AGGLOMERATIVE CLUSTERING FOR COMMUNITY DETECTION IN DYNAMIC GRAPHS

A Thesis
Presented to
The Academic Faculty

by

Pushkar J. Godbole

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2016

AGGLOMERATIVE CLUSTERING FOR COMMUNITY DETECTION IN DYNAMIC GRAPHS

Approved by:

Professor David Bader, Advisor
Committee Chair,
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Jason Riedy
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Bistra Dilkina
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: 25 April 2016

To my grandfather, for showing me that the real-world is much more amazing than fantasy.

ACKNOWLEDGEMENTS

Switching my major from Aerospace Engineering to CSE for my Masters at Georgia Tech, made moving to a new country feel like a rather blanched change in comparison. While the last two years have been anything but cozy, I believe my marginal learning over this short span exceeds everything heretofore. It has truly given me an opportunity to remold myself into someone I aspired to be, at the onset. That indeed is the true purpose of education I believe.

A major part of this learning has come from this Masters Thesis, for which I am indebted to my advisor, Prof. David Bader for giving me the opportunity, guidance and assistance throughout my study and research. I would also like to express my deepest gratitude to Dr. Jason Riedy without who's encouragement, expert advice and generous support, I couldn't even have imagined working on a topic as formidable as Graph Theory, given my distinct background. I always secretly believed Graph Theory was a cool topic. I can now confirm that my belief was well-founded. Thank you to Prof. Bistra Dilkina for being a part of my Thesis Committee and for the valuable insights. And thanks to James and Anita for all their spontaneous assistance in my research.

Many thanks are also in order for my friends, for adding color to my time here at Georgia Tech. And finally, the acknowledgements would not be complete without thanking Reddit, Quora and the countless such time sinks, without which I would've probably been more productive, but life would've been much more drab, had I not known "The longterm effects of immortality on humans".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Agglomerative Clustering techniques work by recursively merging graph vertices into communities, to maximize a clustering quality metric. The metric of Modularity coined by Newman and Girvan, measures the cluster quality based on the premise that, a cluster has collections of vertices more strongly connected internally than would occur from random chance. Various fast and efficient algorithms for community detection based on modularity maximization have been developed for static graphs. However, since many (contemporary) networks are not static but rather evolve over time, the static approaches are rendered inappropriate for clustering of dynamic graphs. Modularity optimization in changing graphs is a relatively new field that entails the need to develop efficient algorithms for detection and maintenance of a community structure while minimizing the "Size of change" and computational effort. The objective of this work was to develop an efficient dynamic agglomerative clustering algorithm that attempts to maximize modularity while minimizing the "size of change" in the transitioning community structure.

First we briefly discuss the previous memoryless dynamic reagglomeration approach with localized vertex freeing and illustrate its performance and limitations. Then we describe the new backtracking algorithm followed by its performance results and observations. In experimental analysis of both typical and pathological cases, we evaluate and justify various backtracking and agglomeration strategies in context of the graph structure and incoming stream topologies. Evaluation of the algorithm on social network datasets, including Facebook (SNAP) and PGP Giant Component networks shows significantly improved performance over its conventional static counterpart in terms of execution time, *Modularity* and *Size of Change*.

CHAPTER I

INTRODUCTION

Community detection and partitioning is one of the most critical aspects of graph analysis, having primal applications in a wide variety of fields ranging from molecular biology to social networks. However, due to the NP-hard nature of the problem and arbitrary nature of the objective, deterministic identification of the optimal community decomposition is practically impossible, at least for large (real-world) datasets. Hence many approximation algorithms and approaches have been developed to achieve near optimal clustering and partitioning. The non-deterministic nature of these algorithms brings about the need of a standardized metric to assess the "goodness" of a decomposition, since the optimal is unknown. To that end, various metrics such as mutuality, reachability, betweenness have been developed, modularity being the most widely accepted one, to evaluate the quality of graph partitioning. These metrics, although very efficient, greatly depend upon the structure and size of the graph, thus making the clustering of dynamic graphs significantly challenging.

In this work, we build upon our previous work on dynamic community detection and discuss the design and implementation of a new algorithm drawing from the limitations of the previous approach. The objective of the re-agglomeration algorithm is maximizing/maintaining the modularity and smoothness of transition from an old community structure to the new, in progressively transforming graphs. We use the metric of "Size of Change" (defined later) to quantify the smoothness of transition and exhibit the strengths and weaknesses of the static and dynamic versions of the algorithm for various graph transformations. The next chapter discusses the past

work related to dynamic community detection, specifically in context of *Modularity* maximization. Chapter III introduces the metrics, modularity and size of change, with their various properties critical for dynamic graph clustering. Chapter IV briefly illustrates the previous algorithm and discusses its merits and pitfalls in context of the observed results. Finally Chapter V describes the backtracking algorithm, implementation and results. Chapter VI concludes with sound recommendations and direction of future work.

CHAPTER II

PREVIOUS WORK

Traditionally, the problem of static graph clustering has been widely studied, in the context of modularity optimization and beyond. The methods used for the same can be broadly classified into four categories:

- *Graph Partitioning*: This method works by dividing the vertices of the graph into a predefined number of groups of predefined sizes by placing cuts in the graph that minimizes the cut-size (number of edges encountered by the cut). Although still frequently used, the primary disadvantage of this method is the necessity of pre-specifying the number of communities and the prescribed community size. These requirements greatly limit the application of this method to various real graphs, particularly the evolving ones, rendering it highly unsuitable for dynamic clustering

- *Partitional clustering*: Similar to graph partitioning, this method requires pre-specifying the number of communities for clustering and then partitions the graph based on well-known techniques such as k-means clustering. This method suffers from the same limitation as the graph clustering due to fixed number of communities making it unsuitable for dynamic community detection

- *Spectral clustering*: This method builds on top of the partitional clustering technique and applies similar partitioning methods to the eigen-transformations of the graph. The eigen-vectors that better represent the community structure of the graph are used to represent the graph in the eigen-space and partitioned using standard techniques such as k-means clustering. This partitions when

translated back to the graph yield better clustering than the direct clustering, in some cases.

- *Divisive clustering*: These methods are localized in nature and work by either starting from individual vertices and merging them into communities that improves the community structure of partitioning or by starting from a large connected component and splitting it into components that yields a community structure. The re-agglomeration technique comes under this paradigm and proves suitable for dynamic graphs, due to the emergent nature of community detection and localized nature.

Additionally, many other heuristic methods such as Simulated Annealing, Genetic Algorithms have been designed and evaluated for graph partitioning. The reader is advised to refer to the Community Detection review by Fortunato [4] for the details on various paradigms of static graph partitioning.

On the other hand, community detection in dynamic graphs is a relatively untouched field. The first work in this field by Hopcroft et al. [8] tracks the evolution of a graph by running an agglomerative clustering on timely snapshots of the graph. This agglomeration however is memoryless and hence does not come under the class of dynamic clustering techniques. Gorke et al. [5] introduce a partial ILP based technique for dynamic graph clustering with low difference updates, however this method proves unsuitable for large changes over time, due to its high computational requirement. Another class of methods for clustering of evolving networks, such as label propagation [7], relies on local information and connectivity patterns instead of using global metrics. Our focus however is on agglomerative methods that optimize a numerical metric like *modularity*. The work closest to this study is a recent paper in the series of papers on this topic by Gorke et al. [6] that evaluates various techniques, particularly a variant of the hierarchal agglomerative clustering, while keeping track of the

change in community structure in order to maintain smoothness of transition between consecutive clusterings. This paper introduces five agglomeration based algorithms with both static and dynamic versions as follows. Modularity is used as a metric to measure the clustering quality while smoothness (R_g) is used to measure the degree of dissimilarity between $[0, 1]$ of clustering C' w.r.t. C, where:

$$R_g(C, C') = 1 - (|E_{11}| + |E_{00}|)/|E|$$

where, E is the set of all edges in C and

$$E_{11} = \{(u, v) \in E : C(u) = C(v) \wedge C'(u) = C'(v)\}$$

$$E_{00} = \{(u, v) \in E : C(u) \neq C(v) \wedge C'(u) \neq C'(v)\}$$

- *Greedy static global agglomeration*: Memorylessly performs greedy agglomeration starting from singletons, at every step of the graph change, in order of differential improvement in modularity, on merge of two vertices

- *Greedy static local agglomeration*: Similar to the previous method, but community vertices only consider their neighbors while determining the best merge

- *Greedy dynamic global re-agglomeration*: Uses the the current and previous clustering along with the history graphs (of communities) to re-agglomerate freed vertices based on a predefined policy (P). Recursively attempts merging communities starting from previous clustering in order of differential improvement in modularity

- *Greedy dynamic local re-agglomeration*: Similar to the previous method, but community vertices only consider their neighbors while determining the best merge

- *Hybrid greedy local agglomeration*: This method is similar to the Greedy static local agglomeration except that the objective function is a convex combination

of modularity and size of change. The algorithm fed with the previous clustering attempts maximizing this objective function by agglomerating neighboring vertices to increase the objective function, starting from singletons at every iteration of graph change.

Local Freeing Policies: The local dynamic re-agglomerative approaches employ three policies for freeing vertices affected by update in edge (u, v), at every iteration, based on the following criteria:

- u, v only
- u, v and their d-hop neighbors (where d is the depth of the hop)
- u, v and their first n neighbors using breadth-first

where u and v are the affected vertices in the graph change.

Global Back Tracking: The strategy used for the global re-agglomeration in the dynamic methods follows the following backtracking rules to free affected vertices:

- If Intra-cluster edge addition: Backtrack to the point where the vertices were separate
- If Inter-cluster edge addition: Backtrack to the point where the vertices become singletons
- If Intra-cluster edge deletion: Backtrack to the point where the vertices are singletons
- If Inter-cluster edge deletion: Do nothing

Amongst these methods, the paper reports that in the static case, the local search performs consistently better than the global search. While in the dynamic case, the global method with backtracking performs the best both in terms of speed and

modularity, although the smoothness decreases. The re-agglomeration method and the size of change metrics used in our study are an extension of the global backtracking approach used in this paper and have been described in section 5.

CHAPTER III

PRELIMINARIES

Graphs, particularly graphs representing real networks of objects, generally exhibit a community structure. In other words, some regions of such graphs are more closely connected than others. These closely knit groups in graph-analytical context are called communities, wherein the the members (vertices) within a community are more densely connected to other members within that community than they are to members outside. The metric of Modularity coined by Newman and Girvan [10], is a widely used measure of the strength of partitioning of a network into modules/communities. Networks with high modularity have dense connections between the vertices within modules but sparse connections between vertices in different modules. In case of dynamic graphs, as the graph changes (due to addition and removal of vertices and/or edges) the community structure and hence the modularity evidently changes apropos. Modifying the partitioning to maximize/maintain modularity entails creation and deletion of communities along with transitions of vertices between communities. Minimizing these transitions in an attempt to maximize modularity ensures a smooth transition from an old partitioning to new. The Size of Change metric as the name suggests, has been defined to quantify this change in partitioning w.r.t. the vertex transfers. To that end, this study aims to analyze the performance of the re-agglomeration algorithm w.r.t. the modularity and size of change.

3.1 Modularity

Mathematically, modularity is defined as the fraction of the edges that fall within the given modules minus the expected value of such fraction if edges of the graph were distributed at random. Amongst the most commonly used methods to calculate

modularity and the one used here, the randomization of edges is based on the criterion that the degree of each vertex is preserved in the canonical random graph. For a graph G(V, E), with n vertices and m edges, we define A_{ij} as the adjacency of vertices i and j, i.e. $A_{ij} = 1$ if an edge exists between vertices i and j in G and 0 otherwise. Similarly, Pij is defined as the expected number of edges between vertices i and j in the canonical random graph R_G. We define δ_{ij} as the community equivalence which is 1 if vertices i and j belong to the same community and 0 otherwise. Then, based on this definition, the modularity(Q) can be expressed as:

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij}) \delta_{ij}$$

The value of P_{ij} can be computed using our randomization model in which the degree of all vertices is kept intact and the connections changed. Graph G with m edges will have in all $2m$ stubs (half edges). For vertices i and j, their stubs can be connected to any of the remaining $2m - 2$ stubs. For large value of m, $2m - 2 \approx 2m$. Therefore, the probability of having an edge between i and j in R_G would be given by $k_i/2m \times k_j/2m = k_i k_j/4m^2$ making the expected number of edges, $P_{ij} = 2m \times k_i k_j/4m^2 = k_i k_j/2m$ (where k_i and k_j are the degrees of vertices i and j in G and hence in R_G). Thus the above equation for modularity can be simplified to:

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - \frac{k_i k_j}{2m}) \delta_{ij}$$

Modularity has certain properties that make it suitable for graph clustering [4, 3]:

- For an undirected and unweighted graph G, the modularity Q lies between the range $-1/2 \leqslant Q \leqslant 1$. A positive Q implies that, the number of edges within communities exceeds the number expected on the basis of R_G. The modularity of the entire graph as a single community is zero while that of all singletons (each vertex, its own community) is negative

- Isolated (degree 0) vertices have no impact on the modularity of a community structure

- In the clustering with maximum modularity, each cluster is a connected subgraph. i.e. placing disconnected subgraphs of G, if they exist, in different clusters yields maximum modularity
- Clustering is a non-local property: Changes to one region of the graph may propagate to the other regions of the community structure, yielding a completely different optimal clustering
- Modularity is non-scalable: The optimal clustering of a graph need not be retained if the graph is replicated (a copy of G added to G)

We direct the discerning reader to the paper by Brandes et al. [3] for proofs of these properties and additional corollaries.

3.2 Size of Change

The Size of Change metric needs to reflect the difference between two community structures based on the community associations of all vertices. However, since the communities do not have explicit tags, the measure for each vertex must be relative to the local change in terms of its neighborhood. This is done based on the following three measures of change for each vertex:

- Neighbors that were not in its community, but now are (Join)
- Neighbors that were in its community, but now are not (Leave)
- Neighbors that were in its community and still are (Stay)

Based on these three changes, we define two parameters to quantify change in the neighborhood of each vertex (v):

$$c_J(v) = \frac{J(v)}{S(v) + J(v)}$$

$$c_L(v) = \frac{L(v)}{S(v) + L(v)}$$

Where $c_J(v)$ and $c_L(v)$ are the joining and leaving parameters for v's neighborhood. $J(v)$ is the number of neighbors that newly joined v's community, $L(v)$ the number of neighbors that left v's community and $S(v)$ is the number of neighbors in v's community that stayed unchanged. Based on $c_J(v) > mean(c_J(v)) + 2stddev(c_J(v))$ or $c_L(v) > mean(c_L(v)) + 2stddev(c_L(v))$, we mark vertex v as changed. The total number of vertices that are marked as changed based on this criterion is finally defined as the Size of Change (SoC).

3.3 Differential Modularity

One of the primary advantages of using Modularity as a metric for agglomeration is that, its computation is localized, in that it is possible to compute the contribution to the overall modularity by each individual community. The expression for the same is given by:

$$\Delta M_C = \frac{E_C}{m} - \frac{Vol_C^2}{4m^2}$$

Where ΔM_C is the contribution to modularity by community C, E_C is the number of internal edges in community C and Vol_C is the volume or the sum of the degrees of all vertices internal to community C. m is the total number of edges in the original graph.

Having said that, the effect of merging two community vertices on the optimality of the subsequent community structure (w.r.t. modularity) is not localized. The reader is advised to refer the the paper on properties of modularity by Brandes et al. [3] for further details.

Most modularity based agglomeration schemes including this one, prioritize merges based on their differential contribution to overall modularity. However, the highest contribution to modularity does not in many cases imply an optimal community structure. Although modularity serves as a good metric to evaluate pre-existing community structure, its not always very well suited to evaluate the best possible

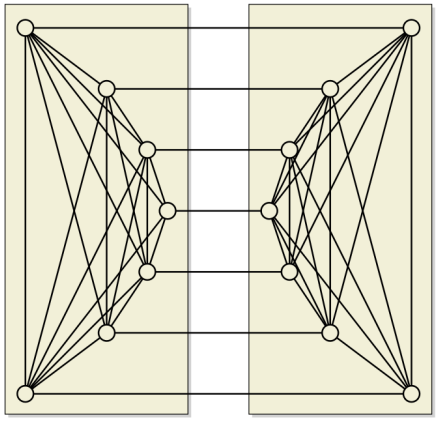

Figure 1: 14-regular two-clique graph [3]

merges. To illustrate this, we consider the following two examples:

- **Regular two-clique:** This is an example instance described by Brandes et al. [3] to illustrate the potential worst case instance of a modularity based agglomeration. A n-regular two-clique is a graph instance comprised of two n/2-cliques with each vertex v_i of clique 1 uniquely connected by an edge to a vertex u_i of clique 2 as shown in the figure below. As is quite obvious, the optimal community structure for such a graph is as shown in figure 1, giving the highest modularity of $1/2 - 2/n$. However, if merged based on the differential modularity, every merge of singletons along any edge gives an improvement in modularity of $2/n^2$. Consider a case of merge along all the bridge edges. This gives an overall increment in modularity from the base $-2/n$ to $-2/n + n \times 2/n^2 = 0$. Any further merge leads to an increment in modularity of 0 and hence would not be executed. Thus choosing the merges that yield the highest differential modularity does not always give the best community structure in this case.

- **Star Lattice:** This example illustrates the failure of modularity as a metric

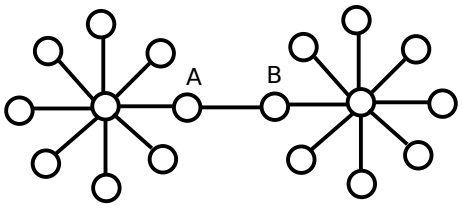

Figure 2: Two connected star graphs

to detect star structures in graphs. Detection of star like structures is of particular importance in community detection since many real-world community graphs such as social networks are majorly comprised of the same. A star graph is essentially a hub-spoke structure with many non-interconnected vertices connected by edges to a single central vertex. The following figure shows two such stars connected by an edge. For such a graph, merging a star center vertex and the corresponding bridge vertex gives an increment in modularity of $1/(2n+1) - 4n/4(2n+1)^2$ while merging along the bridge gives an improvement of $1/(2n+1) - 8/4(2n+1)^2$. Thus for all values of $n > 2$, the bridge merge gives the largest improvement. If we consider a lattice of such stars where multiple leaf vertices of stars are uniquely connected by an edge to the leaf vertices of other stars, or even a case of one-to-one edges between the two star leaves, the locally optimal merge always merges the bridge yielding a suboptimal overall community structure. This particularly happens due to the second term in the modularity expression, that favors merge of community vertices with the minimum degree as opposed to this case. This becomes particularly relevant, considering that many real-world graphs such as social networks follow a scale-free (power law) pattern of connectivity, making the appearance of sparsely connected star like structures common.

Thus, although approximately following the order of differential modularity leads to reasonable community structures with high overall modularity, as will be seen in the

performance of the algorithms described below; various merge ordering criteria need to be evaluated for their effectiveness and relevance in different scenarios.

3.4 Insertions vs Deletions

Aggomerative algorithms work by identifying affected vertices (based on a chosen policy) and freeing them for subsequent reagglomeration. However, there lies a fundamental difference between vertices affected by edge addition and deletion operations. In particular, edge additions are forward looking changes that affect the subsequent community structure. But edge deletions on the other hand are backward looking, in the sense that they preclude the existing community structure. Consider for example the case of a batch that deletes edges from a community splitting it into two disconnected subgraphs. A strict vertex freeing policy may free only vertices local to the affected edges, while other vertices (which must now belong to two separate communities) are still tagged to the same community. Therefore, this difference must be considered while defining the vertex freeing policies for the reagglomerative algorithms.

In the next two sections, we detail the working and performance of the two algorithms, the previous localized scheme and the new global backtracking scheme.

CHAPTER IV

MEMORYLESS RE-AGGLOMERATION

Here we briefly describe the previous reagglomeration approach with a localized vertex freeing policy, that is independent of the history of merges leading up the the current community structure (memoryless). Then we discuss its performance and shortcomings in context of the observed results.

4.1 Algorithm

The algorithm [11] starts by placing every input graph vertex within its own unique community. The algorithm maintains a community graph where every vertex represents a community, edges connect communities when they contain adjacent vertices from the input graph. The edge weights are equal to the total number of adjacent vertices between the two communities in the input graph. Nodes are weighed based on the total number of edges contained in the communities represented by those vertices.

In every iteration of graph change, the affected vertices are freed into singleton communities followed by the following three steps:

- For every edge, compute the differential change in modularity due to a merge along that edge
- The vertex pairs with the best differential improvement are chosen with a greedy maximal matching and merged in case of an improvement in modularity. The local maximization ensures that the algorithm doesn't compare every vertex pair. The maximal matching ensures that the merges happen along disconnected edges making them independent of each other

- Finally the community graph is contracted based on the choices of merges in the previous step and the connectivity updated accordingly

The algorithm exits when no further improvement is observed from any merge. The freed vertices are defined as the end vertices of all affected edges (additions/deletions). Presently, the algorithm is limited to edge additions and deletions only.

4.2 Performance

To evaluate the performance of the dynamic re-agglomeration algorithm, majorly three experiments were conducted with static agglomeration used as a baseline to compare the performance of the dynamic case. Unlike dynamic agglomeration, the static version was rerun for every incoming batch of changes.

Broadly, the following three experiments were run on these algorithms:

1. **RMAT → RMAT**: The graph was initialized to a RMAT graph with 2^{20} vertices with an average degree of 10. RMAT edge batches (of increasing sizes) were then added without deletions

2. **PGPGiantCompo → RMAT**: The initial graph was the PGPGiantCompo undirected graph [2] with 10680 vertices and 24316 edges. This graph was fed with edges generated using RMAT without deletions, in batches of increasing sizes

3. **PGPGiantCompo → Flipped PGPGiantCompo**: The initial graph was the PGPGiantCompo graph [2]. A flipped PGPGiantCompo graph was constructed by flipping the graph along its indices. i.e. vertex 1 was swapped with vertex n, vertex 2 with vertex n-1 and so on. This ensured that the community structure of the flipped graph would be exactly same as the original. This flipped graph was fed with edge replacement (deletions), in batches of increasing. Thus the initial PGPGiantCompo graph was transformed into the

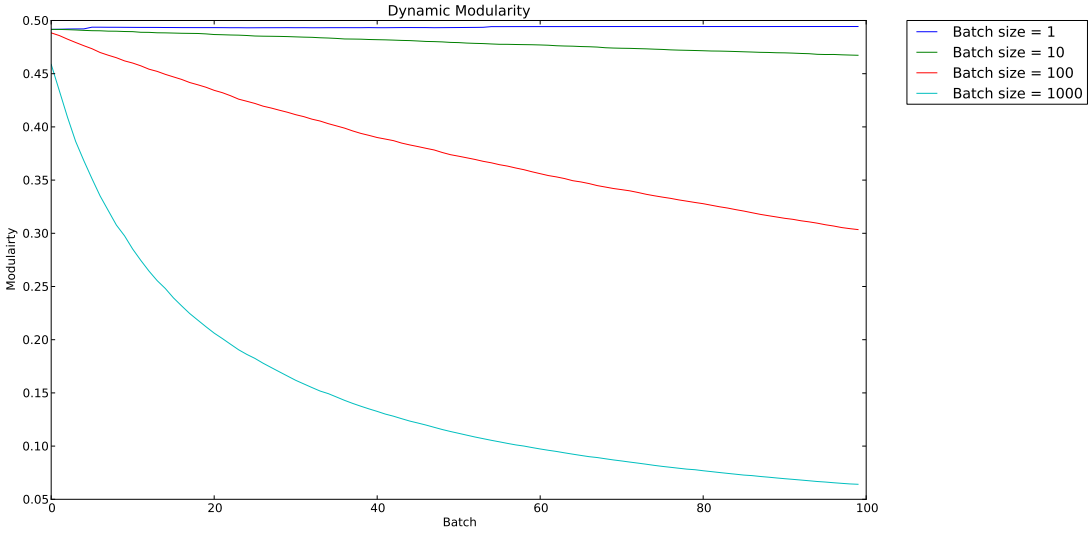

Figure 3: RMAT → RMAT: Dynamic Modularity change

flipped PGPGiantCompo graph with displaced communities but exactly same community structure.

Figures 3 and 4 illustrate the performance of the dynamic algorithm in case of RMAT → RMAT stream for increasing batch sizes: 1 - 10 - 100 - 1000. It can be observed from Figure 3 that the for higher batch sizes (> 1), the modularity of dynamic re-agglomeration keeps dropping as newer batches are added. This drop correlates very well with the drop in the number of communities in Figure 4 and a rise in the max community size, implying that the larger communities suck up the smaller communities and, constantly growing in size leading to the constant drop in the modularity. Figures 5 and 6 exhibit the performance of the algorithms as RMAT edges are added to the PGPGiantCompo graph for batch sizes of 100 and 300 illustrating the peculiar inflection in behavior as the batch size increases.

We observe in Figure 5 that like in the RMAT → RMAT case, the modularity of dynamic re-agglomeration keeps dropping with addition of new batches in case of a batch size of 100. On the other hand, Figure 6 shows that for the batch size of 300, the

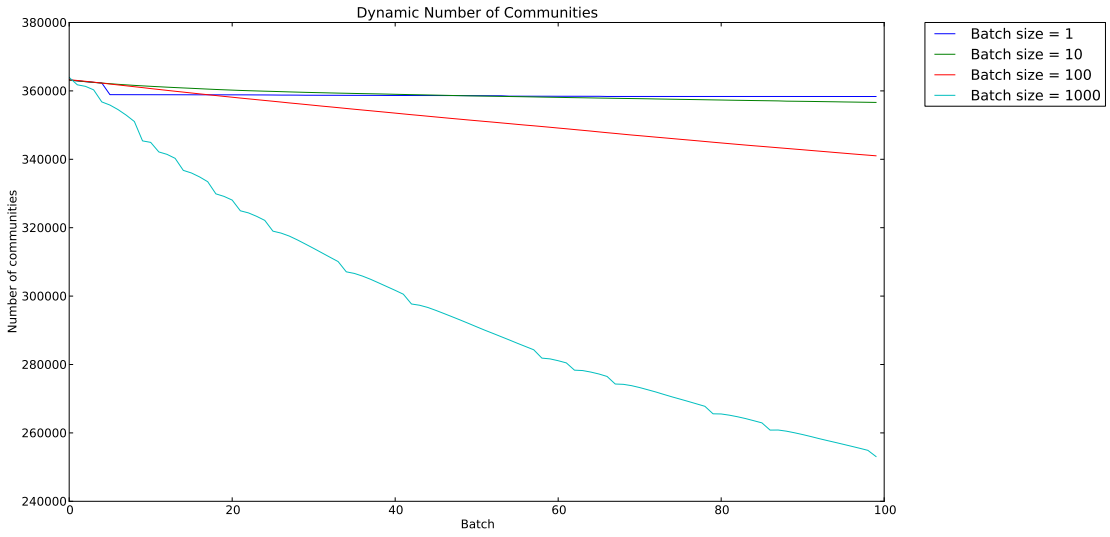

Figure 4: RMAT $\rightarrow$ RMAT: Dynamic Number of Communities change

modularity of the dynamic version drops until a point after which it increases. The point at which this inflection occurs is observed to shifts leftward (happens earlier) as the batch-size increases. The rate of increase also corresponds to the batch-size. Max community size overall increases for dynamic case while that for static remains almost constant at a low value. This inflection in the trend of modularity has a corresponding steep drop in the max community size.

As large batches of RMAT edges are added, the graph after a point transforms into majorly an RMAT graph. This is owing to the fact that the original PGP graph only has 10680 vertices and 48632 edges. Therefore, in case of a batch size of say 300, the graph has as many RMAT edges as PGPGiantCompo edges by the 150^{th} batch. Thus the graph progressively turns into an RMAT graph. As new batches are added, the static agglomeration bursts the vertices into singletons merging them together only if an improvement in *the current* modularity is entailed. Thus every decision in the static agglomeration is optimal (although the converse is not true). In the dynamic case, addition of new batches renders some previous agglomerations

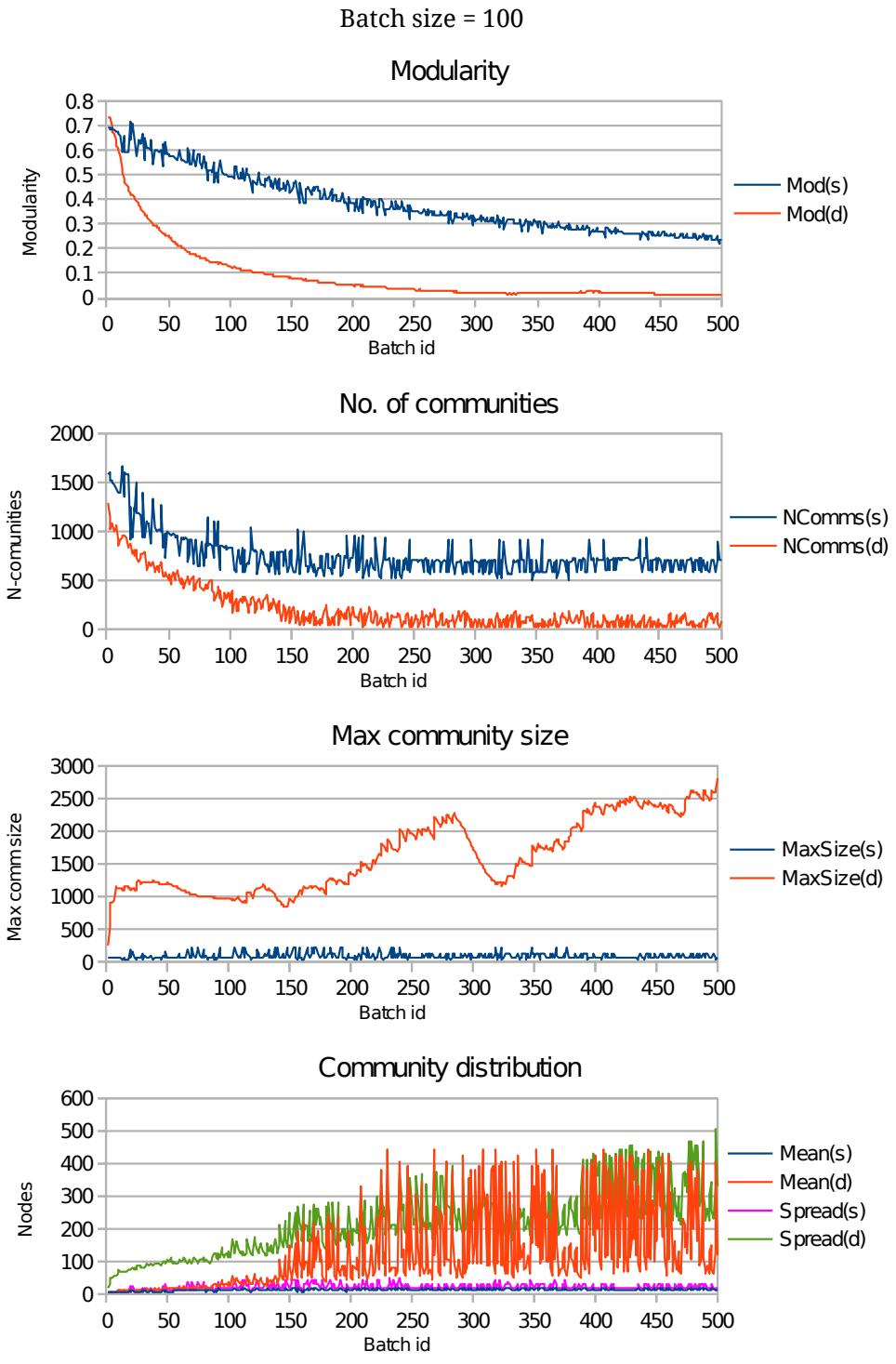

Figure 5: PGPGiantCompo → RMAT: Static vs Dynamic re-agglomeration performance (batch size =100)

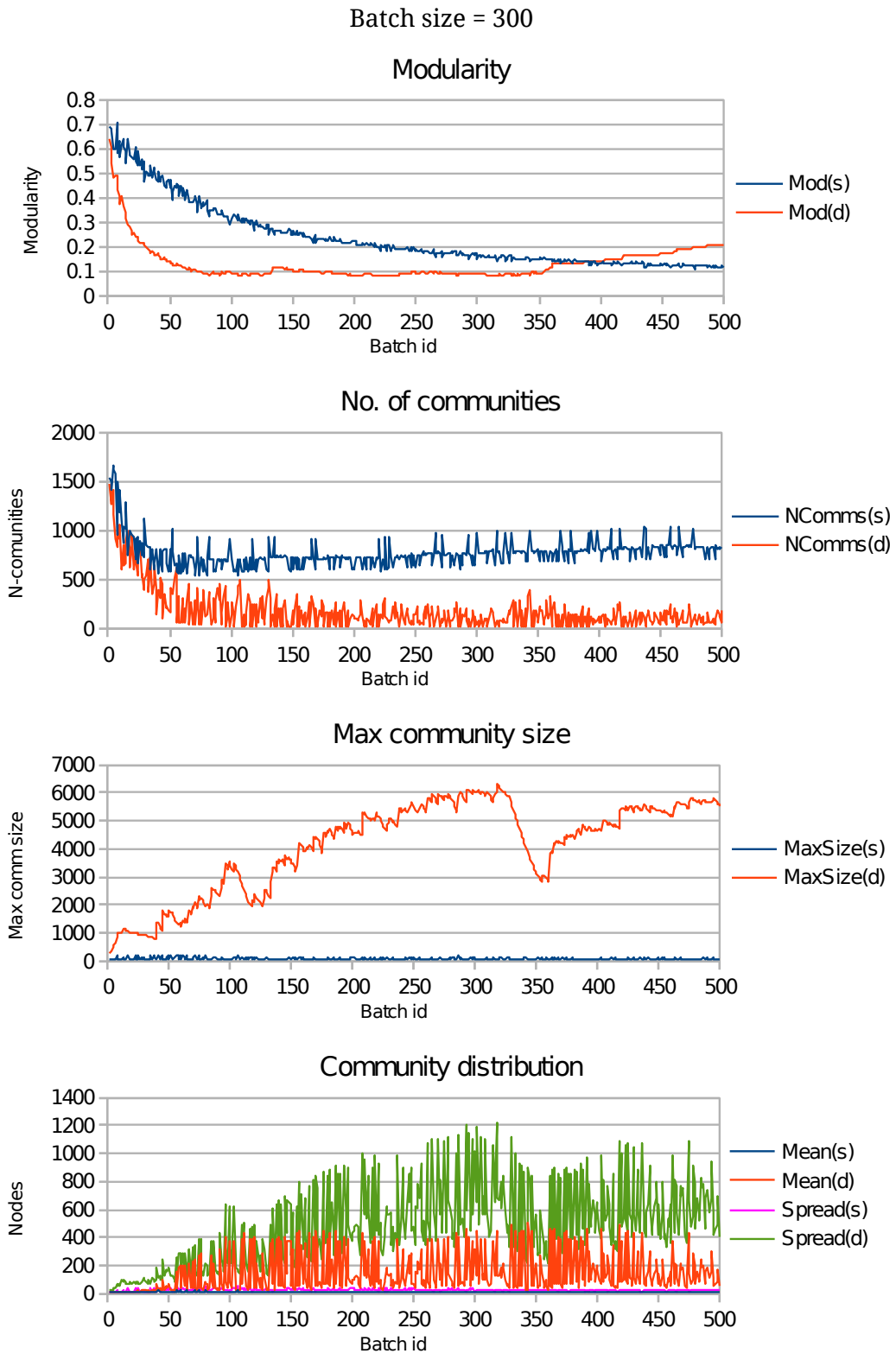

Figure 6: PGPGiantCompo → RMAT: Static vs Dynamic re-agglomeration performance (batch size =300)

sub-optimal (negatively affecting the modularity) however, since not all vertices are freed to singletons in this case, the suboptimal decisions stay leading to a drop in the overall modularity.

The inflective rise in modularity for the dynamic case beyond a point may be attributed to the fact that, addition of larger batch sizes frees up more vertices from the small set of 10680 vertices of the PGP graph, thus making room for a restructuring of the community distribution. This inference can also be verified from the observation that, the rise in modularity is observed to always have a drop in the max-batch size preceding it, thus implying that majority of vertices in the incoming batch belonged to the max-size community. Since smaller batch sizes do not free enough vertices, they fail to achieve this improvement.

The community size mean and spread increase for dynamic re-agglomeration with addition of new batches. The rate of rise is proportional to the batch size. On the other hand, both mean and spread for the static case remain almost constant at a low value all throughout. This implies that, since the static agglomeration begins from scratch at every step, it gets trapped in local optima early on and does not execute further merges due to absence of downhill moves in the vicinity. Based on the previous observation that, freeing more vertices in the later parts of the dynamic re-agglomeration can lead to restructuring the community distribution into a better modular one, the depth = 1 (affected vertex + neighbors) based dynamic case has been evaluated for the PGP to Flipped PGP transition. The fact that flipped PGP is only a translation of the original graph with the exact same community structure helps in evaluating the algorithms for responsiveness to the transition in community structure. Figures 7 and 8 illustrate the performance of the algorithm on batch sizes of 100 and 500, up till complete transformation. It can be observed from figures 7 and 8, that in case of the dynamic re-agglomeration with depth = 1, the modularity is able to catch up with the static case by the end. The size of change for this

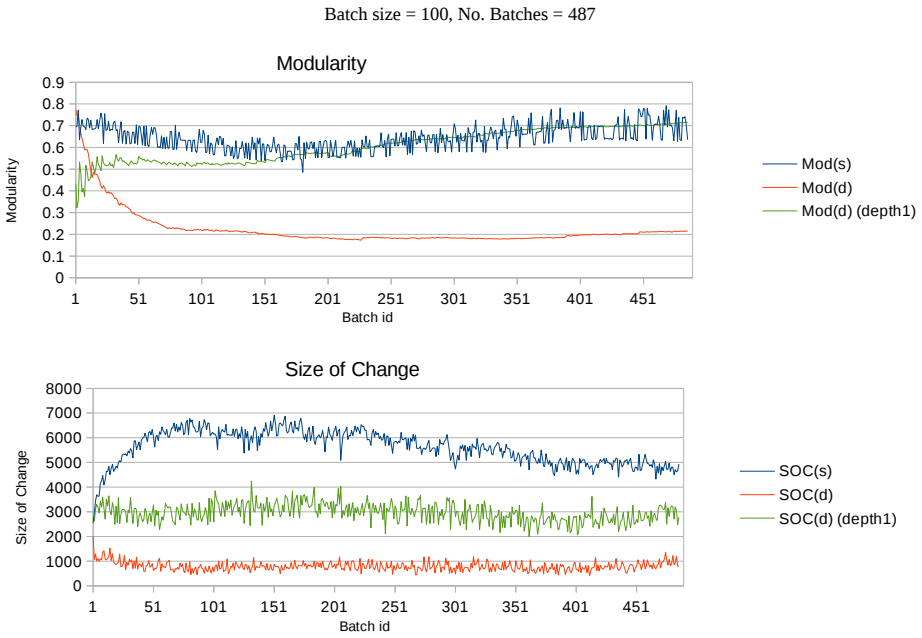

Figure 7: PGPGiantCompo → Flipped PGPGiantCompo: Static/Dynamic/Dynamic (depth = 1) re-agglomeration performance (batch size =100)

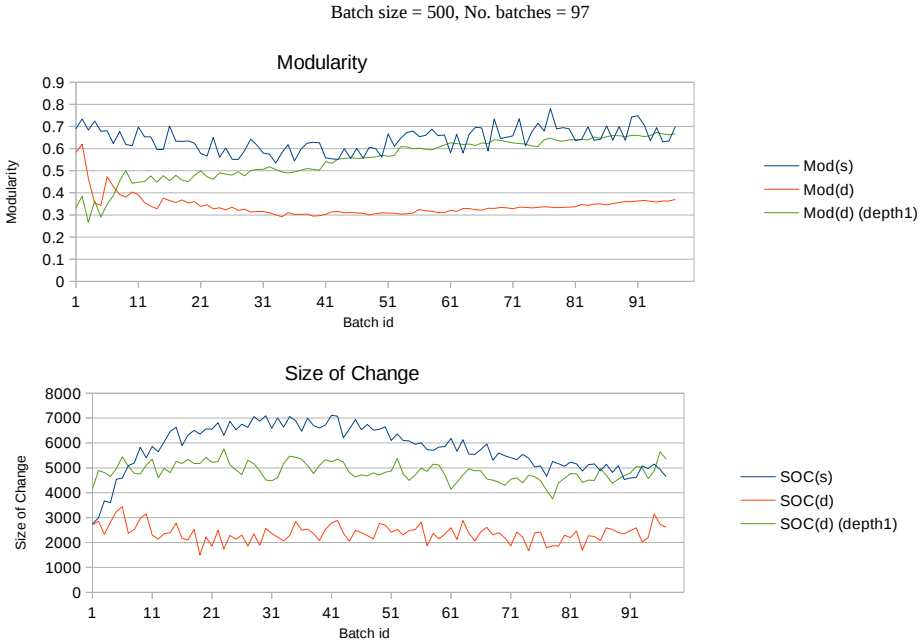

Figure 8: PGPGiantCompo → Flipped PGPGiantCompo: Static/Dynamic/Dynamic (depth = 1) re-agglomeration performance (batch size =500)

approach is seen to lie between the purely static and dynamic versions as expected. This experiment illustrates how slight modifications to the design of the algorithm can lead to significant improvements in the performance over time. Particularly, both the static and dynamic with depth = 1 methods exhibit a convex behavior achieving almost equal modularity at the end of the transition. The size of change in case of the dynamic depth = 1 approach lies between the purely static and dynamic cases exhibiting a sweet-spot in the trade-off between the two cases.

CHAPTER V

BACKTRACKING RE-AGGLOMERATION

Broadly, the algorithm follows an agglomerative clustering approach to identify communities that maximize modularity, using backtracking to handle the stream of edge changes (insertions/removals). It extends the backtracking approach followed by Gorke et al.[6] in their `dGlobal` approach to incorporate localized modifications to the community graph without splitting the parent communities.

Since single vertices are also treated as singleton communities in the agglomerative clustering approach, the terms *vertex* and *community vertex* are interchangeably used in the subsequent description.

5.1 Algorithm

Similar to static agglomeration, the algorithm starts off by agglomerating vertices along edges that improve modularity. But in addition to the present community structure, it also maintains a dendogram of the historical merges of all community vertices in time. For every incoming batch of edge changes, the algorithm splits the relevant communities either partially or completely based on the chosen *backtracking strategy* described later, thus pruning/modifying the dendogram in the process. This is followed by the merge of vertices along community edges improving modularity. The order in which the community edges are considered for merge is based on the *merge strategy*, discussed in the subsequent section.

More specifically, the algorithm can be broadly divided in three steps:

1. **Add Batch:** For every incoming batch of changes, separate edge additions and deletions into two groups

2. **Backtrack:** For every edge deletion (u, v), backtrack based on the chosen strategy. After all deletions are handled, similarly apply the backtracking strategy for each newly added edge. The partial community graph rendered after this step serves as a baseline for the subsequent merge step, thus eliminating the need to start the agglomeration from scratch, effectively maintaining the clustering information in the partial community structure, whilst simultaneously reducing the computation cost per iteration compared to static agglomeration.

3. **Merge:** Finally, merge the community vertices along edges that yield a positive increment in modularity, ordering the merges based on the chosen merge strategy.

5.1.1 Backtracking

The backtracking strategy defines how a batch of incoming changes is handled in the community dendogram of the graph. Broadly, any backtracking strategy is a combination of two methods: `split` and `edit_edge`.

5.1.1.1 Backtracking Methods

- **Split:** Splits a community into its two components by reversing the latest merge.

- **Edit Edge:** Keeps the community structure intact but recursively edits the underlying adjacency of the community vertices to change the weights of the community edges. Editing the underlying adjacency requires the knowledge of the chronological order in which the corresponding merges occurred. Figure 9 illustrates an example of the adjacency induced by the existence of an edge DE in the base graph, on the subsequent community graph with the parent stacks for vertices D and E as shown in Table 1. The solid lines in the above dendogram represent the merges of community vertices as time progresses along the y-axis and the dashed lines represent the edges induced by the base edge

Table 1: Parent stack of base-edge DE

D	E
C	
B	
	G
	B
A	A

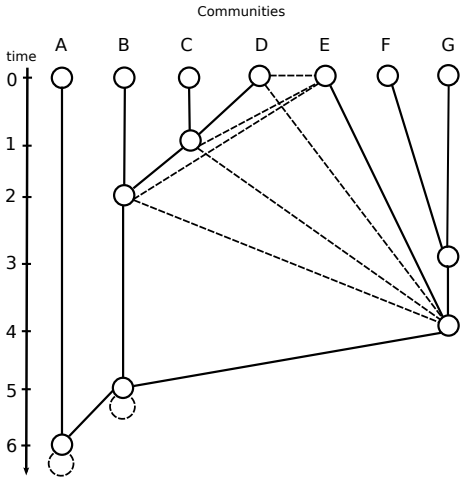

Figure 9: Dendogram of Induced Adjacency by base-edge DE

on the community graph. The merge of vertices D and C induces an edge between vertices E and C of the community graph at $t = 1$ and so on. Thus the timestamps associated with these merges help in determining the order of recursively editing the induced edges. In the above figure, deleting the edge DE in the base graph implies decrementing the weights of the induced edges AA, BB, GB, GC, GD, BE, CE, DE by the weight of the deleted base-edge DE in that order. The above list of affected induced edges can be generated from the parent stack of the base-vertices using the following three rules. In order to find the induced edges due to base edge ab. Assume the parent stacks of a and b are of size N and M respectively. Thus $n = N$ and $m = M$ become the initial current vertex indices for stacks a and b. With $p_i(x)$ denoting the i^{th} vertex

from the top, in the parent stack of vertex x and $t(p_i(x))$ the corresponding time stamp:

1. Assume without loss of generality, $t(p_n(a)) \geqslant t(p_m(b))$, making $p_n(a)$ the source vertex s, for the current iteration
2. Mark edges between $p_n(a)$ and every vertex below $p_m(b)$ in the parent stack of b, including $p_m(b)$. Stop when for some $i \in [m, 1]$, $p_n(a) = p_i(b)$ or $i = 1$
3. Update the current vertices for both stacks and chose the one with higher time stamp as the source vertex for the next iteration. Since $s = p_n(a)$ here, $m = m - 1$ if $t(p_n(a)) = t(p_m(b))$ and $m = m$ otherwise; $n = n - 1$. Then, $s = argmax(t(p_n(a)), t(p_m(b)))$

Repeat these steps for every new source vertex, until the bottom of both stacks is reached.

5.1.1.2 Backtracking Strategies

The handling of edge additions and deletions depends on the strategy chosen for the agglomeration. There are three strategies that may be chosen from, based on four type of the edge change described below. Irrespective of the chosen strategy, the `edit_edge` method needs to be applied to every edge change either by recursively inducing edges in the community graphs or by only adding an edge between two singleton communities, as a special case. Therefore the choice and extent of application of the `split` method distinguishes the strategies.

1. **Split to Singletons:** Recursively `split` the parent communities of both the involved vertices until they are singleton communities. Then add the edge to the new community graph between the newly formed singletons using `edit_edge`.

   This however is a conservative approach and could prove useful in pathological

cases where the added bridge edges largely exceed the edges in the existing communities, making the group of bridge vertices communities in themselves.

2. **Split to Separation:** Recursively `split` the parent communities until the corresponding vertices lie in separate communities. Then call `edit_edge` to recursively modify the induced edge weights of the updated community graph.

3. **Edit Edge:** Keep the existing community structure intact and call `edit_edge` to modify the induced edge weights corresponding to the incoming edge update.

5.1.1.3 Backtracking Policies

The four edge change types and the relevant backtrack strategies to be considered to handle them are described below.

1. **Inter-community edge addition:** An Inter-community edge addition strengthens the bridge edges or in other words obsoletes the present clustering. A large addition of edges between two communities can lead a maximum modularity community structure where:

   - The two communities merge into a single entity
   - The bridge vertices form a strongly connected community and the internal vertices of the component communities drift out into neighboring communities or form smaller independent communities

   The first case can be tackled by simply using the *Edit Edge* strategy where the edge addition acts as a forward looking change towards a community merge, without obviating the present clustering. The second case would need the use of the *Split to Singletons* strategy, giving the algorithm an opportunity to reconsider the present clustering. The cues for choosing one strategy over another and/or transitioning between the two need to be investigated further.

2. **Intra-community edge addition:** An Intra-community edge addition strengthens the existing community. A large addition of edges to a community can yield a maximum modularity community structure where:

   - The community becomes more closely connected thus increasing the overall modularity, if the edge additions are approximately uniform throughout the community
   - A region of the community becomes more closely connected than the rest forming a strong community in itself while the remainder of the vertices drift away into neighboring or self contained communities like in the previous case.

   Similar to the Inter-community edge addition, the first case can be tackled by the *Edit Edge* strategy while the second case would need the use of *Split to Singletons* to enable restructuring of the entire community.

3. **Inter-community edge deletion:** An Inter-community edge deletion strengthens the presently defined community structure irrespective of the number and topology of such deletions. Thus this case can be easily handled by only using the *Edit Edge* strategy.

4. **Intra-community edge deletion:** An Intra-community edge deletion weakens the community. Since the presence of the now deleted edge led to the merge of the corresponding vertices into a single community at some point in the past, it necessitates the re-computation of the clustering without the influence of this edge. The *Split to Separation* strategy would take the dendogram back to the point in history when the two vertices were in separate communities and merging them into a single community prior to this point was sub-optimal.

The backtracking approach facilitates the maintenance of clustering information from the previous agglomeration, leading to consistent growth of communities thus preventing the trapping in local optima, as has been observed in case of static agglomeration.

5.1.2 Merging

The order in which the edges are considered for merge plays a crucial role in the final outcome of the algorithm. Conventionally in agglomerative clustering approaches, the merges are made in a descending order of differential modularity improvement. However, a merge along an edge affects the neighborhood structure of the involved vertices, thus changing the magnitude of differential modularity and hence the ordering of edges left in the wake of the merge. Secondly, as seen in section 3.4, ordering edge merges based on their differential contribution to modularity also does not guarantee the best (maximum modularity) community structure. To that end, three approaches have been attempted here to balance performance and efficiency of the agglomeration. It must be noted here that all edges considered in context of *merging* are inter-community edges between presently active communities. Edges with one or both of the end vertices inside currently active communities are not visible to the *merge* operation.

5.1.2.1 Best Merge First

This is a brute-force approach in which the merges occur in descending order of the current best differential modularity. However, since a merge modifies the neighborhood of the involved vertices, the edge ordering needs to be updated after every merge. This can be naively done by scanning all edges of the graph to choose the next best merge at every iteration. Alternatively, since every merge only locally affects the neighborhood of the involved vertices, merges can be done by maintaining an updatable priority queue of edges, which updates the priorities of affected edges

after each merge. This queue however would have to be reconstructed after every incoming batch of graph changes and the computational overhead of maintaining such a priority queue needs to be investigated.

5.1.2.2 Node Spanning

This is the easiest approach to implement, in which merges are ordered based on vertices. Since the graph changes considered here are limited to edge additions and deletions, the vertices in the graph remain invariant throughout the process.

1. Starting from a vertex i, the algorithm merges the vertex with a neighbor j, that yields a positive change in modularity.

2. In order to maintain consistent community tagging, the merged community vertex is either tagged i or j based on the sizes of i and j, in terms of the number of internal vertices.

3. This continues until i becomes a sub-community of a neighboring vertex i'. The process is then repeated for vertex i' until it becomes a sub-community of one of its neighbors, and so on.

4. The agglomeration stops when there are no more edges, merges along which would yield a positive increment in modularity.

This algorithm is termed *Node Spanning* since it spans the entire graph starting from a vertex, agglomerating and expanding along the frontier of the neighboring vertices. This approach discards the magnitude information regarding the differential modularity and treats all positive differential modularity merges equally, ensuring localized merges and eliminating the need to recompute differential modularities over all affected edges after a merge.

Multi-vertex spanning: An extension of this approach would be to span the graph,

starting simultaneously from multiple vertices. A special case of this extension would be the all-vertex spanning approach, wherein all vertices seek out a neighbor to merge with. For a valid merge of two vertices to occur, they would need to seek out each other. The differential modularity information may be used in this case to allow each vertex to choose the mutually best merge in its neighborhood, thus maximizing the overall improvement in modularity in each step.

5.1.2.3 Matching

Similar to all-vertex spanning, the matching approach finds a set of disconnected edges, merging along which leads to an overall improvement in modularity, contracting the graph at each agglomerative step. For each agglomerative step:

1. Starting from the existing community graph, the algorithm identifies a maximum weight matching of the edges based on differential modularity.

2. The vertices are then merged along the set of matching edges, yielding an updated community graph.

These two steps are repeated until no more edges with a positive differential contribution to modularity remain.

This algorithm can be considered a static version of the *best merge first* approach in which, the change in values of differential modularities after each merge is not considered for other merges within an agglomerative step. Note however that, choosing a disconnected set of edges ensures that any merge does not invalidate the subsequent merges chosen from that matching set.

Cyclically repeating the *Add batch, Backtrack* and *Merge* steps for every incoming batch of edge additions and deletions ensures the maintenance of a good quality (high modularity) dynamic community structure, as the graph evolves.

5.2 Implementation

The algorithm is implemented in C++, on top of the Stinger Graph framework [1]. It however uses native data members and methods to store and update the clustering information.

5.2.1 Attributes

The historical information of the community structure is stored using five attributes described below:

1. **`parents:`** The parenthood history of all vertices is stored as a vector of stacks of merges, where each merge contains the id of the parent and timestamp of the merge. The vector contains one stack corresponding to each vertex and each stack represents the merge history of that vertex in chronological order. Initially, each vertex is its own parent. The parent stack of a vertex grows when it merges into another vertex.

2. **`children:`** The internal vertex information of the community structure is stored as a tree, using a vector of stacks of vertex ids. Each stack in the vector corresponds to a community vertex. Each vertex points to its immediate children in the dendogram, stored in the stack in order of the merges, which in turn point to their own children, thus forming a forest of trees, where each tree represents an active community. Recursing over the subtree of a vertex yields all vertices internal to that community. The use of stacks in this case is justified, since the potential splitting of a community vertex must follow the reverse order of the corresponding merges into the community.

3. **`size:`** Size is a vector that holds the counters for current number of internal vertices in each community. It starts off with each element corresponding to each vertex having a value of 1. When a vertex merges with another, the size

of the parent vertex is incremented by the size of the child vertex.

4. **`active:`** This is a boolean vector that stores the current active status of each community vertex. It starts off with each vertex marked active and marks a vertex inactive if/when it merges into another vertex.

The combination of `Parents` and `Children` stores the complete information of the hierarchical dendogram. Additional information however is needed to store the connections between these vertices as induced by the underlying graph.

5. **`neighbors:`** The neighborhood information of the community structure is stored as a vector of maps of community edges. Each map represents the neighborhood of a vertex with a mapping from the neighbor's id to the edge weight. Use of maps to store the neighborhood information ensures that, inserting, updating and deleting edges can occur in constant time. The neighborhood map of a vertex also includes self-edges, representing the total internal edges in the community. Decrementing an edge weight to 0 automatically deletes the edge from the neighborhood map. Also, it may be noted that the neighborhood map is symmetric i.e. `neighbors[u][v] = neighbors[v][u]`.

5.2.2 Methods

The methods used to execute the clustering algorithm are described below in a top-down order.

1. **`add_batch(batch):`** This function separates the incoming `batch` into lists of edge deletions and additions. Starting with deletions, for each edge change, it iteratively calls the `split` method until the chosen backtracking policy criteria are satisfied. Then it calls the `edit_edge` method to insert this edge into the community dendogram. At the end of this function call, all incoming edge changes have been incorporated into the community graph. The partially

merged community structure serve as the starting point for the subsequent merge operations.

2. **`split(u):`** The `split` method is an *undo* operation on community vertex `u`, that reverts the latest merge `v` $\rightarrow$ `u`. It removes the latest child vertex `v` from the stack `children[u]`, detaching `v`'s subtree from `u` and calls `revert_parent(v)` to recursively revert the `parent` stacks of all vertices in the `children[v]` subtree by 1. Then it subtracts `neighbors[v]` from `neighbors[u]`, separately handling the two special cases:

   - **Internal edge:**

     ```
     neighbors[u][u] -= neighbors[v][v]
     ```

   - **Connecting edge:**

     ```
     neighbors[u][u] -= neighbors[u][v]
     ```

   Finally, the community `v` is again marked as `active` and `size[v]` is subtracted from `size[u]`, to correct the community size of `u`, post the split. Maintaining `neighbors[v]` unmodified even after merging with `u` enables this consistent backtracking of neighborhood.

3. **`edit_edge(u, v, w):`** After the community structure around the involved vertices `u` and `v` has been modified, by splitting the community graph based on the chosen backtracking policy, the `edit_edge` method executes the *edit edge* algorithm, described in Section 5.1.1.1 to update edge weights between the affected vertices in parent stacks of `u` and `v` by `w`.

4. **`agglomerate():`** At the end of `add_batch`, the community graph stands in an intermediate state of partially merged communities. Then the `agglomerate` method merges the community vertices along edges that yield an increment in

modularity by iteratively calling `merge`. The ordering of the merges is defined by the chosen *merge strategy*.

5. **`merge(u, v):`** The `merge` method is the antipode of the `split` method. Calling `merge(u, v)` merges `v` $\rightarrow$ `u` if `size[u]` $\geqslant$ `size[v]` and `u` $\rightarrow$ `v` otherwise. If `v` $\rightarrow$ `u`, it inserts vertex `v` into `children[u]`, attaching `v`'s subtree to `u`, through `v`. Then it calls `change_parent(v, u)` to recursively insert `u` into the `parent` stacks of all vertices in `v`'s subtree. Then it adds `neighbors[v]` to `neighbors[u]`, separately handling the two special cases:

   - **Internal edge:**

     ```
     neighbors[u][u] += neighbors[v][v]
     ```

   - **Connecting edge:**

     ```
     neighbors[u][u] += neighbors[u][v]
     ```

   Finally, the community `v` is marked as `inactive` and `size[u]` is incremented by `size[v]`.

The static agglomeration can be realized as a special case of the dynamic agglomeration where the entire community structure is re-initialized with singleton communities after every new batch insertion.

5.3 Performance

There are multiple variables at multiple levels of the algorithm that can be adjusted to vary the performance of the backtracking agglomeration. These can mainly be divided into three categories top-down:

1. **Incoming Batch Topology:**

   Localized/Distributed

2. **Backtrack Strategy:** Choice and extent of backtracking for each type of change

3. **Merge Strategy:** Ordering of merges for a given incoming batch

In order to simulate different types of *incoming batch topologies*, two approaches have been employed. The localized batches (in which the edge changes are concentrated in a particular region of the graph are simulated by inserting/deleting edges in sorted order of vertex ids thus ensuring that for a reasonable batch size ($>$ average vertex degree), the entire neighborhood of one or more vertices will be altered. The distributed batches are inserted by simply randomizing the incoming edge order thus ensuring that a batch topology will generally be spread out across the graph. Although controlling the topology of incoming changes is beyond the scope of the algorithm, as such the distributed change topologies may be considered to be closer to the pathological cases commonly expected to be observed in realistic scenarios. The localized changes would be rare if not absent in a realistic setting. As will be observed in the following results, the distributed cases yield more predictable outcomes, generally leading to better performance of the algorithm.

In the present study, the *backtracking strategy* has been kept fixed. Following the elementary logic that in absence of drastic hyper-local changes, in general, edge deletions can be considered backward-looking changes while edge additions can be assumed to be forward looking, the backtracking strategy for all edge deletions is set to *Split to Separation* while all edge additions follow *Edit Edge*.

For the *merge strategy*, all three approaches - Best merge first, Node spanning, Matching have been evaluated. The Best merge first strategy took excessive runtime due to its exhaustive nature, while not yielding significant improvement over the Matching approach in case of the PGP graph. Hence, the Node Spanning and Matching strategies have been extensively evaluated.

Broadly, four experiments have been performed to evaluate the performance of the backtracking algorithm based on the above variations:

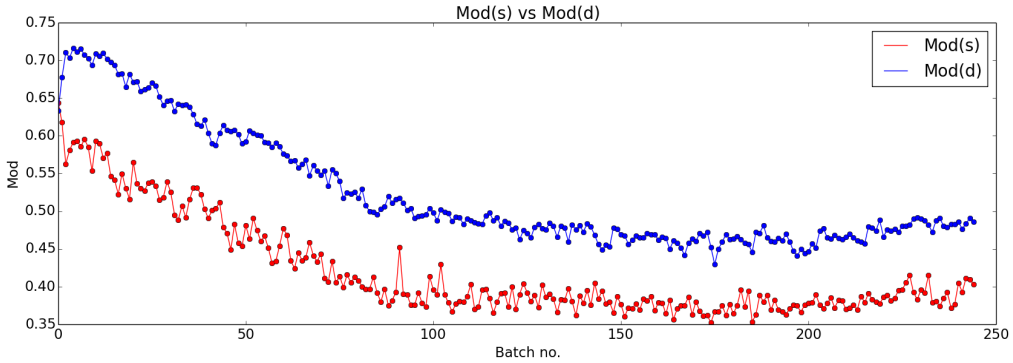

Figure 10: Modularity evolution for PGP graph with Node Spanning and Localized Batch Topology

1. *Node Spanning* with (vertex) *sorted batches*
2. *Node Spanning* with *randomized batches*
3. *Matching* with *sorted batches*
4. *Matching* with *randomized batches*

These experiments have been conducted on two undirected graphs:

1. PGPGiantCompo graph [2] with 10680 vertices and 24316 edges
2. SNAP Facebook graph [9] with 4039 vertices and 88234 edges

flipping them inside-out across vertices to simulate graph transition.

5.3.1 Modularity

Investigating figures 10, 11, 12, 13 for the evolution of modularity in case of the PGP $\rightarrow$ Flipped-PGP transition, we observe that the performance of the four approaches improves in that order. The Node Spanning on localized batches is unable to detect and track the transition in community structure both in the static and dynamic cases. However, the dynamic variant shows marginally better performance than the static version. This points at a failure of the *Merge strategy*, since a *Backtracking Strategy*

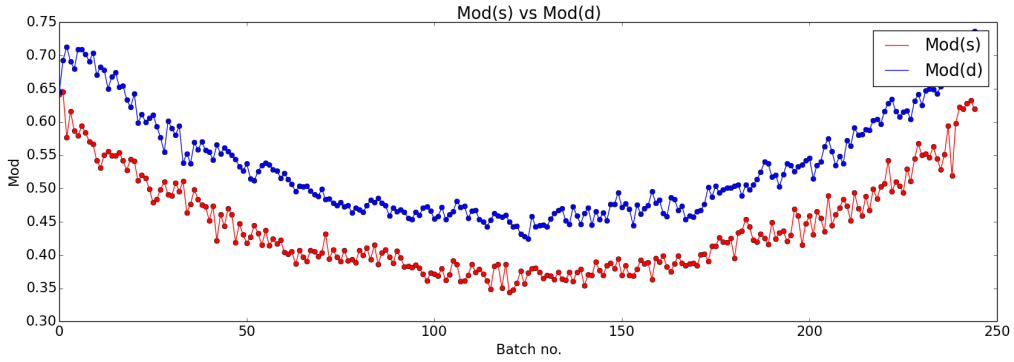

Figure 11: Modularity evolution for PGP graph with Node Spanning and Distributed Batch Topology

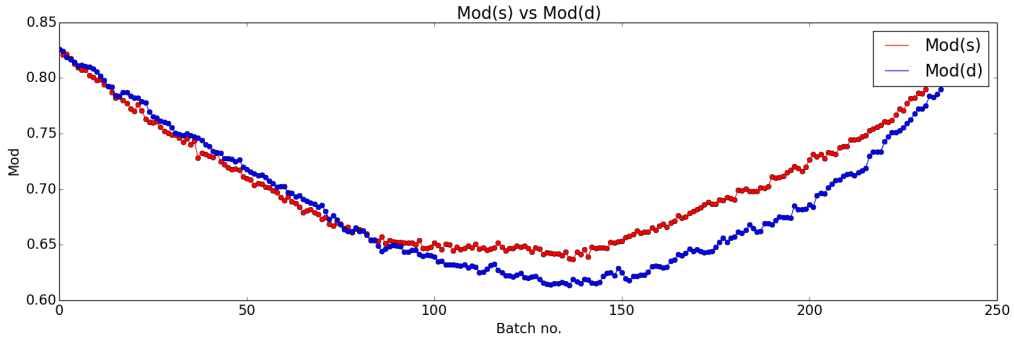

Figure 12: Modularity evolution for PGP graph with Matching and and Localized Batch Topology

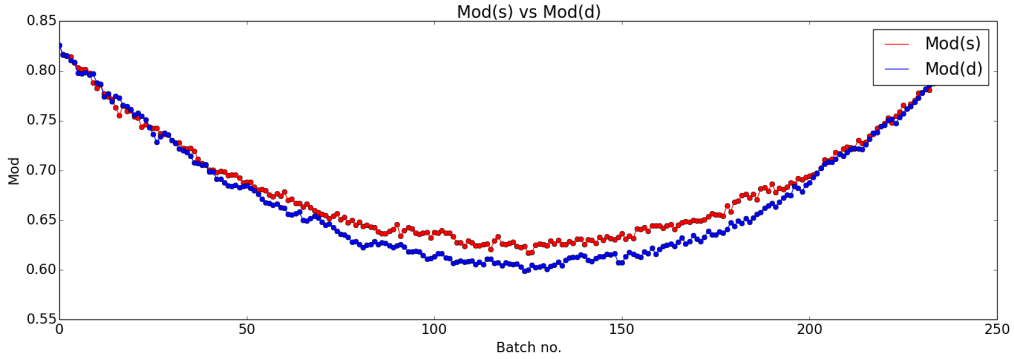

Figure 13: Modularity evolution for PGP graph with Matching and and Distributed Batch Topology

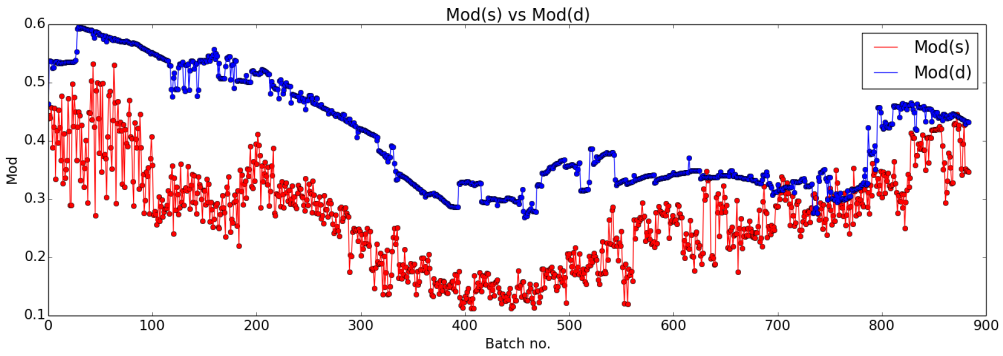

Figure 14: Modularity evolution for Facebook graph with Node Spanning and Localized Batch Topology

does not exist for static agglomeration With more uniformly distributed changes, the performance improves in case of the Distributed Batch Topology and both static and dynamic agglomerations are able to overcome the shortcomings of the Node Spanning strategy and track the community transition. The dynamic version performs better in this case too. In case of the Matching based merging, the performance of static agglomeration drastically improves in both localized and distributed batch topology. It can be seen that in case of localized batches, the dynamic approach lags behind the static one although buy the end it catches up. This lagging may be attributed to the inertia that the dynamic approach gains owing to the additional information stored in its dendogram. This inertia only slightly affects the distributed case due to its uniform nature of transition. Also it should be noted that the Matching approach also leads to higher values of modularity over Node Spanning for either batch topologies. In the Facebook $\rightarrow$ Flipped-Facebook graph transition, the modularity evolution for the four cases as shown in figures 14, 15, 16, 17, appears more non-uniform. Although all four cases loosely track the graph flip, the distributed batch cases exhibit smooth transition with the Matching approach again performing better. The discontinuous arcs in the localized cases can be attributed to the nature of the incoming batch as can also be seen in the drastic jumps in the number of communities in figures 30, 32.

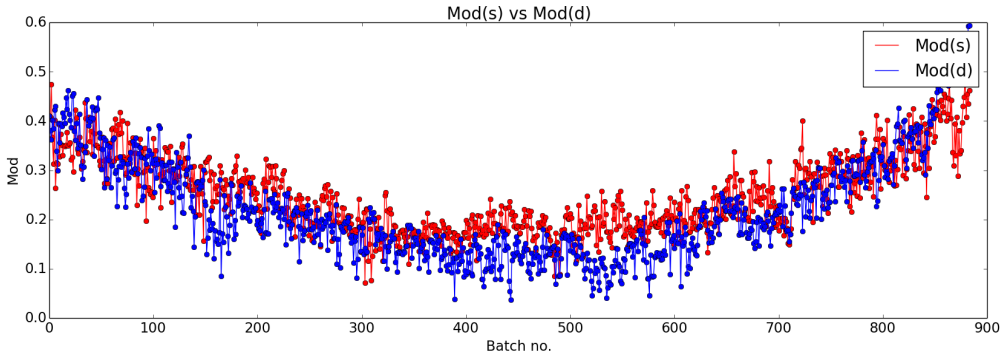

Figure 15: Modularity evolution for Facebook graph with Node Spanning and Distributed Batch Topology

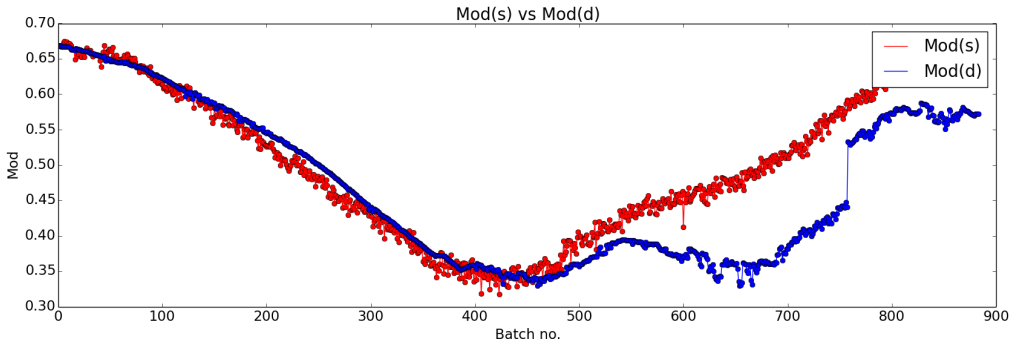

Figure 16: Modularity evolution for Facebook graph with Matching and and Localized Batch Topology

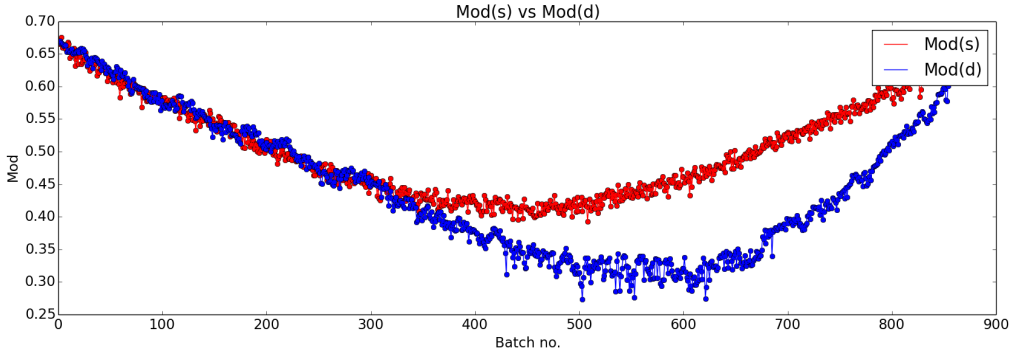

Figure 17: Modularity evolution for Facebook graph with Matching and and Distributed Batch Topology

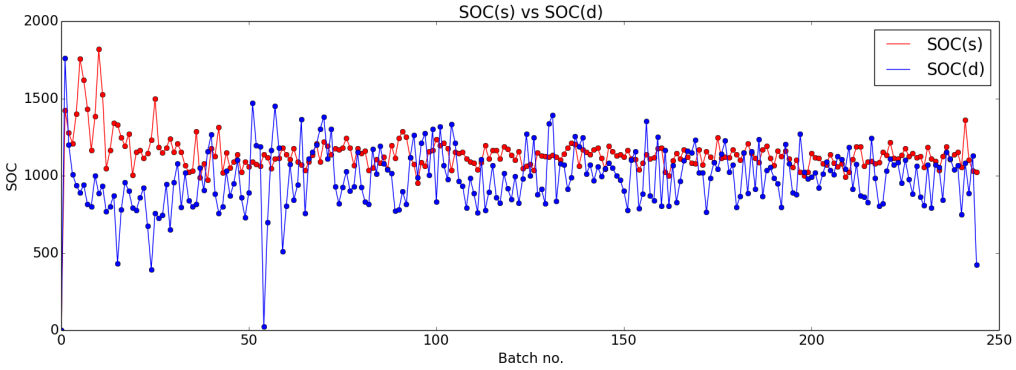

Figure 18: Size of Change evolution for PGP graph with Node Spanning and Localized Batch Topology

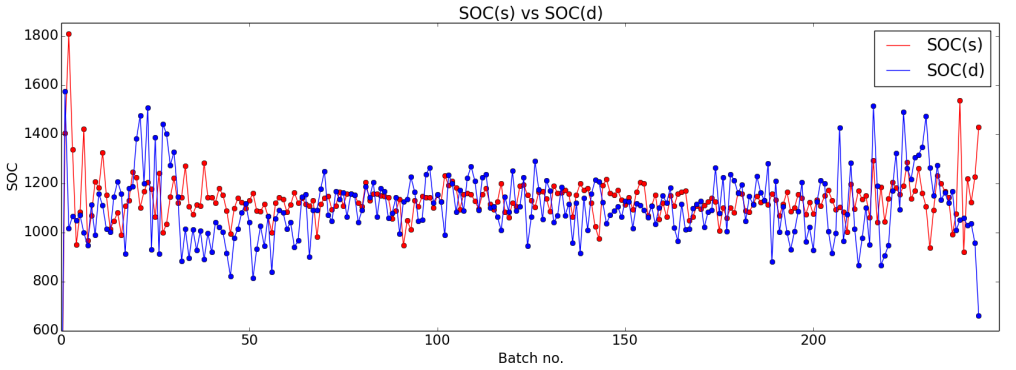

Figure 19: Size of Change evolution for PGP graph with Node Spanning and Distributed Batch Topology

The lag in the dynamic approach can be prominently seen in the Distributed Batch Matching case for the Facebook graph. Similar to the PGP results, Matching yields community structures with higher modularity than Node Spanning.

5.3.2 Size of Change

From figures 20, 21, 24 and 25, we observe that the size of change of dynamic reagglomeration for both PGP and Facebook graph tests with Matching is lower than that of static. This can be expected since the dynamic reagglomeration starts from a partial community structure borrowed from the preceding clustering. Thus it maintains information in the dendogram and passes it from an agglomerative step to another

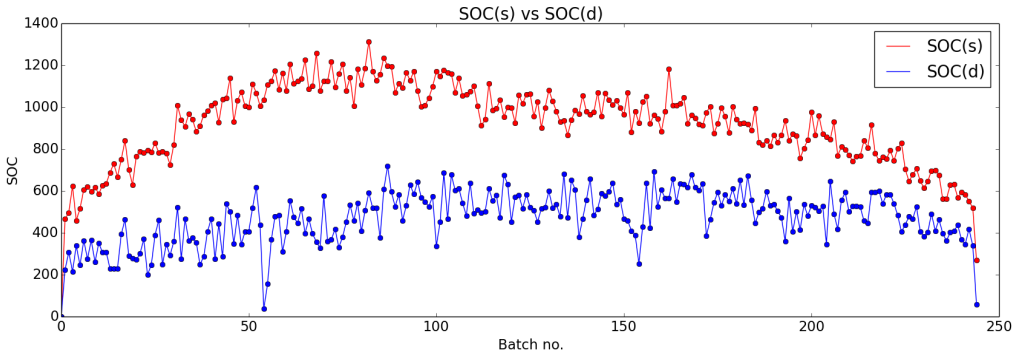

Figure 20: Size of Change evolution for PGP graph with Matching and and Localized Batch Topology

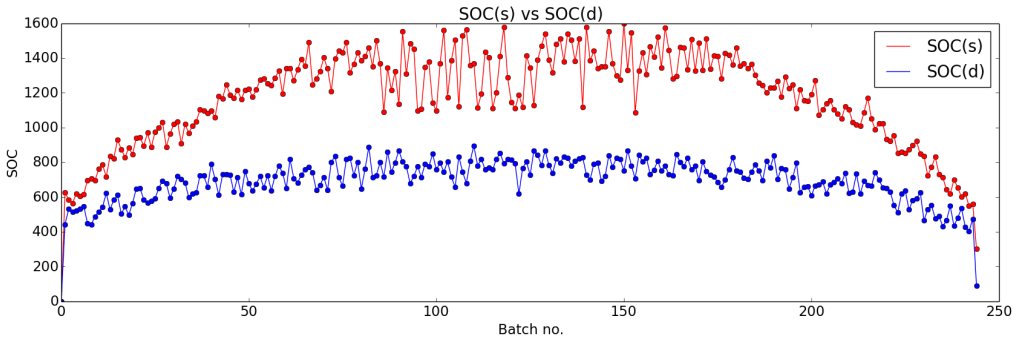

Figure 21: Size of Change evolution for PGP graph with Matching and and Distributed Batch Topology

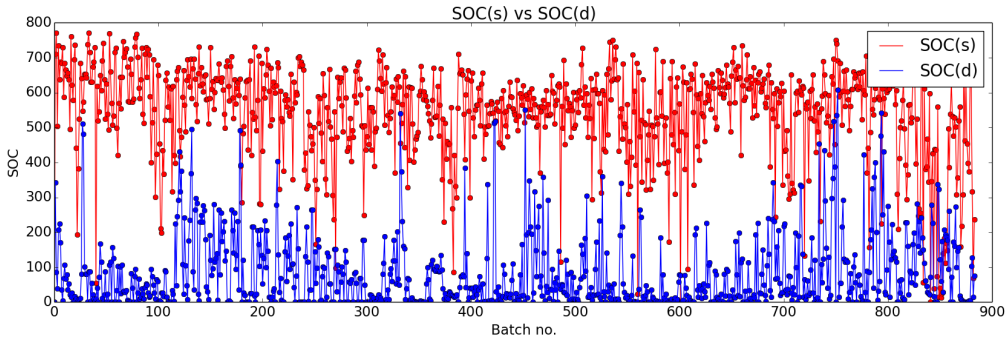

Figure 22: Size of Change evolution for Facebook graph with Node Spanning and Localized Batch Topology

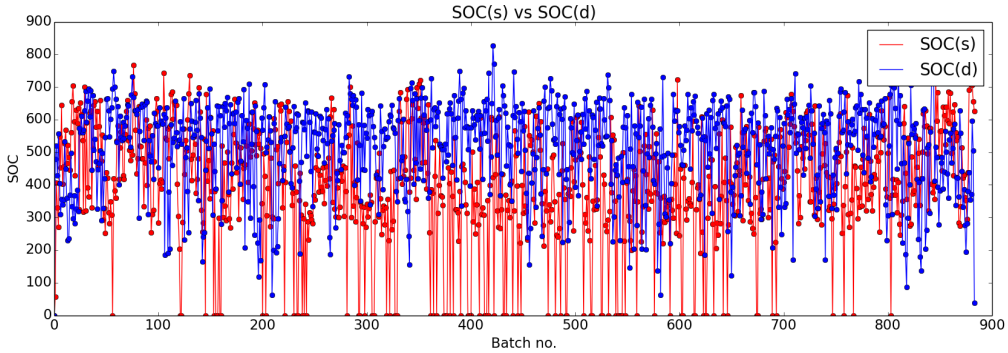

Figure 23: Size of Change evolution for Facebook graph with Node Spanning and Distributed Batch Topology

ensuring smoothness of transition between successive agglomerations. This advantage however appears to have been lost in the Node Spanning case shown in figures 18, 19, 22 and 23, where both static and dynamic show similar size of change values. This however is due to the nature of implementation of Node Spanning. Since Node Spanning followed a fixed order of vertices starting from 1 to n to check for local merges in the neighborhood irrespective of the magnitude of differential modularity, both the static and dynamic approaches follow roughly the same ordering of vertices which leads to broadly a similar ordering of merges with local variations. A significant change in this behavior can be observed only if the incoming batch of edge-updates drastically modifies the graph, thus creating new neighborhoods as can

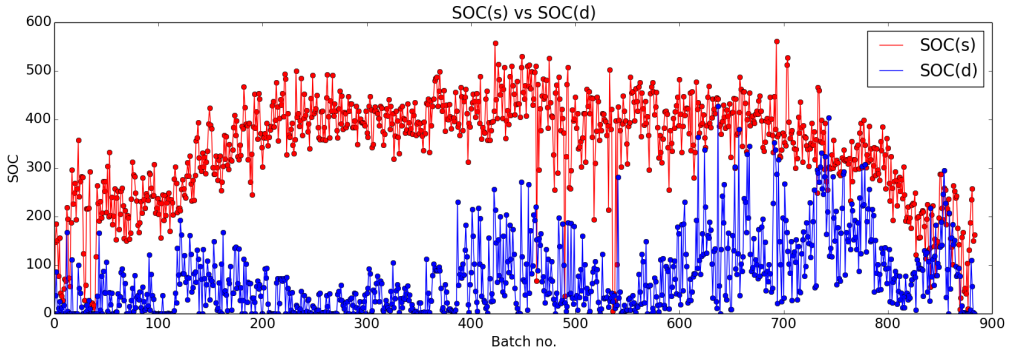

Figure 24: Size of Change evolution for Facebook graph with Matching and and Localized Batch Topology

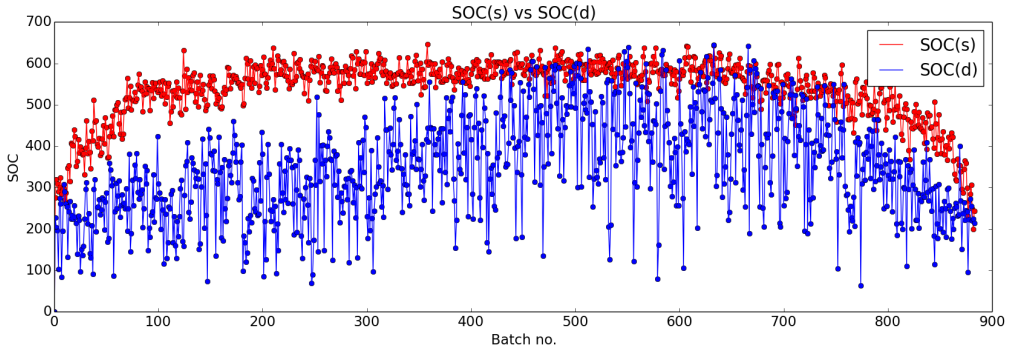

Figure 25: Size of Change evolution for Facebook graph with Matching and and Distributed Batch Topology

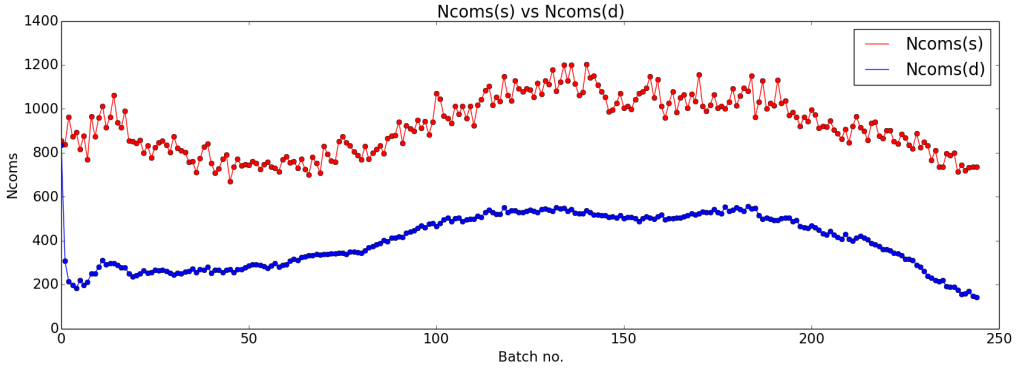

Figure 26: Number of Communities evolution for PGP graph with Node Spanning and Localized Batch Topology

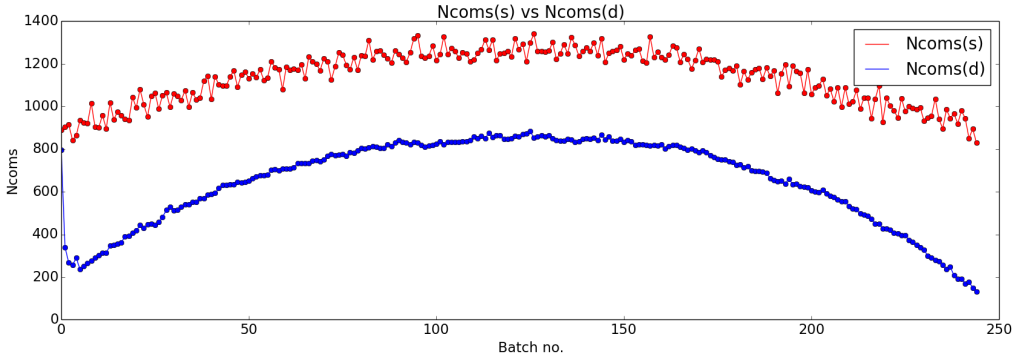

Figure 27: Number of Communities evolution for PGP graph with Node Spanning and Distributed Batch Topology

be seen by comparing the size of change plots for Node Spanning on the Facebook graph in figures 22, 23.

5.3.3 Number of Communities

In case of the PGP graph as seen in figures 26, 27, 28 and 29, the number of communities for the dynamic approach is always lower than those in the static case with. The difference between the two is only marginal in the Matching cases. This implies that the communities formed by dynamic reagglomeration tend to be lower in number and larger in size (higher average community size). This can again be attributed to the transitional property of the dynamic approach, since community vertices tend to

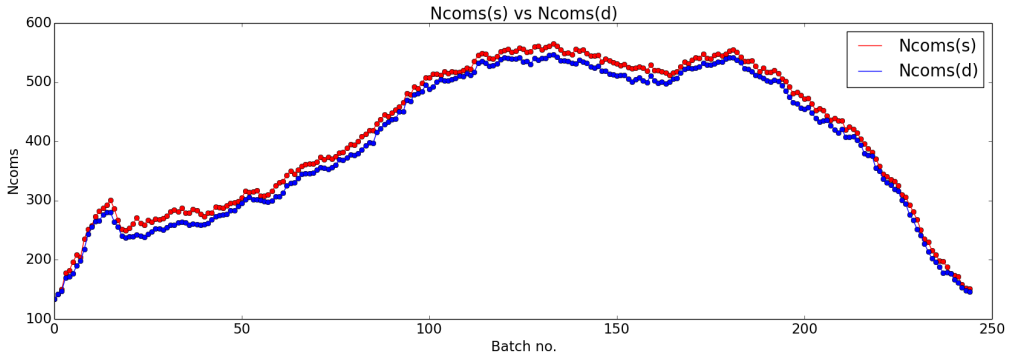

Figure 28: Number of Communities evolution for PGP graph with Matching and and Localized Batch Topology

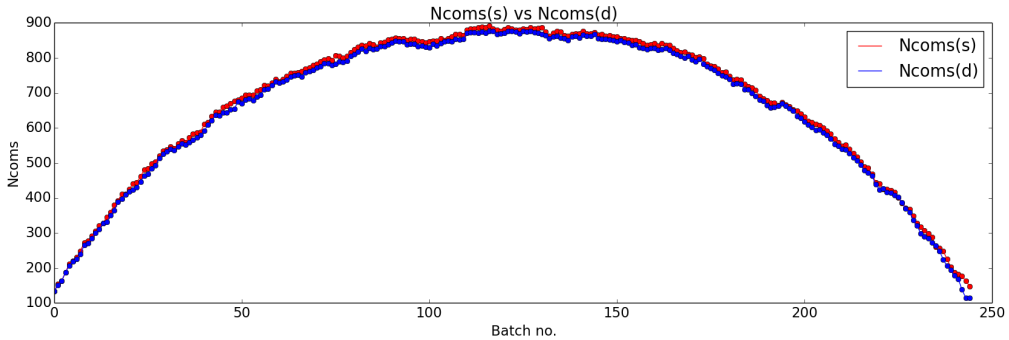

Figure 29: Number of Communities evolution for PGP graph with Matching and and Distributed Batch Topology

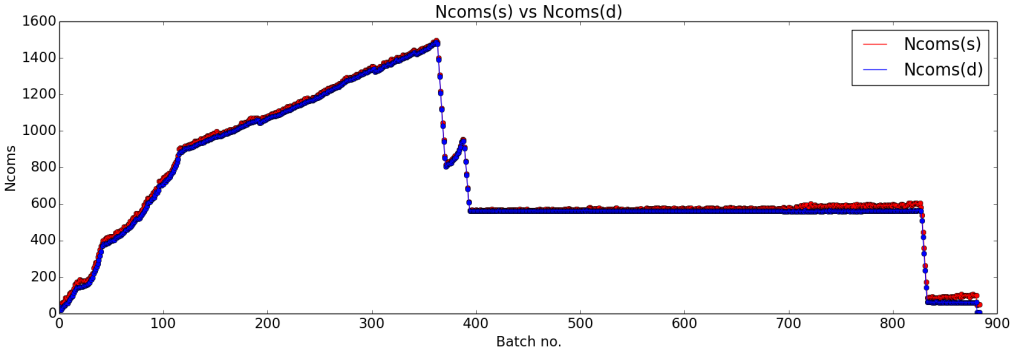

Figure 30: Number of Communities evolution for Facebook graph with Node Spanning and Localized Batch Topology

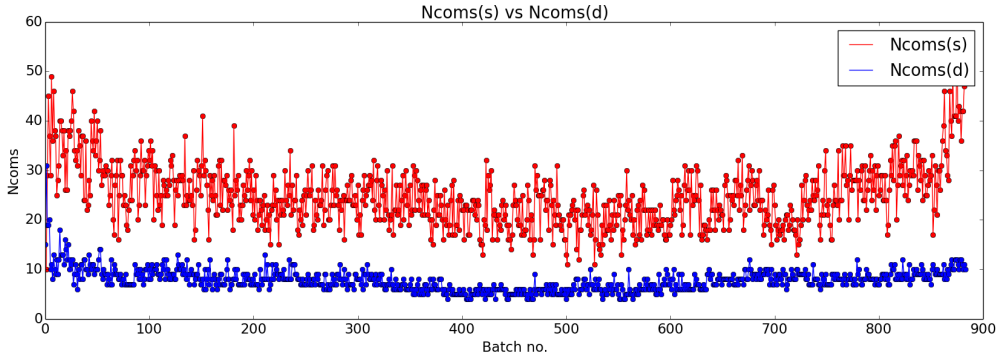

Figure 31: Number of Communities evolution for Facebook graph with Node Spanning and Distributed Batch Topology

merge with pre-existing larger community vertices instead of starting from singletons as in case of static agglomeration. Similar behavior is also seen in case of the the number of communities in the Facebook graph test. An interesting observation with respect to figures 26, 28 for the PGP graph test with localized batch topologies is that the plots for the number of communities in the dynamic case are almost identical irrespective of the merge strategy chosen. This similarity is even more pronounced in case of the Facebook graph as shown in figures 30 and 32. This similarity in case of the distributed batch topologies exists in the shapes of the plots for the dynamic case with an offset as seen in figures 27, 29, 31 and 33. Although further investigation may be necessary to understand this similarity, it appears that the dynamic agglomeration

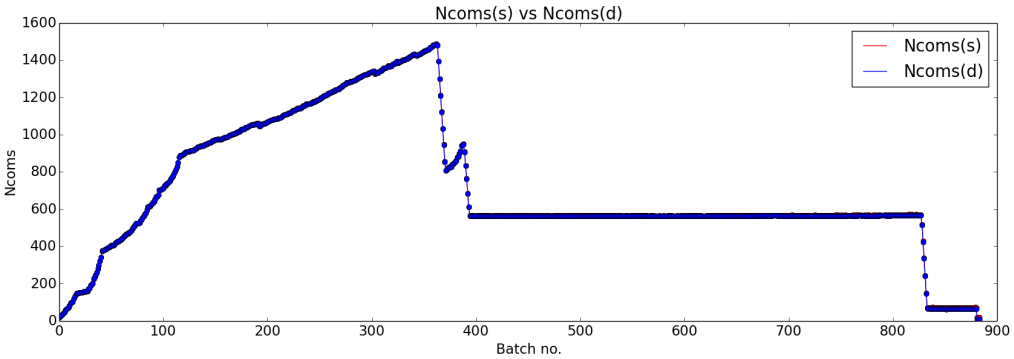

Figure 32: Number of Communities evolution for Facebook graph with Matching and and Localized Batch Topology

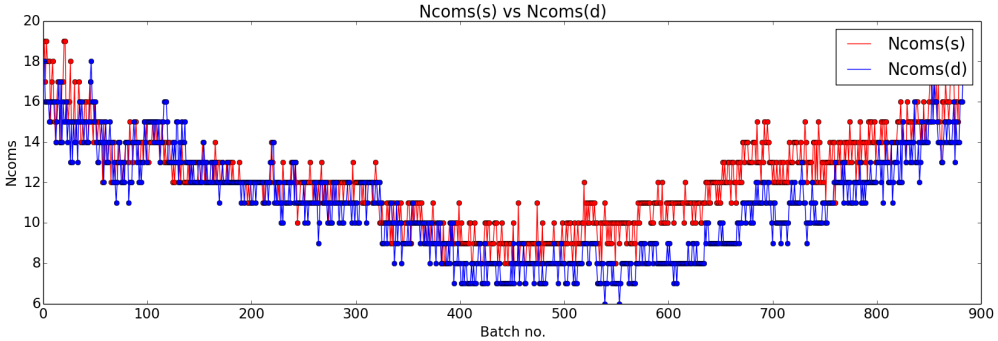

Figure 33: Number of Communities evolution for Facebook graph with Matching and and Distributed Batch Topology

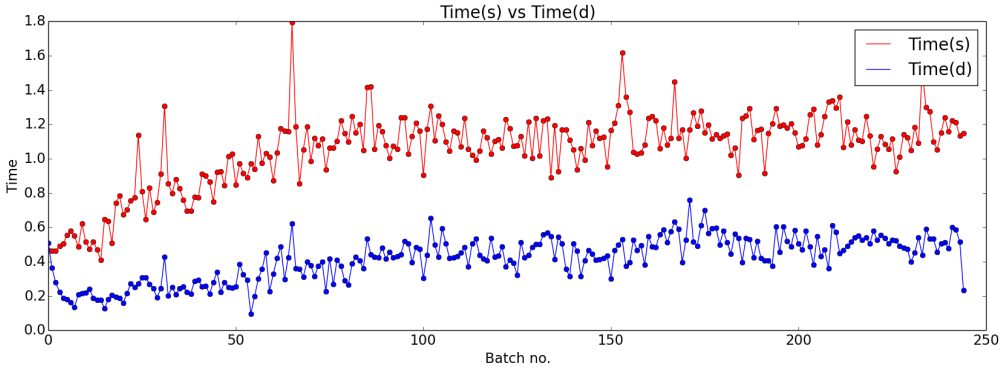

Figure 34: Runtime evolution for PGP graph with Node Spanning and Localized Batch Topology

approach ends up with similar overall number of communities in the final community structure, irrespective of the chosen merge order.

Finally, in order to justify the two sudden drops in the number of communities in the dynamic case for Facebook graph with localized batch additions, in figures 30 and 32, we may expect it to be due to multiple smaller community vertices being sucked into a large community vertex in a short duration. This hypothesis indeed turns out to be correct as seen in the plots of the maximum community size in figures 42, 43. The max community size rises nearly when the number of communities sharply drops. This observation is particularly precise for the Matching case in fig. 43.

5.3.4 Runtime

The runtime of dynamic reagglomeration in the Node Spanning approaches for both PGP and Facebook graph is significantly lower than the runtime for the static agglomeration as may be expected, since the static approach always starts agglomeration from scratch, at every iteration. This can be seen in figures 34, 35, 38 and 39. This however is not observed to be true in case of the Matching approaches for the PGP graph in figures 36, 37. In particular, it was observed that in the Matching approach, the runtime was directly related to the total number of active communities

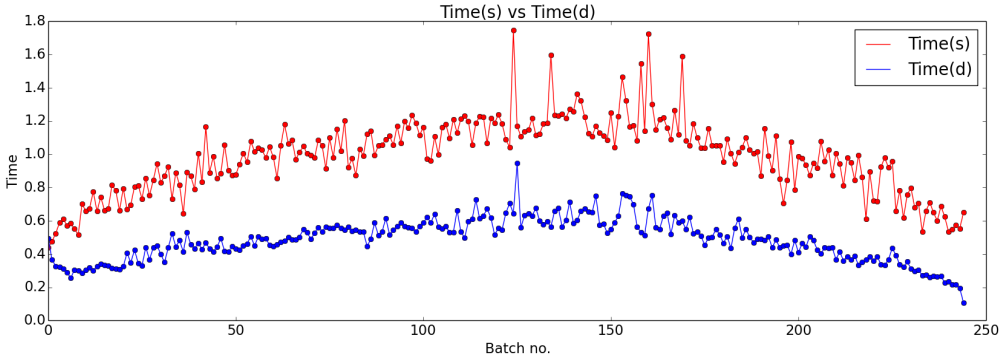

Figure 35: Runtime evolution for PGP graph with Node Spanning and Distributed Batch Topology

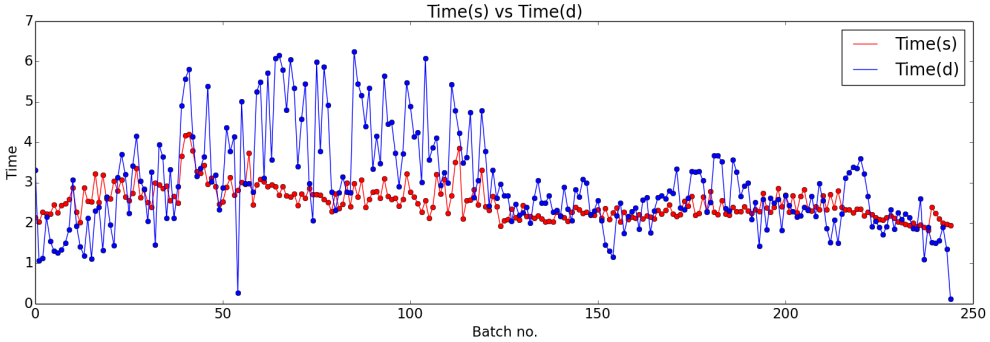

Figure 36: Runtime evolution for PGP graph with Matching and and Localized Batch Topology

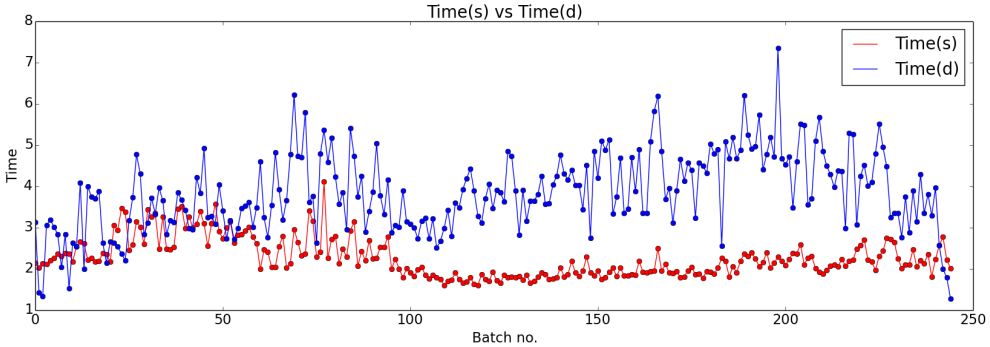

Figure 37: Runtime evolution for PGP graph with Matching and and Distributed Batch Topology

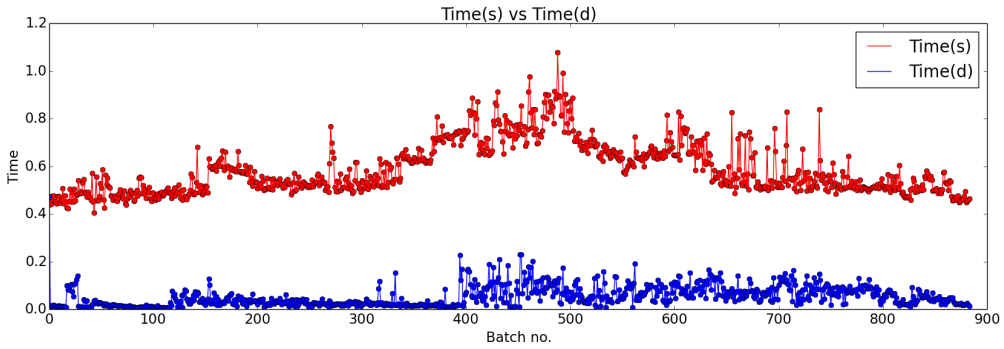

Figure 38: Runtime evolution for Facebook graph with Node Spanning and Localized Batch Topology

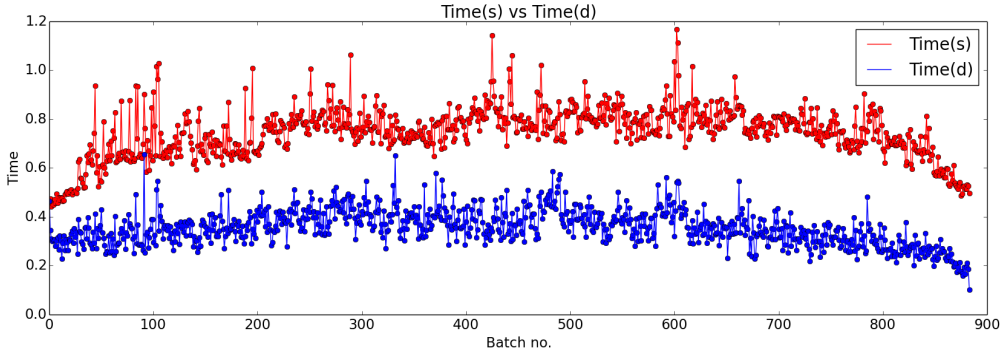

Figure 39: Runtime evolution for Facebook graph with Node Spanning and Distributed Batch Topology

and hence active neighbors to iterate over. Hence in the case of Matching approach for the Facebook graph with distributed batch addition in figure 41, the runtime for dynamic reagglomeration was lower than static. A peculiar behavior is observed in the runtime of the dynamic reagglomeration for Facebook in figure 40 with spikes at the center. This appears to be linked to the sudden surge in the number of merges that occur during these iterations as seen in figure 44. Clearly, the runtime of the algorithm is dependent on multiple drivers and detailed profiling is necessary to evaluate the exact cause of the runtime behavior.

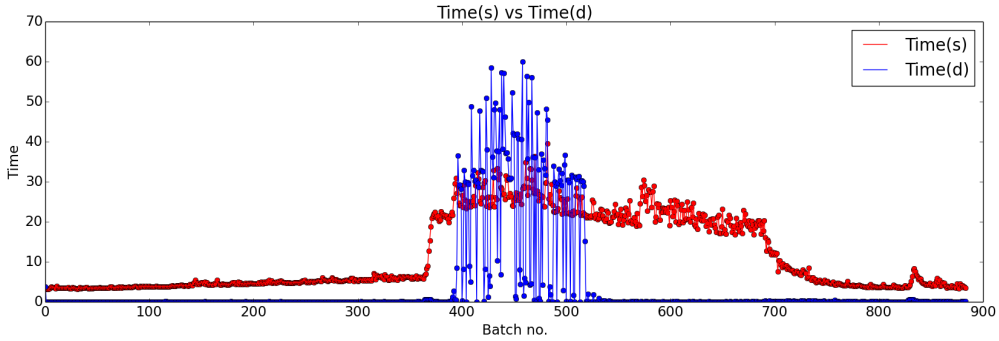

Figure 40: Runtime evolution for Facebook graph with Matching and and Localized Batch Topology

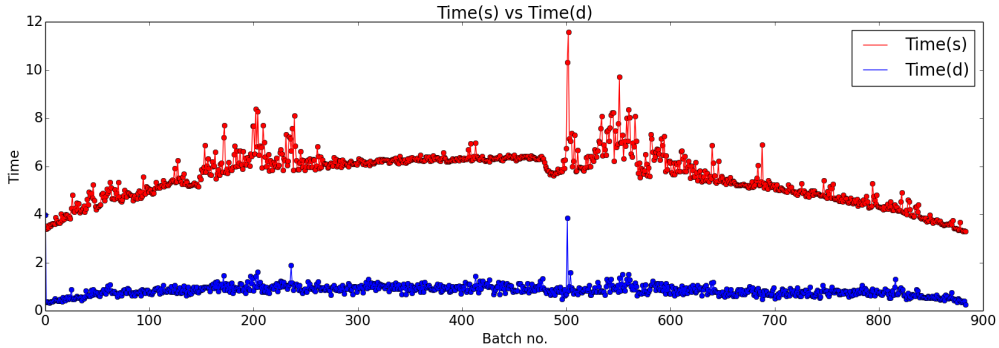

Figure 41: Runtime evolution for Facebook graph with Matching and and Distributed Batch Topology

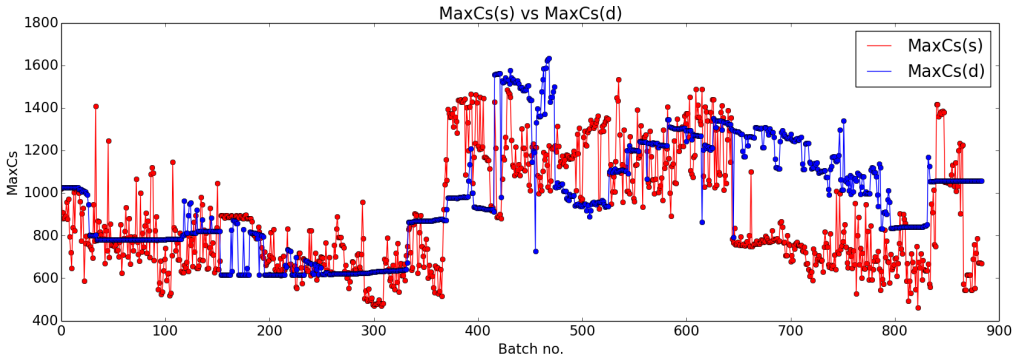

Figure 42: Max Community size evolution for Facebook graph with Node Spanning and Localized Batch Topology

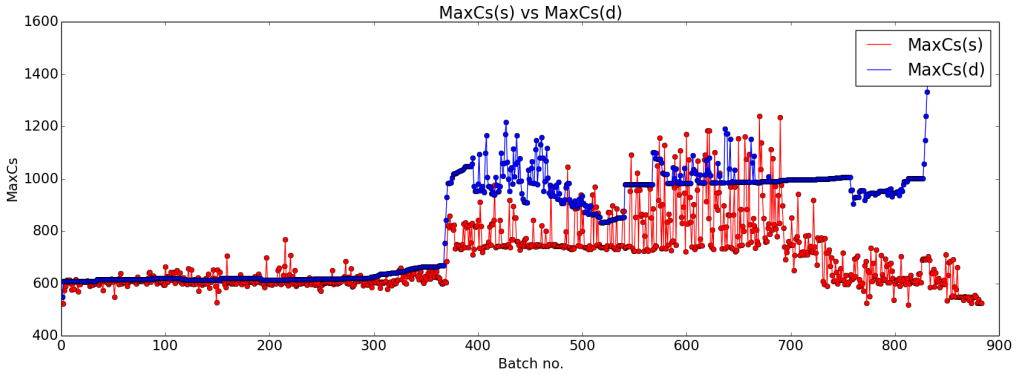

Figure 43: Max Community size evolution for Facebook graph with Matching and Localized Batch Topology

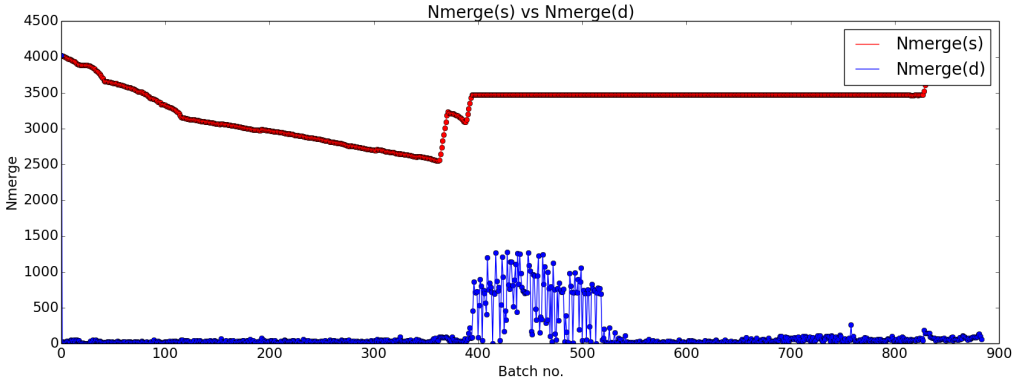

Figure 44: Number of merge operations for Facebook graph with Matching and Localized Batch Topology

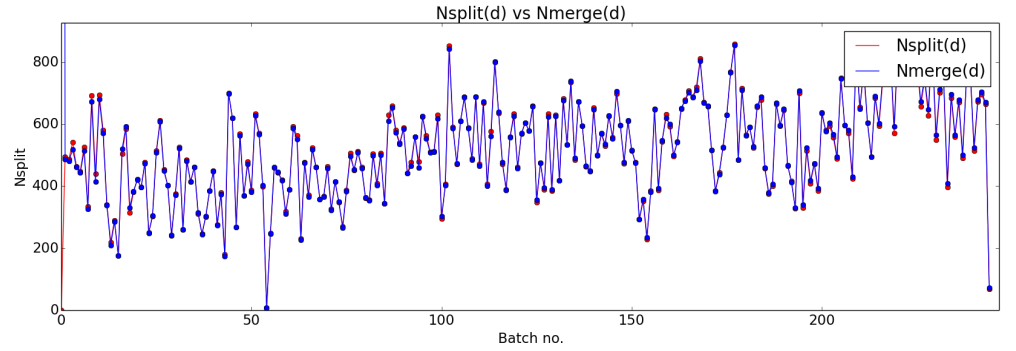

Figure 45: Number of split and merge operations for PGP graph with Matching and Localized Batch Topology

5.3.5 Number of Splits/Merges

The number of Split and Merge operations describe the extent of agglomeration performed at every iteration. From surveying the split and merge trends in the dynamic reagglomeration of the PGP and Facebook graphs, it was observed that in all the cases, the number of splits and merges at every iterations are almost equal. An illustration of the same can be seen in the overlapping plots of the number of merges and number of splits for the PGP graph in figure 45. This shows that although communities change vertex ownerships within an iteration, creation or deletions of new communities tends to have higher odds.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

- Here we have presented in detail the working and implementation of a Modularity based Backtracking Agglomerative scheme for dynamic community detection for efficient and smooth tracking of the community structure as the underlying graph changes.
- The results on the PGP and Facebook graphs show promise regarding the performance improvement of the algorithm over the previous memoryless approach.
- The current understanding of the performance of the algorithm in handling different graph structures is limited. Time profiling of the code to evaluate the behavior of the algorithm for various topologies of base and change graphs is necessary.
- Evaluating all the Backtracking strategies and mapping them to specific use cases is necessary.
- The trade-off between maintenance of agglomerative history in the form of an every growing dendogram and efficiency must be evaluated and quantified.
- While static agglomeration is agile and responsive to graph changes, it also tends to be trapped into local optima due to the non-linear nature of modularity. Dynamic agglomeration on the other hand may fail to respond to certain critical changes in the graph, diverging into a suboptimal solution. Objective criteria to distinguish between information and noise in the dendogram need to be identified as a means to actively transition between static and dynamic behaviors.

REFERENCES

[1] Bader, D. A., Berry, J., Amos-Binks, A., Chavarría-Miranda, D., Hastings, C., Madduri, K., and Poulos, S. C., "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," *Georgia Institute of Technology, Tech. Rep*, 2009.

[2] Boguña, M., Pastor-Satorras, R., Diaz-Guilera, A., and Arenas, A., "PGP giant component user network graph Physical Review E, vol. 70, 056122 (http://www.cc.gatech.edu/dimacs10/archive/clustering.shtml)," 2004.

[3] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D., "On Modularity – NP-Completeness and Beyond," 2006.

[4] Fortunato, S., "Community detection in graphs," *CoRR*, vol. abs/0906.0612, 2009.

[5] Gorke, R., Gaertler, M., Hübner, F., and Wagner, D., "Computational aspects of Lucidity-driven graph clustering," 2010.

[6] Görke, R., Maillard, P., Schumm, A., Staudt, C., and Wagner, D., "Dynamic graph clustering combining Modularity and Smoothness," *J. Exp. Algorithmics*, vol. 18, pp. 1.5:1.1–1.5:1.29, Apr. 2013.

[7] Hartmann, T., Kappes, A., and Wagner, D., "Clustering evolving networks," *ArXiv e-prints*, Jan. 2014.

[8] Hopcroft, J., Khan, O., Kulis, B., and Selman, B., "Tracking evolving communities in large linked networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, 2004.

[9] McAuley, J. and Leskovec, J., "Learning to discover social circles in ego networks, NIPS," 2012.

[10] Newman, M. E. J. and Girvan, M., "Finding and evaluating community structure in networks," *Physical Review*, vol. E 69, no. 026113, 2004.

[11] Riedy, E. J., Meyerhenke, H., Ediger, D., and Bader, D. A., "Parallel community detection for massive graphs," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, (Berlin, Heidelberg), pp. 286–296, Springer-Verlag, 2012.