Subgraph Similarity Search in Large Graphs

Kanigalpula Samanvi

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

June 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

KANIGALPULA SAMANVI

(Kanigalpula Samanvi)

CS13M1001

(Roll No.)

Approval Sheet

This Thesis entitled Subgraph Similarity Search in Large Graphs by Kanigalpula Samanvi is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

Dr(VINEETH N BALASUBRAMANIAN) Examiner
Dept of Computer Science & Engineering.
IITH

[illegible]

(BALASUBRAMANYAM S.) Examiner
Dept of Mathematics
IITH

[illegible]

(Dr. Naveen Sivadasan) Adviser
Dept. of Computer Science and Engineering
IITH

[illegible]

Dr.(C. KRISHNA MOHAN) Chairman
Dept of Computer Science & Engineering
IITH

Acknowledgements

I take this opportunity to express my sincere gratitude to my thesis adviser Dr. Naveen Sivadasan for his constant encouragement, patience and immense knowledge. His guidance and persistent help, not only made this dissertation possible, but also helped me become a better individual than I was. I am also thankful to my seniors and friends for their selfless help, motivation and guidance. Finally, I am greatful to god and my family for their unconditional love, support and encouragement.

Abstract

One of the major challenges in applications related to social networks, computational biology, collaboration networks etc., is to efficiently search for similar patterns in their underlying graphs. These graphs are typically noisy and contain thousands of vertices and millions of edges. In many cases, the graphs are unlabeled and the notion of similarity is also not well defined. We study the problem of searching an induced subgraph in a large target graph that is most similar to the given query graph. We assume that the query graph and target graph are undirected and unlabeled. We use graphlet kernels [1] to define graph similarity. Graphlet kernels are known to perform better than other kernels in different applications.

Our algorithm maps topological neighborhood information of vertices in the query and target graphs to vectors. These local topological informations are then combined to find a target subgraph having highly similar global topology with the given query graph. We tested our algorithm on several real world networks such as facebook network, google plus network, youtube network, amazon network etc. Most of them contain thousands of vertices and million edges. Our algorithm is able to detect highly similar matches when queried in these networks. Our multi-threaded implementation takes about one second to find the match on a 32 core machine, excluding the time for one time preprocessing. Computationally expensive parts of our algorithm can be further scaled to standard parallel and distributed frameworks like map-reduce.

Contents

Chapter 1

Introduction

Similarity based graph searching has attracted considerable attention in the context of social networks, road networks, collaboration networks, software testing, computational biology, molecular chemistry etc. In these domains, underlying graphs are large with tens of thousands of vertices and millions of edges. Subgraph searching is fundamental to the applications, where occurrence of the query graph in the large target graph has to be identified. Searching for exact occurrence of an induced subgraph isomorphic to the query graph is known as the subgraph isomorphism problem, which is known to be NP-complete for undirected unlabeled graphs.

Presence of noise in the underlying graphs and need for searching 'similar' subgraph patterns are characteristic to these applications. For instance, in computational biology, the data is noisy due to possible errors in data collection and different thresholds for experiments. In object-oriented programming, querying typical object usage patterns against the target object dependency graph of a program run can identify deviating locations indicating potential bugs [2]. In molecular chemistry, identifying similar molecular structures is a fundamental problem. Searching for similar subgraphs plays a crucial role in mining and analysis of social networks. Subgraph similarity searching is therefore more natural in these settings in contrast to exact search. In subgraph similarity search problem, induced subgraph of the target graph that is 'most similar' to the query graph has to be identified, where similarity is defined using some distance function. Quality of the solution and computational efficiency are two major challenges in these search problems. In this work, we assume that both the underlying graph and query graph are unlabeled and undirected.

1.1 Distance metric

Most applications work with a distance metric to define similarity between two entities (graphs in our case). More is the distance metric value, less similar the objects are. Popular distance metrics include Euclidean distance, Hamming distance, Edit distance, Kernel functions [3–6] etc. We use graph kernel functions to define graph similarity.

1.1.1 Kernels

Kernels are symmetric functions that map pairs of entities from a domain to real values which indicate their similarity. Kernels that are positive definite not only define similarity between pairs of entities but also allow implicit mapping of objects to a high-dimensional feature space and operating on this space without requiring to compute explicit mapping of objects in the feature space. Kernels implicitly yield inner products between the feature vectors without explicit computation of the same in feature space. This is usually computationally cheaper than explicit computation. This approach is usually referred to as the kernel trick or kernel method. Kernel methods have been widely applied to sequence data, graphs, text, images, videos etc., as many of the standard machine learning algorithms including support vector machine (SVM) and principle component analysis (PCA) can directly work with kernels.

Kernels have been successfully applied in the past in the context of graphs [7–9]. There are several existing graph kernels based on various graph properties, such as random walks in the graphs [10, 11], cyclic patterns [12], graph edit distance [13], shortest paths [14, 15], frequency of occurrences of special subgraphs [16, 17], common subtree-patterns [18] and so on.

1.1.2 Graphlet Kernel

Graphlet kernels are defined based on occurrence frequencies of small induced subgraphs called graphlets in the given graphs [1]. Graphlet kernels have been shown to provide good SVM classification accuracy in comparison to random walk kernel and shortest path kernel on different data sets including protein and enzyme data [1]. Graphlet kernels are also of theoretical interest. It is known that under certain restricted settings, if two graphs have distance zero with respect to their graphlet kernel value then they are isomorphic [1]. Improving the efficiency of computing graphlet kernel is also studied in [1]. Graphlet kernel computation can also be scaled to parallel and distributed setting in a fairly straight forward manner. In our work, we use graphlet kernels to define graph similarity.

1.2 Related Work

Similarity based graph searching has been studied in the past under various settings. In many of the previous works, it is assumed that the graphs are labeled. In one class of problems, a large database of graphs is given and the goal is to find the most similar match in the database with respect to the given query graph [19–24]. In the second class, given a target graph and a query graph, subgraph of the target graph that is most similar to the query graph needs to be identified [25, 26]. Different notions of similarity were also explored in the past for these classes of problems.

In [27], approximate matching of query graph in a database of graphs is studied. The graphs are assumed to be labeled. Structural information of the graph is stored in a hybrid index structure based on B-tree index. Important vertices of a query graph are matched first and then the match is extended progressively. In [28], graph similarity search on labeled graphs from a large database of graphs under minimum edit distance is studied. In [25], algorithm for computing top-k approximate subgraph matches for a given query graph in a large labeled target graph is given. In this work, the target graph is converted into a set of multidimensional vectors based on the labels in the vertex neighborhoods. Only matches above a user defined threshold are computed. With higher threshold values, the match is a trivial vertex to vertex label matching. Subgraph matching in a large target graph for graphs deployed on a distributed memory store was studied in [26]. It looks for exact matching and not similarity matching. Though different techniques were studied in the past for the problem of similarity searching in various settings, to the best of our knowledge, little work has been done on subgraph similarity search on large unlabeled graphs. In many of the previous works, either the vertices are assumed to be labeled or the graphs they work with are small with hundreds of vertices.

1.3 Overview of the Work

We consider undirected graphs with no vertex or edge labels. We use graphlet kernel to define similarity between graphs. We give a subgraph similarity matching algorithm that takes as input a large target graph and a query graph and identifies an induced subgraph of the target graph that is most similar to the query graph with respect to the graphlet kernel value.

In our algorithm, we first compute vertex labels for vertices in both query and target graph. These labels are vectors in some fixed dimension and are computed based on local neighborhood structure of vertices in the graph. Since our vertex labels are vectors, unlike many of the other labeling techniques, our labeling allows us to define the notion of similarity between vertex labels of two vertices to capture the topological similarity of their corresponding neighborhoods in the graph. We build a nearest neighbor data structure for vertices of the target graph based on their vertex labels. Computing vertex label for target graph vertices and building the nearest neighbor data structure are done in the one time pre-processing phase. Using nearest neighbor queries on this data structure, vertices of the target graph that are most similar to the vertices of the query graph are identified. Using this smaller set of candidate vertices of target graph, a seed match is computed for the query graph. Using this seed match as the basis, our algorithm computes the final match for the full query graph.

We study the performance of our algorithm on several real life data sets including facebook network, google plus network, youtube network, road network, amazon network provided by the Stanford Large Network Dataset Collection (SNAP) [29] and DBLP network [30]. We conduct number of experimental studies to measure the search quality and run time efficiency. For instance, while searching these networks with their communities as query graphs, the computed match and the query graph has similarity score close to 1, where 1 is the maximum possible similarity score. In about 30% of the cases, our algorithm is able to identify the exact match and in about 80% of the cases, vertices of exact match are present in the pruned set computed by the algorithm. We validate our results by showing that similarity scores between random subgraphs and similarity scores between random communities in these networks are significantly lower. We also query communities across networks and in noisy networks and obtain matches with significantly high similarity scores. We use our algorithm to search for dense subgraphs and identify subgraphs with significantly high density.

Computationally expensive parts of our algorithm can be easily scaled to standard parallel and distributed computing frameworks such as map-reduce. Most of the networks in our experiments have millions of edges and tens of thousands of vertices. Our multi-threaded implementation of the search algorithm takes close to one second on these networks on a 32 core machine for the search phase. This excludes time taken by the one time pre-processing phase.

1.4 Thesis Outline

This remainder of this thesis is organized as follows. We formally define graphs, graph similarity search and kernels and their properties in chapter 2. We discuss few graph kernels devised in the past along with their advantages and drawbacks in chapter 3. We discuss the aim of our work and our vertex similarity measure in chapter 4. Chapter 5 contains the details of our approach. We present a comprehensive experimental study of our work using various real life datasets in chapter 6 and conclude with a few remarks on the study and future work in chapter 7.

Chapter 2

Preliminaries

Graph is an ordered pair $G = (V, E)$ comprising a set V of vertices and a set E of edges. To avoid ambiguity, we also use $V(G)$ and $E(G)$ to denote the vertex and edge set. We consider only undirected graphs with no vertex or edge labels. A subgraph H of G is a graph whose vertices are a subset of V, and whose edges are a subset of E and is denoted as $H \subseteq G$. An induced subgraph G' is a graph whose vertex set V' is a subset of V and whose edge set is the set of all edges present in G between vertices in V'.

Definition 1 (Graph Isomorphism). Graphs G and G' are isomorphic if there exists a bijection $b : V(G) \to V(G')$ such that any two vertices u and v of G are adjacent in G' if and only if $b(u)$ and $b(v)$ are adjacent in G'.

Definition 2 (Subgraph Isomorphism). Graph G is isomorphic to a subgraph of graph G', if there is an induced subgraph of G' that is isomorphic to G.

Definition 3 (Graph Similarity Searching). Given a collection of graphs and a query graph, find graphs in the collection that are closest to the query graph with respect to a given distance function between graphs.

Definition 4 (Subgraph Similarity Searching). Given graphs G and G', determine a subgraph $G^* \subseteq G$ that is closest to G' with respect to a given distance function between graphs.

2.1 Kernel

The idea of finding some non-linear mapping $\Phi : \chi \to H$, such that we map $x \in \chi$ and $x' \in \chi$, in lower dimensional space χ, to $\Phi(x)$ and $\Phi(x')$ respectively, in some higher dimensional space H and then compare them by taking dot product of $\Phi(x)$ and $\Phi(x')$ is known as kernel. We formally define kernel as follows,

Definition 5 (Kernel). Suppose $\langle x, x' \rangle$ denote dot product of x and x' then kernel function k is defined as,

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \tag{2.1}$$

Note that the kernel function is defined as dot product of x and x' in some higher dimensional space. Thus, it is a symmetric function.

Kernel trick: Mapping the objects from a general set χ into an inner product space H (equipped with its natural norm), without having the need to compute the mapping explicitly is known as the *kernel trick* or *kernel method.* By using kernel trick we save the computation cost of finding explicit mapping function. It can easily be embedded into standard classification methods which uses dot product to classify objects like SVM (support vector machine) and principle component analysis (PCA).

2.2 Positive Semi-definite Kernel

For a kernel function to be able to compare objects it needs to follow certain properties. To understand what kind of kernel functions effectively compare objects, we need to define the following,

Definition 6 (Gram Matrix). Let $\mathbb{R}$ be the set of real numbers and $\mathbb{C}$ be the set of complex numbers. Given a function $k : \chi^2 \rightarrow \mathbb{K}$ (where $\mathbb{K} = \mathbb{C}$ or $K = \mathbb{R}$) and patterns $x_1, x_2, ..., x_m \in \chi$, the $m \times m$ matrix K with elements,

$$K_{ij} = k(x_i, x_j) \tag{2.2}$$

is called the gram matrix (or kernel matrix) of k with respect to $x_1, x_2, ..., x_m$.

Definition 7 (Positive Semi-definite Matrix). A real $m \times m$ matrix K satisfying

$$\sum_{i,j=1}^{m} r_i \bar{r}_j K_{ij} \geq 0 \tag{2.3}$$

for all $r_i \in \mathbb{R}$ is called positive semi-definite matrix, where $\mathbb{R}$ denotes the set of complex numbers.

Definition 8 (Positive Semi-definite Kernel). Let $\mathbb{N}$ be the set of natural numbers and χ be a non empty set. A symmetric function k on $\chi \times \chi$ for all $m \in \mathbb{N}$ and all $x_1, x_2, ..., x_m \in \chi$ that generates a positive definite gram matrix is called a positive semi-definite (psd) kernel. We often refer to psd kernel as a kernel.

If k is a psd kernel function, only then the Hilbert space H can be constructed with,

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \tag{2.4}$$

A Hilbert space is an inner product space, which is closed with respect to inner product. The Hilbert space associated with a kernel is referred to as a Reproducing Kernel Hilbert Space (RKHS) [31]. It is well known fact that every kernel function is associated with a RKHS and every RKHS is associated with a kernel function. So, we need not find explicit mapping of objects in higher dimension when we have psd kernel.

Theorem 1 (Mercer's Theorem). A symmetric function $k(\cdot, \cdot)$ is a kernel if and only if for any finite sample ζ the kernel matrix for ζ is psd [32].

2.3 Properties of Kernel

Every psd kernel satisfies properties mentioned in [33]. Consider any space ζ of samples and kernels $k_1(x, y)$ and $k_2(x, y)$ over ζ with $x, y \in \zeta$. Then $k(x, y)$ is a kernel with,

- $k(x, y) = k_1(x, y) + k_2(x, y)$, sum of two kernels is a kernel.
- $k(x, y) = k_1(x, y)k_2(x, y)$, product of kernels is a kernel.
- $k(x, y) = \alpha k_1(x, y)$ where $\alpha > 0$
- $k(x, y) = f(x)f(y)$ for any function f on x
- $k(x, y) = \frac{k_1(x,y)}{\sqrt{k_1(x,x)}\sqrt{k_1(y,y)}}$

and so on. Combining one or more kernels based on above properties one can get a new kernel.

2.4 Nearest Neighbor Search

Nearest neighbor search is an optimization problem to find the object that is closest to a given object from a set of objects. Inorder to express the notion of closeness, similarity functions are used. Higher the values of similarity function, more similar are the objects. The problem of nearest neighbor search is also referred to as similarity search, proximity search or closest point search.

2.4.1 k Nearest Neighbor Search

k nearest neighbor search problem is a generalization of the nearest neighbor search problem. In k nearest neighbor search, instead of the finding the most similar object to the given object, k most similar ones are identified.

Several solutions have been proposed to efficiently solve the problem of k nearest neighbors search. Few of them are linear search, space partitioning, locality sensitive hashing etc. Of the above mentioned solutions, we use space partitioning in our work, as it is proved to be efficient for large set of objects with less number of dimensions. Since, our work deals with dimensions less than 10, using space partitioning we can efficiently solve the problem of finding k nearest neighbors in our work.

2.4.2 Space Partitioning using k-d trees

Various data structures like k-d tree, R-tree, vp-tree, BK-tree, BSP tree are proposed to solve the k nearest neighbors search problem using space partitioning technique. We use k-d trees for this purpose in this work.

k-d tree is a binary tree in which every vertex is a k-dimensional point. Each non-leaf vertex generates a splitting hyperplane that divides the space into two parts, known as half-spaces. The left subtree of this vertex represents the points to the left of this hyperplane and points right of the hyperplane are represented by the right subtree.

Our proposed algorithm constructs a k-d tree index of the vertex lables of the target graph. These vertex labels are nothing but graphlet vectors of some fixed dimension. The vertex lable generation and k-d tree index construction are discussed in detail in chapter 4 and chapter 5 respectively.

Chapter 3

Existing Graph Kernels

A Kernel that operates on graphs to measure their similarity is known as graph kernel. These graph kernels belong to larger class of R-convolution kernel defined by [3]. We now discuss few approaches for graph kernels that were proposed in the past.

3.1 Random Walk Kernel

Gartner proposed calculating random walks on direct product of graphs is equivalent to calculating common random walks on graphs individually. Direct product graph is also known as tensor product or categorical product.

Definition 9 (Direct Product Graph). The direct product of two graphs $G(V, E)$ and $G'(V', E')$ is denoted as $G_x(V_x, E_x) = G \times G'$. The vertex set V_x and edge set E_x of G_x are respectively defined as,

$$V_x = \{(u, v) : u \in V, v \in V'\} \tag{3.1}$$

$$E_x = \{((u_i, v_p), (u_j, v_q)) : (u_i, u_j) \in E \wedge (v_p, v_q) \in E'\} \tag{3.2}$$

An example of direct product graph is shown in Figure 3.1.

Random Walk Kernel [10], abbreviated as RWK, compares number of common random walks in given graphs. The trick to calculate number of common random walks of length d is by taking direct product of the graphs. Once the direct product is taken, raise the adjacency matrix A, of the resultant graph, by power of d to get the number of common walks. Random walk kernel is formally defined as follows,

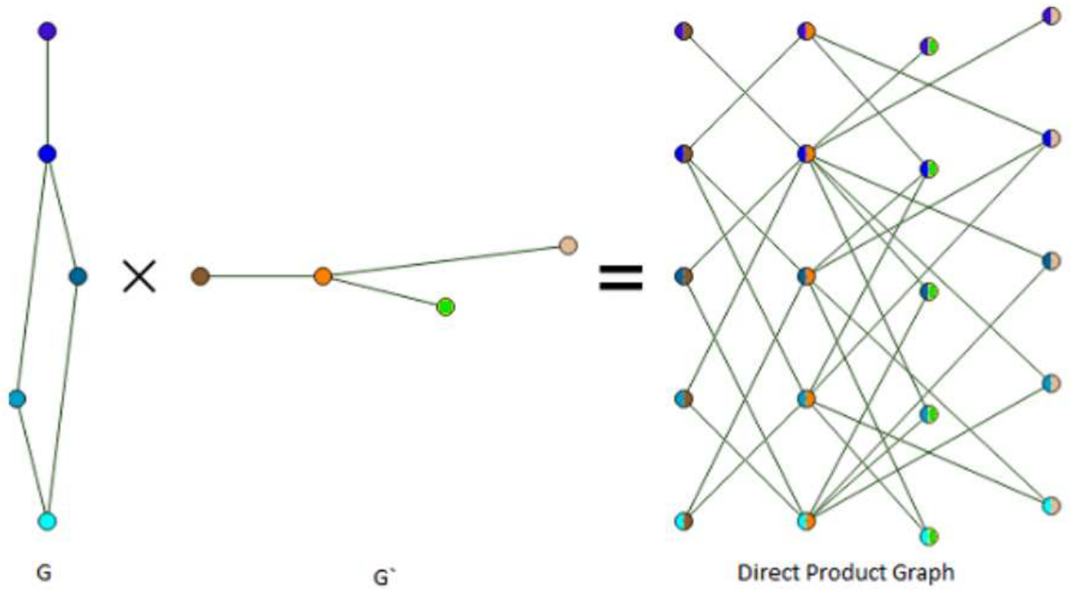


Figure 3.1: Example of a direct product graph

Definition 10 (Random Walk Kernel). Let G and G' be two graphs and A_x denote the adjacency matrix of their direct product graph $G_x(V_x, E_x)$. Given a sequence of weights (also known as decaying factor) $w = w_1, w_2, ..w_d (w_i \in \mathbb{R}, w_i \geq 0$ for all $i \in \mathbb{N})$ the random walk kernel is defined as,

$$k_x(G, G') = \sum_{i,j=1}^{|V_x|} \left(\sum_{d=1}^{\infty} w_d A_x^d \right)_{ij} \tag{3.3}$$

3.1.1 Drawbacks

Tottering: Some vertices might get visited over and over again for walks going back and forth, as walk allows repetition of vertices. This can result in high similarity between two graph because of one edge being in common.

Halting: Decaying factor has to be small (in most of the cases) for convergence. By choosing small w we are down-weighing the random walk of length more than 1 significantly, meaning the random walks of length more than 1 are almost neglected.

High Runtime: $O(n^6)$
If we set $w_d = w^d$ then, the kernel equation 3.3 will be reduced to,

$$k_x(G, G') = \sum_{i,j=1}^{|V_x|} \left[(I - wA_x)^{-1} \right]_{ij} \tag{3.4}$$

As A_x is $n^2 \times n^2$ matrix (assuming G and G' are of size n each) the inverse would take $O(n^6)$.

3.1.2 Fast Computation of RWK

In [34] time complexity of RWK was reduced from $O(n^6)$ to $O(n^3)$. Sylvester equation was used to speed-up the process of finding the inverse of matrix, the term in summation in above equation 3.4.

3.2 Shortest Path Kernel

As seen previously, in the case of random walk kernel, graph kernels that consider walks as their basis for comparison suffer from tottering and halting. On the other hand, paths do not suffer from tottering as there is no repetition of vertices. So, [14] proposed a new kernel based on paths. But computing all paths in a graph is an NP-hard problem. To overcome this difficulty the kernel was based on shortest path distance as it is unique for a graph and can be computed in polynomial time.

Definition 11 (Shortest Path Distance Matrix). Let $G(V, E)$ be a graph of size n. Let $d(v_i, v_j)$ be the length of the shortest path between $v_i, v_j \in V$. The shortest path matrix D of G is a $n \times n$ matrix. Each entry d_{ij} of D is defined as

$$d_{ij} = \left\{ \begin{matrix} d(v_i, v_j) & \text{if } v_i \text{ and } v_j \text{ are connected} \\ \infty & \text{otherwise} \end{matrix} \right\} \tag{3.5}$$

Definition 12 (Shortest Path Graph). Consider a graph $G(V, E)$ with D being its shortest path distance matrix. The shortest-path graph S of G has its vertex set as V , that is the vertex set of G itself and its edge set is defined using the adjacency matrix $A(S)$ as follows,

$$A(S)_{ij} = \left\{ \begin{matrix} 1 & \text{if } D(v_i, v_j) < \infty \\ 0 & \text{otherwise} \end{matrix} \right\} \tag{3.6}$$

where $D(v_i, v_j)$ is the edge label of edge (v_i, v_j) in S.

To compute SPK we have to transform the graphs G and G' into shortest path graphs S and S' using Floyds algorithm. This transformation is often refered to as Floyd-transformation. SPK is formally defined as follows,

Definition 13 (Shortest Path Kernel). Given two graphs G and G'. Consider S and S', the Floyd-transformed graphs of G and G'. The shortest-path graph kernel, $k(G, G') = k(S, S')$, on $S = (V, E)$

and $S' = (V', E')$ is defined as

$$k_{shortestpath}(S, S') = \sum_{v_i, v_j \in V} \sum_{v'_k, v'_l \in V'} k_{length} \left(d(v_i, v_j) \ , \ d(v'_k, v'_l) \right) \tag{3.7}$$

where $k_{length}(\cdot, \cdot)$ is any psd kernel to compare lengths of shortest path like delta kernel or linear kernel.

Linear kernel can be defined as the product of $d(v_i, v_j)$ and $d(v'_k, v'_l)$. *Delta kernel* is a kernel whose value is Equal to 1 if and only if $d(v_i, v_j) = d(v'_k, v'_l)$ otherwise, it is 0. It can be easily proved that SPK is a psd kernel.

3.2.1 Adavantages and Drawbacks

SPK has the following advantages,

- SPK doesnt suffer from tottering.
- It is an accurate classification kernel.
- Time complexity is $O(n^4)$ wich is much less than random walk kernel.

The drawback of this approach is that, $O(n^4)$ is too slow for large Graphs.

3.3 Other Kernels

In this section we briefly discuss few other graph kernels.

Cyclic Pattern Kernel [12] decomposes a graph into cyclic patterns and then counts the number of common cyclic patterns occuring in both graphs. It is computationally expensive as computing the cyclic pattern kernel on a general graph is NP-hard.

Edit Distance based kernel [13] is based on computing the cost c, which is the number of edit operations required to transform a graph G into another graph G'. The edit distance based kernel works by minimizing this cost c over all possible sequences of edit operations to convert graph G into graph G'. It is not psd kernel and worst case time complexity can be exponential and hence not used in practice.

Optimal assignment kernel [16] finds a best match, an optimal assignment between the substructures from G and G'. It is not a psd kernel, hence it is not applicable everywhere.

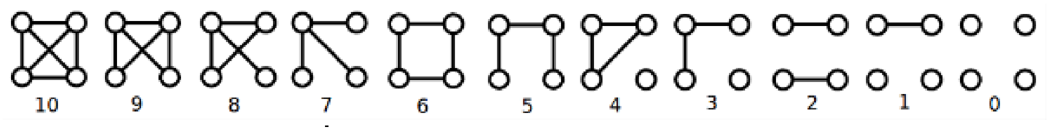


Figure 3.2: Set of all non isomorphic graphlets of size 4

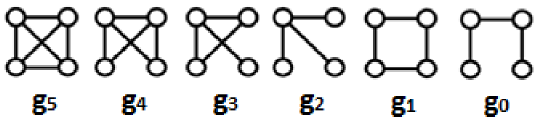


Figure 3.3: Non isomorphic connected graphlets of size 4

Subtree-Pattern Kernel [18] compares subtree-patterns. It suffers from tottering and can have exponential time complexity with respect to the height of the sub-tree patterns considered.

Weighted decomposition kernel [17] decomposes a graph into small subparts and assigns similarity of these subparts with different weights. It works efficiently only for highly simplified representation of graph.

Note that the kernels discussed above work efficiently for small graphs with tens or hundreds of vertices and are quite useful in cheminformatics. Since the applications under consideration for this particular work, like social networks, web graphs, road networks, collaboration networks deal with large graphs, with tens of thousands of vertices and millions of edges, we need efficient graph kernel for large graphs. In the next section we discuss a graph kernel for large graphs.

3.4 Graphlet Kernel

Graphlets are fixed size non isomorphic induced subgraphs of a large graph. Typical graphlet sizes considered in applications are $3, 4$ and 5. Number of graphlets of size l are exponential with respect to l. But number of modulo isomorphic graphlets of size l will be far less. For example, Figure 3.2 shows all possible non isomorphic size 4 graphlets. There are 11 of them, of which 6 are connected. We denote by D_l, the set of all size l graphlets that are connected. The set D_4 is shown in Figure 3.3.

Definition 14 (Graphlet Vector). For a given l, the graphlet vector f_G for a given graph G is a frequency vector of dimension $|D_l|$ where its ith component corresponds to the number of occurrences of the ith graphlet of D_l in G. Here, the graphlet vector f_G is assumed to be normalized by the L_2 norm $||f_G||_2$. In [1], the graphlet vector is normalized by the L_1 norm $||f_G||_1$. We use L_2

normalization instead, as it is directionally invariant.

If graphs G and G' are isomorphic then clearly their corresponding graphlet vectors f_G and $f_{G'}$ are identical. But the reverse need not be true in general. But, it is conjectured that given two graphs G and G' of n vertices and their corresponding graphlet vectors f_G and $f_{G'}$ with respect to $n-1$ sized graphlets D_{n-1}, graph G is isomorphic to G' if f_G is identical to $f_{G'}$ [1]. The conjecture has been verified for $n \leq 11$ [1]. Kernels based on similarity of graphlet vectors provide a natural way to express similarity of underlying graphs.

Definition 15 (Graphlet Kernel). Given two graphs G and G', let f_G and $f_{G'}$ be their corresponding graphlet frequency vectors with respect to size l graphlets for some fixed l. The graphlet kernel value $K(G, G')$ is defined as the dot product of f_G and $f_{G'}$. That is, $K(G, G') = f_G^T f_{G'}$

Graphlet vectors are in fact an explicit embedding of graphs into a vector space whose dimension is $|D_l|$ if size l graphlets are used. Graphlet kernels have been shown to give better classification accuracies in comparison to other graph kernels like random walk kernel and shortest path kernel for certain applications [1]. Values of $K(G, G') \in [0, 1]$ and larger values of $K(G, G')$ indicate higher similarity between G and G'.

To find f_G exactly using naive algorithm we have to consider $\binom{n}{k}$ graphlets of size l for a graph of size n.Thus the theoretical time complexity is $O(n^k)$. $O(n^k)$ is too expensive in practice, therefore two speed-up schemes including sampling and specifically designed scheme for bounded degree graphs are discussed below.

3.4.1 Sampling

Sampling of graphlets from the graph is done with the hope that if sufficient samples are drawn from a graph then the approximate similarity score is close to the actual one. By sampling some graphlets with some confidence in an error bound the kernel value can be calculated.

Theorem 2. Let D be a probability distribution on finite set $\mathcal{A} = \{1, 2, ..., a\}$. Let $X = \{X_j\}_{j=1}^m$ be an independently and identically distributed multiset, with $X_j \sim D$. For a given $\epsilon > 0$ and $\delta > 0$,

$$m = \left\lceil \frac{2\ \left(a \log 2 + log(\frac{1}{\delta})\right)}{\epsilon^2} \right\rceil$$

samples suffice to ensure that, $P\{\|D - \hat{D}^m\|1 \geq \epsilon\} \leq \delta$, where $\hat{D}^m(i) = \frac{1}{m}\sum_{j=1}^m \delta(X_j = 1)$ for $i \in \mathcal{A}$

and $\delta(X_j = i) = \begin{cases} 1, & X_j = i \\ 0, & \text{otherwise} \end{cases}$ [1]

For comparing graphs, consider $\mathcal{A}$ the set of all size l graphlets that are distributed according to an unknown distribution D. Let m be the number of graphlets randomly sampled from the graph. Then from above formula, the value of m gives number of randomly selected samples required to ensure that $\hat{D}^m$ is atmost ϵ distance away from the actual distribution D with a confidence of $1 - \delta$. For example, considering size 4 graphlets we have, 11 non-isomorphic distinct graphlets. If we set $\epsilon = 0.05$ and $\delta = 0.05$ then $m = 8,497$ and if we set, $\epsilon = 0.01$ and $\delta = 0.01$ then, $m = 244,596$.

3.4.2 Bounded Degree Graph

In graphs with bounded degree d, counting of graphlets can be performed efficiently. In other words, a BDG (bounded degree graph) with bounded degree d will not have any vertex with degree more than d. Two algorithms for counting graphlets efficiently in low bounded degree graphs were proposed,

Counting all connected graphlets: Under the assumption that graphs are given in adjacency list representation, a data structure that supports checking of existence of edge in $O(1)$ time was constructed.

Theorem 3. Let G be a BDG, and let d denote maximum degree. Then all connected graphlets of with size $l \in 3, 4, 5$ can be enumerated in $O(nd^{l-1})$ [1].

Counting all graphlets for a fixed vertex v:

Theorem 4. For a fixed node v of G , the distribution of subgraphs of size 3 can be computed in time $O(d^2)$, where d is the maximum degree of any node [1].

3.4.3 Advantages and Drawbacks

The main advantage of graphlet kernel is that it is scalable. That is, graphlet kernel computation can be easily paralellized for large graphs using standard map-reduce framework or OpenMP/MPI framework. Also the time Complexity is $O(nd^{l-1})$ for an n sized graph, where d is the maximum degree of a graph and l is the size of the graphlet considered for comparing graphs.

Despite its advantages, the maximum degree of a graph can be atmost $n - 1$, implying the worst case time complexity can be $O(n^{l-1})$, which is expensive. This kernel considers non-connected components as well for calculating the kernel value, which can unwantedly increase the similarity score between graphs.

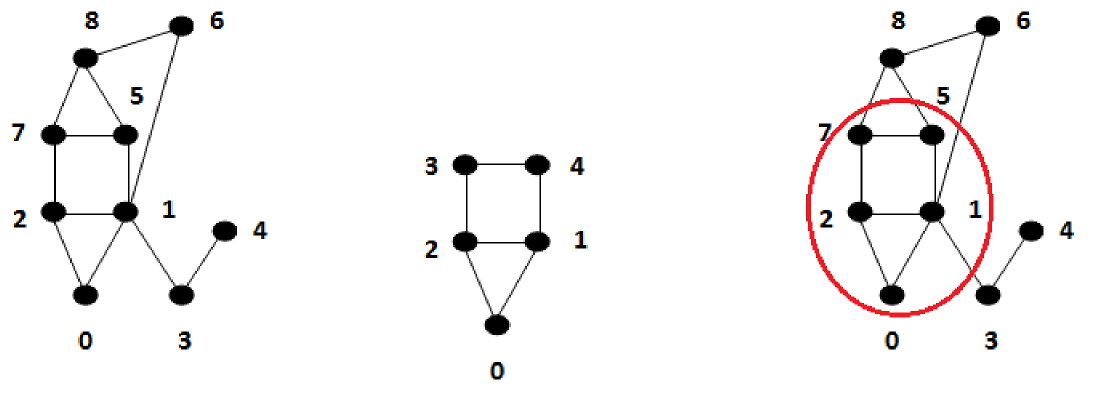
Chapter 4

Graphlet vector based vertex labeling

4.1 Problem Statement

In this section we formally define the main aim of our work.

Problem Statement. Let $K(\cdot,\cdot)$ be a graphlet kernel based on size l graphlets for some fixed l. Given a large connected graph G of size n and a connected query graph Q of size n_q with $n > n_q$, find a subset V^* of vertices in G such that its induced subgraph G^* in G maximizes $K(Q, G^*)$.



(a) Target Graph **(b)** Query Graph **(c)** Expected subgraph of (a)

Figure 4.1: Example Target graph with a given query graph and their expected similar subgraph result

Figure 4.1 shows a sample target graph, query graph and the expected induced subgraph of the target graph as a result of the query graph. The figure highlights in red, the induced subgraph of the target graph that is the expected result of the similarity search.

Computing vertex labels that capture topological neighborhood information of corresponding vertices in the graph and comparing vertex neighborhoods using their labels is crucial in our matching algorithm. Our vertex labels are graphlet vectors (discussed in section 3.4 above) of their corresponding neighborhood subgraphs.

4.2 Vertex Labeling

Given a fixed positive integer t and graph G, let $N(v)$ denote the depth t neighbors of vertex v in G. That is, $N(v)$ is the subset of all vertices in G (including v) that are reachable from v in t or less edges. Let H_v denote the subgraph induced by vertices $N(v)$ in G. We denote by f_v, the graphlet vector corresponding to the graph H_v, with respect to size l graphlets for some fixed l. We note that for defining the graphlet vector f_v for a vertex, there are two implicit parameters l and t. To avoid overloading the notation, we assume them to be some fixed constants and specify them explicitly when required. Values of l and t are parameters to our final algorithm.

For each vertex v of the graph, its vertex label is given by its graphlet vector f_v.

4.3 Vertex Similarity

Given vertex labels f_u and f_v for vertices u and v, we denote by $s(u,v)$ the similarity between labels of f_u and f_v, given by their dot product as

$$s(u,v) = f_u^T f_v \tag{4.1}$$

Values of $s(u,v) \in [0,1]$ and larger values of $s(u,v)$ indicate higher topological similarity between neighborhoods of vertices u and v. Computing the vertex labels of the target graph is done in the preprocessing phase. Implementation details of the vertex labeling algorithm are discussed in the next section.

Chapter 5

Graphlet Based Subgraph Similarity Search

Our subgraph similarity search algorithm has two major phases: one time pre-processing phase and the query graph matching phase. Each of these phases comprise sub-phases as given below. Details of each of these subphases is discussed in the subsequent sections.

A. Pre-processing Phase: This phase has two subphases:

1) In this phase, vertex labels f_v of all the vertices of the target graph G are computed.
2) k-d tree based nearest neighbor data structure on the vertices of G using their label vectors f_v is built.

B. Matching Phase: This phase is further divided into four subphases:

1) Selection Phase: In this phase, vertex labels f_v for vertices of the query graph Q are computed first. Each vertex u of the query graph then selects a subset of vertices from the target graph G closest to u based on their Euclidean distance.
2) Seed Match Generation Phase: In this phase, a one to one mapping of a subset of query graph vertices to target graph vertices is obtained with highest overall similarity score. Subgraph induced by the mapped vertices in the target graph is called the seed match. The seed match is obtained by solving a maximum weighted bipartite matching problem.
3) Match Growing Phase: The above seed match is used as a basis to compute the final match for Q.

4) Match Completion Phase: This phase tries to match those vertices in Q that are still left unmatched in the previous phase.

5.1 Pre-processing Phase

5.1.1 Computation of vertex labels f_v

In this phase, vertex label f_v for each vertex v of the target graph G is computed first. To compute f_v, we require parameter values t and l. These two values are assumed to be provided as parameters to the search algorithm. For each vertex v, a breadth first traversal of depth t is performed starting from v to obtain the depth t neighborhood $N(v)$ of v. The graph H_v induced by the vertex set $N(v)$ is then used to compute the graphlet vector f_v as given in [35]. The pseudo code is given in Algorithm 1.

Major time taken by the pre-processing phase is for computing the graphlet vector for H_v. In [1], methods to improve its efficiency including sampling techniques are discussed. We do not make use of sampling technique in our implementation. We remark that finding the graphlet frequencies can easily be scaled to parallel computing frameworks or distributed computing frameworks such as map-reduce.

Algorithm 1 Compute label f_v for vertex v

```
Input: Graph $G$, vertex $v$, BFS depth $t$, graphlet size $l$
Output: Label vector $f_v$
1: Run BFS on $G$ starting from $v$ till depth $t$. Let $N(v)$ be the set of vertices visited including $v$.
2: Identify the induced subgraph $H_v$ of $G$ induced by $N(v)$.
3: Compute graphlet vector $f_v$ for graph $H_v$.
4: Normalize $f_v$ by $||f_v||_2$.
return $f_v$
```

5.1.2 Nearest neighbor data structure on f_v

After computing vertex labels for G, a nearest neighbor data structure on the vertices of G based on their label vectors f_v is built. We use k-d trees for nearest neighbor data structure [36]. k-d trees are known to be efficient when dimension of vectors is less than 20 [36]. Since the typical graphlet size l that we work with are $3, 4$ and 5, the dimension of f_v (which is $|D_l|$) does not exceed 10.

5.2 Matching Phase

In the following we describe the three subphases of matching phase.

5.2.1 Selection Phase

The vertex labels f_v for all vertices of the query graph Q are computed first using Algorithm 1. Let R_v denote the set of k vertices in G that are closest to v with respect to the Euclidean distance between their label vectors. In our experiments, we usually fix k as 10. For each vertex v of Q, we compute R_v by querying the k-d tree built in the pre-processing phase. Let R denote the union of R_v for each vertex v of the n_q vertices of Q. For the subsequent seed match generation phase, we will only consider the vertex subset R of G. Clearly size of R is at most $k.n_q$ which is typically much smaller than the number of vertices in G.

5.2.2 Seed Match Generation Phase

In this phase, we obtain a one to one mapping of a subset of vertices of the query graph Q to the target graph G with highest overall similarity score. We call the subgraph induced by the mapped vertices in G as the seed match. To do this, we define a bipartite graph $(V(Q), R)$ with weighted edges, where one part is the vertex set $V(Q)$ of the query graph Q and the other part is the pruned vertex set R of G obtained in the previous step. The edges of the bipartite graph and their weights are defined as follows. Each vertex v in the part $V(Q)$ is connected to every vertex w in $R_v \subseteq R$, where R_v is the set of k nearest neighbors of v in G as computed in the previous step.

The weight $\lambda(v, w)$ for the edge (v, w) is defined in the following manner. Let $0 < \alpha < 1$ be a fixed scale factor which is provided as a parameter to the search algorithm. We recall that vertex v belongs to query graph Q and vertex w belongs to target graph G and $s(v, w)$ given by equation (4.1) denote the similarity between their label vectors f_v and f_w. Let V_w denote the neighbors of vertex w in graph G including w. Let Q' denote the subset of $V(Q)$ excluding v such that each vertex in Q' is connected to at least one vertex in V_w in the bipartite graph $(V(Q), R)$. In particular, for each vertex $u \in Q'$, let $s(u)$ denote the maximum $s(u, z)$ value among all its neighbors z in V_w in the bipartite graph. Now the weight $\lambda(v, w)$ for the edge (v, w) of the bipartite graph is given by

$$\lambda(v, w) = \frac{\left(s(v, w)^\alpha + \sum_{u \in Q'} s(u)^\alpha\right)^{1/\alpha}}{(|Q'| + 1)} \tag{5.1}$$

We now solve maximum weighted bipartite matching on this graph to obtain a one to one mapping between a subset of vertices of Q and the vertices of G. Defining edge weights $\lambda(v, w)$ to edge (v, w) in the bipartite graph in the above fashion not only takes into account the similarity value $s(v, w)$, but also the strength of similarity of neighbors of w in G to remaining vertices in the query graph Q. By assigning edge weights as above, we try to ensure that among two vertices in G with equal similarity values to a vertex in Q, the vertex whose neighbors in G also have high similarity to vertices in Q is preferred over the other in the final maximum weighted bipartite matching solution.

Let M denote the solution obtained for the bipartite matching. Let Q_M and G_M respectively denote the subgraphs induced by the subset of matched vertices from graphs Q and G under the matching M. The connectivity of Q_M and G_M may differ. For instance, the number of connected components in G_M and Q_M could differ. Therefore, we do not include all the vertices of G_M in the seed match. Instead, we use the largest connected component of G_M as a seed solution. That is, let $S_G \subset V(G)$ denote the subset of vertices in G_M corresponding to a maximum cardinality connected component. Let S_Q denote their corresponding mapped vertices in Q_M. We call S_G as a seed match. The pseudo code for seed match computation is given in Algorithm 2.

Algorithm 2 Computing seed match S_G in G and its mapped vertices S_Q in Q

```
Input: Vertex sets $V(Q)$, $R$ and $R_v$ for each $v \in V(Q)$ and their labels $f_v$, parameter $\alpha$
Output: $S_G$ and $S_Q$
1: Construct bipartite graph $(V(Q), R)$ with edge weights given by $\lambda(v, w)$.
2: Compute maximum weighted bipartite matching $M$ on $(V(Q), R)$
3: Let $Q_M$ and $G_M$ respectively denote the subgraphs induced by vertices from $Q$ and $G$ in the
   matching $M$.
4: Compute largest connected component in $G_M$. Let $S_G$ denote the vertices in that component.
   Let $S_Q$ denote its mapped vertices in $Q_M$ under the bipartite matching $M$.
return $S_G$ and $S_Q$
```

5.2.3 Match Growing Phase

After computing the seed match S_G in G and its mapped vertices S_Q in Q, we use this seed match as the basis to compute the final match. The final solution is computed in an incremental fashion starting with empty match. In each iteration, we include a new pair of vertices (v, w) to the solution, where v and w belongs to G and Q respectively. In order to do this, we maintain a list of candidate pairs and in each iteration, we include a pair with maximum similarity value $s(v, w)$ to the final solution. We use a max heap to maintain the candidate list. The candidate list is initialized with the mapped pairs between S_G and S_Q as obtained in the previous phase. Thus, the heap is initialized by inserting each of these mapped pairs (v, w) with corresponding weight $s(v, w)$.

We recall that the mapped pairs obtained from previous phase have stronger similarity with respect to the modified weight function $\lambda(v,w)$. Higher value of $\lambda(v,w)$ indicates that not only $s(v,w)$ is high but also their neighbors share high $s()$ value. Hence they are more preferred in the solution over other pairs with similar $s()$ value. By initializing the candidate list with these preferred pairs, the matching algorithm tries to ensure that the incremental solution starts with these pairs first and other potential pairs are considered later. Also, because of the heap data structure, remaining pairs are considered in the decreasing order of their similarity value. Moreover, as will be discussed later, the incremental matching tries to ensure that the partial match in G constructed so far is connected. For this, new pairs that are added to the candidate list are chosen from the neighborhood of the partial match between G and Q.

The incremental matching might still match vertex pairs with low $s()$ value if they are available in the candidate list. Candidate pairs with low $s()$ values should be treated separately as there could be genuine pairs with low $s()$ value. For instance, consider boundary vertices of an optimal subgraph match in G. Boundary vertices are also connected to vertices outside the matched subgraph. Hence, their local neighborhood structure is different from their counterpart in the query graph. In other words, their corresponding graphlet vectors can be very dissimilar and their similarity value $s()$ can be very low even though they are expected to be matched in the final solution. In order to find such genuine pairs, we omit pairs with similarity value below some fixed threshold h_1 in this phase and such pairs are handled in the next phase.

Algorithm 3 Incremental Matching

```
Input: Seed match $S_G$ and its mapped vertices $S_Q$, threshold $h_1$
Output: Partial match $F$
1: Initialize $F$ to empty set.
2: Initialize the candidate list max heap with mapped pairs $(v,w)$ of the seed match where $s(v,w) \geq h_1$.
3: while candidate list is not empty do
4:    Extract maximum weight candidate match $(v,w)$
5:    Add $(v,w)$ to $F$
6:    updateCandidateList(candidate list, $(v,w)$, $h_1$,$F$)
7: end while
return $F$
```

In each iteration of the incremental matching, a pair (v,w) with maximum $s(v,w)$ value is removed from the candidate heap and added to the final match. After this, the candidate list is modified as follows. We recall that v and w belong to G and Q respectively. We call a vertex unmatched if it is not yet present in the final match. The algorithm maintains two invariants: (a) the pairs present in the candidate list are one to one mappings and (b) a query vertex that enters the candidate list will stay in the candidate list (possibly with multiple changes to paired partner vertex) until it is included in the final match. Let U_v denote the unmatched neighbors of v in G that are also not present in the candidate list. Let U_w denote the unmatched neighbors w in Q. For each query vertex y in U_w, let x be a vertex in U_v with maximum similarity value $s(x, y)$. We add (x, y) to the candidate list if y is absent in the list and $s(x, y) \geq h_1$. If y is already present in the candidate list, then replace the current pair for y with (x, y) if $s(x, y)$ has a higher value. The incremental algorithm is given in Algorithm 3. The candidate list modification is described in Algorithm 4.

Algorithm 4 updateCandidateList

```
Input: candidate list, $(v, w)$, $h_1$ and $F$
1:  Compute $U_v$ which is the set of unmatched neighbors of $v$ in $G$ that are also not present in
    candidate list.
2:  Compute $U_w$ which is the set of unmatched neighbors of $w$ in $Q$.
3:  for all vertex $y \in U_w$ do
4:      Find $x \in U_v$ with maximum $s(x, y)$ value.
5:      if $y$ does not exist in candidate list then
6:          Include $(x, y)$ in the candidate list if $s(x, y) \geq h_1$.
7:      else
8:          Replace existing pair for $y$ in the candidate list with $(x, y)$ if $s(x, y)$ has higher value.
9:      end if
10: end for
```

5.2.4 Match Completion Phase

In this phase, vertices of the query graph Q that are left unmatched in the previous phase due to similarity values below the threshold h_1 are handled. Typically, boundary vertices of the final matched subgraph in G remain unmatched in the previous phase. As discussed earlier, this is because, such boundary vertices in G and their matched partners in Q have low $s()$ value as their local neighborhood topologies vastly differ. Hence using neighborhood similarity for such pairs is ineffective. To handle them, we try to match unmatched query vertices with unmatched neighbors of the current match F in G. Since the similarity function $s()$ is ineffective here, we use a different similarity function to compare potential pairs. Let X denote the set of unmatched neighbors of the current match F in G. Let Y denote the set of unmatched query vertices. Let $v \in X$ and let $w \in Y$. We define the similarity $c(v, w)$ as follows. Let Z_v denote the matched neighbors of v in target graph G and let Z_w denote the matched neighbors of w in query graph Q. Let Z'_v denote the matched partners of Z_v in Q. We now define $c(v, w)$ using the standard Jaccard similarity coefficient as

$$c(v, w) = \frac{|Z'_v \cap Z_w|}{|Z'_v \cup Z_w|} \tag{5.2}$$

We use a fixed threshold h_2 that is provided as parameter to the algorithm. We now define a bipartite graph (X, Y) with edge weights as follows. For each $(v, w) \in X \times Y$, insert an edge (v, w) with weight $c(v, w)$ in the bipartite graph if $c(v, w) \geq h_2$. Compute maximum weighted bipartite graph matching on this bipartite graph and include the matched pairs in the final solution F. In our experiments, size of Y (number of unmatched query graph vertices) is very small. The pseudo code is given in Algorithm 5.

Algorithm 5 Match Completion

```
Input: Partial match $F$ and threshold $h_2$
Output: Final match $F$
1: Let $X$ denote the set of unmatched neighbors of the match $F$ in $G$.
2: Let $Y$ denote the set of unmatched vertices in $Q$.
3: Construct bipartite graph $(X, Y)$ by introducing all edges $(v, w)$ with edge weight $c(v, w)$ if
   $c(v, w) \geq h_2$.
4: Compute maximum weighted bipartite matching.
5: Add each of these matches to $F$
return $F$
```

We remark that our searching algorithm finds the matched subset of vertices in G and also their corresponding mapped vertices in the query graph Q.

Chapter 6

Experiments and Results

In this section, we conduct experiments on various real life graph data sets [29] including social networks, collaboration networks, road networks, youtube network, amazon network and on synthetic graph data sets.

6.1 Experimental Data sets

Social Networks: We conduct experiments on facebook and google plus undirected graphs provided by Stanford Large Network Dataset Collection (SNAP) [29]. Facebook graph contains around 4K vertices and 88K edges. In this graph vertices represent anonymized users and an undirected edge connects two friends. google plus graph contains 107K vertices and 13M edges. google plus graph also represents users as vertices and an edge exists between two friends. The data set also contains list of user circles (user communities), where user circle is specified by its corresponding set of vertices. We use these user circles as query graphs and they are queried against the entire facebook network. We also query facebook circles against google plus network to find similar circles across networks. We also experiment querying facebook circles against facebook network after introducing random noise to the facebook network.

DBLP Collaboration Network: We use the DBLP collaboration network downloadable from [30]. This network has around 317K vertices and 1M edges. The vertices of this graph are authors who publish in any conference or journal and an edge exists between any two co-authors. All the authors who contribute to a common conference or a journal form a community. The data set provides a list of such communities by specifying its corresponding set of vertices. We use such communities as query graphs.

Youtube Network: Youtube network is downloaded from [29]. Network has about 1M vertices and 2M edges. Vertices in this network represent users and an edge exists between two users who are friends. In youtube, users can create groups in which other users can join. The data set provides a list of user groups by specifying its corresponding set of vertices. We consider these user-defined groups as our query graphs.

Road Network: We use the road network of California obtained from [29] in our experiments. This network has around 2M vertices and 3M edges. Vertices of this network are road endpoints or road intersections and the edges are the roads connecting these intersections. We use randomly chosen subgraphs from this network as query graphs.

Amazon Network: Amazon network is a product co-purchasing network downloaded from [29]. This network has around 334K vertices and 925K edges. Each vertex represents a product and an edge exists between the products that are frequently co-purchased [29]. All the products under a certain category form a product community. The data set provides a list of product communities by specifying its corresponding set of vertices. We use product communities as query graphs and we query them against the amazon network.

The statistics of the data sets used are listed in Table 6.1.

Data Set	**#vertices**	**#edges**
Facebook	4039	88234
Google Plus	107614	13673453
DBLP	317080	1049866
Amazon	334863	925872
Youtube	1134890	2987624
Road Network	1965206	2766607

Table 6.1: Data set Statistics

6.2 Experimental Setup

All the experiments are carried out on a 32 core 2.60GHz Intel(R) Xeon(R) server with 32GB RAM. The server has Ubuntu 14.04 LTS. Our implementation uses Java 7.

The computationally most expensive part of our algorithm is the computation of vector labels for all vertices of a graph. The preprocessing phase that computes label vectors for each vertex of the graph is multi-threaded and thus executes on all 32 cores. Similarly, in the matching phase, computing label vectors for all vertices of the query graph is also multi-threaded and uses all 32 cores. Remaining phases use only a single core.

6.3 Results

To evaluate the accuracy of the result obtained by our similarity search algorithm, we compute the graphlet kernel value $K(Q, G^*)$ between the query graph Q and the subgraph G^* of G induced by the vertices V^* of the final match F in G. We use this value to show the similarity between the query graph and our obtained match and we refer to this value as *similarity score* in our experiments. We recall that similarity score lies in the range $[0, 1]$ where 1 indicates maximum similarity.

There are six parameters in our algorithm: (1) graphlet size l, (2) BFS depth t for vertex label computation, (3) value of k for the k nearest neighbors from k-d tree, (4) value of α in the edge weight function λ and (5) similarity thresholds h_1 for match growing phase and h_2 for match completion phase. In all our experiments we fix graphlet size l as 4. We performed experiments with different values of k, α, h_1 and h_2 on different data sets. Based on the results, we chose ranges for these parameters. The value of k is chosen from the range 5 to 10. Even for million vertex graphs, $k = 10$ showed good results. We fix scaling factor α to be 0.3 and the thresholds h_1 and h_2 to be 0.4 and 0.95 respectively.

6.3.1 Experiment 1: Effect of bfs depth t

This experiment shows the effect of bfs depth t on the final match. We performed experiments with different values of t. We observed that after the depth of 2, there is very little change in the similarity scores of the final match. But as the depth increases the time to compute graphlet vectors also increases. Thus, the bfs depth t was taken to be 2 for most of our experiments. Table 6.2 shows the similarity scores of querying amazon communities on amazon network and and DBLP communities on DBLP collaboration network for different values of t. These results are averaged

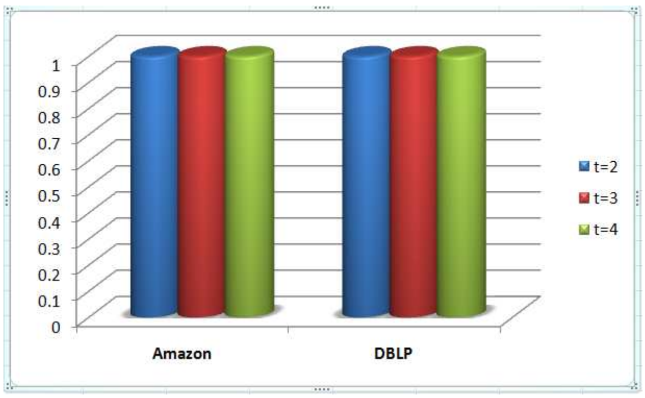


Figure 6.1: Experiment 1: Effect of bfs depth t

over 150 queries.

Data Set	t**=2**	t**=3**	t**=4**
Amazon	0.9999823	0.9999851	0.9999858
DBLP	0.9999942	0.9999896	0.9999917

Table 6.2: Experiment 1 : Similarity Score vs. t

The plot in Figure 6.1 shows the effect of bfs depth t on the final match similarity scores for amazon and DBLP data sets. It is clear from the results that for different values of depth, after a depth of 2, the similarity score doesn't vary much.

6.3.2 Experiment 2: Subgraph Similarity Searching with Induced Subgraphs as Queries

For each of the data sets discussed earlier, we perform subgraph querying against the same network. For each network, we use the given communities as query graphs and measure the quality of the search result. That is, we query facebook communities against facebook network, DBLP communities against DBLP network, youtube groups against youtube network and amazon product communities against amazon network. For road network, we use randomly chosen induced subgraphs from the network as query graph. Second column of Table 6.3 shows the similarity score of the match. All the results are averages over 150 queries. The average community (query graph) size is around 100 for facebook, around 40 for DBLP, around 50 for youtube and around 300 for amazon. Query graphs for road network have about 500 vertices.

To validate the quality of our solution, we do the following for each of the network. We compute the similarity score between random induced subgraphs from the same network. These random subgraphs contain 100 vertices. We also compute the similarity score between different communities

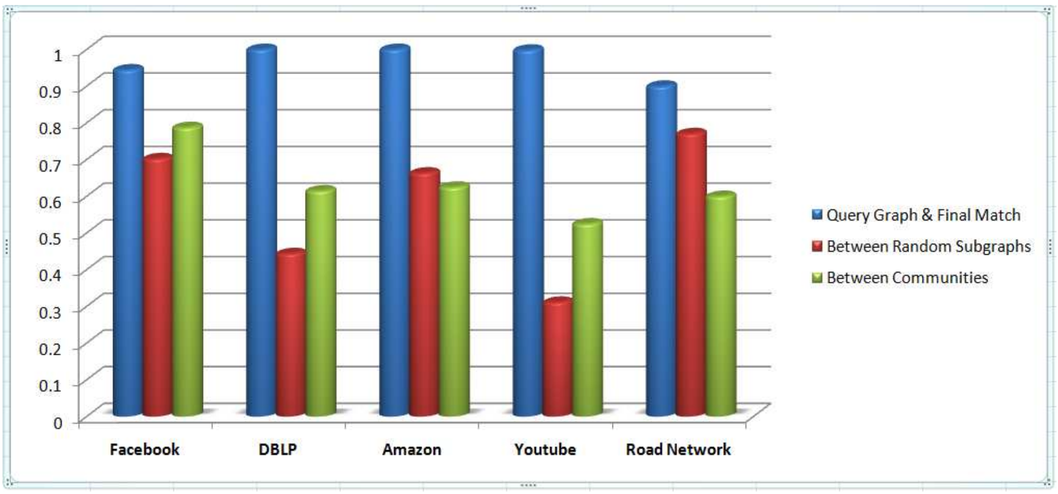


Figure 6.2: Experiment 2: Similarity scores of queries that are induced subgraphs in their respective data sets

from the same network. All results are averaged over 150 scores. Table 6.3 shows the result. The results show that the similarity score of our match close to 1 and is significantly better than scores between random subgraphs and scores between communities in the same network. For road network, the third column shows the average similarity between its query subgraphs.

Data Set	**Query graph & Final Match**	**Between Random Subgraphs**	**Between Communities**
Facebook	0.944231	0.702286	0.787296
DBLP	0.999994	0.443763	0.6144779
Amazon	0.999982	0.663301	0.624756
Youtube	0.998054	0.311256	0.524779
Road Network	0.899956	0.770492	0.599620

Table 6.3: Experiment 2 : Similarity Scores. Second column shows the average similarity score between query graph and the computed match. The query graphs are the given communities. Third column shows the average similarity score between random subgraphs. Fourth column shows average similarity score between communities

In Figure 6.2, the similarity scores of final matches obtained by our approach for the queries which are induced subgraphs of their respective data sets are depicted. The plot also shows the similarity scores between random subgraphs of each network and similarity scores within the query graphs. From the plot our approach can be easily validated, as our final match scores are high and close to 1, where as, the random subgraphs similarity is much less.

Table 6.4 shows the $\#exactMatches$ which is the number of queries that yielded the exact match out of the 150 queries (query graph is a subgraph of the network), and $\#inPruned$ - the percentage of queries where the vertices of the exact target match are present in the pruned subset of vertices

R of target graph G obtained after the selection phase. Table 6.4 shows that, for about 30% of the query graphs, our algorithm identifies the exact match. Also, for about 75% of the queries, vertices of the ideal match are present in our pruned set of vertices R in the target graph after selection phase.

Data Set	$\#exactMatches$ (out of 150)	$\#inPruned$ (percentage)
Facebook	53	83
DBLP	47	82
Amazon	60	72

Table 6.4: Experiment 2 : Exact Match Statistics

Table 6.5 shows the timing results corresponding to *Experiment 2*. The timing information is only for the matching phase and it excludes the one time pre-processing phase. Here δ denotes time taken (in secs) to compute the label vectors for all vertices of the query graph and τ the time taken (in secs) for the entire matching phase (including δ). We recall that the label vector computation is implemented as multithreaded on 32 cores and the remaining part is executed as a single thread. It can be seen that the label vector computation is the computationally expensive part and the remaining phases take much lesser time.

Data Set	$\boldsymbol{\delta}$(in sec)	$\boldsymbol{\tau}$(in sec)
Facebook	0.213596	0.253706
DBLP	0.159492	0.777687
Amazon	0.199767	0.781500
Youtube	0.225131	0.989452
Road Network	0.216644	1.437619

Table 6.5: Experiment 2 : Timing Results

6.3.3 Experiment 3: Subgraph Similarity Searching with Query Graphs that are not Induced Subgraphs of Target Graph

In all previous experiments, query graphs were induced subgraphs of the target network. In this experiment, we evaluate the quality of our solution when the query graph is not necessarily an induced subgraph of the target graph. For this, we conduct two experiments. In the first experiment, we use facebook communities as query graphs and query them against google plus network. To validate the quality of our solution, we measure the similarity score of the query graph with a random induced subgraph in the target graph with same number of vertices. In the second experiment, we create a modified facebook network by randomly removing 5% its original edges. We use this

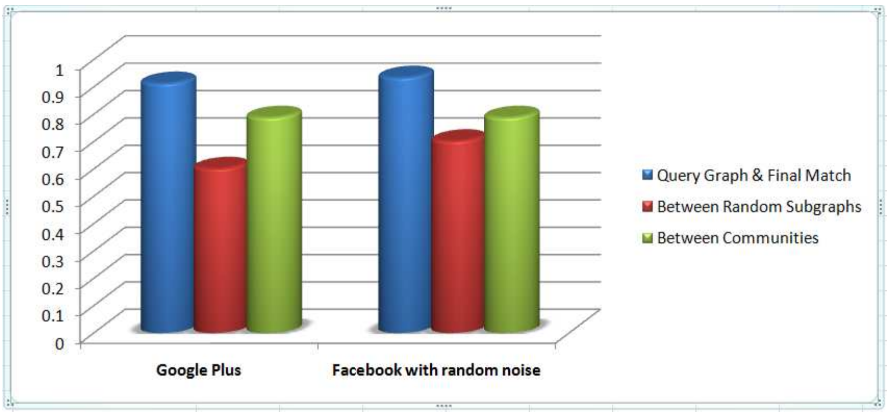


Figure 6.3: Experiment 3: Similarity scores with query graphs that may not be induced subgraphs of target graph

modified network as the target graph and query original facebook communities in this target graph. Here also, we validate the quality of our solution by measuring the similarity score for the query graph with a random induced subgraph of same number of vertices in the target graph. Table 6.6 shows the results. Values shown for both experiments are averaged over 150 scores. The results show that similarity score of our match is close to 1 and is significantly better than a random match.

Data Set	Final Match	Random Subgraph
Google Plus	0.912241	0.600442
Facebook with random noise	0.933662	0.701198

Table 6.6: Experiment 3 : Similarity Scores. Second column shows the similarity score between query graph and match. Third column shows the score between query graph and a random subgraph

Similar to *Experiment 2*, we validate our *Experiment 3* with the help of Figure 6.3. The plot in Figure 6.3 shows high similarity scores with respect to our final matches and query graphs, and relatively less scores for random subgraphs. Thus, our approach extracts an actually similar match to the query graph and not just some random subgraph.

6.3.4 Experiment 4: Dense Subgraph Identification

We use our matching algorithm to identify dense subgraphs in large networks. In particular, we search for dense subgraphs in DBLP and google plus networks. For this, we first generate dense random graphs using the standard $G(n, p)$ model with $n = 500$ and $p = 0.9$. We now use these random graphs as query graphs and query them against the DBLP and google plus networks. We use the standard definition of density ρ of a graph $H = (V, E)$ as

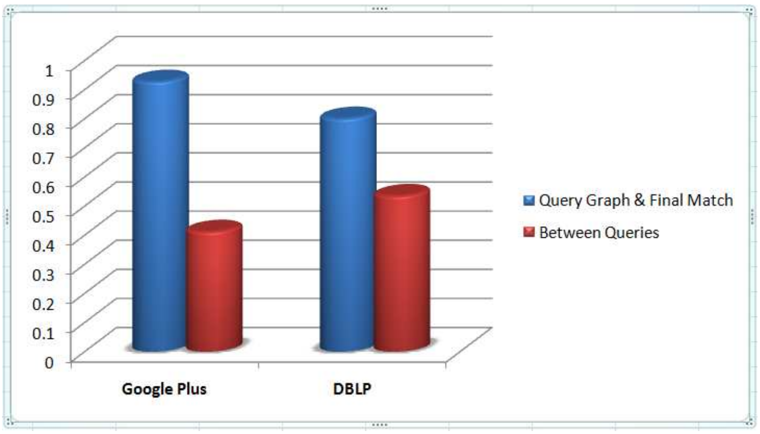


Figure 6.4: Experiment 4 : Similarity scores with respect to dense query graphs

$$\rho = \frac{2|E|}{|V| * |V-1|} \in [0,1] \tag{6.1}$$

The average density of our random query graphs is 0.9. We queried these dense random graphs against DBLP and google plus networks. Table 6.7 shows the results. Column 2 shows the similarity score between query graph and obtained match. Column 3 shows the density ρ for the obtained match. The results are averaged over 150 queries. Results show that the similarity score with matched result is close to 1 for google plus. For DBLP the score is close to 0.8 primarily because DBLP does not have dense subgraphs with about 500 vertices. Also, the density of the obtained match is close to that of the query graph, which is 0.9. Figure 6.4 and Figure 6.5 depict the similarity scores and density results of *Experiment 4* respectively.

Data Set	**Similarity Score**	$\boldsymbol{\rho}$ for the match
Google Plus	0.926670	0.812
DBLP	0.799753	0.730

Table 6.7: Experiment 4 : Dense Subgraph Match Results

6.4 Scalability

Computationally most expensive parts of our algorithm are the vertex label computation for vertices of query and target graphs. Since this is a one time preprocessing for the target graph, it can be easily scaled to a distributed framework using the standard map-reduce paradigm. Vertex label computation for each vertex can be a separate map/reduce job. Vertex label computation for query graph is performed for every search. This can also be parallelized using the standard OpenMP/MPI

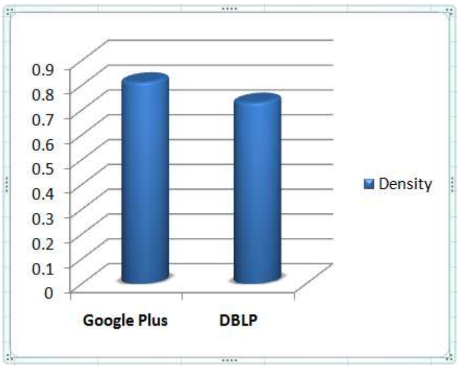


Figure 6.5: Experiment 4 : Density ρ of the obtained matches with respect to dense query graphs

framework as each vertex label computation can be done in parallel. As shown in the experimental results, remaining phases take much lesser time even with serial implementation. Parts of them can also be parallelized to further improve the search efficiency.

Chapter 7

Conclusion and Future Work

Given a large target graph and a query graph, in this work, we implement an effective algorithm to search for an induced subgraph of the target graph that is most similar to the query graph. In this work we use Graphlet Kernel [1] as the similarity measure to determine the similarity of our obtained final match with the query graph. Our approach first identifies a seed match to the query graph and then uses the seed match as a basis to carefully extend it to a final match. To validate the quality of our match, we perform various experiments on large real life data sets including social networks, DBLP collaboration network, amazon network, youtube network and road networks which have tens of thousands of vertices and millions of edges. We also evaluate the computational efficiency of our approach using these data sets.

The major aim of this work is to propose an effective algorithm to retrieve a high quality match for a given query graph. Little focus was laid on improving the computational efficiency of the pre-processing phase, as it is a one time process. Experimental results show that our matching phase is efficient and completes in 1 sec for even networks with tens of thousands of vertices. Improving the pre-processing time will further increase the efficiency of our approach. The pre-processing time can be easily paralellized and distributed using standard map-reduce paradigm or the OpenMP/MPI framework as each vertex label computation can be done in parallel. In this work experiments were performed only to identify dense subgraphs with a given density. This can be further extended to identify given patterns in the target graph. Also instead of using Graphlet Kernel as the similarity measure, further analysis has to be done to use the vertex level labels to define a new more efficient graph kernel.

References

[1] N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S. Vishwanathan. Efficient graphlet kernels for large graph comparison. In International conference on artificial intelligence and statistics. 2009 488–495.

[2] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009 383–392.

[3] D. Haussler. Convolution kernels on discrete structures. Technical Report, Citeseer 1999.

[4] F. Desobry, M. Davy, and W. J. Fitzgerald. A Class of Kernels For Sets of Vectors. In ESANN. Citeseer, 2005 461–466.

[5] R. Kondor and T. Jebara. A kernel between sets of vectors. In ICML, volume 20. 2003 361.

[6] S. Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. *Kernel methods in computational biology* 113–130.

[7] S. Hido and H. Kashima. A linear-time graph kernel. In Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 2009 179–188.

[8] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis* 30, (2011) 129–150.

[9] N. Shervashidze and K. M. Borgwardt. Fast subtree kernels on graphs. In Advances in Neural Information Processing Systems. 2009 1660–1668.

[10] T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In Learning Theory and Kernel Machines, 129–143. Springer, 2003.

[11] H. Kashima and A. Inokuchi. Kernels for graph classification. In ICDM Workshop on Active Mining, volume 2002. Citeseer, 2002 .

[12] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2004 158–167.

[13] M. Neuhaus and H. Bunke. Edit distance based kernel functions for attributed graph matching. In Graph-Based Representations in Pattern Recognition, 352–361. Springer, 2005.

[14] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In Data Mining, Fifth IEEE International Conference on. IEEE, 2005 8–pp.

[15] R. C. Bunescu and R. J. Mooney. A shortest path dependency kernel for relation extraction. In Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2005 724–731.

[16] H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell. Optimal assignment kernels for attributed molecular graphs. In Proceedings of the 22nd international conference on Machine learning. ACM, 2005 225–232.

[17] S. Menchetti, F. Costa, and P. Frasconi. Weighted decomposition kernels. In Proceedings of the 22nd international conference on Machine learning. ACM, 2005 585–592.

[18] J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. In First International Workshop on Mining Graphs, Trees and Sequences. 2003 65–74.

[19] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2002 39–52.

[20] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004 335–346.

[21] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. ACM, 2009 192–203.

[22] M. Mongiovi, R. Di Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *Journal of bioinformatics and computational biology* 8, (2010) 199–218.

[23] S. Zhang, J. Yang, and W. Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *Proceedings of the VLDB Endowment* 3, (2010) 1185–1194.

[24] X. Wang, A. Smalter, J. Huan, and G. H. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In Proceedings of the 12th international conference on extending database technology: advances in database technology. ACM, 2009 472–480.

[25] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011 901–912.

[26] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, (2012) 788–799.

[27] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. IEEE, 2008 963–972.

[28] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Graph similarity search with edit distance constraint in large graph databases. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management. ACM, 2013 1595–1600.

[29] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data` 2014.

[30] DBLP Network. `http://dblp.uni-trier.de/db/`.

[31] K. M. Borgwardt. Graph kernels. Ph.D. thesis, lmu 2007.

[32] J. Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character* 415–446.

[33] B. Schölkopf and A. J. Smola. Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT press, 2002.

[34] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *The Journal of Machine Learning Research* 11, (2010) 1201–1242.

[35] N. Przulj, D. Corneil, and I. Jurisica. Supplementary Information: Efficient Estimation of Graphlet Frequency Distributions in Protein-Protein Interaction Networks .

[36] G. T. Heineman, G. Pollice, and S. Selkow. Algorithms in a Nutshell. " O'Reilly Media, Inc.", 2008.