AOT: Pushing the Efficiency Boundary of Main-memory Triangle Listing

Michael Yu[1], Lu Qin[2], Ying Zhang[2], Wenjie Zhang[1], and Xuemin Lin[1]

[1] University of New South Wales {mryu,wenjie.zhang,lxue}@cse.unsw.edu.au
[2] University of Technology Sydney {lu.qin,ying.zhang}@uts.edu.au

Abstract. Triangle listing is an important topic significant in many practical applications. Efficient algorithms exist for the task of triangle listing. Recent algorithms leverage an orientation framework, which can be thought of as mapping an undirected graph to a directed acylic graph, namely *oriented graph*, with respect to any global vertex order. In this paper, we propose an adaptive orientation technique that satisfies the orientation technique but refines it by traversing carefully based on the out-degree of the vertices in the oriented graph during the computation of triangles. Based on this adaptive orientation technique, we design a new algorithm, namely AOT, to enhance the edge-iterator listing paradigm. We also make improvements to the performance of AOT by exploiting the local order within the adjacent list of the vertices.
We show that AOT is the first work which can achieve best performance in terms of both practical performance and theoretical time complexity. Our comprehensive experiments over 16 real-life large graphs show a superior performance of our AOT algorithm when compared against the state-of-the-art, especially for massive graphs with billions of edges. Theoretically, we show that our proposed algorithm has a time complexity of $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\}))$, where $\boldsymbol{E}$ and $deg^+(x)$ denote the set of directed edges in an oriented graph and the out-degree of vertex x respectively. As to our best knowledge, this is the best time complexity among in-memory triangle listing algorithms.

Keywords: Triangle · Enumeration · Graph algorithm

1 Introduction

Triangle-listing is one of the most fundamental problems in graphs, with numerous applications including structural clustering [25], web spamming discovery [4], community search [5, 18], higher-order graph clustering [26], and role discovery [6]. A large number of algorithms have been proposed to efficiently enumerate all triangles in a given graph. These include in-memory algorithms [20, 19, 15, 27], I/O efficient algorithms [7, 8, 11, 12], and parallel/distributed algorithms [21, 10, 16], etc. In this paper, we focus on in-memory triangle-listing algorithms and aim to achieve the best performance from both theoretical and practical aspects.

State-of-the-art. The existing state-of-the-art in-memory triangle-listing algorithms are based on vertex ordering and orientation techniques [15, 9]. Given

an undirected simple graph, these algorithms first compute a total-order η for all its graph vertices; one example of such ordering is the non-increasing degree order. With the total-order η, a direction can then be specified for each undirected edge (u, v) such that $\eta(u) < \eta(v)$. Once complete, the graph orientation technique converts the initial undirected graph G into a directed acyclic graph $\boldsymbol{G}$. With an oriented graph, the original problem of triangle-listing on an undirected simple graph G is recast to a problem of finding three vertices u, v, w so that directed edges $\langle u, v\rangle$, $\langle v, w\rangle$, $\langle u, w\rangle$ exist in $\boldsymbol{G}$. For directed triangle instances, we refer to the role of vertex u with 2 out-going edges that form a triangle as a pivot.

Motivation. We give an example for finding triangles in an oriented-graph [9]. For each pivot vertex u, the method first initializes an index to the out-neighbors of u; after that, for each out-neighbor v of u, it traverses the out-neighbor w of v and checks to see whether w is also an out-neighbor of v in the index. The advantage of this technique is twofold: First, by simply processing all pivot vertices using the above procedure, it already guarantees that each triangle is generated only once without performing the normal pruning for duplicate triangle solutions. Secondly, parallelization of the algorithm is easy if we process sets of pivot vertices independently. This algorithm has the time complexity of $\Theta(\Sigma_{v \in V} deg^+(v) \cdot deg^-(v))$ and is bounded by $O(m^{1.5})$ [9], where V is the set of vertices in $\boldsymbol{G}$; $deg^+(v)$ and $deg^-(v)$ are the numbers of out-neighbors and in-neighbors of v in $\boldsymbol{G}$ respectively; and m is the number of edges in $\boldsymbol{G}$.

A dominant cost of the above algorithm is that of look-up operations, where the algorithm searches the out-neighbors of each pivot. A natural question is raised: Is it possible to significantly reduce the number of look-up operations needed by the algorithm? To answer this question, we first find the time complexity of the former algorithm is equivalent to that of $\Theta(\Sigma_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v))$, where $\boldsymbol{E}$ is the set of directed edges in $\boldsymbol{G}$. In other words, for each directed edge $\langle u, v\rangle \in \boldsymbol{E}$, the algorithm will always spend $deg^+(v)$ amount of look-up operations irrespective of whether $deg^+(v) \leq deg^+(u)$ holds. We find that if we are able to spend $deg^+(u)$ operations in the case that $deg^+(v) > deg^+(u)$ for the edge $\langle u, v\rangle \in \boldsymbol{E}$, the cost of the algorithm can be further lessened, therefore it motivates us to explore new ways to further leverage the properties from graph orientation at a finer level, to improve the algorithm both theoretically and practically.

Challenges. Faced with the above problem, we ask intuitively: Can we tackle the asymmetry by manually reversing the direction of each edge $\langle u, v\rangle \in \boldsymbol{E}$ if $deg^+(v) > deg^+(u)$ and then reuse the same algorithm on the now modified oriented graph? Unfortunately, this solution is infeasible. To explain, reversing the direction of the selected edges can result in cycles ($\langle u, v\rangle$, $\langle v, w\rangle$, and $\langle w, u\rangle$) in the oriented graph $\boldsymbol{G}$, such cyclic triangles will be missed by the aforementioned algorithm. To ensure algorithmic correctness, for each undirected edge (u, v), up to two orientations need to be kept simultaneously: the original orientation in $\boldsymbol{G}$ and the orientation specified by the comparison of $deg^+(u)$ and $deg^+(v)$ because the two orientations can be inconsistent. Therefore, to make our idea practically applicable, the following issues will be addressed in this paper: (1) How can we integrate the two orientations to improve the algorithm complexity

Table 1. The Summary of Notations

Notation	Definition
$G = (V, E)$	an undirected graph with vertices V and edges E
$\boldsymbol{G}$	a directed graph with vertices V and directed edge $\boldsymbol{E}$
u, v, w, x, y	vertices in the graph
(u, v)	an undirected edge between vertices u and v
$\langle u, v \rangle$	a directed edge from vertex u to v
(u, v, w)	a triangle with vertices u, v and w
$deg(u)$	the degree of the vertex u
$deg^{+}(u)$	the out-degree of u in oriented graph

and also ensure that each triangle is enumerated once and only once? and (2) Can we further improve the efficiency of the algorithm practically by exploring some local vertex orders?

Contributions. In this paper, we answer the above questions and make the following contributions.

(1) We have designed a new triangle listing algorithm named *Adaptive Oriented Triangle-Listing* (AOT) by developing novel adaptive orientation and local ordering techniques, which can achieve the best time complexity among the existing algorithms in the literature.

(2) We conduct an extensive performance study using 16 real-world large graphs at billion scale, to demonstrate the high efficiency of our proposed solutions. The experimental results show that our AOT algorithm is faster than the state-of-the-art solutions by up to an order of magnitude. It is also shown that AOT can be easily extended to parallel setting, significantly outperforming the state-of-the-art parallel solutions.

Organization. The rest of the paper is organized as follows. Section 2 provides a problem definition and states the notations used and introduce two state-of-the-art methods. Section 3 explains some motivation and explains our proposed algorithm. Section 4 describes the experimental studies conducted and reports on findings from the results. Section 5 presents the related work. Section 6 concludes the paper.

2 Background

In this section, we formally introduce the problem and the state-of-the-art techniques. Table 1 is a summary of the mathematical notations used in this paper.

2.1 Notations and Problem Definition

Let $G = (V, E)$ be an undirected simple graph, where V and E are a set of vertices and a set of edges, respectively. Below, we also use V(G) and E(G) to denote V and E of a graph G. The number of vertices and the number of edges is denoted as n and m for $n = |V|$ and $m = |E|$, respectively. For undirected graph G, we denote the set of neighbors of vertex u in G as $N(u)$ and denote the degree of u in G as $deg(u)$ which is equal to $|N(u)|$. For a directed graph $\boldsymbol{G} = (V, \boldsymbol{E})$, we use $\boldsymbol{E}$ to denote the set of directed edges $\{\langle u, v \rangle\}$ where u and v are the starting and ending vertex respectively. We denote the set of outgoing-neighbors

of vertex u in G as $N^+(u)$, and the out-degree as $deg^+(u) = |N^+(u)|$. Likewise, we denote the in-neighborhood of vertex u in $\boldsymbol{G}$ as $N^-(u)$, and the in-degree as $deg^-(u) = |N^-(u)|$. By (u, v), we denote an undirected edge between two vertices u and v. A **triangle** is a set of three vertices fully connected to each other. We denote by (u, v, w) a triangle consisting of three vertices u, v and w.

Problem statement. Given an undirected simple graph $G = (V, E)$, we aim to develop an efficient main-memory algorithm to list all triangles in the graph G one by one, with both good time complexity and practical performance.

2.2 Compact Forward (CF) Algorithm

We consider the method *Compact-forward* (CF) [15] as a state-of-the-art for triangle listing; although it was designed in 2008, its efficiency for triangle listing is still referred to frequently [9]. There are two key components in the CF algorithm: the "*edge-iterator*" computing paradigm and the *orientation* technique.

Edge-iterator. The "edge-iterator" is a recurring computing paradigm for triangle listing, its strategy for triangle listing is to find triangles with reference to pairs of adjacent vertices. Given an edge (u, v), any triangle that includes the edge must contain a third vertex w that has connections to both of u and v. Thus, we can obtain any triangles containing edge (u, v) based on the intersection of $N(u)$ and $N(v)$. For each edge, the edge-iterator returns the set of triangles associated with that edge, and when repeated on all edges, the set of all triangle solutions is made available.

Orientation technique. An orientation technique is recently leveraged in triangle listing algorithms, this involves the generation of a directed (i.e., oriented) graph $\boldsymbol{G}$ from an initially undirected input graph G [15]. Each undirected edge is mapped to a directed edge where the direction (i.e., orientation) is decided by the rank of its endpoints in a vertex-ordering (e.g., out-degree [15]). We refer to vertex u as a *pivot vertex* if u has two out-going edges. We can association a triangle in the undirected graph with only one pivot vertex to ensure one and only one instance of this triangle in the output, which significantly improves the performance.

Algorithm 1: **CF**(G)

```
   Input  : G : an undirected graph
   Output : All triangles in G
1  G ← Orientation graph of G based on degree-order;
2  for each vertex u ∈ G do
3      for each out-going neighbor v do
4          T ← N+(u) ∩ N+(v) ;
5          for each vertex w ∈ T do
6              Output the triangle (u, v, w);
```

Compact Forward (CF) Algorithm. The CF algorithm is designed based on the edge-iterator and the orientation technique. We show its pseudocode in Algorithm 1. In line 1, undirected graph G is transformed into a directed

Algorithm 2: **kClist**(G)

```
   Input  : G : an undirected graph
   Output : All triangles in G
1  G ← Orientation graph of G based on degeneracy order η ;
2  for each vertex u ∈ G do
3      for any two out-going neighbors {v, w} of u with η(v) < η(w) do
4          if there is a directed edge ⟨v, w⟩ ∈ E then
5              Output triangle (u, v, w);
```

graph $\boldsymbol{G}$ via the *orientation* technique. (Line 2 onward follows the edge-iterator framework.) In Line 3, triangles are enumerated by iterating through the outgoing-neighborhoods rather than the full neighborhood. In Line 4, a *merge-based intersection* identifies the common out-going neighbors of u and v, denoted by T. A set of triangles (u, v, w) is then output for every vertex $w \in T$.

Analysis. Since all triangles identified are unique, a naive traversal of the oriented graphs edges (the outgoing-neighborhoods for each vertex) yields the complete set of unique solutions without explicit duplicate pruning. In terms of time complexity, the merge-based intersection operation at Line 4 takes $\Theta(deg^+(u) + deg^+(v))$, assuming that the directed adjacency lists of u and v are sorted. In total, the CF algorithm has a complexity of $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(u) + deg^+(v))$.

Remark 1. There is also an alternative implementation of the CF algorithm that adopts hash tables for the intersection operation, namely CF-Hash. Suppose a hash table has been built for each vertex based on the out-going neighbors in the oriented graph. At Line 4 of Algorithm 1, we may choose the vertex with larger number of neighbors as the hash table for intersection operation with $\Theta(\min\{deg^+(u), deg^+(v)\})$ look-up cost. This can come up with a better time complexity of $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\})$. However, as reported in [15, 21] and our experiments, the practical performance of hash-based CF algorithm is not competitive compared to the above merge-based CF algorithm. Thus, the merge-based CF algorithm is used as the representative of CF algorithm in the literature.

2.3 k-Clique Listing Algorithm (kClist)

We introduce the second state-of-the-art algorithm for in-memory triangle listing. The kClist algorithm [9] lists cliques for a queried size k, we restrict our discussion to the relevant use-case when $k = 3$ for listing triangles. kClist follows the node-iterator triangle listing paradigm which is described below.

Node-iterator. The "node-iterator" triangle listing paradigm lists triangles by inspecting for adjacency between vertex pairs within one vertex neighborhood. For example, consider the neighboring vertices of node u, if there is an edge between two neighbors v_2 and v_3, then the triangle solution (u, v_2, v_3) is output.

k-Clique Listing (kClist) Algorithm. The kClist algorithm begins by generating an oriented graph $\boldsymbol{G}$ based on the degeneracy ordering [2]. We use η

to denote the degeneracy ordering here. In lines 3 - 5 of Algorithm 2, for every two out-going neighbor v and w where $\eta(v) < \eta(w)$, the existence of a directed edge $\langle v, w\rangle$ is assessed; for each edge found, a triangle solution is output.

Analysis. The running time of kClist is $\Theta(m + \sum_{u \in V} deg^+(u) \times deg^-(u))$, this can also be expressed as $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v))$. It is apparent that the time complexity of kClist is an improvement compared to the CF algorithm which takes $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}}(deg^+(u) + deg^+(v)))$ time, its practical performance is also shown to be efficient.

3 Our Approach

We introduce our adaptive orientation technique and implement it in our algorithm, AOT, to push the efficiency boundary for main-memory triangle listing algorithms.

3.1 Motivation and Problem Analysis

Since the proposal of the orientation technique, its nice properties and good practical performance have allowed it to gradually become a valuable technique utilized in subsequent studies of triangle listing.

Shortcoming of Orientation Technique. We recognize the prevalent usage of this orientation technique, however, we respond by showing that although current methods leverage the salient benefits of orientation, there are still finer benefits that are overlooked. We argue that there are still ways to further leverage properties that can improve the existing performances of triangle-listing. Our goal is therefore to maximize the benefits of the orientation technique.

In the following discussion of oriented edges, relative to a vertex u, we refer to edges $\langle u, v\rangle$ as *positive edges* if the out-degree relation of its adjacent vertex v has an out-degree value that is greater or equal to that of the pivot vertex ($deg^+(u) \leq deg^+(v)$); we also refer to edges $\langle u, v\rangle$ as *negative edges* if the pivot vertex has the greater out-degree value ($deg^+(u) > deg^+(v)$).

We refer to the time complexity of the kClist algorithm and see that it is $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v))$ when listing triangles. However, this is not optimal for triangle listing. In Figure 1, consider the point in the triangle listing computation where triangles of edge $\langle u, v\rangle$ are processed. With respect to pivot vertex u, $deg^+(v)$ is larger than that of $deg^+(u)$ since $4 > 3$. The aforementioned complexity is not favorable for processing this ordinary edge. Its issue is because its cost is strictly that of $deg^+(v)$ (i.e., 4). Our observation is that, if we can reverse the direction of the edge $\langle u, v\rangle$ to follow the out-degree vertex-order, we can process $\langle u, v\rangle$ more favorably with $deg^+(u)$ (i.e., 3) and come up with a better time complexity.

Solutions are non-trivial. Obviously, the optimal instance for a running time of $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v))$ is an oriented graph $\boldsymbol{G}$ where all edges are positive. However, most graphs do not exhibit that property: in most cases, not all graphs edges $\{\langle u, v\rangle\}$ are necessarily positive at the same. Recall Figure 1: while edges such as $\langle u, x\rangle$ and $\langle u, w\rangle$ are positive, negative edges such as $\langle u, v\rangle$ are also possible.

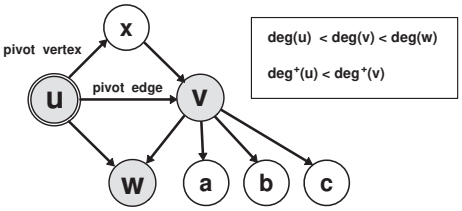


Fig. 1. Limit of orientation technique

We remedy the existence of negative edges by making a series of modifications to the computed orientation of negative edges after it is oriented. One naive way of achieving this is to manually change the direction of the oriented edge. For example, there is a negative edge $\langle u, v\rangle$ in Figure 1, we see that it can become a positive edge if its direction is simply changed to $\langle v, u\rangle$. This methodology is limited because it ultimately undermines the total order used in the orientation, moreover, changing $\langle u, v\rangle$ creates a cycle subgraph (u, x, v, u); this is a critical complication since triangle (u, x, v) would surely be omitted and missing from the result set of existing methods.

Ultimately, the out-degree of a vertex is a result of the orientation of its incident edges, and therefore depends directly on the total ordering used for the orientation techniques, it is difficult to significantly reduce the number of negative edges by manually changing its orientation.

The main idea. As an alternative, we instead suggest modifying the computing order of u and v on the fly when encountering negative edges instances $\{\langle u, v\rangle\}$. We notice that the CF algorithm cannot take advantage of this because its complexity of $\Theta(deg^+(u) + deg^+(v))$ for every edge $\langle u, v\rangle$ suggests that the design of CF is insensitive to the direction of the edge. We also notice that the kClist algorithm cannot do this either, because the accessing order of the vertices has to strictly follow the degeneracy order on the oriented graph to ensure the correctness of the algorithm. We have showed that two state-of-the-art techniques cannot trivially take advantage of this observation. In contrast, our algorithm does not depend on any total order, any total order will be acceptable.

Following the above analysis, we are motivated to develop a technique that tightens the boundary for efficient triangle listing, by taking advantage of the resulting out-going degree order of each incoming edge, and adaptively listing triangle based on its property. Our key idea involves selecting the optimal pivot vertices for each triangle accordingly, such that each triangle is found only from the vertex with a smaller out-going degree. This way we achieve the time complexity $\Theta(\min\{deg^+(u), deg^+(v)\})$ for every edge $\langle u, v\rangle \in \boldsymbol{E}$, which is now *optimal* for a given oriented graph following the edge-iterator paradigm.

When finding the intersection between the out-going neighbors of adjacent vertices u and v (i.e. $N^+(u) \cap N^+(v)$ for an oriented edge $\langle u, v\rangle$), there is a larger and a smaller out-degree vertex, we use the hash-join approach as the appropriate method to perform the set-intersection. Note that one hash-table here contain the out-going neighbors for one single vertex. Following the hash-join approach, we choose to look-up the out-going neighbors of the vertex with the lesser out-degree in the hash structure of the vertex with the greater out-degree. However,

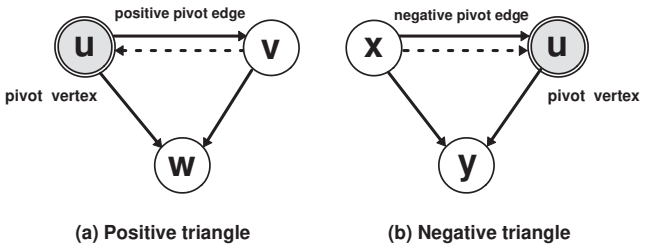


Fig. 2. Motivation for Adaptive Orientation

with one endpoint fixed, within its adjacent neighborhood, the endpoint with the lesser out-degree vertex varies, to accomplish the former statement efficiently is hard because it is not known in advance which endpoint has the smaller out-degree vertex. The known solution requires both hash tables for either endpoints be available when the edge is visited. There are two methods for constructing the two indexes in advance: (1) Building hash tables of all graph vertices prior to listing. Where this is a naive solution, it is computationally infeasible due to its high storage demand and a high look-up cost. (2) For each vertex u, building a hash table for all for its out-going neighbors on the fly. This is also infeasible because one vertex is likely to need to rebuild its hash table multiple times throughout the listing stage.

Categorizing Triangles. To facilitate understanding of technique, we discuss two categorizes for each triangle in an oriented graph $\boldsymbol{G}$: *positive triangle* or *negative triangle*. The category of each triangle depends on its pivot edge: given an oriented triangle, were refer to it as positive if its pivot edge $\langle x, y\rangle$ is positive i.e. $\eta(x) < \eta(y)$ and $deg^+(x) \geq deg^+(y)$; otherwise, it is negative if $\eta(x) < \eta(y)$ and $deg^+(x) < deg^+(y)$, where η is the vertex ordering used in the orientation.

An example instance of positive and negative triangles is shown in Figures 2(a)-(b). We consider the two triangles from a sample graph with a common vertex u, we note that without additional structural information from the graph, the induced subgraphs u,v,w and x,u,y are isomorphic. However, with attention to pivot edges $\langle u, v\rangle$ and $\langle x, u\rangle$, we observe that the two triangles are different in terms of the out-going degree order (the dotted line points to the vertex with the higher out-degree), and there for their orientation is different.

We propose seperate computations for the two types of triangles due to their subtle differences, by selecting different respective piviot veritices. The selection of the pivot vertex affects the amount of computation to list the triangle. If it is a positive triangle, we remain consistent with the orientation technique and use the vertex with two out-going edges as the *pivot vertex* (e.g. the vertex u in Figure 2(a)). However, if it is a negative triangle, we select a different vertex as the pivot vertex (e.g. u in Figure 2(b)).

A direct benefit from the above selection is that, every vertex $u \in \boldsymbol{G}$ is now the pivot vertex for both *positive* and *negative* triangles solutions, where the previous technique rigidly processes all triangles as positive triangles. In the traditional orientation technique, both triangles would be processed equally and listed by vertices u and x respectively, this was because the pivot vertex of each triangle is strictly the vertex with two out-going edges and does not account for

our positive or negative triangle definitions. A desirable property of our *adaptive orientation technique* is that this way each vertex u only needs to build a hash table once for its out-going neighbors.

To conclude: For positive triangles with pivot vertex u, for each out-going neighbor v of u (e.g. v in Figure 2(a)), we will look-up if each out-going neighbors w of v (e.g. w in Figure 2(a)) is also in the hash table, to see if w is also an out-going neighbor of u. For the negative triangles with pivot vertex u, for each in-coming neighbor x of u (e.g. x in Figure 2(b)), we will look-up if each out-going neighbor y of x (e.g. y in Figure 2(b)) is also in the hash table, to see if y is also an out-going neighbor of u.

As we later show in the theoretical analysis in Section 3.2, our proposed *adaptive orientation* technique achieves the time complexity of $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\})$ because, in terms of the hash-based intersection, the look-up operations will always be performed on the vertex with larger out-degree values for each oriented edge.

3.2 The Algorithm

We introduce the algorithm with our proposed adaptive orientation technique. With reference to the pseudo-code in Algorithm 3, In line 1, the orientated graph $\boldsymbol{G}$ is generated following the degree vertex order. In lines 2-13, a vertex u acts as pivot vertex and lists its associated positive and negative triangles. For each pivot vertex u, Line 3 generates a bitmap hash table H based on its adjacency neighborhood.

For pivot vertex u, all *positive triangles* are enumerated in Lines 4-8. That is, for each out-going neighbor v of u with $deg^+(v) < deg^+(u)$ (i.e. positive pivot edge), we find its out-going neighbors which are also out-going neighbors of u by looking up the hash table H as illustrated in Figure 2(a).

Similarly, all *negative triangles* for pivot vertex u are enumerated in Lines 9-13. For each in-coming neighbor x of u with $deg^+(x) < deg^+(u)$ (i.e. negative pivot edge), we find its out-going neighbors which are also out-going neighbors of u by looking up the hash table H as illustrated in Figure 2(b).

Correctness. To explain the correctness of our algorithm, we recall that each oriented triangle in $\boldsymbol{G}$ belongs to either a positive type triangle or a negative type triangle, we note that this is true for any vertex total-order.

Given an oriented triangle (u, v, w): such that u is the pivot vertex, and $\langle u, v\rangle$ is its pivot edge as illustrated in Figure 1. If $deg^+(v) < deg^+(u)$, then (u, v, w) is a positive triangle with pivot vertex u; given w is the common out-going neighbor of u and v, a triangle will be output at Line 8 of Algorithm 3. Otherwise, if the triangle is not positive i.e., if $deg^+(u) < deg^+(v)$[3], (u, v, w) is a negative triangle with pivot vertex v, this oriented triangle will be output at Line 13 of Algorithm 3 when v is the pivot vertex, because w is the common out-going neighbor of u and v. Evidently, this oriented triangle (u, v, w) will not be output under any other scenario when following the oriented triangle technique. Consequently, this showed that (u, v, w) will be output once and only once, the correctness of the Algorithm 3 follows.

[3] Recall that ties are broken by vertex ID.

Algorithm 3: Our Algorithm – AOT (G)

```
   Input  : G : an undirected graph
   Output : All triangles in G
 1 G ← orientation graph of G based on degree-order;
 2 for u ∈ V(G) do
 3     Set-up the hash table H with IDs of the out-going neighbors of u (N+(u)) ;
 4     for every out-going neighbor v of u do
 5         if deg+(v) < deg+(u) then
 6             for every out-going neighbor w of v do
 7                 if Find w in H then
 8                     output triangle (u, v, w);
 9     for every in-coming neighbor x of u do
10         if deg+(x) < deg+(u) then
11             for every out-going neighbor y of x do
12                 if Find y in H then
13                     output triangle (u, x, y);
```

Time Complexity. We use a bitmap with size $|V|$ to implement the hash table H, where $H[v.ID] = 1$ if the vertex v is the out-going neighbor of the pivot vertex. For each pivot vertex u visited, we can use $\Theta(deg^+(u))$ time to initiate or clean the hash table H. Thus, the maintenance of H takes $\Theta(2m)$ time.

Recall that for a pivot edge $\langle u, v\rangle$, the set of triangles it outputs can be a mix of either positive or negative triangles. For every pivot edge $\langle u, v\rangle$, the time complexity for its positive triangles is $\Theta(deg^+(v))$ with $deg^+(v) < deg^+(u)$ since the time complexity of Line 8 is $\Theta(1)$. Similarly, the time complexity for its negative triangles is $\Theta(deg^+(u))$ with $deg^+(u) < deg^+(v)$ since the time complexity of Line 13 is $\Theta(1)$. It follows that the total time complexity of our Algorithm 3 is $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\}))$

Example 1. In Figure 3, the oriented graph has 14 vertices and 21 edges. Out of the 21 edges, 9 for which have a $deg^+(v)$ value of greater than 0. For $\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v))$: Edges $\langle v_1, v_3\rangle$, $\langle v_5, v_7\rangle$ and $\langle v_9, v_{11}\rangle$ each incur a cost of 3. Edges $\langle v_2, v_4\rangle$, $\langle v_6, v_8\rangle$, $\langle v_{10}, v_{12}\rangle$ each incur a cost of 2. Edges $\langle v_3, v_4\rangle$, $\langle v_7, v_8\rangle$ and $\langle v_{11}, v_{12}\rangle$ each also incur a cost of 2. The remaining edges incur no cost. In total, $\sum_{\langle u,v\rangle \in \boldsymbol{E}} deg^+(v)) = 3 + 3 + 3 + 2 + 2 + 2 + 2 + 2 + 2 = 21$. For $\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\}$): Edges $\langle v_1, v_3\rangle$, $\langle v_5, v_7\rangle$ and $\langle v_9, v_1 1\rangle$ each incur a cost of 1. Edges $\langle v_2, v_4\rangle$, $\langle v_6, v_8\rangle$, $\langle v_{10}, v_{12}\rangle$ each also incur a cost of 1. Edges $\langle v_3, v_4\rangle$, $\langle v_7, v_8\rangle$ and $\langle v_{11}, v_{12}\rangle$ each incur a cost of 2. The remaining edges incur no cost. In total, $\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\}) = 1 + 1 + 1 + 1 + 1 + 1 + 2 + 2 + 2 = 12$.

The former is a calculation of the computation required by the state-of-the-art, the latter is the computations required by our algorithm. In comparison, Example 1 illustrates that our algorithm incurs significantly fewer computation to list triangles. Where the costs for some edges is the same between two

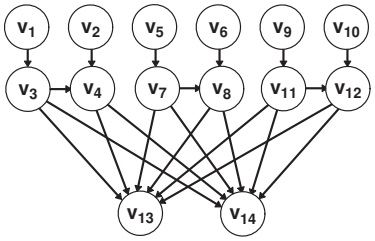


Fig. 3. Example Graph

algorithms, our algorithm uses less computation for edges $\langle v_1, v_3\rangle$, $\langle v_5, v_7\rangle$, $\langle v_9, v_{11}\rangle$, $\langle v_2, v_4\rangle$, $\langle v_6, v_8\rangle$ and $\langle v_{10}, v_{12}\rangle$.

Remark 2. Note that the bitmap hash table cannot be deployed by CF-Hash technique. Clearly, on large graphs we cannot afford to construct $|V|$ bitmap hash tables each of which has size $|V|$. On the other hand, it is time consuming to build H on the fly because, unlike we build the hash table H only once[4] for each vertex in AOT algorithm, H might be built multiple times for a vertex because it's hash table will be chosen (i.e., build on the fly) by CF-Hash algorithm once its hash table size is smaller than that of pivot vertex.

Space Complexity.

We only need to keep the graph G, the oriented graph $\boldsymbol{G}$ and the global hash table H, as a result, Algorithm 3 is space efficient, with space complexity $O(m+n)$ where m is the number of edges and n is the number of vertices in G.

Exploiting Local Order. In addition to the global vertex order, we also consider a local vertex ordering used to store vertices within the scope for each vertex neighborhood list (i.e. *local order*). In Algorithm 3, the dominant cost is the hash table look-ups happen at Lines 7 and 12. There is a good chance that a vertex w will be repeatedly checked because of the overlap of the neighborhood. Ideally, the ID of w should be kept in the CPU cache such that the following look-up of w can be processed efficiently. We may design sophisticate local ordering strategy with some assumptions on the workload such that the neighbors of a (pivot) vertex is well organized by their neighborhood similarity. However, we cannot afford such cost for the preprocessing. In this paper, we order the vertices in the adjacent list of a vertex by the decreasing order of their degree; that is, we visit the vertices at Lines 4 and 9 in Algorithm 3 following the degree order. This is because we believe the vertex with a high degree is more likely to have common neighbors with other vertices. For each vertex, we can keep its neighbors with this local order in the adjacent list. Our empirical study confirms the efficiency of this local order strategy.

4 Experimental Study

Algorithms. To show the efficiency of our proposed technique, we compare our proposed algorithm with the following state-of-the-art methods. In total, we make comparisons between the four algorithms listed below.

[4] When it is chosen as the pivot vertex

Table 2. Statistics of 16 Datasets.

Graph	#Nodes (M)	#Edges (M)	Avg. Degree	Max. Degree	#Triangles (M)
web-baidu-baike	2.14	17.01	8	97,848	25.21
uk-2014-tpd	1.77	15.28	9	63,731	259.04
actor	0.38	15.04	39	3,956	346.81
flicker	1.62	15.48	10	27,236	548.65
uk-2014-host	4.77	40.21	8	726,244	2,509.74
sx-stackoverflow	6.02	28.18	5	44,065	114.21
ljournal-2008	5.36	49.51	9	19,432	411.16
soc-orkut	3.00	106.35	35	27,466	524.64
hollywood-2011	2.18	114.49	53	13,107	7,073.95
indochina-2004	7.41	150.98	20	256,425	60,115.56
soc-sinaweibo	58.66	261.32	4	278,489	212.98
wikipedia_link_en	12.15	288.26	24	962,969	11,686.21
arabic-2005	22.74	553.90	24	3,247	36,895.36
uk-2005	39.46	783.03	20	5,213	21,779.37
it-2004	41.29	1,027.47	25	9,964	48,374.55
twitter-2010	41.65	1,202.51	29	2,997,487	34,824.92

- **CF** [15, 21]. The CF algorithm, presented in Section 2.2.
- **CF-Hash** [15, 21]. A variant of CF, where the intersection of two adjacency lists are implemented by hashing.
- **kClist** [9]. The kClist algorithm for triangle listing, presented in Section 2.3.
- **AOT**. Our proposed algorithm with adaptive orientation and local ordering, presented in Section 3.2.

The source-code for the assessment of CF, $kClist$, and $CF\text{-}Hash$ are acquired from their respective authors. We note that for CF and $CF\text{-}Hash$, we use the implementation from [21] named $TC\text{-}Merge$ and $TC\text{-}Hash$ respectively, due to their more efficient implementations.

Datasets. The datasets used in the experiments are listed in Table 2. We used 16 large real-world graphs with up to a billion edges. Networks are treated as undirected simple graphs, and are processed appropriately.

Settings. The tests are run on a 64 bit Linux machine with a Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz processor, the L1, L2 and L3 cache of 32K, 256K, and 25600K respectively, with 591 GB of available RAM.

4.1 Results against the State-of-the-art

Figure 4 reports the relative running times of the algorithms tested. The measured time captures the elapsed time between the point when the graph is loaded until the point of successful program termination. For the state-of-the-art methods, the kClist algorithm requires fewer running time than the CF algorithm. For datasets containing up to 100 million edges, kClist is observed to significantly outperform CF; this gap in running time is less significant for graphs where the edge count is greater than 100 million. There are also instances where CF is more efficient than kClist, as observed in the social-network *soc-sinaweibo*.

In comparison, our algorithm AOT consistently outperforms the two state-of-the-art, which supports the tightened running bound of

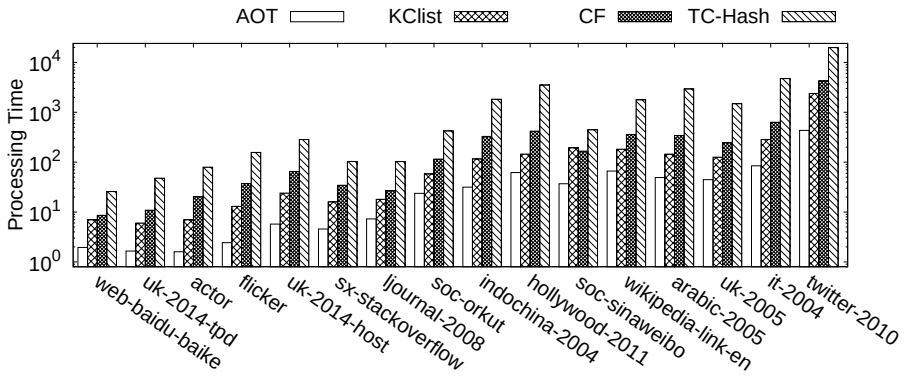


Fig. 4. Performance Analysis

$\sum_{\{u,w\}\in E} min(deg^+(u), deg^+(v))$ from its theoretical analysis. We notice that on a large graph *twitter-2010* that has 41.65 million vertices, 1.2 billion edges and contains 35 billion triangles, the observed running times of kClist and CF are $2,381$ seconds and $4,230$ seconds respectively. In contrast, our algorithm listed all triangles in *twitter-2010* in 433 seconds and achieved a speedup of 10-times. It is noticed that hash-based CF (CF-Hash) is consistently outperformed by AOT with big margins, though two algorithms have the same asymptotic behavior. This is because the high efficiency of look-up operation of the bitmap hash table as well as the local ordering technique in AOT algorithm. Recall that, without the adaptive orientation technique proposed in this paper, hash-based CF cannot take this advantage. This reflects the non-trivial nature of our adaptive orientation technique.

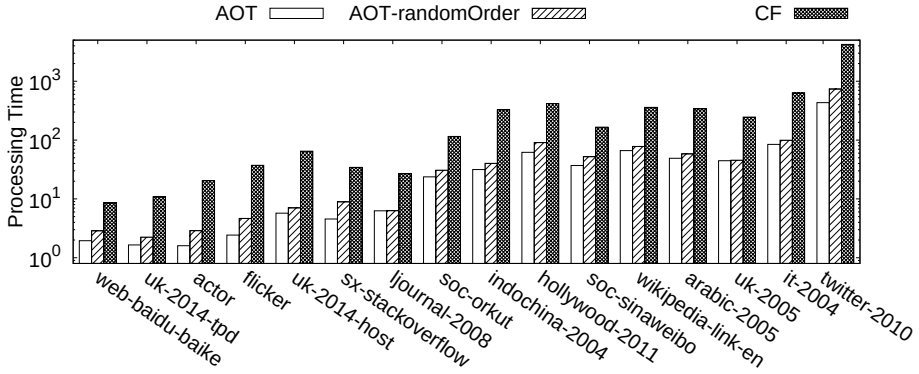


Fig. 5. Incremental Improvements

4.2 More on AOT

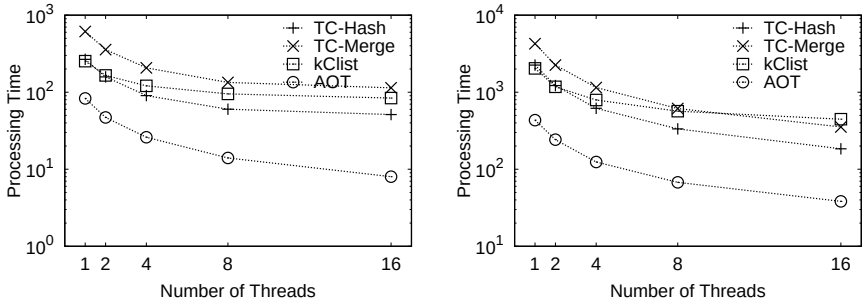
To show the efficiency of our algorithm, we evaluate the benefits for having adaptive orientation and local ordering in our technique. For this setting, a baseline method is one that has neither adaptive orientation or local ordering. We consider CF algorithm as a proper baseline since it uses the existing

orientation technique, but uses neither of the aforementioned techniques. In addition to considering the AOT algorithm with both adaptive orientation and local ordering. We also require an algorithm that uses adaptive orientation without utilizing a local ordering technique, for this, we consider our AOT algorithm with a random local ordering, denoted later as *AOT-randomOrder*.

As we can see in Figure 5, the processing time decreases after introducing adaptive orientation and the local ordering strategy. In comparison to the baseline processing time, the adaptive orientation contributes a greater drop in processing time compared to that from the later adoption of the local-order strategy Where the difference between *AOT-randomOrder* and CF is greater than that between *AOT* and *AOT-randomOrder*. This highlights that our adaptive orientation technique performs better than the orientation technique in its current state. Furthermore, the results also show that using a local order reduces the running time needed on most graphs; this can be explained by an improvement in the algorithms cache performance.

4.3 Parallel Experiments

Our algorithm *AOT* can be easily made parallel. This is achieved by processing vertices in parallel. As a result, we analyze the parallelism of our algorithm and compare it against the state-of-the-art methods. *TC-Merge* (i.e., parallel implementation of *CF* proposed in [15]), *TC-Hash* and *kClist* all provided parallel implementations of their algorithms. For this parallel experiment, we consider the two largest datasets *It*-2004 and *Twitter*-2010.



(a) It-2004 (b) Twitter-2010

Fig. 6. Evaluating Parallel Performance

As seen in Figure 6, increasing the number of threads decreases the amount of processing time needed to list all triangles, this is expected and true for all four algorithms tested. In the case of *kClist*, the drop is less pronounced for both networks after 4 threads. In the case of *TC-Merge* and *TC-Hash*, the drop in processing time is not visible when handling *It*-2004 past 8 threads. In contrast, this decrease is visible for our method *AOT* across both datasets. All in all, *AOT* is consistently the fastest method in this parallel experiments.

5 Related Work

Triangle Listing In-memory algorithms for listing triangles have been extensively studied in the literature. The edge-iterator [1] and node-iterator [13]

are two popular triangle listing computing paradigms, which share the same asymptotic behavior [20]. A lot of subsequent algorithms are mostly improvements based on the original two paradigms. While Ortmann was the first to formalize a generalized framework based on undirected graph orientation, past literature Forward and Compact Forward(CF) had previously considered triangle-listing on induced directed graphs with respect to a vertex ordering [20]. In literature, the orientation technique is observed beyond triangle-listing; it is also applied for higher-order structure enumeration [9]. In more recent years, the topics of interest have shifted to parallel/distributed processing (e.g.,[21, 16]), efficient I/O external memory methods (e.g.,[7, 12], and the asymptotic cost analysis of triangle listing in random graphs [24].

Triangle Counting The triangle counting is a related problem to the triangle listing problem, solving the listing problem solves the counting problem. The triangle counting problem is the task to find the total number of triangles in a graph G. Compared to listing algorithms, counting algorithms find ways to compute the number without relying on the exploration of triangle instances. Many algorithms have been designed to count triangles (e.g., [3, 17, 14]). Approximate methods are useful for settings that handle large-scale graphs, or settings where a given approximation is as useful as knowing the exact triangle count (e.g.,[22, 23]).

6 Conclusion

The triangle listing is a fundamental problem in graph analysis with a wide range of applications. This problem has been extensively studied in the literature. Although many efficient main memory algorithms based on the efficient orientation technique have been proposed, in this paper, we pushed the efficiency boundary of the triangle listing and developed a new algorithm with best theoretical time complexity and practical performance. On the theoretical side, we showed that our proposed algorithm has the time complexity $\Theta(\sum_{\langle u,v\rangle \in \boldsymbol{E}} \min\{deg^+(u), deg^+(v)\}))$ where $\boldsymbol{E}$ is the directed edges in the oriented graph, which is the best known theoretical time complexity for the problem of in-memory triangle listing so far. On the practical side, our comprehensive experiments over 16 real-life large graphs show the superior performance of our AOT algorithm compared with two state-of-the-art techniques, especially on large-scale graphs with billions of edges.

Acknowledgement

Lu Qin is is supported by ARC DP160101513. Ying Zhang is supported by FT170100128 and ARC DP180103096, Wenjie Zhang is supported by ARC DP180103096 and ARC DP200101116. Xuemin Lin is supported by 2018YFB1003504, ARC DP200101338, NSFC61232006, ARC DP180103096 and DP170101628.

References

1. Batagelj, V., Mrvar, A.: A subquadratic triad census algorithm for large sparse networks with small maximum degree. Social Networks **23**(3) (2001)

2. Batagelj, V., Zaveršnik, M.: Generalized cores. arXiv preprint cs/0202039 (2002)
3. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: Proc. of SIGKDD'08 (2008)
4. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient algorithms for large-scale local triangle counting. TKDD **4**(3) (2010)
5. Berry, J.W., Hendrickson, B., LaViolette, R.A., Phillips, C.A.: Tolerating the community detection resolution limit with edge weighting. Physical Review E **83**(5) (2011)
6. Chou, B.H., Suzuki, E.: Discovering community-oriented roles of nodes in a social network. In: Proc. of DaWaK'10 (2010)
7. Chu, S., Cheng, J.: Triangle listing in massive networks and its applications. In: Proc. of KDD'11 (2011)
8. Chu, S., Cheng, J.: Triangle listing in massive networks. TKDD **6**(4) (2012)
9. Danisch, M., Balalau, O., Sozio, M.: Listing k-cliques in sparse real-world graphs. In: Proc. of WWW'18 (2018)
10. Giechaskiel, I., Panagopoulos, G., Yoneki, E.: Pdtl: Parallel and distributed triangle listing for massive graphs. In: Proc. of ICPP'15 (2015)
11. Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: Proc. of SIGMOD'13 (2013)
12. Hu, X., Tao, Y., Chung, C.: I/o-efficient algorithms on triangle listing and counting. ACM Trans. Database Syst. **39**(4) (2014)
13. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM Journal on Computing **7**(4) (1978)
14. Kolda, T.G., Pinar, A., Plantenga, T.D., Seshadhri, C., Task, C.: Counting triangles in massive graphs with mapreduce. SIAM J. Scientific Computing **36**(5) (2014)
15. Latapy, M.: Main-memory triangle computations for very large (sparse (power-law)) graphs. Theor. Comput. Sci. **407**(1-3) (2008)
16. Park, H.M., Myaeng, S.H., Kang, U.: Pte: Enumerating trillion triangles on distributed systems. In: Proc. of SIGKDD'16 (2016)
17. Pavan, A., Tangwongsan, K., Tirthapura, S., Wu, K.: Counting and sampling triangles from a graph stream. PVLDB **6**(14) (2013)
18. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., Parisi, D.: Defining and identifying communities in networks. PNAS **101**(9) (2004)
19. Schank, T.: Algorithmic Aspects of Triangle-Based Network Analysis. Ph.D. thesis, Universitat Karlsruhe (TH) (2007)
20. Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: International workshop on experimental and efficient algorithms. Springer (2005)
21. Shun, J., Tangwongsan, K.: Multicore triangle computations without tuning. In: Proc. of ICDE'15 (2015)
22. Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: DOULION: counting triangles in massive graphs with a coin. In: Proc. of SIGKDD'09 (2009)
23. Türkoglu, D., Turk, A.: Edge-based wedge sampling to estimate triangle counts in very large graphs. In: Proc. of ICDM'17 (2017)
24. Xiao, D., Cui, Y., Cline, D.B., Loguinov, D.: On asymptotic cost of triangle listing in random graphs. In: PODS. pp. 261–272. ACM (2017)
25. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.: Scan: a structural clustering algorithm for networks. In: Proc. of SIGKDD'07 (2007)
26. Yin, H., Benson, A.R., Leskovec, J., Gleich, D.F.: Local higher-order graph clustering. In: Proc. of SIGKDD'07 (2017)
27. Zhang, Y., Parthasarathy, S.: Extracting, analyzing and visualizing triangle k-core motifs within networks. In: Proc. of ICDE'12 (2012)