidance and continuous encouragement throughout to ensure that this project progresses smoothly since its commencement till its completion. His broad view and exceptional insight led me to explore deep into the wonderful field of community detection and helped me avoid so many detours during the progress of my research. I would also like to express deepest appreciation towards the professor for granting me access to computing resources that were of immense help to me for progressing over my project.

ABSTRACT

KEYWORDS: Community Detection; Louvain Method; GPU; Parallel Algorithm; Dynamic GPU Algorithm

Detecting communities in a network is a classification problem, where nodes in a network are classified into multiple sets such that the connection of nodes within a set is denser than the connection between nodes from different sets. Most of the real world networks are dynamic. Real world systems like social networks (Facebook, LinkedIn and Twitter) are evolving continuously and expanding dramatically in terms of size. So, there is a need for incremental/dynamic community detection. In this paper, we present a highly scalable community detection algorithm on GPU based on the Louvain method. We also present a way to process dynamic graphs to find their community structure. Using four real world networks with millions of edges we illustrate that our dynamic community detection method is both fast and accurate.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Community detection for a network is a classification problem. Nodes in a network are classified into multiple sets such that the connection of nodes within a set is denser than the connection between nodes from different sets. If the nodes of a network can be easily classified into different sets satisfying the above criteria, we say that it has community structure. Almost all of the real world networks exhibit community structure. Social networks show community structures in terms of location, interests, occupation, etc.

Detecting the community structure of a network can be very useful. Each community acts as a single meta-node which can be used to create a reduced version of the network and thus makes the processing of the network easier. If we concentrate only on the average properties of a network we will miss out some interesting features of the network, since communities usually have very different properties than the average properties. Community detection thus enables us to more accurately study a network.

There is no standard mathematical instrument to quantify the community structure of a network. There has been many proposed algorithms and metrics for community detection in the literature. For an overview on them see Fortunato (2010). The modularity metric proposed by Newman and Girvan (2004) is often used to measure the community structure of a network. Modularity measure compares the density of links within communities of a network with the average density of links in a random network defined suitably. It is based on the idea that a suitably defined random network will have no community structure.

The Louvain Method is a greedy algorithm for community detection and it is one of the most popular algorithms. It moves nodes from one community to another in such a way that it increases the modularity of the network. It is a two phase algorithm that computes a hierarchy of clusters. The method has various applications in areas such as analyzing social networks (Liu *et al.*, 2011; Traud *et al.*, 2012), mapping of human

brain networks (Meunier *et al.*, 2009; Zuo *et al.*, 2011), and classification of scientific journals (Wallace *et al.*, 2009).

Networks that change with time are called dynamic networks. Most of the real world networks are dynamic. Real world networks like social networks (Facebook, LinkedIn and Twitter) are evolving continuously and expanding dramatically in terms of size. So, there is a need for incremental community detection. Performing Static Louvain method frequently will consume a lot of time. There are also some parallel implementations of the Louvain method (Lu *et al.*, 2015; Naim *et al.*, 2017) which provide high speedups, but still are not fast enough to cater to the need of our social network graphs that change very frequently. So, running the static Louvain method whenever the graph changes is not a feasible solution in terms of the execution time, and there is a need for a faster dynamic community detection algorithm.

In this paper, we present a highly scalable community detection algorithm on GPU based on the Louvain method. The main difference between the previous implementations (Lu *et al.*, 2015; Naim *et al.*, 2017) is that we don't have an explicit aggregation phase in our algorithm. The elimination of an explicit aggregation phase reduces the execution overhead involved in it. We also present a way to process dynamic graphs to find their community structure. We present a dynamic community detection algorithm which is both fast and accurate.

CHAPTER 2

THE LOUVAIN METHOD

A network can be modeled with a we graph $G = (V, E, W)$, where V is the vertex set, E is the edge set and $W = \{w_{(i,j)} \mid (i,j) \in E\}$, be the weights associated with the edges in E. Let $c_1, c_2, c_3, \ldots, c_k$ be k disjoint sets into which V is partitioned. Let $C(i)$ denote the community of vertex i, where $C(i) \in \{c_j \mid 1 \leq j \leq k\}$. We use k_i to denote the in-degree of vertex i. Let a_{c_j} be the sum of k_i of all vertices in community c_j. We define another term $e_{i \to c_k}$ which is the sum of edge weights from vertex i to all the vertices of c_k.

$$k_i = \sum_{j \in N[i]} w_{(i,j)}, \text{ where } N[i] \text{is the neighbor set of } i \tag{2.1}$$

$$a_{c_j} = \sum_{i \in c_j} k_i \tag{2.2}$$

$$m = \sum_{w \in W} w \tag{2.3}$$

$$e_{i \to c_k} = \sum_{j \in c_k} w_{(i,j)} \tag{2.4}$$

Now, with all these notations defined, we can introduce and define modularity. Modularity is a measure that quantifies a community partition of a graph. It is based on the idea that random graph has no community structure and by comparing any graph with such a random graph will give us some quantification of its community structure. It compares the density of links within communities of a network with the average density of links in a random network defined suitably. It takes a real value between -1 and 1 (Newman, 2004) and is defined as-

$$Q = \frac{1}{2m} \sum_{i \in V} e_{i \to C(i)} - \sum_{c \in C} \frac{(a_c)^2}{4m^2} \tag{2.5}$$

$$= \sum_{c \in C} \left(\frac{e_{cc}}{2m} - \frac{(a_c)^2}{4m^2}\right), \tag{2.6}$$

where $e_{c_i c_j} = \sum_{v \in c_i} e_{v \to c_j}$. Using modularity to find a community partition is a

NP-hard problem (Brandes *et al.*, 2008). Louvain method is a greedy algorithm that optimizes modularity (Equation 2.5) by moving a node from its community to another community which leads to an increase in the modularity. The change in modularity when a node i from community $C(i)$ is moved to a neighboring community $C(j)$ is given by

$$\Delta Q = \frac{e_{i \to C(j)} - e_{i \to C(i) \setminus \{i\}}}{m} + k_i \frac{a_{C(i) \setminus \{i\}} - a_{C(j)}}{2m^2} \tag{2.7}$$

Louvain method has two phases - Optimization phase and Aggregation phase. In the beginning, we assign every node as a separate community by themselves. In the optimization phase the algorithm iterates over all the vertices and computes the change in modularity for the cases when it moves to each of its neighboring communities. The change in modularity is calculated using Equation 2.7. The algorithm for each node picks the destination community that yields the highest positive modularity change and moves the node to that community.

In the aggregation phase, a new graph is constructed that contains all the communities as nodes and the sum of inter community edge weights as its new edge weight. The intra community edges become an equivalent self-loop on the corresponding node. The two phases are executed repeatedly till there is an increase in the modularity of the graph.

The main challenge in the optimization phase is the computation of the change in modularity value (ΔQ) for all possible movements of a node. For every node, we have to compute ΔQ when it moves to each of its neighboring community. The main challenge in doing this is that we have to calculate $e_{i \to c_j}$ for all c_j that are neighbors to i. This is done by iterating over all the neighbors $j \in N[i]$ of a vertex i and accumulating the value of $w_{i,j}$ in a hash table using c_j as key. Calculating a_c is straightforward. Initially the value of $a_C(i) = k_i$ and it can be updated as and when nodes move from one community to another. The aggregation phase follows a similar routine to construct the compressed graph.

Our algorithm doesn't involve moving nodes between communities. We move the whole community and merge it with other communities. This makes the algorithm faster without needing an explicit aggregation phase. More detailed explanations are provided in Chapter 4. The change in modularity when a community c_i merges with c_j

and becomes c_j is given by -

$$Q_1 = (\frac{e_{c_i c_i}}{2m} - \frac{(a_{c_i})^2}{4m^2}) + (\frac{e_{c_j c_j}}{2m} - \frac{(a_{c_j})^2}{4m^2}) + \sum_{c \in C \setminus \{c_i, c_j\}} (\frac{e_{cc}}{2m} - \frac{(a_c)^2}{4m^2}) \tag{2.8}$$

$$Q_2 = (\frac{e_{c_i c_i} + e_{c_j c_j} + 2e_{c_i c_j}}{2m} - \frac{(a_{c_i} + a_{c_j})^2}{4m^2}) + \sum_{c \in C \setminus \{c_i, c_j\}} (\frac{e_{cc}}{2m} - \frac{(a_c)^2}{4m^2}) \tag{2.9}$$

$$\Delta Q = (\frac{2e_{c_i c_j}}{2m} - \frac{2(a_{c_i} a_{c_j})}{4m^2}) \tag{2.10}$$

Again, the main challenge here is to calculate $e_{C(i)C(j)}$ which can be calculated in a similar way as described above. Instead of maintaining a hash table for each vertex, we maintain a hash table for every community and accumulate $w_{i,j}$ using $C(j)$ as key. Using Equation 2.10 we can calculate the change in modularity for every possible movement of the community.

CHAPTER 3

RELATED WORK

Many algorithms have been proposed in the literature to detect communities in a network (Fortunato, 2010; Blondel *et al.*, 2008; Newman and Girvan, 2004; Newman, 2006). In Newman and Girvan (2004), they have used a concept of edge betweenness to find communities in a network. In Newman (2006), Newman puts forth a concept of modularity to quantify the strength of a community partition of a network. The Louvain method(Blondel *et al.*, 2008), is a state of the art method that uses greedy modularity optimization to detect communities.

There have been many successful efforts to parallelize Louvain method (Lu *et al.*, 2015; Naim *et al.*, 2017). In Lu *et al.* (2015), they have parallelized the Louvain method for CPU. Their implementation is written using OpenMP and a comparison of this implementation with our algorithm is provided in the Chapter 6. In Naim *et al.* (2017), they have provided a scalable GPU community detection algorithm for static graphs based on Louvain method.

Community detection on dynamic networks has been explored by a few works. In Shang *et al.* (2014), they classify dynamic network changes into four cases that consider on edge addition/increase. For each case, they provided heuristics and have to deal with dynamic networks. For instance, they decide with the help of their heuristics whether to not disturb the community structure or to merge the two communities. They select the action which yields a higher modularity increase. However, they don't consider edge deletion as a network change and have only provided heuristics to handle the edge addition case.

In Nguyen *et al.* (2014), they provide heuristics to handle both addition and deletion of edges in a dynamic network. They treat network changes as a collection of simple events, and update or discover the new community structure based on the network's community structure history. However, they have provided all the solutions for a network which is unweighted. They have also claimed that adding an intra-community edge will not break a community into smaller communities to maintain the optimal

community structure, we provide an example in the Chapter 4 to show that an intra-community edge addition can indeed break the community into smaller communities.

In dSLM (Aktunc *et al.*, 2015), for each dynamic network changes, they keep the old community structure of the graph unchanged, and using that as a starting point they run SLM (Waltman and Van Eck, 2013), a variant of Louvain method. Although it could reduce the dynamic community detection running time dramatically on network changes but its accuracy decreases a lot. The method seems to work for edge additions but not for edge deletions.

CHAPTER 4

THE GPU ALGORITHM

In this chapter, we describe our static and dynamic community detection algorithm. There has been some implementations of parallel Louvain method for static graphs. Those implementations were mainly based on parallelizing access to nodes (Lu *et al.*, 2015) or edge (Naim *et al.*, 2017). Our algorithm is a fine grained implementation of the parallel Louvain Method that parallelizes access to every node and their edges.

We store the graph in the global memory using the Compressed Sparse Row (CSR) format. Thus the graph $G = (V, E, W)$ is stored in contiguous memory location using three arrays $vtxPtr$, $edges$ and $weights$ of size $1 + |V|$, $2|E|$ and $2|E|$. The edge weights of all the edges coming from vertex i are stored in positions $vtxPtr[i]$ up to position $vtxPtr[i + 1]$ in $weights$ array and the corresponding neighbors in $edges$ array. Storing the graph in CSR format and parallelizing the edges access for each node enables the GPU to access the memory in a coalesced manner. For some standard tasks like calculating cumulative sum and copying data to device, we use routines from thrust library which are optimized for these tasks. We use CUDA atomics in places where access to individual memory elements has to be sequentialized. It is required in the implementation of the concurrent hast table. Our implementation of Louvain method is lock free.

4.1 Static Community Detection

Our implementation of the Static Community Detection Algorithm is simply a modification of the Louvain Method. The main difference between our algorithm and the other parallel implementations is that we don't have an explicit aggregation phase. Eliminating the aggregation phase removes the overhead involved in reconstructing the graph which involves a lot of data movement and sequentiality since we are dealing with graphs represented in CSR format.

We start by assigning all the nodes as a community by themselves (which is done before calling Algorithm 1). We also calculate m and k_i values in parallel (lines 2 and 3). Initially $a_{c_i} = k_i$ (line 4). After each iteration, the value of a_c is updated depending on the merged communities (line 23). In the optimization phase, the algorithm calculates the change in modularity when each of the communities merges with each of its neighbors. So, for each community we have an array of modularity change values corresponding to the community's merger with each of its neighboring communities. Now, we calculate the maximum positive modularity change that can be obtained for each of the communities. The maximum value is calculated using reduction method in parallel. The neighboring community which yields the maximum modularity change is the potential destination community for a community. The algorithm iterates over the optimization phase until the gain in the modularity of the network falls below a predefined threshold value.

Algorithm 1 Modularity Optimization

```
1:  procedure COMMDET(mask, propagate = False)
2:      Calculate m in parallel
3:      Calculate k_i for each i ∈ V in parallel
4:      Assign a_i = k_i in parallel
5:      Initialize hashTable[V], an array of V Hash tables.
6:      do
7:          for each i ∈ V in parallel do
8:              if mask[C[i]] == 1 then
9:                  for each v ∈ N[i] in parallel do
10:                     addVal(hashTable[C[i]], C[v], wt_{i,v})
11:         for each i ∈ V in parallel do
12:             for each e ∈ hashTable[i].values in parallel do
13:                 Calculate ΔQ using equation 2.10 inplace
14:         for each i ∈ V in parallel do
15:             newComm[i] = argMin_key(hashTable[i].values)
16:         resolveConflicts()
17:         for each i ∈ V in parallel do
18:             j = newComm[i]
19:             mask[i] = (propagate)?(mask[i] ∨ mask[j]) : (mask[i] ∧ mask[j])
20:         for each i ∈ V in parallel do
21:             C[i] ← newComm[C[i]]
22:         for each c ∈ C in parallel do
23:             Update commSize_c and a_c
24:     while modGain < threshold
```

As mentioned earlier, the main difficulty in the optimization phase is to calculate the value of $e_{c_i c_j}$. To calculate this, we access every edge with weight $w_{i,j}$ in parallel

Algorithm 2 Control community movements

```
1: procedure RESOLVECONFLICTS
2:     Declare and Initialize array tempComm with 0
3:     for each c ∈ C in parallel do
4:         fComm = c
5:         tComm = tempComm[c]
6:         if newComm[tComm] == fComm and fComm < tComm then
7:             tempComm[c] = c
8:     newComm ← tempComm
```

and add this value to its corresponding entry in the hash table. For an edge (i, j) we add it to the hash table of the community $C(i)$ and at the position corresponding to the key $C(j)$ (see lines 7 - 10 of Algorithm 1). Algorithm 3 deals with updating a value in the hash table. If an entry is absent it creates a new entry with the given value. We use open addressing and double hashing (Cormen, 2009) in the hash table. So, in lines 2 and 3 we calculate two hashes of the key. We then iterate by generating values of pos as given in the line 6. If at some point we find an entry with the same key which we have, we just atomically add our current value in its corresponding variable (line 9). If the key value at a position is -1 it denotes that the position is empty. In that case we try to atomically swap the key variable with the key value in hand. If it is successful, we have now blocked that position for this key and hence we can update the value corresponding to that atomically. If it failed then we try again at the next position given by line 6.

Algorithm 3 Update an entry in table

```
1: procedure ADDVAL(hashTable, key, value)
2:     h1 = hash1(key)
3:     h2 = hash2(key)
4:     i = 0
5:     do
6:         pos = (h1 + i * h2)%hashTable.size
7:         i = i + 1
8:         if hashTable[pos].key == key then
9:             atomicAdd(hashTable[pos].value, value)
10:        else if atomicCAS(hashTable[pos].key, −1, key) == −1 then
11:            atomicAdd(hashTable[pos].value, value)
12:    while hashTable[pos].key! = key
```

The algorithm maintains a hash table for each community c_i that stores the sum $e_{c_i c_j}$ for each $c_j \in N[c_i]$, where $N[c_i]$ is the set of community neighbors of c_i. These values are stored in a concurrent hash table with the key as their corresponding community id, c_j. With these values we can find the corresponding change in modularity value when

the community is merged with one of its neighbors. We use reduction (see line 15 of Algorithm 1) to find the maximum of the modularity change and populate a $newComm$ array with the index of that community at a position corresponding to the current community. The elements of $newComm$ array will now contain the ids of the potential destination community of each community.

We use some ideas from Lu *et al.* (2015) to control the movements of communities. In particular, we allow a single neighbor community c_i to move to c_j only if $c_i > c_j$ and if there are more than one c_j on merging with whom gives the same maximum modularity, we pick the community with minimum index as the potential destination community. In addition to that if $newComm[newComm[c]] = c$, i.e., when two communities tend to swap their positions, we move c to $newComm[c]$ only if $newComm[c] < c$(see Algorithm 2). This prevents communities from swapping each other's place instead of merging and considerably reduces the number of iterations required for convergence.

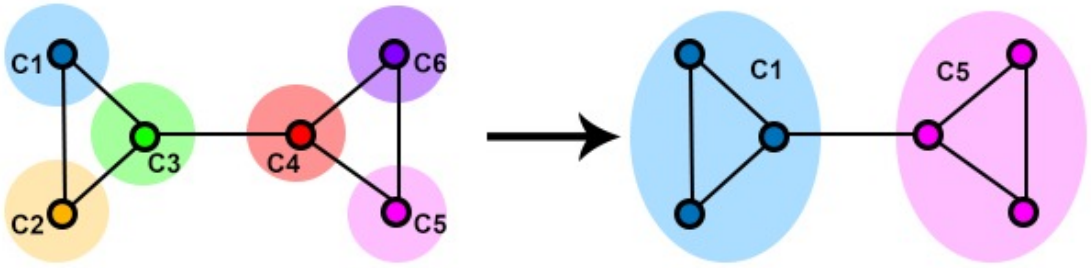


Figure 4.1: The state of the graph initially and after running an iteration of our algorithm.

Community	Modularity change	$newComm$	$newComm$ after $resolveConflict()$
C1	C2:$\frac{5}{49}$, C3:$\frac{4}{49}$	C2	C1
C2	C1:$\frac{5}{49}$, C3:$\frac{4}{49}$	C1	C1
C3	C1:$\frac{4}{49}$, C2:$\frac{4}{49}$, C4:$\frac{2.5}{49}$	C1	C1
C4	C3:$\frac{2.5}{49}$, C5:$\frac{4}{49}$, C6:$\frac{4}{49}$	C5	C5
C5	C4:$\frac{4}{49}$, C6:$\frac{5}{49}$	C6	C5
C6	C4:$\frac{4}{49}$, C5:$\frac{5}{49}$	C5	C5

Table 4.1: Table showing the change in modularity values when each community merges with one of its neighboring community and the final destination community of each community.

We now provide an example to explain the steps involved in our Static Community Detection algorithm. Consider a graph as shown in Figure 4.1 with 6 nodes and 7

edges. Initially we assign each of the nodes to an unique community. Now, for each community in parallel we calculate the change in modularity value if it were to merge with one of its neighboring community. See Table 4.1 for all the change in modularity values that will be calculated by our algorithm. After calculating the values we store the community id corresponding to the maximum modularity change for each community in $newComm$ array. For C3 and C4, we can see that there are two neighboring communities on merging with whom we get the same maximum modularity change of $\frac{4}{49}$. In such a case we populate the $newComm$ array with the community id of the community with the smallest id value. So, for C3 it will be C1 and for C4 it will be C5. As we can see from Table 4.1, there are some community pairs like (C1, C2) and (C5, C6) which will swap to each other's position if $newComm$ array values were the destination community. At this point we call the procedure $resolveConflicts()$ which will use the logic from section 4.1 to control such community swapping. In the rightmost column of Table 4.1, we have the destination community for each of the communities. C1 and C5 which were previously assigned to move to C2 and C6 respectively are now staying in their old community itself.

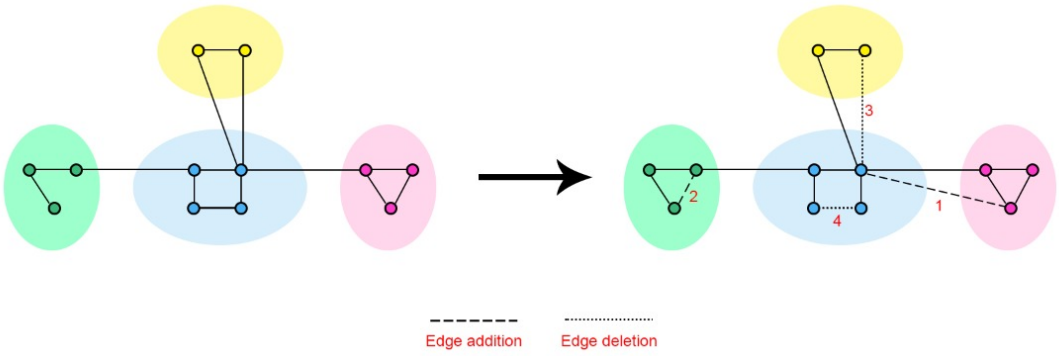


Figure 4.2: A pictorial representation of edge updates - 1. Inter-community edge addition, 2. Intra-community edge addition, 3. Inter community edge deletion, 4. Intra-community edge deletion.

4.2 Dynamic Community Detection

In this section, we describe the various cases that arise when the network is dynamic and how to efficiently perform operations on the network so as to update its community structure without any significant compromise on the modularity value. See Table 4.2

for a summary of the different cases involved in dynamic community detection. We divide the discussion here into single edge and multi edge updates.

Type of Update	Intra-Comm. Addition	Inter-Comm. Addition	Intra-Comm. Deletion	Inter-Comm. Deletion
Single Edge (No Propogation)	No change	Merge/No Merge	Dissociate the community and Set $mask[c] = 1$ for the involved communities. Run SCD.	No change
Multi Edge (Propogation)	Dissociate the community and Set $mask[c] = 1$ for the involved communities. Run SCD.	Set $mask[c] = 1$ for the involved communities. Run SCD.	Dissociate the community and Set $mask[c] = 1$ for the involved communities. Run SCD.	Set $mask[c] = 1$ for the involved communities. Run SCD.

Table 4.2: A summary of various cases involved in Dynamic community detection and the proposed solution. In the table SCD is Static Community Detection algorithm.

4.2.1 Single Edge Updates

This subsection deals with the cases that arises when a single edge is added or deleted from the graph.

Intra Community Edge Addition

When an edge is added connecting two nodes of the same community, since we are only strengthening a community, the modularity value always increases. So, we can simply leave the community assignment of the network unchanged. An important point to be noted here is that even though there is an increase in modularity value, this value may not be the optimal value but still can be tolerated for the sake of execution time.

Inter Community Edge Addition

When an edge is added connecting two nodes of different communities, two possibilities arise - the two communities can either be merged or left as they are. The decision

to merge or to not merge is made based on the modularity change in both the cases. Whichever leads to a higher increase in modularity is preferred over the other.

$$Q_1 = \frac{1}{2m + 2w_{ij}}[(e_{C(i)C(i)} - \frac{(a_{C(i)} + w_{ij})^2}{2m + 2w_{ij}}) + (e_{C(j)C(j)} - \frac{(a_{C(j)} + w_{ij})^2}{2m + 2w_{ij}})] + K \tag{4.1}$$

$$Q_2 = \frac{1}{2m + 2w_{ij}}[e_{C(i)C(i)} + 2e_{C(i)C(j)} + e_{C(j)C(j)} + 2w_{ij} - \frac{(a_{C(i)} + a_{C(j)} + 2w_{ij})^2}{2m + 2w_{ij}}] + K \tag{4.2}$$

$$\Delta Q = \frac{1}{2m + 2w_{ij}}[2e_{C(i)C(j)} + 2w_{ij} - \frac{2(a_{C(i)} + w_{ij}) * (a_{C(j)} + w_{ij})}{2m + 2w_{ij}}] \tag{4.3}$$

$$\Delta Q = \frac{1}{2m'}[2e'_{C(i)C(j)} - \frac{2a'_{C(i)}a'_{C(j)}}{2m'}] \tag{4.4}$$

where $m', e'_{c_i c_j}, a'_c$ are all the corresponding values of m, e, a_c after adding the inter community edge. So, whenever $e'_{C(i)C(j)} > \frac{a'_{C(i)}a'_{C(j)}}{2m'}$, we merge the two communities, otherwise we leave them as they are.

Inter Community Edge Deletion

This case is similar to the intra community edge addition. There will be an increase in the modularity value when we leave its community structure unaltered. Again, this value may not be the optimal value but still can be tolerated for the sake of execution time.

Intra Community Edge Deletion

When we delete an intra community edge, the community gets weak and tends to split into smaller communities. The natural tendency of Louvain method is to reduce the number of communities at the end of every iteration. So, if we blindly run Louvain method on this network with their previous community assignment as the starting point, nothing happens.

Our approach to this case is to break the community into many single node communities (i.e., a community of size c gets converted into c communities of size 1) and then run our Static Community Detection Algorithm only for these node communities (see Algorithm 4). This is done by having a $mask$ array which will hold the value 1 (active)

Algorithm 4 Intra-Community Edge Deletion

```
1: procedure INTRADEL
2:     Delete the edge from graph.
3:     Initialize delComm as the community of deleted edge.
4:     Initialize mask array with 0
5:     for each i ∈ V in parallel do
6:         if C[i] == delComm then
7:             C[i] = i
8:             mask[i] = 1
9:     commDet(mask)
```

for the communities which need further merging with other communities and the value 0 (inactive) for the communities that don't have to be merged with other communities. When a community c_i merges with c_j and becomes the community c_i, the new $mask$ value is given by (see line 19 of Algorithm 1) -

$$mask[c_i] := mask[c_i] \wedge mask[c_j] \tag{4.5}$$

This means that when an active community merges with an inactive community it becomes inactive, whereas when an active community merges with an active community it stays active. This method allows us to contain the merging of communities within a particular part of the network (i.e., we don't propagate it beyond the community). Typically the inactive communities to which a new active community gets merged might also want to merge with other communities. We will discuss more about this in a later sub section.

4.2.2 Multi-Edge Updates

This subsection deals with batch updates, i.e., the cases that arise when multiple edges are added or deleted from the graph at a time. This subsection deals with batch updates on the graph.

Intra Community Edge Updates

The earlier proposed solution for the intra community edge updates favored the execution time in the trade-off between accuracy and execution time. However in this case, since the updates are batched, we give more importance to accuracy of the solution and

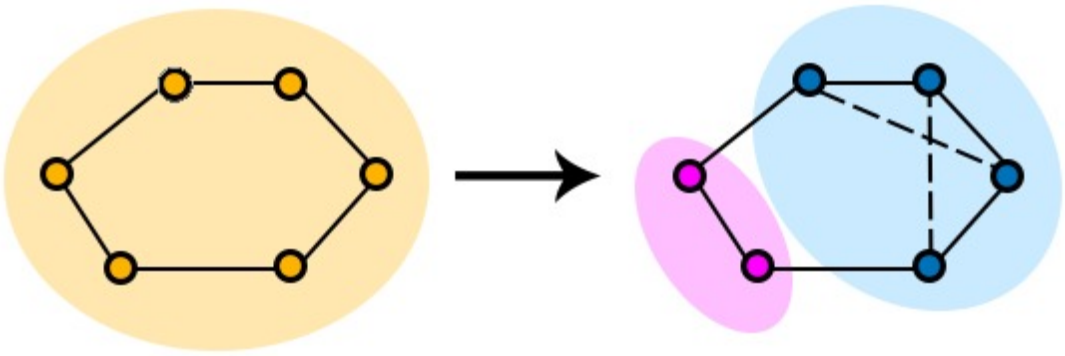


Figure 4.3: Adding more intra-community edges can sometime require the community to break into smaller communities for a better modularity. The graph on the left has $Q = 0$ and the graph on the right has modularity, $Q = 0.125$.

we are liberal on the execution time.

For both intra community edge addition and deletion, there is a tendency for the community to break into smaller communities (see Figure 4.3 and 4.4). To facilitate this process, we break the community into single node communities and then perform Louvain method on these single node communities. Again, we use $mask$ array to specify which communities are active and which are not. As specified in the subsection 4.2.1, the merging of communities need not be confined to a small part of the network. We need to propagate the community merging process beyond the current community (see Figure 4.5). We update the $mask$ array in a different manner to allow for the propagation. When a community c_i merges with c_j and becomes community c_i, the new $mask$ value is given by (see line 19 of Algorithm 1) -

$$mask[c_i] := mask[c_i] \vee mask[c_j] \tag{4.6}$$

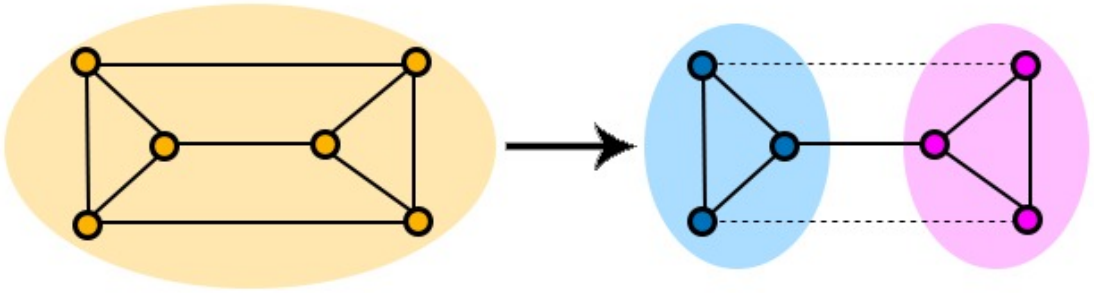


Figure 4.4: Deleting intra-community edges can break the community into smaller communities.

This means that an active community stays active after merging with any other community. This method allows us to propagate the effects of updating an intra community edge.

Inter Community Edge Updates

When an inter community edge is deleted or added, the two communities involved may merge with themselves or merge with their neighboring communities. So, in this case we mark the $mask$ array elements corresponding to these communities with 1 and run Static Community Detection algorithm for these community. Since it is a batched processing, we would again want the effects of update to propagate to the other parts of the network (see Figure 4.5). So, we use Equation 4.6 to update the $mask$ array.

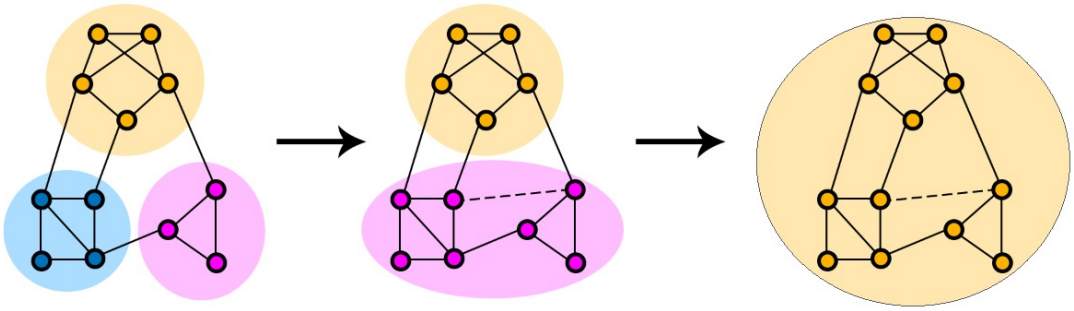


Figure 4.5: The picture shows why merging communities locally is not sufficient. We need to propagate the update effect to get a more accurate modularity value.

Algorithm 5 Multi-Edge Updates

```
1:  procedure BATCHUPDATES
2:      Add/Delete the edges from the graph.
3:      intraC ← Communities with intra edge update
4:      interC ← Communities with inter edge update
5:      for each i ∈ V in parallel do
6:          if C[i] ∈ intraC then
7:              C[i] = i
8:              mask[i] = 1
9:          else if C[i] ∈ interC then
10:             mask[C[i]] = 1
11:     commDet(mask, propagate = True)
```

General Batch Update

When we have a batch of edge additions or deletions where some are intra edge updates and some inter, we first separate them into two sets as given in lines 3 and 4 of Algorithm 5. With the intra community edge updates set we first dissociate the community into single node communities and mark their corresponding $mask$ values as 1. With the inter community edge updates set we mark the corresponding $mask$ value of the communities involved as 1. Now, we simply call the static community detection procedure with $propogate = True$ which enables Equation 4.6 for updating $mask$.

CHAPTER 5

THREAD AND MEMORY ALLOCATION

In this chapter, we describe how we allocate memory and assign thread to different data structures and algorithms.

5.1 Graph Representation

We represent the graph in Compressed Sparse Row(CSR) format. We use global memory to store the graph. In the CSR format all neighbors of a node are stored in contiguous memory location which makes accessing the neighbors of a node very efficient as they will be accessed in a coalesced manner on the GPU. This representation is suitable for static networks. When the network is dynamic, we need to reallocate a new memory and make changes to the CSR representation which will have a lot of sequentiality involved. So, using CSR for a dynamic network will introduce a lot of overhead.

We take ideas from Malhotra *et al.* (2017) for representing dynamic network. They represent a dynamic network with a CSR graph G and a diff-CSR graph G'. G is the base graph which is usually big, whereas G' is a small graph which has the dynamic network changes. Now, whenever some new edges are added we can simply edit the diff-CSR G'. Since G' is small, modifying it will be very fast. When edges are being deleted we simply find the edge's corresponding location in its CSR representation and mark it as -1.

5.2 Kernel Configuration

For each node a dedicated block is assigned while launching a kernel in the GPU. All the blocks are launched with 128 threads. A node with degree less than or equal to 128 gets a dedicated thread for each of its edges. For a node with out degree greater than 128, each thread of the block processes the edges in strides of size 128 (Naim *et al.*,

2017). All computations take place in the kernel. When a global synchronization is required the control is brought back to the host.

5.3 Hash Table

We have implemented a concurrent hash table which is required for the calculation of $e_{c_i c_j}$ for every community $c_i \in C$ where c_j is one of its neighbors. So, we need one hash table for every community. We assign an array of size $\lfloor 1.5|E| \rfloor$ where $|E|$ is the size of $edges$ array in the CSR representation of the network. Now, each community c_i will get an array of size $1.5a_{c_i}$ for using as a hash table. This kind of a segmented hash table is very useful to us. We don't have to redeclare the hash table for every iteration. After each iteration the value of a_{c_i} will change and hence the size of the hash table. Since a_{c_i} represents the sum of degrees of all the nodes in the community c_i, the community can have at most only a_{c_i} neighboring communities. So, having the size of hash table 1.5 times of that value is enough to store all the data. Our hash table implementation is an extension of Naim *et al.* (2017).

CHAPTER 6

EXPERIMENTS

In this chapter we evaluate the effectiveness of our proposed algorithm and its implementation. We implemented our algorithm using CUDA C++. We performed all the experiments involving our algorithm in a Tesla P100-PCIE-12GB GPU. For the experiments which involved running the parallel Louvain method OpenMP code, we ran it on Intel Xeon E5-2640 v4 @ 2.40GHz CPU which has 40 cores. The GPU is attached to the same machine. We use four large datasets as shown in Table 6.2 for the experiments.

6.1 Static Community Detection

6.1.1 Performance

We first analyze the performance of our algorithm in terms of both time and modularity and compare it with Lu *et al.* (2015).

Dataset	Modularity(GPU)	Modularity(CPU)	Time(GPU)	Time(CPU)
com-Amazon	0.89815	0.924185	0.12s	1.47s
com-Youtube	0.608386	0.71082	0.46s	9.30s
com-LiveJournal	0.686947	0.738955	1.59s	69.26s
com-Orkut	0.511545	0.65159	3.55s	64.09s

Table 6.1: Performance of our Static Community Detection algorithm for GPU compared to the Parallel Louvain Method of Lu *et al.* (2015) for CPU

Dataset	V	E
com-Amazon	334,863	1,851,744
com-Youtube	1,134,890	5,975,248
com-LiveJournal	3,997,962	69,362,378
com-Orkut	3,072,441	234,370,166

Table 6.2: Number of nodes and edges in the dataset.

The implementation of Lu *et al.* (2015) parallelizes the access to every vertex as apposed to our algorithm which parallelizes the access to every edge. In their implementation, they use graph coloring (Lu *et al.*, 2017) to pre-process the network which leads to their high modularity value compared to our algorithm. From the experimental data provided above (Table 6.1), we can see that our algorithm gives better results in terms of execution time and comparable results in terms of modularity.

6.1.2 Statistics

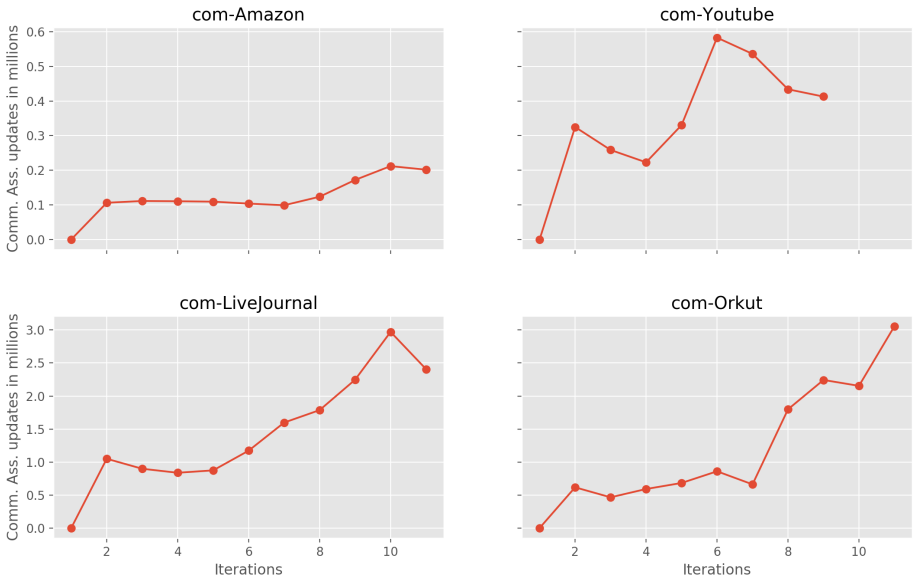


Figure 6.1: Iteration wise community changes.

We have plotted some statistics obtained by executing the static community detection algorithm on four graph datasets - com-Amazon, com-Youtube, com-LiveJournal, com-Orkut (Yang and Leskovec, 2015) as shown in Figures 6.1 and 6.2. In Figure 6.1, we can see a general increasing trend for all the graph datasets, which can be attributed to the increasing size of the communities with iteration. Since, the community size increases with iteration even if a small number of communities merge, the total number of nodes with a change in their community assignment is large.

In Figure 6.2, we have plotted the number of unique communities at the end of each iteration. The semi-log plots are almost linear for all the datasets, which implies that the number of unique communities falls exponentially with iteration. Also, for all the

datasets, the algorithm converges fast in about 10 iterations.

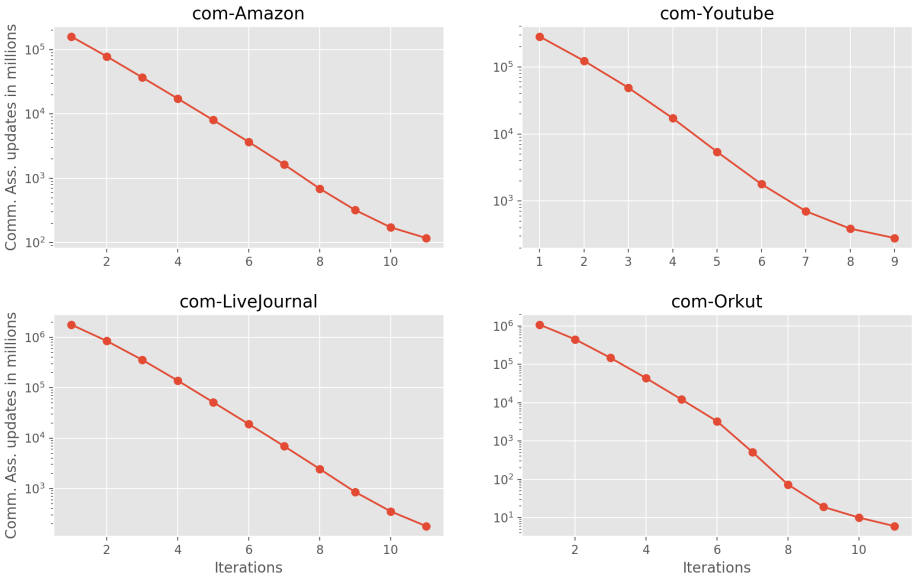


Figure 6.2: Semi-log Plots describing the number of unique communities at the end of each iteration.

6.2 Dynamic Community Detection

In this section, we perform experiments to demonstrate the speed and accuracy of our dynamic community detection algorithm.

6.2.1 Single Edge Updates

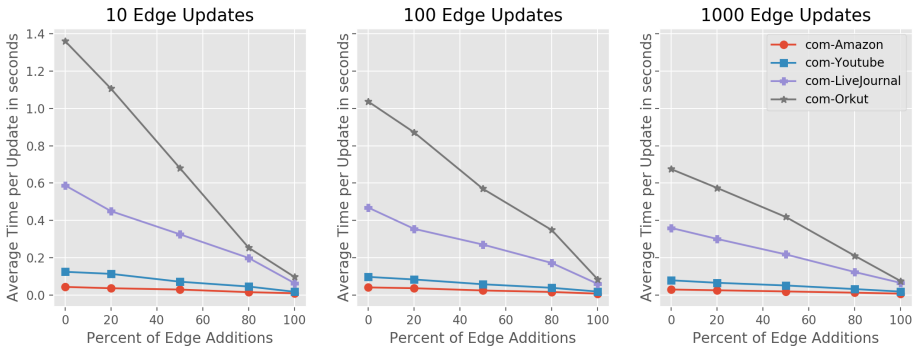


Figure 6.3: Average time per update for a varying number of edge additions and deletions

In Figure 6.3, we have plotted the average time per update by varying the fraction of edge additions and deletions in the update set. We plot for three different update sizes - 10, 100 and 1000. We can see that all the plots are almost linear. Edge deletions are more compute intensive than the edge additions. The updates at the 0% mark are all made up of edge deletions, and the average time for them is the highest. The updates at the 100% mark are all made up of edge additions which take the least amount of time per update. All the plots appear to be a linear interpolation of the 0% and the 100% marks. We also observe that the average time per update reduces with the number of updates which has edge deletions. This is due to the fact that when we delete edges the community size tends to get smaller and processing a smaller community is faster than bigger ones. So, when we increase the number of edge deletions we make the average time per update decreases.

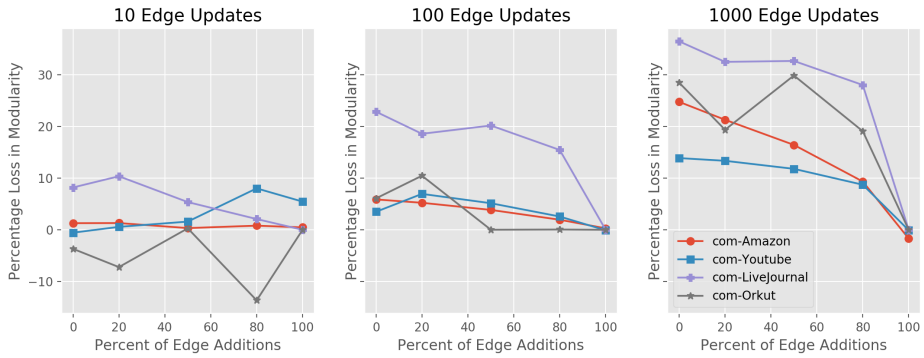
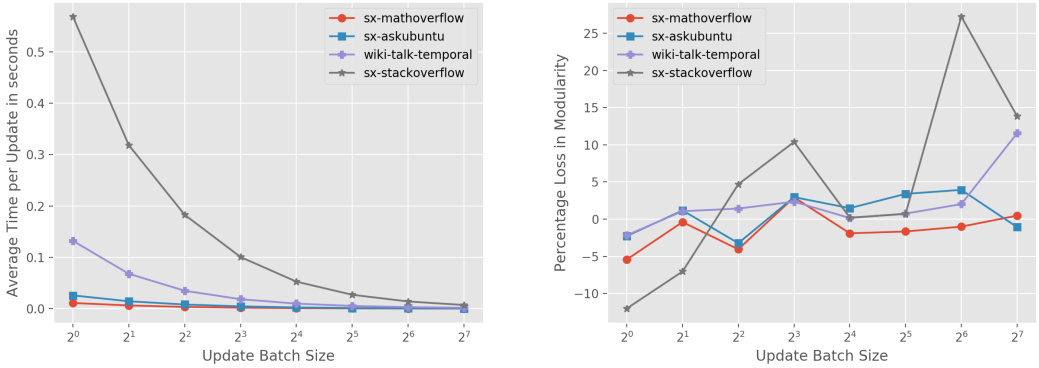


Figure 6.4: Percent loss in modularity for a varying number of edge additions and deletions. Modularity values are better when the number of updates is small.

In Figure 6.4, we have plotted the percentage loss in modularity by varying the fraction of edge additions and deletions. The percentage loss in modularity is given by $Loss_{\%} = (M_{static} - M_{dynamic})/M_{static} * 100$ where $M_{dynamic}$ is the modularity obtained after the dynamic updates and M_{static} is the modularity value after running the static community detection algorithm. We can see that for a small number of updates the dynamic modularity values are almost similar to modularity values obtained from static community detection. When we increase the number of updates, the modularity values start to deviate by a large amount. This behavior is expected as our algorithm detects communities that are approximately correct and when we increase the number of updates the error values get accumulated and deviate by a large amount. Since the number of updates for which the data is plotted varies exponentially, the error value also increases exponentially. We can also notice that the change is modularity value is very

less for updates with more edge additions. Even on doing a large number of updates the error in modularity value for updates with only edge additions is very low.

6.2.2 Multi Edge Updates



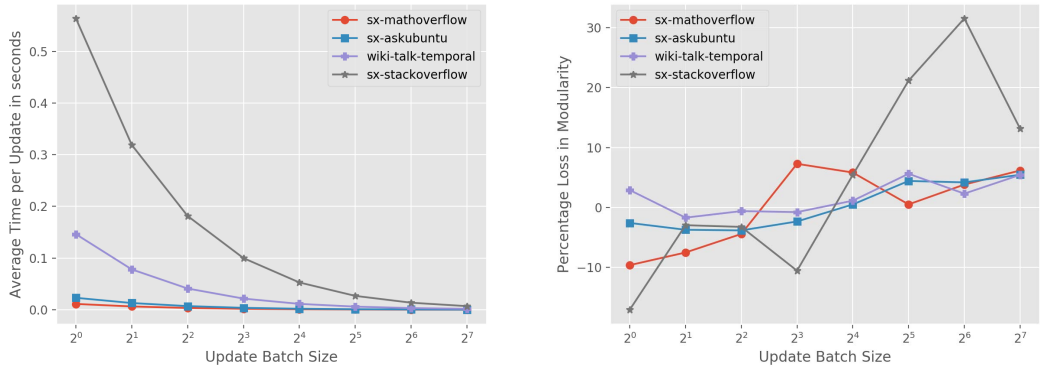
(a) Average Time per update vs Batch size (b) Percentage loss in modularity vs Batch size

Figure 6.5: For all the four graphs (Paranjape *et al.*, 2017) 4096 edges were added in batches of size specified in the x-axis

We took four temporal graph datasets - sx-mathoverflow, sx-askubuntu, wiki-talk-temporal, sx-stackoverflow from SNAP (Paranjape *et al.*, 2017). We excluded the last 4096 edges from the datasets and built a graph using the remaining edges. This graph was used as a base graph for the experiment. We added the remaining 4096 edges in batches of 2^n where $0 \leq n < 8$, to the base graph and used dynamic community detection algorithm to find the community structure of the graph after every batch update. The result is shown in Figure 6.5. It is evident from Figure 6.5a that the average time per update considerably reduces with increasing the batch size. In figure 6.5b, we can see that the error in modularity values are negative when batch size is 1, which means that the algorithm performs better than the static community detection algorithm in terms of modularity. The error percentage here for batch size of 1 is lower than the error percentage we obtained for single edge updates experiments in Section 6.2.1. This improvement in accuracy is due to the propagation we do in batched updates. There is a general trend of increase in accuracy with a decrease in batch size. This situation can be seen as a trade-off between accuracy and execution time.

We took the 4 temporal networks and created a base graph with all the edges from the dataset. We removed 4096 edges from the base graph in batches of size 2^n where

$0 \leq n < 8$ and used dynamic community detection algorithm to find the community structure of the graph after every batch update. The result is shown in Figure 6.6. The data we observe here follows the same trend we observed in Figure 6.5. Again, the improvement in accuracy is due to the propagation we do in batched updates.



(a) Average Time per update vs Batch size (b) Percentage loss in modularity vs Batch size

Figure 6.6: For all the four graphs (Paranjape *et al.*, 2017) 4096 edges were deleted in batches of size specified in the x-axis

CHAPTER 7

CONCLUSION

We have proposed a new Static Community Detection algorithm which is fast and accurate. Our algorithm performs better than the existing OpenMP code in terms of execution time. The main reason for our algorithm to perform faster is that we don't have an explicit aggregation phase. The limits to our algorithm are simply the storage capacity rather than the execution time. A graph with 3 million nodes and 234 million edges just takes about 3.55 seconds for execution.

We have also described the various cases that arise when we deal with a dynamic network and have provided algorithms to incrementally find the community structure of such networks. For single edge updates, our algorithm performs better when the number of updates is less. As single edge updates are approximate solutions, when we accumulate many single edge updates the error percentage increases. Updates with more edge additions deviate less from the static community detection algorithm's modularity values. In the case of multi-edge updates, when the updates are made in small batches it leads to a more accurate solution. The average running time per update is considerably reduced when we run with large batches of updates.

REFERENCES

1. **Aktunc, R.**, **I. H. Toroslu**, **M. Ozer**, and **H. Davulcu**, A dynamic modularity based community detection algorithm for large-scale networks: Dslm. *In Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM International Conference on*. IEEE, 2015.

2. **Blondel, V. D.**, **J.-L. Guillaume**, **R. Lambiotte**, and **E. Lefebvre** (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, **2008**(10), P10008.

3. **Brandes, U.**, **D. Delling**, **M. Gaertler**, **R. Gorke**, **M. Hoefer**, **Z. Nikoloski**, and **D. Wagner** (2008). On modularity clustering. *IEEE transactions on knowledge and data engineering*, **20**(2), 172–188.

4. **Cormen, T. H.**, *Introduction to algorithms*. MIT press, 2009.

5. **Fortunato, S.** (2010). Community detection in graphs. *Physics reports*, **486**(3-5), 75–174.

6. **Liu, Y.**, **K. P. Gummadi**, **B. Krishnamurthy**, and **A. Mislove**, Analyzing facebook privacy settings: user expectations vs. reality. *In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011.

7. **Lu, H.**, **M. Halappanavar**, **D. Chavarria-Miranda**, **A. Gebremedhin**, **A. Panyala**, and **A. Kalyanaraman** (2017). Algorithms for balanced colorings with applications in parallel computing. *IEEE Transactions on Parallel and Distributed Systems*, **28**(5), 1240–1256.

8. **Lu, H.**, **M. Halappanavar**, and **A. Kalyanaraman** (2015). Parallel heuristics for scalable community detection. *Parallel Computing*, **47**, 19–37.

9. **Malhotra, G.**, **H. Chappidi**, and **R. Nasre**, Fast Dynamic Graph Algorithms. *In Proceedings of the 30th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, UK, 2017.

10. **Meunier, D.**, **R. Lambiotte**, **A. Fornito**, **K. Ersche**, and **E. T. Bullmore** (2009). Hierarchical modularity in human brain functional networks. *Frontiers in neuroinformatics*, **3**, 37.

11. **Naim, M.**, **F. Manne**, **M. Halappanavar**, and **A. Tumeo**, Community detection on the gpu. *In Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017.

12. **Newman, M. E.** (2004). Analysis of weighted networks. *Physical review E*, **70**(5), 056131.

13. **Newman, M. E.** (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, **103**(23), 8577–8582.

14. **Newman, M. E.** and **M. Girvan** (2004). Finding and evaluating community structure in networks. *Physical review E*, **69**(2), 026113.

15. **Nguyen, N. P.**, **T. N. Dinh**, **Y. Shen**, and **M. T. Thai** (2014). Dynamic social community detection and its applications. *PloS one*, **9**(4), e91431.

16. **Paranjape, A.**, **A. R. Benson**, and **J. Leskovec**, Motifs in temporal networks. *In Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 2017.

17. **Shang, J.**, **L. Liu**, **F. Xie**, **Z. Chen**, **J. Miao**, **X. Fang**, and **C. Wu** (2014). A real-time detecting algorithm for tracking community structure of dynamic networks. *arXiv preprint arXiv:1407.2683*.

18. **Traud, A. L.**, **P. J. Mucha**, and **M. A. Porter** (2012). Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, **391**(16), 4165–4180.

19. **Wallace, M. L.**, **Y. Gingras**, and **R. Duhon** (2009). A new approach for detecting scientific specialties from raw cocitation networks. *Journal of the Association for Information Science and Technology*, **60**(2), 240–246.

20. **Waltman, L.** and **N. J. Van Eck** (2013). A smart local moving algorithm for large-scale modularity-based community detection. *The European Physical Journal B*, **86**(11), 471.

21. **Yang, J.** and **J. Leskovec** (2015). Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, **42**(1), 181–213.

22. **Zuo, X.-N.**, **R. Ehmke**, **M. Mennes**, **D. Imperati**, **F. X. Castellanos**, **O. Sporns**, and **M. P. Milham** (2011). Network centrality in the human functional connectome. *Cerebral cortex*, **22**(8), 1862–1875.