Backtracking Re-agglomeration for Community Detection in Dynamic Graphs

Pushkar Godbole
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
pushkar.godbole@gatech.edu

Jason Riedy
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
jason.riedy@cc.gatech.edu

David Bader
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
bader@cc.gatech.edu

***Abstract*—In this paper, we present a streaming algorithm for agglomerative clustering, based on *Modularity* maximization for community detection in evolving graphs. We employ a backtracking strategy to selectively and consistently reverse clustering in regions of the graph affected by incoming changes. Selective backtracking and re-agglomeration eliminates the need to re-initiate clustering as the graph evolves, thus reducing the computation cost and the *Size of Change* in transitioning across community structures. In the 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*.**

I. Introduction

Community detection 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 [1] and arbitrary nature of the objective, deterministic identification of the optimal community structure is rendered practically impossible, for large (real-world) datasets. Hence many approximation algorithms and approaches have been developed to achieve near optimal clustering. Since many (contemporary) networks are not static but rather evolve over time, the static approaches are rendered inappropriate for clustering of dynamic graphs. Dynamic community detection however has remained a largely untrodden field aside from the recent research interest in the domain [2]–[5], particularly in context of social networks.

A. Contribution

In this work, we build upon our previous dynamic community detection algorithm based on memoryless localized vertex-freeing [5] and present a backtracking re-agglomerative clustering approach inspired from the dGlobal BT algorithm by Gorke et al. [2]. We test and evaluate the advantages and limitations of various agglomeration and backtracking strategies for the algorithm in context of *snapshot quality* (*Modularity*), *history cost* (Size of Change) and computation cost [6] for different streaming graph topologies. We show that a fast, consistent and good quality clustering is possible by recursively storing and updating the historical community structure as a dendogram, with significantly lesser computation cost in terms of backtracking and agglomeration, over the dGlobal BT algorithm by Gorke et al. [2].

B. Nomenclature

We use the standard graph theoretic notations to represent graph properties through out the paper [1]. $G(V, E)$ represents a graph with V representing the set of vertices and E the set of edges. $|V| = n$ and $|E| = m$. *Modularity* is represented by Q and *Size of Change* by SoC.
The terms, community detection, graph partitioning, clustering, have been interchangeably used to mean mapping of the vertices of G into $\mathcal{C}$ ($V \rightarrow \mathcal{C}$), where $\mathcal{C} = \{C_1, C_2, ..., C_p\}$ is a set of communities/partitions/clusters of vertices of G, such that for each $v \in V$, $v \in C_i$, for exactly one $i \in [1, p]$. Vertices representing agglomerations of multiple vertices into a single community are referred to as 'community vertices'. The *size(C)* of a community node C, refers to the number of vertices held by C. Community vertices with $size = 1$ are referred to as 'singletons' or 'singleton communities'.

C. Outline

Various metrics such as mutuality, conductance, betweenness have been employed, *modularity* being the most widely accepted one, to evaluate the quality of graph partitioning. These metrics, although very efficient, are significantly dependent upon the structure and size of the graph, thus making the clustering of dynamic graphs very challenging. The objective of our re-agglomeration algorithm is maximizing/maintaining *modularity* and minimizing *Size of Change* (defined later) in transitioning from an old community structure to the new, in progressively transforming graphs. The next section discusses the past work related to dynamic community detection, specifically in context of *Modularity* maximization. Section III introduces the metrics, *modularity* and *Size of Change*, and their dynamic charasterics and limitations in context of evolving graphs. Section IV describes our dynamic clustering algorithm with normative analysis of the backtracking and agglomeration strategies. Section V illustrates the performance of the algorithm on social network graph streams. Finally

Section VI concludes the study with sound recommendations and direction of future work.

II. Related Work

Traditionally, the problem of static graph clustering has been widely studied, in context of *modularity* optimization and beyond. The traditional methods of static graph clustering based on *Graph Partitioning*, *Partitional Clustering* and *Spectral Clustering* rely on a pre-specified cluster count and/or cluster size. [1]. These techniques render themselves unsuitable specifically in context of evolving graphs since the community structure of such graph also dynamically evolves with incoming changes, precluding the pre-specfication of such parameters. The *Hierarchical Clustering* techniques on the other hand prove suitable in case of dynamic graphs due to their localized and emergent nature. These methods work by either starting from individual vertices and merging them into communities that improve the community structure of partitioning (Agglomerative) or by starting from a single large community and splitting it into smaller communities that yield a strong community structure (Divisive). 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*. Following the seminal work on *modularity* based greedy agglomerative clustering by Newman and Girvan [8], many modifications and improvements in the algorithm have since been achieved to improve the speed and performance of agglomerative clustering in static graphs [9]–[12].

On the other hand, community detection in dynamic graphs has remained a relatively untouched field. The first work in this field by Hopcroft et al. [4] tracks the evolution of a graph by running 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. [13] 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 of solving the ILP. In our previous work, Riedy et al. [5] employ a localized vertex-freeing (to singletons), in affected regions of the graph and re-agglomerate from the intermediate partial clustering to yield a potentially higher *modularity* community structure. Nguyen et al. [3] also follow a similar approach of localized vertex-freeing around affected vertices and edges followed by their Quick Community Adaptation (QCA) algorithm to agglomerate the singleton vertices into suitable neighbors. Although these approaches yield very fast agglomeration algorithms, they tend to leave less flexibility for the clustering to sufficiently explore the solution space, on account of localized vertex-freeing. This may cause such methods to get trapped in local optima often due to larger communities sucking up smaller ones, referred to as the *blackhole problem* [14]. In this context, our backtracking algorithm allows consistent splitting of communities by storing and reverting past merges in affected regions of the graph and re-agglomerating thence. The dGlobal BT algorithm by Gorke et al. [2] employs a backtracking approach, very relevant to our algorithm. They conclude that their backtracking algorithm yields improved results in terms of *modularity*, *smoothness* and execution time over other algorithms. We expand on these previous works [2], [5] to develop and infer the most suitable backtracking and agglomeration strategy for dynamic community detection.

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 graph has denser connections between the vertices within communities but sparse connections between vertices across communities. The metric of *Modularity* coined by Newman and Girvan [8], is a widely used measure of the strength of partitioning, based on the premise that, a community structure would have collections of vertices more strongly connected internally than would occur from random chance.

In general, *modularity* based agglomeration techniques start off by placing all graph vertices into singleton communities, recursively merging the communitiy vertices along edges, yielding an improvement in *modularity*. The newly agglomerated graph formed from the merged vertices then acts as the starting point for the subsequent clustering, until no further improvement in *modularity* is possible, thus reaching a local maximum.

In case of dynamic graphs, as the graph evolves (due to insertion and deletion 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, has been defined to quantify this change in partitioning w.r.t. vertex transfers.

A. 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:

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

The value of P_{ij} can be computed using the 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 vertics i and j in G and hence in R_G). Thus the above equation for modularity can be simplified to:

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

Notwithstanding its limitations, in particular w.r.t. the resolution limit [15], modularity has certain properties that make it suitable for agglomerative clustering [1], [16]:

- 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 is negative
- Isolated (degree 0) vertics have no impact on the modularity of a community structure. Thus the optimal clustering for graphs $G(V_1, E_1)$ and $G'(V_1 + V_2, E_1)$ would have equal optimal modularities, with all vertices from V_2 isolated into singleton communities.
- In the clustering with maximum modularity, each connected component is a separate cluster. i.e. placing disconnected subgraphs of G, if they exist, in different clusters yields the highest modularity
- Maximum modularity clustering is non-local: Changes to one region of the graph may propagate to the other regions, yielding a completely different optimal clustering

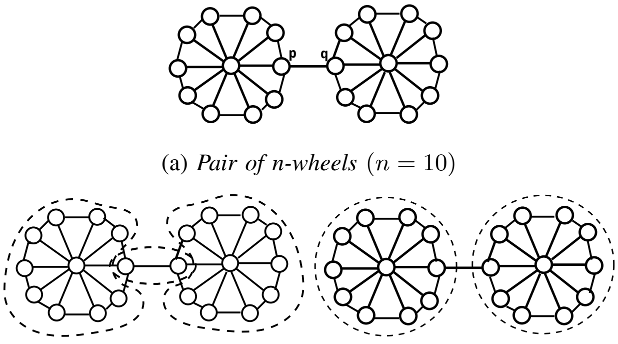
B. Differential Modularity

One of the primary advantages of using Modularity as a metric for dynamic 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:

$$\Delta Q_C = \frac{E_C}{m} - \frac{Vol_C^2}{4m^2} \tag{3}$$

Where ΔQ_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.

Most *modularity* based agglomeration schemes prioritize merges based on their differential contribution to overall *modularity*. However, merge along the highest ΔQ edge doesn't always guarantee a near-optimal community structure. Although *modularity* serves as a good metric to evaluate pre-existing clustering, its not always very well suited to evaluate the best possible merges. This can mainly be attributed to the fact that the initial merges in the agglomeration are driven by the second term in equation (3). The first term becomes dominant only after the vertices merge into non-singleton communities. Thus a best-merge-first technique initially prefers to merge along least volume edges, promoting the merge along bridge edges (edges connecting two communities), among others. The *pair of n-wheels* graph in figure 1a illustrates the impact of using a best-merge-first approach, yielding an additional bridge community shown in figure 1b, bringing down the modularity to $\frac{4n}{4n+1} - \frac{1}{2} - \frac{4n+21}{(4n+1)^2}$ from $\frac{4n}{4n+1} - \frac{1}{2}$, for the optimal clustering shown in figure 1c. A best merge first agglomeration on a rectangular *2D* lattice of such *n-wheels* yields a highly suboptimal clustering populated heavily by bridge communities. The paper by Brandes et al. [16] illustrates another example using a *regular 2-clique* graph showing how the worst possible clustering can manifest using a best-merge-first agglomeration strategy.



(a) *Pair of n-wheels* $(n = 10)$

(b) Best merge first clustering

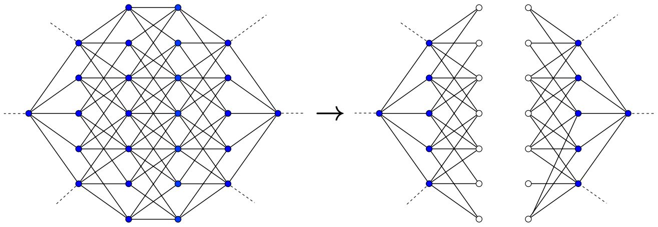
(c) Optimal clustering

Fig. 1: Best merge first clustering in (a) prioritizes merge along bridge-edge *(p, q)*, leading to suboptimal clustering (b) instead of (c) *[Dashed lines represent clusters]*

Secondly, given the ΔQ of the neighboring edges left in the wake of a merge changes after every merge, the complexity of the best-merge-first strategy goes to $O(mn)$ for an instance of agglomeration. Therefore, while the best-merge-first strategy affords good quality clustering in general, its relevance especially in the context of dynamic community detection is contingent upon the graph topology and computation cost considerations.

C. 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 node 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 node:



(a) Initial graph community (b) Updated graph community

Fig. 2: As the graph transitions from (a) to (b), a conventional localized vertex-freeing strategy yields a suboptimal community structure where two disconnected subgraphs end up belonging to the same cluster. *[Blue vertices lie in the same cluster C. White vertices become free singletons, that will likely re-join C through the connecting edges, in subsequent agglomeration.]*

- Nodes that were not in its community, but now are (Join)
- Nodes that were in its community, but now aren't (Leave)
- Nodes 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 node (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 *Join* and *Leave* 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 node 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)*.

D. A case for Backtracking

Dynamic aggomerative algorithms work by identifying affected community vertices (based on a chosen strategy) and modifying them for subsequent reagglomeration. However, there lies a fundamental difference between community vertices affected by edge insertion and deletion operations. In particular, edge insertions 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. Conventional dynamic agglomeration techniques use vertex-freeing around affected regions of the graph, buy plucking out *affected* vertices from communities and placing them in singleton communities. Consider for example the case of a change-batch that deletes edges from a community, splitting it into two disconnected subgraphs as shown in figure 2. A strict neighborhood based vertex-freeing strategy may only free vertices local to the affected edges, while other vertices (which must now ideally belong to two separate communities) are still tagged to the same community. A consistent backtracking strategy on the other hand is not restricted just to the neighborhood of the affected edges. It works by locally reverting the community structure by chronologically splitting community vertices to eliminate the now invalidated merges, reverting the community structure back to a point from where all the affected community vertices (not necessarily in the neighborhood) can re-evaluate their affiliations in the subsequent agglomeration.

IV. The Backtracking Re-agglomeration Algorithm

Any dynamic agglomeration algorithm is primarily composed of two components. For each incoming batch of graph changes, the algorithm must execute the following two steps:

- **Modification:** *Identifying and reverting* affected regions of the community structure in response to the incoming graph change, to transform the past clustering $\mathcal{C}$ into a partial clustering $\mathcal{C}'$ to allow subsequent agglomeration, thus facilitating local exploration of the solution space.
- **Re-agglomeration:** Given the partial clustering $\mathcal{C}'$, *identifying and merging* the community vertices to reach a local optimum with a community structure potentially *better* than the previous, in light of the graph change.

The graph changes in this context are restricted only to edge insertions and removals. Vertex insertions and removals have not been considered. However, it may be noted as presented in Section III-A that, newly added vertices can be considered to be part of a pre-existing *rogue-vertex cloud (V')*, disconnected from rest of the graph and each other, since such vertices dont affect *modularity*. Therefore, this approach can be conveniently extended to vertex insertions and removals. A *better* community structure in context of our algorithm is the one that yields a higer *modularity*.

Similar to static agglomeration, the algorithm starts off by agglomerating singleton community 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 a recursive update of the dendogram to reflect the inertion/remval of the edge. Finally, the algotithm merges community vertices along edges that improve *modularity*. The order in which the community edges are considered for the merge is based on the *agglomeration strategy*, discussed subsequently.

A. Backtracking

Given the advantages of backtracking over the conventional techniques such as vertex-freeing, as noted in Section III-D, it is chosen as the community modification technique for our algorithm. The backtracking strategy defines how a batch of incoming edge changes is handled in the community dendogram of the graph. Broadly, the backtracking strategies presented

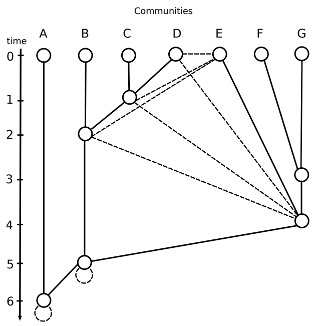


Fig. 3

D	E
C	
B	
	G
	B
A	A

TABLE I

Figure 3: Dendogram of adjacency induced by base-edge DE
TABLE I: Parent stacks of vertices D, E

below are a combination of two methods: *Edit Edge* and *Split Community*.

1) Backtracking Methods:

- **Edit Edge:** Keeps the community structure intact but recursively edits the underlying adjacency of the community vertices to change the weights of the edges. Editing the underlying adjacency requires knowledge of the chronological order in which the corresponding merges occurred. Figure 3 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 I.
  The solid lines in the 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 on the community graph. The timestamps associated with these merges help determine 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 list of edges induced (IE_{ab}) by base edge *ab* can be generated from the parent stacks of a and b using Algorithm 1. Here $P_i(a)$ represents the i^{th} parent of a, a being the zeroth. $T_i(a)$ represents the timestamp of the i^{th} merge of a.
- **Split:** Splits a community into its two components, reversing the latest merge in the community dendogram. For instance, splitting community node A in Figure 3 would be accomplished by subtracting the adjacency of node B from that of A and recursively reverting the parent stack of all vertices in community node B (B, C, D, E, F, G) by one (from A to B).

The three backtracking policies below define the *extent of backtracking*, given an edge change *ab*:

- **Edit Edge:** Merely execute *Edit Edge* for edge *ab* keeping the community structure intact
- **Split to Separation:** Recursively *Split* the community node containing edge *ab* until vertices a and b belong to separate communities followed by *Edit Edge*
- **Split to Singletons:** Recursively *Split* the community vertices containing vertices a and b, until a and b both become singleton communities followed by *Edit Edge*

Algorithm 1 Induced Edges

```
$i = size(P(a))$ // Size of parent stack of a
$j = size(P(b))$ // Size of parent stack of b
while $i! = 0$ or $j! = 0$ do
    $IE_{ab} \leftarrow (P_i(a), P_j(b))$
    if $T_i(a) > T_j(b)$ then
        $k = j - 1$
        while $P_i(a)! = P_k(b)$ and $k > 0$ do
            $IE_{ab} \leftarrow (P_i(a), P_k(b))$
            $k \leftarrow k - 1$
        end while
        $i \leftarrow i - 1$
    else
        $k = i - 1$
        while $P_k(a)! = P_j(b)$ and $k > 0$ do
            $IE_{ab} \leftarrow (P_k(a), P_j(b))$
            $k \leftarrow k - 1$
        end while
        $j \leftarrow j - 1$
    end if
end while
```

Note that *Edit Edge* is executed in all of the backtracking policies. This ensures that the dendogram is always kept updated through the execution, eliminating the need to recompute the adjacency as in case dGlobal BT [2].

2) Backtracking Strategies: The four edge change types and the relevant backtrack policies to handle them are described below.

1. **Inter-community insertion:** Strengthens the bridge edges or in other words obsoletes the present clustering. A large insertion 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* policy where the edge addition acts as a forward looking change towards a community merge. The second case would need the use of *Split to Singletons*, giving the algorithm an opportunity to reconsider the present clustering.
2. **Intra-community insertion:** Strengthens the existing community. A large insertion of edges to a community can yield a maximum modularity community structure where:
   - The community becomes more closely connected, if

edge insertions are approximately uniform throughout the community

- A region of the community becomes more closely connected than the rest, forming a strong community in itself, if edge insertions are concentrated in that region

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

3. **Inter-community deletion:** Strengthens the presently defined community structure irrespective of the number and topology of deletions. Thus this case can be easily handled using the *Edit Edge* policy.
4. **Intra-community deletion:** Weakens the community, since the edge deletion obsoletes the past merges driven by the existence of that edge. It necessitates the re-computation of the clustering without the influence of this edge. The *Split to Separation* or *Split to Singletons* policies 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.

B. Re-agglomeration

Given a partial clusterin $\mathcal{C}'$, 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, the best-merge-first strategy has its limitations as noted in Section III-B. To that end, we propose two approaches to ensure balance between performance and efficiency of the agglomeration. It may be noted here that all edges considered in context of re-agglomeration are inter-community edges between presently active community vertices.

1) Node Spanning: This approach merges positive *differential modularity* edges along an expanding frontier of active community vertices.

- Starting from a node i, the algorithm merges the node with a neighbor j, that yields a positive change in modularity.
- In order to maintain consistent community tagging, the merged community node is tagged i if $size(i) > size(j)$ and j otherwise.
- This continues until i becomes a sub-community of a neighboring node i'. The process is then repeated for node i' until it becomes a sub-community of one of its neighbors, and so on.

The agglomeration stops when no more positive *differential modularity* edges remain. 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 after a merge.

2) Matching: At each agglomerative step, the matching approach finds a set of disconnected edges, merging along which leads to an overall improvement in modularity, contracting the graph along all such edges. For each agglomerative step:

1) Starting from the existing clustering $\mathcal{C}'$, the algorithm identifies a maximum weight greedy 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 $\mathcal{C}''$

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

This algorithm can be considered a static variant of the *best merge first* approach in which, the change in values of *differential modularity* 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 through *Backtracking* and *Re-agglomeration* steps for every incoming batch of edge additions and deletions ensures the maintenance of a good quality (high *Modularity*), consistent (low *Size of Change*) dynamic community structure, as the graph evolves.

V. Results

To test the performance of the algorithm w.r.t. the backtrack and agglomerate strategies, we evaluate it on the (SNAP) Facebook graph [17] (G_{fb}) with 4039 vertices and 88234 edges and the PGP Giant Component graph [18] (G_{pgp}) with 10680 vertices and 24316 edges.

A. Experiment

In order to emulate a significant change in community structure of the graph in a testable manner, the experiment starts off from a graph G, modifying it through a stream of edge deletions from G and insertions from a cannonical graph G^{flip}. G^{flip} in this case represents an image of G with vertex labels flipped ($i \rightarrow n - i$), so that vertex 1 becomes vertex n, 2 becomes $n - 1$ and so on. This ensures that the optimal community structures of the initial graph G and the final graph G^{flip} are exactly the same. A dynamic algorithm can then be objectively evaluated by comparing the initial and final modularities.

For the results below, all edge changes are streamed in equal batches of 100 insertions and 100 deletions each (244 batches for G_{pgp} and 883 for G_{fb}). To simulate real and pathological cases, two stream topologies have been used:

- *Randomized Graph Stream (RGS):* To evaluate the algorithm under typical cases, edge insertions and deletion streams are randomized over all vertices; ensuring the incoming batch of changes is approximately distributed over the entire graph
- *Localized Graph Stream (LGS):* To evaluate the algorithm under extreme cases, edge insertions and deletions streams are sorted based on vertex labels; thus

increasing the likelihood of an entire neighborhood being created/destroyed in a single batch

B. Variations

The experimental results illustrated here show the performance of the algorithm with LGS and RGS for $G_{fb} \rightarrow G_{fb}^{flip}$ and $G_{pgp} \rightarrow G_{pgp}^{flip}$, using two agglomeration strategies:

1) Node Spanning (*NS*)
2) Matching (*M*)

and two backtracking strategies:

1) *BT1*:
   - Inter-community Insertion: *Edit Edge*
   - Inter-community Deletion: *Edit Edge*
   - Intra-community Insertion: *Edit Edge*
   - Intra-community Deletion: *Split to Separation*
2) *BT2*:
   - Inter-community Insertion: *Edit Edge*
   - Inter-community Deletion: *Edit Edge*
   - Intra-community Insertion: *Split to Separation*
   - Intra-community Deletion: *Split to Separation*

In contrast, the dGlobal BT algorithm by Gorke et al. [2] uses a costlier best-merge-first agglomeration approach and a more aggressive backtracking strategy:

- Inter-community Insertion: *Split to Singletons*
- Inter-community Deletion: Nothing
- Intra-community Insertion: *Split to Separation*
- Intra-community Deletion: *Split to Singletons*

C. Analysis

Owing to the significantly high average degree (edges/vertex) of G_{fb}, of 21.85, as compared to 0.44 for G_{pgp}, the effect of LGS is significantly pronounced on G_{fb}. Thus the peculiar behavior of various approaches becomes clearly visible in case of LGS_{fb} as may be observed in the results below.

1) Modularity: Broadly we observe that all approaches are able to track the graph transition and recover modularity with varying degrees of success. Figures 4 and 5 show the trends for LGS_{fb} and RGS_{fb} respectively while TABLE II shows the performance over G_{pgp}. While the Static Matching based agglomeration approach yields the best modularity results in all cases, the Dynamic Matching based approaches are consistently able to closely track the static trend. The Node Spanning techniques perform worse off compared to Matching, with Static Node Spanning performing the worst. Dynamic Node Spanning approaches are however able to detect a final clustering significantly better than their initial clustering with a final modularity between 85 - 90% of the Static Matching optimum in all cases. In all cases, the $BT2$ backtracking strategy performs marginally better than $BT1$, for both Matching and Node Spanning based agglomeration schemes. A notable difference can be observed between $MBT1$ and $MBT2$ for LGS_{fb} in figure 4. This can also be verified from TABLE III, where the final values of p, μ_{size} and σ_{size} for $BT2$ closely match those for Static, while $BT1$ reaches a different local maximum with lower modularity. Finally, we observe that in all cases, the Dynamic modularity trends lag behind Static Matching, owing to the inertia of information from the previous partial clustering $\mathcal{C}'$. This is clearly visible in figure 5 for RGS_{fb}.

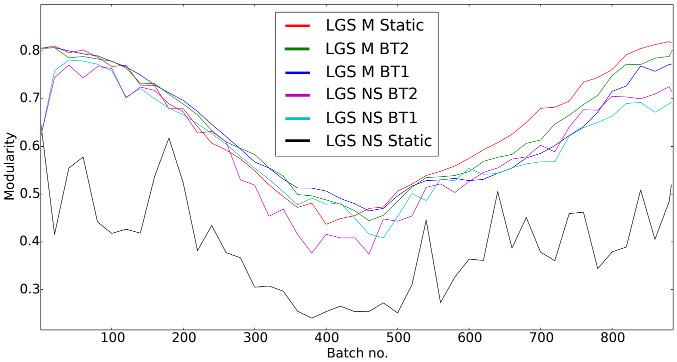


Fig. 4: G_{fb}: *Moduarity* evolution for Localized Graph Stream

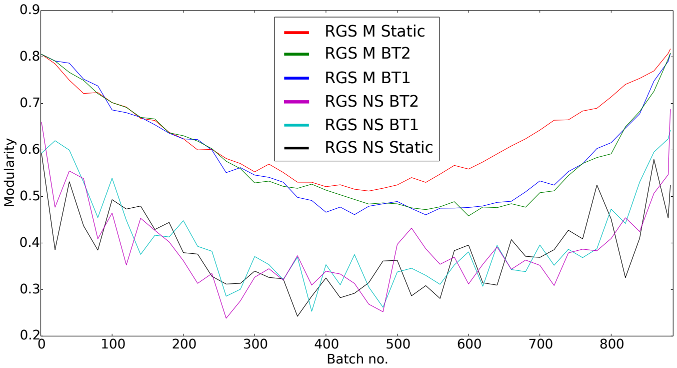


Fig. 5: G_{fb}: *Modularity* evolution for Randomized Graph Stream

2) Size of Change: All Dynamic Matching approaches yield a significantly smoother cluster transitions with fractional *Size of Change* values compared to the Static Matching results as seen in TABLE IV, with *BT1* affording lower values of than *BT2*, due to higher backtracking in the latter. This advantage however is largely lost in the Node Spanning approach with

Modularity G_{pgp}	*Initial*	*Final*			
		LGS		*RGS*	
		Static	*BT1*	*Static*	*BT1*
M	0.858	0.862	0.857	0.862	0.858
NS	0.678	0.658	0.779	0.668	0.768

TABLE II: G_{pgp}: Modularity

LGS_{fb}	Initial	Final		
		M Static	M BT1	M BT2
p	12	12	8	11
μ_{size}	336.58	336.58	504.88	367.18
σ_{size}	179.61	187.71	359.29	213.9

TABLE III: LGS_{fb} Key parameters *[p: No. of communities, μ_{size}: Community size mean, σ_{size}: Community size standard deviation]*

Avg SOC/batch	LGS_{fb}	RGS_{fb}	LGS_{pgp}	RGS_{pgp}
M Static	303	457	983	1205
M BT2	97	259	570	817
M BT1	74	239	538	772
NS Static	559	459	1161	1135
NS BT2	127	526	1080	1143
NS BT1	76	486	1032	1144

TABLE IV: Average *Size of Change* per batch

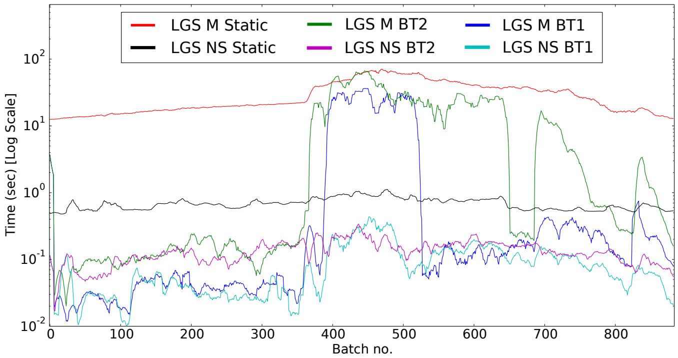


Fig. 6: G_{fb}: Execution time/batch for Localized Graph Stream

Size of Change values for Dynamic approaches comparable to Static. This is owing to the fact that Node Spanning orders the merges based on an expanding frontier of vertices, notwithstanding of the values of *differential modularity*, thus following a similar merge order for both Static and Dynamic approaches. A significant difference between the two can therefore be observed only if the incoming batch of edge-updates drastically modifies the graph, thus creating new neighborhoods as can be expected in case of LGS_{fb}. Indeed the values of *Size of Change* for $NSBT1$ and $NSBT2$ in case of LGS_{fb} are significantly lower than that for Static. Finally, under the matching scheme, the average *Size of Change* values for LGS can be seen to be considerably lower than those for RGS in all cases. This can be attributed to the fact that in an edge stream randomized over the graph, the vertices would have a higher tendency to jump back and forth across communities during the graph transition. Since LGS modifies the graph in a topologically sorted fashion, a region of the graph and hence the local clustering would largely remain unchanged once updated.

3) Execution Time: The critical advantage of dynamic agglomeration becomes evident in the time saving over the static approach, as seen in TABLE V. In all cases, the trend for execution time follows the order: *M Static* > *M BT2* > *M BT1* > *NS Static* > *NS BT2* > *NS BT1*, as may be expected from the computation cost. The Node Spanning approaches exhibit an order of magnitude improvement over Matching. This advantage is most pronounced in case of LGS_{fb} as illustrated in figure 6. The sudden jumps in execution times of the Dynamic Matching trends show the algorithm reacting to a drastic change in the graph.

Thus we observe that critical trade-offs exists between various agglomeration and backtracking schemes. While the Static Matching approach yields the highest Modularity values, it performs worst in terms of *Size of Change* and is prohibitively slow, especially for high degree graphs such as G_{fb}. While the Node Spanning based dynamic approaches yield reasonable modularity values, within 85% of those afforded by Static Matching at a small fraction of the computational cost, they sacrifice the *Size of Change* in doing so. All the dynamic clustering approaches are able to recover, if not exceed the initial modularity in transitioning from G to G^{flip}, with much relaxed backtracking strategies.

Avg Time/batch	LGS_{fb}	RGS_{fb}	LGS_{pgp}	RGS_{pgp}
M Static	29.985	19.367	1.997	1.951
M BT2	10.402	2.166	0.97	1.663
M BT1	3.872	1.199	0.92	1.517
NS Static	0.817	0.998	1.389	1.422
NS BT2	0.135	0.476	0.577	0.737
NS BT1	0.09	0.469	0.537	0.729

TABLE V: Average execution time per batch (seconds)

VI. Conclusions

We have presented here a generic framework for a backtracking based dynamic reagglomeration algorithm. We demonstrate and evaluate various agglomeration and backtracking strategies for realizing the dynamic reagglomeration. The dendogram based backtracking technique yields a much more consistent clustering compared to the conventional vertex-freeing based dynamic schemes. For evaluation of the algorithm, we present an objectively testable experiment that modifies the graph based on vertex label swapping, while maintaining the community structure. Experimental evaluation on G_{fb} [17] and G_{pgp} [18] confirms that the dynamic agglomeration approaches perform at par if not better than the static approaches in terms of modularity, while yielding smoother transitions at fraction of the computation cost, particularly in case of high degree graphs such as social networks. The analysis reveals that the much economical agglomeration strategies such as Matching and Node Spanning; and much relaxed backtracking strategies with respect to conventional approaches, perform exceedingly well under dynamic clustering.

That said, further investigation of the schemes on additional graph data sets and stream topologies is necessary to corroborate the presented results. While the Matching based agglomeration approach performs better than Node Spanning in terms of Modularity and *Size of Change*, the cost of identifying a maximum weight matching is fairly high. Further optimization of the Matching approach may yield better runtime performance. In that context, because matching creates disconnected sets of edges, it makes the matching based agglomeration step parallelizable. The subsequent step would be to build on our previous work to create a backtracking based parallel agglomeration scheme. Finally, further analysis of the backtracking strategies may facilitate the implementation of an adaptive backtracking scheme, that determines the suitable backtrack policy based on the incoming edge-batch topology.

References

[1] S. Fortunato, “Community detection in graphs,” *CoRR*, vol. abs/0906.0612, 2009. [Online]. Available: http://arxiv.org/abs/0906.0612

[2] R. Görke, P. Maillard, A. Schumm, C. Staudt, and D. Wagner, “Dynamic graph clustering combining modularity and smoothness,” *J. Exp. Algorithmics*, vol. 18, pp. 1.5:1.1–1.5:1.29, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2444016.2444021

[3] N. P. Nguyen, T. N. Dinh, Y. Shen, and M. T. Thai, “Dynamic social community detection and its applications,” *PLoS ONE*, vol. 9, no. 4, pp. 1–18, 04 2014. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0091431

[4] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, “Tracking evolving communities in large linked networks,” *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, 2004. [Online]. Available: http://www.pnas.org/content/101/suppl_1/5249.abstract

[5] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, “Parallel community detection for massive graphs,” in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, ser. PPAM’11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 286–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31464-3_29

[6] D. Chakrabarti, R. Kumar, and A. Tomkins, “Evolutionary clustering,” in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’06. New York, NY, USA: ACM, 2006, pp. 554–560. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150467

[7] T. Hartmann, A. Kappes, and D. Wagner, “Clustering Evolving Networks,” *ArXiv e-prints*, Jan. 2014.

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

[9] A. Clauset, M. E. J. Newman, and C. Moore, “Finding community structure in very large networks,” *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.70.066111

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008. [Online]. Available: http://stacks.iop.org/1742-5468/2008/i=10/a=P10008

[11] P. Schuetz and A. Caflisch, “Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement,” *Phys. Rev. E*, vol. 77, p. 046112, Apr 2008. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.77.046112

[12] J. Riedy, D. A. Bader, and H. Meyerhenke, “Scalable multi-threaded community detection in social networks,” in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1619–1628. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2012.203

[13] R. Gorke, M. Gaertler, F. Hbner, and D. Wagner, “Computational aspects of lucidity-driven graph clustering,” 2010.

[14] J. Wu, A. E. Hassan, and R. C. Holt, “Comparison of clustering algorithms in the context of software evolution,” in *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, Sept 2005, pp. 525–535.

[15] S. Fortunato and M. Barthélemy, “Resolution limit in community detection,” *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1073/pnas.0605965104

[16] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, “On modularity – np-completeness and beyond,” 2006.

[17] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.

[18] M. Bogua, R. Pastor-Satorras, A. Diaz-Guilera, and A. Arenas, “Pgp giant component user network graph physical review e, vol. 70, 056122 (http://www.cc.gatech.edu/dimacs10/archive/clustering.shtml),” 2004.