Efficient Enumeration of Four Node Graphlets at Trillion-Scale

Yudi Santoso
University of Victoria
BC, Canada
santoso@uvic.ca

Venkatesh Srinivasan
University of Victoria
BC, Canada
srinivas@uvic.ca

Alex Thomo
University of Victoria
BC, Canada
thomo@uvic.ca

Figure 1: Four node graphlets: a 3-path, a 3-star, a rectangle or 4-cycle, a tailed-triangle, a diamond, and a 4-clique.

ABSTRACT
Graphlet enumeration is known to be a challenging task in graph analysis. This is because the cost is exponential in the order of the graphlet. Triangle is a graphlet of order three that has received special attention because it is relatively small but non-trivial, and can still be enumerated quite fast even for massive graphs of millions of nodes and edges. In this paper, we propose an efficient algorithm for enumerating four node graphlets, such as 4-cycles, 4-cliques, diamonds, etc by leveraging the most efficient algorithm for triangle enumeration. We show that despite the belief that any such enumeration algorithm cannot terminate in reasonable time, our method can handle large graphs containing trillions of such graphlets, using a single commodity machine, within a reasonable amount of time.

1 INTRODUCTION
It is commonly thought that graphlets, beyond three nodes, are difficult to enumerate. This is because the number of possible instances grows as $O(n^k)$, where $k$ is the order of the graphlet and $n$ is the order of the graph. Thus, for massive graphs, it was believed that an enumeration algorithm, which has to touch each graphlet, cannot terminate in a reasonable time [11]. Indeed, previous methods, such as Fanmod [18] and Rage [8], do not scale well and take a very long time to run on million scale graphs. Other proposed solutions, such as Arabesque [17] and PGD [1] use distributed platforms. However, our focus is to explore the limits of what can be achieved using single-machine algorithms.

There are several algorithms proposed in the literature to count the number of the graphlets. They are either estimates using approximation methods, such as Graff [13], or exact counting without full enumeration, notably Orca [6] and Escape [11]. However, what if we need to find each of the graphlet instances? Knowing where the graphlets are is useful in analysing the local structures of the graph. For example, enumerating graphlets is important in detecting cancer through differential graphlet communities [19]. Also, enumeration can yield graphlet degree counts which are useful for uncovering biological network functions [7].

It is worth noting that there have been plenty of studies on triangle enumeration. It was found that triangles can be enumerated quite efficiently using the compact forward edge-iterator algorithm [7]. In general, graphlets of order $k$ can be enumerated using an algorithm with runtime $O(n^{d-1})$ where $d$ is the maximum degree [15]. However, in [11] it was shown that through careful preprocessing, triangle enumeration using edge-iterator can be significantly faster than $O(n^d)$ time. Can we achieve a better runtime than $O(n^{d-1})$, for higher order graphlets?

In this paper we show that efficient enumeration for triangles can be leveraged to enumerate higher order graphlets, in particular four node graphlets. Our algorithm achieves a significantly improved runtime, which depends on the number of three-node graphlets and is able to handle large graphs efficiently on a single machine. Moreover, unlike most in the literature, our solution yields the counts of all four node graphlets in a single run.

Our contributions are as follows:

(1) We propose a new algorithm to enumerate all types of four node graphlets of an undirected graph on a single run. Enumeration is done carefully so that no graphlet is listed more than once.

(2) We provide detailed analyses on the algorithm correctness and time complexity. We refine the time upper-bound of enumeration to depend on the number of three-node graphlets and thus be significantly better than $O(n^d)$ for real-world networks.

(3) We create an efficient implementation of the algorithm for a single machine. Our algorithm is able to run on graphs of millions nodes and edges, which contain trillions of graphlets, within reasonable time.

2 RELATED WORK
Chiba and Nishizeki published several subgraph listing algorithms [5] which can be considered as a pioneering work on graphlet enumeration. Milo et al. [16] analysed frequent subgraph patterns, and called them network motifs. Since then, there have been many studies on how to find and count small subgraphs within a graph or network, including those we already discussed in the Introduction. Also, Silvestri [16] provided another complexity analysis on subgraph enumeration. To the best of our knowledge, there has not been a solution using the method that we propose here, to simultaneously, and fully, enumerate all the graphlets (of order four) through triangles and wedges, and that can scale to large graphs using a single machine.

3 PRELIMINARIES
In this paper we solely work on simple undirected graphs. We denote a graph by $G(V, E)$ where $V$ is the set of nodes and $E$ is the set of edges. Let $n = |V|$ and $m = |E|$. The degree of a node is the number of edges incident on it. For simple graphs, there is no self-loop and the degree is equal to the number of neighbours. We
Algorithm 1 Triangle Enumeration

Input: An undirected graph $G(V, E)$ in an adjacency list representation 
1: for all vertex $u \in V(G)$ do 
2: for all vertex $v \in N(u)$ do 
3: if $v > u$ then 
4: for all $w \in N(u) \cap N(v)$ do 
5: if $w > v$ then 
6: EnumerateTriangle $(u, v, w)$

denote the set of neighbours of node $u$ by $N(u)$, and the degree of $u$ by $d(u) = |N(u)|$. A subgraph of $G(V(G), E(G))$ is a graph $H(V(H), E(H))$ such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We use the notation $H \subseteq G$ to say that $H$ is a subgraph of $G$. A subgraph $H \subseteq G$ is an induced subgraph if any edge $(u, v)$ (which is $(v, u)$ for undirected graphs), with $u, v \in V(H)$, is in $E(H)$ if and only if $(u, v)$ is in $E(G)$. A subgraph is connected if every pair of nodes in it is connected by a path of edges. We assume the following definition: A graphlet is an induced connected subgraph.

There are two kinds of graphlets of three nodes: wedge (Henceforth labeled as $g_1$) and triangle ($g_2$). Note that for induced subgraphs, wedges and triangles cannot be on top of each other (i.e., a wedge and a triangle cannot have the same set of three nodes within the same graph). For four nodes, we have six types of graphlets, depicted in Figure 1. These are 3-path (or four-node-path) ($g_3$), 3-star ($g_4$), 4-cycle or rectangle ($g_5$), tailed-triangle ($g_6$), diamond ($g_7$), and 4-clique ($g_8$). The labels that we use here follow the common labelling used by some papers in the literature [2, 12].

Notice that $g_6$, $g_7$, and $g_8$ contain triangle(s). A $g_7$ contains one triangle, a $g_8$ contains two triangles, and a $g_8$ contains four triangles. This fact suggests that we can find them through the triangles in the graph. Whenever we find a triangle, we can check if this triangle is a part of any $g_6$, $g_7$, and/or $g_8$. Similarly, $g_3$, $g_4$, and $g_5$ contain two, three, and four wedges, respectively. Therefore, we can find them through wedges.

We list graphlets by their nodes. Thus, for example, $(u, v, w, z)$ is a $g_6$ with nodes $u, v, w, z$. In enumeration, some care is needed to avoid multiple listing. Without loss of generality, we can use label 1, 2, 3, 4 to represent the nodes in a graphlet. Clearly, $1 < 2 < 3 < 4$, so 1 represents the smallest node.

There are 3! = 6 permutations of three nodes. Therefore, for wedges, we have 1, 2, 3, 1, 2, 3, 1, 3, 2, 1, 3, 2, 1, and 3, 1, 2, 1, 3, 2, 1 is the same as (3, 2, 1), and (2, 1, 3) is the same as (3, 2, 1). Thus, we have only three possible wedges, only one can be present (for induced case). Our convention is to list with the smaller leg first, i.e. (1, 2, 3), (1, 3, 2), and (2, 1, 3). We can divide these into two types: those with the smallest node at the center of the wedge (type 1), i.e., (1, 2, 3), and those with the smallest node at one of the legs (type 2), i.e., (1, 2, 3) and (3, 1, 2). We will see that they require separate treatment. For triangles, all six permutations are isomorphic. Therefore, we only need one to list. We choose the one with the nodes ordered ascendingly: (1, 2, 3, 2).

Now for four nodes, there are 4! = 24 permutations. For 3-paths, by symmetry we only need half (i.e. twelve) of them. For 3-stars, we have four distinct ones depending on which one is the centre. For 4-cycles, the cyclic symmetry gives us a factor of four, while the clockwise counter-clockwise symmetry gives us a factor of two. Therefore, we have only 24/8 = 3 distinct permutations. For tailed-triangles, the distinguishing nodes are the end node and the centre node, giving us 3! = 6 distinct configurations. For diamonds, we have a pair of triangles. Let us call the two end nodes of the shared edge as the connecting nodes, and the other two nodes as the opposing nodes. There are symmetries between the two opposing nodes, and between the two connecting nodes, giving us 24/2/2 = 6 distinct configurations. For 4-cliques, we can exchange any pair of nodes and get the same clique. Thus there is only one unique configuration, and we choose to list the nodes in order: (1, 2, 3, 4).

4 THE ALGORITHMS

The algorithm that we use for triangle enumeration is an edge iterator algorithm (Algorithm 1) combined with nodes ordering. This combination is similar to the Compact-Forward algorithm [7] but with the ordering done in a pre-processing before the enumeration (Algorithm 2).

Algorithm 2 Graph-Prep

Input: An undirected graph $G(V, E)$ 
1: Sort $V$ based on the degrees, in ascending order. 
2: Relabel the vertices according to their new order. 
3: Build adjacency list of the sorted and relabeled vertices. 
4: Cut out the smaller neighbours from each neighbour list.

Algorithm 3 Triangle and Wedge Enumeration

Input: An undirected graph $G(V, E)$ in an adjacency list representation 
1: for all vertex $u \in V(G)$ do 
2: for all vertex $v \in N(u)$ do 
3: if $u < v$ then 
4: for all $u' \in N(u)$ and $v' \in N(v)$ do 
5: if $(u' > u) \land (v' > v) \text{ then}$ 
6: if $u' = u' > v' \text{ then}$ 
7: EnumerateTriangle $(u, v, u')$ 
8: if $(u' < u') \land (v' > v') \text{ then}$ 
9: EnumerateWedgeType2 $(v, u, u')$ 
10: if $u' > v' \text{ then}$ 
11: EnumerateWedgeType2 $(u, v, u')$

The graph preprocessing is based on the following observations: (i) Because of lines 3 and 5 of Algorithm 1 we need to consider only bigger neighbours of every vertex, i.e., $N^>(u) = \{v \in N(u)\mid v > u\}$. (ii) The triangle count in a graph will not change if we relabel the vertices.

Algorithm 1 can be modified to enumerate the wedges as well. This is shown in Algorithm 3. Notice that by condition on line 3 we assure that $u$ is always smaller than $v$. To avoid multiple listing, when we iterate neighbours of $v$ we consider only those that are bigger than $u$ (line 5). However, we need to include smaller neighbours of $v$ (i.e. those between $u$ and $v$) to catch all of the wedges.

We extend each of the EnumerateTriangle and EnumerateWedge functions above to search for four-node graphlets. Whenever we find a triangle, $(u, v, w)$, we call the ExploreTriangle function (Algorithm 4), which checks for the intersections among the neighbour sets of the three triangle nodes, $N(u), N(v)$ and $N(w)$. If we find a $z \in N(u) \cap N(v) \cap N(w)$, then $(u, v, w, z)$...
Algorithm 4 Explore Triangle

Input: Given triangle \((u, v, w)\), \(u < v < w\): \(N(u), N(v), N(w)\).
1: Compute intersections among the three neighbour sets.
2: for all \(z \in N(u) \cap N(v) \cap N(w)\) with \(z > w\) do
3: Enumerate4Clique \((u, v, w, z)\)
4: for all \(z\) in two sets and \(z > \) opposite node do
5: EnumerateDiamond \((.)\)
6: for all \(z\) in one set only do
7: EnumerateTailedTriangle \((.)\)

is a four-clique (i.e. \(g_4\)). A node \(z\) that is in two of the three neighbour sets gives us a diamond (i.e. \(g_4\)), while a \(z\) that is in only one of the three neighbour sets gives us a tailed triangle (i.e. \(g_6\)). For 4-cliques, we can use the sets of larger neighbours. For diamonds, we can use the sets of neighbours larger than \(u\). For the tailed triangles, however, we need to include all of the neighbours. Due to this last case we lost some of the advantage of the graph preprocessing. As a result, the runtime might be much longer compared to the triangle enumeration time, depending on the maximum degree.

Algorithm 5 Explore Wedge Type-1

Input: Given wedge \((u, v, w)\), \(u < v < w\): \(N^>(u), N^>(v), N^>(w)\).
1: Compute intersections among the three neighbour sets.
2: for all \(z \in N^>(u) \cap N^>(v) \cap N^>(w)\) with \(z \notin N^>(u)\) do
3: EnumerateRectangle \((u, v, z, w)\)
4: for all \(z \in N^>(u)\) only do
5: if \(z > w\) then
6: Enumerate3Star \((u, v, z)\)
7: for all \(z \in N^>(w)\) only do
8: Enumerate3Path \((w, u, z)\)
9: for all \(z \in N^>(w)\) only do
10: EnumeratePath \((u, v, w)\)

For the wedges, we call two different functions depending on the type of the wedge, either Algorithm 5 or 6. In this two functions we only need sets of neighbours that are larger than \(u\), but this is not done in a pre-processing. Notice that in Algorithm 6 \(w\) can be smaller than \(v\), which is the center of the wedge.

Algorithm 6 Explore Wedge Type-2

Input: Given wedge \((u, v, w)\), \(u < v < w\): \(N^>(u), N^>(v), N^>(w)\).
1: Compute intersections among the three neighbour sets.
2: for all \(z \in N^>(u)\) only do
3: if \(z > w\) then
4: Enumerate3Star \((u, v, w)\)
5: for all \(z \in N^>(w)\) only do
6: if \(z < v\) then
7: Enumerate3Path \((u, v, w)\)

5 ANALYSIS

Theorem 1. Algorithm 3 correctly enumerates wedges and triangles in an undirected graph.

Proof. Each edge \((u, v)\), with \(u < v\), is iterated once and only once. For each, we enumerate all the intersecting neighbours (i.e., triangles), and non-intersecting neighbours (i.e., wedges). Thus, all wedges and triangles in the graph would be found.

For triangles, we avoid multiple listing by imposing condition \(u' = v' > v\). For type-1 wedges we impose condition \(u' > v\). For type-2 wedges, since \(u < v'\) there will be no double counting. □

Theorem 2. Algorithms 4, 5 and 6, combined with algorithm 3, correctly enumerate all four node graphlets in an undirected graph.

Proof. As proven above, all triangles and wedges are enumerated once. For each triangle, the three neighbour sets are checked. Each node that is in only one of the sets yields a tailed-triangle. All tails would be found in the sets. A node that is in the intersection of two sets yields a diamond. By asserting that this node is larger than the opposite node in the diamond we assure that any diamond would be listed just once. A node that is in the intersection of all three sets yields a 4-clique. We assert that this node is larger than any node in the triangle to assure that the clique has not been listed in any previous iteration. For wedges, similarly, all four node graphlets attached to each wedge would be found. Multiple listing is avoided by considering only 3-paths, 3-stars and 4-cycles, and by careful conditions on the node ordering. For the 3-paths we make sure that the smallest node is always in the first half of the path. For the 3-stars we make sure that the fourth node is greater than the third node. The center node does not need to be the smallest. For 4-cycles we make sure that the fourth node is opposite to the first node. □

Theorem 3. The runtime of the four node graphlet enumeration is bounded by \(O(N_4 + N_3) \cdot d_{\text{max}} + T_{\text{prep}}\), where \(N_4\) is the number of triangles (wedges), and \(T_{\text{prep}}\) is the time to enumerate triangles and wedges.

Proof. For each triangle and wedge the algorithm runs through the neighbor sets to check the intersections with cost \(\leq (d(u) + d(v) + d(w))\). □

Note that in general \(N_4 + N_3 \leq n d_{\text{max}}^4\). The upper value is satisfied by a regular graph. However, for all real world networks, we have \(N_4 + N_3 \approx n d_{\text{max}}^2\). Also, \(T_{\text{prep}} \approx n d_{\text{max}}^2\) using efficient enumeration. Therefore, in practice, our runtime is much less than worst case bound of \(O(n d_{\text{max}}^4)\).

6 EXPERIMENTS

The networks that we study are listed in Table 2. All of the datasets were downloaded from the Laboratory for Web Algorithms [3, 4], http://law.di.unimi.it/datasets.php. We symmetrized them and got rid of any loops to get simple undirected graphs. We implemented our code in Java with parallel streams, and used Webgraph library [4]. We used a Linux machine with dual Xeon E5-2620 processors of 24 threads and 128 GB of RAM. We notice, however, that the memory usage is \(< 1\) GB throughout the experiment.

The graphlet counts are listed in Table 1. We can check that for all of these graphs, \(N_3 + N_2 \approx 42M\) and \(n d_{\text{max}}^2 \approx 853B\) a four order of magnitude difference. For all of the graphs that we consider here the difference is from three to five orders of magnitude.

The runtimes are shown in Table 3. We include the triangle enumeration time, \(T_3\), for comparison. As wedges cannot take full advantage of the pre-processing, they take longer time for enumeration, hence \(T_3\) is larger than \(T_3\). Notice that the preprocessing time, \(T_{\text{prep}}\), is just about the same magnitude as \(T_3\).
Table 1: Counts of the graphlets. The dewiki dataset needs longer than our time limit to terminate.

<table>
<thead>
<tr>
<th>Graph</th>
<th>(d_1)</th>
<th>(d_2)</th>
<th>(d_3)</th>
<th>(d_4)</th>
<th>(d_5)</th>
<th>(g_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enron</td>
<td>4,039,453</td>
<td>1,067,993</td>
<td>2,511,039,670</td>
<td>8,043,808,283</td>
<td>21,598,984</td>
<td>582,841,848</td>
</tr>
<tr>
<td>cnr</td>
<td>7,798,287,209</td>
<td>20,977,629</td>
<td>6,118,026,632</td>
<td>41,394,015,937,553</td>
<td>37,842,334,745</td>
<td>42,974,515,602</td>
</tr>
<tr>
<td>dblp</td>
<td>81,529,950</td>
<td>7,005,235</td>
<td>2,678,518,695</td>
<td>3,545,925,764</td>
<td>1,485,611</td>
<td>543,447,587</td>
</tr>
<tr>
<td>amazon</td>
<td>38,015,403</td>
<td>4,464,791</td>
<td>372,366,885</td>
<td>609,961,827</td>
<td>2,689,695</td>
<td>9,232,707</td>
</tr>
<tr>
<td>dewiki</td>
<td>51,141,107,679</td>
<td>681,128,282</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ljournal</td>
<td>8,726,048,197</td>
<td>411,155,444</td>
<td>1,812,284,632,329</td>
<td>8,847,128,736,944</td>
<td>8,551,292,956</td>
<td>189,716,360,703</td>
</tr>
</tbody>
</table>

Note that \(T_{d4}\), the time required to enumerate all 3 and 4-node graphlets, does not strongly depend on the size of the graph, but rather on the degrees and the numbers of triangles and wedges, validating our analysis. For example, comparing ljournal with amazon, the ratio of their \((N_L + N_R) d_{max}\) values is about four thousand, while the ratio of their \(T_{d4}\) values is about six thousand, i.e. approximately the same order. This observation experimentally validates the statement of Theorem 3 relating the runtime to the \((N_L + N_R) d_{max}\) value.

Interestingly, cnr requires longer runtime than ljournal. Even though it is smaller by an order of magnitude it has more graphlets. The amazon dataset, which has relatively small maximum degree can be processed in merely 14 seconds. The dewiki dataset has enormous number of wedges and large maximum degree and the algorithm did not terminate even after running for four days.

Table 2: The undirected graphs. Here, \(d_{BG_{max}}\) is the effective maximum degree when only larger neighbours are included after the preprocessing.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>(n)</th>
<th>(m)</th>
<th>(d_{max})</th>
<th>(d_{BG_{max}})</th>
<th>(d_{avg})</th>
</tr>
</thead>
<tbody>
<tr>
<td>enron</td>
<td>69,244</td>
<td>254,449</td>
<td>1,634</td>
<td>87</td>
<td>7.35</td>
</tr>
<tr>
<td>cnr</td>
<td>325,557</td>
<td>2,738,969</td>
<td>18,236</td>
<td>85</td>
<td>16.83</td>
</tr>
<tr>
<td>dblp</td>
<td>986,324</td>
<td>3,353,618</td>
<td>979</td>
<td>118</td>
<td>6.80</td>
</tr>
<tr>
<td>amazon</td>
<td>735,323</td>
<td>3,523,472</td>
<td>1,077</td>
<td>16</td>
<td>9.58</td>
</tr>
<tr>
<td>dewiki</td>
<td>1,532,354</td>
<td>33,093,029</td>
<td>490</td>
<td>43.19</td>
<td></td>
</tr>
<tr>
<td>ljournal</td>
<td>5,563,260</td>
<td>49,514,271</td>
<td>19,432</td>
<td>756</td>
<td>18.46</td>
</tr>
</tbody>
</table>

Table 3: The runtime, in seconds, for preprocessing, for triangle enumeration, for wedges and triangles together, and for all three and four node graphlets together.

<table>
<thead>
<tr>
<th>Graph</th>
<th>(T_{Prep})</th>
<th>(T_{A})</th>
<th>(T_{3g})</th>
<th>(T_{4g})</th>
</tr>
</thead>
<tbody>
<tr>
<td>enron</td>
<td>0.87</td>
<td>1.03</td>
<td>3.49</td>
<td>76.50</td>
</tr>
<tr>
<td>cnr</td>
<td>1.93</td>
<td>1.75</td>
<td>57.03</td>
<td>176K</td>
</tr>
<tr>
<td>dblp</td>
<td>4.87</td>
<td>1.93</td>
<td>3.45</td>
<td>62.05</td>
</tr>
<tr>
<td>amazon</td>
<td>5.80</td>
<td>1.75</td>
<td>2.53</td>
<td>14.05</td>
</tr>
<tr>
<td>dewiki</td>
<td>26.45</td>
<td>12.79</td>
<td>517.3</td>
<td>300K</td>
</tr>
<tr>
<td>ljournal</td>
<td>46.68</td>
<td>32.96</td>
<td>257.1</td>
<td>82K</td>
</tr>
</tbody>
</table>

7 CONCLUSIONS

In this study we have shown that it is possible to enumerate all types of four node graphlets simultaneously with runtime \(O((N_L + N_R) d_{max} + T_{3g})\). Wedges and triangles can be enumerated relatively fast (in a pre-run) and the result can be used to estimate the time needed to enumerate the four node graphlets. We found that the runtime upper bound depends more on the maximum degree than on the size of the graph. Our algorithm can finish the enumeration in seconds when the maximum degree is around 1K. Moreover, it does not require large memory space, and it would run for even larger graphs (provided that we allow enough time). Notably, we were able to process massive graphs of millions of nodes and edges and enumerate about 40 trillions graphlets in a single run, within a reasonable amount of time.

REFERENCES


