Analysis of Current Flows in Electrical Networks for Error-Tolerant Graph Matching

by

Alejandro Gutierrez Munoz

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science & Engineering
College of Engineering
University of South Florida

Major Professor: Lawrence O. Hall, Ph.D.
Dmitry B. Goldgof, Ph.D.
Srinivas Katkoori, Ph.D.

Date of Approval:
November 10, 2008

Keywords: graph mining, compound matching, graph kernel, graph dataset, classifier

DEDICATION

Every journey begins with a goal, with a hope. This journey was no different for me. What made it special was the support and encouragement of my wife Edna, my family, and friends. To them, I would like to dedicate these pages that mark the start and not the end of another journey for me. What made it possible was the constant advice and academic guidance of my major professor, Dr. Lawrence O. Hall. For him, I have no other words but to say thank you. What made it all worth it was the personal satisfaction of a job well done. But who made it all happened was God – Thank you Lord for all the good opportunities and people you have crossed in my path.

ACKNOWLEDGMENTS

I would specially like to thank the company that I have been working for during the last five years, Unisource Administrators. They supported my education and provided me all of the necessary tools and time to be able to accomplish this goal. Many thanks to Patrice Say, former VP of Human Resources, who was very supportive of my career during all these years. Special thanks to Andrew Olwert III, President and CEO of Unisource Administrators, who has always believed in me and my work. I would also like to thank Dr. Dmitry B. Goldgof and to Dr. Srinivas Katkoori for serving as part of my committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ANALYSIS OF CURRENT FLOWS IN ELECTRICAL NETWORKS FOR ERROR-TOLERANT GRAPH MATCHING

Alejandro Gutierrez Munoz

ABSTRACT

Information contained in chemical compounds, fingerprint databases, social networks, and interactions between websites all have one thing in common: they can be represented as graphs. The need to analyze, compare, and classify graph datasets has become more evident over the last decade. The graph isomorphism problem is known to belong to the NP class, and the subgraph isomorphism problem is known to be an NP-complete problem. Several error-tolerant graph matching techniques have been developed during the last two decades in order to overcome the computational complexity associated with these problems. Some of these techniques rely upon similarity measures based on the topology of the graphs. Random walks and edit distance kernels are examples of such methods. In conjunction with learning algorithms like back-propagation neural networks, k-nearest neighbor, and support vector machines (SVM), these methods provide a way of classifying graphs based on a training set of labeled instances.

This thesis presents a novel approach to error-tolerant graph matching based on current flow analysis. Analysis of current flow in electrical networks is a technique that uses the voltages and currents obtained through nodal analysis of a graph representing an electrical circuit. Current flow analysis in electrical networks shares some interesting connections with the number of random walks along the graph.

We propose an algorithm to calculate a similarity measure between two graphs based on the current flows along geodesics of the same degree. This similarity measure can be applied over large graph datasets, allowing these datasets to be compared in a reasonable amount of time. This thesis investigates the classification potential of several data mining algorithms based on the information extracted from a graph dataset and represented as current flow vectors. We describe our operational prototype and evaluate its effectiveness on the NCI-HIV dataset.

CHAPTER 1

INTRODUCTION

Several error-tolerant graph matching techniques have been developed over the last two decades. Some of these techniques rely upon similarity measures based on the topology of the graphs [1] [2] [3] [4] [5] [6]. Random walks and edit distance kernels are examples of such methods that in conjunction with learning algorithms like k-nearest neighbors, neural networks, and support vector machines (SVM) provide a way of classifying graphs based on a training set of labeled instances.

This thesis investigates an error-tolerant graph matching technique based on analysis of current flows in electrical networks. Error-tolerant graph matching between two graphs is performed using a similarity measure here proposed. The similarity measure is generated based on current flow vectors extracted from each graph. Current flow vectors are calculated by applying current flow analysis to the graphs which are treated as electrical circuits. Current flow vectors extracted from the graphs capture information about the topology of the graph, information that is later used with the similarity measure to calculate a value between 0 and 1; the greater the value, the more similar the graphs are as defined by the similarity measure.

This thesis is organized as follows. In Chapter 2, we present a brief introduction to graph theory and related approaches for graph comparison. In Chapter 3, we present the concept of electrical nodal analysis for fast discovery of connection subgraphs as proposed by Faloutsos et al. Nodal analysis applied to undirected graphs is at the heart of our current flows for error-tolerant graph matching approach, and as shown by [7] [8], in a graph where the edge weights represent the conductance of the edge and vertices represents the nodes of the circuit, the electrical current along an edge is proportional to the net number of times that a random walk along the same edge will traverse it. In Chapter 4, we present current flow analysis for the error-tolerant graph matching approach, where several geodesics (shortest paths) are extracted from the graph using as starting and ending points nodes of equal degree. Two sets of geodesics are then evaluated: shortest geodesics and longest geodesics. Current flow analysis is then performed over the two sets of geodesics in order to produce an n-dimensional vectorial representation of the graph. Chapter 4 also introduces a similarity measure based on the n-dimensional vectorial representation of the graphs generated using current flow analysis. This new similarity measure is a function $k : G \times G \rightarrow R$, where two graphs represented by their current flow vectors are compared to each other, and a real number between 0 and 1 is returned as the similarity value between the two graphs. As we will observe, this similarity measure can be used as a kernel in a support vector machine, since k is symmetric and non-negative, it can make up a positive definite matrix [9]. In Chapter 5, we describe the implementation details of our prototype, which consists of two main programs: CF-vectors and CF-compare.

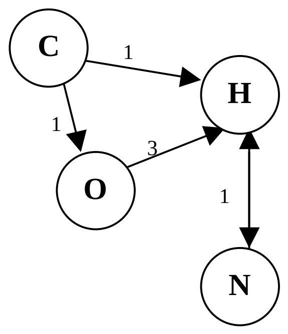
CF-vectors is the program used to generate the vectorial representation of the graphs using current flow analysis. CF-compare is the program used to compare two set of graphs and generate a similarity value between each pair of graphs among both sets. Other tools were developed as part of this thesis in order to facilitate the analysis and visualization of the results. These tools are: sdf2gds and gds2dot. Both of these tools are used to transform the file format used by CF-vectors and CF-compare to a more standard format. Chapter 6 describes the results obtained on the NCI-HIV dataset [10]. Some examples of different chemical compounds represented as graphs, and their closest matches are presented in order to provide a graphical comparison between them. As we will show, the ability to store the graph information as a current flow vector, and later, use this representation to find similar graphs, based on the topology, is quite useful. Finding the best match using our similarity measure against a database of more than 40,000 compounds takes a few seconds, and once the current flow vectors have been calculated and stored there is no need to calculate them again as they will remain unchanged for each graph. Chapter 7 presents a summary and ideas for future research using the technique here proposed.

CHAPTER 2

BACKGROUND AND RELATED WORK

Graphs offer a powerful way to represent structured data. Several applications where graph representation is used like shape analysis, character recognition, and chemical compound matching take advantage of the benefits of graph databases. The ability to compare two graphs represents an important contribution to the area of graph mining. Graph matching usually refers to comparing the structural similarity between two or more graphs. Graph matching approaches are mainly divided in two classes: exact graph matching and error-tolerant graph matching [11].

Let us define a graph g as a four-tuple g = (V,E,u,v), where V denotes a finite set of nodes or vertices, E denotes a finite set of edges, where $E \subseteq V \times V$, u denotes a node labeling function, and v denotes an edge labeling function. Fig. 2.1 shows sample graph g with its correspondent four-tuple.



V = {1,2,3,4}
E = {(1,2), (1,3), (2,3), (3,4), (4,3)}

$$u(x) = \begin{cases} x = 1 & C \\ x = 2 & O \\ x = 3 & H \\ x = 4 & N \end{cases} \qquad v(x,y) = \begin{cases} x = 1, y = 2 & 1 \\ x = 1, y = 3 & 1 \\ x = 2, y = 3 & 3 \\ x = 3, y = 4 & 1 \\ x = 4, y = 3 & 1 \end{cases}$$

Figure 2.1 Sample graph g

Graphs can be classified into two main categories: directed and undirected. Undirected graphs are those where for every edge $(u,v) \in E \Rightarrow (v,u) \in E$. Directed graphs on the contrary are those where there is at least one edge $(u,v) \in E$ such that $(v,u) \notin E$. A subgraph g1 of g2 is a graph such as that for graph g1 = (V1, E1, u1, v1) and g2 = (V2, E2, u2, v2), graph g1 $\subseteq$ g2. A graph is a subset of another graph if it posses the following characteristics:

1. $V1 \subseteq V2$
2. $E1 = E2 \cap (V1 \times V1)$
3. $u1(u) = u2(u)\ \forall u \in v1$
4. $v1(u,v) = v2(u,v)\ \forall (u,v) \in E1$

Based on the definitions of graph and subgraph we can now define exact and error-tolerant graph matching. In exact graph matching the objective is to identify whether all vertices, edges, node labels, and edge labels are identical between two graphs. This is called graph isomorphism. The most common approach to check graph isomorphism is to traverse a search tree checking all node-to-node correspondences. The computational complexity of the search tree procedure is exponential in the number of nodes of both graphs [11]. A similar problem for graph isomorphism is the problem of finding subgraph isomorphism. In other words, to detect if a smaller graph is part of a bigger graph. A subgraph isomorphism between graphs g1 and g2 exists if the larger graph can be transformed into the smaller graph by removing some nodes and edges. The subgraph isomorphism problem belongs to the NP-complete class of computation.

Due to its computational complexity the exact graph matching problem is not implemented in real scenarios, the error-tolerant graph matching problem is more suitable for larger graph databases and graphs with a high number of vertices. Several approaches have been proposed for the error-tolerant graph matching problem [1] [2] [3] [4] [5] [6]. Some of these approaches are based on similarity measures like the graph edit distance, where a list of edit operations is performed in order to transform one graph into another. Edit operations can be edge edit operations or vertex edit operations. An example of an edit vertex operation is to add or remove a node from one graph in order to find a correspondence on the other graph. The concept of graph edit distance is then presented as the cost of the edit path. Each edit operation can be assigned a specific value. For example, removing a vertex could be more costly than changing an edge label. Other error-tolerant graph matching approaches include walk-based graph kernels [20]. Walk-based kernels are defined for directed labeled graphs. The process of mapping a graph to multisets of label sequences, or walks, is what is known as a walk kernel. Cycle pattern kernels (CPK) [20] are based on the idea of mapping graphs into a selected group of cycles and trees. Another approach is to map a graph to a set of frequent subgraphs (FSG) previously indentified from the training dataset [17] [18].

The following chapters will introduce current flow analysis for error-tolerant graph matching as an approach in which graphs are transformed into a vector of current flows over particular paths of the graph in order to compare the current flow vectors between graphs of a graph database.

CHAPTER 3

CURRENT FLOW ANALYSIS

The flow of electrical currents in a network of resistors can be measured between any two nodes in an electrical network. Current flow analysis combines Ohm’s law and Kirchhoff’s current law to determine the voltage values at each node along the electric circuit. Nodal voltage analysis of electrical circuits is performed by solving a system of equations in which the unknowns are the voltages at different nodes in the circuit. Current flow along the various branches of the circuit can be determined based on the voltages at each node in the circuit. The analysis of the current flows between specific pairs of nodes in a graph is at the heart of our error-tolerant graph matching technique.

In [7] an approach related to electrical currents in a network of resistors was proposed. This approach tries to solve the problem of finding a connection subgraph that can deliver as many units of electrical current as possible. For this purpose, a graph $G = (V, E)$ is treated as an electrical network, where edge weights represent conductance ($C(u,v)$ represents the conductance between nodes u and v), and the vertices represent the nodes of the electrical circuit ($V(u)$ represents the voltage at node u). The voltages at each node of the circuit are calculated by combining Ohm's law and Kirchhoff's current law.

$$\forall u,v : I(u,v) = C(u,v)(V(u) - V(v)) \quad (1)$$

$$\forall v \neq s,t : \sum_u I(u,v) = 0 \quad (2)$$

Having s as the source node, and t as the target node, equations (1) and (2) determine the voltages and currents as the solution to the following linear system:

$$V(u) = \sum_v \frac{V(v)C(u,v)}{C(u)} \quad \forall u \neq s,t \quad (3)$$

$$V(s) = 1,\ V(t) = 0 \quad (4)$$

$$C(u) = \sum_v C(u,v) \quad (5)$$

solving (3) with boundary conditions (4) will determine the voltages at each node. $C(u)$ represents the total conductance of node u, this is, the sum of all edge weights adjacent to u.

Once the current, $I(u,v)$, values are available, the current along a particular path: $\widehat{I}(P),\ P = \{s, \ldots, t\}$ is defined as the pro-rated current along that path from source to target.

$$\widehat{I}(s,u) = I(s,u) \quad (6)$$

$$\widehat{I}(s = u_1, \ldots, u_i) = \widehat{I}(s = u_1, \ldots, u_{i-1}) \frac{I(u_{i-1}, u_i)}{I_{out}(u_{i-1})} \quad (7)$$

$$I_{out}(u) = \sum_{\{v \mid u \to v\}} I(u,v) \quad (8)$$

where $I_{out}(u)$ represents the total current leaving a node, which is equal to the sum of all currents leaving the node in a downhill stream, where a downhill stream from node u to node v means that voltage at node u is higher than voltage at node v, $V(u) > V(v)$.

Since the idea of the approach presented in [7] is to find the best connection subgraph, the concept of captured flow $CF(H)$ is introduced. $CF(H)$ of a subgraph *H* of *G* is the total delivered current, summed over all paths from source *s* to target *t* that belong to *H*. For the purpose of this thesis, we are only considering single paths in *G*, not subgraphs of it. The concept of delivered current over a path, $\widehat{I}(P)$, is very important in the calculation of the current flow vectors in the next chapter.

The following example illustrates the process of calculating the voltages, currents, and current flows of graph in Fig. 3.1 which can be treated as an electrical circuit.

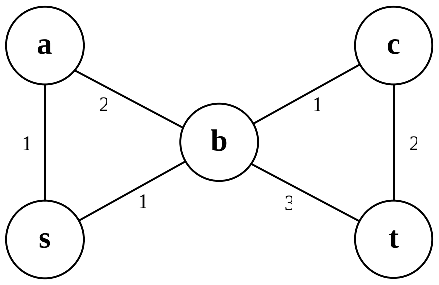


Figure 3.1 A sample graph to calculate current flows

Voltages at the source and target nodes have been fixed to 1 and 0 respectively, $V(s) = 1, V(t) = 0$. Any vertex of degree 1, this means having only one edge connected to the vertex, excluding source and target nodes will have a voltage of 0. This is equivalent to a ground node in an electrical circuit; therefore a voltage of 0 is assigned to represent a sink node in the circuit. Once the voltages for source node and ground nodes have been specified we can proceed to solve a system of linear equations with *n* variables, where *n* is equal to the total number of vertices in the graph excluding the source and ground nodes, in this case *n* is 3. This system can be reduced to solving an eigenvector calculation of the form:

$$\begin{bmatrix} A & B \\ 0 & I \end{bmatrix} V = V \tag{9}$$

$$A = \left\{ a_{ij} = \frac{w_{ij}}{w_j} \right\} \tag{10}$$

where matrix *A* represents the relationship between the nodes based on their connection weights, *B* represents the boundary conditions for *s*, *t*, and other ground nodes, and *I* is the identity matrix of size $n \times n$. The solution to the system of equations represents the voltages at each of the *n* nodes.

$$\begin{bmatrix} -\sum_i w_{ai} & w_{ab} & w_{ac} \\ w_{ba} & -\sum_i w_{bi} & w_{bc} \\ w_{ca} & w_{cb} & -\sum_i w_{ci} \end{bmatrix} V = \begin{bmatrix} -w_{sa} \\ -w_{sb} \\ -w_{sc} \end{bmatrix} \Rightarrow \begin{bmatrix} -3 & 2 & 0 \\ 2 & -7 & 1 \\ 0 & 1 & -3 \end{bmatrix} V = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} \tag{11}$$

$$V = \begin{bmatrix} 0.54 \\ 0.31 \\ 0.10 \end{bmatrix} \Rightarrow V(a) = 0.54,\ V(b) = 0.31,\ V(c) = 0.10 \tag{12}$$

With the voltages for each node we can now calculate the current for each edge using equation (1). Once we have calculated the current along each edge, we need to calculate the current flow along each possible path between source and target. This is done using equations (6) and (7). The current flow is a pro-rated amount from source to target based on the total current along each node in the path and the total current leaving each of the nodes along the path.

Now we proceed to calculate the current along each path between *s* and *t*. Based on the current flow only downhill paths can be calculated. A downhill path from node *u* to node *v* means that voltage at node *u* is higher than voltage at node *v*, $V(u) > V(v)$:

$$\begin{aligned}
&I(s,a) = C(s,a)(V(s) - V(a)) = 1\times(1-0.54) = 0.46\\
&I(s,b) = C(s,b)(V(s) - V(b)) = 1\times(1-0.31) = 0.69\\
&I(a,b) = C(a,b)(V(a) - V(b)) = 2\times(0.54-0.31) = 0.46\\
&I(b,c) = C(b,c)(V(b) - V(c)) = 1\times(0.31-0.10) = 0.21\\
&I(b,t) = C(b,t)(V(b) - V(t)) = 3\times(0.31-0) = 0.93\\
&I(c,t) = C(c,t)(V(c) - V(t)) = 2\times(0.10-0) = 0.20
\end{aligned} \tag{13}$$

the following figure shows all voltages and currents, as well as the flow of the current based on the voltages.

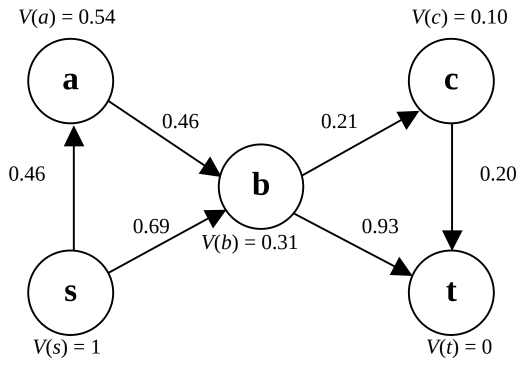


Figure 3.2 A sample graph with voltages and currents

Having the values of all currents along each edge we can proceed to calculate the current along a particular path, $\hat{I}(P),\ P=\{s,\ldots,t\}$. Using equations (7) and (8) the pro-rated amount of current that flows from node *s* to node *t* is calculated. Table 3.1 shows the values of the current flow along different paths from *s* to *t*:

Table 3.1 Current flow along paths in Fig. 3.2

$s \to b \to t$	$0.69\times\frac{0.93}{0.21+0.93}=0.56$
$s \to b \to c \to t$	$0.69\times\frac{0.21}{0.21+0.93}\times\frac{0.20}{0.20}=0.13$
$s \to a \to b \to t$	$0.46\times\frac{0.46}{0.46}\times\frac{0.93}{0.21+0.93}=0.38$
$s \to a \to b \to c \to t$	$0.46\times\frac{0.46}{0.46}\times\frac{0.21}{0.21+0.93}\times\frac{0.20}{0.20}=0.08$

as we can observe from Table 3.1 path $s \to b \to t$ is the one that delivers the most current from *s* to *t*. As we mentioned before, in [7], the concept of captured flow is introduced to denote the current flow along selected paths of graph *G* forming subgraph *H*, *CF(H)* denotes the sum of all the current flows along each path from H. The idea behind the captured flow in subgraph *H* was to identify the subgraph that will deliver the most current relative to the number of nodes being added to *H*. For our purposes we will not consider this concept as we are interested in current flows along several single paths depending on the characteristics of the source and target nodes.

CHAPTER 4

CURRENT FLOW VECTORS

Current flow analysis along the shortest and longest geodesics (shortest paths) of a graph as presented in [12], provides a method to describe the graph structure such that the information needed to represent the graph is reduced significantly compared to the original size of the graph representation. Once a graph has been described using current flow analysis, its new representation is an n-dimensional vector that stores the current flow along geodesics of different node degrees.

$$G = (V, E) \qquad \Delta(G) = \max_{v \in V} \deg(v) \tag{14}$$

$$f : G \to R^{2\Delta(G)},\ 2\Delta(G)\ \textit{is an upper bound}. \tag{15}$$

As we can observe from (15), the current flow vector is represented by function f, which transforms the input graph G into a $2\Delta(G)$-dimensional vector, where $\Delta(G)$ represents the highest node degree among all vertices in the graph. The actual dimension of the vector is twice the size of the maximum degree; this is due to the analysis of the current flow along shortest and longest geodesics of the graph. The size of the vector when based on the highest vertex degree actually represents an upper bound of the final vector size; this is because for a given graph, some geodesics for a specific node degree may not exist resulting in a current flow vector of lower dimensionality.

In the following sections we will describe the steps needed to perform the transformation from a graph representation $G = (V, E)$, to a vectorial representation $G = R^n$. Section 4.1 describes the process of geodesic selection, also called Group-N generation. In Section 4.2, we describe the steps needed to calculate the current flow along each of the selected geodesics. Nodal analysis is used as shown in Chapter 3 in order to calculate the pro-rated current along geodesics.

4.1 Group-N generation

Current flow analysis requires the selection of voltage source *s* and target ground *t* nodes. Once the selection of these nodes has been done, boundary conditions as described in (4) can be applied to solve the linear system in (3) to find the voltages and currents of the circuit. Path selection is made using shortest paths (geodesics) along the graph. Different source and target nodes will provide different paths, hence different current flows along each path. Each current flow along a particular path will capture different characteristics of the topology of the graph as different connections and flows along each path will differ from each other. The idea then is to select a representative number of paths that will capture as much information as possible about the topology of the graph using current flows. To this end, two different sets of paths are defined: shortest geodesics and longest geodesics.

Since different graphs will render different geodesics, we need a way to pair them when comparing them. A good way to describe the characteristics of a geodesic is based on the node degree of its source and target nodes. In order to provide a standard framework of comparison between current flows along geodesics of different graphs, the selection of geodesics is limited to those in which the source and target nodes have the same node degree.

The concept Group-N encompasses the group of geodesics that share the same degree N in their source and target nodes. Each Group-N will have two sets of geodesics: shortest geodesics and longest geodesics. As noted before, by selecting a representative number of paths along the graph we are providing a way to capture as much information as possible about the topology of the graph. Fig. 4.1 and Table 4.1 provide an example of a graph and its corresponding Group-N shortest and longest geodesics.

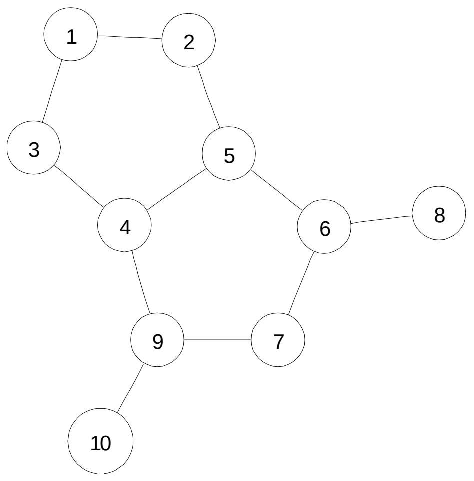


Figure 4.1 A simple graph used in Table 4.1 for Group-N generation

Geodesics in Group-1 are those where their source and target nodes are of degree 1, in this case, nodes 8 and 10. The same applies for other Group-Ns. As we can see, multiple geodesics of the same length of the same group-N can be generated using different source and target nodes. Current flows along these geodesics are averaged to produce a single current flow value for each group-N set. For example, Group-2 shortest geodesics are (1,2) and (1,3), both with the same path length; when calculating the current flow for Group-2, both current flows are calculated, current flow between node 1 and node 2 and current flow between node 1 and node 3. The resulting current flow values are then averaged to produce a single value for Group-2.

Table 4.1 Group-N geodesics for graph in Fig. 4.1

Group-N	**Shortest Geodesic**	**Longest Geodesic**
Group-1	(8,10)	(8,10)
Group-2	(1,2), (1,3)	(1,7)
Group-3	(4,5), (4,9), (5,6)	(4,6), (5,9), (6,9)

As mentioned before, the selection of the geodesics is done using a single-source shortest path algorithm from source to target. In this case we are using Dijkstra's algorithm [13]. It is important to note here that edge weights in the graph represent the cost (or resistance) of going from one node to another. This annotation is important in the sense that while calculating the current flows using equations (1), (2), (3), and (4) the value of the edge weights represent the conductance between the nodes rather than resistance. Therefore a conductance equal to the reciprocal of the edge weight is used while performing the nodal analysis.

4.2 Current flow along Group-N geodesics

Now that we have described how to generate the Group-N sets for a given graph, we can proceed with the current flow calculation along each of the geodesics. Calculating currents is done using nodal analysis as described in Chapter 3. Each geodesic can be described as a path from source to target. $P = (s, \ldots, t)$. As noted by equation (4) voltage values for source and target nodes are initialized to $V(s)=1,\ V(t)=0$. In this scenario, the source will act as the voltage source and the target as a ground. In the event the graph has some nodes of degree 1 that are neither the source nor the target, these nodes are considered to be grounds as well, hence the voltage at these nodes is $V(u)=0$.

Once the voltages for source nodes and ground nodes have been specified we can proceed to solve a system of linear equations with *n* variables, where *n* is equal to the total number of vertices of the graph minus source and ground nodes. The solution to the system of equations represents the voltages at each of the *n* nodes. With the voltages for each node, we can now calculate the current for each edge using equation (1). Once we have calculated the current along each edge, we need to calculate the current flow along the geodesic. This is done using equations (6) and (7). The current flow is a pro-rated amount from source to target based on the total current along each node in the path and the total current leaving each of the nodes along the path. Having calculated the current flow along each geodesic we can calculate the current flow for each Group-N set. For each set (shortest and longest) we average the current of all geodesics of the same degree. This is, for Group-Ns with more than one geodesic of the same length we calculate an average current flow. Having calculated all of the Group-Ns current flows for shortest and longest geodesics, we can produce our vectorial representation of graph *G*:

$$G = R^{2D} \qquad D = \max_{n \in GroupN} \deg(GroupN) \tag{16}$$

the vectorial representation of the graph is defined by:

$$G = \left[SN_1, \ldots, SN_D, LN_1, \ldots, LN_D\right] \tag{17}$$

where SN_i is the average current flow value for the shortest geodesic(s) from Group-i. Similar, LN_i is the average current flow value for the longest geodesic(s) from Group-i. If a particular degree is not represented in the Group-Ns, a value of 0 is assigned to the current flow for that group.

4.3 Current flow under special scenarios

Certain considerations need to be taken into account in order to assure that nodal analysis will yield useful results. For disjoint graph representations where certain sections of the graph are not connected to each other, we need to exclude the nodes where there is no path from the source to the node. This can be accomplished by using BFS (Breath-First Search) [13]. This will prevent calculating currents along non-existing connections in the circuit. Fig. 4.2 shows a graph where nodes 0S, 1O, 2O, and 4O are not connected to the rest of the graph. By running BFS to determine if there is a path from source to any of these nodes the algorithm can decide whether or not to calculate the current flow.

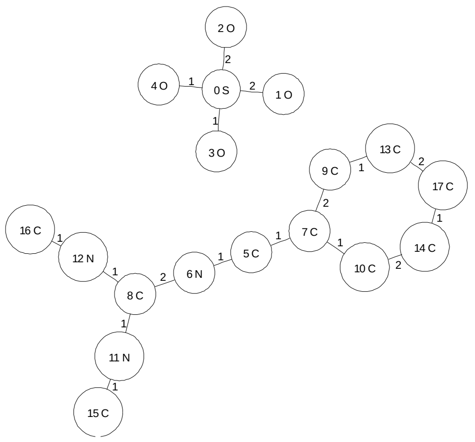


Figure 4.2 Graph with non-connected nodes

On the other hand, certain topology configurations of a graph, in particular where too many ground nodes (nodes of degree 1) are present, and closed rings (cycles) provide alternative routes from the current to flow from source to target avoiding the extra ground nodes, display the potential for an odd distribution of voltages along the geodesic; i.e. voltages along the nodes of the path will not always be in a descending configuration from source to target, causing the current flow calculation to yield negative results. To avoid this scenario, we opted to exclude any node pair that causes this behavior from the pro-rated calculation of the current flow as presented in equation (7). This situation is analogous to short-circuiting an electrical network.

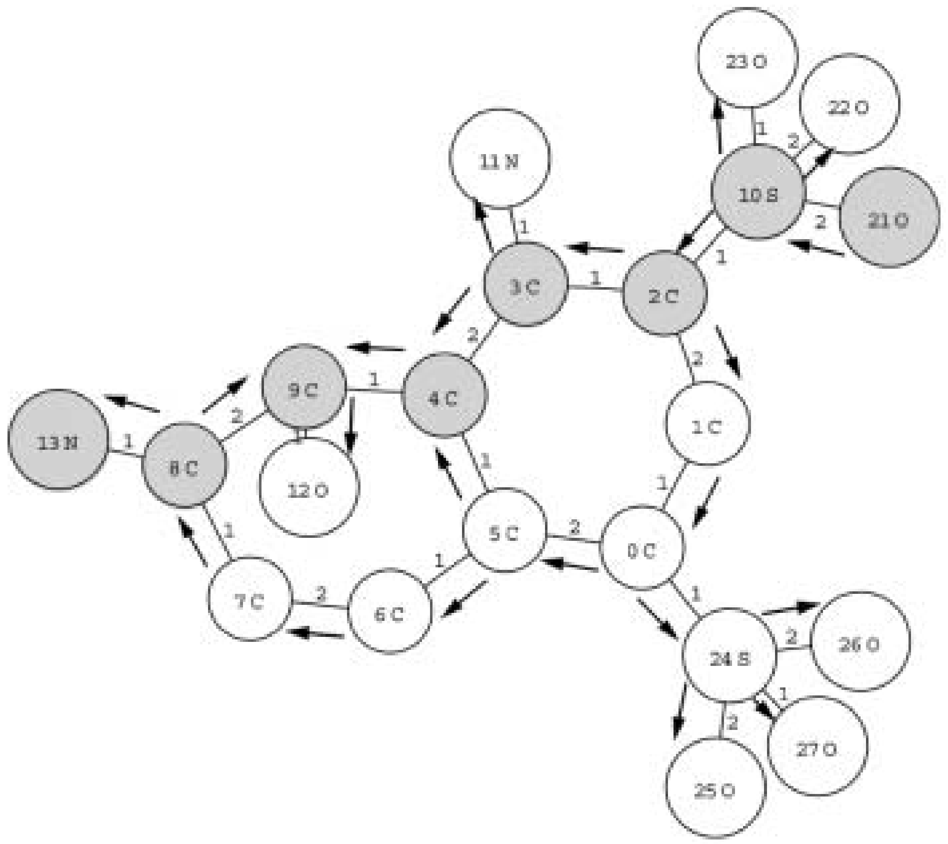


Figure 4.3 Current flow along path causing short-circuit

Fig. 4.3 shows an example of a scenario where a path from source node $s = 21O$ to target node $t = 13N$, flows in a downhill stream (as defined in Chapter 3) until it reaches node 9C. Since node 9C is connected to a ground node (degree of node 12O is 1), the current flows down to this node. Current also flows through a closed ring to reach target node 13N through node 8C. As we can see, since the current flows from 8C to 9C, if we try to calculate the current along the path (grayed out nodes), we would get negative results. As noted, when a scenario like this one arises, we opted for ignoring the portion of the current between nodes 8C and 9C, and short-circuit the network from 4C to 8C.

4.4 Current flow with nodal information

So far the information about the graph being captured using current flow analysis has been limited to shortest and longest geodesics between nodes of same degree. Current flow analysis has only used edge weight information as a conductance equal to the reciprocal of the edge weight to generate the current flow vectors. No information about the vertex labels has been included in any of the calculations. An extension to the technique investigated by Faloutsos et al. in [7] is here proposed. In order to include nodal information, this is, to take into account the label associated with each vertex, we perform an edge weight modification to each edge of the graph based on the vertices that such edge connects. Let us say that for the graph on the left of Fig. 4.4 we want to modify the edge weights based on the vertex labels. We can arbitrarily assign to each label a different value. For example, label C = 1, label O = 0.5, label H = 0.8, and label N = 0.3.

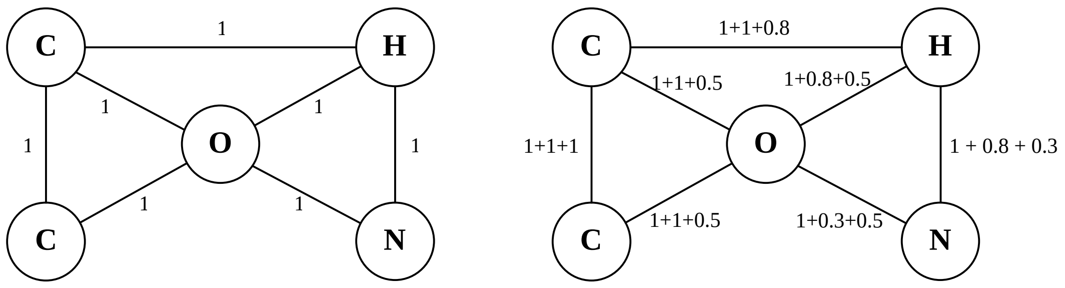


Figure 4.4 Edge weight modification based on vertex label information

As we can observe for the graph on the right of Fig. 4.4 all edge weights are now different. Each edge weight is modified depending upon what vertices it was connecting. By modifying the edge weights based on the vertex information we are trying to incorporate into the current flow calculation some vertex information. Since the current flow calculation is based entirely on the edge weights treated as a network of resistors, by modifying each edge weight we are modifying the current flow along each path. Now the question is how to assign a numeric value to each vertex label? What happens if there are a large number of vertex labels? How big or how small should the values added to the edge weights be compared to the original edge weights? All these questions are better answered based on the characteristics of the graphs to be compared. For example, in our case we will be working with the NCI-HIV dataset. This dataset contains 42,689 chemical compounds that are represented as graphs. The vertex labels are elements of the periodic table; in other words, the number of labels relative small. For our purposes we opted for assigning a value to each vertex based on the frequency of the element in the whole dataset. Elements such as carbon (C), oxygen (O), and nitrogen (N) were the most common; elements such as aluminum (Al) were less common in the whole dataset. For elements with high frequency the value assigned was lower compared to those with less frequency. The idea here is that the most common elements will not provide as distinctive characteristics about the graph topology as those that are more unique. The proportion of the edge weight to the smallest and/or to the largest original edge weight is also important as we do not want the information about the vertices make the original edge weight less important. A percentage of the smallest original edge weight is recommended. For example, if the original edge weights are 1, 2 , and 3; the least common vertex label will be assigned a percentage of the smaller original edge weight, in this case 1. The percentage can be a 10%. For example, the value to be added to edges that connect vertices with the least common label will be of 0.1. This value will be smaller for the next least common label up to the point where the addition to the edge weight will be 0 (the most common label).

Other approaches to include vertex label information into the current flow calculation are also valid. For example, in the case of the NCI-HIV dataset, instead of using the frequency of the labels in the dataset we could have decided to assign similar values to elements with similar chemical characteristics. Other examples of incorporating vertex label information into the current flow calculation will be for computer images represented as graphs; where a color segmentation algorithm can be performed on the image to segregate it into larger sections of similar color that then will be connected with each other; this will construct a graph of color sections. The vertex label for each section will be the color associated with it. Similar colors will be then assigned similar numeric label values; this will allow the current flow analysis to incorporate color information while comparing images represented as graphs.

4.5 Current flow vectors' similarity measure

The representation of the graph topology as an n-dimensional vector allows us to define a similarity measure between two graphs as a numeric value in the range of [0..1], $k : G \times G \to R$. We define the similarity measure *k* between graphs *G*1 and *G*2 as:

$$S = \sum_{d=1}^{D} \frac{|\, G1[SN_d] - G2[SN_d] \,|}{G1[SN_d] + G2[SN_d]} \tag{18}$$

$$L = \sum_{d=1}^{D} \frac{|\, G1[LN_d] - G2[LN_d] \,|}{G1[LN_d] + G2[LN_d]} \tag{19}$$

$$k(G1, G2) = 1 - \frac{S + L}{2D} \tag{20}$$

$$D = \max(\max \deg(G1), \max \deg(G2)) \tag{21}$$

the value of *k* is a real number in [0..1], where the closer the value is to 1, the more similarities are shared between the current flows vector of both graphs. Equation (20) can be described as computing the differences between each pair of Group-N geodesics from graphs *G*1 and *G*2. The first summation *S* compares the shortest geodesics from both graphs, while the second summation *L* compares the longest geodesics from both graphs. As mentioned before, function *k* can be used as a graph kernel. A positive definite Gram matrix *K* can be constructed from function *k*, given that *k* is always positive and symmetric the function *k* can be referred as positive definite (pd) kernel [9].

In the following chapter we will present the implementation details of the similarity measure *k* from the current flow vector creation process to the error-tolerant graph matching approach using the current flow vectors.

CHAPTER 5

IMPLEMENTATION DETAILS

During the implementation of our prototype we developed two main programs: CF-vectors and CF-compare. CF-vectors is the program used to generate the vectorial representation of the graphs using current flow analysis as described in Chapter 4; CF-compare is the program used to compare two set of graphs and generate a similarity value between each pair of graphs among both sets as described in Section 4.5.

5.1 File formats

During the course of our development we defined two file formats to be used by our programs; these are: graph dataset file (.gds) and current flow vectors dataset file (.cfv).

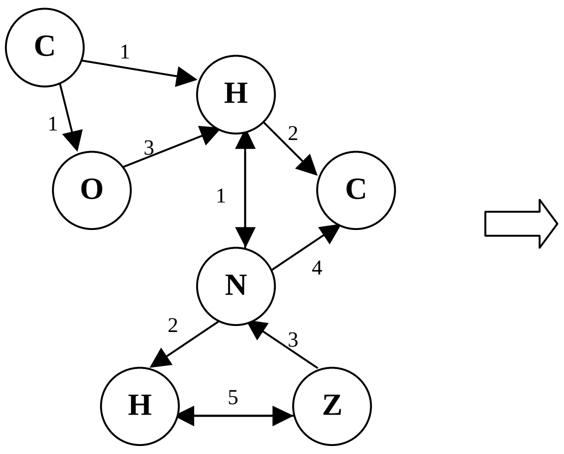
5.1.1 Graph dataset file

The graph dataset file stores a set of directed graphs as described in the next abstract grammar:

```
Graph Dataset: Graph+
Graph: BEGIN graph_name graph_class
             vertices
             edges
         END
vertices: {v vertex_id vertex_label}+
edges: {e from to edge_weight}*
```

Figure 5.1 Graph dataset file abstract grammar

An example of a .gds file representing graph G1 is presented in Fig. 5.2.



Graph G1 – Class: CA

```
BEGIN G1 CA
v 1 C
v 2 O
v 3 H
v 4 N
v 5 C
v 6 H
v 7 Z
e 1 2 1
e 1 3 1
e 2 3 3
e 3 4 1
e 3 5 2
e 4 3 1
e 4 5 4
e 4 6 2
e 6 7 5
e 7 4 3
e 7 6 5
END
```

Figure 5.2 Graph dataset file

5.1.2 Current flow vectors dataset file

The current flow vectors dataset file stores the vectorial representation of the graphs as described in the next abstract grammar:

```
Graph Dataset: Graph+
Graph: graph_name graph_class ([S+ L*] | [S* L+])
S: S:degree:current_flow_value
L: L:degree:current_flow_value
```

Figure 5.3 Current flow vectors dataset file abstract grammar

The current flow vectors dataset format allows for a sparse representation of the current flows. As noted before, not all Group-N degrees will be present in a graph. Only those degrees present in the graph need to be stored in the .cfv files. Fig. 5.4 shows a .cfv file.

Graph	Class	Group-N	Shortest Geodesic	Longest Geodesic
G1	CM	Group-1	0.85467	0.85467
G1	CM	Group-3	0.44655	0.78462
G2	CA	Group-2	0.34677	0.56677
G2	CA	Group-3	0.35477	0.97887
G3	CI	Group-5	0.67878	0.00779

Group-Ns for graphs G1, G2, and G3

```
G1 CM S:1:0.85467 S:3:0.44655 L:1:0.85467 L:3:0.78462
G2 CA S:2:0.34677 S:3:0.35477 L:2:0.56677 L:3:0.97887
G3 CI S:5:0.67878 L:5:0.00779
```

Figure 5.4 Current flow vectors dataset file

5.2 CF-vectors' implementation

CF-vectors was implemented on ANSI C++ using the Template Numerical Toolkit (TNT), which is a collection of interfaces and reference implementations of numerical objects useful for scientific computing in C++ [14]. CF-vectors receives as a parameter a .gds file, and returns as output a .cfv file:

```
$> cf-vectors --help
usage: cf-vectors <inputfile.gds>
```

Figure 5.5 CF-vectors' command line

Once the translation from the graph representation to the vectorial representation of the graph has been performed using CF-vectors, there is no need to perform this step again on the same dataset. The following section shows a pseudocode version of the CF-vectors program.

5.2.1 CF-vectors' algorithm

```
CF-vectors(gdsfile) {

  // array used to store all graphs
  array GraphDataset;

  // array that stores current flows vectors
  array TempCurrents;

  // associative array that holds the counts for each vertex label
  array LabelCounts;

  // global variable that stores the minimum resistance value in the
  // whole dataset
  double MinResistance;

  // output file
  file cfvfile;

  For each graph in gdsfile {

    // extract each graph from gdsfile and add it to the dataset
    GraphDataset.add(graph);

    // count the vertex labels and store the values
    // for example, if graph has 3 vertices with label "C"
    // and 1 with label "N", LabelCounts will add to the
    // overall count of C, 3, and to overall N count, 1.
    LabelCounts.count_labels(graph);

    // find the minimum resistance value in the graph
    // and keep it if it is lower than the current
    // MinResistance value for the whole dataset
    if graph.min_edge_weight() < MinResistance {
      MinResistance = graph.min_edge_weight()
    } // end if

  } // end For each

   // Order the label counts from most common to least common.
   LabelCounts.ReverseSort();

   // Find the amount to be added per each vertex label to the
   // edge weights as described in section 4.4.
   double ResistanceIncrement;
   ResistanceIncrement = MinResistance * 0.1 / LabelCounts.size();


   // the more common the label, the less resistance increment
   For i = 1 to LabelCounts.size() {
     LabelCounts[i].resistance_increment = ResistanceIncrement * i;
   }


   // Get the current flow vector for each graph
   For each graph in GraphDataset {

```

Figure 5.6 CF-vectors' algorithm

```
57       TempCurrents = graph.GetCurrentFlows();
58
59       // write current flow vector to output
60       cfvfile.write(TempCurrents);
61
62     }
63
64     return cfvfile;
65
66 }
```

Figure 5.6 (continued)

The CF-vectors' algorithm receives as a parameter the graph dataset file, line 1. In line 19, each graph inside the graph dataset is processed in order to extract the current flow vector and store it in the output file. In line 28, a global variable used to store the frequency of each vertex label is updated; this section relates to the inclusion of nodal information into the current flow calculation. In lines 33-35 another global variable is modified, the MinResistance variable is used to store the lowest edge weight value in the whole dataset. As described in Section 4.4, the approach to include nodal information into the current flow calculation is to use a percentage of the lowest edge weight value to add to each edge weight depending on the vertex labels it connects. In this case we are using a 10% of the minimum resistance value. Lines 40-51 calculate the appropriate resistance increment value for each vertex label depending on its frequency. The most common label will be at the top of the LabelCounts array after this has been sorted in reverse order, line 40. Starting with the most common label the resistance increment value increases in a proportion equal to the number of labels in the dataset. For example, if there are only 20 different labels in the whole dataset, and the minimum edge weight is 1, then each increment will be 0.1/20 greater than the previous one; with the least common label getting a resistance increment of 0.1. The following algorithm shows the current flow calculation that is done in lines 55-62.

```
 1  graph::GetCurrentFlows() {
 2
 3    array Geodesics;
 4    vertex source;
 5    vertex target;
 6    array CachedCF;
 7    array GroupNcurrents;
 8    array I;
 9    array Iout;
10    double Itemp;
11
12    // Using Dijkstra’s algorithm to find shortest path between all
13    // Group-N pairs of the graph. The Geodesics variable stores both
14    // shortest and longest geodesics of the graph for each
15    // Group-N.
16    Geodesics = this.GetGeodesics();
17
18    // Using the global LabelCounts values modify the edge weights prior to
19    // calculating the current flows
20    For each v in this.vertices {
21      For each e in v.adjency_list {
22          e.weight += LabelCounts[v.label].resistance_increment;
23      }
24    }
25
26    // Calculate the current flows for all geodesics
27    For each x (shortest or longest) geodesic in Geodesics {
28      For each group_i in Geodesics.GroupN {
29        For each g in group_i.Geodesics {
30
31          // set source and target nodes
32          source = g.first_vertex;
33          target = g.last_vertex;
34
35          // check to see if the current flow between
36          // (source,target) has not been calculated
37          if CachedCF[(source,target)] is NULL {
38
39            // Find the voltages for the circuit having source and target
40            // nodes the first and last vertices of the geodesic.
41            // This function solves the system of equations as described
42            // on Chapter 3.
43            voltages = FindVoltages(source,target);
44
45            // calculate the currents using the voltages ONLY for downhill
46            // current flows as defined in Chapter 3.
47            // Current is equal to I = ( V(u) - V (v) ) / R(u,v)
48
49            For each e(u,v) in this.adjency_list {
50              I[u,v] = ( voltages[u] - voltages[v] ) /  edge(u,v).weight;
51
52              // Add each current that goes out of u
53              // to the Iout figure
```

Figure 5.7 Current flow calculation algorithm

```
54              Iout[u] +=  I[u,v];
55
56            }
57
58            // calculate the pro-rated current through he geodesic (s,...,t)
59            Itemp = 1;
60            For each nodepair(u,v) in g {
61              Itemp *= I[(u,v)]/Iout[u];
62            }
63
64            // store current flow value between source and target
65            CachedCF[(source,target)] = Itemp;
66          } // end-if
67
68          // add the CF to the result groupN (shortest or longest)
69          GroupNcurrents[x,group_i].add(CachedCF[(source,target)]);
70
71        } // end For each g
72
73        // Now that all geodesics currents in the ith-groupN
74        // have been calculated create an average in case there is
75        // more than one.
76        if (GroupNcurrents[x,group_i].size() > 1 ) {
77          GroupNcurrents[x,group_i] /= GroupNcurrents[x,group_i].size();
78        }
79
80      } // end For each group_i
81
82    } // end For x (shortest or longest)
83
84    return GroupNcurrents;
85  }
```

Figure 5.7 (continued)

The GetCurrentFlows() function is in charge of generating the current flow vector based on the current graph. In line 16, the shortest and longest geodesics of the graphs are found using Dijkstra’s algorithm. Each geodesic belongs to a particular Group-N. For example, if a graph has 2 vertices, *u* and *v*, of the same degree, there could possibly be more than one path between those 2 vertices. The GetGeodesics() function in line 16 will find all the shortest paths between *u* and *v* and it then will keep one shortest path with the minimum path length (shortest geodesic), and it will keep the shortest path with the maximum path length (longest geodesic). This process will be applied to all Group-N pairs. In the event more than 1 pair of the same degree (same Group-N) exists, then an average for all the shortest geodesics and an average for all the longest geodesics will be calculated, Lines 76-78. Lines 26-85 depict the process to go through all Group-Ns, shortest and longest, and all geodesics for the current graph and the calculation of the current flow for each of the Group-N groups. In lines 32-33 the source and target nodes are selected. These are the start and ending nodes of each geodesic. Since a geodesic could be both the shortest and longest geodesic at the same time, a cache vector is implemented to avoid calculating the same current flow for the same node-pair. Line 37 verifies whether the current flow for a given pair has been already calculated. In line 43, the function FindVoltages(source,target) calculates all the voltages for each node in the graph. This function solves the system of equations using a LU-decomposition after the initial voltage values for source, target, and ground nodes have been specified. In order to prevent trying to calculate voltages for non-connected vertices (in the case of disjoint graphs), BFS (breath-first-search) is used to determine if a vertex is connected through any path to the source node of the circuit.

Once all voltages have been calculated, we can proceed to calculate the current value for each edge. Lines 50-56 calculate the current using Ohm's law. The value for Iout(u), this is the total current that exits from node u is calculated in line 54. Once all currents have been calculated we can find the pro-rated current along the geodesic; this is done in lines 59-62. In line 69, the GroupNcurrents variable is modified to add the pro-rated current amount calculated in lines 59-62; each current flow belongs to a particular Group-N and shortest or longest set. In lines 76-78, once all current flows have been calculated, an average of the current flows for each Group-N shortest, and Group-N longest set is calculated if there is more than 1 current flow per set. The output of the function is the current flows for all Group-N sets of the current graph.

5.2.2 CF-vectors' computational complexity

In order to analyze the computational complexity of the CF-vectors' algorithm we will assume that the graph dataset file contains only one graph. The label count and resistance increment sections, lines 28-51, are dominated by the sorting of the labels by frequency. Line 28, the label count is done in *O*(V), while the resistance increment calculated in lines 33-35 is done in *O*(E). The sorting of the labels based on their frequency is done in *O*(V log(V)). The modification of the resistance increment values is done in *O*(V). We can say that the section prior to the calculation of the current flows is done in *O*((2V + E) + V log(V)).

The current flows calculation is much more computationally expensive compared to the prior section. Starting with the discovery of the geodesics and Group-Ns in line 16 of the GetCurrentFlows() function. As described on the previous section, in order to find all geodesics, Group-Ns must be identified first. The process of identifying Group-Ns requires evaluating all possible paths between node pairs of the same degree. For example, for a graph with 6 nodes of degree 3 the number of paths that can be form with between two nodes of degree 3, one as the source and the one as the target, is:

$$\binom{n}{2} = \frac{n!}{2!\times(n-2)!} = \frac{n\times(n-1)}{2} \tag{22}$$

$$\binom{6}{2} = \frac{6!}{2!\times(6-2)!} = \frac{6\times(6-1)}{2} = 15 \tag{23}$$

The number of paths to be evaluated for each node degree is on the order of $n^2/2$ where *n* is the number of nodes of a particular degree. The worst case scenario given a particular graph structure is for a fully connected graph where a path exists between every single pair of nodes. In this case, the number of paths to be evaluated is on the order of $V^2/2$. For each of the Group-N node-pairs both shortest and longest geodesics must be found. This process is being done using Dijkstra's algorithm which can be done in *O*((E+V) log(V)) using a priority queue [13]. The whole process of finding all geodesics takes approximately $O\left((V^2 \times (E+V)\log(V))/2\right)$.

Once all geodesics have been found, the process of calculating voltages and currents is on the order of $O(V^3)$ due to the LU- decomposition to find the voltages. For sparse graphs, the voltage calculation can be improved to *O*(E) operations per iterations, and the number of iterations depends on the gap between both the largest and second largest eigenvalues [7]. The section that calculates the current flows can be said to be on the order of $O\left((V^2 \times (E+V)\log(V))/2\right)$ and the whole CF-vectors algorithm is on the order of $O((2V+E)+V\log(V)+((V^2 \times (E+V)\log(V))/2)+V^3)$. Since we assumed that the graph dataset will have only 1 graph, the total computational complexity of processing a full graph dataset will increase proportional to the number of graphs in the dataset. We can observe that the computational complexity of the algorithm is relatively high, but we must keep in mind that this step must be done only once for each graph. Once a graph has been transformed from its graph representation to a vectorial representation, the current flow vector that represents the graph will never change and it can be used in any future comparison of the graph against another current flow vector representing another graph.

It is worth mentioning that the size of the graphs in the NCI-HIV dataset is relatively small, with the largest graph having only 214 nodes.

5.3 CF-compare's implementation

CF-compare was implemented on ANSI C++. CF-compare provides several options to compare two .cfv files. The two datasets to be compared are called: query dataset, and base dataset. The query dataset is usually a smaller dataset that we want to compare against our base dataset. Since the number of results that can be obtained from comparing the query dataset to the base dataset is equal to the number of graphs in the query dataset multiplied by the number of graphs in the base dataset, CF-compare provides the ability to limit the number of results to avoid generating huge output files. These options are -*n* and -*t*. Option -*n* allows the user to define the top N results to be generated. Option -*t* allows the user to define a value from 0 to 1, this value represents a threshold for the similarity measure, meaning that only graphs where the similarity measure is equal or greater than the provided threshold value would be returned. Results are stored in a text file that shows the name of the graph being compared, followed by the graphs that met the criteria provided by the user (either top N, above or equal to threshold, or all base graphs) in descending order based on the similarity value (closest matches are listed first), this helps to identify the closest matches in a more efficient manner. In case the results are needed for classification purposes, CF-compare can provide counts based on the class labels in the base dataset. Option -*c* allows the user to request class counts to be included. Class counts will be generated in a separate file from the results, showing the total number of graphs from each class that met the criteria provided by the user.

```
$> cf-compare --help
Usage: cf-compare [options] query_set_file [base_set_file]
If base set is not provided, it compares the query set to itself.
options:
-t (0..1): Match value threshold
-n (1..n): Top n best matches
-c output class counts
```

Figure 5.8 CF-compare's command line

5.3.1 CF-compare's algorithm

```
CF-compare(QueryFile,BaseFile,topN,threshold,produceClassCounts) {

array QueryDataset;
array BaseDataset;
array ClassCounts;
array topN_matches;
int TotalDegrees;
double g1_shortest,g1_longest, g2_shortest, g2_longest;
double mv,S,L;
file ResultsFile, ClassCountsFile;

// Load files
For each graph in BaseFile {
  BaseDataset.add(graph);
}

if QueryFile == BaseFile {
  QueryDataset = BaseDataset;
}
else {
  For each graph in QueryFile {
    QueryDataset.add(graph);
  }
}

For each q in QueryDataset {
  For each b in BaseDataset {

    // Select the max degree between the two graphs
    TotalDegrees = max( q.CurrentFlows.size(), b.CurrentFlows.size());

    // calculate the differences between the each current flow of the
    // same degree
    For d=0 to TotalDegrees {

      g1_shortest = q.CurrentFlows[d].shortest();
      g1_longest = q.CurrentFlows[d].longest();
      g2_shortest = b.CurrentFlows[d].shortest();
      g2_longest = b.CurrentFlows[d].longest();

    }

    if g1_shortest == g2_shortest {
      // No difference
      S += 0.0;
    }
```

Figure 5.9 CF-compare's algorithm

```
47        else {
48          // Partial Difference
49             S +=  abs(g1_shortest-g2_shortest) / (g1_shortest+g2_shortest);
50        }
51
52        if (g1_longest == g2_longest){
53          // No difference
54          L += 0.0;
55        }
56        else {
57          // Partial Difference
58          L +=  abs(g1_longest-g2_longest) / (g1_longest+g2_longest);
59        }
60
61        // TotalDegrees * 2 is to account for d shortest
62        // and d longest geodesics
63        mv = 1.0 - ( (S + L) / (TotalDegrees * 2.0) );
64
65        // If the match value is greater or equal than
66        // the threshold set by the user then store the value
67        // for pair (q,b)
68        if (mv >= threshold)  {
69          topN_matches.add((q,b),mv);
70        }
71
72      } // end For each b
73
74
75      topN_matches.ReverseSort();
76
77      // Output first top N matches;
78      For i=0 to topN {
79        ResultsFile.write(topN_matches[i]);
80
81        // Output class counts if requested by user
82        if ProduceClassCounts = True {
83          // count per each class how many graphs in the top N
84          ClassCounts[q,topN_matches[i].class]++;
85        }
86
87      } // end For i to topN
88
89      if ProduceClassCounts = True {
90        ClassCountsFile.write(ClassCounts[q]);
91      }
92
93    } // end For each q
94
95    return;
96  }
```

Figure 5.9 (continued)

CF-compare first loads the query and base files into datasets, lines 12-24. If the query set and the base set are the same, the load will only take place one time as both datasets will be the same. Once both datasets are loaded, each pair of graph from the query set and the base set are compared to one another, lines 26-93. First, the TotalDegrees variable is calculated, this is equivalent to equation (21) in Section 4.5. For each set of shortest and longest current flow values for each node degree the difference between graph q from the query dataset and graph b from the base dataset is calculated, lines 34-59. Please note that in the event that one of the graphs, either q or b, does not have a particular current flow value for a determined node degree, a value of 0 will be assigned. With all the current flow values for a particular node degree, that is, current flow values for the shortest and longest geodesics between vertices of the selected node degree, the match value is then calculated as per equation (20), line 63. In lines 68-70, the match value is compared to the threshold set by the user. If the match value is above or equal to the threshold then the match is stored. In line 75, all matches that were above or equal to the threshold are ordered from bigger to smaller. Please remember that the closer the value is to 1 the closer both graph's current flow vectors are similar to each other. In Lines 78-85 we write the output of only the first top N matches based on their match values. If the user requested class counts to be created, a file containing the count of how many graphs of a particular class were in the top N matches for each graph. Fig. 5.10 shows a sample result output file.

```
9168,CA 58368:CI:0.99474 50851:CI:0.52679 50848:CA:0.51256
50848,CA 50851:CI:0.95929 64052:CI:0.93493 9168:CA:0.51256
50851,CI 50848:CA:0.95929 64052:CI:0.90196 9168:CA:0.52679
58368,CI 9168:CA:0.99474 50851:CI:0.52153 50848:CA:0.50731
64052,CI 50848:CA:0.93493 50851:CI:0.90196 58368:CI:0.49733
```

Figure 5.10 CF-compare results output file

This file compared five graphs to each other storing only the top 3 matches. Both query and base files were the same. Since both, query and base dataset, are the same for this experiment, each graph will be compared to all other graphs in the dataset excluding itself. As we can observe, the file first shows the name of the graph followed by the class of the graph (if available). For example, graph 9168 belongs to the CA class. The first match for graph 9168 is graph 58368, the class of the matched graph is also shown, followed by the match value. In this case graph 58368 belongs to class CI and the match value between graph 9168 and graph 58368 was 0.99474. Fig. 5.11 shows graph 9168 and its three closest matches. As we can observe, the closest match, graph 58368, is similar to 9168, while other matches are clearly different.

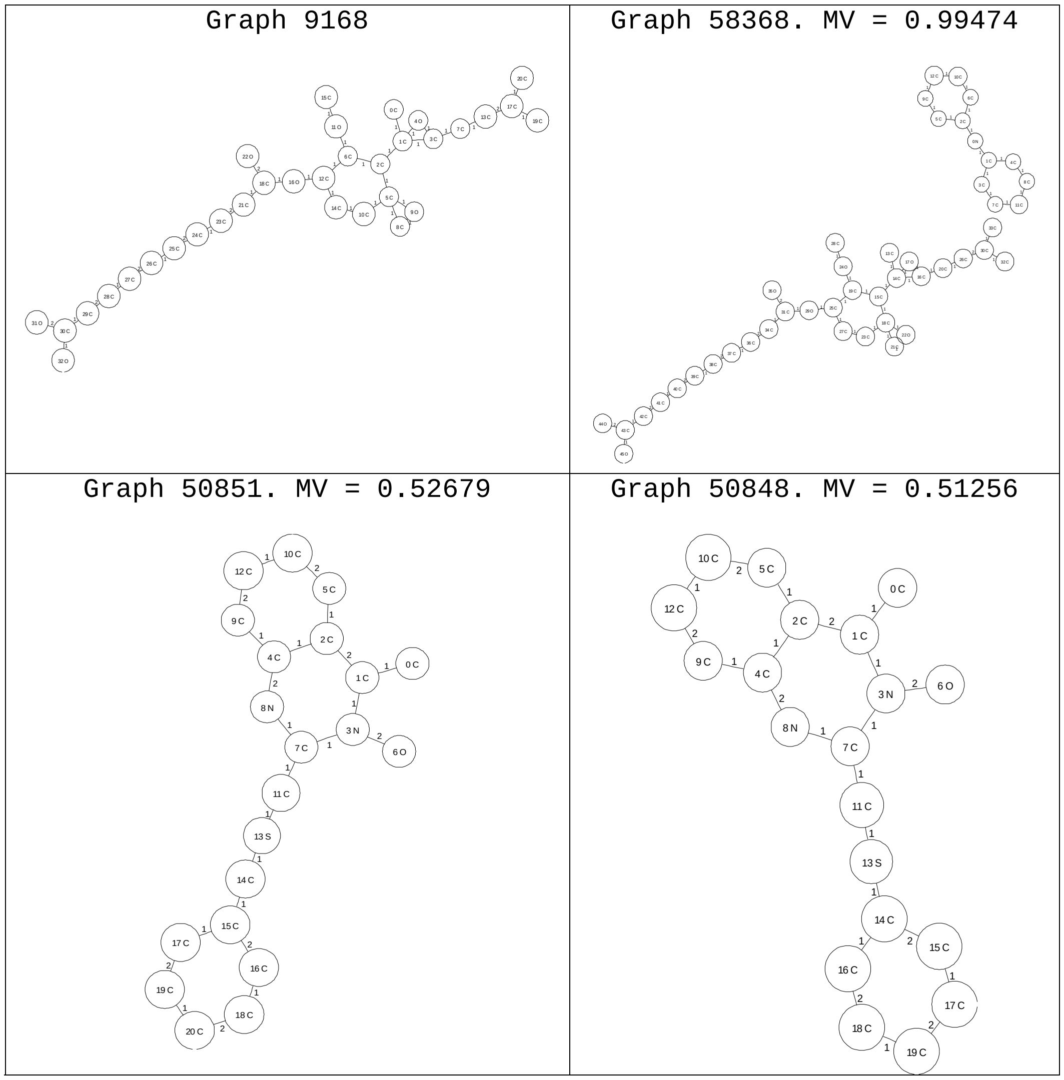

Graph 9168
Graph 58368. MV = 0.99474
Graph 50851. MV = 0.52679
Graph 50848. MV = 0.51256

Figure 5.11 Graphs 9168, 58368, 50851, and 50848

```
graph,  class,  top,  CA,  MaxMV_CA,  CI,  MaxMV_CI
9168,   CA,     3,    1,   0.51256,   2,   0.99474
50848,  CA,     3     1,   0.51256,   2,   0.95929
50851,  CI,     3,    2,   0.95929,   1,   0.90196
58368,  CI,     3,    2,   0.99474,   1,   0.52153
64052,  CI,     3,    1,   0.93493,   2,   0.90196
```

Figure 5.12 Class counts file

Fig. 5.12 shows the class counts file the graphs in Fig. 5.11. Class counts files also store the maximum match value for each class. For example for graph 50848, in the top 3, it has 1 graph from class CA with a value of 0.5126 and 2 from class CI from which the maximum match value was 0.95929. In this particular example class CM is not represented as the dataset file only contained two classes, CA and CI.

5.3.2 CF-compare's computational complexity

CF-compare's computational complexity is linear in time to the dimension of the current flow vector representing the graph, $O(\mathrm{D})$. The process of comparing one graph to another boils down to solving equation (20); when comparing one graph to a base dataset the complexity increases to $O(\mathrm{D} \log \mathrm{D})$, this is caused by the ordering of the top N results. CF-compare's computational complexity highlights the benefits of the proposed approach, the process of converting the graph to a vectorial representation, albeit costly in time, is only needed one time per graph; any future comparison of such graph to a database of current flows representing graphs will be almost linear in time.

5.4 Additional tools

Other tools were developed as part of this thesis in order to facilitate the analysis and visualization of the results. These tools are: SDF2GDS and GDS2DOT. SDF2GDS converts an .sdf file also known as Structures Data File which is a common file format developed by Molecular Design Limited to handle a list of molecular structures with associated properties [15] into a .gds file, which is the format expected by CF-vectors. GDS2DOT exports each graph in the graph dataset file to separate .dot files for each graph. .dot files as defined by [16] are used by Graphviz as its input format. Graphviz is a popular open source suite of tools developed by AT&T research labs for graph visualization.

GDS2DOT performs a special ordering of the vertices in order to prepare the graph for a better rendering using Graphviz’s neato layout engine. Vertices with a larger number of edges are defined first in the .dot file; this will tell Graphviz to position those vertices first, producing a better graphical representation of the graph.

```
graph CA50848{
  node[shape="circle"]

  "v14" [label ="14 C"]
  "v7" [label ="7 C"]
  "v4" [label ="4 C"]
  "v3" [label ="3 N"]
  "v2" [label ="2 C"]
  "v1" [label ="1 C"]
  "v19" [label ="19 C"]
  "v18" [label ="18 C"]
  "v17" [label ="17 C"]
  "v16" [label ="16 C"]
  "v15" [label ="15 C"]
  "v13" [label ="13 S"]
  "v12" [label ="12 C"]
  "v11" [label ="11 C"]
  "v10" [label ="10 C"]
  "v9" [label ="9 C"]
  "v8" [label ="8 N"]
  "v5" [label ="5 C"]
  "v6" [label ="6 O"]
  "v0" [label ="0 C"]
  "v0"--"v1" [label ="1"]
  "v1"--"v2" [label ="2"]
  "v1"--"v3" [label ="1"]
  "v2"--"v4" [label ="1"]
```
Figure 5.13 Graphviz’s dot file example

```
   "v2"--"v5" [label ="1"]
   "v3"--"v6" [label ="2"]
   "v3"--"v7" [label ="1"]
   "v4"--"v8" [label ="2"]
   "v4"--"v9" [label ="1"]
   "v5"--"v10" [label ="2"]
   "v7"--"v11" [label ="1"]
   "v7"--"v8" [label ="1"]
   "v9"--"v12" [label ="2"]
   "v10"--"v12" [label ="1"]
   "v11"--"v13" [label ="1"]
   "v13"--"v14" [label ="1"]
   "v14"--"v15" [label ="2"]
   "v14"--"v16" [label ="1"]
   "v15"--"v17" [label ="1"]
   "v16"--"v18" [label ="2"]
   "v17"--"v19" [label ="2"]
   "v18"--"v19" [label ="1"]
}
```
Figure 5.13 (continued)

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Experimental setup

During the testing of our prototype we applied our algorithms to the NCI-HIV dataset of chemical compounds [10]. This dataset contains 42,689 chemical compounds, 423 of which are active (CA), 1081 are moderately active (CM), and 41,185 are inactive (CI). The NCI-HIV dataset has been used in the empirical evaluation of several graph mining techniques [17] [18] [19] [20]. The first step of our experiments was to convert the NCI-HIV dataset from .sdf to .gds. We used SDF2GDS for this purpose. Once we had our graph dataset file, the next step was creating the current flow vectors file. Using CF-vectors on the NCI-HIV.gds file took a little over 10 minutes on a 2.4 GHz Pentium 4 with 512Mb of RAM. With the NCI-HIV.cfv file at hand we were ready to test the graph matching potential of our algorithm.

The first set of experiments as described in Section 6.1.1 will show the results of several comparisons of multiple graphs against the whole NCI-HIV dataset. The second set of experiments as described in Section 6.1.2 will show the results on the graph classification problem for the NCI-HIV dataset by using the class counts obtained from the CF-compare algorithm as the input to several classification models like neural networks, k-nearest neighbors, and rule based systems. The results are compared to those reported in [17].

6.1.1 Graph visual comparison experiments

After converting the NCI-HIV dataset from an .sdf file to a .gds file and generating the current flow vectors file (NCI-HIV.cfv) the next step in our research was to find out how the similarity measure works for finding isomorphisms. We compared the NCI-HIV dataset against itself with a threshold of 1.0. As described before, CF-compare will compare each graph in the query dataset against all other graphs in the base dataset. Since both, query and base dataset, are the same for this experiment, each graph will be compared to all other graphs in the dataset excluding itself. For this experiment, we used two current flow dataset files (.cfv) extracted out of the same NCI-HIV dataset. Each .cfv file was produced using a slightly different version of CF-vectors for each file. The versions vary from each other on the percentage used during the nodal information integration step described in Section 4.4. Our default implementation, as described in the pseudocode in Fig. 5.7, used only 10% of the lowest resistance value in the dataset. The NCI-HIV dataset contains only three possible values for the edge weights, or bonds, between its nodes, or atoms, single, double, and triple bond, represented with weights of 1, 2, and 3 respectively. The other implementation of the CF-vectors uses a different percentage of the lowest edge weight value; in this case the percentage is 0% - *this is equivalent to excluding nodal information when calculating the current flow*.

The first current flow vectors dataset file to be evaluated was produced using the 0% percentage; we will refer to this current flow vectors dataset as HIV00.cfv. The second file produced was using the 10% percentage; we will refer to this dataset as HIV10.cfv.

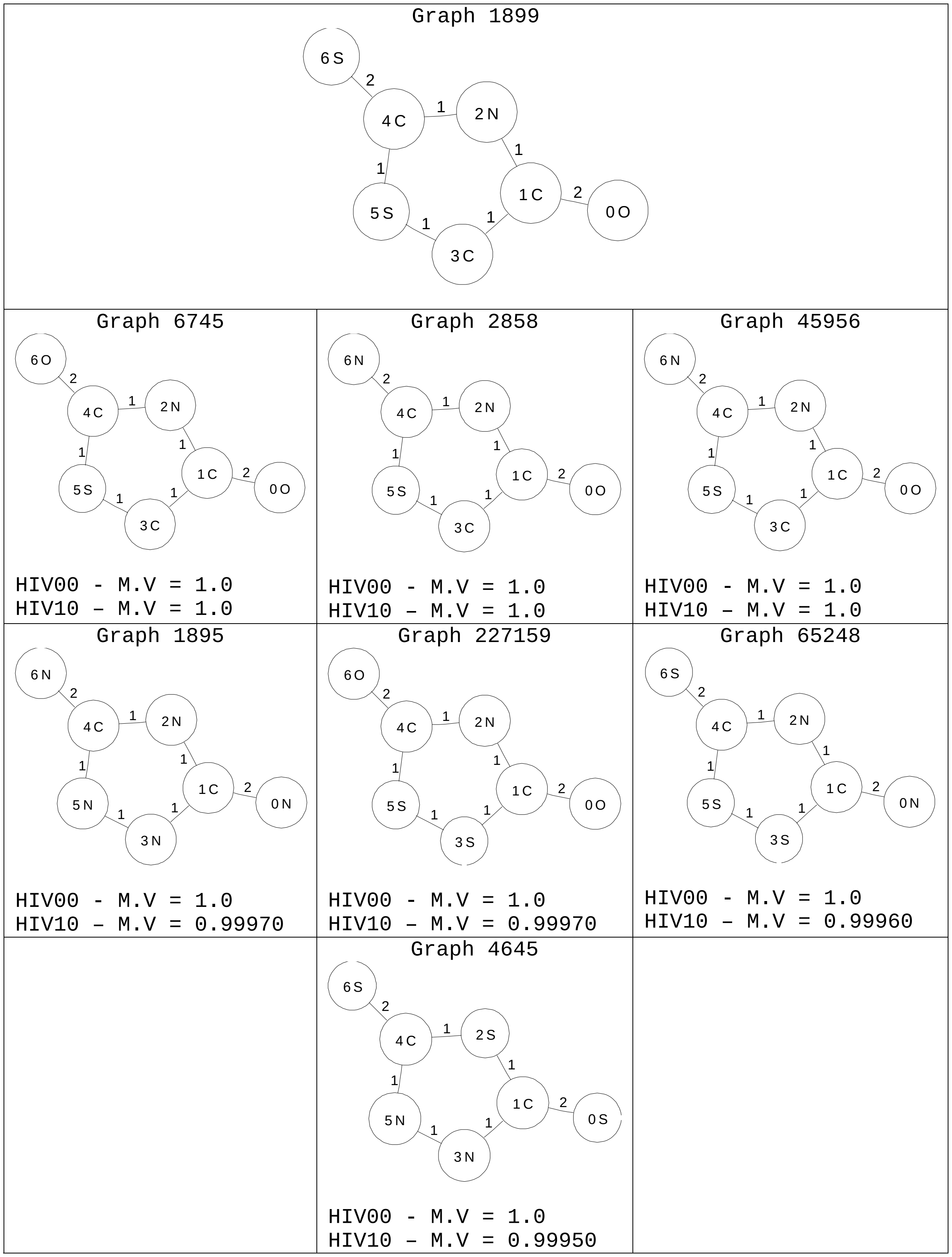


Figure 6.1 Graph 1899 matches at 0% and 10%

For the HIV00.cfv dataset we found 4,371 compounds with matches having a 1.0 matching value. For the HIV10.cfv dataset we found 2,216 compounds with matches having a 1.0 matching value, indicating the match criteria was more difficult. Fig. 6.1 shows the matches for graph 1899 and their corresponding match values at both 0% and 10%. As we can observe from the figure, all graphs are nearly identical. The only difference between graphs 1899, 2858, 6745, 45956 is at vertex 6, where for graph 1899 it is a sulfur atom, S; for graphs 2858 and 45956 it is a nitrogen atom, N; for graph 6745 it is an oxygen atom, O. Current flow analysis allows for an error-tolerant graph matching where the matches will not always be perfect, allowing for a graph to be matched to closely related graphs.

From the two datasets, HIV00 and HIV10, we can observe the impact that excluding the nodal information during the current flow calculation process has, in particular for the HIV00, where no nodal information is included, more matches were obtained compared to the number of matches for the HIV10. The number of compounds with matches at a 1.0 threshold for the HIV00 dataset is double the number of matches for the HIV10 dataset. This highlights the benefits of including nodal information in the current flow analysis, as not only the structure of the graph, but also the label information could be important during the graph matching process.

Due to the sheer size of the dataset we cannot visually verify each of the results, but after a random verification of several compounds, each and every one of those that were visually verified were perfect isomorphisms (excluding the same graph compared to itself). This, of course, by no means allows us to present a sound statement about the efficiency of our prototype, but it is encouraging to see the excellent results achieved when finding isomorphisms.

The next experiment in our research was to perform a graph matching with a lower matching value. Instead of applying a threshold of 1.0, we tested with different thresholds. We compared the HIV10.cfv dataset to itself with a threshold: 0.97. Applying the lower threshold we obtained many more compounds with matches; in this case 27,684. Fig. 6.1 shows the match values for the HIV10.cfv dataset that were above the 0.97 threshold for compound 1899. As we can observe, when excluding the vertex label information from the current flow analysis in the HIV00.cfv dataset, the algorithm selected all graphs with identical structure, regardless of the vertex labels. When using the HIV10.cfv, which incorporates the vertex label information into the current flow analysis, only those with labels with very similar values where selected.

We can observe that both 2858 and 45956 are identical on structure and vertex labels. Graph 6745 only differs in one vertex and given the fact that both oxygen (O) and nitrogen (N) are very common labels in the dataset, both should have very similar values. Other graphs that after comparison returned a lower match value differed in more than one vertex label.

Fig. 6.2 shows results for compound 3417 at the 0.97 threshold. Only three matches were found for this graph with a match value higher than 0.97. Two of the matches, graphs 629861 and 629864 display a very similar structure, but as we can see from their match values, their current flows are different when compared to graph 3417.

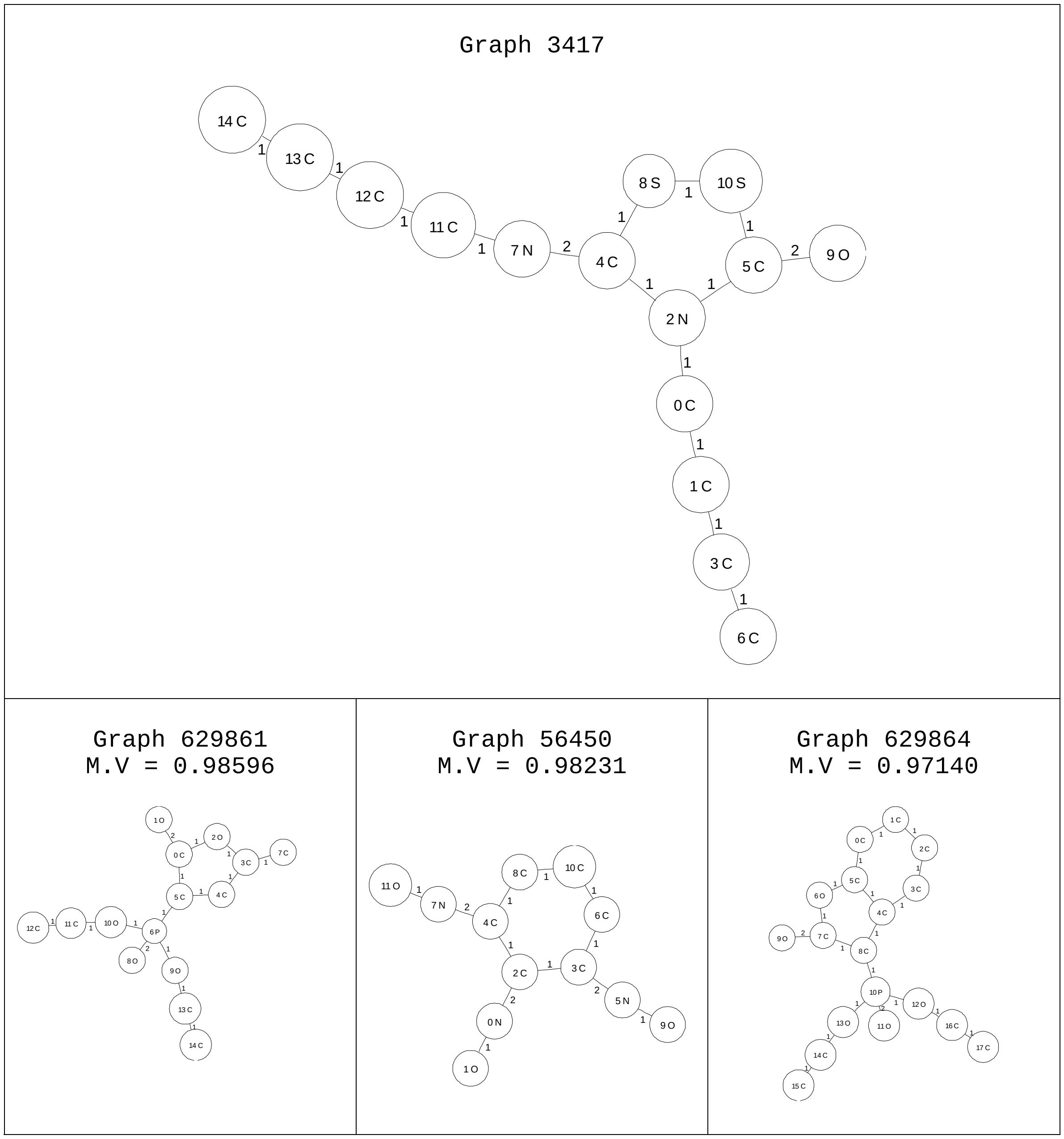


Figure 6.2 Graph 3417 matches above 0.97

From Fig. 6.2 we can observe that all the graphs share a structure with a main body and two long appendixes. The main idea behind current flow analysis for error-tolerant graph matching is that by calculating the current flows along specific paths of the graph, the algorithm will capture information not only about the vertices along the path, but also about vertices along side the path as electric current flows through them. The characteristics of each graph’s structure and vertex information will provide a very singular footprint that will allow matching similar graphs as the current flows values should be similar for similar structures.

Our next figure shows one graph that only returned two matches within the 0.97 threshold. One match value is really high and the other one is much closer to the threshold.

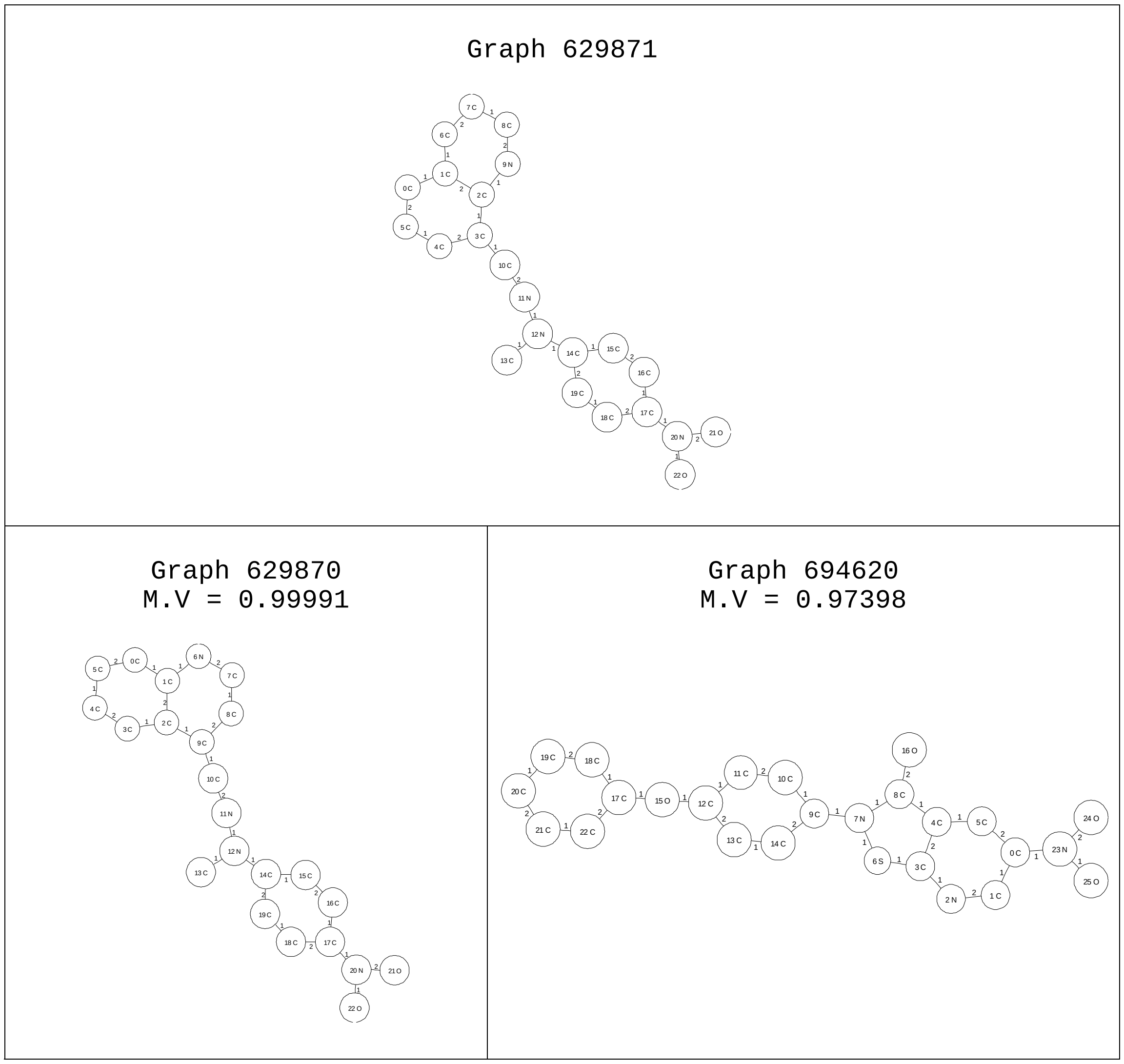


Figure 6.3 Graph 629871 matches above 0.97

Graphs 629871 and 629870 are nearly identical as illustrated by their match value. On the other hand, compound 694620 shares certain characteristics with compound 629871, especially at the end of the graph which is made up of two oxygen and one nitrogen atom. We expect that during the current flow analysis these characteristics are the ones providing a particular current flow for specific geodesics that when compared to those of another graph will provide the similarities needed to obtain a high match value.

Figure 6.4 shows compound 633892, which only had one match within the 0.97 threshold. As we can observe, it is hard to discern particular characteristics between these two graphs, other than the current flow vectors are similar.

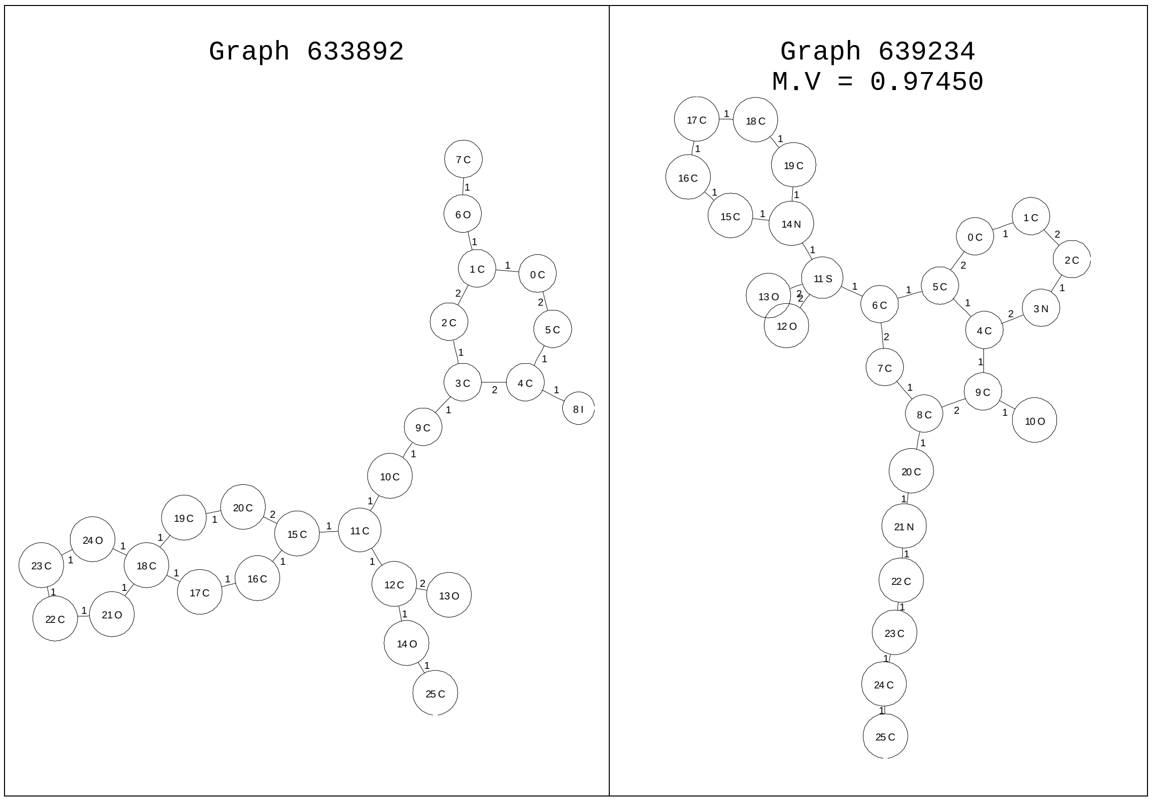


Figure 6.4 Graph 633892 matches above 0.97

Figures 6.5-6.8 show different graphs and their closest two matches with their correspondent match values for the HIV10.cfv dataset. The graphs displayed here were chosen at random from the 42,689 compounds in the NCI-HIV dataset.

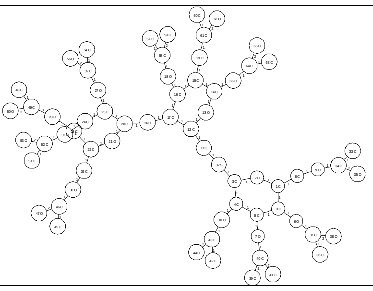
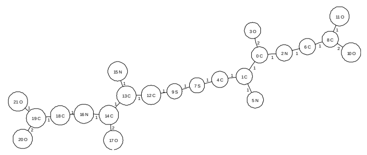
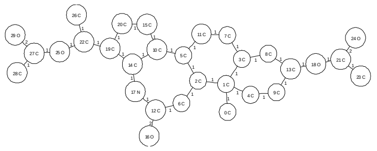
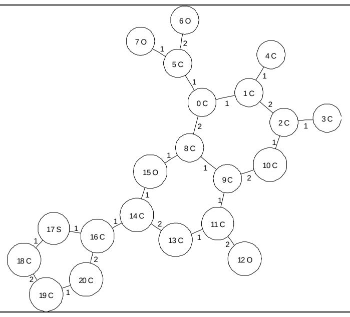
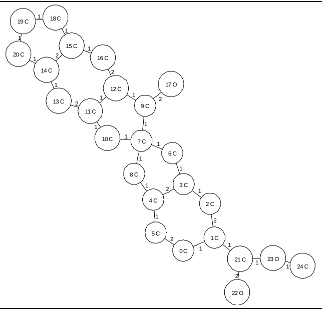
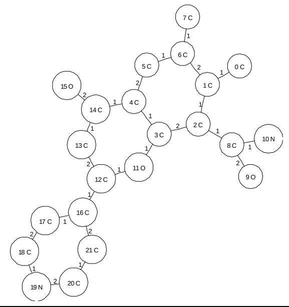
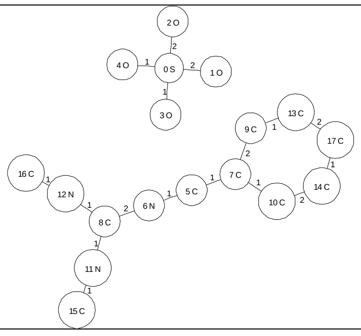
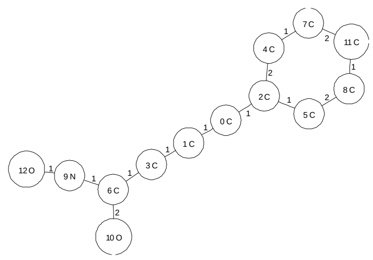
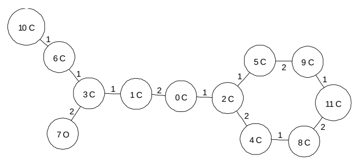
GRAPH	MATCH 1	MATCH 2
Graph 642970 Class CI	Graph 333711 Class CI M.V = 0.99296	Graph 119076 Class CI M.V = 0.98010
		
Graph 629789 Class CI	Graph 670310 Class CI M.V = 0.97940	Graph 637419 Class CI M.V = 0.97739
		
Graph 106563 Class CI	Graph 131300 Class CI M.V = 0.97703	Graph 148201 Class CI M.V = 0.97109
		

Figure 6.5 Graphs 642970, 629789, 106563 matches

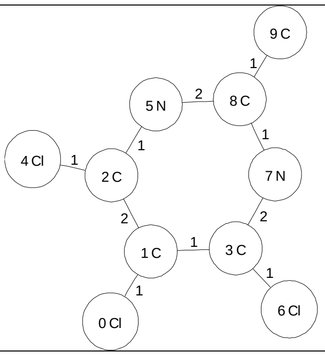
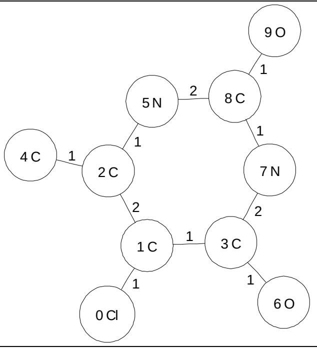
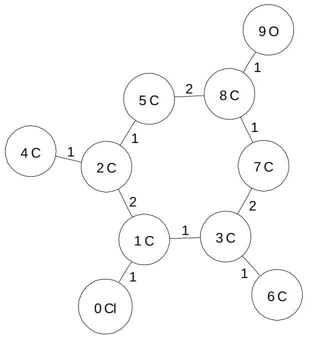
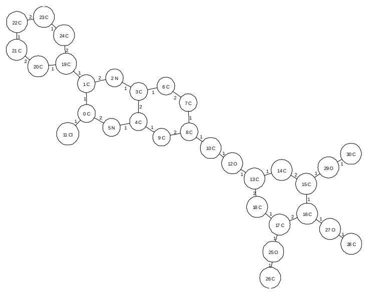
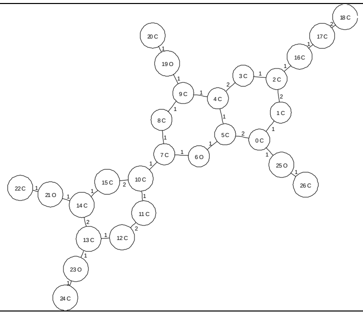
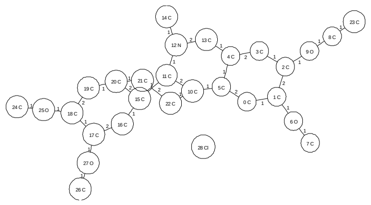
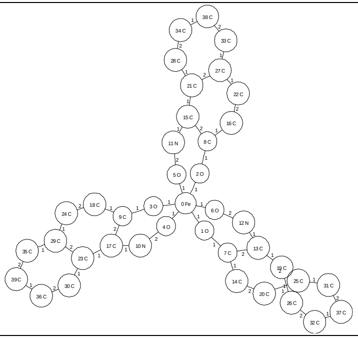
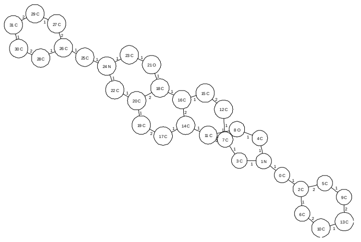
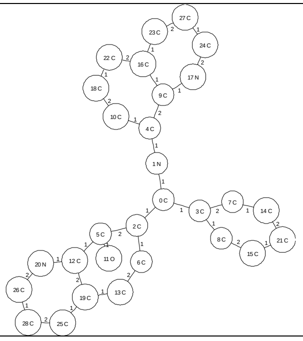
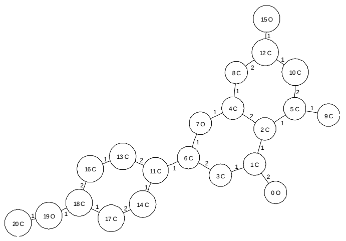
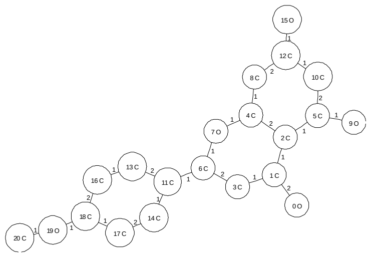
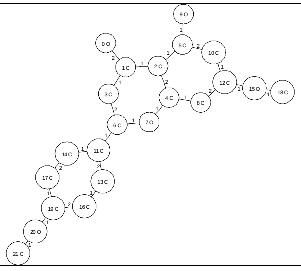
GRAPH	MATCH 1	MATCH 2
Graph 26540 Class CI	Graph 26542 Class CI M.V = 1.00000	Graph 4971 Class CI M.V = 0.99973
		
Graph 693764 Class CI	Graph 641523 Class CI M.V = 0.98283	Graph 645311 Class CI M.V = 0.97586
		
Graph 16086 Class CM	Graph 79050 Class CI M.V = 0.97385	Graph 84096 Class CI M.V = 0.97365
		
Graph 121858 Class CM	Graph 76061 Class CI M.V = 0.99992	Graph 94547 Class CI M.V = 0.98027
		

Figure 6.6 Graphs 26540, 693764, 16086, 121858 matches

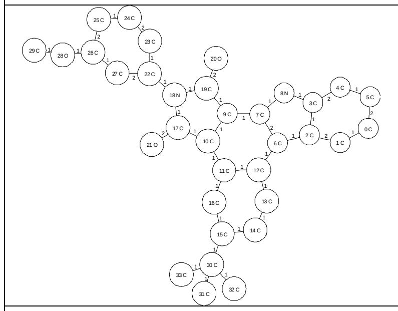
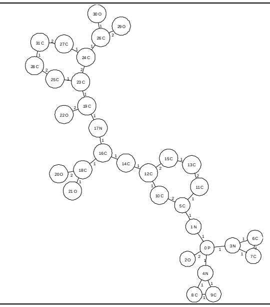
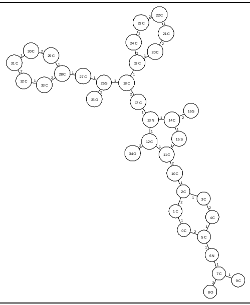
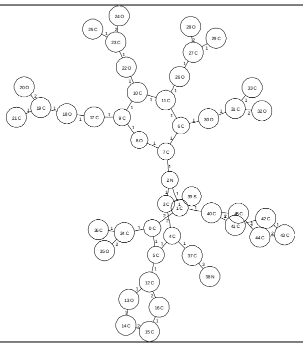
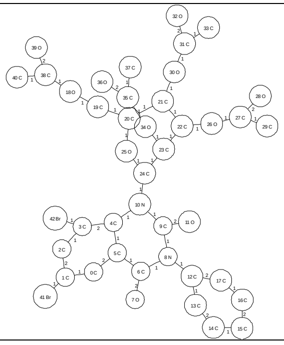
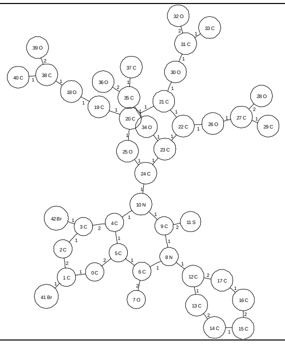
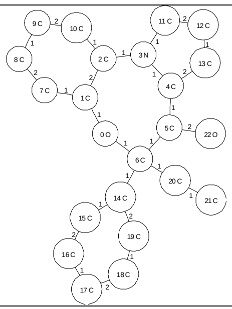
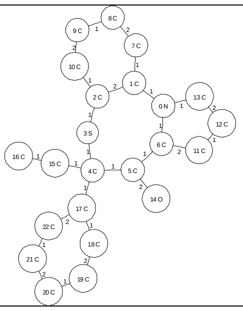
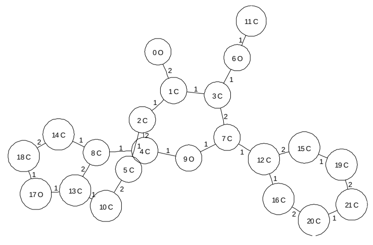
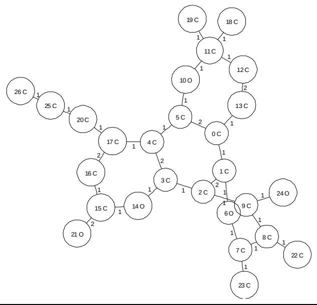
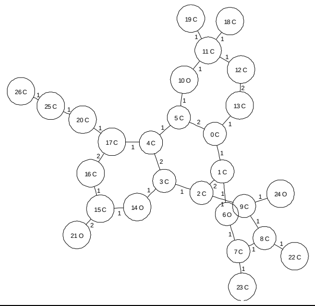
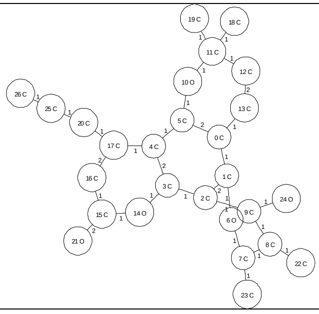
GRAPH	MATCH 1	MATCH 2
Graph 643418 Class CM	Graph 70804 Class CA M.V = 0.97309	Graph 659624 Class CI M.V = 0.97285
		
Graph 676606 Class CM	Graph 639749 Class CI M.V = 0.97875	Graph 639734 Class CI M.V = 0.97875
		
Graph 676419 Class CA	Graph 661186 Class CA M.V = 0.99984	Graph 335755 Class CI M.V = 0.98643
		
Graph 675451 Class CA	Graph 675450 Class CM M.V = 1.00000	Graph 675449 Class CM M.V = 1.00000
		

Figure 6.7 Graphs 643418, 676606, 676419, 675451 matches

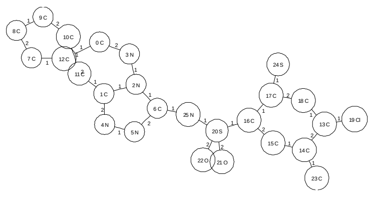
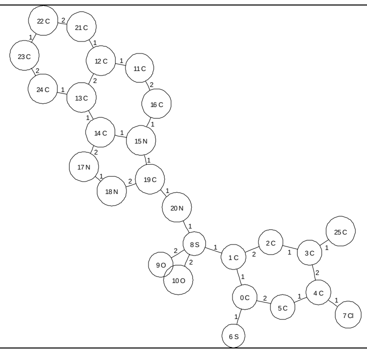
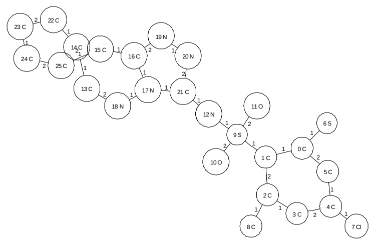
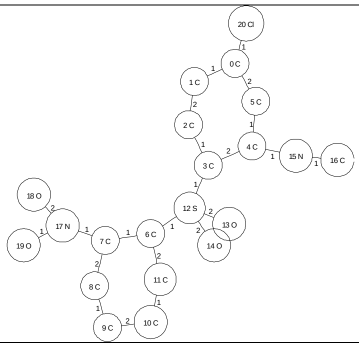
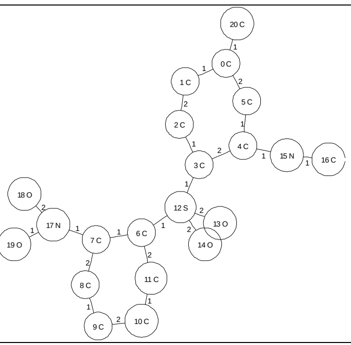
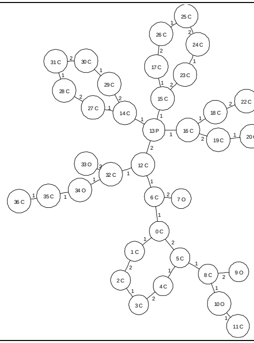
GRAPH	MATCH 1	MATCH 2
Graph 673997 Class CA	Graph 686774 Class CM M.V = 0.97777	Graph 696894 Class CM M.V = 0.97552
		
Graph 671292 Class CA	Graph 671291 Class CA M.V = 1.00000	Graph 662767 Class CM M.V = 0.97861
		

Figure 6.8 Graphs 673997, 671292 matches

As we can observe from Fig. 6.5-6.8, current flow analysis for error-tolerant graph matching allows for a fast comparison of a graph against a dataset of graphs of a considerable size. In some cases, the results are quite good, like for graphs 26540 and 106563; while on other cases, like graph 642970 it is hard to tell similarities between the selected graph and its matches. Overall, the selection of matches based on their current flow vectors should provide results where the structure of the graphs are very similar due to the fact that by calculating the current flow along shortest and longest geodesics of each Group-N pair of nodes should provide a very distinctive current flow vector for each, and a similar current flow vector for graphs with similar structure.

In the next section, we will explore the predictive power of current flow vectors on the NCI-HIV dataset. As mentioned before, the NCI-HIV dataset classifies its 42,689 compounds in active (CA), 423, moderately active (CM), 1081, and 41,185 are inactive (CI). By using the results of current flow analysis for graph classification we will show that for the NCI-HIV dataset the similarity of the compounds is an indicator of the class they belong to. We will also show that by using current flow analysis for graph comparison will allow us to determine which compounds are similar to each other, therefore when predicting the class for a compound we will rely on its closest matches as obtained from using current flow analysis.

6.1.2 Graph classification problem on the NCI-HIV dataset

In order to evaluate the predictive power of current flow vectors on the NCI-HIV dataset we compare the results against the frequent subgraph discovery algorithm (FSG). In [18] [19], M. Deshpande et al. investigated the predictive power of the FSG algorithm using support vector machines (SVM) as the classification technique. The use of SVM enabled them to associate a higher cost for the mis-classification of positive instances. Three different classification problems were defined in [18] and [19]:

1. CA vs. CM
2. CA+CM vs. CI
3. CA vs. CI

We compared the results of current flow vectors for each of these classification problems. The first step in our experiment was to generate the current flow vectors for all 42,689 compounds. We calculated the current flow vectors for both 0% and 10% of the lowest edge weight to incorporate nodal information into the current flow analysis. This step took approximately 10 minutes for each dataset, HIV00.cfv and HIV10.cfv. The next step was performed for each of the datasets.

The second step was to compare each of the compounds against all other compounds in the dataset using our similarity measure (excluding the compound compared to it). The comparison took a little over 6 hours for each dataset. The results of the second step were the similarity values (match values) for each of the compounds in the dataset compared to all the other compounds as well as the class counts files. Given the size of the dataset we only stored the 100 closest matches for each compound.

The following table shows the average number of compounds of a particular class within the top 30 closest matches for the HIV00.cfv dataset and for the HIV10.cfv dataset.

Table 6.1 Average number of compounds within top 30 matches

Class	Dataset	Average number of CA	Average number of CI	Average number of CM
CA	HIV00	4.012 (± 4.538)	23.281 (± 6.105)	2.707 (± 2.755)
	HIV10	4.019 (± 4.553)	23.300 (± 6.094)	2.681 (± 2.746)
CI	HIV00	0.241 (± 0.836)	29.034 (± 1.506)	0.725 (± 1.037)
	HIV10	0.242 (± 0.840)	29.033 (± 1.507)	0.725 (± 1.037)
CM	HIV00	0.901 (± 2.039)	27.441 (± 3.899)	1.658 (± 2.294)
	HIV10	0.899 (± 2.038)	27.439 (± 3.901)	1.661 (± 2.296)

From the results on Table 6.1 we can observe several facts. First, the number of average active compounds within the top 30 matches (for the HIV00 dataset) for a compound that is active is 4.0132 with and standard deviation of 4.538; this is higher compared to the average number of compounds that are inactive, 23.281 with a standard deviation of 6.105, and moderately active, 2.707 with a standard deviation of 2.755. This is a good indicator that the closest matches' class for a compound could help to determine its class. If we observe results for the inactive compounds, we notice that within the top 30 matches the average number of inactive compounds 29.034 with a standard deviation of 1.506. This indicates that an inactive compound should have within the first 30 matches at least 29 matches. If we compare this to the number of inactive compounds within the top 30 for the active compounds, 23.281 to 29.034, we can clearly see the difference. Another fact to notice from Table 6.1 is how close the results are for both HIV00.cfv and HIV10.cfv datasets. This could indicate several things; first, that the nodal information did not have enough influence on the current flow calculation; second, that the nodal information does not play a pivotal role in the determination of the class for the NCI-HIV dataset, or third, that even after including the nodal information based on the vertex labels (atoms) on the current flow calculation the matches returned were very similar due to the structure of the compounds. Based on this fact, we will show the results for the HIV00.cfv dataset when using the class counts obtained for the top 100 matches.

As mentioned before, class count files store the number of compounds and the maximum match value per each class within a particular top N. Therefore, after obtaining the top 100 matches for each graph in the dataset we proceeded to extract the feature vectors out of the class counts files for each top N/experiment combination.

The idea behind creating one training dataset out of each class count file is to evaluate different classification algorithms with different top Ns, it could be that the best classification is obtained by only looking at the first 30 matches, or it could be that when looking at the first 80 matches better results would be obtained.

Let us start with experiment 1, CA vs. CM. Each training dataset for this experiment will contain 1,504 feature vectors, one feature vector for each graph that belongs to classes CA or CM. We created one training dataset for each top N. Since each top N generates a particular class count file, we have 100 class count files from which we extracted 100 training datasets each with 1,504 feature vectors.

Each feature vector contains 6 attribute values and 1 class label. The attributes are:

1. Number of CA compounds within the top N matches
2. Number of CM compounds within the top N matches
3. Number of CI compounds within the top N matches
4. Maximum match value for a CA compound
5. Maximum match value for a CM compound
6. Maximum match value for a CI compound

The class label indicates the real class that the compound being compared belongs to. For attributes 4, 5, and 6 the maximum match value is determined by the highest ranking compound for a particular class.

In order to identify the different training datasets we used the following naming convention: Ei_topN, where *i* is the experiment number and N is the top N class count file used to extract the feature vectors. For example, one training dataset for experiment 1 will be the one extracted from the class count file produced out of the top 10 results. We will refer to this particular training dataset as E1_top10, where E1 represents that is for experiment 1 and top10 indicates that it was extracted from the class count file for the top 10 results. The reason for identifying the particular experiment in the name of the dataset is due to the fact that we are evaluating a 2-class classification problem. For experiment 2 when we combine CA+CM, any compound belonging to class CA or CM will be class 1 while CI would be class 2.

Similarly to the training dataset E1_top10, we will have 99 more training datasets for experiment 1, from E1_top1 to E1_top100. Analogously, we will have 100 training datasets for experiment 2, CA+CM vs. CI; each training dataset with 42,689 training instances (one for each graph in the NCI-HIV dataset). For experiment 3, CA vs. CI, we have another 100 training datasets, each with 41,608 training instances (one for any graph belonging to classes CA or CI).

Using the 300 training datasets, we used a variety of classification algorithms including naïve Bayes, back-propagation neural networks, and support vector machines in order to attempt to classify the compounds based on the six attributes defined. Each of the algorithms was tested with different parameters in order to find the best set of settings for each algorithm. The idea behind testing several classification algorithms was to find the best classifier that will capture the underlying patterns stored in the feature vectors extracted out of the class count files. This was done using Weka 3.5.6 [21]. In order to determine the best combination of dataset/algorithm/settings we performed a 5x2 cross-validation with an F-test. The most accurate on all three classification problems was a back-propagation neural network.

6.2 Result evaluation and comparison

After determining the best training dataset/classifier combination using a 5x2 cross-validation with an F-test, we performed a five-fold cross-validation on the best datasets. The reason to perform a five-fold cross-validation is to be able to compare the results against those presented in [18] and [19], where a five-fold cross-validation was also performed. The results for dataset HIV00.cfv are shown in Table 6.2.

Table 6.2 Current flow vectors results on HIV00.cfv dataset

Dataset	Classifier	Weka Options L: learning curve M: momentum N: epochs H: neurons on hidden layer K: kernel type (2 – RBF) C: cost parameter C for C-SVC G: gamma value for RBF kernel	Area under a receiver operating characteristic curve (ROC-AUC)
E1_Top36	Neural Network	-L 0.3 -M 0.2 -N 150 -H 2	0.781 (±0.022)
E2_Top57	Neural Network	-L 0.3 -M 0.2 -N 500 -H 4	0.715 (±0.013)
E3_Top10	Neural Network	-L 0.3 -M 0.2 -N 150 -H 4	0.865 (±0.022)
E1_Top20	Naïve Bayes		0.754 (±0.012)
E2_Top38	Naïve Bayes		0.711 (±0.020)
E3_Top53	Naïve Bayes		0.859 (±0.015)
E1_Top22	SVM	-K 2 –C 1.0 –G 0.125	0.764 (±0.019)
E2_Top57	SVM	-K 2 –C 1.1 –G 0.200	0.710 (±0.032)
E3_Top32	SVM	-K 2 –C 0.9 –G 0.175	0.861 (±0.027)

As we can observe from Table 6.2 for experiment 1, CA vs. CM, the best dataset/classifier was E1_top36 with a neural network; this means that by using a back propagation neural network with a learning rate of 0.3 (-L 0.3), a momentum of 0.2 (-M 0.2), during 150 epochs (-N 150), and with 2 units in the hidden layer (-H 2) to classify the class counts within the top 36 matches we obtained an area under the curve (AUC) of 0.781 with a standard deviation of 0.022. The AUC was calculated using the default method provided by Weka, which for the multilayer perceptron (back propagation neural network) produces a receiver operating characteristic (ROC) curve [21] by modifying the threshold of the output unit to determine what class the instance belongs to. Once the ROC curve has been determined the area under the curve is calculated.

In [18] [19], each classification problem was evaluated using a five-fold cross-validation and ROC curves. In order to determine statistical significance when comparing the results of current flow vectors against frequent subgraph kernel (FSG) we obtained an AUC average over a five-fold cross-validation. Since we do not have the variance of the AUC for the FSG results we will assume the same sampled pool variance as the one for current flow vectors. Table 5.3 shows the results for each of the three classification problems comparing current flow vectors to FSG.

Table 6.3 Statistical significance of the results for HIV00

Class. Problem	AUC-CF	AUC-FSG (cost 1.0)	Mean Diff	STDev	Non-Paired T-test	Confidence Level
(1) CA vs CM	0.781	0.774	0.007	0.021	0.50309	68% (win)
(2) CA+CM vs CI	0.715	0.742	-0.027	0.013	-3.2839	98% (loss)
(3) CA vs CI	0.865	0.839	0.026	0.020	1.868619	93% (win)

As we can observe from the results, current flow vectors performed better on classification problems (1), and (3). In classification problem (2) FSG outperformed current flow vectors. Based on the analysis of statistical significance we can see that in classification problem (1) current flow vectors performed better but only with a 68% level of confidence that there is statistical significance. On the other hand, performance on classification problem (3) showed statistical significance at 93% favoring the results obtained when using current flow vectors. In classification problem (2) FSG outperformed current flows and it is clear given the high level of confidence, 98%, that the results for this particular problem were better than current flow vectors.

Similar results were obtained when applying the most accurate classifiers to the class count results for the HIV10.cfv dataset. As noted previously from the averages between the HIV00.cfv and the HIV10.cfv dataset, class counts for both datasets are nearly identical. Table 6.4 shows the results for the HIV10.cfv dataset.

Table 6.4 Statistical significance of the results for HIV10

Class. Problem	AUC-CF	AUC-FSG (cost 1.0)	Mean Diff	STDev	Non-Paired T-test	Confidence Level
(1) CA vs CM	0.779	0.774	0.005	0.025	0.31623	62% (win)
(2) CA+CM vs CI	0.717	0.742	-0.025	0.014	-2.82346	98% (loss)
(3) CA vs CI	0.867	0.839	0.028	0.019	2.33010	97% (win)

CHAPTER 7

SUMMARY AND FUTURE WORK

7.1 Summary

Current flow analysis in electrical networks as a tool for error-tolerant graph matching holds the potential to be a very powerful approach for structural graph similarity. This technique can prove very valuable in datasets where the topological information of the graphs holds most of the information; by including nodal information during the current flow calculation, the incorporation of the information stored in the vertex labels is taken into account while generating a current flow vector to represent a particular graph. Examples of such graph datasets are chemical compound datasets, fingerprint matching, and handwriting recognition datasets. As shown in our empirical results, current flow analysis emerges as a promising technique to detect graph isomorphisms, even on datasets where the vertex label information is important, as seems likely in the case of chemical compounds. Similar graph structures could provide hints about the chemical composition, and current flow similarity could yield good results to find similar compounds.

The potential of current flow analysis for graph classification is very promising as demonstrated by the results obtained on the NCI-HIV dataset. Comparing the results obtained using current flow vectors (CFV) against the frequent subgraph kernel (FSG) we observed that for experiment number 1, CA vs. CM, the results were about the same. For experiment number 2, CA+CM vs. CI, FSG is better than our approach; and for experiment number 3, CA vs. CI, current flow vectors produced better results. Based on these results, it is encouraging to see a somehow competitive performance given the fact that it was the first set of experiments for a new technique. The usage of class counts is only one of many options available to utilize the results provided by current flow vectors analysis. The usage of a voting mechanism between the different classifiers created by using different top Ns matches could be another avenue to investigate and hopefully obtain better results. Another use of the results produced by comparing the current flow vectors of a graph database is to classify graph structures with kernel methods. As mentioned before the characteristics of a function $k : G \times G \to R$ to be referred as a graph kernel is to be a valid positive kernel. Since *k* is symmetric and non-negative, it can make up a positive definite matrix.

7.2 Future work

Future work analyzing other datasets and further exploration of the classification capabilities of the current flow similarity measure is needed in order to develop the full potential of this promising technique. The usage of match values to make up a graph kernel that incorporates into a support vector machine or other kernel-based algorithm that isolate the learning algorithm from the instances, in other words, the learning algorithm does not need to access any of the information contained in the graph directly.

Further work on the area of visual comparison is also needed in order to consolidate current flow analysis as a viable technique for graph comparison. As mentioned before, experiments on image databases in order to find similar images would be a great area of research to apply our technique. Different methods to incorporate nodal information into the current flow calculation can be adapted depending on the dataset to be evaluated. In the case of images, similar color information could add similar resistance values to the edges connected by specific nodes.

REFERENCES

[1] B. T. Messmer and H. Bunke, “A new algorithm for error-tolerant subgraph isomorphism detection,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 20, no. 5, pp. 493–504, 1998.

[2] M. Neuhaus and H. Bunke, “Edit distance-based kernel functions for structural pattern classification,” Pattern Recogn., vol. 39, no. 10, pp. 1852–1863, 2006.

[3] H. Bunke and K. Shearer, “A graph distance metric based on the maximal common subgraph,” Pattern Recogn. Lett., vol. 19, no. 3-4, pp. 255–259, 1998.

[4] D. Justice, “A binary linear programming formulation of the graph edit distance,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 28, no. 8, pp. 1200–1214, 2006, fellow-Alfred Hero.

[5] M.-L. Fernández and G. Valiente, “A graph distance metric combining maximum common subgraph and minimum common supergraph,” Pattern Recogn. Lett., vol. 22, no. 6-7, pp. 753–758, 2001.

[6] W. D. Wallis, P. Shoubridge, M. Kraetz, and D. Ray, “Graph distances using graph union,” Pattern Recogn. Lett., vol. 22, no. 6-7, pp. 701–704, 2001.

[7] C. Faloutsos, K. S. McCurley, and A. Tomkins, “Fast discovery of connection subgraphs,” in KDD ’04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM Press, 2004, pp. 118–127.

[8] P. G. Doyle and J. L. Snell, “Random walks and electric networks,” Mathematical Association America, vol. 22, 1984.

[9] B. Scholkopf and A. J. Smola, Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. Cambridge, MA, USA: MIT Press, 2001.

[10] DTP, “AID2DA99 42,689 2d structures with aids test data as of october 1999, in sdf format.” Downloaded from http://cactus.nci.nih.gov/ncidb/download.html, Oct. 1999.

[11] D. J. Cook and L. B. Holder, Mining Graph Data. John Wiley & Sons, 2007.

[12] L. O. Hall and A. Hildoer, “Compound matching using current flows,” 2006.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 2nd ed. The MIT Press, 2001.

[14] R. Pozo, “TNT Home Page,” 2004. [Online]. Available: http://math.nist.gov/tnt/

[15] A. Dalby, J. G. Nourse, W. D. Hounshell, A. K. I. Gushurst, D. L. Grier, B. A. Leland, and J. Laufer, “Description of several chemical structure file formats used by computer programs developed at molecular design limited,” Journal of Chemical Information and Computer Sciences, vol. 32, no. 3, pp. 244–255, 1992.

[16] E. Gansner, E. Koutsofios, and S. North, “Drawing graphs with dot,” AT&T Bell Laboratories, Murray Hill, NJ, USA, Technical Report, Feb. 2002. [Online]. Available: http://www.research.att.com/sw/tools/graphviz/dotguide.pdf

[17] C. Borgelt and M. R. Berthold, “Mining molecular fragments: Finding relevant substructures of molecules,” in ICDM ’02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM’02). Washington, DC, USA: IEEE Computer Society, 2002, p. 51.

[18] M. Deshpande, M. Kuramochi, and G. Karypis, “Automated approaches for classifying structures,” in BIOKDD, 2002, pp. 11–18.

[19] _____, “Frequent sub-structure-based approaches for classifying chemical compounds,” in ICDM ’03: Proceedings of the Third IEEE International Conference on Data Mining. Washington, DC, USA: IEEE Computer Society, 2003, p. 35.

[20] S. Kramer, L. D. Raedt, and C. Helma, “Molecular feature mining in HIV data,” in KDD ’01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM, 2001, pp. 136–143

[21] I. H. Witten, E. Frank, Data Mining: practical machine learning tools and techniques, 2nd ed. Morgan Kaufmann Publishers, 2005.