8-22-2018

Improvements on ORCA for Fast Computation of Graphlet Degree Vectors of any Graphlet Order

Wilberto Z. Nunez
wzn2144@rit.edu

Improvements on ORCA for Fast Computation of Graphlet Degree Vectors of any Graphlet Order

by

Wilberto Z. Nunez

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Applied and Computational Mathematics
School of Mathematical Sciences, College of Science

Rochester Institute of Technology
Rochester, NY

August 22, 2018

Committee Approval:

Dr. Carlos R. Rivero — Date

Thomas B. Golisano College of

Computing and Information Sciences

Thesis Advisor

Dr. Darren Narayan — Date

School of Mathematical Sciences

Committee Member

Dr. Jobby Jacob — Date

School of Mathematical Sciences

Committee Member

Dr. Matthew Hoffman — Date

School of Mathematical Sciences

Director, Applied and Computational

Mathematics MS Program

Abstract

It is increasingly common to find real-life structures or behaviors represented as graphs in many areas of the computing sciences. Comparing these graphs is a hard task, especially when we are interested in assigning a non-binary similarity score between two large graphs based on some domain-specific context. In bioinformatics, social network analysis and other areas is frequently necessary to compute graph similarities based on the local topological information of each vertex of the given graphs. This is why graphlet degree vectors have become more and more popular in these areas. They provide a simple yet detailed representation of a vertex's topology by counting the number of times such vertex touches a list of small predefined sub-structures called graphlets. In this thesis, we study the state-of-the-art algorithm to compute graphlet degree vectors, the Orbit Counting Algorithm (ORCA). ORCA generates a triangular system of linear equations that can be quickly solved to obtain the graphlet degree vector of a vertex. We make theoretical and practical improvements to this algorithm and measure the difference in speed after these improvements. The theoretical improvement consists of finding automorphisms of graphlets given a fixed vertex that is required to map to itself in such automorphisms. We observe that one piece of the algorithm runs much faster than before with this improvement, especially for larger graphlet orders. This helps the algorithm take less time in generating the linear system that we use to find the desired graphlet degree vector. The practical improvement consists of making a flexible implementation of the algorithm, which can take any graphlet size as input, any number of input graphs, and compute the graphlet degree vector for every vertex in each one of those graphs.

Contents

Chapter 1

Introduction

1.1. Overview

Computing and analyzing topological similarities between graphs is an important task in various fields of the computing sciences, from machine learning to protein-protein interaction to social network analysis and other areas [1] [2] [3]. Graph alignment, as it is called in some literature, is often performed by comparing properties of each vertex in one graph against every vertex in a second graph, then assigning a similarity score to each pair of vertices and finally finding a stable matching between the two vertex sets. It is simple to show that two graphs are different by simply listing a few properties in which they differ, but showing that they are similar can be difficult, especially when they are non-isomorphic and/or when they are both very large, in which case it is required to demonstrate their similarity in their numerous topological properties. Exact subgraph matching techniques are not effective in this type of problem, since most of the time there is not an exact match of one graph into the other, but some approximate matching techniques could do the job.

Pržulj [4] came up with the concept of graphlet degrees and graphlet degree vector, which we explore in Section 1.2. The concept has been used in recent years in different algorithms for computing graph similarity between biological networks [5] [6] [7]. Computing graphlet degree vectors is an interesting theoretical problem in itself, and it is convenient to analyze it using some combinatorial insights. In this thesis we focus on a particular approach of determining graphlet degree vectors, called Orbit Counting Algorithm (ORCA) [8] [9], we make improvements to it and run different experiments to compare results against the original algorithm. In this chapter

we cover the basic definitions and notations required to understand our work, Chapter 2 goes over the different algorithms that we have re-used or modified, as well as some related work, and Chapter 3 presents our contribution in detail. The final chapter discusses our conclusions and future work.

1.2. Definitions

In this section we define some of the basic graph theory concepts used throughout this thesis. A graph G is a set of vertices or nodes connected by links called edges. The vertex set of G is denoted by $V(G)$ and the edge set by $E(G)$. The *degree* $d(v)$ of a vertex v is the number of edges that are incident to v in the graph. Two vertices are said to be *adjacent* if they have an edge connecting them. The *neighborhood* of a vertex v, denoted $N(v)$, is the set of vertices that are adjacent to v. Given a set of vertices $S \subseteq V(G)$, the *common neighbors* of S is the subset of vertices in $V(G)$, not contained in S, which are adjacent to every vertex in S. We define below some classic graph theory problems that are also used throughout this thesis.

1.2.1 The graph isomorphism problem

Graph isomorphism is an equivalence relationship between two graphs. Two graphs G_1 and G_2 are isomorphic ($G_1 \cong G_2$) if there exists a bijection $f : V(G_1) \to V(G_2)$ such that $(u, v) \in E(G_1) \iff (f(u), f(v)) \in E(G_2)$.

An **automorphism** of a graph G is a graph isomorphism with itself, that is, a bijection from $V(G)$ back to $V(G)$ such that the resulting graph is isomorphic to G. The equivalence classes of the vertices of a graph under the action of all possible automorphisms are called automorphism orbits or just **orbits**, hence two vertices belong to the same orbit if they map to each other in some automorphism.

A graph isomorphism problem is not as hard as NP-complete, but in general the problem cannot be solved in polynomial time [10]. Some researchers have defined a separate complexity class, GI-complete/GI-hard, to study this problem and other problems that can be reduced to graph isomorphism in polynomial time.

The general case of the graph isomorphism problem, in theory, had been solved the fastest

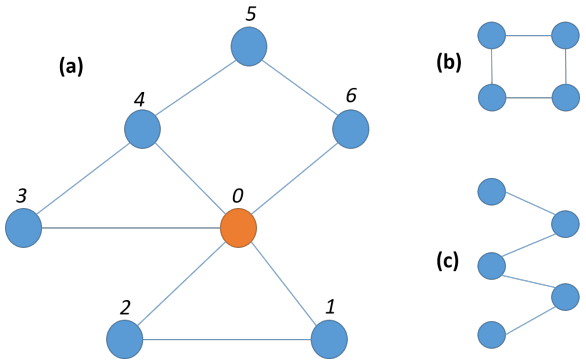

Figure 1.1: Graphs (b) and (c) are both subgraphs of (a), but only (b) is also an *induced* subgraph

by the algorithm in Babai and Luks [11] for over three decades. Their paper presented an algorithm to detect graph isomorphism with a time complexity bound of $2^{O(\sqrt{n \log n})}$, where n is the number of vertices in one of the given graphs (the largest graph, if they are different). A recent improvement on this upper bound was recently published by Babai [12] who proposes an algorithm that can solve the problem in quasi-polynomial time: $2^{O((\log n)^c)}$ for some $c > 0$.

There are currently other practical approaches that generally perform better on random graphs, such as the algorithm designed (and later improved) by McKay [13]. This algorithm is implemented in the widely used Nauty tools [14] created by the same author.

1.2.2 Subgraph matching

Graph isomorphism is a special case of the **subgraph isomorphism** problem. Given a graph H, a **subgraph** H' is composed by a subset of vertices $V(H') \subseteq V(H)$, and a subset of edges $E(H') \subseteq E(H) \cap (V(H') \times V(H'))$. The subgraph isomorphism problem is a computational task where, given two graphs G and H, one must determine whether there is a subgraph H' in H, such that $H' \cong G$. In other words, one must find an injective mapping $m : V(G) \to V(H)$, such that $(u, v) \in E(G) \Rightarrow (m(u), m(v)) \in E(H)$. As an example, if we look at graph (a) in Figure 1.1 (borrowed from [5]), we can find a subgraph that is isomorphic to graph (b), formed by vertices 0, 4, 5 and 6 along with the edges between them. We can also find several subgraphs isomorphic to graph (c), which is a P_5, the path on five vertices, for example: $(2) - (1) - (0) - (6) - (5)$, and $(3) - (4) - (5) - (6) - (0)$. While subgraph isomorphism is a decision problem (with true/false output), the name **subgraph matching** is commonly used when the task involves finding all or some possible mappings m.

A similar problem is **induced subgraph matching**, where we find one or more injective mappings $m : V(G) \to V(H)$, such that $(u, v) \in E(G) \iff (m(u), m(v)) \in E(H)$. Notice there is a biconditional here, implying that if u and v are not adjacent in G, they must be non-adjacent in H. If a mapping exists, G is said to be an **induced subgraph** of H, as opposed to a non-induced subgraph like in the regular subgraph matching problem defined above. An induced subgraph is also a non-induced subgraph, but not the other way around. Graph (b) in Figure 1.1 is both an induced and non-induced subgraph of graph (a), using the same vertices and edges we found before. However, we cannot find in (a) an induced subgraph isomorphic to (c). There are several P_4's induced in graph (a), e.g. $(5) - (6) - (0) - (2)$; $(5) - (4) - (0) - (1)$. However, $(3) - (4) - (0) - (1)$ is not an induced P_4 because there is an edge between vertex 0 and vertex 3.

In both induced and non-induced subgraph matching problems, the "smaller" graph G is sometimes called a query graph, while the "larger" graph H is called the host graph. The name **graphlet** is sometimes used to refer to small connected simple query graphs with at least two vertices in induced subgraph matching problems. We use $\mathcal{G}_k$ to denote the set of all non-isomorphic simple connected graphs (graphlets) on k vertices and $\mathcal{G}^k$ to denote the set of all graphlets from 2 to k vertices. Thus, $\mathcal{G}^k = \mathcal{G}_2 \cup \mathcal{G}_3 \cup ... \cup \mathcal{G}_k$.

Some of the most widely used algorithms for subgraph matching are VF2 [15], Turbo-ISO [16] and the subgraph matching procedure presented in [17]. We use the latter in this thesis with slight adaptations borrowed from Turbo-ISO (Section 3.3).

1.2.3 Orbit counts

Pržulj [4] presented an ordered list of all 30 possible graphlets of $k \leq 5$ vertices (i.e. $\mathcal{G}^5$), and labeled all 73 orbits in these graphlets as shown in Figure 1.2. The same paper introduced the concept of graphlet degrees, also simply called orbit counts in other literature after it [18] [8] [9]. An *orbit count*, under this context, does not refer to the number of orbits in a graph. Given a host graph H and an orbit O_i in some orbit enumeration (like the enumeration in Figure 1.2), such that orbit O_i is contained in some graphlet G_a, define **orbit count** $o_i(v)$ as the number of times a vertex $v \in V(H)$ appears in orbit O_i of an induced subgraph of H that is isomorphic to G_a. Kuchaiev points out in [5] that it is topologically relevant to distinguish between, for example, vertices touching a graphlet G_{10} at one end from vertices touching G_{10} in the middle, which is why we count each orbit of

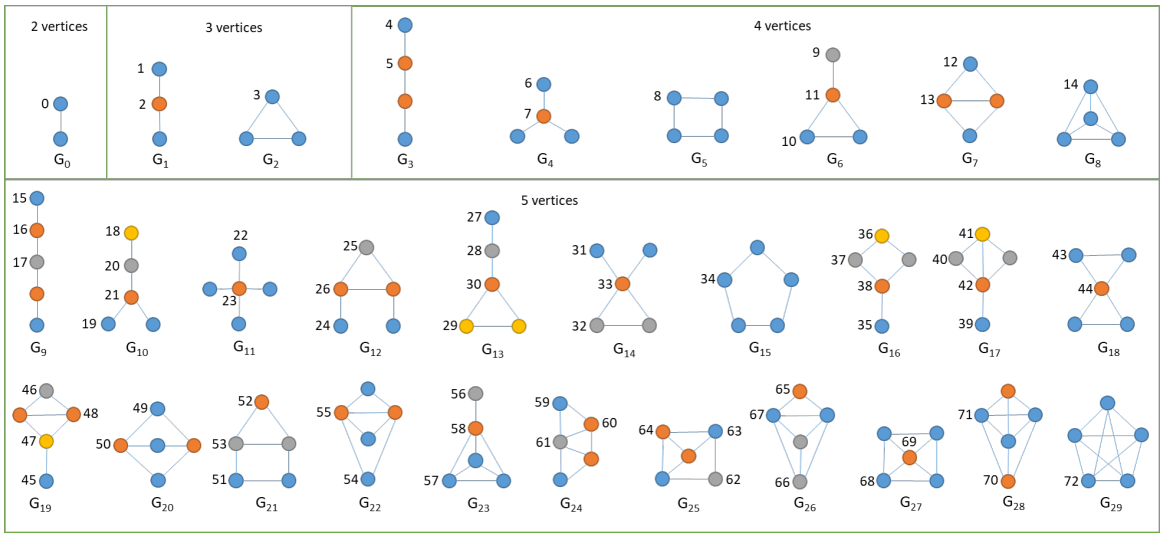

Figure 1.2: Graphlets of up to five nodes and their enumerated orbits (Figure adapted from [4])

Orbit index	0	1	2	3	4	5	6	7	8	9	10	11	12...20	21
Orbit count	5	2	8	2	0	5	0	4	1	0	1	6	0...0	2
Orbit index	22...25	26	27...29	30	31	32	33	34...37	38	39...43	44	45...52	53	54...72
Orbit count	0...0	2	0...0	2	0	0	4	0...0	2	0...0	1	0...0	1	0...0

Table 1.1: Signature vector of vertex 0 from Fig. 1.1

the same graphlet separately. Since there are 73 orbits in Pržulj's enumeration, we can build a 73-component vector for each v, where each component of the vector is an orbit count for v. This is called the *graphlet degree vector*, or the signature vector of the vertex. Note that the degree of a vertex is the first element in this vector, since the number of times a vertex touches orbit 0 (from Figure 1.2) would be equivalent to the number of edges incident to that vertex. Table 1.1 illustrates the graphlet degree vector, computed in [5], for vertex 0 in the sample graph from Figure 1.1. The 73-component graphlet degree vector is a helpful tool in graph alignment algorithms [5] [7] [6], since it tells a lot about a vertex's local topological properties. These algorithms can compare vectors from vertices in one graph against vectors from vertices in another graph to measure the similarity between the two, either by computing a simple Euclidean distance or a weighted average of the components or any other similarity function. It is assummed by the aforementioned literature that vertices with similar graphlet degree vectors will be similar in other important observed properties.

Pržulj's enumeration contains 73 orbits from 30 distinct graphlets. The total number of or-

bits in an enumeration increases dramatically as k increases. There are 30 graphlets and 73 orbits in $\mathcal{G}^5$, whereas in $\mathcal{G}^6$ we have 142 graphlets with 480 orbits. In $\mathcal{G}^7$ we have 995 graphlets with 4,787 orbits. Computing graphlet degree vectors for any k is a notoriously difficult task that requires the search for new counting methods and graph theoretical insights.

Chapter 2

Background and related work

2.1. Subgraph matching algorithm

Subgraph matching is an NP-complete problem and a very expensive operation in terms of time complexity. The first step in many subgraph matching algorithms such as [17] is selecting and pruning the set of *feasible mates*, also called the *feasible set* of every vertex of the query graph in the host graph. When we are searching for subgraphs of a graph H that are isomorphic to some query graph G, the feasible set $\Phi(u) \subseteq V(H)$ of every $u \in V(G)$ is the set of vertices $v \in V(H)$ that "could be" mapped to u in some subgraph isomorphism. After that, we compute the *search space*, which is defined as the product of feasible sets of the query graph: $\Phi(u_1) \times ... \times \Phi(u_k)$, where k is the number of vertices in such query graph. Not every feasible mate will result in or be part of a valid mapping of G in H because we need to retrieve those feasible sets as quickly as possible and let the rest of the algorithm discard some of the invalid vertex combinations in the search space. As an example, let us assume our query graph is G_6 from Figure 1.2 (sometimes called the "paw" graph or the 3-pan) and we are doing subgraph matching on some large host graph H. The feasible set of the orange vertex will contain all vertices of H with a degree that is greater than or equal to 3. No vertex in H with a degree of 2 or less can possibly be mapped to this orange vertex in a subgraph isomorphism. Using vertex degrees is a quick and simple way to obtain the feasible set for each query graph vertex, but should not be the only one if we are dealing with large and/or dense host graphs. The feasible set of the gray vertex, for instance, which has degree of 1, would contain all vertices in H with this approach.

```
input  : Query graph G, Host graph H
output : All valid mappings m : V(G) → V(H)
for u ∈ V(G) do
    Φ(u) ← {v ∈ V(H), v is feasible};
    Local pruning of Φ(u);
end
Reduce Φ(u_1) × ... × Φ(u_k) globally;
Optimize search order of {u_1, ..., u_k};
Search(1);

void Search(i) begin
    k ← |V(G)|;
    for v ∈ Φ(u_i), v is free do
        if i < k and CanBeMapped(u_i, v) then
            m(u_i) ← v;
            Search(i + 1);
            m(u_i) ← null;
        end
        if Size(m) = k then
            Report mapping m;
        end
    end
end

boolean CanBeMapped(u_i, v) begin
    for edge (u_i, u_j) ∈ E(G) do
        if edge (v, m(u_j)) ∉ E(H) then
            return false;
        end
    end
    return true;
end
```

Algorithm 1: Subgraph Matching

The subgraph matching procedure from [17] is outlined in Algorithm 1 for reference. The original

paper describes several techniques to trim the feasible sets as much as possible without sacrificing much performance, but these techniques only apply to attributed graphs, where every vertex in the query graph and the host graph has attributes or labels that must match in the subgraph matching process.

After selecting the feasible set of each query graph vertex, the next step in the algorithm is to reduce the overall search space $\Phi(u_1) \times ... \times \Phi(u_k)$ by discarding combinations that are obviously invalid without much computational effort. This is a key concept we explore in Section 3.3.

The rest of the algorithm (the *Search* subroutine) basically tries all the combinations remaining in the search space after pruning, looking for valid mappings of G onto H. The *CanBeMapped* subroutine receives a query vertex u_i and a host graph vertex v and checks whether every neighbor of v that is already mapped is actually mapped to a neighbor of u_i.

2.2. ORCA

The Orbit Counting Algorithm (ORCA) introduced in [8] and further detailed in [9] is the state-of-the-art approach for computing graphlet degree vectors. The authors found that each orbit count (i.e. each component of the vector) for a particular vertex in a host graph can be computed from other orbit counts belonging to graphlets with more edges. They present an algorithm that builds a triangular linear system relating nearly all of the orbit counts for any orbit enumeration, a system that can quickly be solved by performing back substitution. Only those orbits that belong to cliques must be counted explicitly. In order to fully understand the algorithm, one must become familiar with the notation in table 2.1 and some of the concepts we explain below. Later in this thesis we present some improvements on ORCA and its implementation. The goal of the algorithm is to create a linear equation relating one orbit to other orbits in the same enumeration. Given an orbit O_i, the algorithm will output a linear equation of the form

$$\sum_p f_p o_p = r,$$

where each o_p is the yet unknown orbit count for orbit p, and f_p is a coefficient that relates orbit O_i to orbit O_p. The number r on the right-hand side is computed by searching for induced subgraphs in the host graph H; we will discuss how to calculate r further down this section. The algorithm has to be run for every orbit in the enumeration to obtain a system of equations and, by solving this system we get the values of every orbit count and hence the desired graphlet degree vector.

Notation	Definition
H	Host graph within which we count graphlets and orbits
$N(v)$	Neighborhood of a vertex v
$c(v_1, v_2, ..., v_j)$	Number of common neighbors of vertices $v_1, v_2, ..., v_j$
$c(S)$	Number of common neighbors of a set of vertices S
$\mathcal{G}_k$	Set of all graphlets on k vertices
$\mathcal{G}^k$	Set of all graphlets on 2 to k vertices
G_a	Graphlet a according to some enumeration.
O_i	Orbit i according to some enumeration
$\mathcal{O}$	List of all orbits in a given enumeration
$o_i(v)$	Number of times vertex v appears in orbit i of an induced subgraph
G^H	An induced subgraph of H that is isomorphic to G
S^H	The set of vertices in $V(H)$ that are mapped to $S \subseteq V(G)$ in an induced subgraph isomorphism
$\{v_1, v_2, ..., v_j\}^H$	The set of vertices in $V(H)$ that are mapped to $v_1, v_2, ..., v_j \in V(G)$ in an induced subgraph isomorphism

Table 2.1: Notations used in [9] and more

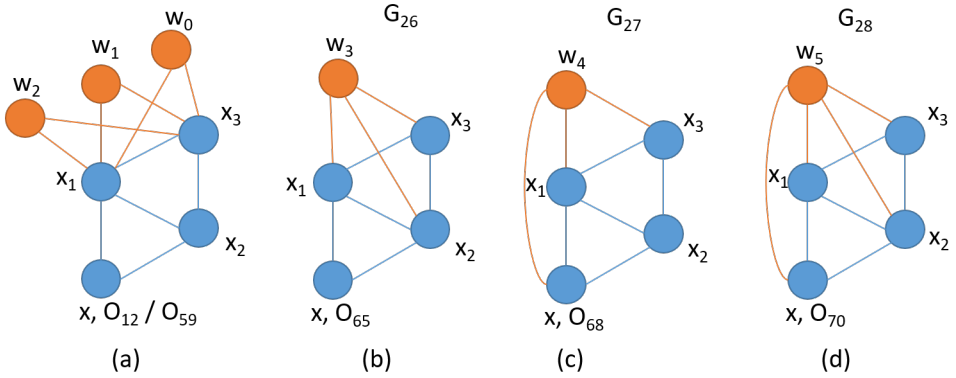

Figure 2.1: The number common neighbors of x_1 and x_3 has a relationship with the number of times x appears in orbits 59, 65, 68 and 70. The same can be done with x_2 and x_3 because of symmetry.

As an example, the linear equation found by ORCA for orbit 59 (see Figure 2.1 (a)) is as follows:

$$o_{59} + 4o_{65} + 2o_{68} + 6o_{70} = \sum_{G_7^H, x \in O_{12}} [(c(x_1, x_3) - 1) + (c(x_2, x_3) - 1)]. \tag{2.2.1}$$

The general reasoning behind the algorithm is that, by finding occurrences of a given graphlet on $k-1$ vertices in the host graph, and by counting the number of common neighbors of specific vertex sets in this graphlet, we obtain all the occurrences of other larger graphlets (on k vertices). We will focus on deriving the example above and then generalize it while explaining the algorithm.

Let us take a look at the graph in Figure 2.1 (a). In this setup, the blue vertices and edges form a G_7, which is a graphlet on 4 vertices. We can also see several instances of G_{24}, a graphlet on 5 vertices, if we combine the G_7 with vertex w_0, w_1, or w_2. Vertex x appears in orbit 12 (from G_7), and appears in 59 three times in this graph. Notice the number of times x appears in orbit 59 is equal to the number of common neighbors of x_1 and x_3 minus 1 (because we do not want to count x_2), that is, the number of common neighbors outside the 4-vertex graphlet. Thus, $o_{59}(x) = c(x_1, x_3) - c(\{x_1, x_3\}^{G_7}) = c(x_1, x_3) - 1$.

Now, let us assume this graph is actually a subgraph of a larger graph H, which may have more edges with other vertices not shown here, as well as extra edges between the vertices shown. In this scenario, some common neighbors of x_1 and x_3 may also be neighbors of x_2, like shown in Figure 2.1 (b), in which case vertex x would be touching orbit 65 (from G_{26}) and not orbit 59. Some common neighbors of x_1 and x_3 could also be neighbors of x itself (Fig. 2.1 (c)), or

they could be neighbors of both x and x_2 (Fig. 2.1 (d)), making x appear in orbits 68 and 70 respectively. Therefore, in such graph H, taking all these combinations into account, we would relate the orbit counts of x by saying that $o_{59}(x) + o_{65}(x) + o_{68}(x) + o_{70}(x) = c(x_1, x_3) - 1$. However, we would still be missing some counts. Notice that G_7 has symmetry and we can switch the roles of vertices x_1 and x_2 while keeping x in the same orbit. Thus, the counts we found with the common neighbors of x_1 and x_3 can also be found with the common neighbors of x_2 and x_3. Each graphlet G_{26} (Fig. 2.1 (b)) appearing in the larger graph is counted 4 times with roles of x_1, x_2, x_3 and w_3 swapped because of symmetry. When we consider the symmetries that cause counting the same graphlet multiple times with different roles of its vertices, we get $o_{59}(x) + 4o_{65}(x) + 2o_{68}(x) + 6o_{70}(x) = [c(x_1, x_3) - 1] + [c(x_2, x_3) - 1]$.. Finally, we must sum this over all occurrences of G_7 touching vertex x in orbit 12, hence obtaining Equation 2.2.1.

This process can be done with other 4-vertex graphlets to obtain equations relating orbits on 5-vertex graphlets, or with 3-vertex graphlets to obtain equations on 4-vertex graphlets. The final result is a triangular system of linear equations that must be solved to obtained the graphlet degree vector of x.

In the algorithm, for each orbit O_i in some graphlet G_a, we pick a vertex x as a fixed point. Having picked x, we select a vertex $y \neq x$ such that $G' = G_a \backslash y$ is still a connected graph. We call G' a *mutilated* graphlet. Vertex y may or may not be in the same orbit as x, but the authors impose other constraints on y to achieve an efficient computation of the linear equation:

1. Vertex y is the lowest degree vertex at a longest distance L_x from x,
2. $G' = G_a \backslash y$ is connected,
3. $d(y) \leq k - 2$,
4. if $d(y) = k - 2$, the neighbors of y induce a connected graph,
5. if (2), (3) or (4) cannot be true for the lowest degree vertex at a distance L_x from x, pick the lowest degree vertex at a distance $L_x - 1$ and ignore conditions (3) and (4). This only happens in exceptional cases such as the cycle of four vertices.

Now that we know how to select our vertex y, we can find the extension sets, an important concept in this algorithm and in our thesis:

Definition 1. *Given an orbit O_i in a graphlet G_a, a vertex x in orbit O_i, a selected vertex $y \in V(G_a)$, and $G' = G_a \backslash y$, an **extension set** $S \subset V(G')$ is a set of vertices such that adding a new vertex connected to*

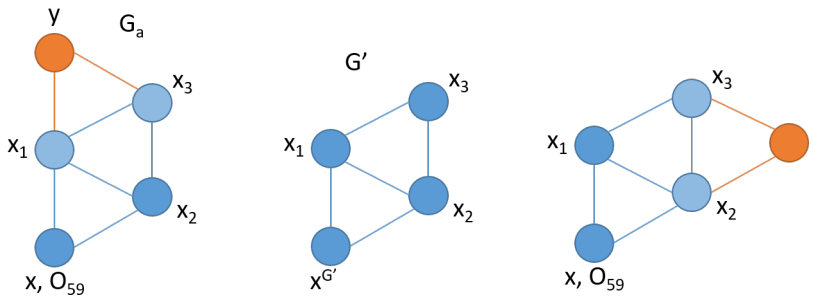

Figure 2.2: Extension sets for orbit 59: $\{x_1, x_3\}$ (the neighborhood of y), and $\{x_2, x_3\}$

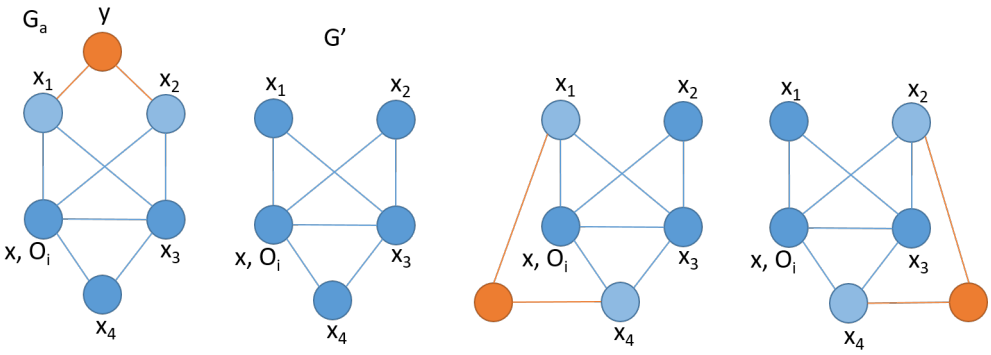

Figure 2.3: Extension sets for an orbit on a 6-vertex graphlet: $\{x_1, x_4\}$, $\{x_2, x_4\}$, and $N(y)$ itself, $\{x_1, x_2\}$

all vertices in S yields G_a with x in orbit O_i [9].

The number of extension sets is then equivalent to the number of ways in which G' can be extended to G_a. The set of all such sets for a particular orbit is denoted by $\mathcal{S}$. The example in Figure 2.2 illustrates this concept for orbit 59. After picking vertex y such that G' is still a connected graph, we find those sets that would give us back G_a (which is G_{24} in this example) by adding back y and connecting it to such sets. Not only we need to get G_a back, but also we need vertex x to remain in the same orbit, O_{59}. The extension sets for this example are $\{x_1, x_3\}$ (the original neighbors of y), and $\{x_2, x_3\}$. Other sets like $\{x, x_1\}$ and $\{x, x_2\}$ would give us back G_a, but our vertex x would end up in a different orbit, O_{60}. Both papers [8] and [9] focus on this example to illustrate the algorithm and all the concepts behind it. See Figure 2.3 for a different example, using a graphlet on 6 vertices. In this second example, there are three extension sets for the selected vertex y: one is the neighborhood of y itself, $\{x_1, x_2\}$, and the other two are $\{x_1, x_4\}$ and $\{x_2, x_4\}$. It can be manually verified that the two graphs on the right are isomorphic to G_a while maintaining x in the same position.

The right-hand side of each equation is calculated with a nested summation over all occurrences of G' in H as follows:

$$r_i = \sum_{G'^H : x^H \in O_i} \left\{ \sum_{S \in \mathcal{S}} (c(S^H) - c(S^{G'})) \right\}. \tag{2.2.2}$$

Recall from Table 2.1 that G'^H represents an induced subgraph of H isomorphic to G'. It is important to note that each one of these G' induced in H must have different vertex sets. Under this condition, if G' were, for instance, a path on three vertices and H was the sample graph in Figure 1.1 (a), the subgraph $(3) - (4) - (5)$ would be the same as $(5) - (4) - (3)$. Current subgraph matching algorithms do not take this into account, so for every induced subgraph found, we must check for potential duplicates or create a mechanism to avoid duplicates in the first place. We discuss this further in Chapter 3.

Result: Equation for O_i
$G_a \leftarrow$ Graphlet containing O_i;
$x \in O_i$;
$y \leftarrow SelectY(x)$;
$G' \leftarrow G_a \backslash y$;
$\mathcal{S} \leftarrow$ Extension sets for G';
$RHS \leftarrow r_i$ /* Equation 2.2.2 */;
for $p \in \mathcal{O}$ **do**
  $G^* \leftarrow$ Graphlet containing O_p;
  $f_p \leftarrow 0$;
  for $z \in G^* : (G^* \backslash z) \cong G'$ **do**
    $f_p \leftarrow f_p + |\{S \in \mathcal{S} : N(z) \supseteq S\}|$
  end
end
$LHS \leftarrow \sum_p f_p o_p$;
return $LHS = RHS$;

Algorithm 2: Orbit Counting Algorithm (ORCA) for deriving equation of orbit O_i

In Algorithm 2 there is a segregation of the computation of the left-hand side from the computation of the right-hand side. The coefficients on the left-hand side reflect the relationships between graphlet orbits and do not depend on the host graph, hence they can be derived in advance for the whole matrix. The same matrix can be reused for analyzing multiple input host graphs, whereas the right-hand side requires the search for induced subgraphs for all graphlets of up to $k-1$ vertices (G') in each host graph. It may seem like finding all the induced subgraphs of up to $k-1$

Set	Graphlets	Orbits
$\mathcal{G}^4$ $(2 \leq k \leq 4)$	9	15
$\mathcal{G}_5$ $(k = 5)$	21	58
$\mathcal{G}^5$ $(2 \leq k \leq 5)$	30	73
$\mathcal{G}_6$ $(k = 6)$	112	407
$\mathcal{G}^6$ $(2 \leq k \leq 6)$	142	480
$\mathcal{G}_7$ $(k = 7)$	853	4,306

Table 2.2: Total number of graphlets and orbits by graphlet order k

vertices is an expensive task, but in reality, there are many more graphlets of order k alone than there are graphlets of order 2 to $k-1$ put together (for any k), and therefore many many more *orbits* for those k-order graphlets. See table 2.2 with some examples for reference.

2.3. Nauty

Nauty (**n**o **aut**omorphism, **y**es?) is a set of procedures that, among other things, can quickly compute the automorphism group of a vertex-labeled graph [14]. A vertex-labeled graph in this context is a graph G together with a bijection $l : V(G) \to M$, where M is a set of numeric labels. While Nauty computes the automorphism group of a graph, it also finds its orbits, a feature that comes in handy for us.

In this thesis, Nauty's libraries are used for two important purposes: (1) generating all graphlets of a given order k, and (2) computing their orbits. For the former, we use the convenient `geng` utility that can be run from a Unix command line, and for the latter we use the core `nauty` function that computes automorphism groups and orbits. The tricky part of using these libraries and utilities is that there is no way to fully predict how the graphlets and orbits will be generated, so the resulting orbit enumeration will have a different order every time we run the application. We explain why this is a disadvantage and how to handle it in Section 3.5.

Chapter 3

Improvements on ORCA

In this thesis we make theoretical and practical optimizations to ORCA. The main theoretical improvement is the computation of extension sets using fixed-point automorphisms, which we describe in Section 3.1. This change helps speed up the computation of the matrix on the left-hand side of the triangular linear system. In addition, from a practical standpoint, we have automated the generation of the linear system for any graphlet size k, and compiled the implementation in a single program that takes only two arguments: k, and the path to a directory containing all the host graphs to be analyzed. The script will output several files showing all the graphlet degree vectors for every vertex in every graph provided, one file per graph and one line per vertex. This is a relevant practical improvement over the original implementation, for which the authors manually computed the system of equations for $k \leq 5$, which contains 73 orbits (hence, 73 equations), and proceeded to hard-code the coefficients directly into the code. Sections 3.2 and 3.3 describe two relevant optimizations we have made to the original subgraph matching algorithm presented in [17]. The latter is based on the concept of *candidate regions* proposed in [16]. The purpose of these two improvements is to quicken the search for graphlets in the host graph, the most time consuming part of computing the right-hand side of the linear system. The architectural details are described in Section 3.4.

This chapter and the ones that follow use the same notations and terms defined in Section 2.2.

3.1. Combinatorial improvement

Hočevar and Demšar define an extension set $S \subset V(G')$ as stated in Definition 1. For a computer program, finding all such sets implies testing all possible vertex subsets of a certain size in G', then adding one vertex to G', connecting that vertex to all $v \in S$, and then testing for isomorphism against G_a. This is a computationally heavy approach, especially because we must do it for every orbit O_i and the number of orbits increases exponentially with k. Thus, we redefine extension sets in this thesis as follows:

Definition 2. *Let m_x be an automorphism of G' where vertex x maps to itself. Thus, an **extension set** $S \subset V(G')$ is a set of vertices $v \in V(G')$ such that for every $u \in N(y)$, $m_x(u) = v$.*

Essentially, for every such automorphism m_x we would have a valid extension set S, which under this definition is simply the mapping of $N(y)$ under m_x. One of such sets is $N(y)$ itself (the trivial automorphism case), which counts as one of the extension sets. We call m_x a fixed-point automorphism.

Looking at Figure 2.2, for instance, the possible automorphisms on G' are:

1. $(x, x), (x_1, x_1), (x_2, x_2), (x_3, x_3)$ - (the trivial case),
2. $(x, x), (x_1, x_2), (x_2, x_1), (x_3, x_3)$,
3. $(x, x_3), (x_1, x_1), (x_2, x_2), (x_3, x)$,
4. $(x, x_3), (x_1, x_2), (x_2, x_1), (x_3, x)$.

Of these, only the first two have x mapped to itself. Since x_3 is the selected vertex y, we get the sets which $N(x_3)$ maps to according to automorphism (1) and (2). Clearly, the trivial case requires no computational effort. Also, the algorithm in [17] that we re-adapted to find these automorphisms, allows us to "fix" a vertex before starting the mapping search, hence we can avoid considering invalid mappings like (3) and (4) where x does not map to itself. The algorithm will only search then for potential mappings of the remaining vertices, which means that if our G' has $k-1$ vertices, we only search for mappings of $k-2$ vertices, whereas in the original definition of extension sets it is required to find an isomorphism against G_a (k vertices) for a graph composed of G' and an extra vertex. This is already a significant advantage, but in addition, let us remember that there are many more graphlets on k vertices than there are graphlets on less than k vertices all put together.

3.2. Obtaining distinct subgraphs in induced subgraph matching

To the best of our knowledge, there is no standard algorithm that is able to find subgraphs with different vertex sets in a given host graph without having to eliminate duplicate mappings after they have already been found. Recall the example of finding a P_3 in the host graph of Figure 1.1 (a). Most subgraph matching algorithms, induced or non-induced, would include $(3)-(4)-(5)$ and $(5)-(4)-(3)$ in their output as two distinct subgraphs (and they are distinct, for some applications). However, in our case we need subgraphs with different vertex sets as required by ORCA, even if these sets have some common vertices, so we will call *distinct subgraphs* to those subgraphs that do not have identical vertex sets. We could simply filter non-distinct subgraphs after the subgraph matching process is done, but a more efficient approach would avoid considering non-distinct matches before they are even found. In this thesis, we accomplish this by taking advantage of the orbit information we have on every graphlet, as well as the fact that these graphlets are generated by Nauty with distinct numeric labels on their vertices. The host graphs are also numerically labeled in our implementation.

Our technique is as follows: Let u and v be two vertices in a graphlet or a mutilated graphlet G'. In the traditional definition of the problem, we are trying to find all induced subgraphs in H that are isomorphic to G', so $(u,v) \in E(G') \iff (m(u), m(v)) \in E(H)$, where $m : V(G') \to V(H)$ is an injective mapping. Since we have numerically labeled graphlets and host graphs, as well as the orbit information of the graphlets, we add one more condition to our induced subgraph matching procedure: let $l(w)$ denote the label of any vertex w; thus, if u and v are in the same orbit of G' and $l(u) > l(v)$, then $l(m(u)) > l(m(v))$. With this added condition, we make sure that when we perform induced subgraph matching of G' in H, we only get the mappings that have the same label ordering as that of G'. The result of this is that no two matching subgraphs will have the same set of vertices.

3.3. Using candidate regions to cut search spaces in fixed-point subgraph matching

In Section 2.1, we explained how the first step in most subgraph matching algorithms, induced or non-induced, is to select the feasible set in the host graph for each vertex in the query graph (i.e. the graphlet). Since we do not have attributed vertices in our graphlets or in our host graphs

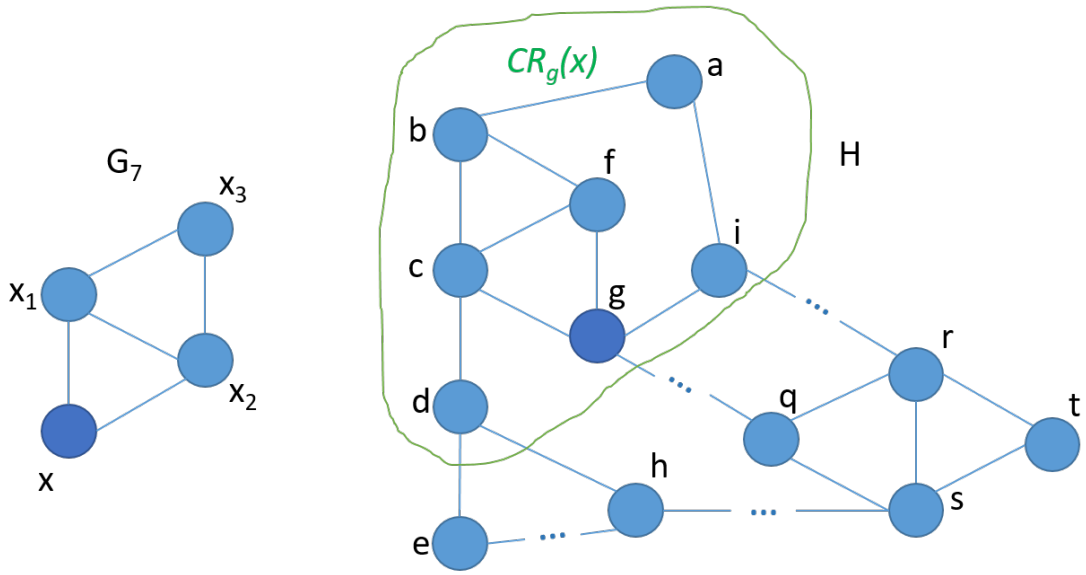

Figure 3.1: Candidate Region $CR_g(x)$ contains only those vertices in H at a distance from g that is less than or equal to the diameter of G_7

with labels that could help us restrict our search, we can only select the feasible sets by looking at the degree of all the vertices. For instance, take the graphlet in Figure 3.1 (which is a G_7). The feasible set for x_1 would initially be all the vertices in H which degree is greater than or equal to 3, whereas the feasible set for x_3 would contain all the vertices in H with a degree that is greater than or equal to 2. This is a quick approach, but it may result in very large feasible sets for both x_1 and x_3, hence a gigantic search space $\Phi(x) \times \Phi(x_1) \times \Phi(x_2) \times \Phi(x_3)$. It is imperative to quickly but significantly reduce this search space.

In TurboISO [16], the authors discuss the concept of *candidate regions*. The general idea is that, for some query vertex $u_1 \in V(G')$ and for each host graph vertex $v_j \in \Phi(u_1) \subseteq V(H)$, we can restrict the number of combinations in the search space $\Phi(u_1) \times ... \times \Phi(u_k)$ by finding candidate regions $CR_{v_j}(u_2) \subset \Phi(u_2), CR_{v_j}(u_3) \subset \Phi(u_3), ..., CR_{v_j}(u_k) \subset \Phi(u_k)$, which contain only those vertices close enough to v_j to form a subgraph with the same diameter as G'. Let us look at the example in Figure 3.1. Graphlet G_7 has a diameter (a longest "shortest" path) of 2. The feasible sets of vertices x and x_3 contain potentially every vertex in H. The feasible sets for x_1 and x_2 contain less vertices but are still quite large. Trying every combination of vertices in $\Phi(x) \times \Phi(x_1) \times \Phi(x_2) \times \Phi(x_3)$ will eventually give us all induced matches of G_7, but it would take too much time. Vertex g is in $\Phi(x)$ and is a good matching candidate for x, and vertex r is in $\Phi(x_1)$ and a good matching

candidate for x_1, but we do not want to try those combinations where x maps to g and x_1 maps to r. Therefore, when we "fix" vertex x to g, we must cut the feasible sets of x_1, x_2 and x_3 to consider only the candidate region $CR_g(x) = \{v \in V(H) : dist(g, v) \leq diam(G_7)\}$. This will immensely reduce the number of checks we do in the search space.

We can optimize this approach even further. Note that vertex x in G_7 is in orbit 12 from Pržulj's enumeration. The distance between x and any other vertex in G_7 is at most 2, the same as the diameter of G_7. If we are computing $o_{12}(g)$, i.e. the number of times that vertex g touches orbit 12, then it makes sense to select the candidate region as described above. However, if we were computing $o_{13}(g)$ (vertex x_1 is in orbit 13), then we can select an even more restrictive candidate region. The distance between x_1 and any other vertex of G_7 is at most 1, so when we fix x_1 to g, we can cut the feasible set of x, x_2 and x_3 to consider only the candidate region $CR_g(x_1) = \{v \in V(H) : dist(g, v) \leq L_{x_1}\}$, where L_{x_1} is the distance from x_1 to its furthest vertex within G_7, which is 1. We generalize the definition as follows:

Definition 3. *Given an orbit O_i in a graphlet G, some vertex $x \in O_i$, and a vertex w of a host graph H, let L_x be the distance from x to its furthest vertex in G, and let $dist(p, q)$ represent the distance between any two vertices p and q of any graph. A* ***candidate region*** *$CR_w(x) = \{v \in V(H) : dist(w, v) \leq L_x\}$. The computation of $o_i(w)$ must always involve an induced subgraph matching search that is restricted to the candidate region $CR_w(x)$.*

Besides significantly pruning search spaces, this definition offers another advantage. Recall from Section 2.2 that for every orbit O_i in the enumeration, we must select a vertex $y \neq x$ with certain conditions. The first condition requires us to find "the lowest degree vertex at a longest distance L_x from x", where x is a vertex in O_i. This means that for every orbit, we are already pre-computing L_x. In the implementation of this thesis we save these values and re-use them to select the corresponding candidate regions in every vertex of every host graph being analyzed.

3.4. Architectural improvements and implementation

The original implementation of ORCA is able to compute graphlet degree vectors up to $k = 4$ or $k = 5$. The user must pick one of the two and run the application to get an output. Their code is highly optimized for these specific values of k, and it contains all the coefficients from the 73 linear equations manually written there. This is a useful implementation for most practical purposes, but the design lacks flexibility, one of our concerns in this thesis.

In order to achieve higher flexibility, there are a few considerations to make. First, each input k value will give a different collection of graphlets and orbits, but for a particular enumerated collection, the matrix on the left-hand side will be the same for every host graph, hence this matrix can be computed in advance and reutilized. Secondly, and more importantly, the matrix is not the only object that can be reutilized for all host graphs. The mutilated graphlets, $G' = G_a \backslash y$, corresponding to each orbit O_i, have been obtained after carefully selecting a vertex y for O_i, and have to be used in the calculation of the right-hand side of each equation. Thus, it is worth using memory space to save a mapping from each orbit index i to its corresponding G'. This is helpful since, for every orbit, the algorithm piece for the right-hand side must search for subgraphs isomorphic to G' in H, so it is good to have quick access to the G' corresponding to O_i. Better yet, instead of keeping a mapping from every orbit index i to G', we could keep a mapping from i to orbit index j, where O_j is an orbit in graphlet $G_b \cong G'$ from the same graphlet enumeration as G_a. More specifically, O_j would be the orbit in G_b such that vertex $x \in O_i$ maps to a vertex in O_j for the isomorphism $G_b \cong G'$. For instance, orbits 59, 65, 68 and 70, correspond to graphlets G_{24}, G_{26}, G_{27}, and G_{28} respectively. These graphlets, after getting a vertex y removed, they all result in a G' that is isomorphic to G_7, as pointed out by [8] (see Figure 3.2). Vertex x in orbits 59, 65, 68 and 70, ends up in orbit 12 (from G_7) after removing y from those graphlets. Therefore, we can save the following mappings in a hash table: $\{(59, 12), (65, 12), (68, 12), (70, 12)\}$. There is a significant advantage in this approach: as we compute the right-hand side of the linear equation for O_{59}, the algorithm finds all subgraphs isomorphic to its G' in H, that is, all G_7's. Then, when we are computing the right-hand side of O_{65}, O_{68} and O_{70}, we do not need to search again in H for subgraphs isomorphic to their respective G', because we already found those subgraphs in a previous iteration. Finally, if we look at Equation 2.2.2, the term $c(E^{G'})$ is the number of common neighbors for a particular extension set in G'. These values are also independent of the host graph, so they can be computed in advance and re-utilized.

For a better illustration of our implementation of a *flexible* ORCA, see the high-level design of the application in Figure 3.3. The components shown in the diagram are:

1. `geng` wrapper. This is the entry point of the application. It receives k as an argument and uses Nauty's `geng` utility to compute all non-isomorphic, connected graphs (graphlets) from 2 to k vertices. The generated graphlets are in G6 format [14].

2. Orbit calculator. This is a C program that internally calls the main `nauty` function to compute the orbits of each graphlet received by the `geng` wrapper.

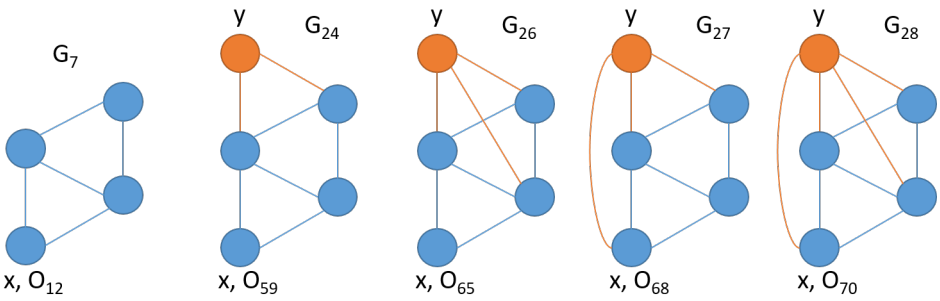

Figure 3.2: Several graphlets result in the same $G' = G_a \backslash y$ with x in the same orbit, e.g. G_{24}, G_{26}, G_{27} and G_{28} all become G_7 after removing y, with x in O_{12}

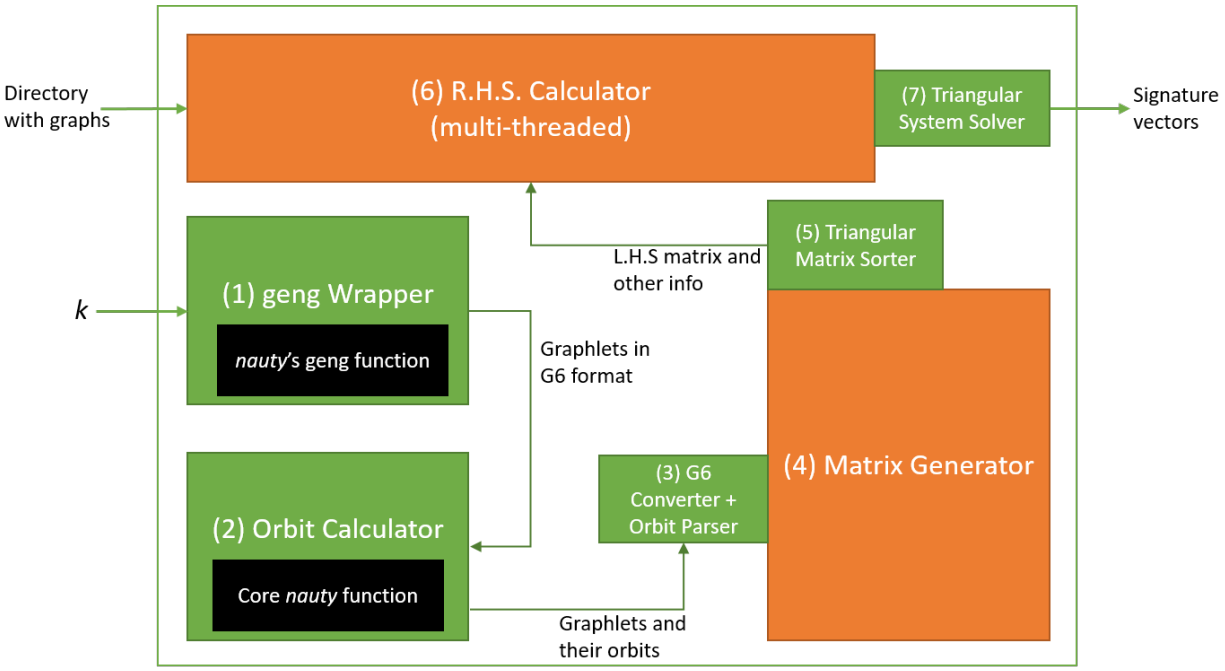

Figure 3.3: High-level architecture of our flexible ORCA implementation. Each box represents an application component, whereas each arrow represents data going in or out of each component. Orange boxes enclose the core functionalities of the orbit counting algorithm.

3. G6 Converter + Orbit Parser. This module processes the output from the orbit calculator, converting all graphlets from G6 format to JGraphT SimpleGraph objects [19]. JGraphT is a widely used Java library to handle graphs. Our converter module also saves all the orbits in plain old Java arrays.

4. Matrix Generator. This module takes all the graphlets and their orbits and computes the left-hand side of the linear system. It contains all the logic in Algorithm 2 that is related to the left-hand side, and it can use either one of the definitions of *extension sets* to do so.

5. Triangular Matrix Sorter. This module is necessary to arrange the rows and columns of the linear system and make it triangular. Read section 3.5 for more details on this module.

6. R.H.S. Calculator. This module computes the right hand side of the system for each input host graph. It reuses some information that is determined in advance by the matrix generator, such as the extension sets for each orbit O_i.

7. Triangular System Solver. Simple solver for triangular linear systems. The output of this module is written into one file per input graph. Each file contains the graphlet degree vector for every vertex in the input host graph.

3.5. Triangular matrix sorter

With or without our new definition of extension sets, ORCA will create a triangular matrix for several linear systems that need to be solved. However, the order in which Nauty generates graphs and helps us find orbits is somewhat unpredictable. Therefore, the orbit enumeration generated by `geng` and our Orbit Calculator will not have a specific order, causing the rows and columns of the resulting triangular matrix to be shuffled. The matrix will not look triangular, making it difficult to do back substitution and solve the system quickly. Since the same matrix will be used to solve one linear system for each vertex in every host graph, it would certainly be valuable to sort this matrix before completing and solving the systems for all the vertices.

Sorting a shuffled triangular matrix is generally a simple task (we could try sorting by the number of non-zero entries), but in this scenario, our matrix is large and highly sparse. Many rows and columns have the same number of non-zero entries. For this reason, we have included a special matrix sorting procedure for large sparse matrices.

Order	$\mathcal{E}$ orig. timing	$\mathcal{E}$ new. timing	Matrix orig. timing	Matrix new timing
$k \leq 5$	63.49	33.72	1196.62	1209.44
$k \leq 6$	331.91	198.46	14570.76	14106.49
$k \leq 7$	4028.81	680.65	1329381.00	1307006.25

Table 3.1: Elapsed time for ORCA using original definition of extension sets vs. using fixed-point automorphisms. All times are in milliseconds.

3.6. Experiments

Our first set of experiments are focused on measuring the impact of the combinatorial improvements described in Section 3.1. We look at the elapsed time required for computing all extension sets using fixed-point automorphisms and compare that to the time it takes when we use the original definition in [9]. We also observe how this improvement in computing extension sets affects the total elapsed time of generating the matrix. Our tests were ran using different values of k in a 4-thread AMD Opteron 4226 CPU, with 8 GB of RAM on Linux. Table 3.1 summarizes the final results.

As we can see in Table 3.1, the improvement of computing extension sets using fixed-point automorphisms is significant, and even more so for larger graphlet orders. However, the influence of this improvement on the total time it takes to generate the matrix does not seem as large, even though it also grows with higher values of k. We think this is because most of the computation effort in the algorithm is spent after the extension sets have been determined and we start comparing all orbits against each other (see the `for` loop in Algorithm 2).

Host Graph	No. of Vertices	Flexible ORCA	Iterative Orbit Count
backbones_1DS3.grf	733	55661.64	49391.62
backbones_1BDV.grf	878	65952.61	64025.71
backbones_1D7E.grf	1673	135150.80	201278.45
backbones_140L.grf	2609	213529.37	511534.59
backbones_1AUJ.grf	3254	280887.00	784767.57
backbones_1AF7.grf	4445	367612.13	1459601.30
backbones_1BES.grf	4581	379104.79	1582779.36
backbones_1APM.grf	5607	450131.45	2368644.35
backbones_1DLO.grf	9113	856498.67	6387231.02

Table 3.2: Elapsed time for computation of graphlet degree vectors on different protein graphs. All times are in milliseconds.

Result: Graphlet degree vector of every $v \in V(H)$
$o_i(v) \leftarrow 0$ (for all $O_i \in \mathcal{O}$ and for all $v \in V(H)$);
$\mathcal{G}^k \leftarrow$ List of graphlets;
for $G_a \in \mathcal{G}^k$ **do**
  for *All distinct* G_a^H **do**
    for $v \in G_a^H$ **do**
      $i \leftarrow$ Index of orbit of v in G_a^H;
      $o_i(v) \leftarrow o_i(v) + 1$;
    end
  end
end
return $\{o_i(v) | O_i \in \mathcal{O}, v \in V(H)\}$;

Algorithm 3: Simple iterative orbit counter

Our second set of experiments consists of comparing the computation of graphlet degree vectors using our improved ORCA versus a simple iterative orbit counter like the one shown in Algorithm 3. For the input host graphs we have used a subset of the proteins dataset. The graphlet order we used for this test was $k \leq 6$ and both algorithms printed out exactly the same graphlet degree vectors for the given host graphs. The performance results can be found in table 3.2. We can see that for smaller host graphs, the iterative orbit counter seems to do slightly better, but for the larger graphs, ORCA performs better by an order of magnitude.

Chapter 4

Conclusions

In this thesis we have studied graphlet degree vectors, which are composed of orbit counts of a vertex given a particular orbit enumeration. We focused on improving the state-of-the-art algorithm for computing graphlet degree vectors, the Orbit Counting Algorithm (ORCA). This algorithm creates a triangular linear system for each vertex in a host graph, and the system can be easily solved to obtain the desired graphlet degree vector. Both theoretical and practical improvements have been made to the original implementation presented by authors Hočevar and Demšar, by (1) re-defining the concept of extension sets by orbit and (2) providing a flexible architecture able to compute graphlet degree vectors for any given graphlet order and any number of input host graphs. The re-definition of extension sets helps us speed up the generation of the matrix for the aforementioned linear system. We observed a significant improvement in the time it takes to compute extension sets using our new definition, but the impact on the total time to generate the matrix was not as high as expected. We do see that for graphlets of higher order, the computation of extension sets has more influence on the total matrix generation time, so in those cases our new definition has more relevance.

We tested our program on a set of protein graphs and compared its performance against a simple iterative algorithm. We observed that our version of ORCA runs faster on larger graphs, demonstrating scalability.

Future work

As future work, we aim to improve memory usage of the application to be able to process larger matrices. For graphlets in $\mathcal{G}^8$, the application would need to handle 77,275 orbits from 12,112 graphlets, which implies having to solve systems of 77,275 equations and unknowns, and managing all the objects that we save in memory for re-use and better speed (G', mappings of G' onto H, common neighbor counts, etc.). We also want to study the nested loop in ORCA in order to further optimize the algorithm. We already know that any orbit O_i is only related to an orbit O_p if the graphlet containing O_p has at least as many edges as the graphlet containing O_i. This is the reason why the resulting linear system is triangular. In this loop we skip those orbits from graphlets of lesser edges, but we suspect that there are other iterations that can be skipped, given the very high sparsity of the resulting matrix (most of its entries are zero).

Acknowledgements

I would like to thank Dr. Paul Wenger and Dr. Jobby Jacob for inspiring me to switch to the Discrete Mathematics concentration of the master's program. Their teachings were crucial for me to understand the fundamentals of Graph Theory, Combinatorics, as well as mathematical reasoning in general. Thank you and Dr. Darren Narayan also for accepting my invitation to be in my thesis committee.

I would like to express my gratitude to Dr. Stanisław Radziszowski (Staszek) for his guidance and support in helping me understand Nauty during my independent study on Ramsey Theory. I enjoyed every assignment of that course.

Finally, I am especially grateful to my advisor Dr. Carlos R. Rivero, for his guidance and mentorship ever since we met in Fall 2016. I will always appreciate the time and effort he has spent in teaching me how to think like a researcher, and I will always be indebted to him for letting me tag along in some of his very challenging research.

Bibliography

[1] M. S. Amin, R. L. Finley Jr, and H. M. Jamil, "Top-k similar graph matching using tram in biological networks", *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, no. 6, pp. 1790–1804, 2012.

[2] D. Koutra, H. Tong, and D. Lubensky, "Big-align: Fast bipartite graph alignment", in *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, IEEE, 2013, pp. 389–398.

[3] J. Jin, S. Khemmarat, L. Gao, and J. Luo, "Querying web-scale information networks through bounding matching scores", in *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2015, pp. 527–537.

[4] N. Pržulj, "Biological network comparison using graphlet degree distribution", *Bioinformatics*, vol. 23, no. 2, e177–e183, 2007.

[5] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj, "Topological network alignment uncovers biological function and phylogeny", *Journal of the Royal Society Interface*, rsif20100063, 2010.

[6] J. Crawford and T. Milenković, "Great: Graphlet edge-based network alignment", in *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, IEEE, 2015, pp. 220–227.

[7] R. W. Solava, R. P. Michaels, and T Milenković, "Graphlet-based edge clustering reveals pathogen-interacting proteins", *Bioinformatics*, vol. 28, no. 18, pp. i480–i486, 2012.

[8] T. Hočevar and J. Demšar, "A combinatorial approach to graphlet counting", *Bioinformatics*, vol. 30, no. 4, pp. 559–565, 2014.

[9] ——, "Combinatorial algorithm for counting small induced graphs and orbits", *PloS one*, vol. 12, no. 2, e0171428, 2017.

[10] U. Schöning, "Graph isomorphism is in the low hierarchy", *Journal of Computer and System Sciences*, vol. 37, no. 3, pp. 312–323, 1988.

[11] L. Babai and E. M. Luks, "Canonical labeling of graphs", in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, ACM, 1983, pp. 171–183.

[12] L. Babai, "Graph isomorphism in quasipolynomial time", in *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, ACM, 2016, pp. 684–697.

[13] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii", *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[14] B. D. McKay, "Nauty user's guide (version 2.2)", Technical Report TR-CS-9002, Australian National University, Tech. Rep., 2003.

[15] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs", in *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, 2001, pp. 149–159.

[16] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo iso: Towards ultrafast and robust subgraph isomorphism search in large graph databases", in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 2013, pp. 337–348.

[17] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases", in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, 2008, pp. 405–418.

[18] Ö. N. Yaveroğlu, N. Malod-Dognin, D. Davis, Z. Levnajic, V. Janjic, R. Karapandza, A. Stojmirovic, and N. Pržulj, "Revealing the hidden language of complex networks", *Scientific reports*, vol. 4, p. 4547, 2014.

[19] *Jgrapht - a free java graph library*, Access Date: 05-30-2018, 2018. [Online]. Available: `https://jgrapht.org/`.