Mining Small Subgraphs in Massive Graphs

by

Yudi Santoso

B.Sc. (Physics), Gadjah Mada University, Indonesia, 1993

M.Sc. (Physics), Bandung Institute of Technology, Indonesia, 1996

Ph.D. (Physics), Texas A&M University, Texas, USA, 2001

M.Sc. (Computer Science), University of Victoria, Victoria, BC, Canada, 2018

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Yudi Santoso, 2023

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Mining Small Subgraphs in Massive Graphs

by

Yudi Santoso

B.Sc. (Physics), Gadjah Mada University, Indonesia, 1993

M.Sc. (Physics), Bandung Institute of Technology, Indonesia, 1996

Ph.D. (Physics), Texas A&M University, Texas, USA, 2001

M.Sc. (Computer Science), University of Victoria, Victoria, BC, Canada, 2018

Supervisory Committee

Dr. Alex Thomo, Co-supervisor

(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-supervisor

(Department of Computer Science)

Dr. Xuekui Zhang, Outside Member

(Department of Mathematics and Statistics)

ABSTRACT

Graph or network analysis is a much needed method of analysis as it can reveal some insights that will not be obvious through other methods. In a graph, entities are represented by nodes or vertices, and relations or connections among the entities are represented by edges. By analysing a graph we can get invaluable information on how a system works and on how one part of the system is related to the others. Many graph analytical problems require that we find and locate all subgraphs of specific patterns within a given graph. This task is not trivial when we are dealing with massive graphs, of millions or even billions of nodes and edges. In particular, it becomes harder when we want to get it done within a limited time, and with a limited amount of computational resources. In this dissertation, we focus on building efficient algorithms and methods to enumerate graphlets, or small connected induced subgraphs, for both undirected and directed graphs. With our solutions we are able to enumerate up to 5-node graphlets in some massive graphs by using only a single commodity machine, producing trillions of graphlets.

Contents

Supervisory Committee ii

Abstract iii

Contents iv

List of Tables viii

List of Figures xii

Acknowledgements xvi

Dedication xvii

1 Introduction 1

2 Theoretical Background 6

2.1 Graphs and Digraphs 6

2.2 Graph Representations 10

2.3 Subgraphs and Subgraph Patterns 12

2.4 Wedges, Triangles, and Triads 14

2.5 Cliques 19

2.6 Graphlets 21

2.7 Orbits and GDV 22

2.8 Induced and Non-Induced Subgraphs 24

3 Related Works **26**

3.1 Triangles 26

3.2 Triads 26

3.3 Graphlets 27

3.4 Cliques 28

3.5 Distributed Enumeration 28

4 Undirected Graphlet Enumeration **30**

4.1 Triangles 30

4.1.1 Algorithms 31

4.1.2 Analysis 33

4.1.3 Experiment 34

4.2 Four-Node-Graphlets 35

4.2.1 Algorithm 38

4.2.2 Analysis 39

4.2.3 Experiment 41

4.3 Five-Node Graphlets 44

4.3.1 Idea 45

4.3.2 Algorithm 53

4.3.3 Analysis 55

4.3.4 Experiment 57

4.4 GDV 59

4.5 Beyond 5 Nodes 59

5 Directed Graphlets Enumeration **61**

5.1 Directed Triangles 61

5.1.1 Algorithms 62
5.1.2 Analysis 63
5.1.3 Experiment 65
5.2 Triads 66
5.2.1 Algorithm 68
5.2.2 Experiment 71
5.3 Directed Graphlets 73

6 Distributed Enumeration 75

6.1 Graph Partition and Subproblems 75
6.2 Distributed Four-node Graphlet Enumeration 77
6.2.1 Previous Distributed Enumeration 77
6.2.2 Generalized Color-Direction 78
6.2.3 S4GE with Color Direction 82
6.2.4 Compact-Forward for 4-clique listing 83
6.2.5 Analysis 85
6.2.6 Experiment 89
6.2.7 Discussion 97
6.3 Distributed Triad Enumeration 99
6.3.1 Experiment 101

7 Future Work 105

7.1 Larger Directed Graphlets 106
7.2 5-node Graphlet Enumeration 106
7.3 Larger Order Graphlets 106
7.4 Probabilistic Graphlets 107
7.5 Typed and Labeled Graphs 108

8 Conclusion **110**

Bibliography **112**

A Graph Datasets **123**

A.1 Directed Graph Datasets 123

A.2 Undirected Graph Datasets 124

B Algorithms for 5-node Graphlet Enumeration **126**

C Directed 4-node Graphlets **140**

C.1 Directed 3-path 140

C.2 Directed 3-star 141

C.3 Directed 4-cycle 142

C.4 Directed tailed-triangle 142

C.5 Directed diamond 143

C.6 Directed 4-clique 144

List of Tables

Table 2.1 Our notation on graphs and digraphs. 7

Table 2.2 Our notation on triangles and triads. 19

Table 2.3 Triad types and description. 19

Table 4.1 The undirected graphs. Here, $d_{\max}^{\mathrm{BG}}$ is the effective maximum degree when only larger neighbours are included after the preprocessing. 42

Table 4.2 Counts of the graphlets. The `dewiki` dataset needs longer than our time limit to terminate. 43

Table 4.3 The runtime, in seconds, for triangle enumeration T_Δ, for wedges and triangles together T_{3g}, and for all three and four-node graphlets together T_{4g}. 45

Table 4.4 The 5GT functions. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions. 54

Table 4.5 The 5GW1 functions. The input is a type-1 wedge with the smallest node u at the center of the wedge. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions. 55

Table 4.6 The 5GW2 functions. The input is a type-2 wedge with the smallest node u at one of the legs of the wedge, and the center node is v. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions. 56

Table 4.7 The runtime, in seconds, for S4GE T_{4g}, and S5GE T_{5g}. As a reference, we also list the maximum degree of each graph. 57

Table 4.8 Graphlet counts, output of S5GE. 58

Table 5.1 Link encoding using two binary digits. We assume that the first node is on the left and the second one on the right. 67

Table 5.2 Triad types and binary encoding. 70

Table 5.3 Maximum degrees before and after the preprocessing. The degrees after are listed as d^{eff}. 72

Table 5.4 The counts of triads of each type on the selected graphs. 72

Table 5.5 The running time (in seconds) of triad enumeration using FPTE algorithm, and the preprocessing time. 73

Table 5.6 Four-node directed graphlets. 74

Table 6.1 The numbers of vertices n, edges m, wedges $|\angle|$, and triangles $|\Delta|$, and the max degree of the symmetrized graphs. The last three graphs are the largest and they require more computing power than the others. 89

Table 6.2 The enumeration time (minutes) of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, 120 workers, against S4GE (single machine) with 24 threads. . . 93

Table 6.3 The enumeration time (minutes) of D4GE/S4GE$_{\text{CD}}$ against PSE/S4GE, with $\rho = 16$, 120 workers. 94

Table 6.4 Enumeration time (minutes) of D4GE/CF4$_{\text{CD}}$ against PSE/VF2, with $\rho = 16$ 94

Table 6.5 Duplicated emissions from PSE partitioning scheme with different local algorithms. 95

Table 6.6 The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, on a cluster of 120 workers. 96

Table 6.7 The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, on a cluster of 120 workers. 97

Table 6.8 The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 25$, on a cluster of 672 workers. 97

Table 6.9 The numbers of vertices n, edges m, maximum degree of the original graph $d_{\max}$ and its transpose $d^T_{\max}$, and the effective maximum degrees after the preprocessing, $d^{\text{eff}}_{\max}$ and $d^{T\text{eff}}_{\max}$, of the graph datasets. 101

Table 6.10 The enumeration time (seconds) of D3GE/FPTE$_{\text{CD}}$ with $\rho = 12$ and 128 workers, against original FPTE with 16 threads on a single machine. 102

Table A.1 Properties of the directed graphs. 124

Table A.2 Properties of the undirected graphs. Note that $d_{\text{avg}} = 2|E|/|V|$. 125

List of Figures

Figure 2.1 The process of symmetrizing a directed graph to get an undirected graph. (a) The graph $\overset{\leftrightarrow}{G}$. (b) The transpose graph $\overset{\leftrightarrow}{G}^T$. (c) The union graph $\overset{\leftrightarrow}{G} \cup \overset{\leftrightarrow}{G}^T$. (d) The corresponding underlying undirected graph $\widetilde{G}$. 9

Figure 2.2 A simple graph of 5 nodes and 6 edges. 11

Figure 2.3 (a) A graph, (b) an induced subgraph (induced by vertex set $\{1, 2, 3, 6\}$) and (c) an edge-induced subgraph (induced by edge set {1-2, 2-7, 5-6}). Note that (c) is not an induced subgraph of (a). 13

Figure 2.4 Using node 1 as the reference node, we have two types of wedges: Type 1 wedge $(2, 1, 3)$ (a) and Type 2 wedges $(1, 2, 3)$ and $(1, 3, 2)$ (b and c). 14

Figure 2.5 A triangle, $(1, 2, 3)_\Delta$. 15

Figure 2.6 (a) A cycle triangle, $(1, 2, 3)_C$ and (b) a trust triangle, $(1, 2, 3)_T$. 16

Figure 2.7 Sixteen types of triads. 16

Figure 2.8 Seven types of (triangle) triads. 17

Figure 2.9 There are six trust triangles in a Type 7 triad: $(1, 2, 3)_T$, $(1, 3, 2)_T$, $(2, 3, 1)_T$, $(2, 1, 3)_T$, $(3, 1, 2)_T$, and $(3, 2, 1)_T$ respectively. 18

Figure 2.10 There are two cycle triangles in a Type 7 triad: labeled as $(1, 2, 3)_C$ and $(1, 3, 2)_C$ respectively. 18

Figure 2.11 A 3-clique, a 4-clique, a 5-clique and a 6-clique. 20

Figure 2.12 Four node graphlets: a 3-path (g_3), a 3-star (g_4), a rectangle or 4-cycle (g_5), a tailed-triangle (g_6), a diamond (g_7), and a 4-clique (g_8). 21

Figure 2.13 Five node graphlets. 22

Figure 2.14 Orbits for 2, 3, 4, and 5 node graphlets. The labeling is following the convention used by Przulj [52]. 23

Figure 4.1 For diamonds we start with the smaller opposing node and end with the other opposing node. In between we list the two connecting nodes in order. For example, in the diamond shown above 1 and 3 are the opposing nodes, and 2 and 4 are the connecting nodes. Therefore, it is listed as $(1, 2, 4, 3)_7$. 37

Figure 4.2 Finding g_{12} and g_{21} through a triangle $\Delta(u, v, w)$. Here $x \in N_1(u)$ and $y \in N_1(v)$. We get a g_{12} when x and y are not connected, or g_{21} when x and y are connected. 46

Figure 4.3 Finding a g_{13} through a wedge $\angle(u, v, w)$. Here $x \in N_1(u)$ and $y \in N_2(v, w)$, and x and y are not connected. 47

Figure 4.4 There are three ways leading to g_{26}: (a) N_2 with itself, connected; (b) N_2 and N_3, not connected; (c) N_1 and N_3, connected. . . . 48

Figure 4.5 The symmetry of g_{26}. 49

Figure 4.6 Listing all possible combinations in g_{26}. The two yellow nodes are the two highest labeled nodes, 4 and 5. In (c1), (c2), and (c2), we allow the node at the top of the figure to be either higher or lower than the others. 49

Figure 4.7 There are two ways leading to g_{24}: (a) two N_2, not connected; (b) N_1 and N_3, connected. 50

Figure 4.8 There are two ways leading to g_{28}: (a) N_2 with N_3, connected; (b) N_3 with itself, not connected. 51

Figure 4.9 There are two ways of finding a g_{13} through a wedge (green nodes): (a) $N_1(v)$ with $N_2(u, w)$, not connected, and (b) $N_1(v)$ with itself, connected. For the sake of notation, here we assume u is the center node of the green wedge. 51

Figure 4.10 There are two ways of finding a g_{10} through a wedge (green nodes): (a) $N_1(v)$ with $N_1(u)$, not connected, and (b) $N_1(v)$ with itself, also not connected. For the sake of notation, here we assume u is the center node of the green wedge. 52

Figure 4.11 There are three ways of finding a g_{16} through a wedge (green nodes): (a) $N_1(v)$ with $N_1(u)$, connected, (b) $N_1(u)$ with $N_2(v, w)$, not connected, and (c) $N_1(v)$ with $N_2(v, w)$, not connected. For the sake of notation, here we assume u is the center node of the wedge. 52

Figure 4.12 There are two ways of finding a g_{20} through a wedge (green nodes): (a) $N_1(u)$ with $N_2(v, w)$, connected, and (b) $N_2(v, w)$ with $N_2(v, w)$, not connected. For the sake of notation, here we assume u is the center node of the wedge. 53

Figure 4.13 Finding 6-node graphlets through a triangle. Here we take $N_1(u)$, $N_1(v)$ and $N_1(w)$. We get four types of 6-node graphlets depending on whether we have zero, one, two, or three edges among the neighbour nodes. 60

Figure 5.1 A trust triangle, $(1, 2, 3)_T$ and the labeling of its edges. 62

Figure 5.2 Four pointers 70

Figure 6.1 A graph illustrating the need for symmetrization. 81
Figure 6.2 The enumeration time (minutes) of D4GE/S4GE$_{\text{CD}}$ on several graphs, with varying value of ρ. Higher ρ does not add much overhead; the lines flatten out rather than sloping up perceptibly. 91
Figure 6.3 Machine scalability of D4GE/S4GE on `cnr` and `hollywood09`. This shows very strong scalability with slopes -0.899 and -0.968, which is very close to -1, the perfect value. 92
Figure 6.4 Strong correlation between the enumeration time and $d_{\max}^{\text{sym}}(|\Delta|+|\angle|)$ on the small-medium datasets. 98
Figure 6.5 Scalability of D3GE/FPTE$_{\text{CD}}$ on `uk02` and `arabic`. D3GE/FPTE$_{\text{CD}}$ again presents very strong scalability with slope -0.866 and -0.894. 104

Figure 7.1 Typed triangles with two types of nodes. 108
Figure 7.2 Two weighted graphlets of the same type, but with different sets of weights. 109
Figure 7.3 A multidimensional graph. Each edge has a type based on the dimension, represented by color and line-type. 109

ACKNOWLEDGEMENTS

I would like to thank:

my family, for sharing an extraordinary journey, support, and faith.

Dr. Thomo & Dr. Srinivasan, for their mentoring, guidance, and patience.

University of Victoria, for funding me with a scholarship.

VADA, for funding me with a scholarship and providing a training program in data analysis.

Don't let schooling interfere with your education.

Mark Twain

The only thing that interferes with my learning is my education.

Albert Einstein

DEDICATION

For Julie, Sarah and Lucas.

Chapter 1

Introduction

Graph/Network Analytics has become an indispensable tool for data analysis [3]. Whenever we have a set of objects with some relations or connections among them, we have a network or a graph. For example: computer networks, internet networks, social networks, financial networks, neural networks, road networks, and many more. A graph is a composite mathematical object, consisting of a set of nodes and a set of edges where each edge connects a pair of nodes [13, 16]. There are many methods that have been developed for analysing a graph. Some of them require finding, counting and/or enumerating small subgraphs inside an input graph. The applications can be found in various fields: in biology [34, 43] chemistry [24, 56], social study [28, 14], network analysis and classification [77], and more.

Finding means that we locate a subgraph that matches a given pattern by listing the labels of all of its nodes. Counting means that given a subgraph pattern, we count the number of its occurrences inside the input graph. Enumerating/listing means that we list each occurrence by its nodes. Note that by enumeration we would get the count as well. However, counting does not have to be done through enumeration as there are other methods to do this, e.g., by using some combinatorial formulas. Out of

these three tasks, enumeration has the highest computational complexity, and this is the topic of this dissertation.

Here, I focus on enumerating small subgraphs in massive graphs. By 'small' we typically consider subgraphs with less than ten nodes. Nevertheless, when the input graph is very large, of order, a million or more nodes, enumeration becomes challenging even for subgraphs of order four. The computational complexity grows exponentially on the order of the subgraphs that we want to enumerate. This can be understood combinatorially. Suppose we want to find subgraphs of k nodes in a graph of n nodes, then there are $\binom{n}{k} \propto n(n-1)\ldots(n-k+1)$ possible combinations that we need to check. If $n \gg k$, this is approximately n^k. For a graph of a million nodes, each increment of the subgraph order, k, would increase the complexity by a million times. On the other hand, an order of magnitude increase in n would increase the complexity by 10^k.

Of course, in practice, not all combinations need to be checked. Efficient algorithms were built by minimizing unnecessary checking. It was shown by Shervashidze et al. [67] that by using DFS (Depth First Search), enumeration can be done in $O(nd^{k-1})$ where d is the maximum degree. We will see below that this bound can still be improved. Nonetheless, it is generally true that the number of subgraphs grows rapidly with the size of the graph, and since in enumeration we need to 'touch' each of the subgraph instances, the running time is bounded from below by this number. For this reason, many papers in the literature focus their attention on counting; either finding approximate counts by using probabilistic methods, such as Graft [55], or utilizing combinatorial properties of graph to get the counts without full enumeration, such as Orca [32] and Escape [51].

One may ask, if we can get the counts by other means why do we need to do enumeration? While counts of subgraphs are useful in computing some graph proper-

ties, such as the global clustering coefficient [39, 74], their usage is limited. With the advancements of Artificial Intelligence (AI) and Machine Learning (ML), people are looking for ways of applying ML to graph data [17, 29, 36, 1], a field of study generally known as Graph Machine Learning or Graph Learning [78]. In order to do this, we need to build features that can be used as input by the ML algorithms [31], counting is no longer sufficient. Enumeration gives us more detailed information about the graph that can be used as features. Some analyses use node degrees as a feature [6]. Extending from degrees (i.e., the number of neighbours) to graphlet degrees (see Section 2.7) is an improvement [52]. Similarly, if, instead of an adjacency list, we can use e.g. some adjacent subgraphs lists to build the features, then it would improve the ML performance. Another option that we may explore is to use a list of graphlets, which are small induced connected subgraphs (see Chapter 2 for the definitions), together with their nodes, as a feature. We will not be able to build this feature simply by counting.

In summary, the problem that we want to solve is: given a massive graph how can we enumerate the graphlets inside it efficiently and produce a list of all graphlets within a limited time budget and limited computing resources. I made contributions by proposing several unique algorithms that enumerate all types of graphlets up to a certain order efficiently in a single run, for both undirected and directed graphs. My research has resulted in four published papers [64, 63, 38, 62], so far. I am going to discuss the topics briefly here and will discuss them in detail in subsequent chapters. I also propose a new solution for 5-node graphlet enumeration, that has never been published before, in this dissertation. In addition, I also discuss my study on extending directed graphlet enumeration from 3-node to 4-node.

A subgraph that gained lots of attention from researchers is a triangle. Triangle plays an important role in network analyses. For example, the presence of trian-

gles is an indicator of communities in the network [54]. Triangles are also central to computing the connectivity of a graph [5], the clustering coefficient [75], and the transitivity [45]. There are many practical applications of these, for example, detecting fake users in social networks [80] and uncovering hidden thematic layers in the Web [27]. I show that with careful algorithmic engineering, it is possible to enumerate triangles in a graph of a billion nodes using just a single commodity machine.

A lot of real-world networks have directed relationships, and therefore we should use directed graphs to represent those networks. A triad is a subgraph of three nodes in a directed graph [23, 4]. When each pair of nodes is connected we have a closely connected triad, but we will simply call closely connected triads as triads here. Enumerating triads means listing the edges as well as the nodes inside every triad. Triad enumeration would reveal a more detailed picture of the network, and hence open up more possible applications. For example, transitivity can be more accurately analyzed by using triads [5], and directed clustering coefficient can be used as a measure of systemic risk in complex banking networks [70]. Also, triad enumeration is an important element in social network analysis [74]. I have been able to make a contribution to triad enumeration, published in EDBT-2019 [64]. My algorithm is able to enumerate triads on a graph of a billion nodes and billions of edges using a single commodity machine.

A triangle is a graphlet of three nodes. A wedge is another three-node graphlet that has two edges. Beyond three nodes, graphlets in a massive graph are difficult to enumerate for the same reason as for general subgraphs. Previous belief was that an enumeration algorithm, which has to touch each graphlet, cannot terminate in a reasonable time [51]. Indeed, previous methods, such as Fanmod [76] and Rage [40], do not scale well and take a very long time to run on million scale graphs. In [63], published in EDBT-2020, I proposed an efficient enumeration algorithm for four-node

graphlets. My algorithm achieves a running time of $O((N_\Delta + N_\angle)\, d + T_3)$, where N_Δ is the number of triangles, $N_\angle$ is the number of wedges, and T_3 is the time to enumerate the three-node graphlets. This is a significant improvement compared to the previous $O(nd^{k-1})$ bound, for $k = 4$. Moreover, unlike most in the literature, our solution yields the counts of *all* four node graphlets in a *single* run.

I extended the scalability of my 4-node graphlet enumeration algorithm, which I call S4GE (Simultaneous 4-node Graphlet Enumeration), by porting it to a distributed computing platform. While a distributed platform opens up the possibility of bringing in as much computational power as needed, we cannot simply run the single-machine solution on a distributed platform. There are many technical challenges. Especially, we need an efficient solution, which minimizes the overhead cost and redundancy, to justify the cost of the distributed system. Our solution has been accepted for publication in the SSDBM 2021 conference [38]. In the same paper, I also extended my solution to analyse probabilistic graphs. Furthermore, I extended the paper to include more details and also to bring my triad solution onto a distributed platform as well. This extended paper is published in DAPD 2022 [62].

Continuing my solution on four-node graphlet enumeration, I expand the solution to enumerate five-node graphlets. I propose a solution called S5GE (Simultaneous 5-node Graphlet Enumeration). Similar to S4GE, S5GE simultaneously enumerates all types of 3, 4, and 5-node graphlets in a single run. I have succeeded in building an algorithm and implementing it in a code. The running time of this algorithm is less than the running time of S4GE times the maximum degree. My solution is described in detail in Section 4.3. I also discuss the foundation to expand triad enumeration to enumerate 4-node directed graphlets in Section 5.3 and in Appendix C, where I computed the number of distinct types of 4-node directed graphlets. Once we know these types, we can proceed with a method similar to the triad enumeration.

Chapter 2

Theoretical Background

In this chapter, we provide the necessary theoretical background and the conventions we use in this dissertation.

2.1 Graphs and Digraphs

A *graph*, also known as a *network*, consists of a set of nodes, or vertices, and a set of edges which are connections among the nodes. We commonly denote a graph by $G(V, E)$ where V is the set of nodes and E is the set of edges. An edge represents a relationship or a connection between two nodes. It can be either directed or undirected. In the directed case, the graph is called a *directed graph* or a *digraph*, while in the undirected case the graph is called an *undirected graph*. We consider both directed and undirected graphs in our study. When it is necessary to make a distinction, we denote an undirected graph by G and a directed graph by $\overset{\leftrightarrow}{G}$ [1]. When we talk about the general case we will just use G. The notation that we use for graphs and digraphs is summarized in Table 2.1.

[1] Note that some works of literature use D to denote a directed graph, and some others simply use G (or $\mathcal{G}$) for both cases.

An edge connects a pair of nodes. In general, there could be more than one edge between any pair of nodes. In that case, we call the edges *multi-edges*. For digraphs, an edge from node u to node v (denoted by $u \to v$, or (u, v)) is distinct from an edge from node v to node u, $v \to u$. Therefore, when both are present they are not considered as multi-edges. Also, there could be an edge from a node to itself. This kind of edge is called a *self-loop*. A *simple graph* (or a simple digraph) has neither multi-edge nor self-loop. We focus our research on simple graphs and digraphs, hence it will be assumed from here on that the graphs are simple unless stated otherwise [2].

Symbol	Definition
$\overleftrightarrow{G}(V, E)$	A directed graph, or digraph.
$\overleftrightarrow{G}^T(V, E^T)$	The transpose digraph of $\overleftrightarrow{G}(V, E)$, E^T is the same set of edges as E but with the direction of every edge is reversed.
$u \to v = (u, v)$	A directed edge from u to v.
$N^+(u)$	The set of neighbours from node u.
$N^-(u)$	The set of neighbours to node u.
$d^+(u)$	The out-degree of node u, $d^+(u) = \lvert N^+(u)\rvert$.
$d^-(u)$	The in-degree of node u, $d^-(u) = \lvert N^-(u)\rvert$.
$G(V, E)$	An undirected graph.
$uv = u\text{-}v$	An undirected edge with end-nodes u and v, $uv = vu$.
$N(u)$	The set of neighbours of node u.
$d(u)$	The degree of node u, $d(u) = \lvert N(u)\rvert$.

Table 2.1: Our notation on graphs and digraphs.

The *degree* of a node u, $d(u)$, is the number of edges incident on it. The *out-degree* d^+ is the number of edges *from* the node, while the *in-degree* d^- is the number of edges *to* the node. Given a directed graph $\overleftrightarrow{G}$, its *transpose graph*, $\overleftrightarrow{G}^T$, is a graph of the same set of nodes and edges but with all of the edges are reversed. The out-degree

[2] Note that a graph with multi-edges can be represented as a weighted simple graph where the number of multi-edges in the original graph becomes an edge weight in the weighted graph.

of node u in $\overset{\hookrightarrow}{G}$ is the same as its in-degree in $\overset{\hookrightarrow}{G}^T$, and vice versa, i.e.,

$$d^{\pm}_{\overset{\hookrightarrow}{G}}(u) = d^{\mp}_{\overset{\hookrightarrow}{G}^T}(u) \tag{2.1}$$

The number of nodes in a graph $G(V, E)$ is known as the *order* of the graph, commonly denoted by $n = |V|$. The number of edges is known as the *size* of the graph, commonly denoted by $m = |E|$. Note that for an undirected graph, we have

$$2m = \sum_{u \in V} d(u) \tag{2.2}$$

while for a directed graph

$$m = \sum_{u \in V} d^{+}(u) = \sum_{u \in V} d^{-}(u) \tag{2.3}$$

We can symmetrize a directed graph $\overset{\hookrightarrow}{G}$ to get an undirected graph $\widetilde{G}$ by taking the union between $\overset{\hookrightarrow}{G}$ and its transpose, $\overset{\hookrightarrow}{G} \cup \overset{\hookrightarrow}{G}^T$, and treat each pair of edges as a single undirected edge. This process is illustrated in Fig. 2.1. The undirected graph $\widetilde{G}$ is also known as the *underlying graph* of $\overset{\hookrightarrow}{G}$. Note that the set of nodes remains the same,

$$V(\widetilde{G}) = V(\overset{\hookrightarrow}{G}) = V(\overset{\hookrightarrow}{G}^T) \tag{2.4}$$

while the numbers of edges obey

$$|E(\overset{\hookrightarrow}{G})| \leq 2|E(\widetilde{G})| \leq 2|E(\overset{\hookrightarrow}{G})| \tag{2.5}$$

For a simple undirected graph, the number of edges is bounded by

$$0 \leq m \leq n(n-1)/2 \tag{2.6}$$

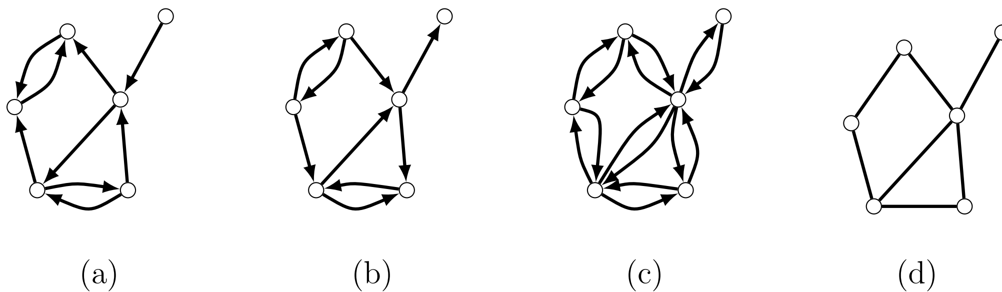


Figure 2.1: The process of symmetrizing a directed graph to get an undirected graph. (a) The graph $\overset{\hookrightarrow}{G}$. (b) The transpose graph $\overset{\hookrightarrow}{G}^T$. (c) The union graph $\overset{\hookrightarrow}{G} \cup \overset{\hookrightarrow}{G}^T$. (d) The corresponding underlying undirected graph $\widetilde{G}$.

The upper bound occurs when every pair of nodes in the graph are connected. In this case, the graph is a *complete graph*. A *regular graph* is a graph where all of its nodes have the same degree. If the degree is r, it is called a regular graph of degree r, or an r-regular graph. In a regular graph of degree r and n nodes, the number of edges is $rn/2$. A complete graph of n nodes, denoted by K_n, is a regular graph of degree $n-1$. A *cycle*, C_n, is a graph where each node has two neighbours, and the edges form a single loop. Thus, a cycle is a regular graph of degree 2.

A complete graph has the maximum number of edges a simple graph can have. That is $m_{\max} = n(n-1)/2$ for a graph of order n. On the other side of the spectrum, we may have a graph without any edge. This is called an *empty graph*. *Graph density*, ρ, is a measure to specify how densely connected a graph is. It is defined as

$$\rho = \frac{m}{m_{\max}} = \frac{m}{n(n-1)/2} \tag{2.7}$$

which has a value between 0 and 1. Related to this we can roughly categorize a graph as sparse or dense. A *sparse graph* is a graph with $m \ll m_{\max}$, while a *dense graph* is a graph with m close to $m_{\max}$. A graph with $m = O(n)$ is considered sparse, while a graph of $m = O(n^2)$ is considered dense.

2.2 Graph Representations

For computational purposes it is common to label the nodes with integers: 1, 2, 3, ... (or 0, 1, 2, 3, ...). There are several ways to represent the edges: by using an adjacency matrix, an incidence matrix, or an adjacency list.

An *adjacency matrix*, A, is an $n \times n$ matrix, where n is the number of nodes. The entries in the matrix represent the number of edges between the row node and the column node. For example, A_{ij} is the number of edges between node i and node j. For undirected graphs, the matrix is symmetric, $A_{ij} = A_{ji}$, while for directed graphs in general the matrix is not symmetric. The sum of the entries along one row is the degree of the node represented by that row. For a directed graph, A_{ij} is the number of edges from node i to node j, the horizontal sum gives us the out-degrees, and the vertical sum gives us the in-degrees. For a simple graph, the entries (or matrix elements) are either 1 or 0, with the diagonal entries all 0.

An *incidence matrix*, I, is an $n \times m$ matrix, where n is the number of nodes and m is the number of edges. Only two elements per column are 1, and the rest are 0. The two rows where they are 1 represent the two end nodes of the edge represented by that column. Incidence matrix is less commonly used in graph computation compared to the adjacency matrix.

An *adjacency list* is a list of n rows. Each row is prefixed by a node label, and it contains the neighbours of that node. It is common to have them listed in ascending order. The number of entries in each row is equal to the degree of the node represented by that row. Because we list nodes that are connected to the row node, an adjacency list is a more efficient representation for sparse graphs.

As an illustration, consider the simple graph in Figure 2.2 which has 5 nodes and

6 edges. The adjacency matrix for this graph is

$$
\begin{array}{ccccccc}
 & 1 & 2 & 3 & 4 & 5 & \\
A = & \left(\begin{array}{ccccc} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{array}\right) & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\end{array} \tag{2.8}
$$

The adjacency list is

$$
\begin{array}{ccl}
1 & : & 2, 3, 4, 5 \\
2 & : & 1 \\
3 & : & 1, 5 \\
4 & : & 1, 5 \\
5 & : & 1, 3, 4
\end{array} \tag{2.9}
$$

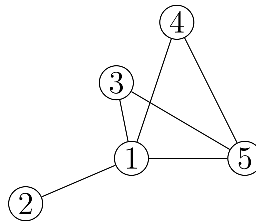


Figure 2.2: A simple graph of 5 nodes and 6 edges.

The advantage of using an adjacency matrix is that some graph problems can be solved through matrix computations. For example, to compute the number of triangles in the graph, Δ, we can compute A^3, and use the relation

$$\Delta = \frac{1}{6}\mathrm{Tr}\left(A^3\right) \tag{2.10}$$

In this example, it is 2. However, when dealing with a very large graph, the adjacency

matrix requires a lot of memory space. Thus, matrix methods may not be applicable in this case. An adjacency list may be able to represent the same graph by using much less memory, especially when the graph is sparse - with a lot of 0s in the adjacency matrix.

Furthermore, using the adjacency list, there is a compression scheme that can be utilized to reduce the size of the representation even smaller. This scheme is the WebGraph [11, 12], which is very useful in dealing with massive graphs. Instead of listing the labels as they are, WebGraph lists just the first number and then subsequently the difference to the next number in the list. Overall, the differences are smaller than the numbers themselves, and smaller numbers can be represented by fewer number of bits. Since we are focusing on massive graphs, we mainly use an adjacency list, and we employ WebGraph in our experiments.

2.3 Subgraphs and Subgraph Patterns

A graph $H(V_H, E_H)$ is a *subgraph* of graph $G(V_G, E_G)$ if and only if $V_H \subseteq V_G$ and $E_H \subseteq E_G$. We use the notation $H \subseteq G$ to express that H is a subgraph of G. A subgraph $H \subseteq G$ is an *induced subgraph* if for any $u, v \in V_H$, uv is in E_H if and only if uv is in E_G. Without this condition, the subgraph is *non-induced*.

An *edge induced* subgraph is a subgraph formed by a set of selected edges from the graph and the end nodes of those edges. Note that the notion of an edge-induced subgraph is not the same as that of an induced subgraph, because an induced subgraph is a subgraph of chosen vertices while an edge-induced subgraph is a subgraph of chosen edges. This is illustrated by an example shown in Figure 2.3. As we will see below, induced subgraphs are related to graphlets while edge-induced subgraphs are used for graph partitioning.

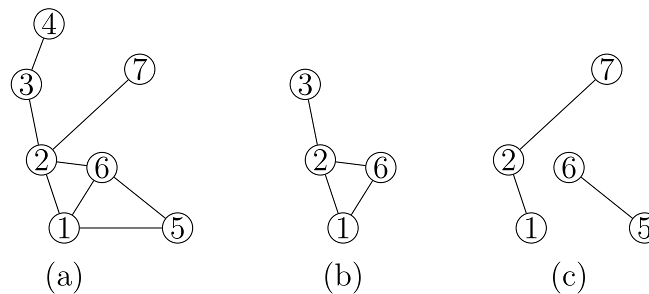


Figure 2.3: (a) A graph, (b) an induced subgraph (induced by vertex set $\{1, 2, 3, 6\}$) and (c) an edge-induced subgraph (induced by edge set {1-2, 2-7, 5-6}). Note that (c) is not an induced subgraph of (a).

A *path* is a sequence of edges where each pair of subsequence edges are connected by a common node, $u_0u_1, u_1u_2, u_2u_3, \ldots, u_{k-1}u_k$, and each node appears only once, i.e., $u_0 \neq u_1 \neq \ldots \neq u_k$. In the directed case the edges should have the same orientation, i.e., they are connected by head-to-tail, $u_0 \to u_1, u_1 \to u_2, \ldots, u_{k-1} \to u_k$. If $u_0 = u_k$, then we have a *cycle*.

An undirected graph is *connected* if any two nodes in it are connected by at least one path. A directed graph is *weakly connected* if its underlying undirected graph is connected. A directed graph is *strongly connected* if for every pair of nodes u and v there are directed paths from u to v and from v to u. A subgraph that contains all nodes connected to each other, and all the edges among the nodes, is called a *connected component*. Using this definition, a graph is connected if and only if it has only a single connected component.

Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are *isomorphic* if there is a mapping $\varphi : V_1 \to V_2$ such that $uv \in E_1$ if and only if $\varphi(u)\varphi(v) \in E_2$. It is necessary that $|V_1| = |V_2|$ and $|E_1| = |E_2|$. Given a small graph $H(V_H, E_H)$ and a graph $G(V_G, E_G)$, the *enumeration problem* is to find and list all subgraphs of G that are isomorphic to H. In this context, H is also called the *subgraph pattern* for the enumeration. We can either impose that the subgraphs are induced, or not. This leads to two types of enumeration problems: induced and non-induced. A *clique* is a subgraph that is

isomorphic to a complete graph. Note that a clique is always induced.

Related to enumeration and/or counting problems, there is also the motif problem. *Motifs* are statistically significant subgraph patterns. To classify a given graph, sometimes we just need to know the dominant subgraph patterns inside the graph.

2.4 Wedges, Triangles, and Triads

A *wedge* is a small simple graph of three nodes and two edges. Given a reference node, we can classify wedges into two types: (1) one with the reference node at the center, where the two edges meet, and (2) one with the reference node at one of its legs. This is illustrated in Figure 2.4, where we use the lowest labeled node as the reference node. We will see later that these two types require different ways to enumerate.

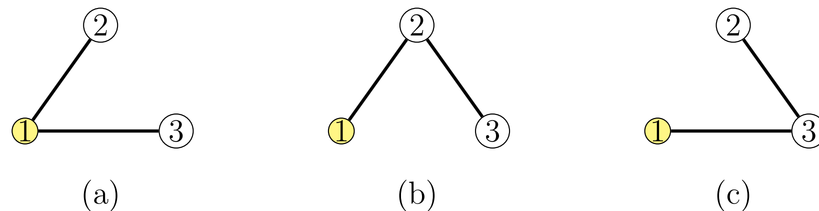


Figure 2.4: Using node 1 as the reference node, we have two types of wedges: Type 1 wedge $(2, 1, 3)$ (a) and Type 2 wedges $(1, 2, 3)$ and $(1, 3, 2)$ (b and c).

A *triangle* is a graph of three nodes connected to each other by three edges. An undirected triangle is shown in Fig. 2.5. It is known as a clique of order 3, K_3, which is also a cycle of order 3, C_3. We represent a triangle by its nodes, e.g., $(u, v, w)_\Delta$ for a triangle with nodes u, v, and w. Note that, for example, $(w, v, u)_\Delta$ represents the same triangle as $(u, v, w)_\Delta$. In fact, any permutation of the three nodes would represent the same triangle. This leads to a multiple counting problem in enumeration. There are six permutations of three labels. So, if we do not take care of the ordering of the

labels, a triangle could be counted six times. To avoid this problem we can impose a condition that we list a triangle only in ordered labels, $u < v < w$.

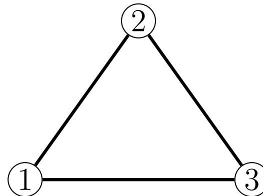


Figure 2.5: A triangle, $(1, 2, 3)_\Delta$.

In directed graphs, a cycle is a set of edges connecting nodes in a cyclic path. A cycle of three nodes is depicted in Fig. 2.6(a). We call this a *cycle triangle*, and denote it by $(u, v, w)_C$. Now, note that $(u, v, w)_C = (w, u, v)_C = (v, w, u)_C$, but $(u, v, w)_C \neq (u, w, v)_C$. To avoid multiple counting for a cycle we anchor the first node label to be the smallest. Thus we might, for example, enumerate $(1, 2, 3)_C$ and $(1, 3, 2)_C$, but not $(2, 3, 1)_C$. When one of the edges of a cycle triangle is flipped, we have what we call a *trust triangle*. This is depicted in Fig. 2.6(b). We use the name trust to reflect that it shows a propagation of trust. That is, if 1 trusts 2 and 2 trusts 3, then 1 also trusts 3. A trust triangle is also known as a *truss*, as in a truss bridge, or a *transitive triangle*. We use index T to denote a trust triangle, e.g., $(u, v, w)_T$ is a trust triangle of nodes u, v and w, connected by edges $u \to v$, $v \to w$ and $u \to w$. For trust triangles, the order of the nodes matters. We start with the source node and end with the sink node. We do not impose any ordering condition when enumerating trusts since each permutation represents a different trust. Cycle and trust triangles are the only possible types of directed triangles. Typically, we consider them as non-induced subgraphs within a larger graph.

On the other hand, given a directed graph we can explore all of its induced subgraphs of three nodes, known as *triads* [33, 74, 4]. There are sixteen of those. A unique coding for triads that describes the type of connections inside is well known.

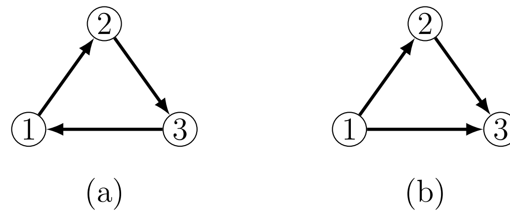


Figure 2.6: (a) A cycle triangle, $(1, 2, 3)_C$ and (b) a trust triangle, $(1, 2, 3)_T$.

We show these sixteen triads and their coding in Figure 2.7.

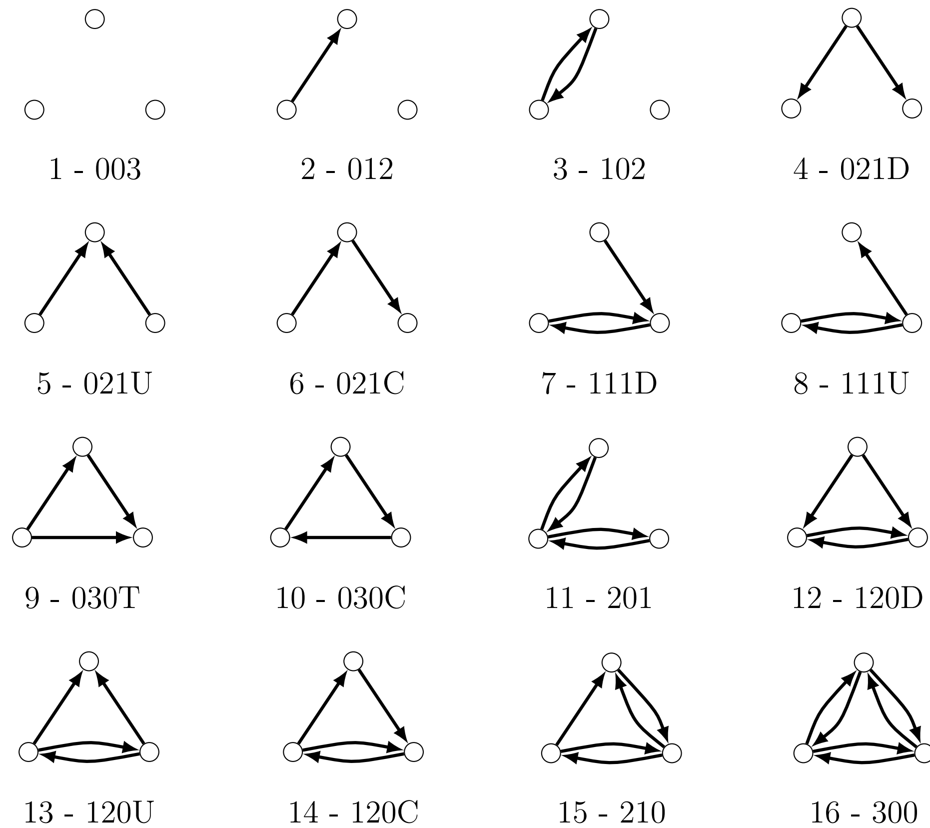


Figure 2.7: Sixteen types of triads.

The coding consists of three digits and may be followed by a letter to specify the orientation. The three digits are for the numbers of mutual connection, asymmetric connection, and no connection (or null dyad), respectively. The possible orientations are up (U), down (D), cyclical (C), and transitive (T). The triads can be classified as having no connection (empty), single connection, two connections, and three connec-

tions (or triangle).

Note that to count/list triads we need to consider every triple node combination in the graph, there are $O(n^3)$ of them. We may restrict the search to only triads whose underlying graph is a triangle. These are called *triangle triads*, and there are seven types of them as depicted in Fig 2.8. In fact, this is what we do here, and hence from here on when we say triads we will implicitly mean triangle triads. We label them as Type 1, Type 2, ... Type 7 [3], to simplify our notation. Note that, using the coding above, Type 1 triad is 030C (10), Type 2 triad is 030T (9), Type 3 triad is 120C (14), and so on.

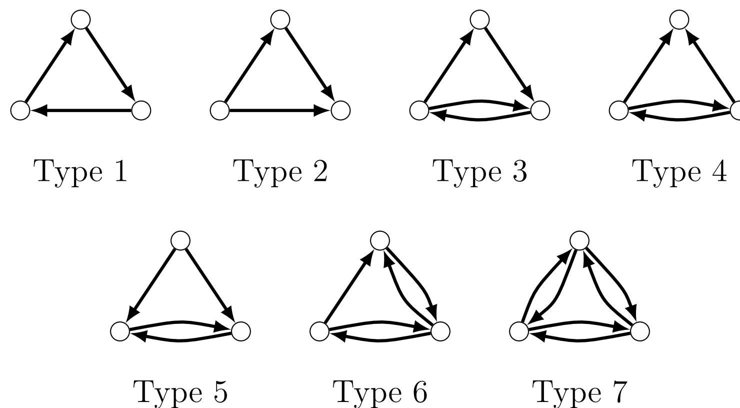


Figure 2.8: Seven types of (triangle) triads.

Notice that Type 1 triad is isomorphic to a cycle triangle and Type 2 is isomorphic to a trust triangle. However, we should make the distinction between triads and directed triangles clear here. When we enumerate directed triangles we do not impose on them to be induced subgraphs. However, triads, by definition, are always induced subgraphs. For any triad, there could be from zero to six trust triangles, and from zero to two cycle triangles. The maximum number of trust and cycle triangles are found in the Type 7 triad, which has six directed edges. This is shown in Fig. 2.9 and Fig. 2.10 respectively. Our notations for triangles and triads are summarized in

[3]Our numbering convention is slightly different from the one used by other authors [20, 46].

Table 2.2.

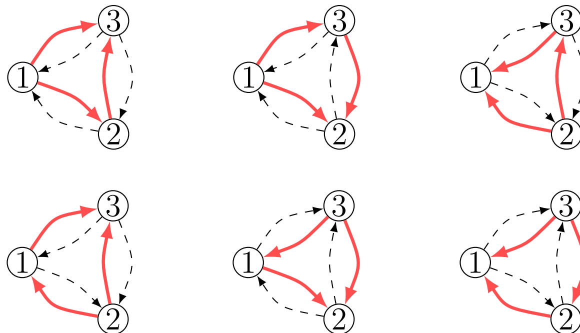


Figure 2.9: There are six trust triangles in a Type 7 triad: $(1,2,3)_T$, $(1,3,2)_T$, $(2,3,1)_T$, $(2,1,3)_T$, $(3,1,2)_T$, and $(3,2,1)_T$ respectively.

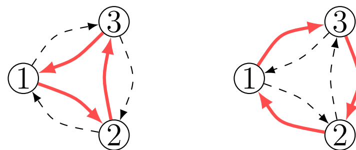


Figure 2.10: There are two cycle triangles in a Type 7 triad: labeled as $(1,2,3)_C$ and $(1,3,2)_C$ respectively.

The underlying graph of any (triangle) triad is an undirected triangle. Therefore, if we just need to know the total number of triads in a graph, without needing to know the number of each type, we can count the number of triangles in the corresponding undirected graph instead. That is

$$\Delta = \sum_{i=1}^{7} \Delta_i \tag{2.11}$$

The type of a triad is characterized by the number of cycles and trusts that it contains. This is listed in Table 2.3. This table tells us that we can count the number of cycle triangles and trust triangles from the number of triads by the following relations

$$\Delta_C = \Delta_1 + \Delta_3 + \Delta_6 + 2\Delta_7 \tag{2.12}$$

Symbol	Definition
$(u, v, w)_\Delta$	An undirected triangle with nodes u, v, and w.
$(u, v, w)_T$	A trust with nodes u, v, and w, connected by edges $u \to v$, $v \to w$, and $u \to w$.
$(u, v, w)_C$	A cycle with nodes u, v, and w, connected by edges $u \to v$, $v \to w$, and $w \to u$.
$(u, v, w)_i$	A triad of type i with nodes u, v, and w,
Δ	The number of (undirected) triangles.
Δ_T	The number of trust triangles.
Δ_C	The number of cycle triangles.
Δ_i	The number of triads of type i.

Table 2.2: Our notation on triangles and triads.

and

$$\Delta_T = \Delta_2 + \Delta_3 + 2\Delta_4 + 2\Delta_5 + 3\Delta_6 + 6\Delta_7 \tag{2.13}$$

respectively.

Triad	Description
Type 1	1 cycle
Type 2	1 trust
Type 3	1 trust + 1 cycle
Type 4	2 trusts merging
Type 5	2 trusts parting
Type 6	3 trusts + 1 cycle
Type 7	6 trusts + 2 cycles

Table 2.3: Triad types and description.

2.5 Cliques

A *clique* is a subgraph that is isomorphic to a complete graph. Because each pair of nodes in a clique is connected, the clique must include all edges within the subgraph, hence it must be an induced subgraph. A triangle (viewed as a subgraph)

is a 3-clique. A four-node subgraph with each pair of nodes connected is a 4-clique. In general, a k-clique is a subgraph of k nodes where each pair is connected by an edge. We show some lower-order cliques in Fig. 2.11. Consider a k-clique with k nodes $1, 2, \ldots, k$. Note that there is only one k-clique containing all of these specific nodes in the whole graph. Thus, any permutation of $1, 2, \ldots, k$ would refer to the same k-clique. By convention, to avoid duplicate listing, we can choose to list a k-clique by its nodes in ascending order, e.g., $(1, 2, \ldots, k)_k$.

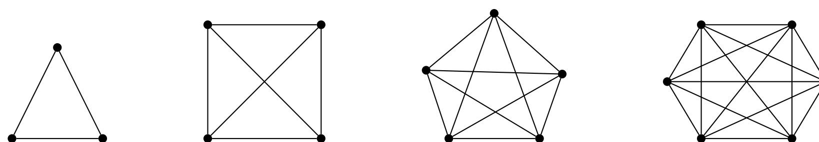


Figure 2.11: A 3-clique, a 4-clique, a 5-clique and a 6-clique.

We observe that a k-clique contains k of $(k-1)$-cliques. So for example, a 4-clique contains four 3-cliques (or triangles), a 5-clique contains five 4-cliques, and a 6-clique contains six 5-clique. We can continue the decomposition all the way down to triangles, and find that a 4-clique contains four triangles, a 5-clique contains ten triangles, and a 6-clique contains twenty triangles. Or in general, a k-clique contains $\binom{k}{3}$ triangles, for $k \geq 3$, $\binom{k}{4}$ 4-cliques, for $k \geq 4$, $\binom{k}{5}$ 5-cliques, for $k \geq 5$, and so on.

Moving to the other direction, given a k-clique we can try to find another node in the graph that is connected to all nodes in the k-clique. This new node together with the k-clique form a $(k+1)$-clique. We can continue this process of growing a clique. At some point, there will be no more point that is connected to all nodes in the clique, and hence the process stops. A clique that is not part of a higher order clique is called a *maximal clique*.

2.6 Graphlets

Next, we generalize our discussion from triangles and cliques to graphlets. We assume the following definition: A *graphlet* is a small induced connected subgraph. Small here is not rigidly defined, but roughly speaking of the order less than ten. For simplicity, we will restrict our discussion here to the undirected case. There are two types of three-node graphlets: triangles and wedges. A wedge consists of three nodes and two edges where the two edges are connected by a common node. A triangle has three nodes and three edges. We label wedges as g_1 and triangles as g_2.

For four nodes, there are six types of graphlets as shown in Figure 2.12. We have 3-paths (or four-node-paths) (g_3), 3-stars (g_4), 4-cycles or rectangles (g_5), tailed-triangles (g_6), diamonds (g_7), and 4-cliques (g_8). The labeling convention that we use here follows the one commonly used in the literature [53, 8] [4].

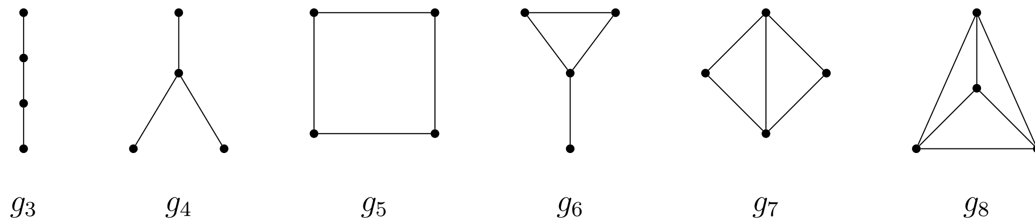


Figure 2.12: Four node graphlets: a 3-path (g_3), a 3-star (g_4), a rectangle or 4-cycle (g_5), a tailed-triangle (g_6), a diamond (g_7), and a 4-clique (g_8).

Note that, as 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.). Similarly, the four-node graphlets cannot be on top of each other. Notice that g_3, g_4 and g_5 do not contain any triangle, while g_6, g_7 and g_8 contain 1, 2 and 4 triangles respectively.

Next, on to the five-node graphlets. There are 21 types of (or non-isomorphic)

[4] Except that here we use small g, while other authors denote them using capital G.

graphlets of five nodes [8]. These are shown in Figure 2.13. Again, we can see that some of them contain triangles while others do not. However, the hierarchical structure of five-node graphlets is more complicated.

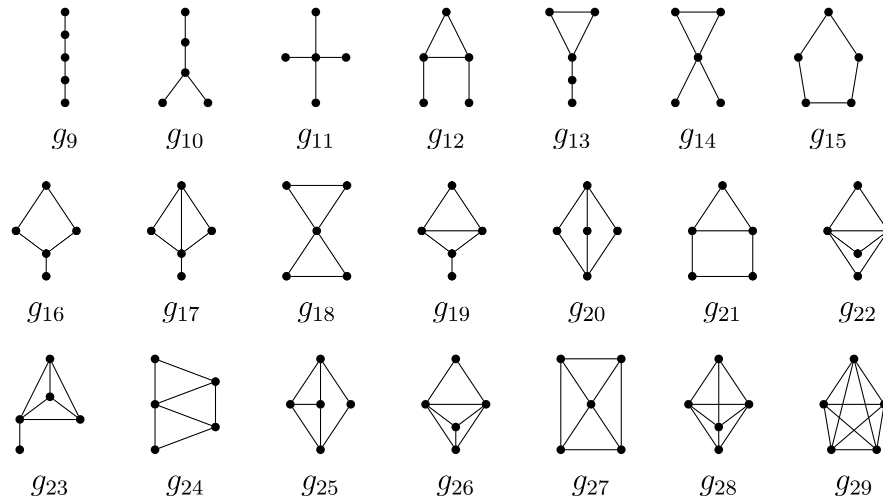


Figure 2.13: Five node graphlets.

In summary, there are 2 non-isomorphic 3-node graphlets, 6 non-isomorphic 4-node graphlets, and 21 non-isomorphic 5-node graphlets. Higher-order graphlets are even more numerous. For 6-node, there are 112 non-isomorphic graphlets, for 7-node, there are 853 non-isomorphic graphlets, and for 8-node, there are 11117 (https://oeis.org/A001349).

2.7 Orbits and GDV

Taking a node point of view, we can count the number of each graphlet type that it is a part of. This is an extension of the node degree, which basically is the count of two node graphlets attached to the node being considered.

For a given graphlet type, we can still differentiate further by the location of the node within the graphlet. For a wedge, for example, it matters whether the node is

at the joint or at one of the legs. This leads to the notion of *orbits*. Up to five node graphlets, we have 73 orbits as shown in Figure 2.14 below.

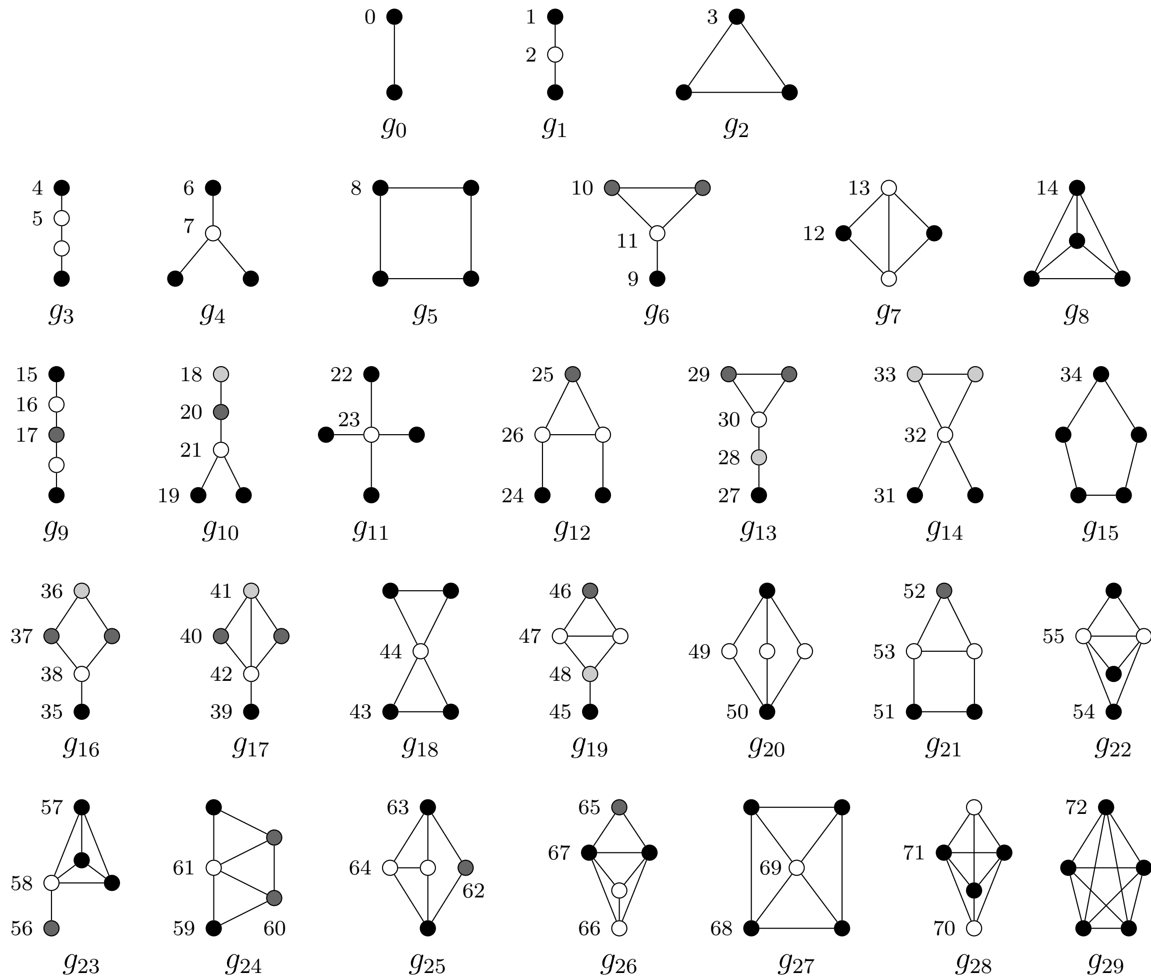


Figure 2.14: Orbits for 2, 3, 4, and 5 node graphlets. The labeling is following the convention used by Przulj [52].

This brings us to the notion of *Graphlet Degree Vector* (GDV), which is the 73 counts attached to each node. The collection of GDVs for all nodes in a graph forms what is called the *Graphlet Degree Distribution* (GDD).

2.8 Induced and Non-Induced Subgraphs

We have talked about induced and non-induced subgraphs. They differ on whether or not to include all connected edges from the original graph. We have noted that a clique is always an induced subgraph. However, we can view a clique as non-induced as well, if we do not impose the induced condition.

Given the counts of non-induced subgraphs, we can derive the counts of induced subgraphs, and vice versa. The relation is given by

$$N = M\, I \tag{2.14}$$

where N is the vector for the non-induced counts, and I is the vector for the induced counts. For 3-node subgraphs, wedges, and triangles, the conversion matrix is

$$M = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix} \tag{2.15}$$

This is telling us that for every triangle, there are 3 non-induced wedges inside it. For 4-node (3-path, 3-star, square, tailed triangle, diamond, 4-clique, in this order), the matrix M is

$$M = \begin{bmatrix} 1 & 0 & 4 & 2 & 6 & 12 \\ 0 & 1 & 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 & 4 & 12 \\ 0 & 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

The get induced counts from the non-induced counts, we just need to reverse the

equation to

$$I = M^{-1} N \tag{2.17}$$

where the inverse matrices are

$$M^{-1} = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix} \tag{2.18}$$

for 3-node, and

$$M^{-1} = \begin{bmatrix} 1 & 0 & -4 & -2 & 6 & -12 \\ 0 & 1 & 0 & -1 & 2 & -4 \\ 0 & 0 & 1 & 0 & -1 & 3 \\ 0 & 0 & 0 & 1 & -4 & 12 \\ 0 & 0 & 0 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.19}$$

for 4-node.

Chapter 3

Related Works

3.1 Triangles

Triangle enumeration and counting have been the subject of numerous papers. For this seemingly simple task, many algorithms/methods had been proposed, the list of which can be found in [37]. Schank and Wagner [65] did an experimental comparison among some of the algorithms. Latapy [37] did a more detailed study and confirmed the finding. Since then it is commonly agreed that Compact Forward algorithm is the most efficient algorithm for triangle enumeration on shared memory platforms. Some more details about enumerating triangles and related engineering are presented in [61].

3.2 Triads

Pajek (`http://mrvar.fdv.uni-lj.si/pajek/`) is a well-known graph analysis software that can be used for triad enumeration. The triad census algorithm by Batagelj and Mrvar [4], which is employed in Pajek, has been known as the standard algorithm to enumerate triads for several years. However, this program is not suitable

for very large graphs with millions of nodes and edges because it is too slow.

Chin et al. [20] proposed a compact data structure where both outgoing and incoming edges are listed in the same adjacency list, and the edge direction is encoded using the 2 lowest bits out of the 32 bits (i.e., using `int`) in each entry. This data structure is suitable for parallel computation using shared memory architectures. This idea was further refined by Parimalarangan et al. [46], who proposed two types of algorithms, intersection based (AI) and marking based (AM), as the most efficient algorithms to enumerate triads on shared memory platforms.

While those ideas significantly improve the running time, there is a cost due to the scalability. With two bits used for edge direction, the order of the graphs that can be processed is reduced. This becomes a problem when we want to analyse a graph of a billion nodes. Theoretically, we can switch to 64-bit integers and the problem should be moot. Practically, however, this would at least double the memory requirement. With a limited memory budget, space is already a problem for analysing very large graphs.

We proposed a different approach where the edge encoding is computed on the fly [64]. With this method, we were able to enumerate triads in a graph with around a billion nodes using a single commodity machine. Moreover, the running time is competitive with the previous solution.

3.3 Graphlets

Chiba and Nishizeki published several subgraph listing algorithms [19] which can be considered as a pioneering work on graphlet enumeration. Milo et al. [44] 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. Fanmod [76] and Rage [40] are such solutions, but they do not scale well and take a very long time to run on graphs with millions of nodes and edges. Another proposed solution, PGD [2], uses parallelized algorithm.

There are several algorithms proposed in the literature to count (but not enumerate) the number of graphlets. They are either estimates using approximation methods, such as Graft [55], or exact counting without full enumeration, notably Orca [32] and Escape [51]. In addition, Silvestri [68] provided another complexity analysis on subgraph enumeration.

We proposed a graphlet enumeration solution that enumerates all types of graphlets of 3 and 4-nodes in a single run, called S4GE [63]. In this dissertation, we propose S5GE, which enumerates 3, 4, and 5-node graphlets in a single run.

3.4 Cliques

Clique enumeration had been the subject of many papers long before computers are widely available. There are two influential papers on cliques, from which later algorithms on cliques were based: Bron and Kerbosch [15] which was published in 1973, and Chiba and Nishizeki [19] which was published in 1985. The current state-of-the-art (SOTA) is by Danisch, Balalau, and Sozio [22] which is an improved implementation based on Chiba and Nishizeki algorithm. Jain and Seshadhri [35] proposed a counting method that does not require full enumeration.

3.5 Distributed Enumeration

Suri and Vassilvitskii [69], proposed a MapReduce triangle enumeration algorithm, GP, based on graph partitioning. The idea is to partition the graph into overlapping subsets of sub-graphs such that each triangle in the graph comes in at least one of the

partitions. However, GP has a lot of redundant operations. To rectify the problem in GP, Park and Chung [47] proposed another Map-Reduce algorithm, called Triangle Type Partition (TTP). However, for very large graphs, TTP gets out of memory error when the size of shuffled data is getting larger than the total available space. Park et al. subsequently proposed an improvement [49] with a multi-round MapReduce algorithm, Colored Triangle Type Partition (CTTP), to reduce the amount of shuffled data in each round. However, the net shuffled data across all rounds remains the same as TTP. Finally, Park et al. [48] proposed a new MapReduce algorithm, Pre-partitioned Triangle Enumeration (PTE), which significantly reduces the amount of shuffled data and has better performance. Bhojwani [7] expanded PTE to enumerate trust and cycle triangles in directed graphs.

For general graphlets, Park et al. generalized PTE to PSE [50]. It uses VF2 algorithm [21] as its serial enumeration algorithm. We consider PSE as the state of the art for distributed solutions of subgraph enumeration problems.

There are several distributed methods/frameworks for graph processing published in the literature, notably Arabesque [72] and its descendant Fractal [25], DistGraph [71], GraphZero [41], G-Miner [18], G-thinker [79], RADS [57] and DISC [81]. They are multipurpose graph applications that can be used to enumerate subgraphs as well. They are query based, where the users need to supply the subgraph finding algorithm. In general, they are more suitable for non-induced subgraphs enumeration.

To the best of our knowledge, there has not been a distributed solution to simultaneously, and fully, enumerate all the induced 4-node graphlets, that can scale to large graphs using a modest cluster of machines, before our proposed solution, D4GE [38]. D4GE is bringing our single machine enumeration S4GE to the distributed platform, using an extended version of the partition scheme used by PSE.

Chapter 4

Undirected Graphlet Enumeration

4.1 Triangles

Triangle is the smallest nontrivial graphlet that has an important role in graph analysis. Triangle enumeration had been studied extensively in the literature and many algorithms had been proposed. The most efficient algorithm known for triangle enumeration is the Compact Forward algorithm [37]. This algorithm works by realizing that when iterating through the nodes, we only need to consider higher labeled neighbours of each node. This is related to the symmetrical property of a triangle, that we only need to enumerate each triangle once when the nodes are ordered. We built a modified algorithm by employing a graph pre-processing before doing the triangle enumeration. In the pre-processing, we first sort the nodes in ascending order of their degrees and cut off lower ordered neighbours from each node's adjacency list [1]. With careful design and implementation, we were able to enumerate triangles in a graph with almost a billion nodes and more than forty billion edges using a single commodity machine within a reasonable amount of time (i.e., less than a day) [61].

[1] In Compact Forward Algorithm, the nodes are sorted in descending order of the degrees, and only lower ordered neighbours are kept for finding the third node. This filtering is done on the fly while iterating through the adjacency lists.

4.1.1 Algorithms

The algorithm that we use for undirected triangle enumeration is described in Algorithm 1. It iterates through the nodes and the neighbours of each node. For each neighbour with a higher index (line 4) it calls the INTERSECTION function to find the common neighbours. We keep only common neighbours with higher indices (line 7) to avoid multiple counting.

Algorithm 1 UNDIRECTED TRIANGLE ENUMERATION

Input: An undirected graph $G = (V, E)$ and its adjacency list $N(v \in V)$.
Output: The list of the triangles and the number of triangles in G.

```
1:  S ← ∅, c ← 0
2:  for all nodes u ∈ V do
3:      for all nodes v ∈ N(u) do
4:          if u < v then
5:              S ← INTERSECTION(N(u), N(v), d(u), d(v))
6:              for all nodes w ∈ S do
7:                  if v < w then
8:                      Enum (u, v, w)                    ▷ For Listing
9:                      c ← c + 1
10: return c
```

The INTERSECTION function is described in Algorithm 2. It works on any two sorted sets of integers. It starts by considering the lowest numbers of each set and checks whether the two numbers are equal. If yes, then the number is put into the intersection list. It then proceeds by moving up in the set with a lower number, or both if equal.

To avoid multiple counting we impose a condition on the ordering of the nodes, $u < v < w$. This condition leads to an interesting consequence. Since $u < v$ and $u < w$, only the larger neighbours of u need to be considered. Similarly, since $v < w$, only the larger neighbours of v need to be considered. As a result, we can actually cut the adjacency list to keep only larger neighbours before starting the triangle enumeration and we would obtain the same result. In this case, line 4 and 7 of

Algorithm 2 INTERSECTION

Input: Two sorted sets of integers A and B, $a =$ size of A, $b =$ size of B
Output: The set $A \cap B$, the size of the intersection.

```
1:  C ← ∅
2:  i ← 0, j ← 0, k ← 0
3:  while i < a and j < b do
4:      if A[i] == B[j] then
5:          C[k] ← A[i]
6:          k ← k + 1, i ← i + 1, j ← j + 1
7:      else if A[i] < B[j] then
8:          i ← i + 1
9:      else                                    ▷ A[i] > B[j]
10:         j ← j + 1
11: return C (and optionally k)
```

Algorithm 1 is no longer needed. Also, since the degrees are reduced, the number of iterations is reduced, and the running time is shorter. Note that the sorting can be done in $O(n \log n)$ time using some common sorting algorithms. The graph preparation is described in Algorithm 3. First, we order the nodes according to the degrees and relabel (or map) them accordingly. Then we omit smaller neighbours in the adjacency lists of the relabeled nodes. The node with the highest degree would be labeled the highest, which after omitting the smaller neighbours would get zero neighbour. This idea had also been realized in the Compact Forward algorithm [37]. However, here we separate these sorting and cutting processes into a separate phase. This pre-processing reduces the maximum effective degree of the graph. It has the greatest impact on graphs with exponential (or power-law) degree distribution, where few nodes have very large degrees and many have relatively small degrees.

Algorithm 3 GRAPH-PREP

Input: An undirected graph $G(V, E)$

```
1: Sort V based on the degrees, in ascending order.
2: Relabel (or map) 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.
```

4.1.2 Analysis

We give the proof of correctness and analyse the running time of the algorithm.

Theorem 1. *The Undirected Triangle Enumeration algorithm (Algorithm 1), applied on an undirected graph that is preprocessed using the Graph Preprocessing algorithm (Algorithm 3), correctly enumerates the triangles in the input graph.*

Proof. Given an undirected graph, we iterate through all of its nodes, and for each node, we iterate over all its neighbours. Thus, all edges (or all pairs of nodes in the graph with an edge between them) would be iterated. By using the INTERSECTION function, all common neighbours of the two end nodes of each edge would be found. Thus, all triangles in the graph would be found. Without imposing the ordering on the nodes, a triangle with nodes u, v and w would be found as (u, v, w), (u, w, v) (when iterate node u), (v, w, u), (v, u, w) (when iterate node v), (w, u, v), and (w, v, u) (when iterate node w). That is six permutations of three indices. One of them would be the ordered permutation, which, without loss of generality, is (u, v, w) with $u < v < w$. Thus, to enumerate a triangle we just need to find this ordered one. This is done by checking the condition $u < v$ and $v < w$ in the algorithm.

Relabelling the nodes of a graph is equivalent to finding an isomorphic graph. Thus relabelling does not change the number of triangles inside the graph. The listing can be translated back to the old labels if needed.

Now, if an edge (u, v) is in the original graph and $u < v$, then (u, v) would also be in the preprocessed graph (assuming that we already use the new labels). Therefore, running the Triangle Enumeration on the preprocessed graph, the edge (u, v) would be iterated. Since $w > u$ and $w > v$, w would be kept in the list of higher ordered neighbours of u and v, respectively, in the preprocessed graph. Therefore the triangle (u, v, w) would be found. □

For the running time, we claim the following.

Theorem 2. *The running time of the Undirected Triangle Enumeration on graph $G(V, E)$ is bounded by $O(|V|(d_{\max}(G))^2)$.*

Proof. Recall that the algorithm practically iterates over the edges of the graph. For each edge (u, v) the intersection computation runs in time $d(u) + d(v)$. Thus, the running time is $\sum_{u \to v}(d(u) + d(v)) \leq \sum_u d(u) \left(d(u) + \max[d(v \in N(u))]\right)$. The worst case is $2|V|(d_{\max}(G))^2$. □

Keep in mind that what we have here is the upper bound on the running time, not the exact running time.

Note that the running time on the preprocessing part (Algorithm 3) is dominated by the sorting of the whole set of nodes V. Assuming an efficient sorting algorithm such as Merge Sort, the running time is $O(n \log n)$, where $n = |V|$. This is commonly less than the enumeration time.

4.1.3 Experiment

We conducted experiments using our algorithms on several very large graphs. The data sets that we use in our experiments are listed in Appendix A. We used a Xeon E5620 machine with 64 GB of RAM. This machine has 16 threads in total. To utilize the multi-thread feature of the machine, we parallelize the node iteration in our code implementation (using Java-8).

In our experiment we used eleven graphs: `words`, `enron`, `uk-2007`, `cnr-2000`, `ljournal`, `uk-2002`, `arabic`, `uk-2005`, `webbase`, `twitter`, and `clueweb`, which are symmetrized to be undirected graphs. The order of the graphs ranges from ten thousand to close to a billion, and the size ranges from sixty four thousand to thirty

seven billion. The maximum degree for the smallest graph (`words`) is 332, while for the largest graph (`clueweb`) is more than 75 million.

We found that the pre-processing can reduce the effective maximum degrees of some graphs significantly, by four orders of magnitudes in the case of `clueweb` graph. As a result, the running time is much shorter, even after including the preprocessing time. We ran the triangle enumeration on both the preprocessed and unprocessed graphs. The reduction in running time, however, is less than the reduction in $d_{\max}^2$, in general. Recall that the $O(nd_{\max}^2)$ is only an upper bound, but not the exact formula for the running time. Perhaps, we should consider the average degree to get a better estimate. Moreover, parallelization also yields an overhead cost. It may also indicate that our code implementation can still be optimized further. Nevertheless, this reduction is enough to enable us to enumerate the triangles in `clueweb`, which has almost a billion nodes and forty billion edges in less than a day. It would not be possible without the preprocessing. We found that `clueweb` has almost 2 trillion triangles.

4.2 Four-Node-Graphlets

Consider four node graphlets in Figure 2.12. Notice that g_6, g_7 and g_8 contain triangle(s). A g_6 contains one triangle, a g_7 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, $(u, v, w, z)_8$ is a g_8 with nodes u, v, w and z. In enumerating the graphlets, some care is needed to avoid multiple

listing. Without loss of generality, we can use labels 1, 2, 3, and 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 $(1,2,3)_1$, $(1,3,2)_1$, $(2,1,3)_1$, $(2,3,1)_1$, $(3,1,2)_1$ and $(3,2,1)_1$. However, $(1,2,3)_1$ is the same wedge as $(3,2,1)_1$, $(1,3,2)_1$ is the same as $(2,3,1)_1$, and $(2,1,3)_1$ is the same as $(3,1,2)_1$. Thus, we have only three possible wedges, only one can be present (for induced case). Our convention is to list it with the smaller leg first, i.e. $(1,2,3)_1$, $(1,3,2)_1$, and $(2,1,3)_1$. We can divide these into two types: those with the smallest node at the center of the wedge (type 1), i.e., $(2,1,3)_1$, and those with the smallest node at one of the legs (type 2), i.e., $(1,2,3)_1$ and $(1,3,2)_1$. 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. Our convention is to list them such that the smallest node is in the first half: $(1,2,3,4)_3$, $(1,2,4,3)_3$, $(1,3,2,4)_3$, $(1,3,4,2)_3$, $(1,4,2,3)_3$, $(1,4,3,2)_3$, $(2,1,3,4)_3$, $(2,1,4,3)_3$, $(3,1,2,4)_3$, $(3,1,4,2)_3$, $(4,1,2,3)_3$, and $(4,1,3,2)_3$. Note that only one of them can exist.

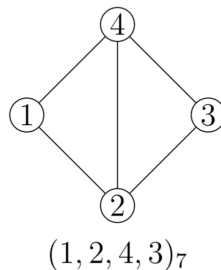
- For **3-stars**, we have four distinct ones depending on which one is the center. Our convention is to list the center first, and then the rest in order from smallest to largest. Thus, we have $(1,2,3,4)_4$, $(2,1,3,4)_4$, $(3,1,2,4)_4$, and $(4,1,2,3)_4$.

- 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. Suppose we list the nodes from the smallest

node, those three are distinguished by which node is opposite to this smallest one, i.e. at the third position. They are $(1,3,2,4)_5$, $(1,2,3,4)_5$, and $(1,2,4,3)_5$. We list them such that the second node is smaller than the fourth node.

- For **tailed-triangles**, the distinguishing nodes are the end node and the center node, giving us $\binom{4}{2}$ or twelve distinct configurations. We choose to list them by starting with the end node, followed by the center node, and then the last two nodes in ascending order. Thus we have $(1,2,3,4)_6$, $(2,1,3,4)_6$, $(1,3,2,4)_6$, $(3,1,2,4)_6$, $(1,4,2,3)_6$, $(4,1,2,3)_6$, $(2,3,1,4)_6$, $(3,2,1,4)_6$, $(2,4,1,3)_6$, $(4,2,1,3)_6$, $(3,4,1,2)_6$ and $(4,3,1,2)_6$.

- 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. Our convention is to start from the smaller opposing node and end with the other opposing node. The two connecting nodes are listed in between in ascending order. Thus we have: $(1,3,4,2)_7$, $(1,2,4,3)_7$, $(1,2,3,4)_7$, $(2,1,3,4)_7$, $(2,1,4,3)_7$, and $(3,1,2,4)_7$. This convention is illustrated in Figure 4.1.



$(1,2,4,3)_7$

Figure 4.1: For diamonds we start with the smaller opposing node and end with the other opposing node. In between we list the two connecting nodes in order. For example, in the diamond shown above 1 and 3 are the opposing nodes, and 2 and 4 are the connecting nodes. Therefore, it is listed as $(1,2,4,3)_7$.

- For **4-cliques**, we can swap any pair of nodes and still get the same clique. Thus there is only one unique configuration, and we choose to list the nodes in ascending order: $(1, 2, 3, 4)_8$.

4.2.1 Algorithm

For four-node graphlet enumeration, we built a set of algorithms that we call S4GE (Simultaneous 4-node Graphlets Enumeration). It works by first searching for triangles and wedges in the graph (as shown in Algorithm 4), and for each that is found then expand the search to find the four-node graphlets. Note that here we use the unprocessed graph as the input because wedge enumeration does not work with pre-processed one, i.e., we have to allow for lower neighbours as well.

Algorithm 4 TRIANGLE AND WEDGE ENUMERATION

Input: An undirected graph $G(V, E)$ in an adjacency list representation

```
1:  for all vertex u ∈ V(G) do
2:      for all vertex v ∈ N(u) do
3:          if u < v then
4:              for all u′ ∈ N(u) and v′ ∈ N(v) do
5:                  if (u′ > u) ∧ (v′ > u) then
6:                      if u′ = v′ > v then
7:                          EnumerateTriangle (u, v, u′)
8:                      if ((u′ < v′) ∨ (v′ = u)) ∧ (u′ > v) then
9:                          EnumerateWedgeType1 (v, u, u′)
10:                     if (u′ > v′) ∧ (v′ ≠ u) then
11:                         EnumerateWedgeType2 (u, v, v′)
```

When we find a triangle, we search for 4-cliques, diamonds, and tailed-triangles by using Algorithm 5. When we find a wedge, we search for rectangles, 3-stars, and 3-paths by using either Algorithm 6 or Algorithm 7 depending on the type of the wedge. There are two types of wedges, based on whether the smallest node is at the center (type 1) or not (type 2). Note that we have taken care of multiple

counting by imposing some conditions on the ordering of the node labels to follow the prescriptions above. Notice that S4GE enumerates all six types of non-isomorphic four-node graphlets in a single run.

Algorithm 5 Explore Triangle

Input: A triangle $(u, v, w)_2$, $u < v < w$, $N(u)$, $N(v)$, $N(w)$.
1: **for all** $z \in N(u) \cap N(v) \cap N(w)$ with $z > w$ **do**
2: Enumerate4Clique $(u, v, w, z)_8$
3: **for all** z in two sets and $z >$ opposite node **do**
4: EnumerateDiamond $(.)_7$
5: **for all** z in one set only **do**
6: EnumerateTailedTriangle $(.)_6$

Algorithm 6 Explore Wedge Type-1

Input: A type-1 wedge $(v, u, w)_1$, $u < v < w$, $N^{>u}(u)$, $N^{>u}(v)$, $N^{>u}(w)$.
1: **for all** $z \in N^{>u}(v) \cap N^{>u}(w)$ with $z \notin N^{>u}(u)$ **do**
2: EnumerateRectangle $(u, v, z, w)_5$
3: **for all** $z \in N^{>u}(u)$ only **do**
4: **if** $z > w$ **then**
5: Enumerate3Star $(u, v, w, z)_4$
6: **for all** $z \in N^{>u}(v)$ only **do**
7: Enumerate3Path $(w, u, v, z)_3$
8: **for all** $z \in N^{>u}(w)$ only **do**
9: Enumerate3Path $(v, u, w, z)_3$

4.2.2 Analysis

Let us first prove the correctness of our algorithm. We need to show that all of the graphlets would be found and listed just once.

Theorem 3. *Algorithm 4,* Triangle and Wedge Enumeration*, correctly enumerates wedges and triangles in an undirected graph.*

Algorithm 7 Explore Wedge Type-2

Input: A type-2 wedge $(u, v, w)_1$, $u < v$, $u < w$, $N^{>u}(u)$, $N^{>u}(v)$, $N^{>u}(w)$.

1: **for all** $z \in N^{>u}(v)$ only **do**
2:     **if** $z > w$ **then**
3:         Enumerate3Star $(v, u, w, z)_4$
4: **for all** $z \in N^{>u}(w)$ only **do**
5:     **if** $z \neq v$ **then**
6:         Enumerate3Path $(u, v, w, z)_3$

Proof. Each edge uv is iterated once and only once, when $u < v$. 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 $u < v'$ (line 5) there will be no double counting. □

Theorem 4. *Algorithms 5, 6 and 7, combined with Algorithm 4, 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 (Algorithm 5). 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, either by Algorithm 6 or Algorithm 7. 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 5. *The runtime of S4GE is bounded by* $O((N_\Delta + N_\angle)\, d_{\max} + T_{3g})$*, where* N_Δ *(*$N_\angle$*) is the number of triangles (wedges), and* T_{3g} *is the time to enumerate the triangles and wedges.*

Proof. First, S4GE searches for triangles and wedges, using T_{3g} time. For each triangle and wedge the algorithm runs through the neighbor sets to check the intersections with cost $\leq (d(u) + d(v) + d(w)) \leq 3d_{\max}$. □

Note that in general $(N_\Delta + N_\angle) \lesssim nd^2_{\max}$, with the upper value is satisfied by a regular graph. However, for all real-world networks, we have $(N_\Delta + N_\angle) \ll nd^2_{\max}$. Also, $T_{3g} \ll nd^2_{\max}$ using efficient enumeration. Therefore, in practice, our runtime is much less than worst case bound of $O(nd^3_{\max})$ from general DFS algorithms [67].

4.2.3 Experiment

For experimenting on S4GE we chose the following graphs: `enron`, `cnr-2000`, `dblp-2011`, `amazon-2008`, `dewiki-2013`, and `ljournal`. The networks that we studied are listed in Table 4.1. All of the datasets were downloaded from the Laboratory for Web Algorithmics [11, 10], `http://law.di.unimi.it/datasets.php`. We symmetrized them and took out any self-loops to get simple undirected graphs. We implemented our code in Java, using the WebGraph library [11]. 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 basic properties of the graph datasets are listed in Table 4.1. We include a column for the effective maximum degree after preprocessing, $d^{\text{BG}}_{\max}$. However, note

that preprocessed graphs can only be used for the triangle enumeration part, but not for the wedge enumeration part. We also include the average degree in the original graph, d_{avg}. As we can see, the average degree is much smaller than the maximum degree for these graphs, indicating a power-law degree distribution.

Dataset	n	m	d_{max}	$d_{\text{max}}^{\text{BG}}$	d_{avg}
`enron`	69,244	254,449	1,634	87	7.35
`cnr`	325,557	2,738,969	18,236	85	16.83
`dblp`	986,324	3,353,618	979	118	6.80
`amazon`	735,323	3,523,472	1,077	16	9.58
`dewiki`	1,532,354	33,093,029	118,246	490	43.19
`ljournal`	5,363,260	49,514,271	19,432	756	18.46

Table 4.1: The undirected graphs. Here, $d_{\text{max}}^{\text{BG}}$ is the effective maximum degree when only larger neighbours are included after the preprocessing.

The graphlet counts that we got as the results of our enumeration are listed in Table 4.2. We can check that for all of these graphs, $N_\Delta + N_\angle \ll n d_{\text{max}}^2$ using their d_{max} values from Table 4.1. For example, for amazon we have $N_\Delta + N_\angle \approx 42\text{M}$ and $n d_{\text{max}}^2 \approx 853\text{B}$, 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.

Note that, even though `dewiki` has smaller order and size compared to `ljournal`, it has a much larger maximum degree, and a larger average degree. We found that `dewiki` has an enormous number of wedges, more than 51 billion. Because of this, we actually were not able to finish the enumeration, hence we do not have the 4-node counts for this graph.

The runtimes are shown in Table 4.3. We include the triangle enumeration time, T_Δ, for comparison. For this triangle enumeration, we use the pre-processed graphs. However, wedges cannot take advantage of the pre-processing, so they require longer

Graph	g_1	g_2	g_3
`enron`	40,309,453	1,067,993	2,511,039,670
`cnr`	7,798,287,209	20,977,629	6,118,026,632
`dblp`	81,529,950	7,005,235	2,678,518,695
`amazon`	38,015,403	4,464,791	372,366,885
`dewiki`	51,141,107,679	88,611,282	..
`ljournal`	8,726,048,197	411,155,444	1,812,284,632,329

Graph	g_4	g_5	g_6
`enron`	8,043,804,283	21,598,984	582,841,848
`cnr`	41,392,015,937,553	37,876,822,234	79,429,334,745
`dblp`	3,545,925,764	1,483,611	543,447,587
`amazon`	609,961,827	2,689,696	9,232,707
`dewiki`	..	..	..
`ljournal`	8,847,128,736,944	8,551,292,956	189,716,360,703

Graph	g_7	g_8
`enron`	46,141,288	5,001,773
`cnr`	42,974,515,602	159,814,399
`dblp`	21,608,538	40,910,658
`amazon`	13,096,219	4,192,682
`dewiki`	..	..
`ljournal`	26,962,410,402	16,129,080,442

Table 4.2: Counts of the graphlets. The `dewiki` dataset needs longer than our time limit to terminate.

enumeration time, hence T_{3g}, which includes the time to enumerate both triangles and wedges, is larger than T_Δ. For `enron`, `dblp` and `amazon`, the numbers of wedges (and triangles) are not very large. Therefore, for these graphs, T_{3g} is not much larger than T_Δ, as both are actually still dominated by the overhead cost. On the other hand, `cnr` has a very large number of wedges, and we see that T_{3g} is an order of magnitude larger than T_Δ.

Note that T_{4g}, 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 $(N_\angle + N_\Delta)\, d_{\max}$ values is about four thousand, while the ratio of their T_{4g} values is about six thousand, i.e. approximately the same order. This observation experimentally validates the statement of Theorem 5 relating the runtime to the $(N_\angle + N_\Delta)\, d_{\max}$ value.

Interestingly, `cnr` requires a 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 an enormous number of wedges and a large maximum degree. The program did not terminate even after running for four days, and we had to abort the run.

4.3 Five-Node Graphlets

A direct continuation of our work on four-node graphlets would be to look for algorithms to enumerate larger graphlets. Here we propose a solution to enumerate five-node graphlets.

Graph	T_Δ	T_{3g}	T_{4g}
enron	1.03	3.49	76.50
cnr	1.75	57.03	176K
dblp	1.93	3.45	62.05
amazon	1.73	2.53	14.05
dewiki	12.79	517.3	$>$ 300K
ljournal	32.96	257.1	82K

Table 4.3: The runtime, in seconds, for triangle enumeration T_Δ, for wedges and triangles together T_{3g}, and for all three and four-node graphlets together T_{4g}.

4.3.1 Idea

Looking at Fig. 2.13, the five-node graphlets, we notice that all of them contain triangles and/or wedges. There are 15 out of 21 that contain one or more triangles. Now, we are going to show that they can be discovered directly through either a triangle or a wedge. Recall that, in S4GE, in order to find four node graphlets, once a triangle or a wedge is found, say $\Delta(u, v, w)$ or $\angle(u, v, w)$, we line up the neighbour sets of the three nodes and check for any intersections. Through this process, we can divide the neighbour sets into three categories: those nodes that are neighbours to only one, two or all three of u, v, and w. With $N(u)$ is the neighbour set of u, $N(v)$ is the neighbour set of v and $N(w)$ is the neighbour set of w, we have

$$\begin{aligned} N(u) \cup N(v) \cup N(w) &= N_1(u) \cup N_1(v) \cup N_1(w) \cup \\ &\quad N_2(u,v) \cup N_2(u,w) \cup N_2(v,w) \cup \\ &\quad N_3(u,v,w) \end{aligned} \tag{4.1}$$

where $N_1(u)$ is the set of neighbours of u that are not neighbours of v or w, and similarly for the other sets. Note that the sets on the right hand side are all disjoint from each other, while the ones on the left hand side are in general not. Also, note

that for example,

$$N(u) = N_1(u) \cup N_2(u,v) \cup N_2(u,w) \cup N_3(u,v,w) \tag{4.2}$$

and similarly for $N(v)$ and $N(w)$. Now, once we have this division, we can take two or more of these sets simultaneously and check if the nodes in those sets are connected, to find five-node graphlets. For example, if we take $N_1(u)$ and $N_1(v)$ we can get some g_{12} (if the two nodes are not connected) and g_{21} (if the two nodes are connected). This is illustrated in Figure 4.2.

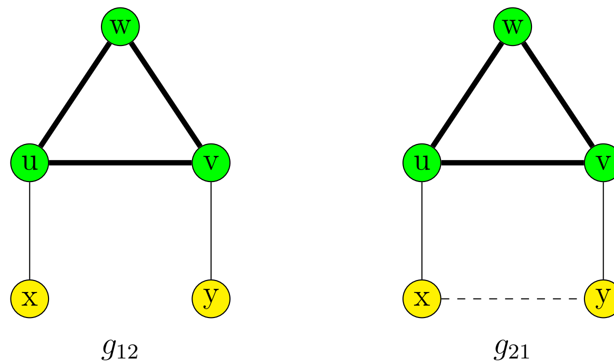


Figure 4.2: Finding g_{12} and g_{21} through a triangle $\Delta(u,v,w)$. Here $x \in N_1(u)$ and $y \in N_1(v)$. We get a g_{12} when x and y are not connected, or g_{21} when x and y are connected.

We can also take a set and combine it with itself, e.g., $N_1(u)$ with $N_1(u)$, to find two-tailed triangles g_{14}, and also g_{18}. We can prove that we can find all of the five-node graphlets through this method, either through a triangle or a wedge. The g_{13} is a special case. If we look at the triangle inside it, we might think that we need to go two steps from the triangle and that this method of taking two neighbour sets at a time does not work. That is, we need to search for the neighbours of the tail and find the ones that are not connected to any of u, v, or w. However, if we start from a wedge instead, then we will see that we can find g_{13} through this method as well,

e.g., by taking $N_1(u)$ and $N_2(v, w)$. This is illustrated in Figure 4.3. Thus, out of the 21 types, 7 of them need to be discovered through wedges, while the rest can be discovered through triangles.

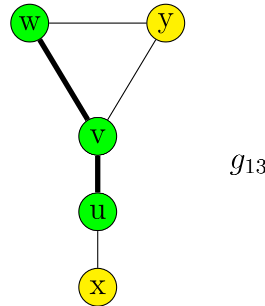


Figure 4.3: Finding a g_{13} through a wedge $\angle(u, v, w)$. Here $x \in N_1(u)$ and $y \in N_2(v, w)$, and x and y are not connected.

For a proper enumeration, we still need to make sure that there is no double counting while including all of the graphlets. For this, we have to consider each graphlet type separately and utilize the symmetrical properties of the graphlet. For example, if we look at g_{13} in Figure 4.3, we see that it is symmetric under the interchange $w \leftrightarrow y$. In other words, we would get the same g_{13} when $w = 5, y = 8$ or when $w = 8, y = 5$, for example, with all the other nodes being the same. Thus, to avoid double counting we can impose a condition that we only enumerate when $w < y$. Note that x can be greater or lower than (u, v, w), hence no condition shall be imposed on x. Notice also that, assuming $u < v$ and $u < w$ (but no ordering imposed between v and w), here we use a type 2 wedge $\angle(u, v, w)$ where the smallest node is at one of the legs. To find all g_{13} in the graph, we need to consider $N_1(w)$ together with $N_2(u, v)$ as well, and also the type 1 wedges, i.e., $\angle(v, u, w)$, where the smallest node is at the centre of the wedge.

Furthermore, there are graphlets that can be discovered through more than one combination. For example, consider g_{26}. Using our method, there are three ways

leading to g_{26}, as illustrated in Figure 4.4. From a triangle, we can take an N_2 and combine it with itself; we get a g_{26} when the two neighbours are connected. We can also take N_3 and N_2; we get a g_{26} when the two neighbours are not connected. Or, we can take N_3 and N_1; we get a g_{26} when the two neighbours are connected.

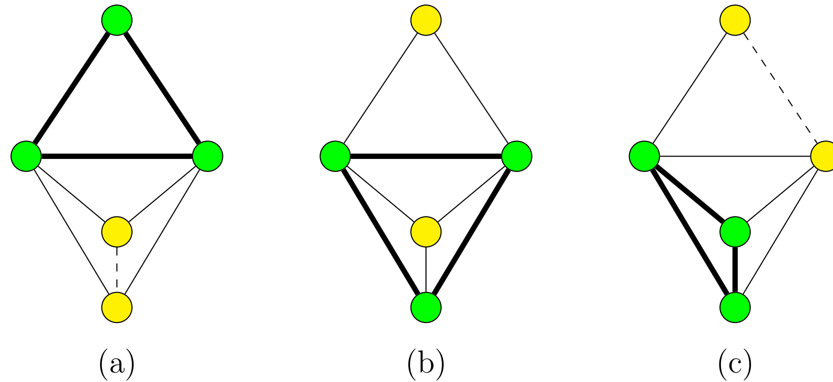


Figure 4.4: There are three ways leading to g_{26}: (a) N_2 with itself, connected; (b) N_2 and N_3, not connected; (c) N_1 and N_3, connected.

The question now is how do we make sure that there is no multiple counting while also making sure that all g_{26} will be counted? Let us look at the symmetry of g_{26}, as shown in Figure 4.5. We see that g_{26} is isomorphic under $a \leftrightarrow b$, and $c \leftrightarrow d$. Now, suppose we start with a triangle $\Delta(a, b, x)$ and proceed on finding g_{26} via $N_2(a, b)$ with itself (Figure 4.4(a)), then we can avoid double counting by imposing condition $c < d$. It is implicit from the triangle that $a < b$ is already imposed (i.e., one node must be lower than the other).

Note that, at this moment, there is no condition imposed between (a, b) and (c, d). However, we can also go through Figure 4.4(b) and (c), and the results may repeat those from (a). This double-counting problem can be solved if we impose that the yellow nodes in Figure 4.4(a) are higher than the green nodes. Similarly for Figure 4.4(b). For Figure 4.4(c), however, we need to allow the top yellow node (i.e., the element of N_1) to be either greater or lower than the green nodes. To show and

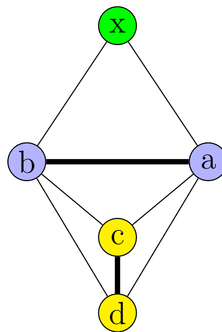


Figure 4.5: The symmetry of g_{26}.

prove that we will get all g_{26} without double counting through this method, let us think of the nodes as labeled by 1, 2, 3, 4, 5, with $1 < 2 < 3 < 4 < 5$. We just need to show that we cover all possible positions for the two highest labels 4 and 5. This is shown in Figure 4.6 where the two highest labeled nodes are the yellow ones. These can be viewed as all possible combinations of orbits of two nodes (See Figure 2.14).

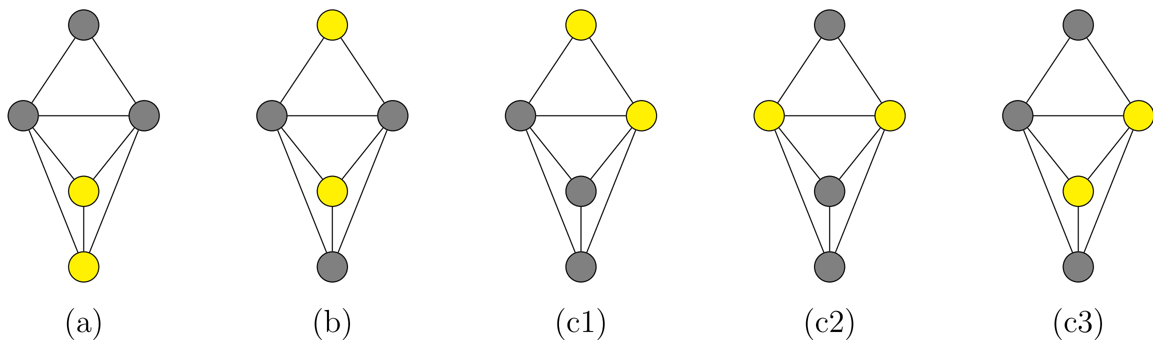


Figure 4.6: Listing all possible combinations in g_{26}. The two yellow nodes are the two highest labeled nodes, 4 and 5. In (c1), (c2), and (c2), we allow the node at the top of the figure to be either higher or lower than the others.

Other 5-node graphlets that can be discovered in multiple ways through triangles are g_{24} and g_{28}. The case for g_{24} is shown in Figure 4.7. It can be seen that g_{24} can be discovered through a triangle and either by two unconnected N_2 or by connected N_1 and N_2. Note that there is only one symmetry for g_{24}, that is the up-down mirror

symmetry. In this case, we can see that if we allow the neighbours to be of higher or lower labels in Figure 4.7(a) we would already cover all possible patterns. Thus, we do not need Figure 4.7(b). With the triangle fixed (say the lower positioned node is assumed to have a lower label than the higher positioned node), we do not need to worry about double counting from the mirror symmetry.

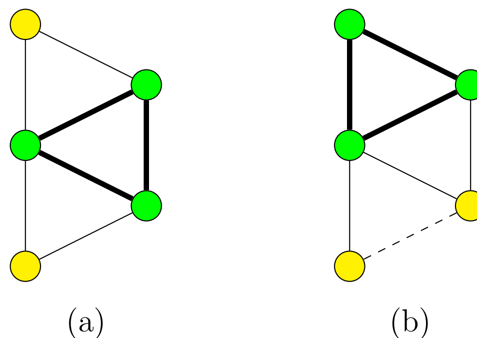


Figure 4.7: There are two ways leading to g_{24}: (a) two N_2, not connected; (b) N_1 and N_3, connected.

The case for g_{28} is shown in Figure 4.8. We see that, from a triangle, we can get a g_{28} through N_2 and N_3 connected, or through N_3 with itself, not connected. From Figure 2.14 we can see that g_{28} has two types of orbits. If we go via Figure 4.8(b), then with the triangle fixed, the only symmetry is on the two neighbour nodes (the yellow nodes). This can easily be taken care of since both are coming from the same set N_3, to make sure that there is no double counting. If we allow the yellow nodes to be either higher or lower than the green nodes all g_{28} graphlets would be found through this way. Thus, we do not need to consider Figure 4.8(a).

Next, move on to 5-node graphlets that can be discovered through wedges, there are also some that can be discovered in more than one way. In fact, the g_{13} that we discussed earlier can also be discovered through $N_1(v)$ with $N_1(v)$ when the two neighbours are connected. Here, we assume v is one of the wedge legs. These two ways

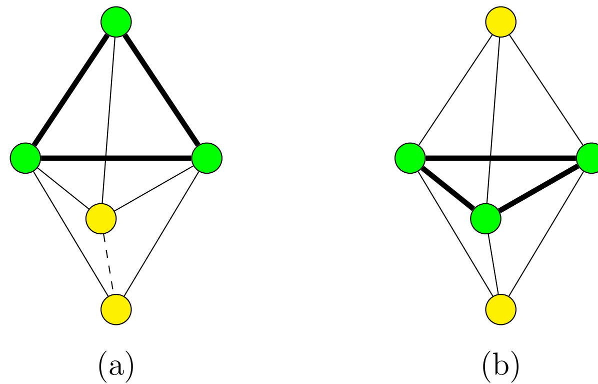


Figure 4.8: There are two ways leading to g_{28}: (a) N_2 with N_3, connected; (b) N_3 with itself, not connected.

can be seen in Figure 4.9. However, following our previous discussion, all of the g_{13} can already be discovered through (a), when the bottom node is let free. Therefore, (b) is not needed.

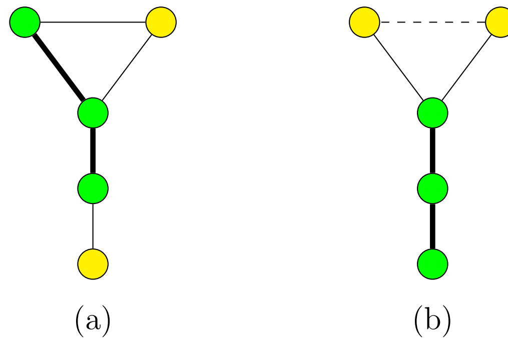


Figure 4.9: There are two ways of finding a g_{13} through a wedge (green nodes): (a) $N_1(v)$ with $N_2(u, w)$, not connected, and (b) $N_1(v)$ with itself, connected. For the sake of notation, here we assume u is the center node of the green wedge.

Other 5-node graphlets that can be discovered through wedges in multiple ways are g_{10}, g_{16}, and g_{20}. The case for g_{10} can be seen in Figure 4.10. The symmetry looks the same as the one for g_{13}. By the same argument, we only need Figure 4.10(a) to enumerate all g_{10} in a graph, with no need for (b).

Next, g_{16} can be discovered in three ways, as shown in Figure 4.11. If we let the

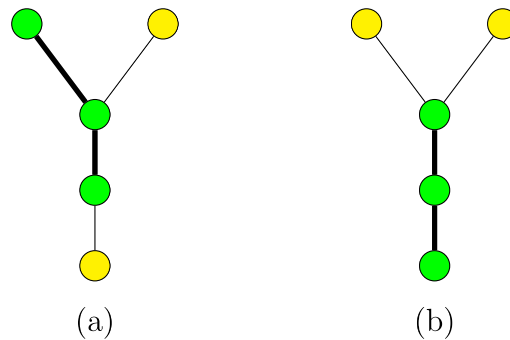


Figure 4.10: There are two ways of finding a g_{10} through a wedge (green nodes): (a) $N_1(v)$ with $N_1(u)$, not connected, and (b) $N_1(v)$ with itself, also not connected. For the sake of notation, here we assume u is the center node of the green wedge.

two yellow nodes be free, so that they can be higher or lower than the other nodes, then we can show that all g_{16} would be found through (b). We just need to make sure that we do not double-count the wedge, which we have already done. Thus, for g_{16} we do not need to consider (a) and (c).

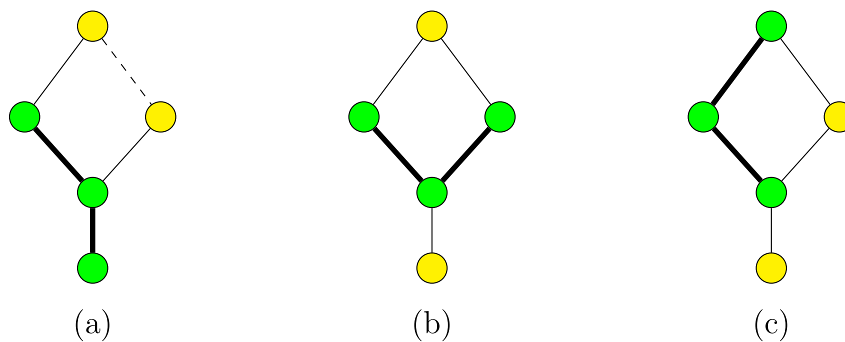


Figure 4.11: There are three ways of finding a g_{16} through a wedge (green nodes): (a) $N_1(v)$ with $N_1(u)$, connected, (b) $N_1(u)$ with $N_2(v, w)$, not connected, and (c) $N_1(v)$ with $N_2(v, w)$, not connected. For the sake of notation, here we assume u is the center node of the wedge.

Lastly, for g_{20}, there are two ways as shown in Figure 4.12. This graphlet has two symmetries: between the two nodes with degree three, and among the nodes with degree two. All g_{20} can be found through (b) if we let the yellow nodes be either

higher or lower than the green nodes. Multiple counting can be avoided if we impose that the central green node is lower than the yellow nodes.

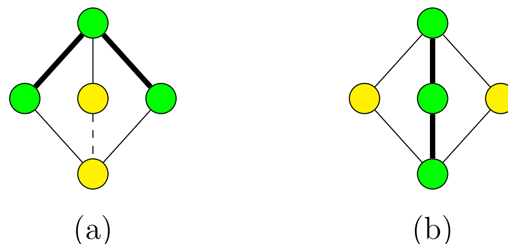


Figure 4.12: There are two ways of finding a g_{20} through a wedge (green nodes): (a) $N_1(u)$ with $N_2(v,w)$, connected, and (b) $N_2(v,w)$ with $N_2(v,w)$, not connected. For the sake of notation, here we assume u is the center node of the wedge.

4.3.2 Algorithm

The algorithm for enumerating five-node graphlets can be considered as an extension of the S4GE. Let us call the whole set S5GE. First, we find triangles and wedges in the same way as we did in S4GE. This part is similar to Algorithm 4 of S4GE. The only difference is in the procedures that are called from within. When a triangle is found, we call the Extended Explore Triangle procedure. This procedure works similarly to the Explore Triangle procedure of S4GE, and in fact also enumerates the 4-node graphlets along the way. However, here we have more to be done: to build the sets $N_1(u), \ldots, N_3(u,v,w)$, and after that, to call all the 5GT functions to enumerate five node graphlets that are attached to the triangle. Because of the long list of algorithms, we put the S5GE algorithms in Appendix B. The Extended Explore Triangle is shown in Algorithm 22. Similarly, for wedges, we have the Extended Explore Wedges procedures, shown in Algorithms 23 and 24. The corresponding functions for wedges are called 5GW1 and 5GW2 respectively.

There are nine 5GT functions: one that pairs an N_1 with itself (Algorithm 25),

one that pairs an N_1 with another N_1 (Algorithm 26), one that pairs an N_1 with an overlapping N_2 (Algorithm 27), one that pairs an N_1 with a non-overlapping N_2 (Algorithm 28), one that pairs an N_2 with itself (Algorithm 29), one that pairs an N_2 with another N_2 (Algorithm 30), one that pairs an N_1 with N_3 (Algorithm 31), one that pairs an N_2 with N_3 (Algorithm 32), and one that pairs N_3 with itself (Algorithm 33). We summarize these functions and their outputs in Table 4.4. We should mention here that, because of the triangle symmetry, we should treat u, v and w equally. Although we list $N_1(u)$ and $N_1(u)$ for 5GT-UU, it is also ran over $N_1(v)$ (paired with itself) and $N_1(w)$ (paired with itself). Similarly, the other functions are also ran over all possible choices of u, v, w.

Function	Pairing		Discover
5GT-UU	$N_1(u)$	$N_1(u)$	g_{14}, g_{18}
5GT-UV	$N_1(u)$	$N_1(v)$	g_{12}, g_{21}
5GT-U2UV	$N_1(u)$	$N_2(u, v)$	g_{17}, (g_{24})
5GT-U2VW	$N_1(u)$	$N_2(v, w)$	g_{19}, g_{25}
5GT-2UV2UV	$N_2(u, v)$	$N_2(u, v)$	g_{22}, g_{26}
5GT-2UV2UW	$N_2(u, v)$	$N_2(v, w)$	g_{24}, g_{27}
5GT-U3UVW	$N_1(u)$	$N_3(u, v, w)$	g_{23}, g_{26}
5GT-2UV3UVW	$N_2(u, v)$	$N_3(u, v, w)$	g_{26}, (g_{28})
5GT-3UVW3UVW	$N_3(u, v, w)$	$N_3(u, v, w)$	g_{28}, g_{29}

Table 4.4: The 5GT functions. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions.

The 5GW1 and 5GW2 functions are only to discover (and enumerate) the rest of the 5-node graphlets: g_9, g_{10}, g_{11}, g_{13}, g_{15}, g_{16}, and g_{20}. Therefore, not all of the neighbour sets are needed. That is, we should not include those that form a triangle, except for the one in g_{13}. However, for the wedges, we need to differentiate whether we are looking at the neighbours of the centre node or the leg nodes. The 5GW1 functions are summarized in Table 4.5.

Function	Pairing		Discover
5GW1-UU	$N_1(u)$	$N_1(u)$	g_{11}, (g_{14})
5GW1-VV	$N_1(v)$	$N_1(v)$	(g_{10}), (g_{13})
5GW1-UV	$N_1(u)$	$N_1(v)$	g_{10}, (g_{16})
5GW1-VW	$N_1(v)$	$N_1(w)$	g_9, g_{15}
5GW1-U2VW	$N_1(u)$	$N_2(v, w)$	g_{16}, (g_{20})
5GW1-V2UW	$N_1(v)$	$N_2(u, w)$	g_{13}, (g_{21})
5GW1-V2VW	$N_1(v)$	$N_2(v, w)$	(g_{16}), (g_{21})
5GW1-2VW2VW	$N_2(v, w)$	$N_2(v, w)$	g_{20}, (g_{25})

Table 4.5: The 5GW1 functions. The input is a type-1 wedge with the smallest node u at the center of the wedge. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions.

Notice that we skip 5GW1-U2UV since the 5-node graphlets have been enumerated through triangles. From the table shown, we can see that we actually do not need 5GW1-VV and 5GW1-V2VW either. Note that we need to run some functions, such as 5GW1-UV, on the other leg, with v replaced by w.

For 5GW2 the input is a type-2 wedge with the smallest node u at one of the legs of the wedge. By convention, we list u as the first node and denote the center node as v, i.e., $\angle(u, v, w)$. Note that $u < v$ and $u < w$, but there is no ordering imposed between v and w. This is different from the type-1 above. The 5GW2 functions are similar to the 5GW1 functions but with the roles of u and v exchanged. However, there are some other subtle, yet important, differences. These functions are summarized in Table 4.6. They can be seen in Appendix B.

4.3.3 Analysis

Theorem 6. S5GE *correctly enumerates all five node graphlets in an undirected graph.*

Function	Pairing		Discover
5GW2-VV	$N_1(v)$	$N_1(v)$	g_{11}, (g_{14})
5GW2-UV	$N_1(u)$	$N_1(v)$	g_{10}, (g_{16})
5GW2-UW	$N_1(u)$	$N_1(w)$	g_9, (g_{15})
5GW2-V2UW	$N_1(v)$	$N_2(u, w)$	g_{16}, (g_{20})
5GW2-U2VW	$N_1(u)$	$N_2(v, w)$	g_{13}, (g_{21})
5GW2-2UW2UW	$N_2(v, w)$	$N_2(v, w)$	g_{20}, (g_{25})

Table 4.6: The 5GW2 functions. The input is a type-2 wedge with the smallest node u at one of the legs of the wedge, and the center node is v. The first graphlet in each list is for when the two neighbours are not connected, while the second one is for when the two neighbours are connected. The graphlets in the parentheses are not enumerated as they are already discovered by some other functions.

Proof. As has already been proven in the case of S4GE, all triangles and wedges would be found and enumerated once. For each, the neighbours of the three nodes are in either one of the N_1, N_2, or N_3 sets. By considering all possible combinations of two of these sets, all 5-node graphlets that contain the triangle or the wedge would be found. Since any 5-node graphlet contains at least a triangle or a wedge, all 5-node graphlets in the input graph would be found. Multiple countings are avoided by considering the symmetrical properties of the graphlets. □

Theorem 7. *The runtime of* S5GE *is bounded by* $O((N_\Delta + N_\angle)\, d_{\max}^2 + T_{3g})$*, where* N_Δ *(*$N_\angle$*) is the number of triangles (wedges), and* T_{3g} *is the time to enumerate the triangles and wedges.*

Proof. First, S5GE searches for triangles and wedges, using T_{3g} time. For each triangle and wedge, the algorithm runs through the neighbor sets to check for intersections with cost $\leq (d(u) + d(v) + d(w)) \leq 3d_{\max}$, forming the disjoint neighbour sets N_1, N_2 and N_3. It then runs the 5G functions, where each checks the neighbours of the neighbours to see any connections. The cost is $d(x)$. Thus, overall we have $O(d_{\max}^2)$. □

Note: The T_{3g} is typically smaller, and we can simply write the bound as $O((N_\Delta +$

$N_\angle)\, d_{\max}^2)$.

4.3.4 Experiment

We implemented the S5GE algorithm in Java, and experimented on several graphs: `wordassociation`, `enron`, `amazon` and `dblp`. They are chosen since they have relatively small maximum degrees. We used a Xeon machine with 24 threads and 64 GB of RAM. The running times are listed in Table 4.7. We compare these with the running times of S4GE using the same machine.

Graph	T_{4g}	T_{5g}	$d_{\max}$
`wordassociation`	1.77	133.43	332
`enron`	73.34	47,411.44	1,634
`amazon`	13.09	1,402.82	1,077
`dblp`	63.73	17,826.07	979

Table 4.7: The runtime, in seconds, for S4GE T_{4g}, and S5GE T_{5g}. As a reference, we also list the maximum degree of each graph.

It is interesting to note that T_{5g}/T_{4g} is generally smaller than $d_{\max}$. This validates our analysis that the difference between T_{5g} and T_{4g} is bounded by a factor $d_{\max}$.

Due to the time constraint, we do not run S5GE on any other graphs that we have (Appendix A). For `ljournal`, for example, T_{4g} is already around 1 day, and its $d_{\max}$ is just below 20 thousand. If we assume a factor of 1000, as a rough estimate, we already need 1000 days to run the enumeration on the same machine.

The runs yielded graphlet counts that are listed in Table 4.8. The largest count in this table is 2.6 trillions for g_{11} in `enron`. We see that g_{10} and g_{11} tend to have the largest counts in each graph.

Graphlet	`wordassociation`	`enron`	`amazon`	`dblp`
g_1	2,181,681	40,309,965	38,029,668	81,534,875
g_2	61,795	1,067,993	4,464,791	7,005,235
g_3	61,182,256	2,511,108,017	372,441,882	2,678,632,511
g_4	59,531,563	8,043,807,281	609,970,485	3,545,929,648
g_5	214,883	21,598,984	2,689,696	1,483,611
g_6	7,383,771	582,841,848	92,327,207	543,447,587
g_7	339,674	46,141,288	13,096,219	21,608,538
g_8	28,258	5,001,773	4,192,682	40,910,658
g_9	2,372,920,633	237,853,341,447	5,538,677,838	104,932,594,398
g_{10}	9,058,521,999	2,488,523,105,141	24,161,440,795	844,270,600,140
g_{11}	4,183,563,421	2,614,152,202,537	76,617,522,566	542,061,274,958
g_{12}	284,095,186	100,576,301,784	1,046,863,494	23,191,416,092
g_{13}	820,768,531	169,356,292,757	1,464,098,155	59,952,351,960
g_{14}	514,678,642	156,468,737,188	1,846,987,095	41,790,666,219
g_{15}	2,175,530	441,436,983	4,750,305	18,991,534
g_{16}	636,893,718	316,091,186,540	844,908,317	16,123,769,262
g_{17}	88,117,927	37,275,612,382	354,792,620	4,768,657,594
g_{18}	5,568,058	1,590,007,904	29,777,114	1,220,161,037
g_{19}	117,731,184	38,994,981,365	447,001,216	6,000,243,276
g_{20}	522,101	3,751,148,426	2,502,602	21,280,525
g_{21}	2,071,516	992,955,577	10,351,656	35,091,797
g_{22}	4,738,106	8,648,341,377	56,226,399	845,154,386
g_{23}	59,455,510	42,038,568,231	449,075,244	21,883,544,694
g_{24}	6,737,933	4,784,316,007	74,061,563	305,618,895
g_{25}	93,904	171,971,082	1,344,863	1,617,053
g_{26}	246,608	793,982,638	8,313,900	265,088,604
g_{27}	8,520	27,486,607	191,230	169,035
g_{28}	591,109	3,918,967,836	64,332,031	8,626,672,675
g_{29}	2,044	16,894,713	1,263,171	531,434,127

Table 4.8: Graphlet counts, output of S5GE.

4.4 GDV

We can extend our solution for enumerating graphlets to find the Graphlet Degree Vector (GDV) of each node. In this case, we need to create a vector of 73 elements for each node and initiate with values 0. The first element is just the degree of the node. We then run S5GE. Every time we enumerate a 3/4/5-node graphlet, we check the nodes inside it and their orbits. Then, we increase the associated elements in their GDV by one. Since each graphlet is enumerated once, we ensure that we count the orbits correctly. Therefore, by the end of the run, we would have the correct GDV for every node. Note, however, that the memory space requirement may restrict the scalability of this solution.

4.5 Beyond 5 Nodes

An idea similar to the one we used for 5-node graphlets can, in principle, be employed to find even larger graphlets. Taking three neighbouring nodes of a triangle or a wedge simultaneously we can find 6-node graphlets. An example of this is shown in Figure 4.13. Take four neighbouring nodes simultaneously to find 7-node graphlets, and so on. Note, however, that not all higher order graphlets can be found through this method, at least not without any extended computation. For example, a path and a cycle of 6 nodes.

We can see that the number of possible configurations grows fast. For 6-node graphlets, we need three neighbour nodes. For 7-node graphlets, we need four neighbour nodes. While there are two non-isomorphic graphs of two nodes, there are four of three nodes, and eleven for four nodes. In addition, we also need to consider the difference when a connection is between two N_1 nodes, or between an N_1 and an N_2 nodes, for example. The number of combinations grows as well. With seven neigh-

bour sets to choose from (see Eq. 4.1), with duplicates the number grows as 7^N (but is reduced by symmetry), where N is the number of neighbour nodes to consider at a time.

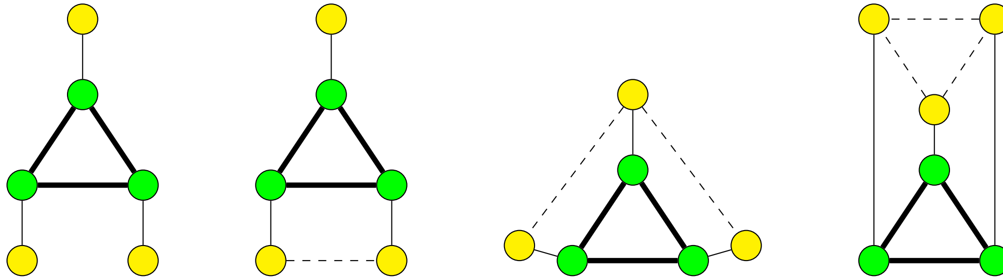


Figure 4.13: Finding 6-node graphlets through a triangle. Here we take $N_1(u)$, $N_1(v)$ and $N_1(w)$. We get four types of 6-node graphlets depending on whether we have zero, one, two, or three edges among the neighbour nodes.

Moreover, again, we need to make sure that we do not count any graphlet twice. The analytical complexity is higher than what we have done so far. Recall that there are 112 types of six-node graphlets and 853 types of seven-node graphlets, and many of them can be found in multiple ways. We may need to find a different method to enumerate higher-order graphlets. Alternatively, we can use flags to let us know if a graphlet has already been enumerated.

Also, the running time would restrict the order of the graph that we can enumerate. We may have to be less ambitious here, and instead of trying to enumerate all types at once, like what we do with four and five-node graphlets, we can look at some but not all of the patterns at a time. We may find some optimization that works for specific patterns. For example, as shown by Danisch et al [22], cliques of higher order can be enumerated by using a specialized algorithm.

Chapter 5

Directed Graphlets Enumeration

So far we have been focusing our study on undirected graphs. However, many real-world networks are directed, where the relations among the nodes are not symmetric, but have directions from one node to another. Note, however, that a pair of nodes can have two relations that are opposite to each other (i.e., bidirectional). Let us now turn our attention to subgraphs in directed graphs. We assume directed graphs as the input of our enumeration throughout this chapter. Therefore, we will simply use the notation G to denote a directed graph here.

5.1 Directed Triangles

Given a directed graph, we can enumerate the directed triangles inside it. Here, by a triangle, we mean a subgraph of three nodes and three (directed) edges. There are only two kinds of directed triangles: cyclic triangle and trust triangle. In general, they are not induced subgraphs, as explained in Chapter 2. Directed triangle enumeration has some similarities to undirected triangle enumeration, especially for the trust triangle.

5.1.1 Algorithms

The algorithm that we use to enumerate trust triangles is described in Algorithm 8. Notice the similarity between this algorithm and Algorithm 1. The difference is that here we do not put any restriction on the order of the labels u, v, and w. Thus, u can be greater or smaller than v and/or w, and also v can be greater or smaller than w. Here, we assume that the graph is simple, with no multi-edge or self-loop, so $u \neq v \neq w$. We only need the directed graph as input for this algorithm, without the transpose graph. However, note that this algorithm can also be applied to the transpose graph by itself, without the original graph. This algorithm involves computing the intersection between two neighbour sets. The algorithm for this is given in Algorithm 2.

Algorithm 8 TRUST TRIANGLE ENUMERATION

Input: A directed graph $G = (V, E)$
Output: The list and number of trust triangles in G.

```
1: S ← ∅, k ← 0, c ← 0
2: for all nodes u ∈ V do
3:     for all nodes v ∈ N⁺(u) do
4:         (S, k) ← INTERSECTION(N⁺(u), N⁺(v), d⁺(u), d⁺(v))
5:         for all nodes w ∈ S do
6:             ENUM ((u, v, w)_T)                    ▷ For Listing
7:         c ← c + k
8: return c
```

For a trust triangle, we can label the edges as the first, second and third edges, as illustrated in Figure 5.1.

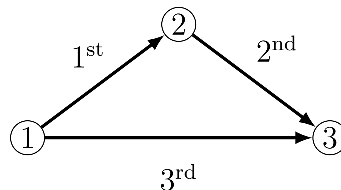


Figure 5.1: A trust triangle, $(1, 2, 3)_T$ and the labeling of its edges.

Notice that if we flip the third edge in a trust triangle, we will get a cycle triangle. Based on this observation, we can modify the algorithm for trust triangle enumeration to enumerate cyclic triangles. The result is described in Algorithm 9. In this case, it requires as input both the graph and its transpose. The in-going neighbour set of u is read from the out-going neighbour set of u in the transpose graph, i.e., $N^-(u,G) = N^+(u,G^T)$. Notice that in INTERSECTION call we use $N^-(u)$ as the argument. In Cycle Triangle Enumeration we impose the condition that $u < v$ and $u < w$ to avoid double counting.

Algorithm 9 CYCLE TRIANGLE ENUMERATION

Input: A directed graph $G = (V, E)$, its transpose $G^T = (V, E^T)$
Output: The list and number of cycle triangles in G.

```
1: S ← ∅, k ← 0, c ← 0
2: for all nodes u ∈ V do
3:     for all nodes v ∈ N⁺(u) do
4:         if u < v then
5:             (S, k) ← INTERSECTION(N⁻(u), N⁺(v), d⁻(u), d⁺(v))
6:             for all nodes w ∈ S do
7:                 if u < w then
8:                     ENUM ((u, v, w)_C)                    ▷ For Listing
9:                     c ← c + 1
10: return c
```

5.1.2 Analysis

We give the proof of correctness and analyse the running times of the algorithms.

Theorem 8. *The Trust Triangle Enumeration algorithm (Algorithm 8) correctly enumerates the trust triangles in a simple directed graph.*

Proof. The algorithm iterates over each node once, and for each node, its outgoing neighbours are checked once. Thus, it goes through every edge in the graph, and each edge is checked once. Any common outgoing neighbours of the two end nodes of an

edge would be found by the intersection computation, once. Thus all trust triangles containing these two nodes in the first edge would be found once. Without any ordering condition on the nodes, all trust triangles found would be enumerated. □

Theorem 9. *The Cycle Triangle Enumeration algorithm (Algorithm 9) correctly enumerates the cycle triangles in a simple directed graph.*

Proof. A cycle triangle $(u, v, w)_C$ has three nodes u, v, w, connected by edges $u \to v$ (first edge), $v \to w$ (second edge), and $w \to u$ (third edge). When we reverse the third edge of a cycle triangle we get a trust triangle, and vice versa. Consequently, we can search for cycle triangles by using the same Intersection algorithm as in Trust Triangle Counting provided that we use an in-going edge instead of an out-going edge for the third edge. The in-going edges can be read from the transpose graph. Since all cycle triangles will appear like a trust triangle by this procedure, and by Theorem 8 we have shown that all trust triangles would be found, and all cycle triangles would be found.

Now, notice that $(u, v, w)_C$, $(v, w, u)_C$ and $(w, u, v)_C$ are identical cycle triangles. We can avoid double counting by anchoring the first node. We do this by imposing the condition that the first node must be smaller than the other two in the triple, $u < v$ and $u < w$. Note that we do not impose any condition on the order of v and w, because $(u, v, w)_C$ and $(u, w, v)_C$ are distinct cycles and hence should be counted separately. Thus, Algorithm 9 would find all cycle triangles and enumerate each of them once. □

Theorem 10. *The running time of the Trust Triangle Enumeration on graph $G(V, E)$ is bounded by $O(|V|(d^+_{\max}(G))^2)$.*

Proof. Recall that the algorithm iterates over the edges in the graph by going through the nodes and for each node its adjacent nodes. For each edge $u \to v$ the intersection

computation runs in time $d^+(u)+d^+(v)$. Thus, the running time for the trust triangle counting is $\sum_{u \to v}(d^+(u) + d^+(v)) \leq \sum_u d^+(u)(d^+(u) + \max[d^+(v \in N^+(u))])$. The worst case is $2|V|(d^+_{\max}(G))^2$. $\square$

Theorem 11. *The running time of the Cycle Triangle Counting on graph $G(V, E)$ is bounded by $O(|V|d^+_{\max}(G)(\max[d^+_{\max}(G), d^-_{\max}(G)]))$.*

Proof. The proof is similar to the one for the trust triangle above. The difference is that for each edge $u \to v$ here, the running time for the intersection computation is $d^-(u) + d^+(v)$. Therefore, the cycle triangle counting running time is $\sum_{u \to v}(d^-(u) + d^+(v)) \leq \sum_u d^+(u)(d^-(u) + \max[d^+(v \in N^+(u))])$. The worst case is when all the degrees are equal to the maximum, yielding running time
$O(|V|d^+_{\max}(G)(\max[d^+_{\max}(G), d^-_{\max}(G)]))$. $\square$

Now, suppose that we have a graph where $d^+_{\max}$ is much larger than $d^-_{\max}$, then we can run the Trust Triangle Enumeration on the transpose graph instead, and get the answer in a shorter time. On the other hand, if $d^+_{\max}$ is much smaller than $d^-_{\max}$, we should run the Trust Triangle Enumeration on the graph itself. For the Cycle Triangle Enumeration, on the other hand, reversing the role between the graph and the transpose graph does not guarantee a clear advantage in terms of the running time.

5.1.3 Experiment

For experimenting on directed triangle enumeration, we use the same settings as we used for the undirected triangle enumeration, in the previous chapter. For the datasets we also use the same graphs, but without the symmetrization, so the graphs are still directed.

There were 11 directed graphs used in this experiment, ranging from `words` with 10,617 nodes and 72,172 directed edges, to `clueweb` with 978,408,098 nodes and 42,574,107,469 directed edges. The `clueweb` is a highly asymmetric graph. Its maximum out-degree is only 7,447, but its maximum in-degree is 75,611,690. We were able to run both the cyclic and trust triangle enumeration on `clueweb`, within less than a day each. We were even able to run the enumeration using just a third-generation i7 machine, also within one day's time. However, when we ran the trust enumeration on the transpose of `clueweb`, the running time became so long that we had to abort the run before finished. These results validate our observation that the running time of trust triangle enumeration depends on the maximum out-degree of the input graph.

We found 1,036,190,284,927 cycle triangles, and 5,508,820,034,813 trust triangles in `clueweb`, more than a trillion. In comparison, when we enumerate the undirected triangles in the underlying graph of `clueweb`, we found 1,995,295,290,765 triangles. So the count of undirected triangles is between the count of the cycle triangles and the count of the trust triangles in this case. This makes sense because each undirected triangle is correlated to between zero and two cycle triangles, and between zero and six trust triangles.

5.2 Triads

In a directed graph each edge has a direction. Edges connected to a node u can be classified into two types: edges *to* u and edges *from* u. Accordingly, we have outgoing neighbours of u, $N^+(u)$, and ingoing neighbours of u, $N^-(u)$, for the neighbouring nodes of u. The out-degree of u is $d^+(u) = |N^+(u)|$, and the in-degree of u is $d^-(u) = |N^-(u)|$. Note that $N^+(u)$ and $N^-(u)$ may overlap, because, for a pair of nodes u and v, we may have both $u \to v$ and $u \leftarrow v$ edges. We call the connection

between two nodes a link. There are three types of links between nodes u and v: from u to v, from v to u, and bidirectional. We encode the links by using two binary digits as shown in Table 5.1.

Link	Value	Binary
○ ○	0	00
○⟶○	1	01
○⟵○	2	10
○⟷○	3	11

Table 5.1: Link encoding using two binary digits. We assume that the first node is on the left and the second one on the right.

A triad is a subgraph of three nodes in a directed graph [23, 4]. When each pair of the nodes is connected we have a closely connected triad. Here, since we restrict our study to only closely connected triads, we will simply call them triads. Other authors use the term triangles [66, 73], but we do not want to confuse them with the undirected triangles, or the directed triangles as in the previous section. There are seven types of triads, as shown in Figure 2.8. Enumerating triads means listing the edges (or links) as well as the nodes inside every triad. Thus, triad enumeration is more complex than triangle enumeration. Nonetheless, we have shown that it is possible to devise an efficient algorithm that, when combined with a compression framework such as WebGraph [11], is able to enumerate triads on a graph with a billion nodes and billions of edges using a single commodity machine [64].

The Batagelj and Mrvar triad census algorithm [4] assigns a code to each pair of nodes to represent the directed edges between them. For each triple of nodes, it then uses a table to find the triad types based on the combined codes. Although this algorithm can do triad enumeration in subquadratic time, it is not fast enough for very large graphs with millions of nodes and edges.

Chin et al. [20] developed a compact data structure that makes it easier to parallelize the computation. They combined the adjacency list to contain both outgoing and incoming edges. The edge information or the link is coded using 2-bits: 01 (forward), 10 (backward), and 11 (both), embedded in the neighbour node labels inside the list. Suppose the nodes were labeled by using 32-bit integers. The bits are shifted to the left by two, and the two lowest bits are then used for the edge direction. Thus, only 30 bits can actually be used to label the nodes.

The drawback of the compact data structure solution is that it leads to reduced scalability. In Java, the 32-bit integer data type can be used to label up to 2^{31} nodes (because we can have only signed integers), but with 2 bits used for edge information, it can label only up to 2^{29} (or about 1/2 billion) nodes. This becomes problematic when we want to analyze a graph such as `clueweb12` which has almost a billion nodes and about forty two billion edges. Theoretically, we can switch to 64-bit (or 8-byte) long data type and be able to do `clueweb12`. However, we still need to overcome the memory limitation problem. With `clueweb12`, forty two billion edges translate to more than 300 GB RAM if we use 8 bytes for each, which is way beyond the typical amount of RAM in current commodity machines.

We develop a new algorithm that computes the type of connections between each pair of nodes on the fly, using both the graph and its transpose as input. Using this algorithm, and partial loading method of WebGraph [11], we were able to process `clueweb12` on a single machine with a memory budget of 32 GB RAM.

5.2.1 Algorithm

Our serial algorithm for triad enumeration is described in Algorithm 10. The algorithm requires both a directed graph and its transpose graph as input, similar to the case for cycle triangle enumeration. Recall that the transpose of a directed

graph $G = (V, E)$ is another directed graph $G^T = (V, E^T)$, where E^T is the same set of edges as E but with each edge is reversed. The key idea here is that the ingoing neighbours of node u in G are the outgoing neighbours of u in G^T, i.e., $N^-(u) \equiv N^-_G(u) = N^+_{G^T}(u)$. Therefore, we can consider only the outgoing adjacency lists from each G and G^T in the computation. That is, we find $N^+(u)$ from G and $N^-(u)$ from G^T. To find the triads, this algorithm employs four pointers, one on each of $N^+(u)$, $N^-(u)$, $N^+(v)$, and $N^-(v)$. Therefore, we call it *Four Pointers Triad Enumeration* (FPTE) algorithm.

Algorithm 10 FPTE

Input: A directed graph $G = (V, E)$ and its transpose G^T
Output: The list and number of each type of triads in G, Δ_i

```
 1: Δ1 ← 0, ..., Δ7 ← 0
 2: for all u ∈ V do                                    ▷ Can be parallelized
 3:     while there is next do
 4:         Find next neighbour in N+(u) and/or N−(u): v.
 5:         Code the link uv as e1: either 01, 10 or 11
 6:         while there is next do
 7:             Find next common neighbour of u and v: w.
 8:             Code the links vw as e2, and wu as e3.
 9:             Look up triad type i using e1, e2, e3.
10:             ENUM(u, v, w, e1, e2, e3)
11:             Δi ← Δi + 1
```

The algorithm iterates over the first node u. This iteration can easily be parallelized. For each u, it checks both $N^+(u)$ and $N^-(u)$ to find the neighbours of u and their respective links. For each neighbour of u, v, it finds their common neighbours using four pointers (Line 7). For each common neighbour, w, it looks up the triad type based on the links among the three nodes (u, v, w). The encodings are listed in Table 5.2. In line 10, ENUM() is a space holder for an enumeration or listing function.

The 4-pointers algorithm is an expansion of the 2-pointers algorithm commonly used for set intersections in triangle enumeration (See Algorithm 2). The flow of the

Triad	Binary Code
Type 1	010101, 101010
Type 2	010110, 011001, 100101, 101001, 100110, 011010
Type 3	010111, 011101, 110101, 101011, 101110, 111010
Type 4	011011, 110110, 101101
Type 5	100111, 111001, 011110
Type 6	011111, 110111, 111101, 101111, 111011, 111110
Type 7	111111

Table 5.2: Triad types and binary encoding.

4-pointers algorithm is illustrated in Figure 5.2. At the start, each pointer is set to the lowest member of each corresponding set. It checks on the lowest pointer to see if there is another pointer at the same level, and if the member is a common neighbour of u and v. If so, it then computes the link type and enumerates the triad. It then proceeds by moving the lowest pointer(s) to the next neighbour. Thus, it searches for intersection between $(N^+(u), N^-(u))$ and $(N^+(v), N^-(v))$.

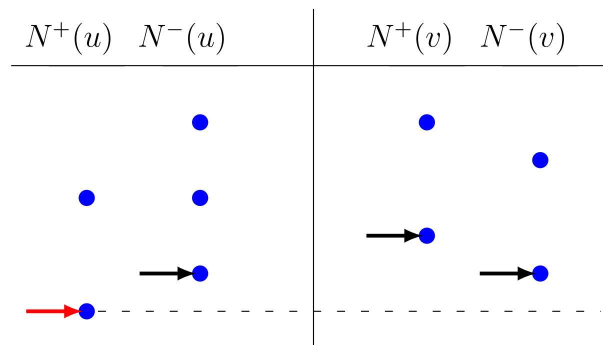


Figure 5.2: Four pointers

As with triangle enumeration, preprocessing the input graph before the enumeration is crucial in shortening the runtime. In this case, the preprocessing needs to be done simultaneously on the graph and its transpose, so that any relabelling would be consistent between the two. To get the greatest benefit, the sorting is based on whichever has the bigger maximum degree. The pseudocode for this preprocessing is

given in Algorithm 11.

Algorithm 11 DiGraph-Prep

Input: An directed graph $G(V, E)$ and its transpose $G^T(V, E^T)$
1: Check the maximum out-degrees of G and G^T.
2: Sort V based on the out-degrees of either G or G^T, whichever has the higher maximum out-degree, in ascending order.
3: Relabel the vertices according to their new order.
4: Build adjacency list of the sorted and relabeled vertices.
5: Cut out the smaller out-neighbours from each neighbour list.

5.2.2 Experiment

We ran our experiments on a machine with dual Intel Xeon E5620 CPUs and 64 GB of RAM. However, to make a better comparison with other papers, we allowed only 32 GB of RAM to be used by the Java virtual machine. The Xeon CPU has a clock speed of 2.40 GHz and 8 threads (16 threads total for the dual).

We select five datasets to be used for this experiment are `cnr-2000`, `ljournal`, `arabic-2005`, `uk-2005`, and `clueweb`. See Appendix A, for the descriptions. The smallest dataset, `cnr-2000`, has 3.2M edges, while the largest dataset, `clueweb`, has more than 42B edges. The graphs and their transpose graphs are preprocessed according to Algorithm 11. The degree statistics of the graphs before and after the preprocessing are listed in Table 5.3. Here, d^- refers to the out-degree in the transpose graph. The degrees after are denoted by d^{eff}. Notice that the preprocessing managed to reduce the effective maximum degree by four orders of magnitude, in the case of `clueweb`. Recall that the preprocessing first chooses the larger between $d^+_{\max}$ and $d^-_{\max}$, and proceeds based on the one chosen. For these five graphs, we see that the transpose graphs are the ones with larger maximum degrees. This is why the reduction in d^- is bigger compared to that for d^+.

Name	$d^+_{\max}$	$d^-_{\max}$	$d^{+,\mathrm{eff}}_{\max}$	$d^{-,\mathrm{eff}}_{\max}$
`cnr-2000`	2,716	18,235	1,336	81
`ljournal`	2,469	19,409	1,257	397
`arabic`	9,905	575,618	6,646	3,126
`uk-2005`	5,213	1,776,852	5,213	584
`clueweb`	7,447	75,611,690	5,873	4,242

Table 5.3: Maximum degrees before and after the preprocessing. The degrees after are listed as d^{eff}.

The enumeration produced the counts for each triad type. The results are shown in Table 5.4. We can see that, for these graphs, Δ_1 is typically the smallest, followed by Δ_3. Recall that type-1 contains one cycle triangle, and type-3 contains one cycle and one trust triangle. Interestingly, Δ_7 is much bigger than Δ_1, for all of the graphs that we consider here. The biggest number in this table is Δ_5 for the `clueweb` which is greater than 790 billion.

Name	Δ_1	Δ_2	Δ_3	Δ_4
`cnr-2000`	10,342	9,899,367	85,969	2,433,041
`ljournal`	530,051	86,777,707	10,421,919	69,748,792
`arabic`	2,668,704	6,906,765,421	30,427,662	1,571,745,235
`uk-2005`	5,335,890	5,198,533,331	48,779,535	1,773,901,843
`clueweb`	281,444,867	517,684,665,693	2,261,300,705	153,674,084,413

Name	Δ_5	Δ_6	Δ_7
`cnr-2000`	6,736,504	419,472	1,392,934
`ljournal`	44,608,271	80,177,727	118,890,977
`arabic`	11,765,868,185	384,594,679	16,233,290,956
`uk-2005`	9,499,139,863	411,396,906	4,842,278,688
`clueweb`	790,291,640,762	28,556,769,295	502,545,385,030

Table 5.4: The counts of triads of each type on the selected graphs.

In Table 5.5 we list the running times of the triad enumeration on the selected graphs, in seconds. The preprocessing time is shown as T_{prep}. The running times on

graphs without preprocessing are not shown here. However, note that this preprocessing is important in keeping the running time low. The running time on `clueweb` is more than a day, but less than one and a half days.

Name	T_{prep}	T_{FPTE}
`cnr-2000`	2.75	3.0
`ljournal`	74	81
`arabic`	200	2,961
`uk2005`	311	796
`clueweb`	12,870	115,960

Table 5.5: The running time (in seconds) of triad enumeration using FPTE algorithm, and the preprocessing time.

We also compared the performance of our FPTE algorithm to an algorithm by Parimalarangan et al. [46] which uses the compact data structure described above, the AI algorithm. We found that FPTE has a comparable running time compared to AI. Moreover, FPTE was able to process `clueweb`, while AI cannot. This is because FPTE does not require compact data structure, hence can use all 32 bits of integer for the labels.

5.3 Directed Graphlets

As with triangle and (triangle) triads, we can look for directed wedges, which are directed graphs whose underlying graph is a wedge. As can be seen in Figure 2.7, there are six of them: 4-021D, 5-021U, 6-021C, 7-111D, 8-111U, and 11-201. A way of looking into this is as follows: recall that there are three types of links between two nodes, so with two links in a wedge there are $3^2 = 9$ possible combinations. However, some of them are isomorphic. A wedge can be viewed as a 2-path, ad it has a mirror symmetry. Three of the configurations are self-image under this symmetry. So we are left with $(9 - 3)/2 + 3 = 6$ non-isomorphic directed wedges.

We can proceed in a similar fashion to consider directed four-node graphlets. For each type of graphlet, we need to find non-isomorphic configurations of possible links. As an example, let us consider the directed three path (g_3). There are three links in this type of graphlet, so $3^3 = 27$ possible combinations. The only symmetry is the mirror symmetry. One of the combinations is where all three links are bidirectional, which is its own mirror image. Two other combinations, where the center link is bidirectional and the other two links are in opposite directions, are also self-imaged. The rest are paired by mirror symmetry. Thus, we have $(27 - 3)/2 + 3 = 15$ non-isomorphic configurations (or types), which is quite plenty. We can continue with the other graphlet types as well, however, the symmetrical properties are more complex. More formally, these computations can be done by employing Burnside's Lemma which computes the number of orbits (or distinct objects) based on the number of invariant objects under some symmetry operations. The computation details can be found in Appendix C. We summarize the result for 4-node graphlets in Table 5.6. So, in total there are 199 types of 4-node directed graphlets.

Underlying graph	Number of links	Number of types
g_3	3	15
g_4	3	10
g_5	4	15
g_6	4	45
g_7	5	72
g_8	6	42

Table 5.6: Four-node directed graphlets.

The same idea that we use in triad enumeration can be used to enumerate directed 4-node graphlets as well. It is just that the number of types and the look-up table, similar to Table 5.2, will be very large.

Chapter 6

Distributed Enumeration

Since with enumeration, we have to touch each subgraph instance in the graph, the running time is bounded from below by the number of the subgraphs. Because this number grows rapidly with the size (and order) of the input graph, we are limited on the size of the graph that we can process on a given machine for a given time budget. Distributed computing is often proposed as a solution to push this limit further.

On a distributed platform there are compute nodes (or workers) that are connected by a computer network. Our problem is deciding on how to distribute the enumeration tasks among the workers, and how to collect and combine the outputs. In this setting, overhead costs and redundancy are inevitable. We need to minimize these to justify the economic cost of using a distributed platform [42].

6.1 Graph Partition and Subproblems

To distribute a subgraph enumeration, the input graph needs to be partitioned. We use a partition scheme by coloring as in [48]. Below are the definitions used in this scheme.

Coloring. Coloring refers to a technique of applying a modulo function with respect to a chosen number of colors, ρ, to each edge $uv \in E$. An edge uv has "color" (i, j) where $i = u\%\rho$, $j = v\%\rho$ and % is the mod operator. Edges with the same color can be grouped together to form an edge-induced sub-graph.

Edge-orientation. Edge-orientation is a technique widely used in sub-graph enumeration because following an orientation helps eliminate duplicate outputs and speeds up the enumeration. It assigns orientation to each edge in an undirected graph by following a prescribed rule. A common rule is as follows. First, define a function η which determines a total ordering of the nodes in V. An edge $uv = vu$ is orientated by η, such that if $\eta(u) < \eta(v)$, we list only uv but not vu. This oriented edge is then denoted by $(\overrightarrow{u,v})$. As is common in practice, we use the degrees of the nodes to define the total ordering η, i.e., if $d(u) < d(v)$ then $\eta(u) < \eta(v)$. If the degrees are equal we just use the node labels to determine the order.

Directed acyclic graph. Using edge-orientation, the undirected input graph is transformed into a directed acyclic graph (DAG), i.e., a directed graph without any directed cycle, denoted by $\overrightarrow{G}(V, \overrightarrow{E})$. The out-neighbouring vertices of vertex u are denoted by $N^+(u)$. The out-degree of vertex u is denoted by $d^+(u)$.

Edge set. An edge set E_{ij} is an edge-induced sub-graph of the undirected input graph formed by all edges with color (i, j). Note that orienting the edges does not change the edge set.

Symmetrization. Symmetrization is the process of making all edges in a directed graph bi-directional. As we will see below, symmetrization is needed for our distributed solution.

Directed edge set. A directed edge set E^*_{ij} is an edge-induced sub-graph of the edge-oriented DAG, where each edge $(\overrightarrow{u,v}) \in E^*_{ij}$ points from color i to color j. Directed edge set E^*_{ij} is a subset of edge set E_{ij}. For $i \neq j$, $E^*_{ij} \cup E^*_{ji} = E_{ij}$. For $i = j$,

$E^*_{ii} = E_{ii}$.

Sub-problems. A sub-problem refers to the union of edge-sets of particular colors, or more precisely the problem of finding the graphlets in that union-set. For a k-order graphlet enumeration, we denote sub-problems by $S_{\{c_0,c_1,...,c_l\}}$ where $|\{c_0, c_1, ..., c_l\}| \in \{1, 2, ...k\}$ and $c_l \in \{1, 2, ...\rho\}$. For example, for $\rho = 3$ and $k = 3$ (e.g., triangle), the sub-problems are: S_0, S_1, S_2, S_{01}, S_{02}, S_{12} and S_{012}, where, $S_i = E_{ii}$, $S_{ij} = E_{ii} \cup E_{ij} \cup E_{jj}$, and $S_{ijk} = E_{ij} \cup E_{ik} \cup E_{jk}$. Note that $S_i \subset S_{ij}$, but $S_{ij} \not\subset S_{ijk}$.

6.2 Distributed Four-node Graphlet Enumeration

6.2.1 Previous Distributed Enumeration

Park et al. [48] proposed a distributed solution for triangle enumeration called PTE (Pre-partitioned Triangle Enumeration). PTE employs Compact-Forward algorithm as the local serial algorithm. On each distributed worker, PTE does $O(m^{1.5}/\rho^3)$ amount of work. Summing over all $O(\rho^3)$ sub-problems, PTE recovers $O(m^{1.5})$ amount of work overall, which is the same asymptotic behaviour as the Compact-Forward on a single machine. Note, however, that because of the distribution of the subproblems, there are inherently some redundant computations. Park et al. reduced the total number of operations by a factor of $2 - \frac{2}{\rho}$ by employing color directions to minimize this redundancy.

Park et al. generalised PTE to support non-induced sub-graph query of arbitrary order, called PSE (Pre-partitioned Subgraph Enumeration) [50]. PSE takes a query sub-graph $G_q(V_q, E_q)$ of order k as input where $k = |V_q|$, and enumerates all the sub-graphs matching G_q. PSE employs VF2 algorithm [21] as the local serial algorithm for query graph matching. We stress that VF2 can only take one non-induced subgraph query at a time, in contrast to S4GE, which enumerates all types of four-node induced

connected subgraphs simultaneously. PSE starts by defining $\sum_{l=1}^{k}\binom{\rho}{l}$ sub-problems. For example, with $\rho = 4$ and $k = 4$, PSE first defines the following sub-problems: S_0, S_1, S_2, S_3, S_{01}, S_{02}, S_{03}, S_{12}, S_{13}, S_{23}, S_{012}, S_{013}, S_{023}, S_{123} and S_{0123}. Park et al. observed that solving the sub-problems independently introduces duplicate emissions and that some sub-problems can be grouped together to reduce duplication. For example, since $S_0 \subset S_{01} \subset S_{012}$, enumerating the sub-graphs from S_{012} also enumerates all the sub-graphs from S_0 and S_{01}. PSE introduced a sub-problem group as the fundamental computing task of each distributed worker. For example, $\{S_{012}, S_0, S_1, S_2\}$, is a valid sub-problem group, where solving S_{012} is sufficient to solve for the entire group. Park et al. showed that PSE requires at most $\binom{\rho-1}{k-2}|E|$ amount of network read, for querying k-order sub-graphs from an input graph of size $|E|$.

Note that PTE can only enumerate triangles, while PSE can enumerate graphlets of any size (given the appropriate serial algorithm to do the task). However, PSE is not fine-tuned for enumerating four-node graphlets using the S4GE as the serial algorithm.

6.2.2 Generalized Color-Direction

PSE, while correctly enumerating all sub-graphs that match the query, discovers certain sub-graphs more than once. Consider the following: if there is a 4-node sub-graph (u, v, w, z) whose color is $(0, 0, 1, 1)$, it can be discovered from the group $\{S_{012}, S_0, S_1, S_2\}$ as well as from the group $\{S_{013}, S_{01}, S_{03}\}$, since the first group S_{012} reads edge sets $E_{00} \cup E_{11} \cup E_{22} \cup E_{01} \cup E_{02} \cup E_{12}$, and the second group S_{013} reads edge sets $E_{00} \cup E_{11} \cup E_{33} \cup E_{01} \cup E_{03} \cup E_{13}$. Both contains $E_{00} \cup E_{01} \cup E_{11}$ where (u, v, w, z) of color $(0, 0, 1, 1)$ would be found.

We observe that: (1) Given a 4-node graphlet and ρ colors, there are in total ρ^4 possible color assignments of the four vertices (denoted by K_{ijkl}). (2) When a 4-node

graphlet (u, v, w, z) is emitted, it is imposed that the graphlet edges are oriented following the ordering of the vertices: $(\overrightarrow{u,v})$, $(\overrightarrow{u,w})$, $(\overrightarrow{u,z})$, $(\overrightarrow{v,w})$, $(\overrightarrow{v,z})$ and $(\overrightarrow{w,z})$. Combining both observations, each color assignment K_{ijkl} can be used to represent the set of all possible 4-node graphlets (u, v, w, z) of ordered colors (i, j, k, l), where the edges can only point from color i to colors $\{j, k, l\}$, from color j to colors $\{k, l\}$, and from color k to color l. Any 4-node graphlet can have only one unique ordered color assignment. Each ordered color assignment contains all the 4-node graphlets that meet the criteria, and there is no overlap among different ordered color assignments. Hence, enumerating from all ordered color assignments enumerates all the 4-node graphlets once and once only. The ordered color assignment can be viewed as a directed version of a sub-problem. Unlike a sub-problem S_{ijkl} that requires the union of edge sets E_{ij}'s, a color assignment K_{ijkl} requires the union of directed edge sets E^*_{ij}'s. A color assignment K_{ijkl} requires knowledge of $E^*_{ij} \cup E^*_{ik} \cup E^*_{il} \cup E^*_{jk} \cup E^*_{jl} \cup E^*_{kl}$. However, simply solving all the ordered color assignments on distributed workers will incur unnecessary network read. The color assignments, therefore, are grouped into sub-problems to reduce the network read, with a strategy introduced below. Sub-problems become the fundamental task assigned to each distributed worker.

Algorithm 12 D4GE

Input: An undirected graph $G(V, E)$; the number of colors ρ

1: Construct $\overrightarrow{G}(V, \overrightarrow{E})$ by applying edge-orientation to $G(V, E)$
2: Symmetrise $\overrightarrow{G}(V, \overrightarrow{E})$ into $G^{\text{sym}}(V, E^{\text{sym}})$
3: Partition E^{sym} into directed edge sets E^*_{ij} using ρ
4: Generate ordered color assignments and sub-problems $\{S_{\text{Cs}} \mapsto \{K_{ijkl}\}\}$
5: **for all** S_{Cs}, $\{K_{ijkl}\}$ **do** ▷ Distributed-for
6: $E_{\text{map}} \leftarrow$ ReadEdgeSets$_{\text{CD}}$ $(S_{\text{Cs}}, 4)$
7: **for all** $K_{ijkl} \in \{C_0, C_1, ...\}$ **do**
8: S4GE$_{\text{CD}}$ $(E_{\text{map}}, ijkl)$

We call our scheme, applied to four-node graphlets, as Distributed 4-node Graphlet Enumeration (D4GE). The pseudo-code of D4GE is given as Algorithm 12 and Al-

gorithm 13. We want to stress that while PTE employs the idea of color-direction to reduce the amount of work performed, we exploit both the linearity of the DAG and the color-direction, and are able to observe that the color-assignment problem is essentially a combination problem, and the unique relationship between any sub-graph to its color assignment guarantees the duplication-freeness of our algorithm. In addition, PTE explicitly lists all the ordered color-tuples in the algorithm, while we use combinations to generalize color-assignment. This works not only for $k = 4$, but also to any order k (with ρ^k color-assignments).

Algorithm 13 $\textsc{ReadEdgeSets}_{\text{CD}}$

Input: Sub-problem S_{Cs} with $Cs = \{c_0, c_1, ..., c_l\}$; order of query graph k
1: Initialize empty map $E_{\text{map}} \equiv \{(i, j) \mapsto E^*_{ij}\}$
2: **for all** $(i, j) \in Cs^2$ **do**
3:     **if** $i = j$ and $|\{c_0, c_1, ..., c_l\}| \neq k$ **then**
4:         $E_{\text{map}}[(i, i)] \leftarrow E^*_{ii}$
5:     **else**
6:         $E_{\text{map}}[(i, j)] \leftarrow E^*_{ij}$
7: **return** E_{map}

D4GE takes a DAG as input and symmetrizes it. Symmetrization is necessary to ensure the correctness of S4GE to enumerate all the wedge-based 4-node graphlets. Consider the graph in Figure 6.1 with DAG adjacency list: 1: {4}, 2: {4}, 3: {4}, 4: {5,6}, 5: {6}, 6: $\emptyset$. If we apply S4GE algorithm on this adjacency list, we will only find triangle (4, 5, 6) but will not discover tailed-triangle (1,4,5,6), (2,4,5,6) and (3,4,5,6), as vertices 1, 2, 3 are not in the adjacency list of vertex 4. With symmetrization, the adjacency list is now 1: {4}, 2: {4}, 3: {4}, 4: {1,2,3,5,6}, 5: {4,6}, 6: $\{4, 5\}$, and S4GE can now successfully enumerate the three tailed-triangles, as vertices 1, 2 and 3 are added into the neighbourhood of vertex 4.

Now, we introduce the grouping strategy to form sub-problems from ordered color assignments. Consider color assignments K_{0001} and K_{0002}. By definition K_{0001} re-

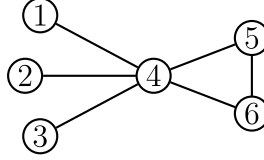


Figure 6.1: A graph illustrating the need for symmetrization.

quires knowledge of $E^*_{00} \cup E^*_{01}$ and K_{0002} requires knowledge of $E^*_{00} \cup E^*_{02}$. If these two color assignments are computed on two different workers, the partitioned edge-set E^*_{00} is then loaded twice. To address this, color assignments K_{pqrs} are grouped into sub-problems. We use S_{ijkl} to denote a sub-problem. The color assignments are grouped by the following rule: K_{pqrs} belongs to sub-problem S_{ijkl} if the *sorted* and *reduced* form of $\{p, q, r, s\}$ is $\{i, j, k, l\}$, where *sorted* means $\{p, q, r, s\}$ is sorted in ascending order, and *reduced* means removing the duplicated colors from the sequence $\{p, q, r, s\}$. For example, the sorted and reduced form of $\{2, 0, 1, 0\}$ is $\{0, 1, 2\}$. Therefore K_{2010} belongs to S_{012}.

It is not hard to see that sub-problem S_{ijkl} contains $4! = 24$ color assignments - precisely the number of permutations of the sequence $\{i, j, k, l\}$. To fully cover all the color assignments, D4GE generates $\binom{\rho}{2}$ number of S_{ij}, $\binom{\rho}{3}$ number of S_{ijk} and $\binom{\rho}{4}$ number of S_{ijkl}. Sub-problem S_{ijkl} contains all the ordered color assignments K_{pqrs} where $p, q, r, s \in \{i, j, k, l\}$; sub-problem S_{ijk} contains all the ordered color assignments K_{pqrs} where $p, q, r, s \in \{i, j, k\}$; sub-problem S_{ij} contains all the ordered color assignments K_{pqrs} where $p, q, r, s \in \{i, j\}$. In the special case of ordered color assignments K_{iiii} where all four colors are the same, we omitted S_i and instead attach K_{iiii} to sub-problem S_{ij} where $i + 1 = j\%\rho$. Each sub-problem is computed independently on a distributed worker.

For all ordered color assignments under sub-problem S_{ijkl}, there are only two possible relative orders of two arbitrary colors p and q: p precedes q or the reverse, meaning for any p and q from $\{i, j, k, l\}$, $E^*_{pq} \cup E^*_{qp} = E_{pq}$ is needed. Hence overall, to

fully enumerate S_{ijkl}, $E_{ij} \cup E_{ik} \cup E_{il} \cup E_{jk} \cup E_{jl} \cup E_{kl}$ needs to be read. Sub-problem S_{ijk} can be treated as $S_{iijk} \cup S_{ijjk} \cup S_{ijkk}$ to reflect that it requires $E_{ii} \cup E_{ij} \cup E_{ik} \cup E_{jj} \cup E_{jk} \cup E_{kk}$, and sub-problem S_{ij} can be treated as $S_{iiij} \cup S_{iijj} \cup S_{ijjj}$ to reflect that it requires $E_{ii} \cup E_{ij} \cup E_{jj}$. Each of the sub-problems and the associated ordered color assignments are sent to a distributed worker; the worker iterates over all the ordered color assignments. For each colored assignment, the worker reads the directed edge sets from distributed storage and enumerates the 4-node graphlets by applying the modified S4GE algorithm.

Algorithm 14 S4GE$_{\mathrm{CD}}$

Input: A mapping from the colors of directed edge set to the edge set $E_{\mathrm{map}} \equiv \{(i,j) \mapsto E^*_{ij}\}$; ordered color assignment $ijkl$

```
 1: E*_ij ≡ E_map[ij], E*_ik ≡ E_map[ik], E*_jk ≡ E_map[jk]
 2: for all (u,v) ∈ E*_ij do
 3:     if η(u) < η(v) then
 4:         for u' ∈ N(u) ⊂ E*_ik and v' ∈ N(v) ⊂ E*_jk do
 5:             if (u' > u) ∧ (v' > u) then
 6:                 if u' = v' > v then
 7:                     DExploreTriangle (u, v, u', E_map, ijkl)
 8:                 if (u' < v') ∧ (u' > v) then
 9:                     DExploreWedge-1 (v, u, u', E_map, ijkl)
10:                 if u' > v' then
11:                     DExploreWedge-2 (u, v, v', E_map, ijkl)
```

6.2.3 S4GE with Color Direction

S4GE is modified accordingly so that it is able to enumerate all 4-node graphlets for an ordered color assignment $ijkl$, and we call this modified version S4GE$_{\mathrm{CD}}$. The pseudocode for S4GE$_{\mathrm{CD}}$ is given in Algorithm 14, and the details of the explore-functions are given in Algorithms 15, 16, and 17 respectively.

Instead of enumerating on a complete graph, S4GE$_{\mathrm{CD}}$ now enumerates on a sub-graph denoted by the color assignment $ijkl$. The sub-graph consists of a mapping

Algorithm 15 DExploreTriangle

Input: Given triangle (v, u, w); $E_{\text{map}} \equiv \{(i,j) \mapsto E^*_{ij}\}$; ordered color assignment $ijkl$.

1: $N^{>u}(u) \equiv \{z \mid z \in N(u)|_{E_{\text{map}}[il]}\ ,\ \eta(z) > \eta(u)\}$
2: $N^{>u}(v) \equiv \{z \mid z \in N(v)|_{E_{\text{map}}[jl]}\ ,\ \eta(z) > \eta(u)\}$
3: $N^{>u}(w) \equiv \{z \mid z \in N(w)|_{E_{\text{map}}[kl]}\ ,\ \eta(z) > \eta(u)\}$
4: **for all** $z \in N^{>u} \cap N^{>u}(v) \cap N^{>u}(w)$ with $z > w$ **do**
5: Enumerate4Clique (u, v, w, z)
6: **for all** z in two sets and $z >$ opposite node **do**
7: EnumerateDiamond (u, v, w, z)
8: **for all** z in one set only **do**
9: EnumerateTailedTriangle (u, v, w, z)

between the ordered color 2-tuples (i, j) and the corresponding directed edge-sets E^*_{ij}. For an ordered color assignment, there are $\binom{4}{2} = 6$ such 2-tuples: (i, j), (i, k), (i, l), (j, k), (j, l) and (k, l). (i, j), (i, k), (j, k) and the corresponding edge-sets are used to discover the wedge or triangle, and (i, l), (j, l), (k, l) and the corresponding edge-sets are used to discover the graphlet after the base wedge or triangle have been discovered. S4GE$_{\text{CD}}$ inherits the correctness from S4GE since the actual intersection logic is untouched, whereas S4GE$_{\text{CD}}$ solely focuses on a particular edge-induced sub-set of the input graph, with all the edges pointing from color i to j, k, l, from j to k, l and from k to l.

The modification of S4GE shows the expandability of the D4GE partitioning scheme. Since the intersection is not modified, the partitioning scheme can be applied to different edge-based enumeration algorithms to suit different needs. All it requires is to modify the input to accommodate a directed sub-set of the input graph.

6.2.4 Compact-Forward for 4-clique listing

Since PSE with VF2 only supports one query graph per run, for the purpose of comparison we build an algorithm to enumerate 4-cliques. Furthermore, we fit it to

Algorithm 16 DExploreWedge-1

Input: Given wedge (v, u, w); $E_{\text{map}} \equiv \{(i, j) \mapsto E^*_{ij}\}$; ordered color assignment $ijkl$.

1: $N^{>u}(u) \equiv \{z \mid z \in N(u)|_{E_{\text{map}}[il]}\,,\ \eta(z) > \eta(u)\}$
2: $N^{>u}(v) \equiv \{z \mid z \in N(v)|_{E_{\text{map}}[jl]}\,,\ \eta(z) > \eta(u)\}$
3: $N^{>u}(w) \equiv \{z \mid z \in N(w)|_{E_{\text{map}}[kl]}\,,\ \eta(z) > \eta(u)\}$
4: **for all** $z \in N^{>u}(v) \cap N^{>u}(w)$ with $z \notin N^{>u}(u)$ **do**
5:     EnumerateRectangle (u, v, z, w)
6: **for all** $z \in N^{>u}(u)$ only **do**
7:     **if** $z > w$ **then**
8:         Enumerate3Star (u, v, w, z)
9: **for all** $z \in N^{>u}(v)$ only **do**
10:     Enumerate3Path (w, u, v, z)
11: **for all** $z \in N^{>u}(w)$ only **do**
12:     Enumerate3Path (v, u, w, z)

Algorithm 17 DExploreWedge-2

Input: Given wedge (v, u, w); $E_{\text{map}} \equiv \{(i, j) \mapsto E^*_{ij}\}$; ordered color assignment $ijkl$.

1: $E^*_{jl} \equiv E_{\text{map}}[jl]$, $E^*_{kl} \equiv E_{\text{map}}[kl]$
2: $N^{>u}(v) \equiv \{z \mid z \in N(v)|_{E^*_{jl}}\,,\ \eta(z) > \eta(u)\}$
3: $N^{>u}(w) \equiv \{z \mid z \in N(w)|_{E^*_{kl}}\,,\ \eta(z) > \eta(u)\}$
4: **for all** $z \in N^{>u}(v)$ only **do**
5:     **if** $z > w$ **then**
6:         Enumerate3Star (v, u, w, z)
7: **for all** $z \in N^{>u}(w)$ only **do**
8:     **if** $z \neq v$ **then**
9:         Enumerate3Path (u, v, w, z)

be used with the color direction scheme of D4GE. This algorithm, called CF4_{CD}, is given as Algorithm 18.

Note that CF4 extends the idea of Compact-Forward algorithm from triangles to four-cliques, hence the name CF4. The correctness of CF4_{CD} is intuitive. CF4_{CD} does $O(m^2)$ work for a given graph.

Algorithm 18 CF4_{CD}

Input: An edge-oriented edge set E^*_{ij}, E^*_{ik}, E^*_{jk}, E^*_{il}, E^*_{jl}, E^*_{kl}
1: **for all** $(\overrightarrow{u,v}) \in E^*_{ij}$ **do**
2: **for all** $w \in \{ N^+(u)|_{E^*_{ik}} \cap N^+(v)|_{E^*_{jk}} \}$ **do**
3: **for all** $z \in \{ N^+(u)|_{E^*_{il}} \cap N^+(v)|_{E^*_{jl}} \cap N^+(w)|_{E^*_{kl}} \}$ **do**
4: Enumerate (u, v, w, z)

6.2.5 Analysis

In this analysis, first we show that D4GE with S4GE_{CD} correctly enumerates all the 4-node graphlets. D4GE works by generating all possible colored assignments of all 4-node graphlets. Any 4-node graphlet must be found from one and only one of the colored assignments. D4GE then applies S4GE_{CD} algorithm on each individual color assignment. Since S4GE correctly enumerates all 4-node graphlets for any given graph, D4GE/S4GE_{CD} correctly enumerates all 4-node graphlets for all color assignments of $G^{\text{sym}}(V, E^{\text{sym}})$.

Second, we show that D4GE with S4GE_{CD} is expected to require no more than $2\,m^{\text{sym}}$ amount of network read in addition to PSE, where m^{sym} is the number of edges in E^{sym}. For D4GE with S4GE_{CD}, the edge set E_{ii} is requested $(\rho - 1)$ times (by sub-problems S_{ik}), and $\binom{\rho-1}{2}$ times (by sub-problems S_{ikl}), hence the amount of network read is $\sum_{i=0}^{\rho-1} |E_{ii}| \binom{\rho}{2}$. The E_{ij} with $i \neq j$ is requested once by sub-problems S_{ij}, $\binom{\rho-2}{1}$ times by sub-problems S_{ijk}, and $\binom{\rho-2}{2}$ times by sub-problems S_{ijkl}. Thus, the amount of network read is $\sum_{i=0}^{\rho-1} \sum_{j=i+1}^{\rho-1} |E_{ij}| \left[1 + \binom{\rho-1}{2}\right]$. Combining both cases:

$$
\begin{aligned}
&\sum_{i=0}^{\rho-1}|E_{ii}|\binom{\rho}{2} + \sum_{i=0}^{\rho-1}\sum_{j=i+1}^{\rho-1}|E_{ij}|\left[1+\binom{\rho-1}{2}\right] \\
&= \binom{\rho}{2}\left[\sum_{i=0}^{\rho-1}|E_{ii}| + \sum_{i=0}^{\rho-1}\sum_{j=i+1}^{\rho-1}|E_{ij}|\right] - (\rho-2)\sum_{i=0}^{\rho-1}\sum_{j=i+1}^{\rho-1}|E_{ij}| \\
&\equiv \binom{\rho}{2} m^{\mathrm{sym}} - (\rho-2) m_{\neq}^{\mathrm{sym}}
\end{aligned}
\tag{6.1}
$$

If we assume the edges are distributed evenly, the expected size of $m_{\neq}^{\mathrm{sym}}$ is $\frac{\rho^2-\rho}{\rho^2}{=}1-\frac{1}{\rho}$ of m^{sym}. Recall that, for $k = 4$, PSE requires $\binom{\rho-1}{2} m^{\mathrm{sym}}$ amount of network read. Thus the difference to PSE is

$$
\left[\binom{\rho}{2} - \rho + 3 - \frac{2}{\rho}\right] m^{\mathrm{sym}} - \binom{\rho-1}{2} m^{\mathrm{sym}} = \left(2 - \frac{2}{\rho}\right) m^{\mathrm{sym}} \tag{6.2}
$$

which is less than $2\, m^{\mathrm{sym}}$.

Last, we show D4GE reduces the amount of work compared to PSE. For this comparison, we are using S4GE$_{\mathrm{CD}}$ as the localized algorithm. Since S4GE$_{\mathrm{CD}}$ enumerates all 4-node graphlets by discovering the base triangle and wedges first, we separate the work calculation into two parts: one being the amount of work to discover all the base triangle and wedges, the other to discover the fourth vertex.

Let us consider the first part. We can see that $\sum_{(u,v)\in E^{\mathrm{sym}}}(d^{\mathrm{sym}}(u) + d^{\mathrm{sym}}(v))$ is the amount of work to intersect all pairs of edges for a symmetrised graph. This sum is bounded by and can be estimated by $2\, m^{\mathrm{sym}} d_{\max}^{\mathrm{sym}}$, where $d_{\max}^{\mathrm{sym}}$ is the maximum degree of the symmetrised graph. Following the analysis to derive expression 6.1, D4GE does

$$
2\binom{\rho}{2} m_{=}^{\mathrm{sym}} d_{\max}^{\mathrm{sym}} + 2\left[1+\binom{\rho-1}{2}\right] m_{\neq}^{\mathrm{sym}} d_{\max}^{\mathrm{sym}} \tag{6.3}
$$

amount of work for discovering all the base triangles and wedges. For PSE, each E^*_{ii} is read $\binom{\rho-1}{2}$ times; each E^*_{ij} with $i \neq j$ is read $\binom{\rho-2}{1} + \binom{\rho-2}{2} = \binom{\rho-1}{2}$ times. So the total amount of work done by PSE to list all base triangles and wedges is

$$2\binom{\rho-1}{2} m^{\text{sym}}_{=} d^{\text{sym}}_{\max} + 2\binom{\rho-1}{2} m^{\text{sym}}_{\neq} d^{\text{sym}}_{\max}. \tag{6.4}$$

Subtracting Expression 6.3 by Expression 6.4 yields

$$\begin{aligned} &2\binom{\rho-1}{1} m^{\text{sym}}_{=} d^{\text{sym}}_{\max} + 2m^{\text{sym}}_{\neq} d^{\text{sym}}_{\max} \\ &= 2(\rho-2)\, m^{\text{sym}}_{=} d^{\text{sym}}_{\max} + 2m^{\text{sym}} d^{\text{sym}}_{\max} \end{aligned} \tag{6.5}$$

recall that if we assume edges are distributed evenly, the expected value of $m^{\text{sym}}_{=}$ is $\frac{1}{\rho} m^{\text{sym}}$. Thus expression 6.5 can be simplified to

$$\begin{aligned} &2(\rho-2)\, m^{\text{sym}}_{=} d^{\text{sym}}_{\max} + 2m^{\text{sym}} d^{\text{sym}}_{\max} \\ &= 2\frac{\rho-2}{\rho} m^{\text{sym}} d^{\text{sym}}_{\max} + 2m^{\text{sym}} d^{\text{sym}}_{\max} \\ &= \left(4 - \frac{4}{\rho}\right) m^{\text{sym}} d^{\text{sym}}_{\max} \end{aligned} \tag{6.6}$$

Now consider the second part - the work required to locate the fourth vertex after listing all the base shapes. Given a particular base triangle or wedge (u, v, w), the amount of work by S4GE_{CD} to locate the 4th vertex z through intersection is $d^{\text{sym}}(u) + d^{\text{sym}}(v) + d^{\text{sym}}(w)$. Similarly, this expression is upper bounded by $3\, d^{\text{sym}}_{\max}$. Also for each graph dataset, the numbers of triangles and wedges are fixed. $\text{D4GE}/\text{S4GE}_{\text{CD}}$ enumerates each triangle and wedges ρ times. This is required because given a triangle or wedge of color (i, j, k), the 4th vertex can have ρ different colors. All ρ colors are necessary to ensure that all graphlets would be enumerated. Thus overall, D4GE

with $\mathrm{S4GE_{CD}}$ does

$$3\,\rho\, d^{\mathrm{sym}}_{\mathrm{max}}(|\Delta| + |\angle|) \tag{6.7}$$

amount of work.

As discussed briefly at the beginning of Subsection 6.2.2, PSE may discover a four-node-graphlet in more than one subproblem group. The number of duplications depends on the number of colors of the triangles and wedges. This, in turn, determines the amount of work. We denote the uni-color triangles and wedges by Δ_I and $\angle_I$, the bi-color ones by Δ_{II} and $\angle_{II}$, and the tri-color ones by Δ_{III} and $\angle_{III}$. We can write

$$\begin{aligned} W^{\mathrm{PSE}}_{I} &= (1 + a(\rho))\, 3\, d^{\mathrm{sym}}_{\mathrm{max}}(|\Delta_I| + |\angle_I|) \\ W^{\mathrm{PSE}}_{II} &= (1 + b(\rho))\, 3\, d^{\mathrm{sym}}_{\mathrm{max}}(|\Delta_{II}| + |\angle_{II}|) \\ W^{\mathrm{PSE}}_{III} &= (1 + c(\rho))\, 3\, d^{\mathrm{sym}}_{\mathrm{max}}(|\Delta_{III}| + |\angle_{III}|) \end{aligned} \tag{6.8}$$

where $a(\rho), b(\rho), c(\rho)$ are positive functions of ρ representing the duplications in the three types. Their exact values depend on the instance of the input graph.

Expression 6.8 minus expression 6.7 yields

$$3\, d^{\mathrm{sym}}_{\mathrm{max}}\ (a(\rho)\,(|\Delta_I| + |\angle_I|) + b(\rho)\,(|\Delta_{II}| + |\angle_{II}|) + c(\rho)\,(|\Delta_{III}| + |\angle_{III}|)) \tag{6.9}$$

which is the amount of extra work PSE/S4GE does compared to D4GE/$\mathrm{S4GE_{CD}}$ for enumerating all the 4-node graphlets after all the wedges and triangles are discovered.

Now consider expressions 6.6 and 6.9. Expression 6.6 shows that the extra work performed by D4GE with $\mathrm{S4GE_{CD}}$ to discover all base triangles and wedges is sensitive to the size of the symmetrised graph, *ie.*, the number of edges and degrees; expression 6.9 shows that the extra work performed by PSE with S4GE to list the 4th vertex,

grows with respect to ρ and, is sensitive to the number of triangles and wedges. Note that for real-world graphs, the number of wedges plus triangles is often a magnitude greater than the number of edges, and for a reasonable-sized cluster, ρ is often set to a large value. As a result, D4GE with S4GE_{CD} can often achieve greater performance improvement. This will be confirmed in the experiments below.

6.2.6 Experiment

Our solution, D4GE, is implemented in Apache Spark 2.4.5 with OpenJDK 1.8.0. We experimented with it on several large real-world datasets, symmetrized (See Appendix A). In Table 6.1 we list the graphs we use here, along with the number of wedges and triangles.

Dataset	n	m	$\vert\angle\vert$	$\vert\Delta\vert$	$d^{\text{sym}}_{\text{max}}$
`enron`	69K	510K	40M	1M	1.6K
`cnr`	326K	5.6M	7.8B	21M	18K
`amazon`	735K	7M	38M	4.5M	1.1K
`hollywood09`	1.1M	114M	33B	4.9B	11K
`dewiki`	1.5M	33M	51B	89M	118K
`hollywood11`	2.2M	229M	100B	7.1B	13K
`orkut`	3M	234M	44B	628M	33K
`ljournal`	5.4M	100M	8.7B	441M	19K
`uk02`	18.5M	529M	188B	4.5B	195K
`enwiki18`	5.6M	235M	297B	378M	248K
`indochina`	7.4M	304M	392B	61B	256K

Table 6.1: The numbers of vertices n, edges m, wedges $|\angle|$, and triangles $|\Delta|$, and the max degree of the symmetrized graphs. The last three graphs are the largest and they require more computing power than the others.

For the smaller graphs, unless specifically stated otherwise, the experiments were conducted using 30 Intel E5430 quad-core machines with 6 GB of RAM each. This

gives equivalently 120 distributed workers[1] and 1.5 GB of RAM per worker. For the three largest graphs, `uk02`, `enwiki18` and `indochina`, we employed a larger cluster on Compute Canada[2] using 14 compute nodes, with 48 cores and 192 GB RAM per node. This configuration effectively gives us 672 workers with 4 GB RAM per worker, which can still be considered modest.

We compared the performance with the performance of the SotA, the PSE, and the single-machine solution S4GE. The single-machine experiment was conducted using a machine with dual Xeon E5-2620 processors and 128 GB of RAM. The total number of threads in this machine is 24. We set our time budget to be six days for each run.

The impact of ρ on performance

Let us first address the impact of ρ on the overall performance of our distributed algorithm. In previous literature, Suri and Vassilvitskii [69] regarded ρ as a trade-off between the network read and the input size of each distributed worker: a larger value of ρ increases the amount of network read, but also decreases the input size as each task becomes smaller. Park et al. [48] on the other hand adjusted ρ accordingly to the input graph size, to fully utilize the amount of available memory for each worker.

We show that while ρ affects the amount of network read, a large ρ value in practice can help with balancing the workload distribution, even when the number of sub-problems over-saturates the number of workers. Also, with a large enough ρ, the input size of each task shall never exceed the allocated memory for each worker. We experimented with D4GE/S4GE$_{\text{CD}}$ on three different datasets and varying $\rho = 8, 12, 16$ and 20. The result is shown in Figure 6.2.

When $\rho = 8$ there are $\binom{8}{2} + \binom{8}{3} + \binom{8}{4} = 154$ sub-problems, hence $\rho = 8$ is the minimum value to saturate our cluster of 120 workers. Any ρ greater than 8

[1] Each worker is equivalent to a physical CPU core.
[2] https://docs.computecanada.ca/wiki/Cedar

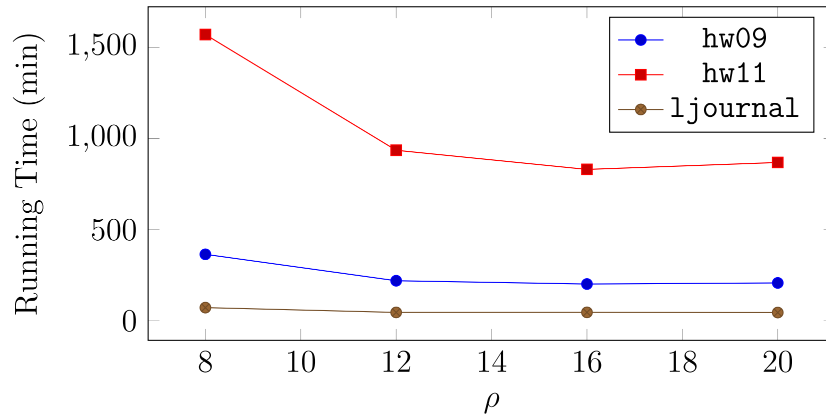


Figure 6.2: The enumeration time (minutes) of D4GE/S4GE$_{CD}$ on several graphs, with varying value of ρ. Higher ρ does not add much overhead; the lines flatten out rather than sloping up perceptibly.

will over-saturate the cluster. Theoretically, we should not see any improvement after $\rho = 8$, but in fact, we do. This is because of better load balancing. From $\rho = 8$ to 12, we observe improvement, consistently on various datasets. This shows that $\rho = 12$, with almost 5 times more sub-problems than $\rho = 8$, gives us a better workload distribution. However, the improvement diminishes and the performance would eventually decrease as ρ gets higher. The overhead of network read and Apache Spark framework itself could dwarf the computation when ρ is too large. We would like to note that the network read in our experiment is through internal traffic - i.e., traffic between distributed workers and distributed storage. Internal traffic is often free even on a commercial platform, and the internal network connection can be an order of magnitude faster than an external one. Even though a large ρ value introduces more network read, the performance penalty from the network read is negligible.

Machine scalability

We investigate the machine scalability of D4GE/S4GE by measuring the running time on `hollywood09` and `cnr` datasets while varying the number of distributed workers from 32 to 256. The results are presented in Figure 6.3. D4GE/S4GE shows

strong scalability: with slopes -0.968 and -0.899 respectively, which are very close to the perfect value -1. It means that the running time decreases by $2^{-0.968} = 1.956$ and $2^{0.899} = 1.865$ times, respectively, when the number of machines is doubled. We emphasize that this is on par with the SotA Map-Reduce based algorithms [48] and [50].

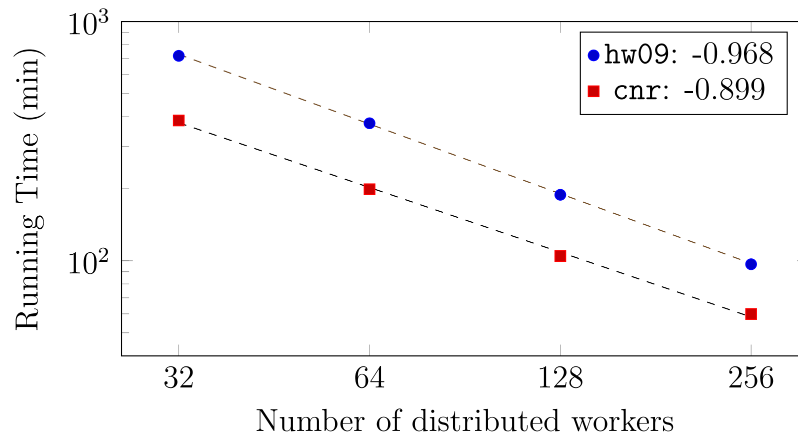


Figure 6.3: Machine scalability of D4GE/S4GE on `cnr` and `hollywood09`. This shows very strong scalability with slopes -0.899 and -0.968, which is very close to -1, the perfect value.

S4GE vs D4GE/S4GE$_{\text{CD}}$

Here we compare the running time of D4GE/S4GE$_{\text{CD}}$ to S4GE on a single machine. For this experiment, for D4GE/S4GE$_{\text{CD}}$ we used a cluster of 120 distributed workers, while for S4GE we used a machine with 24 threads. We set $\rho = 16$ for the distributed runs. Our results are shown in Table 6.2. For `hollywood09`, `dewiki`, `hollywood11`, and `orkut` using S4GE we abort the runs because they are over our time budget of 6 days. Note that 6 days is 8640 minutes. We notice that workload imbalance has a big impact on the S4GE runtime. Except for `amazon`, we see a big speedup for all the datasets. For `amazon`, the runtime is too short and the overhead for the distributed computing is larger than the gain. The runtime of `dewiki` is longer than the others

because it has a higher maximum degree.

Dataset	S4GE	D4GE/S4GE$_{\text{CD}}$	Speedup
`enron`	1.28	0.18	7.1
`cnr`	2933	132	22.2
`amazon`	0.23	0.37	0.6
`hollywood09`	$>$ 6 days	204	/
`dewiki`	$>$ 6 days	2328	/
`hollywood11`	$>$ 6 days	864	/
`orkut`	$>$ 6 days	390	/
`ljournal`	1367	47	29

Table 6.2: The enumeration time (minutes) of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, 120 workers, against S4GE (single machine) with 24 threads.

PSE/S4GE vs D4GE/S4GE$_{\text{CD}}$

Next, we modified PSE and replaced VF2 with S4GE in the PSE. We then compared D4GE/S4GE$_{\text{CD}}$ against PSE/S4GE. For this experiment, we set $\rho = 16$ and use the same cluster configuration for both. The enumeration times are listed in Table 6.3. We found that D4GE/S4GE$_{\text{CD}}$ is more efficient for all of the tested graphs, and D4GE/S4GE$_{\text{CD}}$ is able to achieve up to 11x speedup, which is on the **cnr** dataset.

A significant speedup is achieved on `cnr`, `hollywood09`, `hollywood11`, `orkut` and `ljournal`. For `dewiki`, we can deduce that the speedup is > 3.7 (i.e., 8640/2328). For these datasets, the number of wedges plus triangles is much greater than the number of edges, as can be seen in Table 6.1. According to expressions 6.6 and 6.9, PSE's performance is penalized by the number of triangles from type-1 sub-problems and wedges, whereas D4GE's performance is penalized no more than the number of edges from the symmetrised graph. This gives advantage to D4GE/S4GE$_{\text{CD}}$ compared to PSE/S4GE.

Dataset	PSE/S4GE	D4GE/S4GE$_{\text{CD}}$	Speedup
`enron`	0.55	0.18	3.1
`cnr`	1446	132	11.0
`amazon`	0.37	0.37	1.0
`hollywood09`	2190	204	10.7
`dewiki`	$>$ 6 days	2328	/
`hollywood11`	9186	864	10.6
`orkut`	3799	390	9.7
`ljournal`	432	47	9.2

Table 6.3: The enumeration time (minutes) of D4GE/S4GE$_{\text{CD}}$ against PSE/S4GE, with $\rho = 16$, 120 workers.

PSE/VF2 vs D4GE/CF4$_{\text{CD}}$

Lastly, we compare the performance of Park et al.'s PSE/VF2 implementation on 4-clique query, against our D4GE/CF4$_{\text{CD}}$. The results are shown in Table 6.4. Comparing our D4GE/CF4$_{\text{CD}}$ suite against one of the *state-of-the-art* sub-graph enumeration algorithm, up to 5.2 fold speedup is observed on a small graph such as `amazon`, and $>$ 20 fold speedup on a large graph such as `hollywood09`.

Dataset	PSE/VF2	CD$_{\text{ext}}$/CF4$_{\text{CD}}$	Speedup
`enron`	0.7	0.15	4.7
`cnr`	1.1	0.33	3.3
`amazon`	1.3	0.25	5.2
`hollywood09`	324	16	20.3
`dewiki`	4.5	1.5	3.0
`hollywood11`	288	31	9.3
`orkut`	16	5.0	3.2
`ljournal`	11	2.5	4.4

Table 6.4: Enumeration time (minutes) of D4GE/CF4$_{\text{CD}}$ against PSE/VF2, with $\rho = 16$

We emphasize that the overall speedup of D4GE against PSE is also because

D4GE guarantees no duplication during the enumeration. We obtained Park et al.'s PSE+VF2 implementation [3], version 3.0.1, and we modified the source code to count the number of duplicate emissions. We list the percentages of duplicate emissions from PSE/S4GE, PSE/CF, and PSE/VF2. For PSE/S4GE, we list the median percentage of the duplications for all six types of 4-node graphlets, and we query 4-clique against VF2. The results are presented in Table 6.5. We can see that the PSE partitioning scheme, when combined with the S4GE algorithm, emits around 300% of duplicates. The percentages are around 40% for PSE/CF4, and lower for PSE/VF2. From this table, we might deduce that PSE was indeed designed to work together with VF2, but not suited for S4GE.

Dataset	S4GE	CF4	VF2(K_4)
`enron`	255%	36%	19%
`cnr`	245%	32%	14%
`amazon`	270%	36%	1.2%
`hollywood09`	379%	41%	29%
`dewiki`	/	39%	25%
`hollywood11`	305%	41%	29%
`orkut`	246%	41%	29%
`ljournal`	309%	41%	26%

Table 6.5: Duplicated emissions from PSE partitioning scheme with different local algorithms.

Comparing the second and the third column of Table 6.5 on duplicate emissions, we can see that localized VF2 algorithm emits fewer duplicated 4-cliques than CF4, when both are using the same PSE partitioning scheme. Yet, still up to 29% of duplicates are emitted by VF2, from both `hollywood` and `orkut` datasets.

The overall results show that D4GE/$CF4_{CD}$ has better performance than PSE/VF2 for enumerating 4-cliques. However, we also acknowledge that PSE/VF2 might suffer

[3] From https://datalab.snu.ac.kr/pegasusn/download.php

from its generality in this particular comparison. D4GE/CF4$_{\text{CD}}$ is tuned to enumerating 4-cliques only whereas VF2 is capable of answering any k-order sub-graph query.

We also want to emphasize that the comparison here is aimed to show the performance gain of D4GE over PSE; while D4GE/S4GE$_{\text{CD}}$ can be revised to query 4-cliques, it is designed for a bigger goal - enumerating all 4-node graphlets.

The Output of D4GE/S4GE$_{\text{CD}}$

Here we summarize the results of our experiments with D4GE/S4GE$_{\text{CD}}$. We list the counts of graphlets in Tables 6.6 and 6.7.

Graphlet	`enron`	`cnr`	`amazon`	`hw09`
3-path	2.51B	6.12B	372M	21.4T
3-star	8.04B	41.4T	610M	16.7T
4-cycle	21.6M	37.9B	2.69M	168B
tailed-triangle	583M	79.4B	92.3M	8.87T
diamond	46.1M	43.0B	13.1M	635B
4-clique	5M	160M	4.19M	1.39T
Running Time (min):	0.18	132	0.37	204

Table 6.6: The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, on a cluster of 120 workers.

For the largest datasets, `uk02`, `enwiki18` and `indochina`, we employed a larger cluster of 672 workers with 4 GB RAM per worker. On this cluster, we set ρ to 25, which gives us 15,250 sub-problems. The results are shown in Table 6.8. D4GE/S4GE$_{\text{CD}}$ was able to complete `uk02` in about 30 hours, `enwiki18` in 82 hours, and `indochina` in 124 hours, enumerating more than 2, 7.5 and 10 quadrillion graphlets in total. We emphasize that, to the best of our knowledge, there is no existing algorithm that can enumerate all the 4-node graphlets in a dataset of this

Graphlet	`dewiki`	`hw11`	`orkut`	`ljournal`
3-path	10.4T	104T	18.6T	1.81T
3-star	661T	92.8T	97.8T	8.85T
4-cycle	13.1B	643B	70.1B	8.55B
tailed-triangle	993B	26.8T	1.51T	190B
diamond	11.9B	1.88T	47.8B	27B
4-clique	158M	728B	3.22B	16.1B
Running Time (min):	2328	864	390	47

Table 6.7: The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 16$, on a cluster of 120 workers.

scale in a feasible amount of time. We estimate that, for each, PSE/S4GE would take more than 7 days to run using the same cluster, which is impractical. Note that 1 day = 24 hours = 1440 minutes.

Graphlet	`uk02`	`enwiki18`	`indochina`
3-path	1.9T	66.2T	7.6T
3-star	1.97Q	7.4Q	10.01Q
4-cycle	238B	76B	617B
tailed-triangle	6.1T	5.1T	9.3T
diamond	1.8T	61.7B	3.3T
4-clique	157B	876M	99.3T
Running Time (min):	1800	4885	7416

Table 6.8: The outputs of D4GE/S4GE$_{\text{CD}}$ with $\rho = 25$, on a cluster of 672 workers.

6.2.7 Discussion

It is common in the literature that performance or scalability is measured against the size of the input graph, either by the number of vertices or more commonly the number of edges. We would like to point out that in the context of 4-node graphlet enumeration, using the S4GE algorithm, the number of vertices or edges should not

be the primary consideration when it comes to the amount of computation. In [63] it was shown that S4GE algorithm is bounded by $T_{3g} + (|\angle| + |\Delta|)d_{\max}^{\text{sym}}$, where T_{3g} is the time to enumerate all the wedges and triangles. As a consequence, a *small* graph such as `dewiki` can have a much longer runtime than graphs of larger size, such as `ljournal`. As can be seen in Table 6.3, `ljournal`, which is three times larger than the `dewiki` in size, has a runtime that is only 2% of the `dewiki`'s. Notice that `dewiki` has a much larger number of graphlets, in particular the 3-stars. We plot the enumeration time of eight small-medium datasets against $d_{\max}^{\text{sym}}(|\Delta| + |\angle|)$ in Figure 6.4. From our experiments, the enumeration time demonstrates a high correlation with respect to $d_{\max}^{\text{sym}}(|\Delta| + |\angle|)$. This shows that the total number of graphlets is an important metric to measure the performance of an enumeration algorithm. The correlation also shows that the D4GE performs as expected, i.e. it does not distort the single-machine solution expectation.

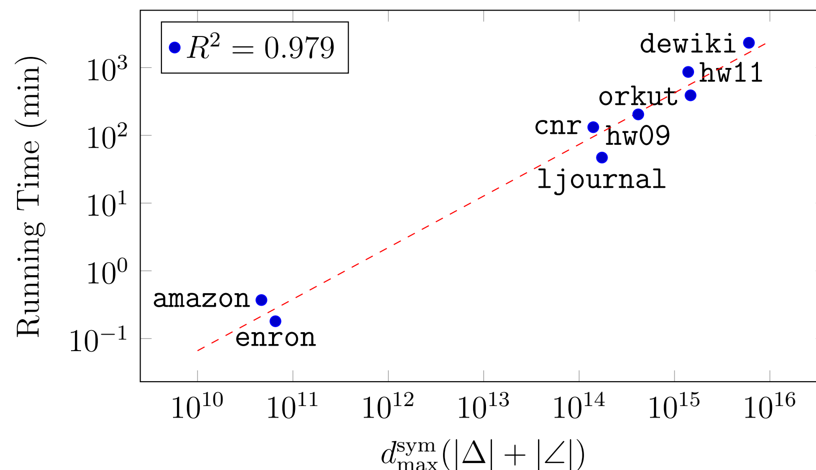


Figure 6.4: Strong correlation between the enumeration time and $d_{\max}^{\text{sym}}(|\Delta| + |\angle|)$ on the small-medium datasets.

Another question that the readers might ask is why enumerate all types of 4-node graphlets in a single run? Why not just one type at a time, like many other solutions? Our answer is that we can turn off any pattern that we do not want in the S4GE,

and do some optimization for each. However, if we need all types of graphlets, for a complete analysis, it will be more efficient to do them all at once rather than do them one by one. Notice that due to its design, the running time of S4GE is less than the sum of the times for enumerating the six graphlet types individually.

6.3 Distributed Triad Enumeration

While Algorithm 10 (FPTE) is already parallelized (line 2), its scalability is limited to a single machine-shared memory model. In order for the FPTE to enjoy the multi-machine - discrete memory computing clusters with a much higher degree of parallelism, we fit FPTE under the duplication-free partition scheme that D4GE proposed earlier. To achieve this, we modified D4GE to handle 3-node sub-graphs and modified FPTE algorithm to work with directed edgesets.

Because the D4GE partitioning scheme operates independently of the serial algorithm, there are only three minor changes required. First, the generated sub-problems and color-assignments are reduced from size 4 down to size 3, so the total number of sub-problems is now $\binom{\rho}{2} + \binom{\rho}{3}$. Second, there is no need for symmetrization. Third, because FPTE operates on both G and G^T, the partitioning is applied to both G and G^T as well, and for each single sub-problem, the directed edgesets of G and G^T are loaded into memory. The modified partitioning scheme is described as Algorithm 19.

Modification to FPTE is also minimal. The modification here follows the same fashion as migrating S4GE to S4GE$_{\text{CD}}$: instead of the entire graph G and its transpose G^T, modified FPTE enumerates over directed edgesets E^*_{pq} and E^{T*}_{pq} given a color-assignment ijk. Specifically, given color-assignment ijk, line 2 and 3 of FPTE (Algorithm 10) requires edgesets E^*_{ij} and E^{T*}_{ij}; and line 7 of FPTE requires edgesets E^*_{ik} and E^{T*}_{ik} for knowledge of $(N^+(u)$ and $N^-(u))$, and E^*_{jk} and E^{T*}_{jk} for knowledge

Algorithm 19 D3GE

Input: A directed graph $G(V, E)$ and its transpose $G^T(V, E^T)$; number of colors ρ

1: Construct $\overrightarrow{G}(V, \overrightarrow{E})$ and $\overrightarrow{G^T}(V, \overrightarrow{E^T})$ by edge-orientation of $G(V, E)$ and $G^T(V, E)$
2: Partition $\overrightarrow{E}$ and $\overrightarrow{E^T}$ into directed edge sets E^*_{ij} and E^{T*}_{ij} using ρ
3: Generate ordered color assignments and sub-problems $\{S_{\text{Cs}} \mapsto \{K_{ijkl}\}\}$
4: **for all** S_{Cs}, $\{K_{ijkl}\}$ **do** ▷ Distributed-for
5: $E_{\text{map}}, E^T_{\text{map}} \leftarrow$ ReadEdgeSets$_{\text{CD}}$ $(S_{\text{Cs}}, 3)$
6: **for all** $K_{ijkl} \in \{C_0, C_1, ...\}$ **do**
7: FPTE$_{\text{CD}}$ $(E_{\text{map}}, E^T_{\text{map}}, ijkl)$

of ($N^+(v)$ and $N^-(v)$). The rest of FPTE stays unmodified, as the edgesets solely supply the corresponding neighbourhood information but do not alter the behavior of the algorithm. We call this modified version of FPTE as FPTE$_{\text{CD}}$, and is summarized as Algorithm 20.

Algorithm 20 FPTE$_{\text{CD}}$

Input: Two mappings from the colors of directed edge set to the edge set $E_{\text{map}} \equiv \{(i, j) \mapsto E^*_{ij}\}$ and $E^T_{\text{map}} \equiv \{(i, j) \mapsto E^{T*}_{ij}\}$; ordered color assignment $ijkl$
Output: The number of each type of triads in E_{map} and E^T_{map}, Δ_i.

1: $\Delta_1 \leftarrow 0, \ldots, \Delta_7 \leftarrow 0$
2: $E^*_{ij} \equiv E_{\text{map}}[ij]$, $E^*_{ik} \equiv E_{\text{map}}[ik]$, $E^*_{jk} \equiv E_{\text{map}}[jk]$
3: $E^{T*}_{ij} \equiv E^T_{\text{map}}[ij]$, $E^{T*}_{ik} \equiv E^T_{\text{map}}[ik]$, $E^{T*}_{jk} \equiv E^T_{\text{map}}[jk]$
4: **for all** $u \in E^*_{ij} \cup E^{T*}_{ij}$ **do**
5: **while** there is next **do**
6: $N^+(u) \equiv E^*_{ij}[u]$, $N^-(u) \equiv E^{T*}_{ij}[u]$.
7: Find next neighbour in $N^+(u)$ and/or $N^-(u)$: v.
8: Code the link uv as $e1$: either 01, 10 or 11
9: **while** there is next **do**
10: $N^+(u) \equiv E^*_{ik}[u]$, $N^-(u) \equiv E^{T*}_{ik}[u]$.
11: $N^+(v) \equiv E^*_{jk}[v]$, $N^-(v) \equiv E^{T*}_{jk}[v]$.
12: Find next common neighbour of u and v: w, in $(N^+(u), N^-(u))$ and $(N^+(v), N^-(v))$.
13: Code the links vw as $e2$, and wu as $e3$.
14: Look up triad type i using $e1, e2, e3$.
15: `enum`$(u, v, w, e1, e2, e3)$
16: $\Delta_i \leftarrow \Delta_i + 1$

6.3.1 Experiment

For the experiments, we used a Compute Canada cluster with 4 compute nodes, each node with 32 cores and 128 GB RAM per node. This configuration effectively gives us 128 workers with 4 GB RAM per worker. We set ρ to 12, yielding 286 sub-problems. For comparison, we also ran experiments on a single machine. The configuration of the machine is of dual Intel Xeon E5620 CPUs, for a total of 16 threads, and 64 GB RAM. The datasets are listed in Table 6.9. These are selected to cover the comparison against the ones already in [64], plus five additional datasets of varying sizes.

Dataset	n	m	$d_{\max}$	$d^T_{\max}$	$d^{\text{eff}}_{\max}$	$d^{T\text{eff}}_{\max}$
`cnr`	326K	3.2M	2,716	18,235	1,336	81
`dewiki`	1.5M	36M	5,032	117,908	5,032	409
`ljournal`	5.4M	79M	2,469	19,409	1,257	397
`enwiki18`	5.6M	128M	7,948	247,628	7,620	311
`indochina`	7.4M	194M	6,985	256,425	6,870	6,821
`uk02`	18.5M	298M	2,450	194,942	2,288	942
`arabic`	22.7M	640M	9,905	575,618	6,646	3,126
`uk05`	39.5M	936M	5,213	1,776,852	5,213	584
`twitter`	41.7M	1.5B	2,997,469	770,155	2,896	5,745

Table 6.9: The numbers of vertices n, edges m, maximum degree of the original graph $d_{\max}$ and its transpose $d^T_{\max}$, and the effective maximum degrees after the preprocessing, $d^{\text{eff}}_{\max}$ and $d^{T\text{eff}}_{\max}$, of the graph datasets.

The enumeration times are listed in Table 6.10. The last five graphs were not listed in the FPTE paper. Even for the largest graph, `twitter`, with 42M vertices and 1.5B edges, D3GE/FPTE$_{\text{CD}}$ is able to enumerate all seven types of triads within 8 minutes, delivering a very strong performance. With eight times the parallelism, compared against the single machine FPTE, for `cnr`, `ljournal`, `uk05`, `dewiki`, `enwiki18` and `uk02`, the speedups are less than 4 fold. While the performance is still

improved, these low speedups do not meet the expectation. This is because the original FPTE, while limited on a single machine, has the advantage of the shared-memory model, which makes the computation efficient - no partitioning is required. D3GE exposes FPTE to a cluster of workers, and this bears a cost. Because of the discrete-memory model of the clusters, we have to pre-partition the input graph into **overlapping** and independent sub-graphs (sub-problems) and let the workers solve each of the sub-problems. The overlapping of the sub-graphs is necessary because, in the discrete-memory model, the workers cannot access each other's memory content. In other words, if D3GE/FPTE$_{\mathrm{CD}}$ and FPTE are given the same number of workers/threads, D3GE/FPTE$_{\mathrm{CD}}$ inherently does more work per worker, due to the overlap. Additionally, the overlapping portion grows with respect to ρ, further discounting the distributed solution as compared to the shared-memory model. However, we would like to stress that this problem is not particular to D3GE/FPTE$_{\mathrm{CD}}$. All known partition schemes suffer from the inevitable overlap. Note that this comparison here only shows the performance improvement over the single machine, not the scalability. The true scalability of D3GE will be discussed later.

Dataset	FPTE	D3GE/FPTE$_{\mathrm{CD}}$	Speedup
`cnr`	3.0	2.2	1.36
`ljournal`	81	24.4	3.3
`arabic`	2961	107.9	27.4
`uk05`	796	207.1	3.8
`dewiki`	30.3	14.0	2.16
`enwiki18`	136.4	34.5	3.95
`indochina`	9,228.8	57.2	161.3
`uk02`	191.5	92.7	2.07
`twitter`	51,984.0	445.6	116.7

Table 6.10: The enumeration time (seconds) of D3GE/FPTE$_{\mathrm{CD}}$ with $\rho = 12$ and 128 workers, against original FPTE with 16 threads on a single machine.

On the other hand, D3GE/FPTE_{CD} is able to achieve a 27.4 speedup on the `arabic` dataset, a 161.3 speedup on the `indochina` dataset, and a 116.7 speedup on the `twitter` dataset. However, this is not because D3GE/FPTE_{CD} works much faster on these datasets, but rather it reflects the poor performance of FPTE. Upon inspecting the datasets in Table 6.9, we can see that these three datasets have relatively large maximum effective degrees. FPTE suffers from having to do all the work for the node with the highest effective degree on a single thread. This is similar to 'the curse of the last reducer problem' as pointed out by Suri and Vassilvitskii [69]. D3GE/FPTE_{CD} avoids this problem through a partitioning scheme that leads to a better workload balance.

Next, we experimented on the scalability of D3GE/FPTE_{CD} by plotting the enumeration time over the `arabic` and `uk02` datasets, using varying numbers of distributed workers from 32 to 256. The results are shown in Figure 6.5. Similar to Figure 6.3, FPTE_{CD} fitted under D3GE scales almost perfectly with respect to the degree of parallelism: the slopes are -0.894 and -0.866 respectively. This means that every time the number of the distributed workers doubles, the enumeration time is reduced by factors of $2^{0.894} = 1.858$ and $2^{0.866} = 1.822$ respectively. We re-confirm that the scalability of our proposed distributed partitioning scheme is on-par with the SotA Map-Reduce-based ones [48] and [50].

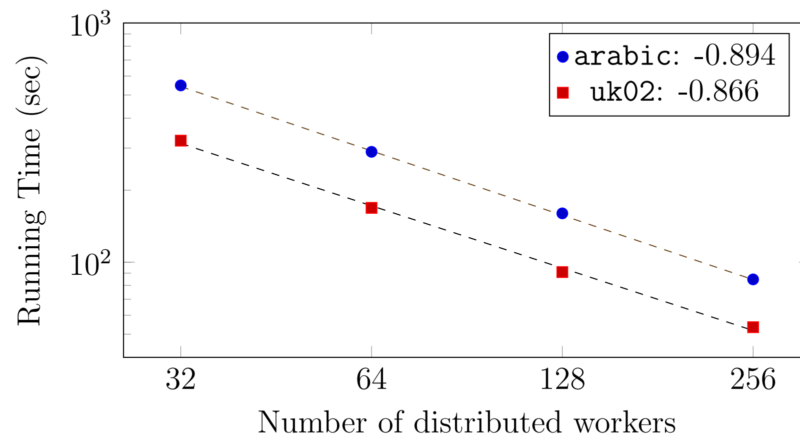


Figure 6.5: Scalability of D3GE/FPTE$_{\text{CD}}$ on `uk02` and `arabic`. D3GE/FPTE$_{\text{CD}}$ again presents very strong scalability with slope -0.866 and -0.894.

Chapter 7

Future Work

We have done a comprehensive study on graphlet enumeration and proposed many efficient algorithms and solutions. Nonetheless, there is still more work that can be done in the future. This includes, but is not limited to:

- Building an algorithm for and implementing directed 4-node graphlet enumeration.
- Conducting more experiments on our 5-node graphlet enumeration solution, S5GE. Also, deploying this onto a distributed platform, similar to what we did with S4GE and D4GE.
- Exploring higher-order graphlets (with six or more nodes), and searching for efficient enumeration for some special types.
- Extending our study to probabilistic graphlets.
- Extending our study to typed and labeled graphs.

We expand these ideas a bit further below.

7.1 Larger Directed Graphlets

We have laid the foundation for building an enumeration program for 4-node directed graphlets in Section 5.3. However, we still need to build the algorithm and the code implementation. This would be an extension of our triad enumeration FPTE, combined with the 4-node graphlet enumeration S4GE. We can continue our effort on the 5-node directed graphlets as well. The work might be tedious, because of the large number of graphlet types, but should be quite straightforward following our methods.

7.2 5-node Graphlet Enumeration

Up to now, we have completed our experiments on our 5-node graphlet enumeration algorithm S5GE only on a few graphs. We can do more experiments by applying it to other various graphs and checking, for example, the running time, and its scalability. As we have mentioned before, the running time of any enumeration program is bounded below by the number of graphlets. This, in practice, would limit the size of the graphs that we can process using a single machine. For this reason, we may want to bring our solution to the distributed platform as well, through a similar method used in D4GE.

7.3 Larger Order Graphlets

The number of graphlet types grows rapidly with the order of the graphlet. However, this, in principle, should not stop us from considering higher-order graphlets. Using triangles and wedges as the base, we have shown that it is not difficult to discover all of the graphlets in an input graph. The bigger problem is on avoiding

multi-listings, as some graphlets can be discovered in many ways. So far, we have not found a systematic way to avoid double-counting. In S5GE, we solve this problem ad-hoc, by looking at case by case. To be able to do higher order graphlets we need a better way.

On the other hand, we can also look at how to enumerate graphlets of some special types. Triangles are easier to enumerate than wedges because triangles have higher symmetries. In general, cliques are easier to enumerate compared to other graphlets of the same order. So, there have been efficient solutions to enumerate cliques, e.g., [22]. We can look at other types of graphlets as well, to see if we can enumerate those types more efficiently than the rest, although less efficient than cliques.

7.4 Probabilistic Graphlets

Throughout this dissertation, we have talked solely about deterministic graphs, where the number of nodes and edges are fixed. In a probabilistic graph, edges are not definite, but probabilistic. Consequently, graphlets are also probabilistic. The problem that we want to solve is as follows: Given a probabilistic graph $\tilde{G} = (G, P)$, we want to enumerate all graphlets that have a probability to occur more than a certain value (or threshold), γ. Notice that answering this question would also answer the question of *how many* that has a probability greater than the threshold, i.e. the counts for each type of graphlet.

The naive solution for probabilistic graphlets enumeration is to post-filter out the graphlets whose probabilities are less than the threshold. However, this approach does not take advantage of the probabilistic nature at all. In fact, with this approach, any γ will yield the same enumeration time, while in principle a large γ should require much less computing power as it has less number of graphlets to enumerate.

We found that S4GE is suitable to answer this question. We leverage the fact that S4GE is an intersection-based algorithm and that it discovers 4-node graphlets gradually: from edges to wedges/triangles, and then to 4-node graphlets. This gives us a hint that we can apply probability-based pruning at each stage of the discovery pipeline. First and foremost, only edges whose probabilities $p_e \geq \gamma$ need to be considered. Upon the discovery of wedges/triangles, only those with a probability greater than γ need to be processed further. Lastly, the surviving wedges/triangles are used to discover 4-node graphlets. The final probability of the 4-node graphlets is checked against γ before being emitted.

We have done some preliminary work on this possible solution in [38]. However, we still need to do a more detailed analysis of the theoretical framework.

7.5 Typed and Labeled Graphs

A typed graphlet is a graphlet that has more than one type of nodes [59, 58]. This is illustrated by the example in Figure 7.1. A graphlet enumeration, therefore, needs to keep track of the types. The algorithms that had been proposed in those papers are estimation algorithms. To the best of our knowledge, there has not been an algorithm that fully enumerates typed graphlets.

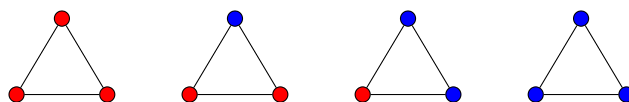


Figure 7.1: Typed triangles with two types of nodes.

Another interesting topic is how to compute graphlets in a weighted graph [30]. In a weighted graph, each edge has a weight. An example is shown in Figure 7.2. In a real-world network, the weights can be an important characteristic that needs to

be incorporated when we search for graphlets.

Figure 7.2: Two weighted graphlets of the same type, but with different sets of weights.

We can also take a look at multidimensional or multiplex graphs [26], where now the edges are the ones that have types, and each type represents a relationship. Furthermore, two nodes can have many edges of various types between them. An illustration is shown in Figure 7.3. In graphlet enumeration we may want to look for graphlets in each type, or graphlets in all types, or some combinations. More generally, we can also consider graphlets in multilayer graphs [60], where the edges can traverse the dimensions.

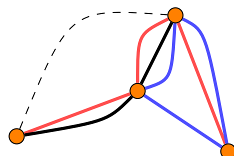


Figure 7.3: A multidimensional graph. Each edge has a type based on the dimension, represented by color and line-type.

Chapter 8

Conclusion

Graphlet enumeration is an important component in graph analysis. There have been many studies in the literature about this topic. Nevertheless, this task is not trivial when the input graph is large, requiring high efficiency in both the running time and the memory space. Not many papers deal with this particular problem, especially for graphlets of order four or higher, and for directed graphlets.

We have contributed to the progress in this field by proposing several algorithms and solutions. Up to now, we are able to enumerate all types of 3, 4, and 5-node graphlets in a single run using only a single commodity machine, even on some graphs of the order of a million nodes. Note, however, that the running time depends on the maximum degree. Thus, not all graphs of a million nodes can be processed within a limited time. More generally, we cannot avoid the fact that the running time of any enumeration algorithm, no matter how efficient they are, is bounded from below by the number of graphlets. With a single machine, we were able to enumerate trillions of graphlets in less than a day.

This limitation can be alleviated by using a distributed computing platform. We have been, up to now, able to deploy our 4-node graphlet enumeration solution on

a distributed platform, and we were able to enumerate quadrillions of graphlets in a few days. Experimentally we showed that the scheme that we use yields a good scalability, of around 0.9.

Nevertheless, there are still many challenges that we would like to study in our future work. These include enumerating higher-order graphlets, for both directed and undirected cases, and how to avoid multiple-listing in a systematic way.

Enumeration provides us with rich information about a graph. This can be used to build features that can then be used as input to some graph machine learning solutions. It could lead to a potentially significant improvement in the machine learning performance.

Bibliography

[1] Carlo Abrate and Francesco Bonchi. Counterfactual graphs for explainable classification of brain networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2495–2504, 2021.

[2] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining*, pages 1–10. IEEE, 2015.

[3] Albert-László Barabási. Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987):20120375, 2013.

[4] Vladimir Batagelj and Andrej Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23(3):237–243, 2001.

[5] Vladimir Batagelj and Matjaž Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 307(3-5):310–318, 2007.

[6] Smriti Bhagat, Graham Cormode, and S Muthukrishnan. Node classification in social networks. *arXiv preprint arXiv:1101.3291*, 2011.

[7] Pooja Bhojwani. Triangle enumeration in massive graphs using Map Reduce. Master's thesis, University of Victoria, 2018.

[8] Mansurul A Bhuiyan, Mahmudur Rahman, Mahmuda Rahman, and Mohammad Al Hasan. Guise: Uniform sampling of graphlets for large graph analysis. In *2012 IEEE 12th International Conference on Data Mining*, pages 91–100, 2012.

[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[11] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601. ACM Press, 2004.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework II: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 528. IEEE, 2004.

[13] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory*. Springer Publishing Company, Incorporated, 2008.

[14] Matthias Bröcheler, Andrea Pugliese, and Venkatramanan S Subrahmanian. Cosi: Cloud oriented subgraph identification in massive social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 248–255. IEEE, 2010.

[15] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[16] Gary Chartrand, Linda Lesniak, and Ping Zhang. *Graphs & digraphs*, volume 39. CRC press, 2010.

[17] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials*, 31(9):3564–3572, 2019.

[18] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[19] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[20] George Chin Jr, Andres Marquez, Sutanay Choudhury, and John Feo. Scalable triadic analysis of large-scale graphs: Multi-core vs. multi-processor vs. multi-threaded shared memory architectures. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 163–170. IEEE, 2012.

[21] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[22] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 589–598. International World Wide Web Conferences Steering Committee, 2018.

[23] James A Davis and Samuel Leinhardt. The structure of positive interpersonal relations in small groups. *Sociological Theories in Progress*, 2:218–251, 1972.

[24] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1036–1050, 2005.

[25] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[26] Tamara Dimitrova, Kristijan Petrovski, and Ljupcho Kocarev. Graphlets in multiplex networks. *Scientific reports*, 10(1):1–13, 2020.

[27] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences*, 99(9):5825–5829, 2002.

[28] Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221–233, 2010.

[29] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics*, 22(6):bbab159, 2021.

[30] Hongyu Guo, Khalique Newaz, Scott Emrich, Tijana Milenkovic, and Jun Li. Weighted graphlets and deep neural networks for protein structure classification. *arXiv preprint arXiv:1910.02594*, 2019.

[31] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[32] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[33] Paul W Holland and Samuel Leinhardt. Local structure in social networks. *Sociological methodology*, 7:1–45, 1976.

[34] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl_1):i213–i221, 2005.

[35] Shweta Jain and C Seshadhri. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 268–276, 2020.

[36] Tommaso Lanciano, Francesco Bonchi, and Aristides Gionis. Explainable classification of brain networks via contrast subgraphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3308–3318, 2020.

[37] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[38] Xiaozhou Liu, Yudi Santoso, Venkatesh Srinivasan, and Alex Thomo. Distributed enumeration of four node graphlets at quadrillion-scale. In *33rd International Conference on Scientific and Statistical Database Management*, pages 85–96, 2021.

[39] R Duncan Luce and Albert D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[40] Dror Marcus and Yuval Shavitt. Rage–a rapid graphlet enumerator for large networks. *Computer Networks*, 56(2):810–819, 2012.

[41] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877*, 2019.

[42] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[43] Tijana Milenković and Nataša Pržulj. Uncovering biological network function via graphlet degree signatures. *Cancer informatics*, 6:CIN–S680, 2008.

[44] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[45] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[46] Sindhuja Parimalarangan, George M Slota, and Kamesh Madduri. Fast parallel graph triad census and triangle counting on shared-memory platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1500–1509. IEEE, 2017.

[47] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13*, pages 539–548, 2013.

[48] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. Pte: Enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1115–1124. ACM, 2016.

[49] Ha-Myung Park, Francesco Silvestri, U. Kang, and Rasmus Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014*, pages 1739–1748, 2014.

[50] Ha-Myung Park, Francesco Silvestri, Rasmus Pagh, Chin-Wan Chung, Sung-Hyon Myaeng, and U Kang. Enumerating trillion subgraphs on distributed systems. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(6):1–30, 2018.

[51] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1431–1440. International World Wide Web Conferences Steering Committee, 2017.

[52] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[53] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[54] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences*, 101(9):2658–2663, 2004.

[55] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, 2014.

[56] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural networks*, 18(8):1093–1110, 2005.

[57] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and robust distributed subgraph enumeration. *arXiv preprint arXiv:1901.07747*, 2019.

[58] Ryan A Rossi, Nesreen K Ahmed, Aldo Carranza, David Arbour, Anup Rao, Sungchul Kim, and Eunyee Koh. Heterogeneous graphlets. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(1):1–43, 2020.

[59] Ryan A Rossi, Anup Rao, Tung Mai, and Nesreen K Ahmed. Fast and accurate estimation of typed graphlets. In *Companion Proceedings of the Web Conference 2020*, pages 32–34, 2020.

[60] Sallamari Sallmen, Tarmo Nurmi, and Mikko Kivelä. Graphlets in multilayer networks. *arXiv preprint arXiv:2106.13011*, 2021.

[61] Yudi Santoso. Triangle counting and listing in directed and undirected graphs using single machines. Master's thesis, University of Victoria, 2018.

[62] Yudi Santoso, Xiaozhou Liu, Venkatesh Srinivasan, and Alex Thomo. Four node graphlet and triad enumeration on distributed platforms. *Distributed and Parallel Databases*, pages 1–38, 2022.

[63] Yudi Santoso, Venkatesh Srinivasan, and Alex Thomo. Efficient enumeration of four node graphlets at trillion-scale. In *23rd International Conference on Extending Database Technology*, pages 439–442, 2020.

[64] Yudi Santoso, Alex Thomo, Venkatesh Srinivasan, and Sean Chester. Triad enumeration at trillion-scale using a single commodity machine. In *22nd International Conference on Extending Database Technology*. OpenProceedings.org, 2019.

[65] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of 4th International Workshop on Experimental and Efficient Algorithms, WEA 2005,*, pages 606–609, 2005.

[66] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, volume 4, page 5, 2013.

[67] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495, 2009.

[68] Francesco Silvestri. Subgraph enumeration in massive graphs. *arXiv preprint arXiv:1402.3444*, 2014.

[69] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pages 607–614. ACM, 2011.

[70] Benjamin M Tabak, Marcelo Takami, Jadson MC Rocha, Daniel O Cajueiro, and Sergio RS Souza. Directed clustering coefficient as a measure of systemic risk in complex banking networks. *Physica A: Statistical Mechanics and its Applications*, 394:211–216, 2014.

[71] Nilothpal Talukder and Mohammed J Zaki. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery*, 30(5):1024–1052, 2016.

[72] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[73] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment*, 11(2):162–175, 2017.

[74] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge University Press, 1994.

[75] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[76] Sebastian Wernicke and Florian Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[77] Serene WH Wong, Nick Cercone, and Igor Jurisica. Comparative network analysis via differential graphlet communities. *Proteomics*, 15(2-3):608–617, 2015.

[78] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence*, 2(2):109–127, 2021.

[79] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. G-thinker: A distributed framework for mining

subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380. IEEE, 2020.

[80] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):2, 2014.

[81] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. Distributed subgraph counting: a general approach. *Proceedings of the VLDB Endowment*, 13(12):2493–2507, 2020.

Appendix A

Graph Datasets

For our experiments, we used several data sets that we retrieved from the WebGraph website [11, 10, 9]:

http://law.di.unimi.it/datasets.php

Most of those graphs are directed and come in pairs of graphs and transpose graphs. For experimenting on undirected graphs, we symmetrize the graphs using the mechanism described in Section 2.1. In the experiments, we use the same names (abbreviated) for both the directed graphs and the undirected graphs (after symmetrization). However, it should be clear from the context which ones we refer to.

Here we collect all graphs that we used in various experiments. Note that any one of the experiments does not use all of the graphs listed here, but only a subset of them.

A.1 Directed Graph Datasets

The summaries of the directed graphs are shown in Table A.1. We sort the graphs in ascending order, $|V|$. Note that the sorting will be different if we use the size,

$|E|$, instead. The smallest graph is the `wordassociation-2011`, which we often abbreviate as `word` and has about ten thousand nodes and seventy two thousand edges. The largest one is the `clueweb12`, or `clueweb` for short, with almost a billion nodes and forty two billion edges. In between, we have graphs of various orders and sizes. They are grouped roughly on the magnitude of the orders. The `clueweb` is in its own class, as it is much larger than the others.

Name	$\|V\|$	$\|E\|$	$d^+_{\max}$	$d^-_{\max}$
`wordassociation-2011`	10.6K	72.2K	34	324
`enron`	69.2K	276.1K	1,392	1,394
`uk-2007-05@100000`	100K	3.05M	3,753	55,252
`cnr-2000`	325.6K	3.22M	2,716	18,235
`amazon-2008`	735.3K	5.16M	10	1,076
`dewiki-2013`	1.53M	36.72M	5,032	117,908
`ljournal-2008`	5.36M	79.02M	2,469	19,409
`enwiki-2018`	5.62M	128.8M	7,948	247,628
`indochina-2004`	7.41M	194.1M	6.985	256,425
`uk-2002`	18.52M	298.1M	2,450	194,942
`arabic-2005`	22.74M	640.0M	9,905	575,618
`uk-2005`	39.46M	936.4M	5,213	1,776,852
`twitter-2010`	41.65M	1.47B	2,997,469	770,155
`webbase-2001`	118.1M	1.02B	3,841	816,127
`clueweb12`	978.4M	42.57B	7,447	75,611,690

Table A.1: Properties of the directed graphs.

A.2 Undirected Graph Datasets

The summary of the undirected graphs is shown in Table A.2. Most of them are the symmetrized version of the directed graphs above, except for the two `hollywood`, `dblp`, and `orkut` datasets which are originally undirected. Here we just use abbreviated names when not ambiguous. The `orkut` dataset was retrieved from https:

//snap.stanford.edu/data/com-Orkut.html.

Name	$\lvert V\rvert$	$\lvert E\rvert$	d_{max}	d_{avg}
words	10,617	63,788	332	12.0
enron	69,244	254,449	1,634	7.35
uk-2007	100,000	2,779,575	55,252	55.59
cnr	325,557	2,738,969	18,236	16.83
amazon	735,323	3,523,472	1,077	9.58
dblp-2011	986,324	3,353,618	979	6.80
hollywood09	1,139,905	113,891,327	11,468	199.83
dewiki	1,532,354	33,093,029	118,246	43.19
hollywood11	2,180,759	228,985,632	13,107	210.00
orkut-2007	3,072,626	234,370,166	33,313	152.55
ljournal	5,363,260	49,514,271	19,432	18.46
enwiki18	5,616,717	234,488,590	248,444	83.7
indochina	7,414,866	301,969,638	256,425	82.0
uk-2002	18,520,486	261,787,258	194,955	28.27
arabic	22,744,080	553,903,073	575,628	48.71
uk-2005	39,459,925	783,027,125	1,776,858	39.69
twitter	41,652,230	1,202,513,046	2,997,487	57.74
webbase	118,142,155	854,809,761	816,127	14.47
clueweb	978,408,098	37,372,179,311	75,611,690	76.39

Table A.2: Properties of the undirected graphs. Note that $d_{\text{avg}} = 2|E|/|V|$.

Appendix B

Algorithms for 5-node Graphlet Enumeration

Below are the algorithms for S5GE, a solution to enumerate 3, 4, and 5-node graphlets simultaneously. Here, we denote a wedge by $\angle(a, b, c)$, where b is the center node and $a < c$. If $b < a$ we have a type 1 wedge, if $b > a$ we have a type 2 wedge. We can attach a subscript when we want to make the type obvious. Suppose we have $u < v$ and $u < w$, then $\angle_1(v, u, w)$ is a type 1 wedge, and $\angle_2(u, v, w)$ is a type 2 wedge. We denote a triangle by $\Delta(a, b, c)$, where $a < b < c$.

Algorithm 21 S5GE

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** $(u' > u) \wedge (v' > u)$ **then**
6:                     **if** $u' = v' > v$ **then**
7:                         ExtendedExploreTriangle (u, v, u')
8:                     **if** $((u' < v') \vee (v' = u)) \wedge (u' > v)$ **then**
9:                         ExtendedExploreWedgeType1 (v, u, u')
10:                     **if** $(u' > v') \wedge (v' \neq u)$ **then**
11:                         ExtendedExploreWedgeType2 (u, v, v')

Algorithm 22 EXTENDED EXPLORE TRIANGLE

Input: A triangle $\Delta(u, v, w)$, $u < v < w$, $N(u)$, $N(v)$, $N(w)$.

1: $N_1(u), N_1(v), N_1(w), N_2(u, v), N_2(u, w), N_2(v, w), N_3(u, v, w) \leftarrow \emptyset$
2: **for all** $z \in N(u) \cap N(v) \cap N(w)$ **do**
3: $\quad N_3(u, v, w) \leftarrow N_3(u, v, w) \cup \{z\}$
4: $\quad$ **if** $z > w$ **then**
5: $\quad\quad$ ENUMERATE4CLIQUE $(u, v, w, z)_8$
6: **for all** $z \in N(u) \cap N(v)$ and $z \notin N(w)$ **do**
7: $\quad N_2(u, v) \leftarrow N_2(u, v) \cup \{z\}$
8: $\quad$ **if** $z > w$ **then**
9: $\quad\quad$ ENUMERATEDIAMOND $(.)_7$
10: **for all** $z \in N(u) \cap N(w)$ and $z \notin N(v)$ **do**
11: $\quad N_2(u, w) \leftarrow N_2(u, w) \cup \{z\}$
12: $\quad$ **if** $z > v$ **then**
13: $\quad\quad$ ENUMERATEDIAMOND $(.)_7$
14: **for all** $z \in N(v) \cap N(w)$ and $z \notin N(u)$ **do**
15: $\quad N_2(v, w) \leftarrow N_2(v, w) \cup \{z\}$
16: $\quad$ **if** $z > u$ **then**
17: $\quad\quad$ ENUMERATEDIAMOND $(.)_7$
18: **for all** $z \in N(u)$ only **do**
19: $\quad$ ENUMERATETAILEDTRIANGLE $(.)_6$
20: $\quad N_1(u) \leftarrow N_1(u) \cup \{z\}$
21: **for all** $z \in N(v)$ only **do**
22: $\quad$ ENUMERATETAILEDTRIANGLE $(.)_6$
23: $\quad N_1(v) \leftarrow N_1(v) \cup \{z\}$
24: **for all** $z \in N(w)$ only **do**
25: $\quad$ ENUMERATETAILEDTRIANGLE $(.)_6$
26: $\quad N_1(w) \leftarrow N_1(w) \cup \{z\}$
27: Call 5GT functions

Algorithm 23 EXTENDED EXPLORE WEDGE TYPE-1

Input: A type-1 wedge $\angle_1(v, u, w)$, $u < v < w$, $N(u)$, $N(v)$, $N(w)$.

1: $N_1(u), N_1(v), N_1(w), N_2(u, v), N_2(u, w), N_2(v, w) \leftarrow \emptyset$
2: **for all** $z \in N(v) \cap N(w)$ with $z \notin N(u)$ **do**
3: $\quad N_2(v, w) \leftarrow N_2(v, w) \cup \{z\}$
4: $\quad$ **if** $z > u$ **then**
5: $\quad\quad$ ENUMERATERECTANGLE $(u, v, z, w)_5$
6: **for all** $z \in N(u) \cap N(v)$ with $z \notin N(w)$ **do**
7: $\quad N_2(u, v) \leftarrow N_2(u, v) \cup \{z\}$
8: **for all** $z \in N(u) \cap N(w)$ with $z \notin N(v)$ **do**
9: $\quad N_2(u, w) \leftarrow N_2(u, w) \cup \{z\}$
10: **for all** $z \in N(u)$ only **do**
11: $\quad N_1(u) \leftarrow N_1(u) \cup \{z\}$
12: $\quad$ **if** $z > u$ and $z > w$ **then**
13: $\quad\quad$ ENUMERATE3STAR $(u, v, w, z)_4$
14: **for all** $z \in N(v)$ only **do**
15: $\quad N_1(v) \leftarrow N_1(v) \cup \{z\}$
16: $\quad$ **if** $z > u$ **then**
17: $\quad\quad$ ENUMERATE3PATH $(w, u, v, z)_3$
18: **for all** $z \in N(w)$ only **do**
19: $\quad N_1(w) \leftarrow N_1(w) \cup \{z\}$
20: $\quad$ **if** $z > u$ **then**
21: $\quad\quad$ ENUMERATE3PATH $(v, u, w, z)_3$
22: Call 5GW1 functions

Algorithm 24 EXTENDED EXPLORE WEDGE TYPE-2

Input: A type-2 wedge $\angle_2(u, v, w)$, $u < v$, $u < w$, $N(u)$, $N(v)$, $N(w)$.

1: $N_1(u), N_1(v), N_1(w), N_2(u, v), N_2(u, w), N_2(v, w) \leftarrow \emptyset$
2: **for all** $z \in N(v) \cap N(w)$ with $z \notin N(u)$ **do**
3: $\quad N_2(v, w) \leftarrow N_2(v, w) \cup \{z\}$
4: **for all** $z \in N(u) \cap N(v)$ with $z \notin N(w)$ **do**
5: $\quad N_2(u, v) \leftarrow N_2(u, v) \cup \{z\}$
6: **for all** $z \in N(u) \cap N(w)$ with $z \notin N(v)$ **do**
7: $\quad N_2(u, w) \leftarrow N_2(u, w) \cup \{z\}$
8: **for all** $z \in N(u)$ only **do**
9: $\quad N_1(u) \leftarrow N_1(u) \cup \{z\}$
10: **for all** $z \in N^{>u}(v)$ only **do**
11: $\quad N_1(v) \leftarrow N_1(v) \cup \{z\}$
12: $\quad$ **if** $z > u$ and $z > w$ **then**
13: $\quad\quad$ ENUMERATE3STAR $(v, u, w, z)_4$
14: **for all** $z \in N^{>u}(w)$ only **do**
15: $\quad N_1(w) \leftarrow N_1(w) \cup \{z\}$
16: $\quad$ **if** $z > u$ and $z \neq v$ **then**
17: $\quad\quad$ ENUMERATE3PATH $(u, v, w, z)_3$
18: Call 5GW2 functions

Algorithm 25 5GT-UU

Input: $\Delta(u, v, w)$, with $u < v < w$, and $N_1(u), N_1(v), N_1(w)$.

1: **for all** $z_1, z_2 \in N_1(u)$, $z_1 < z_2$ **do**
2:     **if** $z_2 \in N(z_1)$ **then**
3:         **if** $z_1 > v$ **then**
4:             ENUMERATE_G18(.)
5:     **else**
6:         ENUMERATE_G14(.)
7: **for all** $z_1, z_2 \in N_1(v)$, $z_1 < z_2$ **do**
8:     **if** $z_2 \in N(z_1)$ **then**
9:         **if** $z_1 > u$ **then**
10:             ENUMERATE_G18(.)
11:     **else**
12:         ENUMERATE_G14(.)
13: **for all** $z_1, z_2 \in N_1(w)$, $z_1 < z_2$ **do**
14:     **if** $z_2 \in N(z_1)$ **then**
15:         **if** $z_1 > u$ **then**
16:             ENUMERATE_G18(.)
17:     **else**
18:         ENUMERATE_G14(.)

Algorithm 26 5GT-UV

Input: $\Delta(u, v, w)$, with $u < v < w$, and $N_1(u), N_1(v), N_1(w)$.

1: **for all** $z_1 \in N_1(u)$, $z_2 \in N_1(v)$ **do**
2:     **if** $z_2 \in N(z_1)$ **then**
3:         ENUMERATE_G21(.)
4:     **else**
5:         ENUMERATE_G12(.)
6: **for all** $z_1 \in N_1(u)$, $z_2 \in N_1(w)$ **do**
7:     **if** $z_2 \in N(z_1)$ **then**
8:         ENUMERATE_G21(.)
9:     **else**
10:         ENUMERATE_G12(.)
11: **for all** $z_1 \in N_1(v)$, $z_2 \in N_1(w)$ **do**
12:     **if** $z_2 \in N(z_1)$ **then**
13:         ENUMERATE_G21(.)
14:     **else**
15:         ENUMERATE_G12(.)

Algorithm 27 5GT-U2UV

```
Input: Δ(u,v,w), u < v < w, N1(u), N1(v), N1(w), N2(u,v), N2(u,w), N2(v,w).
 1: for all z1 ∈ N1(u), z2 ∈ N2(u,v) do
 2:     if z2 ∉ N(z1) then
 3:         if z2 > w then
 4:             ENUMERATE_G17(.)
 5: for all z1 ∈ N1(u), z2 ∈ N2(u,w) do
 6:     if z2 ∉ N(z1) then
 7:         if z2 > v then
 8:             ENUMERATE_G17(.)
 9: for all z1 ∈ N1(v), z2 ∈ N2(u,v) do
10:     if z2 ∉ N(z1) then
11:         if z2 > w then
12:             ENUMERATE_G17(.)
13: for all z1 ∈ N1(v), z2 ∈ N2(v,w) do
14:     if z2 ∉ N(z1) then
15:         if z2 > u then
16:             ENUMERATE_G17(.)
17: for all z1 ∈ N1(w), z2 ∈ N2(u,w) do
18:     if z2 ∉ N(z1) then
19:         if z2 > v then
20:             ENUMERATE_G17(.)
21: for all z1 ∈ N1(w), z2 ∈ N2(v,w) do
22:     if z2 ∉ N(z1) then
23:         if z2 > u then
24:             ENUMERATE_G17(.)
```

Algorithm 28 5GT-u2vw

Input: $\Delta(u, v, w)$, $u < v < w$, $N_1(u), N_1(v), N_1(w), N_2(u, v), N_2(u, w), N_2(v, w)$.

```
1: for all z_1 ∈ N_1(u), z_2 ∈ N_2(v,w) do
2:     if z_2 ∈ N(z_1) then
3:         if z_2 > u then
4:             Enumerate_g25(.)
5:     else
6:         Enumerate_g19(.)
7: for all z_1 ∈ N_1(v), z_2 ∈ N_2(u,w) do
8:     if z_2 ∈ N(z_1) then
9:         if z_2 > v then
10:            Enumerate_g25(.)
11:    else
12:        Enumerate_g19(.)
13: for all z_1 ∈ N_1(w), z_2 ∈ N_2(u,v) do
14:    if z_2 ∈ N(z_1) then
15:        if z_2 > w then
16:            Enumerate_g25(.)
17:    else
18:        Enumerate_g19(.)
```

Algorithm 29 5GT-2UV2UV

```
Input: Δ(u,v,w), u < v < w, N_2(u,v), N_2(u,w), N_2(v,w).
 1: for all z_1, z_2 ∈ N_2(u,v), z_1 < z_2 do
 2:     if z_1 > w then
 3:         if z_2 ∈ N(z_1) then
 4:             ENUMERATE_G26(.)
 5:         else
 6:             ENUMERATE_G22(.)
 7: for all z_1, z_2 ∈ N_2(u,w), z_1 < z_2 do
 8:     if z_2 ∈ N(z_1) then
 9:         if z_1 > w then
10:             ENUMERATE_G26(.)
11:     else
12:         if z_1 > v then
13:             ENUMERATE_G22(.)
14: for all z_1, z_2 ∈ N_2(v,w), z_1 < z_2 do
15:     if z_2 ∈ N(z_1) then
16:         if z_1 > w then
17:             ENUMERATE_G26(.)
18:     else
19:         if z_1 > u then
20:             ENUMERATE_G22(.)
```

Algorithm 30 5GT-2UV2UW

Input: $\Delta(u, v, w)$, $u < v < w$, $N_2(u, v), N_2(u, w), N_2(v, w)$.

```
1: for all z1 ∈ N2(u,v), z2 ∈ N2(u,w) do
2:     if z2 ∈ N(z1) then
3:         if z1 > w then
4:             ENUMERATE_G27(.)
5:     else
6:         ENUMERATE_G24(.)
7: for all z1 ∈ N2(u,v), z2 ∈ N2(v,w) do
8:     if z2 ∈ N(z1) then
9:         if z1 > w then
10:            ENUMERATE_G27(.)
11:    else
12:        ENUMERATE_G24(.)
13: for all z1 ∈ N2(u,w), z2 ∈ N2(v,w) do
14:    if z2 ∈ N(z1) then
15:        if z1 > v then
16:            ENUMERATE_G27(.)
17:    else
18:        ENUMERATE_G24(.)
```

Algorithm 31 5GT-U3UVW

Input: $\Delta(u, v, w)$, $u < v < w$, $N_1(u), N_1(v), N_1(w), N_3(u, v, w)$.

```
1:  for all z1 ∈ N1(u), z2 ∈ N3(u, v, w) do
2:      if z2 ∈ N(z1) then
3:          if z2 > v then                                  ▷ Allow z2 < w
4:              Enumerate_g26(.)
5:      else
6:          Enumerate_g23(.)
7:  for all z1 ∈ N1(v), z2 ∈ N3(u, v, w) do
8:      if z2 ∈ N(z1) then
9:          if z2 > v then                                  ▷ Allow z2 < w
10:             Enumerate_g26(.)
11:     else
12:         Enumerate_g23(.)
13: for all z1 ∈ N1(w), z2 ∈ N3(u, v, w) do
14:     if z2 ∈ N(z1) then
15:         if z2 > w then                                  ▷ To avoid double counting
16:             Enumerate_g26(.)
17:     else
18:         Enumerate_g23(.)
```

Algorithm 32 5GT-2UV3UVW

Input: $\Delta(u, v, w)$, $u < v < w$, $N_2(u, v), N_2(u, w), N_2(v, w), N_3(u, v, w)$.

```
1:  for all z1 ∈ N2(u, v), z2 ∈ N3(u, v, w) do
2:      if z2 ∉ N(z1) then
3:          if z1 > w and z2 > w then
4:              Enumerate_g26(.)
5:  for all z1 ∈ N2(u, w), z2 ∈ N3(u, v, w) do
6:      if z2 ∉ N(z1) then
7:          if z1 > w and z2 > w then
8:              Enumerate_g26(.)
9:  for all z1 ∈ N2(v, w), z2 ∈ N3(u, v, w) do
10:     if z2 ∉ N(z1) then
11:         if z1 > w and z2 > w then
12:             Enumerate_g26(.)
```

Algorithm 33 5GT-3UVW3UVW

Input: $\Delta(u, v, w)$, $u < v < w$, $N_3(u, v, w)$.

1: **for all** $z_1, z_2 \in N_3(u, v, w), z_1 < z_2$ **do**
2:     **if** $z_2 \in N(z_1)$ **then**
3:         **if** $z_1 > w$ **then**
4:             ENUMERATE_G29(.)
5:     **else**
6:         ENUMERATE_G28(.)

Algorithm 34 5GW1-UU

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_1(u)$.

1: **for all** $z_1, z_2 \in N_1(u)$, $z_1 < z_2$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         **if** $z_1 > w$ **then**
4:             ENUMERATE_G11(.)

Algorithm 35 5GW1-UV

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_1(u)$, $N_1(v)$, $N_1(w)$.

1: **for all** $z_1 \in N_1(u)$, $z_2 \in N_1(v)$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         **if** $z_1 > w$ **then**
4:             ENUMERATE_G10(.)
5: **for all** $z_1 \in N_1(u)$, $z_2 \in N_1(w)$ **do**
6:     **if** $z_2 \notin N(z_1)$ **then**
7:         **if** $z_1 > v$ **then**
8:             ENUMERATE_G10(.)

Algorithm 36 5GW1-VW

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_1(v), N_1(w)$.

1: **for all** $z_1 \in N_1(v)$, $z_2 \in N_1(w)$ **do**
2:     **if** $z_2 \in N(z_1)$ **then**
3:         **if** $z_2 > u$ and $z_1 > u$ **then**
4:             ENUMERATE_G15(.)
5:     **else**
6:         ENUMERATE_G9(.)

Algorithm 37 5GW1-U2VW

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_1(u)$, $N_2(v, w)$.

1: **for all** $z_1 \in N_1(u)$, $z_2 \in N_2(v, w)$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         ENUMERATE_G16(.)

Algorithm 38 5GW1-v2uw

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_1(v)$, $N_1(w)$, $N_2(u, w)$, $N_2(u, v)$.

```
1: for all z1 ∈ N1(v), z2 ∈ N2(u,w) do
2:     if z2 ∉ N(z1) then
3:         if z2 > w then
4:             Enumerate_g13(.)
5: for all z1 ∈ N1(w), z2 ∈ N2(u,v) do
6:     if z2 ∉ N(z1) then
7:         if z2 > v then
8:             Enumerate_g13(.)
```

Algorithm 39 5GW1-2vw2vw

Input: $\angle_1(v, u, w)$, with $u < v < w$, and $N_2(v, w)$.

```
1: for all z1, z2 ∈ N2(v,w), z1 < z2 do
2:     if z2 ∉ N(z1) then
3:         if z1 > u and z2 > u then
4:             Enumerate_g20(.)
```

Algorithm 40 5GW2-vv

Input: $\angle_2(u, v, w)$, with $u < v$, and $u < w$; $N_1(v)$.

```
1: for all z1, z2 ∈ N1(v), z1 < z2 do
2:     if z2 ∉ N(z1) then
3:         if z1 > w then
4:             Enumerate_g11(.)
```

Algorithm 41 5GW2-uv

Input: $\angle_2(u, v, w)$, with $u < v$, and $u < w$; $N_1(u)$, $N_1(v)$, $N_1(w)$.

```
1: for all z1 ∈ N1(v), z2 ∈ N1(u) do
2:     if z2 ∉ N(z1) then
3:         if z1 > w then
4:             Enumerate_g10(.)
5: for all z1 ∈ N1(v), z2 ∈ N1(w) do
6:     if z2 ∉ N(z1) then
7:         if z1 > u then
8:             Enumerate_g10(.)
```

Algorithm 42 5GW2-uw

Input: $\angle_2(u, v, w)$, with $u < v$, and $u < w$; $N_1(u), N_1(w)$.

```
1: for all z1 ∈ N1(u), z2 ∈ N1(w) do
2:     if z2 ∉ N(z1) then
3:         Enumerate_g9(.)
```

Algorithm 43 5GW2-V2UW

Input: $\angle_2(u,v,w)$, with $u < v$, and $u < w$; $N_1(v)$, $N_2(u,w)$.

1: **for all** $z_1 \in N_1(v)$, $z_2 \in N_2(u,w)$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         ENUMERATE_G16(.)

Algorithm 44 5GW2-U2VW

Input: $\angle_2(u,v,w)$, with $u < v$, and $u < w$; $N_1(u)$, $N_1(w)$, $N_2(v,w)$, $N_2(u,v)$.

1: **for all** $z_1 \in N_1(u)$, $z_2 \in N_2(v,w)$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         **if** $z_2 > w$ **then**
4:             ENUMERATE_G13(.)
5: **for all** $z_1 \in N_1(w)$, $z_2 \in N_2(u,v)$ **do**
6:     **if** $z_2 \notin N(z_1)$ **then**
7:         **if** $z_2 > u$ **then**
8:             ENUMERATE_G13(.)

Algorithm 45 5GW2-2UW2UW

Input: $\angle_{2a}(u,v,w)$, with $u < v$, and $u < w$; $N_2(u,w)$.

1: **for all** $z_1, z_2 \in N_2(u,w), z_1 < z_2$ **do**
2:     **if** $z_2 \notin N(z_1)$ **then**
3:         **if** $z_1 > v$ and $z_2 > v$ **then**
4:             ENUMERATE_G20(.)

Appendix C

Directed 4-node Graphlets

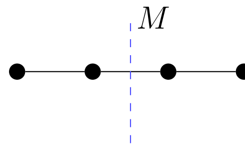
In this appendix, we compute the number of types of directed 4-node graphlets. We use Burnside's Lemma (see for example https://en.wikipedia.org/wiki/Burnside's_lemma) to derive our results. It states that the number of orbits (or distinct objects) is equal to

$$|X/G| = \frac{1}{|G|}\sum_{g}|X^g| \tag{C.1}$$

where X is the set of all possible configurations, and G is the symmetry group. Here, X^g is the subset of X that is left invariant by the operation $g \in G$.

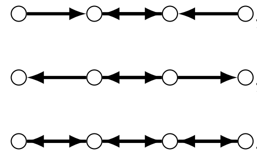
C.1 Directed 3-path

For the three path (g_3)



aside from the identity or the null symmetry, there is only one symmetry, which is the mirror symmetry (M) with respect to the dashed line above. Thus, we have $G =$

$\{\text{Id}, M\}$, so $|G| = 2$. With 3 links and 3 possible link types, we have $|X| = 3^3 = 27$. Counting the invariant configurations, we have $\left|X^{\text{Id}}\right| = 27$ and $\left|X^{M}\right| = 3$. The three configurations that are invariant under M are

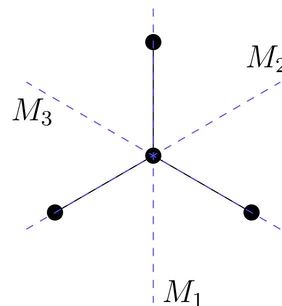


Thus, for the 3-path, the number of non-isomorphic directed graphs is

$$|X/G| = \frac{1}{2}(27 + 3) = 15$$

C.2 Directed 3-star

For the 3-star (g_4)



the symmetry group is $G = \{\text{Id}, R_{120}, R_{240}, M_1, M_2, M3\}$ where R_x is rotation over x degrees, and M_x is mirror operation that exchanges the two nodes other than the x node. Thus, $|G| = 6$. With 3 links, we have $|X| = 3^3 = 27$. We have,

$\left|X^{\text{Id}}\right| = 27$

$\left|X^{R_{120}}\right| = \left|X^{R_{240}}\right| = 3$

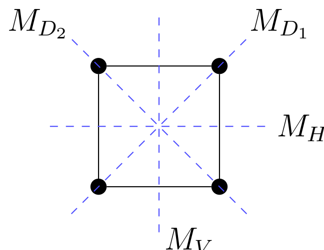
$\left|X^{M_1}\right| = \left|X^{M_2}\right| = \left|X^{M_3}\right| = 9.$

Thus, for 3-start,

$$|X/G| = \frac{1}{6}(27 + 3 + 3 + 9 + 9 + 9) = 10$$

C.3 Directed 4-cycle

For 4-cycle (g_5)



we have $|X| = 3^4 = 81$. The symmetry group is

$G = \{\text{Id}, R_{90}, R_{180}, R_{270}, M_V, M_H, M_{D_1}, M_{D_2}\}$, where M_V (M_H) is the mirror symmetry with respect to the vertical (horizontal) axis, and M_{D_1} and M_{D_2} are the mirror symmetry with respect to the two diagonal axes. We can check that there is no other symmetry by making sure that G satisfies the closure property. Thus, $|G| = 8$. We have the following:

$\left|X^{\text{Id}}\right| = 81$

$\left|X^{R_{90}}\right| = \left|X^{R_{270}}\right| = 3$

$\left|X^{R_{180}}\right| = 9$

$\left|X^{M_V}\right| = \left|X^{M_H}\right| = 3$

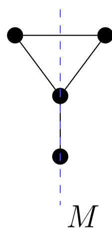
$\left|X^{M_{D_1}}\right| = \left|X^{M_{D_2}}\right| = 9$

Thus, for 4-cycle,

$$|X/G| = \frac{1}{8}\left(81 + 3 + 3 + 9 + 3 + 3 + 9 + 9\right) = 15$$

C.4 Directed tailed-triangle

Attaching a tail to a triangle breaks the triangle symmetry, and we are left with only one mirror symmetry, M, as shown below

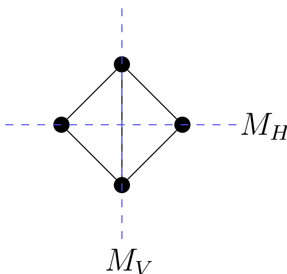


So, for tailed-triangle (g_6) the symmetry group is $G = \{\text{Id}, M\}$. hence $|G| = 2$. We have $|X| = 3^4 = 81$, $\left|X^{\text{Id}}\right| = 81$, and $\left|X^M\right| = 9$. Thus, for a tailed triangle,

$$|X/G| = \frac{1}{2}(81 + 9) = 45$$

C.5 Directed diamond

For diamond (g_7)



there are two mirror symmetries, which we denote by M_V and M_H. We also have one rotational symmetry over 180 degrees. Thus, $G = \{\text{Id}, R_{180}, M_V, M_H\}$, hence $|G| = 4$. We have $|X| = 3^5 = 243$, and

$\left|X^{\text{Id}}\right| = 243$,

$\left|X^{R_{180}}\right| = 9$,

$\left|X^{M_H}\right| = 9$,

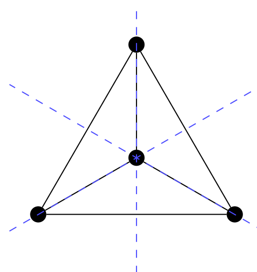
$\left|X^{M_V}\right| = 27$,

Thus, for diamonds, we have

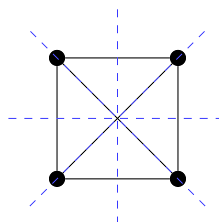
$$|X/G| = \frac{1}{4}(243 + 9 + 9 + 27) = 72$$

C.6 Directed 4-clique

For 4-clique (g_8)



it is no longer easy to see all of the symmetries geometrically, at least not with only one picture. This graphlet can also be drawn differently, as



which indicates different symmetries (i.e., double transpositions and four-cycles). The symmetry group is the permutation group of four objects, S_4 (https://en.wikiversity.org/wiki/Symmetric_group_S4).

There are 24 members of this group: the identity, eight 3-cycles, six transpositions, three double transpositions, and six 4-cycles. There are 6 links in a clique, leads to $|X| = 3^6 = 729 = \left|X^{\mathrm{Id}}\right|$.

For each of the transpositions (swapping a pair of nodes) $|X^g| = 27$,

for each of the double transpositions $|X^g| = 9$,

for each of the 3-cycles $|X^g| = 9$, and

for each of the 4-cycles $|X^g| = 3$.

Thus, for 4-clique, we have

$$|X/G| = \frac{1}{24}\left(729 + 6 * 27 + 3 * 9 + 8 * 9 + 6 * 3\right) = 42$$