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 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 *Modularity* and *Size of Change*, with average speedups ranging between 3x and 7x for various schemes.**

I. Introduction

Community detection is one of the most critical aspects of graph analysis, having 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 graph clustering. Since many (contemporary) networks are not static but rather evolve over time, re-running static community detection over such graphs is inefficient. Yet, dynamic community detection 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], based on selective split and merge operations over the community *dendogram*. We test and evaluate 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 introduce a testable streaming graph experiment based on vertex label-swapping to compare these strategies. We show that a fast, smooth and good quality dynamic clustering is possible by storing and updating the historical community structure as a dendogram, with significantly relaxed backtracking and agglomeration strategies, compared to the dGlobal BT algorithm by Gorke et al. [2]. Over all the experiments, we achieve 3x to 7x speedups for various dynamic approaches with final Modularity values ranging between 85 - 99.5% of the static best (from the static Matching method) for all schemes.

B. Nomenclature

We use common 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 $\mathcal{C}$ ($V \to \mathcal{C}$), 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 vertex 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, Modularity have been employed to evaluate the quality of graph partitioning. [7] These metrics, although very efficient, are significantly dependent upon the structure and size of the graph, thus making the clustering of dynamic graphs 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 characteristics 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 are rendered unsuitable in context of evolving graphs since they require pre-specification of cluster count and/or cluster size. [1]. The Hierarchical Clustering techniques on the other hand prove suitable for dynamic graphs due to their localized and emergent nature. These methods work by either starting from individual vertices recursively merging them into communities that improve the community structure (Agglomerative) or by starting from a single large community, recursively splitting it into smaller ones, yielding a strong community structure (Divisive). Another class of methods dubbed Label Propagation [8], relies on local exchange of vertices across communities based on connectivity patterns. Our focus however is on agglomerative methods that optimize the numerical metric Modularity. Following the seminal work on Modularity based greedy agglomerative clustering by Newman and Girvan [9], many modifications and improvements in the algorithm have since been achieved to improve the speed and performance of agglomerative clustering in static graphs [10]–[13].
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. [14] 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* [15]. 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 of transition 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, wherein the graph has denser connections between the vertices within communities but sparse connections between vertices across communities. The metric of Modularity defined by Newman and Girvan [9], 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 community 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 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 changes apropos. Modifying the clustering to maximize/maintain Modularity entails creation and deletion of communities along with transitions of vertices between communities. Minimizing these changes ensures a smooth transition from an old clustering to new. The Size of Change metric, has been defined to quantify this change across successive clusterings.

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), A represents the $n \times n$ adjacency matrix such that, $A_{ij} = 1$ if an edge exists between vertices i and j in G and 0 otherwise. Similarly, Pij represents expected number of edges between vertices i and j in the canonical random graph R_G. The community equivalence $\delta_{ij} = 1$ 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:

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

Graph G with m edges has $2m$ stubs (half edges). For vertices i and j, their stubs can be connected to any of the remaining $2m-1$ stubs. For $m >> 1$, $2m-1 \approx 2m$. Therefore, the probability of having an edge between i and j in R_G is given by $k_i/2m \times k_j/2m = k_ik_j/4m^2$, making the expected number of edges, $P_{ij} = 2m \times k_ik_j/4m^2 = k_ik_j/2m$ (where k_i and k_j are the degrees of vertics i and j in G and hence in R_G). Notwithstanding its limitations, in particular with regards to the resolution limit [16], Modularity has certain properties that make it suitable for dynamic agglomerative clustering [1], [17]:

- For an undirected and unweighted graph G, the Modularity Q lies between the range $-1/2 \leqslant Q \leqslant 1$. A positive Q implies, the number of edges within communities exceeds the number expected based on R_G. The Modularity of the entire graph as a single community is zero while that with 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.
- 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*

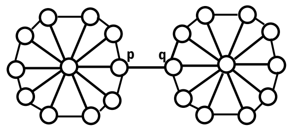
One of the primary advantages of using Modularity as a metric for dynamic agglomeration is that, its computation is localized, in that it is possible to compute the contribution to the overall Modularity by each individual community. The expression for the same is given by:

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

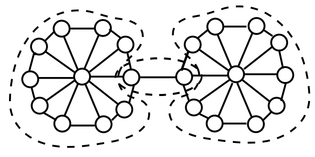
Where Q_C is the contribution to Modularity by community C, E_C is the number of internal edges in community C and Vol_C is the volume or the sum of the degrees of all vertices internal to community C. Thus the *Differential Modularity* contribution ΔQ, due to merge of communities C_i and C_j, along community edge ij, of weight w can be expressed as:

$$\Delta Q_C = \frac{w}{m} - \frac{Vol_{C_i}Vol_{C_j}}{2m^2} \tag{3}$$

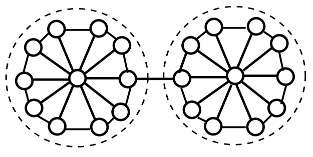
Most Modularity based agglomeration schemes prioritize merges based on their Differential Modularity. Although Modularity serves as a good metric to evaluate pre-existing clustering, its not always very well suited to evaluate the best possible merges. This can mainly be attributed to the fact that the 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



(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]*

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. [17] illustrates another example using a *regular 2-clique* graph showing how the worst possible clustering can manifest using a best-merge-first agglomeration strategy.

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[1] metric must reflect the difference between two clusterings, based on the cluster associations of all vertices. However, since the clusters do not have explicit tags, the measure of change for each vertex must be defined relative to its neighborhood. This is realized based on the following three quantities for each vertex (v):

- $J(v) = |\{w \in N(v) : C(v) = C(w) \wedge C'(v) \neq C'(w)\}|$
- $L(v) = |\{w \in N(v) : C(v) \neq C(w) \wedge C'(v) = C'(w)\}|$
- $S(v) = |\{w \in N(v) : C(v) = C(w) \wedge C'(v) = C'(w)\}|$

Where $N(v)$ is the set of current neighbors of vertex v. $C(v)$ and $C'(v)$ respectively represent the current and previous community labels for vertex v. and $J(v)$, $L(v)$ and $S(v)$ represent the number of neighbors that joined, left and stayed in v's community respectively. Based on these three quantities, two parameters are defined to quantify change in the neighborhood of each vertex (v):

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

[1]In submission: Anita Zakrzewska, Eisha Nathan, James Fairbanks, David Bader "A Local Measure of Community Change in Dynamic Graphs", ASONAM 2016

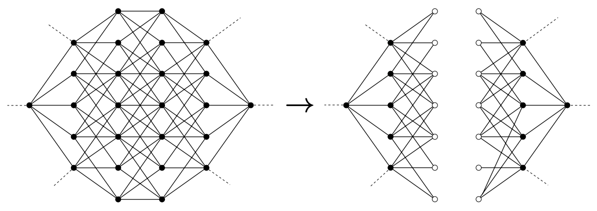

(a) Initial community (b) Updated community

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. *[Black vertices lie in the same cluster C. White vertices become free singletons]*

Where $c_J(v)$ and $c_L(v)$ are the *Join* and *Leave* parameters for v's neighborhood. Then Size of Change(SoC) is defined as:

$$SoC = \sum_{v \in V} |\{v : c_J(v) > \mu_{c_J} + 2\sigma_{c_J} \vee c_L(v) > \mu_{c_L} + 2\sigma_{c_L}\}|$$

Where μ and σ represent the mean and standard deviation respectively.

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 (generally) forward looking changes that affect the subsequent community structure while edge deletions are backward looking, since they preclude the existing community structure. Conventional dynamic agglomeration techniques use vertex-freeing around affected regions of the graph, by 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 to the neighborhood of the affected edges. It works by locally reverting the community structure by splitting community vertices to eliminate the now invalidated merges, reverting the community structure back to a point where the clustering was agnostic to the presence/absence of the affected edges.

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:

- **Reversion:** *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 $\mathcal{C}'$
- **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 deletions. Vertex insertions and deletions 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 don't affect Modularity. Therefore, this approach can be conveniently extended to vertex insertions and deletions.
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. 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 insertion/deletion of the edge. Finally, the algorithm 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

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

1) Backtracking Methods:

- **Edit:** 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 agglomeration order. 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 the agglomeration progresses along the y-axis and the dashed lines represent the edges induced by the base edge on the community graph. In the 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 first. $N_i(a)$ represents the step number of the i^{th} merge of a.

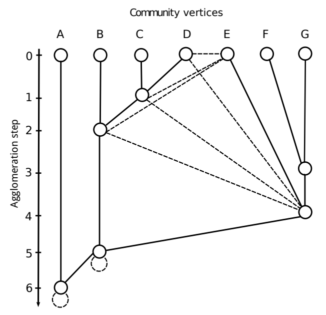


Fig. 3

A	A
	B
	G
B	
C	
D	E

TABLE I

Figure 3: Dendogram of adjacency induced by base-edge DE

TABLE I: Parent stacks of vertices D, E

Algorithm 1 Induced Edges

```
$i \leftarrow size(P(a))$; $j \leftarrow size(P(b))$
**while** $i > 0$ **and** $j > 0$ **do**
    $m \leftarrow i$; $n \leftarrow j$
    **do**
        $IE_{ab} \leftarrow (P_m(a), P_n(b))$
        **if** $P_m(a) = P_n(b)$ **then break end if**
        **if** $N_m(a) \geqslant N_n(b)$ **then** $n \leftarrow n - 1$ **end if**
        **if** $N_m(a) \leqslant N_n(b)$ **then** $m \leftarrow m - 1$ **end if**
    **while** $m > 0$ **and** $n > 0$
    **if** $m = 0$ **then** $j \leftarrow n - 1$ **else**
    **if** $n = 0$ **then** $i \leftarrow m - 1$ **else**
    **if** $P_m(a) = P_n(b)$ **then**
        **if** $N_m(a) \geqslant N_n(b)$ **then** $i \leftarrow m - 1$ **end if**
        **if** $N_m(a) \leqslant N_n(b)$ **then** $j \leftarrow n - 1$ **end if**
    **end if**
**end while**
```

- **Split:** Splits a community into its two components, reversing the latest merge in the community dendogram. For instance, splitting community vertex A in Figure 3 would be accomplished by subtracting the adjacency of vertex B from that of A and recursively reverting the parent stack of all vertices in community vertex 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** (*EE*): Execute *Edit* for edge *ab* keeping the community structure intact
- **Split to Separation** (*Sep*): Recursively *Split* the community vertex containing edge *ab* until vertices *a* and *b* belong to separate communities, followed by *Edit*
- **Split to Singletons** (*Sing*): Recursively *Split* the community vertices containing vertices *a* and *b*, until *a* and *b* both become singleton communities, followed by *Edit*

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 of 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:**
   - 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 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:**
   - 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 a 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. Thus can be 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 clustering $\mathcal{C}'$, 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 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) Vertex Spanning: This approach merges positive Differential Modularity edges along an expanding frontier of active community vertices.

- Starting from a vertex i, the algorithm merges the vertex with a neighbor j, such that $\Delta Q_{ij} > 0$

- In order to maintain consistent community tagging, the merged community vertex is tagged i if $size(i) > size(j)$ and j otherwise.
- This continues until i becomes a sub-community of a neighbor i'. The process is then repeated for vertex 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 and treats all positive ΔQ 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:

- Starting from the existing clustering $\mathcal{C}'$, the algorithm identifies a maximum weight greedy matching of the edges based on Differential Modularity.
- 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.

V. Results

To compare the backtracking and agglomeration strategies, we evaluate them on the (SNAP) Facebook graph [18] (G_{fb}) with 4039 vertices and 88234 edges and the PGP Giant Component graph [19] (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 0 becomes vertex n, 1 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

Edge change	BT1	BT2	dGlobal BT [2]
Inter-community Insertion	EE	EE	Sing
Inter-community Deletion	EE	EE	Nothing
Intra-community Insertion	EE	Sep	Sep
Intra-community Deletion	Sep	Sep	Sing

TABLE II: Backtracking Strategies

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, Matching (*M*) and Vertex Span (*VS*) and two backtracking strategies, *BT1* and *BT2* detailed in Table II. 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. Additionally, not updating induced edges necessitates re-computation of adjacency after every batch change in case of dGlobal BT.

C. Analysis

The larger average degree of G_fb (43.7) compared to G_{pgp} (4.55) means that the LGS tests are able to re-wire fewer vertices per batch in G_{fb} than in G_{pgp}. 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 III 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 Vertex Spanning techniques perform worse off compared to Matching, with Static Vertex Spanning performing the worst. Dynamic Vertex 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 Vertex 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 IV, 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

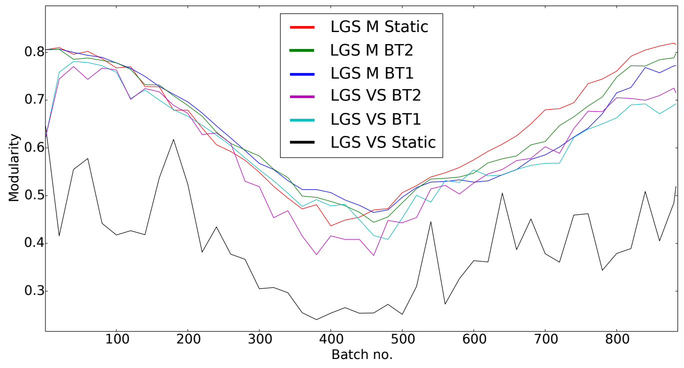


Fig. 4: G_{fb}: Moduarity evolution for LGS

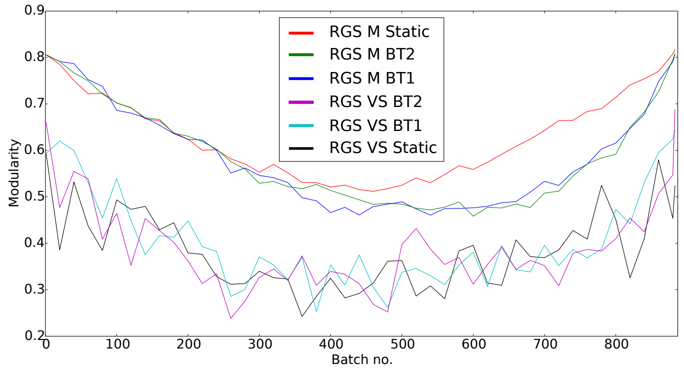


Fig. 5: G_{fb}: Modularity evolution for RGS

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}.

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 as seen in Table V, with *BT1* affording lower values of than *BT2*, due to higher backtracking in the latter. This advantage however is largely lost in the Vertex Spanning approach with Size of Change values for Dynamic approaches comparable to Static.

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

TABLE III: 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 IV: 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
VS Static	559	459	1161	1135
VS BT2	127	526	1080	1143
VS BT1	76	486	1032	1144

TABLE V: Average Size of Change per batch

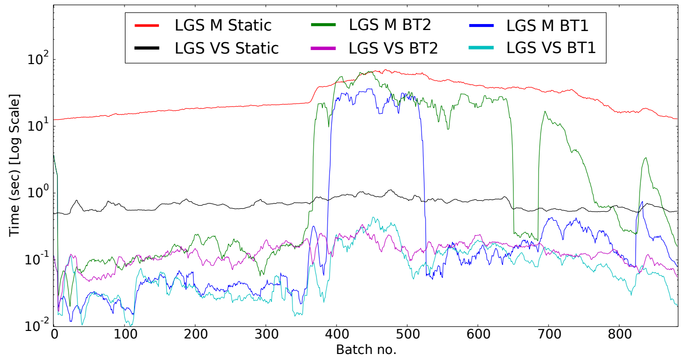


Fig. 6: G_{fb}: Execution time/batch for Localized Graph Stream

This is owing to the fact that Vertex Spanning orders the merges based on an expanding frontier of vertices, notwithstanding of the values of Differential Modularity, thus potentially following a significantly different merge order in affected regions of the graph across successive agglomerations. A notable saving may therefore be observed only in cases where localized regions of the graph are modified, thus largely retaining the adjacency and hence the merge order in the remaining graph, as can be expected in case of LGS_{fb}. Indeed the values of Size of Change for $VSBT1$ and $VSBT2$ 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 advantage of dynamic agglomeration becomes evident in the time saving over the static approach, as seen in Table VI. In all cases, the trend for execution time follows the order: *M Static* $>$ *M BT2* $>$ *M BT1* $>$ *VS Static* $>$ *VS BT2* $>$ *VS BT1*, as may be expected from the computation cost. The Vertex 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 while remaining computationally less intensive in nominal conditions.

Thus we observe that a trade-offs exists between various

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
VS Static	0.817	0.998	1.389	1.422
VS BT2	0.135	0.476	0.577	0.737
VS BT1	0.09	0.469	0.537	0.729

TABLE VI: Average execution time per batch (seconds)

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 Vertex 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 relaxed backtracking strategies.

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 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} [18] and G_{pgp} [19] confirms that the dynamic agglomeration approaches perform on 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 economical agglomeration strategies such as Matching and Vertex Spanning; and relaxed backtracking strategies with respect to conventional approaches, perform very well under dynamic clustering.

While the Matching based agglomeration approach performs better than Vertex 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 [5] 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.

Acknowledgment

The work depicted in this paper was partially sponsored by NSF Grant ACI-1339745 (XScala). The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of NSF or the U.S. Government, no official endorsement should be inferred.

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.

[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.

[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] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering," *10th DIMACS Implementation Challenge Workshop*, February 2012.

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

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

[10] 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

[11] 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

[12] 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

[13] 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

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

[15] 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.

[16] 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

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

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

[19] M. Bogua, R. Pastor-Satorras, A. Diaz-Guilera, and A. Arenas, "PGP giant component user network graph Physical Review E, vol. 70," 2004.