# Extracting Analyzing and Visualizing Triangle K-Core Motifs within Networks

Yang Zhang [#1], Srinivasan Parthasarathy [#2]

[#]*Department of Computer Science, Ohio State University, OH, USA*
[1]zhang.863@osu.edu
[2]srini@cse.ohio-state.edu

***Abstract*— Cliques are topological structures that usually provide important information for understanding the structure of a graph or network. However, detecting and extracting cliques efficiently is known to be very hard. In this paper, we define and introduce the notion of a *Triangle K-Core*, a simpler topological structure and one that is more tractable and can moreover be used as a proxy for probing and extracting relevant clique-like structure from large dynamic graphs and networks. Based on this definition we first develop a localized algorithm for extracting Triangle K-Cores from large graphs. Subsequently we extend the simple algorithm to accommodate dynamic graphs (where edges can be dynamically added and deleted). Finally, we extend the basic definition to support various template pattern cliques with applications to network visualization and event detection on graphs and networks. Our empirical results reveal the efficiency and efficacy of the proposed methods on many real world datasets.**

## I. Introduction

Many real world problems can be modeled as complex entity-relationship networks where nodes represent entities of interest and edges mimic the relationships among them. Fueled by technological advances and inspired by empirical analysis, the number of such problems and the diversity of domains from which they arise – physics, sociology, technology, biology, chemistry, metabolism and nutrition – is growing steadily. The study of such networks can help us understand the structure and function of such systems, potentially allowing one to predict interesting aspects of their behavior.

Of particular interest in many of these applications, is to probe, uncover, extract and understand the *dense* (clique-like) structure of interactions of such networks. This problem is very challenging. The first challenge relates to the characteristics of the data (scale-free nature, presence of hub nodes) as well as the scale and size of the data. The second challenge relates to the dynamic nature of the data – changes to the network – often requiring re-computation from scratch which can be very expensive. Identifying the portions of the network that are changing, characterizing the type of change, predicting future events (e.g. link prediction), and developing generic models for evolving networks are critical. The third challenge relates to visualization and confirmation. Fundamental to most data analysis is visual confirmation – from Galileo seeing the moons of Jupiter to Gerd Binnig and Heinrich Rohrer seeing atoms on a surface. Visualizing such complex networks and honing in on important topological characteristics is difficult, given the size and complexity of such systems, but nonetheless important.

In this article we attack a small region of this problem space. Specifically, we develop a scalable visual-analytic framework, for probing and uncovering dense substructures within networks. Central to our approach is the notion of a *Triangle K-Core motif*. We develop a simple algorithm for computing Triangle K-Cores from graphs. We then discuss a mechanism to plot such Triangle K-Cores – essentially realizing a density plot in a manner analogous to a CSV plot[1]. This plot follows an Optics[2]-style enumeration of vertices in the network. The complexity of this algorithm is linear in the number of triangles in the graph (so it is very fast for sparse graphs). In fact as our experimental results show, we produce plots that are very similar to CSV at a fraction of the cost.

Subsequently we extend the above static algorithm to handle dynamic graphs. Additionally, a key challenge addressed here is that of correspondence – the same community in two different density plots must be clearly identified as long as the local relationship structure has not changed significantly. We develop a suitable incremental algorithm, with cognitive correspondence (by relying on an adaptation of dual-view plots), which we show to be significantly faster than the naive approach which recomputes Triangle K-Cores from scratch.

An additional feature of our algorithms is the ability for the user to dynamically specify, explore and probe the network for various template patterns defined upon the base Triangle K-Core pattern. Such template patterns are extremely informative for identifying patterns of interest to the user. We design and adapt our density plot framework (here density is defined by the density of the template pattern of interest) based on this notion and discuss possible application of this work on several real world datasets. To sum up, the main contribution of our work is to introduce a new motif for estimating clique like structure in graphs (Triangle K-Core). Specifically in this article we demonstrate its:

1) **Utility:** We demonstrate its use for visualization (in a manner similar to a CSV plot), probing, exploring and highlighting interesting patterns in both static as well as dynamic graphs. We compare its utility with respect to recent state of the art alternative (e.g. CSV[1] and DN-Graph[3] motifs.
2) **Efficiency:** We present a localized algorithm for extracting such motifs and demonstrate its efficiency by several

factors over competing strategies such as DN-Graph[3] and CSV[1]. Additionally, we present an incremental variant that can be extended to handle dynamic graphs with much lower cost than the iterative method[3] and global method[1] used by extant approaches.

3) **Flexibility:** An important feature of the Triangle K-Core motif is its inherent simplicity which lends itself to flexible probing of user- defined template clique patterns of interest within both static and dynamic graphs.

## II. Related Work

In the context of graph clustering, several methods have often found favor. For example, spectral methods[4], stochastic flow methods[5], multi-level methods[6], [7] have all been used for discovering dense subgraphs of interest. While several of these algorithms scale well to large datasets they do not precisely target the problem of detecting clique-like structures.

In spite of the fact that CLIQUE problem is NP-Hard[8], and approximating the size of the largest clique in a graph is almost NP-complete[9], mining cliques for a graph has received much attention recently. The CLAN method [10] for example, aims to mine exact cliques in large graph datasets, CLAN uses the canonical form to represent a clique, and the clique detection task becomes mining strings representing cliques. Some other methods[11], [12] have been proposed to detect quasi-clique, which is a clique with some edges missing. Wang et al.[1] propose CSV to visualize approximate cliques. CSV uses a notion of local density, co-clique size, and plots all vertices based on co-clique sizes. The plot is a OPTICS [2] style plot, and visualizes the distribution of all the potential cliques. However, calculating co-clique size in CSV is still fairly expensive and makes CSV costly on large scale graphs. Other clique-like dense subgraph patterns, such as DN-graph[3], are also expensive to compute.

Many methods have been proposed to analyze dynamically changing graphs. Leskovec et al.[13] study the topological properties of some evolving real-world graphs, and propose "forest fire" spreading process including these properties. Backstrom et al.[14] study the relation between the evolution of communities and the structure of the underlying social networks. Asur et al.[15] define several events based on graph clusters evolution, and analyze group behavior through these events. Sun et al.[16] present a non-user-defined parameters approach to cluster evolving graphs based on Minimum Description Length principle. Lin et al.[17] propose FacetNet framework to detect community structure both by the network data and the historic community evolution patterns.

Graph visualization is often helpful for providing important insights of graph datasets. Namata et al.[18] develop a dual-view approach to provide multiple views of a network simultaneously. Yang et al.[19] propose a Visual-Analytic Toolkit to help analyze behavioral properties of nodes and communities, such as stability and influence. Abello et al.[20] propose a graph visualization system which uses clustering to construct a hierarchy of large scale graphs.

## III. Preliminaries

Given a graph $G = \{V,E\}$, $V$ is the set of distinct vertices $\{v_1, ..., v_{|V|}\}$, and $E$ is the set of edges $\{e_1, ..., e_{|E|}\}$. A graph $G' = \{V',E'\}$ is a subgraph of $G$ if $V' \subseteq V$, $E' \subseteq E$.

The Triangle K-Core subgraph proposed in this paper is derived from K-Core subgraph, and we explain and compare them as follows.

*Definition 1:* A **K-Core** is a subgraph $G'$ of $G$ that each vertex of $G'$ participates in at least k edges within the subgraph $G'$. The **K-Core number** of such a subgraph is k.

*Definition 2:* The **maximum K-Core** associated with a vertex $v$ is defined by the subgraph $G_v$ containing $v$ whose K-Core number is the maximum from among all subgraphs containing $v$. The K-Core number of $G_v$ is the **maximum K-Core number** of $v$.

Batagelj et al [21] propose an efficient method to compute every vertex's maximum K-Core number with O($|E|$) time complexity.

Based on definition of K-Core, we are now in a position to define the notion of a Triangle K-Core:

*Definition 3:* A **Triangle K-Core** is a subgraph $G'$ of $G$ that each edge of $G'$ is contained within at least k triangles in the subgraph. Analogously, the **Triangle K-Core number** of this Triangle K-Core is refered to as k.

*Definition 4:* The **maximum Triangle K-Core** associated with an edge $e$ is the subgraph $G_e$ containing $e$ that has the maximum Triangle K-Core number. Analogously, the Triangle K-Core number of $G_e$ is the **maximum Triangle K-Core number** of edge $e$. We use $\kappa(e)$ to denote the maximum Triangle K-Core number of edge $e$.

The main advantage of a Triangle K-Core over a K-Core is that it offers a natural approximation of clique, we illustrate this in the Figure 1.

(a) K-Core Number = 2 (b) Triangle K-Core Number = 2

Fig. 1. K-Core vs. Triangle K-Core

Figure 1(a) is a 5-vertex K-Core with K-Core number 2 constructed by minimal number of edges, Figure 1(b) is a 5-vertex Triangle K-Core with Triangle K-Core number 2 constructed by minimal number of edges, and we can easily see that the Triangle K-Core is much closer to a 5-vertex clique than the K-Core. In fact, Triangle K-Core is a relaxation of clique, a *n*-vertex clique is equivalent to a *n*-vertex Triangle K-Core with Triangle K-Core Number *n-2*.

For edge $e_t$, there is a triangle *T* containing $e_t$ in $e_t$'s maximum Triangle K-Core, we have the following property for *T*:

*Theorem 1:* If triangle *T* is in $e_t$'s maximum Triangle K-Core, and contains three edges, $e_t$, $e_1$ and $e_2$, then $\kappa(e_i) \geq \kappa(e_t)$ (i = 1,2).

*Proof:* Since edge $e_i$ is in triangle $T$, and $T$ is in $e_t$'s maximum Triangle K-Core, denoted as $G_{e_t}$, we have subgraph $G_{e_t}$ contains $e_i$. According to Definition 4, $e_i$'s maximum Triangle K-Core should have Triangle K-Core number no less than $G_{e_t}$'s Triangle K-Core number, that is $\kappa(e_i) \geq \kappa(e_t)$. ■

## IV. Triangle K-Core Algorithm

### A. Detecting Maximum Triangle K-Core

In Algorithm 1, input is Graph $G$, output is the maximum Triangle K-Core number and optionally the maximum Triangle K-Core associated with each edge. In each iteration, this algorithm processes a particular edge $e_i$ and determines its maximum Triangle K-Core number.

**Algorithm 1** Detect each edge's maximum Triangle K-Core

```
1: for each edge e in the graph do
2:     set e to be unprocessed;
3:     find all the triangles on e, set them to be unprocessed;
4:     for each triangle t on edge e do
5:         AddToCore(t, e);
6:         κ̃(e) + +;
7: Place all the edges in list Edges, sort them in increasing order
   of κ̃ value;
8: for i = 0 to |E|−1 do
9:     e_i = Edges[i];
10:    κ(e_i) = κ̃(e_i);
11:    for each unprocessed triangle T on e_i do
12:        for each edge e_t other than e_i in T do
13:            if κ̃(e_t) > κ̃(e_i) then
14:                DelFromCore(T, e_t);
15:                κ̃(e_t) − −;
16:                update e_t's position in the sorted list Edges;
17:        set triangle T to be processed;
18:    set e_i to be processed;
```

Before describing the Algorithm 1 we define the notions of processing an edge and a triangle. If an edge's maximum Triangle K-Core number has been determined, it is considered to be *processed*. A triangle $T$ is *processed* if *any one* of its edges is *processed*.

In step 2, each edge is set to *unprocessed*. In step 3, each triangle on edge $e$ is constructed by $e$'s two vertices and one common neighbor of them. One triangle could be constructed three times by its three edges, but we only store one instance of each triangle. [1]

Note that all triangles on edge $e$ could be in $e$*'s* maximum Triangle K-Core, so in step 5 the algorithm (*AddToCore*), updates its bookkeeping to reflect the fact that each triangle $t$ is possibly[2] in $e$'s maximum Triangle K-Core. Finally, $\tilde{\kappa}(e)$ contains the upper bound of $e$'s maximum Triangle K-Core number $\kappa(e)$.

In step 7 we place all the edges in a list sorted by increasing order of $\tilde{\kappa}$ value. Bucket sort can be used as an optimization step here with time complexity O($|E|$). In steps 8-18, we process each edge $e_i$ and determine its exact maximum Triangle K-Core number $\kappa(e_i)$ since thus far we only had an upper bound. In step 10, we determine that $\kappa(e_i)$ is exactly $\tilde{\kappa}(e_i)$, the correctness is proved later. Then we update $e_i$'s neighbor edges' $\tilde{\kappa}$ value in steps 11-17. If an *unprocessed* triangle $T$ on $e_i$ contains edge $e_t$ that $\tilde{\kappa}(e_t)$ is greater than $\tilde{\kappa}(e_i)$ (step 13), we delete $T$ from the upperbound of $e_t$'s maximum Triangle K-Core. *DelFromCore* updates its bookkeeping to indicate that $T$ is *not* in the upperbound of $e_t$'s maximum Triangle K-Core. In step 16, based on bucket sort the update could be optimized with complexity O(1).

In fact, steps 5 and 14 are not necessary here, but it will be useful for dynamic update algorithms. The time complexity for Steps 8-18 is O($|Tri|$), where $|Tri|$ is the total number of triangles in the graph.

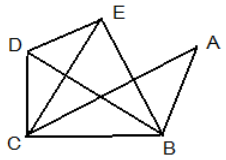



Fig. 2. Example of Algorithm 1

**Example:** Figure 2 is an example to illustrate Algorithm 1. We find the triangles on each edge, and sort edges in increasing order of $\tilde{\kappa}$ value, {AB(1), AC(1), BD(2), BE(2), CD(2), CE(2), DE(2), BC(3)}, where the number in parenthesis indicates the $\tilde{\kappa}$ value of the edge. We process AB first, and get $\kappa(AB)$=1. For *unprocessed* $\triangle ABC$ on AB, $\tilde{\kappa}(BC)$=3 is greater than $\tilde{\kappa}(AB)$=1, so $\tilde{\kappa}(BC)$ decrease 1 to be 2 (step 15), and $\triangle ABC$ becomes *processed*. Then we process edge AC, and have $\kappa(AC)$=1, there is no *unprocessed* triangle on AC, so no update is needed. Next we process edge BD, and get $\kappa(BD)$=2, $\triangle BDC$ and $\triangle BDE$ on BD are *unprocessed*, but no edge of the two triangles has greater $\tilde{\kappa}$ value than $\tilde{\kappa}(BD)$, so no update. In the same way we find all left edges having $\kappa$ value equals 2.

**Proof of Correctness of Algorithm 1:** We show the following invariances of Algorithm 1: at the end of each iteration $i$, (1)for the edge $e_t$ whose $\tilde{\kappa}(e_t)$ value updated, $\tilde{\kappa}(e_t)$ is still the upperbound of $\kappa(e_t)$; (2) for the edge $e_i$ processed in current iteration, $\tilde{\kappa}(e_i)$ is equal to $\kappa(e_i)$.

We firstly prove the invariance (1) of Algorithm 1. In steps 11-12, for an *unprocessed* triangle $T$ on edge $e_i$, all $T$'s edges are *unprocessed*, so $T$ is still in the upperbound of maximum Triangle K-Cores of all its edges(including edge $e_i$ and $e_t$). If $\tilde{\kappa}(e_t) > \tilde{\kappa}(e_i)$ (step 13), we have:

*Claim 1:* $\tilde{\kappa}(e_t) > \kappa(e_t)$

*Proof:* We prove by contradiction. Assume $\tilde{\kappa}(e_t) = \kappa(e_t)$, then all the triangles in the current upper bound of $e_t$'s maximum Triangle K-Core are exactly in $e_t$'s maximum Triangle K-Core, so $T$ is in $e_t$'s maximum Triangle K-Core. However, in triangle $T$, $\kappa(e_t) = \tilde{\kappa}(e_t) > \tilde{\kappa}(e_i) >= \kappa(e_i)$, which violates Theorem 1, so the assumption is incorrect. We have $\tilde{\kappa}(e_t) > \kappa(e_t)$. ■

According to the proof of Claim 1, after decreasing $\tilde{\kappa}(e_t)$ by

[1] We ensure this by giving a unique id to each edge, and only creating a triangle instance on its edge with smallest id.

[2] The term possibly reflects the fact that this is currently an upper bound.

1 (step 15), $\tilde{\kappa}(e_t)$ still remains as the upper bound of $\kappa(e_t)$. So invariance (1) is held.

Now we prove invariance (2). In iteration i, assume $\tilde{\kappa}(e_i)$ = k, we use the edges whose current $\tilde{\kappa} \geq k$ to construct a subgraph $G_k$ (including $e_i$), and have the following claim:

*Claim 2:* The subgraph $G_k$ is a Triangle K-Core with Triangle K-Core number k.

*Proof:* For any edge *e* in $G_k$, $\tilde{\kappa}(e) \geq k$, so the upper bound of *e*'s maximum Triangle K-Core now contains at least k triangles. Assume triangle *T* is one of them, considering *T*'s two other edges *e1* and *e2*, if *e1* is not in subgraph $G_k$, then $\tilde{\kappa}(e1) < k$. We could see that Algorithm 1 processes edges in increasing order of $\tilde{\kappa}$, so *e1* should already be processed. When processing $e1$, $\tilde{\kappa}(e1) < \tilde{\kappa}(e)$ (step 13) is true, so triangle *T* should be deleted from the upper bound of *e*'s maximum Triangle K-Core (step 14), which is a contradiction to the assumption that triangle *T* is in upper bound of *e*'s maximum Triangle K-Core. So *e1* is in subgraph $G_k$, and so is *e2*. Because edges *e*, *e1* and *e2* are all in subgraph $G_k$, triangle *T* is in $G_k$. So all the triangles now in upper bound of *e*'s maximum Triangle K-Core are in subgraph $G_k$, which means any *e* in $G_k$ is contained in at least k triangles in $G_k$, so $G_k$ is a Triangle K-Core with Triangle K-Core number k. ■

In Claim 2, we have a subgraph $G_k$ containing $e_i$ with Triangle K-Core number equals $\tilde{\kappa}(e_i)$, so $\tilde{\kappa}(e_i)$ is exactly $\kappa(e_i)$, and $G_k$ is obviously the maximum Triangle K-Core of $e_i$.

In steps 3 we could store all triangles in main memory, then reuse them in step 11. However for a large graph, storing all triangles in main memory might be impossible. In such a case, we do not store triangles in step 3, and compute each edge's triangles again in step 11, then we test whether a triangle is *unprocessed* by testing whether its three edges are all *unprocessed*.

### *B. Updating Maximum Triangle K-Core*

So far we have worked on static graphs. In scenarios when edges are added and removed from a graph over time however, rather than recomputing the Triangle K-Cores from scratch after each change, we can use Algorithm 2 to efficiently update edges' maximum Triangle K-Cores. The detailed pseudo code of Algorithm 2 is in Appendix (Section IX-A).

Adding/deleting one edge might add/delete multiple triangles simultaneously, in Algorithm 2 we process added/deleted triangles one by one (step 1). Initially all added/deleted triangles are *not updated*, and when processing one triangle *T* we set it to be *updated* (step 2). In step 3, we identify *T's* edges whose maximum Triangle K-Cores might change, and store them in *PotentialList*. We use Rule 0 to help find the edges whose maximum Triangle K-Cores might change. Rule 0 is derived from Theorem 1, the proof is omitted for brevity.

- Rule 0: when triangle *t* is added/deleted to graph $G$, assume $\mu$ is smallest $\kappa$ value of *t*'s three edges, then only the edges in $G$ whose $\kappa$ value equals $\mu$ might have their maximum Triangle K-Cores changed.

Then we process each edge *e* in *PotentialList* to update its $\kappa(e)$. All the triangles associated with edge *e* should obey Theorem 1, so we process them based on Theorem 1 (steps 6-7). If $\kappa(e)$ finally changes, we put *e* in *ChangingList*, which stores edges whose $\kappa(e)$ has been changed, and put *e's* neighbor edges whose maximum Triangle K-Cores might change to *PotentialList*(step 8). We use Rule 0 to help select the edges to be put in *PotentialList*. After processing all edges in *PotentialList*, we could determine edges' maximum Triangle K-Core numbers in *ChangingList*(step 9).

Please note that if an added triangle is *not updated*, or a deleted triangle is *updated*, we do not involve them in the Algorithm 2. A brief illustration of Algorithm 2 is as follows.

**Algorithm 2** Update maximum Triangle K-Cores

1: **for** each added/deleted triangle *T* **do**
2: Set *T* to be *updated*;
3: Put *T*'s edges whose maximum Triangle K-Cores might change to *PotentialList*;
4: Add/delete *T* from the maximum Triangle K-Cores of edges in *PotentialList*, update those edges' $\kappa$ value;
5: **for** each edge *e* in *PotentialList* **do**
6: Find *e*'s "illegal" triangles that violate Theorem 1;
7: Process *e*'s "illegal" triangles to obey Theorem 1, meanwhile update $\kappa(e)$;
8: If $\kappa(e)$ changes, put *e* in *ChangingList*, put *e*'s neighbor edges whose maximum Triangle K-Cores might change to *PotentialList*;
9: update $\kappa(e)$ of each edge *e* in *ChangingList*;

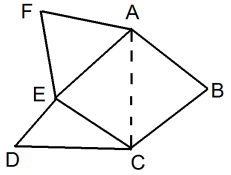



Fig. 3. Example of Algorithm 2

**Example:** In Figure 3, the original graph is comprised with solid edges, and edge AC is added. The original $\kappa$ value for each edge is {AB(0), BC(0), AE(1), AF(1), EF(1), CD(1), CE(1), DE(1)}. The initial value for $\kappa(AC)$ is 0. After adding edge AC, two triangles are added, $\triangle ABC$ and $\triangle AEC$.

Firstly, we process newly added $\triangle ABC$, now all its three edges are {AB(0), BC(0), AC(0)}, so we put all three edges in *PotentialList* (Rule 0), and add $\triangle ABC$ to their maximum Triangle K-Cores (step 4), their $\kappa$ value increases to be 1. Then we process each edge in *PotentialList*, assume AC is the first edge. In step 6 we find $\triangle ABC$ on edge AC is "legal", and $\triangle AEC$ is not taken into consideration because it is *not updated*. In step 8, because $\kappa(AC)$ changes to be 1, we put edge AC's neighbor edges AB, BC in *PotentialList*(they are already in). In the following iterations we process left edges in *PotentialList* (AB and BC) similarly, and update $\kappa(AB)$ and $\kappa(BC)$ to be 1.

Then, we process newly added $\triangle AEC$, now its three edges are {AE(1), EC(1), AC(1)}, so we put all of them in *PotentialList*, and add $\triangle AEC$ to their maximum Triangle K-Cores,

their $\kappa$ value increases to be 2. Let's process edge AC first, we find $\triangle ABC$ on edge AC is "illegal", because $\triangle ABC$ is in AC's maximum Triangle K-Core while $\kappa(AC) = 2$ is greater than $\kappa(BC) = 1$ and $\kappa(AB) = 1$, which violates Theorem 1. So in step 7 we delete $\triangle ABC$ from AC's maximum Triangle K-Core and decrease $\kappa(AC)$ to be 1. Similarly edges AE and EC in *PotentialList* both are processed to decrease $\kappa(AE)$ and $\kappa(EC)$ to be 1.

If we do not store triangles in Algorithm 1, then in Algorithm 2 we need to recompute triangles from edges, we explain this in Appendix (Section IX-A).

## V. Extensions

**Visualizing Clique-like Structures:** We now describe how Triangle K-Cores can be used for detecting and visualizing interesting cliques-like structures within networks. Before describing our technique we briefly review the CSV method [1] to visualize all potential cliques in graph.

*CSV plot*: CSV first estimates *co_clique_size* for each edge, which is the size of maximum clique that each edge participants in. Then subsequently CSV plots vertices along X-axis in a certain order, and the Y-axis value for each vertex is one of its neighbor edges' *co_clique_size* value. The final plot is the clique distribution of the graph, and the flat peaks in the plot indicate potential cliques.

However, estimating *co_clique_size* for each edge takes up most of the time cost in CSV. Instead we propose to use each edge's maximum Triangle K-Core as a proxy to approximate the maximum clique it participates in. In other words we estimate *e.co_clique_size* as $\kappa(e)+2$ for each edge *e*, and then plot the clique distribution using exactly the same method as that of CSV. As we demonstrate in experiments our method produces plots that are *inherently similar or identical to that of CSV at a fraction of the cost*.

**Dual View Plots:** When edges are added to a dynamic graph *G*, some clique structures might be changed, we propose to use *Dual View Plots* to capture the change of cliques in the graph.

The idea is, we first plot the cliques of the original graph in plot(a), then after adding edges, we only plot the changed cliques in plot(b). By comparing plot(a) and plot(b), we could visually analyze how cliques in plot(b) are formed from cliques in plot(a). We use the the same plot method as CSV to plot clique distribution. The steps are as follows:

We illustrate the benefits of *Dual View Plots* in the experimental section.

**Detecting Template Pattern Cliques:** In this section we describe a mechanism wherein allows users to detect cliques of patterns to their interest. In evolving graphs, these cliques could include, for example, cliques formed by a group of new actors (nodes), cliques formed by merging two existing cliques etc. At a higher level of abstraction, such cliques can allow a user to probe an evolving network to discover interesting or anomalous behavior[22]. The benefit of our method is it gives users flexibility to define the patterns on their own.

**Algorithm 3** Dual View Plots

1: Execute Algorithm 1 to compute $\kappa(e)$ for each edge *e*;
2: For each edge *e*, $e.co\_clique\_size = \kappa(e) + 2$;
3: Plot clique distribution plot(a) of *G*;
4: When edges are added, execute Algorithm 2 to update edges' $\kappa$ value;
5: Recompute $co\_clique\_size$ for each edge *e*, if *e* is newly added edge, $e.co\_clique\_size = \kappa(e) + 2$, otherwise $e.co\_clique\_size = 0$;
6: Plot clique distribution plot(b) based on new *co_clique_size* of each edge;
7: In plot(b) select one Clique *C* of interest, locate the vertices of *C* in plot(a), and analyze how Clique *C* is formed;

Several examples of template pattern cliques in evolving graphs are illustrated below and in Figure 4. Here the original graph is denoted as OG, the new graph is denoted as NG. In Figure 4 black vertices/edges are original vertices/edges, red vertices/edges are newly added vertices/edges.

1. New Form Clique is formed by new edges, as illustrated in Figure 4(a) where ABCDE is a New Form Clique.

2. Bridge Clique is formed by vertices from two disconnected cliques in OG, as illustrated in Figure 4(b) where ABCDE is a Bridge Clique.

3. New Join Clique is formed by a clique in OG and new vertices in NG, as illustrated in Figure 4(c) where ABCDEF is a New Join Clique.

**Algorithm 4** Detecting template pattern Cliques in Graph $G$

1: Define and detect the characteristic triangles of the template pattern cliques;
2: **for** each characteristic triangle $T_c$ **do**
3: Mark $T_c$'s edges and vertices as *special*;
4: Define and detect the possible triangles formed by *special* vertices;
5: **for** each possible triangle $T_p$ **do**
6: Mark $T_p$'s edges as *special*;
7: Use the *special* vertices and *special* edges to build a new graph $G_{spe}$;
8: Execute Algorithm 1 on $G_{spe}$ to calculate each *special* edge's $\kappa$ value;
9: **for** each edge $e$ in $G$ **do**
10: **if** $e$ is a *speical* edge **then**
11: *e.co_clique_size* = $\kappa(e)$+2;
12: **else**
13: *e.co_clique_size* = 0;
14: Use the same plot method as CSV to plot clique distribution of graph $G$;

We propose Algorithm 4 to detect the template pattern cliques. In step 1, we introduce the concept "characteristic triangles" of the template pattern cliques. The characteristic triangles should satisfy the following two requirements:

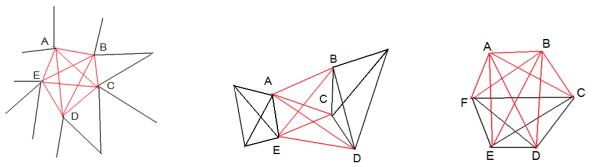
(1) a characteristic triangle is a 3-vertex clique of such template pattern.

(2) for any vertex $v$ in the template pattern clique, there is a characteristic triangle containing $v$ in the same clique.

From the requirement 1, we know that every vertex in a

characteristic triangle belongs to the template pattern clique; from the requirement 2, we know that any vertex of such a template pattern clique belongs to a characteristic triangle. So the vertices of all characteristic triangles are exactly the vertices of all such template pattern cliques. We should note that if users are unable to define characteristic triangle for a template pattern clique, Algorithm 4 could not be used.

In steps 2-3 we mark all characteristic triangles and their edges and vertices to be *special*. Also, we notice that besides characteristic triangles, some other types of triangles are also possible in template pattern cliques, so we define and detect all these possible triangles of the template pattern cliques in step 4, note that the vertices of possible triangles are among the the vertices of characteristic triangles. In steps 5-6 we mark all possible triangles and their edges as *special*. In step 7, we build a subgraph $G_{spe}$ made of *special* triangles, *special* edges and *special* vertices. In step 8 we detect cliques in the subgraph $G_{spe}$ using Algorithm 1. In steps 9-13 we compute *co_clique_size* for *special* edges, and set *co_clique_size* of *non-special* edges to be 0, because they do not participate in any template pattern cliques. Finally we plot the distribution of the template pattern cliques.




(a) New Form (b) Bridge (c) New Join

(d) New Form (e) Bridge (f) New Join

Fig. 4. Several template pattern cliques and their characteristic triangles

In the following we will detect the 3 template pattern cliques introduced before. We need to specify the characteristic triangles and possible triangles in Algorithm 4, and then detect the template pattern cliques using Algorithm 4.

**Detect New Form Cliques:** the characteristic triangle of a New Form Clique obviously has 3 new edges and 3 original vertices, as illustrated in Figure 4(d), and no other types of triangles are possible in New Form cliques.

**Detect Bridge Cliques:** for the characteristic triangle of a Bridge Clique, its vertices should be in two disconnected cliques in OG, so the triangle has 3 original vertices, 2 new edges, and 1 original edge, as illustrated in Figure 4(e). We find that there is another type of possible triangle in Bridge Clique, that is triangle comprised of 3 original edges, such as $\triangle BCD$ in the Figure 4(b).

**Detect New Join Cliques:** the characteristic triangle of a New Join Clique should be comprised of a clique in OG and new vertices, so the only possibility is that the triangle contains a new vertex, and two connected original vertices (2-vertex clique) in OG, as Figure 4(f) shows. There are two types of possible triangles in New Join Clique. One type is made of all new edges, such as $\triangle ABC$ in the Figure 4(c), and another type is made of all original edges, such as $\triangle DEF$ in the Figure 4(c).

Please note that Algorithm 4 not only works for evolving graphs, but also for static graphs. If edges and vertices have different attributes, we could mark different attributes as red or black as in Figure 4, and detect similar template pattern cliques using Algorithm 4. In Experiments section we will illustrate detecting template pattern cliques on static and dynamic graphs.

## VI. Relationship to DN-Graph

Before we discuss the empirical evaluation we would like to highlight an interesting connection between our approach and the recent approach proposed by Wang et al[3]. It is interesting to note that this connection was initially observed during our empirical evaluation, where we found both DN-Graph and our method to essentially converge to identical values of $co\_clique\_size$ (density). We are now in a position to also provide a theoretical justification for this connection.

DN-Graph *G'(V', E', λ)* is a subgraph pattern proposed by Nan Wang et al[3], it satisfies two requirements
(1) every connected pair of vertices in *G'* has at least $\lambda$ common neighbors; (2) for vertex *v not* in *G'*, adding *v* to *G'* will decrease the $\lambda$ value of *G'*, for vertex *v'* in *G'*, removing *v'* from *G* will *not* increase the $\lambda$ value of *G'*.

A subgraph with Triangle K-Core number $\lambda$ only satisfies requirement (1), so it is a relaxation of DN-Graph. Requirement (2) actually makes DN-Graph a locally maximum densest subgraph.

However, there are two problems with DN-graph: (1) Not every vertex in a graph belongs to a DN-Graph subgraph. In Figure 5, only BCDE is a DN-Graph, there is no DN-Graph

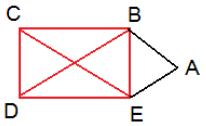



Fig. 5. DN-Graph example

covering vertex A, so DN-Graph might not be helpful if we want to get every vertex's local density. (2) Detecting all DN-Graphs in a graph is NP-Complete[3].

To tackle problem (2), Nan Wang et al.[3] propose to detect $\lambda(e)$, which is the maximum $\lambda$ value of the DN-Graph that edge *e* participates in. However, detecting $\lambda(e)$ is still difficult, so they propose to compute a *valid* upperbound of $\lambda(e)$, denoted as *valid* $\tilde{\lambda}(e)$, iteratively. Interestingly, we find that $\kappa(e)$ is actually *valid* $\tilde{\lambda}(e)$ (the proof is below).

*Definition 5: valid* $\tilde{\lambda}(e)$
Inside $\triangle(u, v, w)$, if $\tilde{\lambda}(u, v) \leq \min(\tilde{\lambda}(u, w), \tilde{\lambda}(v, w))$, we say $w$ supports $\tilde{\lambda}(u, v)$. $\tilde{\lambda}(u, v)$ is *valid* if and only if $|\{w|\ w$ supports $\tilde{\lambda}(u, v)\}| \geq \tilde{\lambda}(u, v)$.

*Claim 3:* For any edge *e*, $\kappa(e)$ is *valid* $\tilde{\lambda}(e)$.

*Proof:* Since the maximum Triangle K-Core of $e$ is a relaxation of the maximum DN-Graph containing $e$, $\kappa(e)$ is

upperbound of $\lambda(e)$, denoted as $\tilde{\lambda}(e)$. In graph G we assign $\tilde{\lambda}(e)$ as $\kappa(e)$ for every edge $e$.

Next we prove $\kappa(e)$ is *valid* $\tilde{\lambda}(e)$. For edge *e(u, v)*, assume its maximum Triangle K-Core is subgraph $G_e$. For any $\triangle(u, v, w)$ containing *e* in $G_e$, according to Theorem 1, we have $\kappa(v, w) \geq \kappa(e)$, $\kappa(u, w) \geq \kappa(e)$, so $\tilde{\lambda}(v, w) \geq \tilde{\lambda}(e)$ , $\tilde{\lambda}(u, w) \geq \tilde{\lambda}(e)$. According to Definition 5, vertex *w* supports $\tilde{\lambda}(e)$. There are at least $\kappa(e)$ triangles containing edge *e* in $G_e$, so there are at least $\kappa(e)$ vertices supporting $\tilde{\lambda}(e)$. $\tilde{\lambda}(e)$= $\kappa(e)$, therefore $\tilde{\lambda}(e)$ is *valid*, and $\kappa(e)$ which equals $\tilde{\lambda}(e)$ is *valid* $\tilde{\lambda}(e)$. ■

As a result of this connection we empirically observe that for almost all the real-world graphs we have evaluated both DN-Graph variants TriDN and BiTriDN essentially converge to identical density values as our method. The main computational advantage of our method is in avoiding the costly iterative approach to estimate $\lambda$ and to directly compute $\kappa(e)$.

## VII. Experiments

In this section we present our experimental results. All experiments, unless otherwise noted, are evaluated on a 3.2GHz CPU, 16G RAM Linux-based system at the Ohio Supercomputer Center (OSC). The main datasets we evaluated our results on can be found in Table I.

TABLE I
Data Sets

| Graph Dataset | Vertices | Edges |
|---|---|---|
| Synthetic | 60 | 308 |
| Stocks | 425 | 1680 |
| PPI | 4741 | 15147 |
| DBLP | 6445 | 11848 |
| Astro-Author | 17903 | 196972 |
| Epinions | 75879 | 405741 |
| Amazon | 262111 | 899792 |
| Wiki | 176265 | 1010204 |
| Flickr | 1,715,255 | 15,555,041 |
| LiveJournal | 4,847,571 | 42,851,237 |

### A. Comparison with CSV and DN-Graph

In our first set of experiments we compare the performance of our Triangle K-Core algorithm (Algorithm1) with other recent approaches such as CSV[1] and DN-Graph variants(TriDN and BiTriDN (an improvement over TriDN))[3] both in terms of efficiency and efficacy. As noted in Section VI we can theoretically show that the DN-Graph variants (TriDN and BiTriDN) proposed by Wang et al[3] converges to the same value as our algorithm. Table II documents the execution time performance of these algorithms on various datasets, while Figure 6 conveys a qualitative comparison by realizing the density plots produced by each algorithm (note that since DNGraph and Triangle K-Core converge to the same values the density plots are identical). First, for all the datasets it is clear that CSV is the slowest while Triangle K-Core is the fastest to finish (note for the three largest datasets we could not run CSV or TriDN due to memory thrashing issues on the OSC machine and BiTriDN was taking too long to converge[3]

TABLE II
Time Cost Comparison (seconds)

| Graph | CSV | TriDN | BiTriDN | T-K-Core |
|---|---|---|---|---|
| Synthetic | 0.043 | 0.0012 | 0.0011 | 0.0010 |
| Stocks | 0.51 | 0.017 | 0.010 | 0.006 |
| PPI | 2.51 | 0.211 | 0.121 | 0.097 |
| DBLP | 1.47 | 0.062 | 0.046 | 0.034 |
| Astro-Author | 17393.7 | 73.8 | 7.79 | 1.03 |
| Epinions | - | 262.13 | 15.71 | 4.09 |
| Amazon | - | 34.9 | 10.59 | 3.81 |
| Wiki | - | 435.8 | 17.15 | 7.89 |
| Flickr | - | - | *60 hours | 747 |
| LiveJournal | - | - | - | 443 |

Second, when comparing our results with CSV plots on the qualitative visual assesment (the interested reader is refered to Figure 6) we observe that while the order in which vertices are processed may on occasion be slightly different – arising due to the differences in the estimation procedure of co-clique size and resulting in a shift of the main trends – the main trends themselves are quite similar and easy to discern.

### B. Protein-Protein Interaction (PPI) Case Study

We also do a case study on PPI network, the plot is in Figure 7. The 3 red circles in the plot indicate 3 approximate cliques, we draw the 3 cliques (from left to right) in Figure 7. We find that clique 1 is exactly the same DN-Graph detected in [3]. The names in the parenthesis are the names used in [3]. Clique 2 is shown to be 10-vertex clique in the plot, in fact it is an exact 10-vertex clique. Clique 3 has 10 vertices, but it is shown to be 9-vertex clique, because the edge between APC4 and CDC16 is missed.

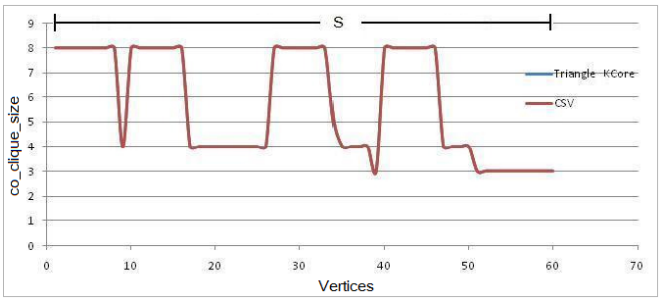
### C. Experimental Results of Update Algorithm

To evaluate the effectiveness of our update algorithm we randomly add/delete 1% of edges from the five largest datasets in Table I, and in Table III we compare the time costs of re-computing and updating the maximum Triangle K-Cores incrementally. Results reported are averaged over 5 runs. Here Re-compute time is actually the execution time of steps 8-18 in Algorithm 1, and Update time is the execution time of the Algorithm 2. The results clearly demonstrate that the incremental algorithm is effective.

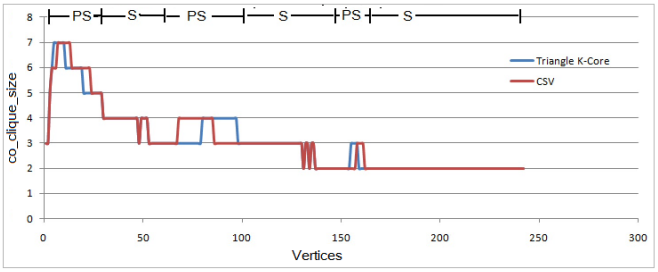
### D. Dual View Plots: Wiki Case Study

In Figure 8, we present an example to illustrate how *Dual View Plots* can highlight the change of clique-like structures within a dynamic graph setting.

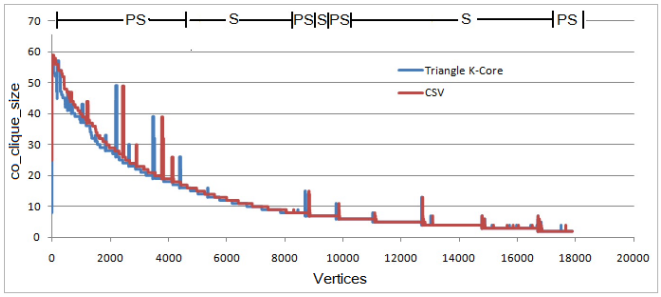
[3]We had a job time limit imposed by the Ohio Supercomputer Center job scheduler. The Flickr results for DNGraph are taken from[3], to give the reader a ballpark figure – the machine they used had a comparable processor but with larger memory. The reason for this high processing time for the DNGraph variants is attributed to the fact that each iteration is expensive (55 min per iteration for Flickr) and a number of iterations (66 are needed for convergence[3]).



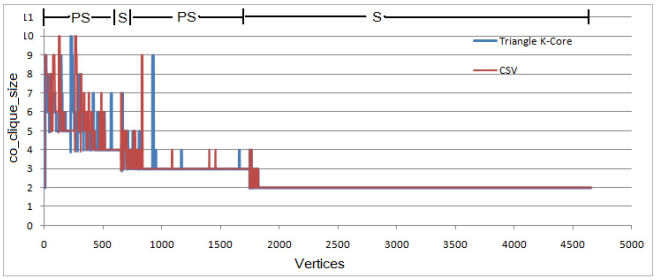

(a) Synthetic Dataset



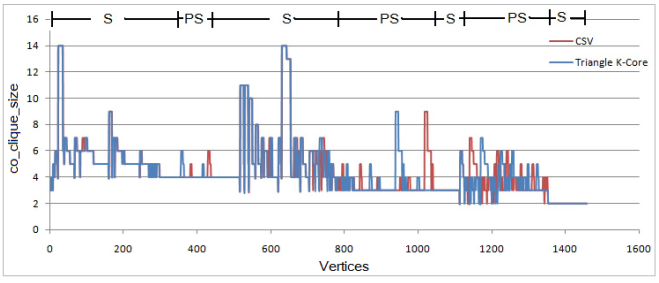

(b) Stocks Dataset




(c) Astro-Author Dataset

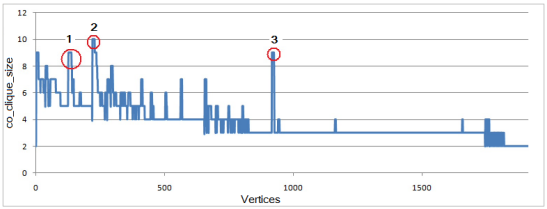



(d) PPI Dataset

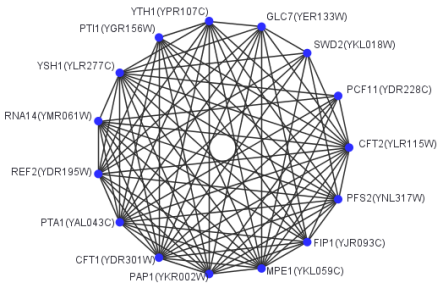



(e) DBLP Dataset

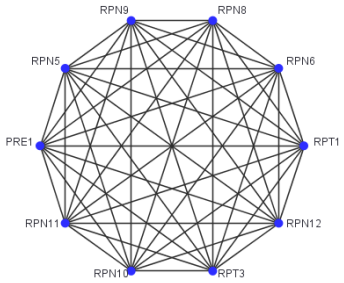
Fig. 6. Qualitative Comparison between CSV and Triangle K-Core Note that in the figure we note regions in the plot where the two plots are near identical or similar (S) and regions where there is a distinct phase shift (PS).



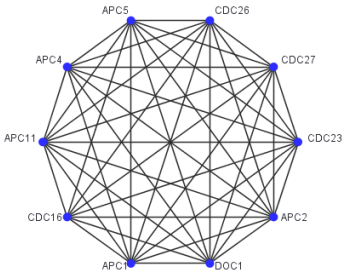

(a) PPI clique distribution




(b) PPI clique 1




(c) PPI clique 2




(d) PPI clique 3

Fig. 7. Cliques in PPI dataset

We use two consecutive snapshots of Wiki datasets for this purpose. A snapshot of Wiki dataset is comprised of vertices, which are Wiki articles, and references among them. Figure 8(a) represents the clique distribution plot of 1st snapshot, and it corresponds to plot(a) in Algorithm 3. Figure 8(b) visualizes the cliques containing new edges in the 2nd snapshot, and it corresponds to plot(b) in Algorithm 3. The users could use the tool to choose a subset of clique-like structures of interest to enable cognitive correspondence.

We illustrate this in Figure 8(b) by selecting the 3 cliques with highest density on the plot – denoted using a green

TABLE III
UPDATE ALGORITHM TIME COST (SECONDS)

| Graph | Total Edges | Edges Changed | Re-Compute | Update |
|---|---|---|---|---|
| Astro-Author | 196972 | 1814 | 0.27 | 0.005 |
| Epinions | 405741 | 3953 | 0.70 | 0.06 |
| Amazon | 899792 | 7958 | 0.61 | 0.01 |
| Flickr | 15,555,041 | 14996 | 561 | 1.4 |
| LiveJournal | 42,851,237 | 41996 | 306 | 2.4 |

triangle, a red rectangle, and an orange ellipse for expository convenience. The *Dual View Plot* tool can then locate their corresponding vertices in Figure 8(a) using the same markers, allowing the user to gain insights into how these clique-like structures evolved. For example, one can observe that the vertices (green triangle) are located in two places in Figure 8(a); some vertices are in a 10-vertex clique, and one single vertex is in a 5-vertex clique. Drilling down as shown in Figure 8(c), "Astrology" is the single vertex, the red edges are new added edges. Essentially between two consecutive snapshots, a new Wiki page and the corresponding Wiki links were established thereby forming a larger clique. The details about the other 2 clique-like structures are presented Figure 8(d) and Figure 8(e) and are also self explanatory – the two cliques are formed by merging vertices from different original cliques, they both indicate an expanding trend on specific topics.

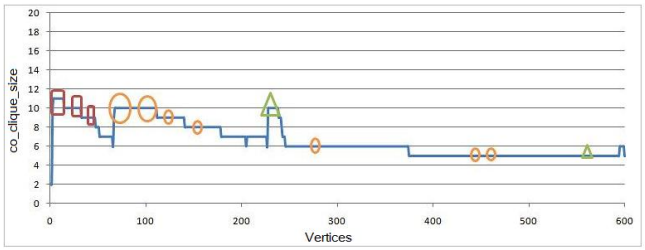
### *E. Dynamic Template Pattern Cliques: DBLP Study*

The DBLP graph data set is consisted of authors(vertices) and their collaborations(edges) in each year. In the following we detect some template pattern cliques in DBLP data set, and show that such cliques reveal some interesting hidden information about paper topics.

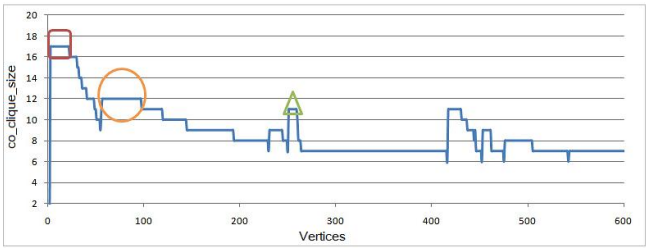
To illustrate the New Form Clique, we use the DBLP 2003 and 2004 (dynamic) data. New Form Clique Plot for DBLP in 2004 is shown in Figure 9. The red circle highlights the densest (6-vertex) New Form clique. The authors are ***Rudi Studer, Karl Aberer, Arantza Illarramendi, Vipul Kashyap, Steffen Staab, Luca De Santis***. They are from 5 different countries, and they collaborated for the first time in 2004.

In a similar manner we plot the Bridge Clique distribution of DBLP between the years 2003 and 2004 in Figure 10. The first major clique on the plot (red circle) is an interesting 6-vertex bridge clique. In 2003, the 6 authors were in two independent groups: Group 1: ***Divesh Srivastava, Graham Cormode, S. Muthukrishnan, Flip Korn***; and Group 2: ***Theodore Johnson , Oliver Spatscheck***. In Group 1, the authors primarily worked on data streams, and in Group 2 the researchers mainly worked on networking in 2003. In 2004, the 6 authors worked together on "Holistic UDAFs at Streaming Speeds", which is a topic "merged" by data stream and network.

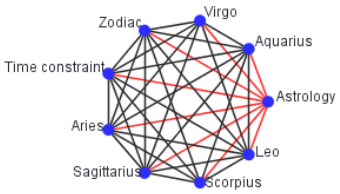
Using datasets DBLP 2000 and DBLP 2001, we plot the New Join Cliques in DBLP 2001 in Figure 11. The densest New Join clique (denoted by a red circle) shows a 9-vertex clique. in 2000, the 3 authors ***Quan Wang, David Maier,***



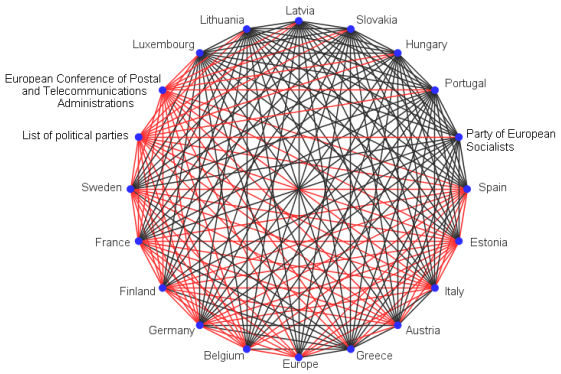

(a) Distribution of original cliques (Plot(a))



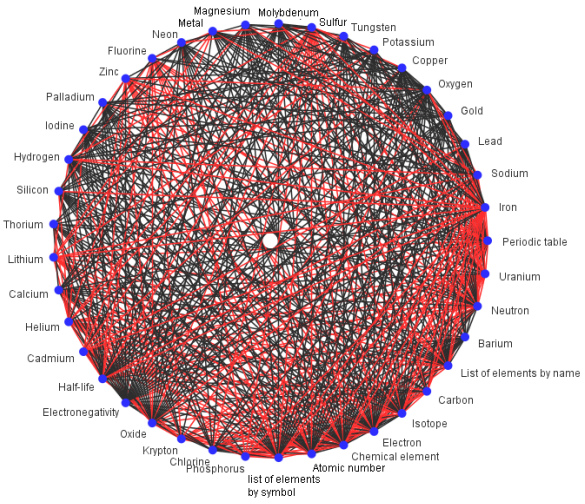

(b) Distribution of changed cliques (Plot(b))




(c) Clique details (green triangle)




(d) Clique details (red rectangle)




(e) Clique details (orange ellipse)

Fig. 8. Dual View Plots for Clique Changes

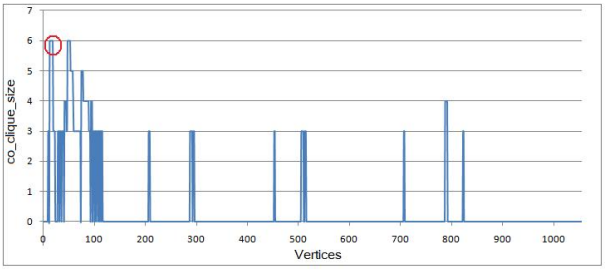



Fig. 9. Plot of New Form Cliques in DBLP 2004

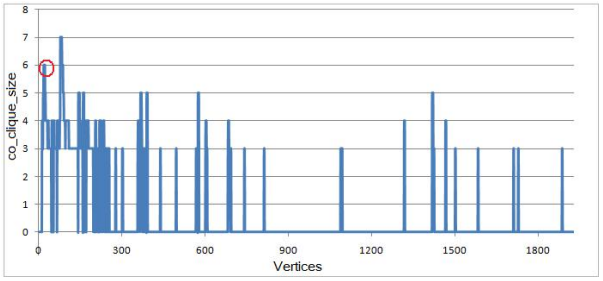



Fig. 10. Plot of Bridge Cliques in DBLP 2004

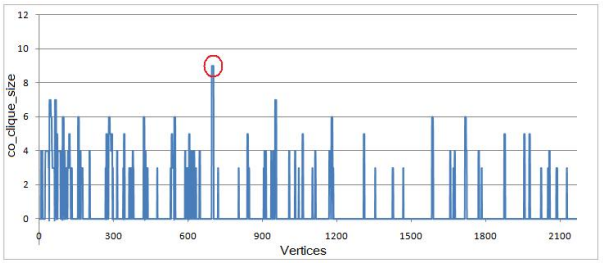


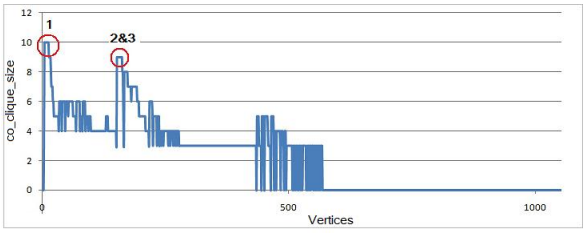

Fig. 11. Plot of New Join Cliques in DBLP 2001

***Leonard D. Shapiro*** worked on a paper about Query Processing. In 2001, the 3 authors were joined by 6 other authors who did not appear in DBLP 2000 dataset, ***Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Yu Zhang, Hsiao-min Wu***, and they worked on one paper "Exploiting Upper and Lower Bounds in Top-Down Query Optimization", which is an extension of the previous work in 2000.

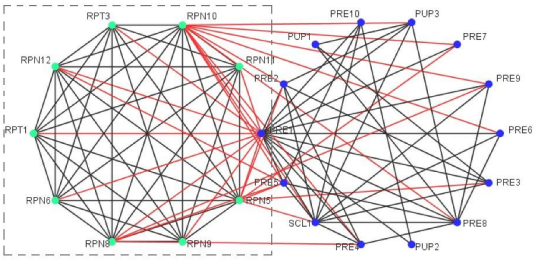
### *F. Static Template Pattern Cliques: PPI Case Study*

We next discuss how domain-driven template pattern cliques based on Triangle K-Cores can be exploited in the case of static data such as Protein Protein Interaction (PPI) data. In PPI dataset, each vertex represents a protein, and each protein belongs to a complex, which includes proteins of similar functions. Now we define a variant of Bridge Clique to be *a clique that connects vertices from two different complexes*. Here we define an edge to be "new edge" when it connects two vertices from different complexes, otherwise it is "original edge". Then we apply the previously described Bridge Clique detection algorithm on PPI dataset, and get the Bridge Clique distribution plot in Figure 12(a).

We highlight two peaks using red circles, the Bridge Clique 1 in left red circle is comprised of vertices from the following two complexes:




(a) Plot of Bridge Cliques in PPI dataset




(b) Details of bridge clique 1

Fig. 12. Detect Bridge Cliques in PPI dataset

- **20S proteasome complex**: PRE1
- **19/22S regulator complex**: RPN11, RPN12, RPN9, RPT1, RPN5, RPN5, RPT3, RPN8

In Figure 12(b), we draw the details of Bridge Clique 1 in the dashed-line rectangle, where the green vertices belong to the complex "19/22S regulator", the blue vertices belong to complex "20S proteasome", black edges are intra-complex edges, red edges are inter-complex edges. Besides drawing Bridge Clique 1, we also draw other vertices in complex "20S proteasome", and find that the vertex "PRE1" is an important bridge node connecting the two complexes.

The proteins in right red circle comprise 2 Bridge Cliques, the first is Bridge Clique 2:

- **Gac1p/Glc7p complex**: GLC7
- **mRNA cleavage and polyadenylation specificity factor complex**: PAP1, CFT2, CFT1, PTA1, MPE1, YSH1, YTH1, REF2

the second is Bridge Clique 3:

- **mRNA cleavage factor complex**: RNA14
- **mRNA cleavage and polyadenylation specificity factor complex**: PAP1, CFT2, CFT1, PTA1, MPE1, YSH1, YTH1, FIP1

We find that Bridge Clique 2 and 3 have a lot of overlap vertices, which indicate that all the vertices in them are very closely related in function.

## VIII. Conclusions

In this paper, we define and introduce the notion of a *Triangle K-Core*, a simple topological motif and demonstrate how to extract such structures efficiently from both static and dynamic graphs. We empirically demonstrate on a range of real-world data that this motif can be used as a proxy for probing and visualizing relevant clique-like structure from large dynamic graphs and networks. Finally, we discuss a method

to extend the basic definition to support user defined clique *template* patterns with applications to network visualization, correspondence analysis and event detection on graphs and networks.

## REFERENCES


[1] N. Wang, S. Parthasarathy, K.-L. Tan, and A. K. H. Tung, "CSV: Visualizing and Mining Cohesive Subgraphs," *ACM SIGMOD*, pp. 445–458, 2008.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: ordering points to identify the clustering structure," *ACM SIGMOD*, pp. 49 – 60, 1999.

[3] N. Wang, J. Zhang, K. Tan, and A. K. H. Tung, "On Triangulation-based Dense Neighborhood Graphs Discovery," *PVLDB*, 2010.

[4] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On Spectral Clustering: Analysis and an algorithm," *Advances in Neural Information Processing Systems*, vol. 14, 2001.

[5] V. Satuluri and S. Parthasarathy, "Scalable Graph Clustering Using Stochastic Flows: Applications to Community Discovery," *ACM SIGKDD*, 2009.

[6] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, 1998.

[7] I. Dhillon, Y. Guan, and B. Kulis, "A Fast Kernelbased Multilevel Algorithm for Graph Clustering," *ACM SIGKDD*, 2005.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[9] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy, "Approximating Clique is Almost NP-Complete," *FOCS*, 1991.

[10] J. Wang, Z. Zeng, and L. Zhou, "CLAN: An Algorithm for Mining Closed Cliques from Large Dense Graph Databases," *ICDE*, p. 73, 2006.

[11] J. Abello, M. G. C. Resende, and S. Sudarsky, "Massive Quasi-Clique Detection," *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, 2002.

[12] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Coherent closed quasi-clique discovery from large dense graph databases," *ACM SIGKDD*, 2006.

[13] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," *ACM SIGKDD*, 2005.

[14] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," *ACM SIGKDD*, 2006.

[15] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM TKDD*, vol. 3, no. 16, 2009.

[16] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "GraphScope: parameter-free mining of large time-evolving graphs," *ACM SIGKDD*, 2007.

[17] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, "Facetnet:a framework for analyzing communities and their evolutions in dynamic networks." *WWW*, 2008.

[18] G. M. Namata, B. Staats, L. Getoor, and B. Shneiderman, "A dual-view approach to interactive network visualization," *ACM CIKM*, pp. 939–942, 2007.

[19] X. Yang, S. Asur, S. Parthasarathy, and S. Mehta, "A Visual-Analytic Toolkit for Dynamic Interaction Graphs," *ACM SIGKDD*, 2008.

[20] J. Abello, F. V. Ham, and N. Krishnan, "ASK-GraphView: A Large Scale Graph Visualization System ," *IEEE TVCG*, 2006.

[21] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks," *CoRR, arXiv.org/cs.DS/0310049*, 2003.

[22] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, "Web graph similarity for anomaly detection," *Proceeding of the 17th international conference on World Wide Web*, 2008.


## IX. APPENDIX

### A. Triangle K-Core Update Algorithm

Before executing the update algorithm, for each edge *e*, we firstly initialize *e.order*, which indicates the "time stamp" when *e* is *processed* in Algorithm 1. If *e.order* is less than *e'.order*, then *e* is processed earlier than *e'*. *e.order* is initialized as the index of edge *e* in list *Edges* after execution of Algorithm 1.

**Algorithm 5** Update Algorithm for Adding Edges

```
1: for each added triangle t_new do
2:    Create empty lists ChangingList, PotentialList, TempList;
3:    Find the smallest value μ of t_new's edges' κ value;
4:    Put t_new's edges whose κ value equals μ in PotentialList in
      order;
5:    AddToCore(t_new, e_0); // e_0 is the first edge of PotentialList
6:    κ(e_0) + +;
7:    for each edge e in PotentialList do
8:       ori_κ(e) = μ;
9:       Construct triangles set e.addTris;
10:      for each triangle t_a in e.addTris do
11:         AddToCore(t_a, e);
12:         κ(e) + +;
13:      Construct triangles set e.delTris;
14:      for each triangle t_d in e.delTris do
15:         if κ(e) > ori_κ(e) then
16:            DelFromCore(t_d, e);
17:            κ(e) − −;
18:      Remove e from PotentialList;
19:      if κ(e) > ori_κ(e) then
20:         put e to ChangingList;
21:         Insert e.post_edges to PotentialList in order;
22:      else
23:         TempList = Simulate_Algo1(e);
24:         Insert edges in TempList between e's previous and next
            edge in Edges list;
25:   while ChangingList is not empty do
26:      TempList = Simulate_Algo1(ChangingList.min_edge);
27:      Insert edges in TempList in Edges list, between the last edge
         with κ(e) = μ and first edge with κ(e) = μ + 1;
```

**Algorithm 6** Simulate_Algo1($e_{init}$)

```
1: Create an empty list TempList;
2: Add e_init to TempList;
3: for each edge e in TempList do
4:    Construct triangles set e.addTris;
5:    for each edge e' that shares a triangle T in e.addTris with e
      and e' is in ChangingList do
6:       if κ(e') > κ(e) then
7:          DelFromCore(T, e');
8:          κ(e') − −;
9:       if κ(e') = κ(e) then
10:         Move e' from ChangingList to TempList;
11: Return TempList;
```

Algorithm 5 is to update edges' maximum Triangle K-Cores when adding edges. In step 4, according to Rule 0, we put some edges of $t_{new}$ in *PotentialList* because their maximum Triangle K-Cores might change. All edges in *PotentialList* are sorted in the increasing order of *e.order*, that is because we will simulate Algorithm 1 to recompute on *PotentialList*, we need to maintain the order. The newly added triangle $t_{new}$ is not yet in any edge's maximum Triangle K-Core, so in steps 5-6, we add it to the maximum Triangle K-Core of the first edge of *PotentialList*.

Steps 7-24 update $\kappa(e)$ for each edge *e* in *PotentialList*. In step 8, $ori\_\kappa(e)$ stores the original maximum Triangle K-Core number of *e* before update, according to Rule 0, this value is equal to $\mu$. In step 9 we construct the following set of triangles that violate Theorem 1 (*IsInCore(t, e)* tests whether triangle *t* is in edge *e's* maximum Triangle K-Core):

- *e.addTris* ={$\triangle t$ | $\triangle t$ is on edge *e*, and $\triangle t$ contains edge *e'* that $\kappa(e') > \kappa(e) \wedge IsInCore(t, e') \wedge !IsInCore(t, e)$}

Steps 10-12 then process these "illegal" triangles in *e.addTris*. After that, $\kappa(e)$ might increase and lead to the following set of triangles that violate Theorem 1:

- *e.delTris* ={$\triangle t$ | $\triangle t$ is on edge *e*, and $\triangle t$ contains edge *e'* that $e'.order < e.order \wedge \kappa(e') < \kappa(e) \wedge IsInCore(t, e') \wedge IsInCore(t, e)$},

Steps 14-17 then process these "illegal" triangles in *e.delTris*.

In step 19, if $\kappa(e)$ increases, some of *e's* neighbor edges might change $\kappa$ value, according to Rule 0, these edges are in the following set,

- *e.post_edges* = {Edge $e'$ | *e'* shares a triangle with *e*, and $\kappa(e') = \mu \wedge e'.order > e.order$}

we put these edges in *PotentialList*.

If $\kappa(e)$ does not change, then edge *e* is processed now, in step 23 we use method Simulate_Algo1 to simulate Algorithm 1 to update *e* and its neighbors' maximum Triangle K-Cores. Simulate_Algo1 will return a list of edges whose $\kappa$ value is determined.

When all edges in *PotentialList* have been processed, we update maximum Triangle K-Cores of edges in *ChangingList* (step 26), *ChangingList.min_edge* is the edge in *ChangingList* with the minimum $\kappa$ value. In step 27 we put all edges in *ChangingList* in the corresponding positions in sorted list *Edges*.

Algorithm 7 is to update edges' maximum Triangle K-Cores when deleting edges. In step 4, according to Rule 0, we put some edges of $t_{del}$ in *PotentialList*. In steps 5-8, we remove deleted triangles from its edges' maximum Triangle K-Cores. In step 11, we construct two sets of triangles on e:

- *e.addTris* = {$\triangle t$ | $\triangle t$ is on edge *e*, and contains edge *e'* that, $\kappa(e') = ori\_\kappa(e) \wedge e'.order < e.order \wedge IsInCore(t, e') \wedge !IsInCore(t, e)$ }
- *e.delTris* = {$\triangle t$ | $\triangle t$ is on edge *e*, and contains edge *e'* that, $\kappa(e') < ori\_\kappa(e) \wedge IsInCore(t, e') \wedge IsInCore(t, e)$ }

When step 13 is satisfied, all the triangles in *e.addTris* violate Theorem 1, so we add the first triangle of *e.addTris* to *e's* maximum Triangle K-Core to obey Theorem 1. Then $\kappa(e)$ changes and we test step 20, if step 20 is satisfied, all the triangles in *e.delTris* violate Theorem 1, so we remove the first triangle of *e.delTris* from maximum Triangle K-Core of *e* to obey Theorem 1. In steps 28-30, if $\kappa(e)$ changes, according to Rule 0 we find the following set of edges whose maximum Triangle K-Core might change, and insert them in *PotentialList*.

**Algorithm 7** Update Algorithm for Deleting Edges

```
1:  for each deleted triangle t_del do
2:      Create empty lists ChangingList, PotentialList;
3:      Find the smallest value μ of t_del's edges' κ value;
4:      Put t_del's edges whose κ value equals μ in PotentialList in
        order;
5:      for each edge e in PotentialList do
6:          if IsInCore(t_del, e) then
7:              DelFromCore(t_del, e);
8:              κ(e) − −;
9:      for each edge e in PotentialList do
10:         ori_κ(e) = μ;
11:         Construct triangles sets e.addTris and e.delTris;
12:         while true do
13:             if κ(e) < ori_κ(e) then
14:                 if e.addTris is not empty then
15:                     AddToCore(e.addTris.first, e);
16:                     κ(e) + +;
17:                     remove e.addTris.first from e.addTris;
18:                 else
19:                     break;
20:             if κ(e) = ori_κ(e) then
21:                 if e.delTris is not empty then
22:                     DelFromCore(e.delTris.first, e);
23:                     κ(e) − −;
24:                     remove e.delTris.first from e.delTris;
25:                 else
26:                     break;
27:         Remove e from PotentialList;
28:         if κ(e) < ori_κ(e) then
29:             Put e in ChangingList;
30:             Insert e.share_edges to PotentialList in order;
31:     Insert edges in ChangingList in Edges list, between the last
        edge with κ(e) = μ − 1 and first edge with κ(e) = μ;
```

- *e.share_edges* = {Edge $e'$ | *e'* shares a triangle with *e*, $\kappa(e') = \mu$}

Finally we put the edges in *ChangingList* in correct positions in list *Edges*.

In Algorithm 5 and 7, after each iteration, each edge's *order* value needs to be re-computed, which will be costly. In our implementation, we only update edges whose *order* value have been changed, that is, when a set of edges {*e1, e2, ...en*} are inserted between two edges *Ea, Eb,* then $ei.order = Ea.order + (Eb.order - Ea.order) * i/(n+1)$.

If we do not store triangles in Algorithm 1, when updating edge *e* in *PotentialList* we need to re-construct *e*'s triangles, and the triangle information we need to know is whether a triangle of *e* is in *e*'s maximum Triangle K-Core. We recover this information as following: we firstly get triangle *t*'s "process time", which is the smallest *order* value of its edges, then we apply the following Rule to find all *e*'s triangles in *e*'s maximum Triangle K-Core.

- Rule 1: if $\kappa(e)$=k, then we sort *e*'s triangles in the increasing order of their "process time", the last k triangles will be in *e*'s maximum Triangle K-Core.

### *B. Proof of Correctness of Triangle K-Core Update Algorithm*

**Rule 0**: when triangle *t* is added/deleted to graph $G$, assume $\mu$ is smallest $\kappa$ value of *t*'s three edges, then only the edges in

$G$ whose $\kappa$ value equals $\mu$ might have their maximum Triangle K-Cores changed.

**Proof of Rule 0:** When triangle *t* is added/deleted to graph $G$, we first prove the following two lemmas:

*Lemma1*: for edges of *t*, only the edges with smallest $\kappa$ value $\mu$ might change change $\kappa$ value, and the change is 1.

*Proof*: According to Theorem 1, only the edges in *t* with smallest $\kappa$ value could have triangle *t* in their maximum Triangle K-Cores, so adding/deleting *t* will only affect those edges in *t*, and their $\kappa$ value would increase/decrease by 1.

*Lemma2*: If one edge e with $\kappa(e) = \mu$ changes $\kappa(e)$ by 1, then for any neighbor edge e' of e, e' might change $\kappa$ value only when $\kappa(e') = \mu$, and the change is 1.

*Proof*: Here we prove Lemma 2 when $\kappa(e)$ increases by 1, the case when $\kappa(e)$ decreases by 1 could be proved in the similar way. Assume e' and e share a common triangle T'.
If $\kappa(e') >= \mu+1$, then according to Theorem 1, originally T' is not in maximum Triangle K-Core of e'. When e increases $\kappa(e)$ to $\mu+1$, if T' is added in maximum Triangle k-core of e', then $\kappa(e')+1 >= \mu+2 > \kappa(e)$, this violates Theorem 1, so T' should not be added in maximum Triangle K-Core of e' and $\kappa(e')$ should stay the same.
If $\kappa(e') <= \mu-1$, then according to Theorem 1, originally T' is not in maximum Triangle K-Core of e. When e increases $\kappa(e)$ to $\mu+1$, T' should still not be in maximum Triangle K-Core of e, and thus $\kappa(e')$ is not affected.
So only when $\kappa(e') = \mu$, $\kappa(e')$ could change, The change is due to adding(deleting) T' to(from) the maximum Triangle K-Core of e', so the change could only be 1.

Lemma 1 shows that the edges with $\kappa = \mu$ firstly change $\kappa$, and Lemma 2 shows that the change is then propagated to edges with $\kappa = \mu$, so Rule 0 is proved.

**Rule 1**: if $\kappa(e_t)$=k, then we sort $e_t$'s triangles in the increasing order of their "process time", the last k triangles will be in $e_t$'s maximum Triangle K-Core.

**Proof of Rule 1:** For edge $e_t$, during iteration i of Algorithm 1 when $\tilde{\kappa}(e_t) > k$, if one triangle T on $e_t$ is processed in step 17, then during that iteration edge $e_i$ is processed before $e_t$, that means $\kappa(e_i) <= k < \tilde{\kappa}(e_t)$, so step 13 is true, T is removed from $e_t$'s maximum Triangle K-Core in step 14. This means for a triangle T on $e_t$, if it is processed before $\tilde{\kappa}(e_t)$ reaches k, it is not in et's maximum Triangle K-Core. So when $\tilde{\kappa}(e_t)$ reaches k, it has k unprocessed triangles in its maximum Triangle K-Core, Rule 1 is proved.

**Proof of Correctness of Algorithm 2:** The correctness of Algorithm 2 is guaranteed by achieving following two objectives:
1: identifying all those edges whose $\kappa$ value might change (these edges are stored in *PotentialList*);
2: re-computing the $\kappa$ value of those edges in *PotentialList*;

Next we show that how the two objectives above are achieved.
1. An edge e might change $\kappa$ value only when its triangle(s) is added to or deleted from its current maximum Triangle K-Core, this would happen when e belongs to an added/deleted triangle T, or e's neighbor edges change $\kappa$ value. The former case is handled by step 3, and the latter case is handled in step 8, so objective 1 is achieved.
2. To re-compute the $\kappa$ value of those edges in *PotentialList*, we simulate running Algorithm 1 on them. In order to do so, we store edges in PotentialList in the same order as they are processed in Algorithm 1, and simulate the steps in Algorithm 1 on them (please refer to Algorithm 5 and Algorithm 7 for details). Since in Algorithm 1 each edge's triangles are processed to obey Theorem 1, we also process edges stored in PotentialList based on Theorem 1.