# GAIA: Graph Classification Using Evolutionary Computation

Ning Jin
University of North Carolina
at Chapel Hill
Chapel Hill, NC, USA
njin@cs.unc.edu

Calvin Young
University of North Carolina
at Chapel Hill
Chapel Hill, NC, USA
youngc@cs.unc.edu

Wei Wang
University of North Carolina
at Chapel Hill
Chapel Hill, NC, USA
weiwang@cs.unc.edu

## ABSTRACT
Discriminative subgraphs are widely used to define the feature space for graph classification in large graph databases. Several scalable approaches have been proposed to mine discriminative subgraphs. However, their intensive computation needs prevent them from mining large databases. We propose an efficient method GAIA for mining discriminative subgraphs for graph classification in large databases. Our method employs a novel subgraph encoding approach to support an arbitrary subgraph pattern exploration order and explores the subgraph pattern space in a process resembling biological evolution. In this manner, GAIA is able to find discriminative subgraph patterns much faster than other algorithms. Additionally, we take advantage of parallel computing to further improve the quality of resulting patterns. In the end, we employ sequential coverage to generate association rules as graph classifiers using patterns mined by GAIA. Extensive experiments have been performed to analyze the performance of GAIA and to compare it with two other state-of-the-art approaches. GAIA outperforms the other approaches both in terms of classification accuracy and runtime efficiency.

## Categories and Subject Descriptors
H.2.8 [Database management]: Database Applications---data mining; I.5.2 [Pattern Recognition]: Design Methodology---Classifier design and evaluation; Feature evaluation and selection

## General Terms
Algorithms, Experimentation, Performance

## Keywords
Graph mining, graph classification

## 1. INTRODUCTION
Graphs can be used to represent complex structural information in many scientific applications, including chemical compound structures, 3-D protein structures, and program dependence graphs and so on. There is a great need for building automated graph classification models and identifying discriminative graph features that separate different graph classes. For example, chemists want to be able to predict which chemical compounds are toxic and which components are characteristics of chemical toxicity [Helma, 2004]; biologists are interested in studying which proteins are able to bind certain ligands and which can be used to treat diseases [Bandyopadhyay, 2006]; computer scientists seek to find out how to locate bugs in programs by identifying discriminative subgraphs in program flow graphs [Cheng, 2009]. Performing these classification tasks by hand is intractable computationally, thus increasing attention has been devoted in developing graph classification methods in recent years.


*SIGMOD'10*, June 6–10, 2010, Indianapolis, Indiana, USA.


### 1.1 Related Work
Existing research often assumes a binary graph classification task where a target graph set and a background graph set are given and the objective is to construct a classification model for distinguishing them. One straightforward solution [Deshpande, 2005; Bandyopadhyay, 2006] to graph classification is first finding frequent subgraph patterns [Inokuchi, 2000; Kuramochi, 2001; Yan 2002; Huan 2003] in the target graph set and then selecting as features those that rarely occur in the background set. The subgraph features can be used to represent each graph as a feature vector and the problem of graph classification thus converts to classification of high dimensional data points. One major drawback of this approach is that it can find an enormous quantity of frequent subgraph patterns in the target set, which inhibits its ability to handle large datasets.

To overcome this drawback, recent approaches search directly for discriminative subgraph patterns that can better assist graph classification rather than for frequent subgraph patterns which may not necessarily be more discriminative. Leap [Yan, 2008] is a pioneer in discriminative subgraph pattern mining. It looks for the optimal subgraph pattern in terms of discrimination power with a branch-and-bound technique, taking advantage of the fact that structurally similar subgraphs tend to have similar discrimination power. It also uses a technique called "frequency descending mining" to exploit the correlation between subgraph frequency and subgraph discrimination power. Another algorithm gPLS [Saigo, 2008] adopts the powerful mathematical tool of partial least squares regression for discriminative subgraph pattern mining to collect informative subgraph patterns and build a classifier directly. CORK [Thoma, 2009] proposes to use correspondence to measure the discrimination power of subgraph patterns and thereby achieves a theoretically near-optimal solution. Given a set of subgraph patterns, the number of correspondences is the total number of pairs of graphs that these subgraphs cannot discriminate.

All three of these algorithms are theoretically sound and can guarantee optimal or near-optimal solutions in some sense. They

outperform previous graph classification methods considerably both in terms of speed and classification accuracy. However, the pursuit of theoretical optimality led to relatively poor runtime efficiency which became the focus of some more recent algorithms such as graphSig [Ranu, 2009] and COM [Jin, 2009]. The major contribution of graphSig [Ranu, 2009] is that it provides an efficient solution to mining discriminative subgraph patterns with extremely low frequencies. It first converts graphs to feature vectors by performing Random Walk with Restarts on each node. Then it divides graphs into small groups such that graphs in the same group have similar vectors. It mines frequent subgraphs in each group with high frequency thresholds because high similarity in vectors in the same group indicates that the corresponding graphs in the group share highly frequent subgraphs. It also proposes a k-NN classification method using the frequent subgraphs. COM [Jin, 2009] makes use of a heuristic subgraph exploration order to find discriminative patterns faster and use them to prune redundant subgraph patterns. In addition to subgraph patterns, it also takes into account co-occurrences of subgraph patterns. Co-occurrences of small subgraph patterns are often able to approximate large patterns and thereby significantly reduce the mining time for large patterns. Moreover, there are cases where co-occurrences of subgraph patterns are able to discriminate graphs where individual subgraph patterns fail. COM uses association rules as classifiers instead of Support Vector Machine to improve the efficiency in classifier construction and interpretability of the classifier. Both COM and graphSig are much faster than Leap. GraphSig produces higher accuracy than Leap in classifying chemical compounds. COM has comparable classification accuracy to that of Leap for chemical compounds but gives better accuracy for proteins.

In addition to the discriminative subgraph pattern mining algorithms mentioned above, there is some other recent work aiming at improving the efficiency of subgraph mining. [Hasan, 2009] proposes the idea of output space sampling in the domain of frequent subgraph mining, which is a generic sampling approach and can use different probability distributions to focus on different types of subgraph patterns. One of its applications is to sample discriminative subgraph patterns. It uses Ullmann's subgraph-isomorphism [Ullmann, 1976] algorithm to compute the support of subgraph patterns instead of using embeddings of subgraph patterns because it visits subgraph patterns randomly and embeddings may be unavailable when it visits a pattern from its super-pattern. As a result, its runtime efficiency is relatively poor. In addition, as is mentioned in the paper, it is difficult to find a probability distribution for discriminative subgraph pattern sampling. Another recent frequent subgraph mining algorithm, SUMMARIZE-MINE [Chen, 2009], attempts to solve the problem that arises with large graph databases whose embedding information requires more storage than is available in memory. SUMMARIZE-MINE randomly merges some nodes with the same labels into one node and compresses edges accordingly to reduce the size of graph databases. It performs the ad-hoc compression several times independently to reduce the probability of false negatives.

## 1.2 Our Contribution

In this paper, we propose a novel algorithm GAIA (**G**raph cl**A**ssification with evolut**I**onary comput**A**tion) to mine discriminative subgraph patterns for graph classification. We introduce a novel subgraph encoding method using the notion of conditional canonical adjacency matrix. Given a graph database and the embedding information of a subgraph pattern we are able to calculate its canonical sequence representation in $O(|V|^2)$ instead of the exponential time needed in previous methods, where $|V|$ is the number of nodes in the pattern. We also apply evolutionary computation, which is a randomized searching strategy for optimal solution which simulates biological evolution, to look for discriminative subgraph patterns. To the best of our knowledge, this is the first work to introduce evolutionary computation to the field of discriminative subgraph mining. The major difficulty of using evolutionary computation to find discriminative subgraphs is that there is no existing subgraph exploration method that can explore subgraph patterns randomly and track such exploration in an efficient way. However, randomized exploration order is essential to the success of evolutionary computation. We overcome this difficulty by using the novel subgraph encoding method. Using evolutionary computation also enables us to take advantage of the more and more widely available parallel computing resources. We improve the quality of resulting subgraph patterns by running many instances of the algorithm in parallel and then generate a consensus result that has better discrimination power than any resulting set from an individual execution.

## 1.3 Organization

The remainder of this paper is organized as follows. We introduce the notion of conditional canonical adjacency matrix and the subgraph encoding method in Section 2. Section 3 describes the framework and mechanisms of the evolutionary computation used in GAIA. Section 4 explains how we generate classifiers with discriminative subgraph patterns and how we perform parallel computing to improve the quality of resulting patterns. Experimental results are given in Section 5. Section 6 concludes the paper.

# 2. ENCODING SUBGRAPH PATTERNS WITH CONDITIONAL CANONICAL ADJACENCY MATRIX

**DEFINITION 1 (Graph)**. A graph is denoted by $g = (V, E)$, where $V$ is a set of nodes and $E$ is a set of edges connecting the nodes. Each graph in the graph database has a unique graph ID starting from 1. In a graph, each node has a unique ID starting from 1. Both nodes and edges may have labels.

For example, in Figure 1, there are two graphs in the graph database and they have graph IDs 1 and 2 respectively. The text in each node is in the form of (node ID : node label). Two nodes in a graph may have the same label but they cannot have the same node ID. Two nodes in two different graphs can have the same node ID but they do not necessarily represent the same entity and may have different labels. For simplicity, the presence or absence of an edge can be encoded by a binary label (1 for presence and 0 for absence).

**DEFINITION 2 (Subgraph Isomorphism)**. The label of a node with node ID $u$ is denoted as $label(u)$ and the label of an edge $(u, v)$ is denoted as $label((u, v))$. For two graphs $g$ and $g'$, if there is an injection $f: V(g) \rightarrow V(g')$, such that for any node with node ID $u$ in $V(g)$, $label(u) = label(f(u))$ and for any edge $(u, v)$ in $E(g)$, $label((u, v)) = label((f(u), f(v)))$, then $g$ is a subgraph of $g'$ and $g'$ is a supergraph of $g$, or $g'$ supports $g$.

For example, in Figure 1, Pattern 1 is a subgraph of both Graph 1 and Graph 2.

**DEFINITION 3 (Embedding)**. Given a subgraph isomorphism injection *f: V(g)* → *V(g')*, the node set {*f(u)* | *u* ∈ *V(g)*} is an embedding of *g* in *g'*. *g* can have multiple embeddings in *g'* because there may exist more than one injection. A sorted embedding organizes the nodes in an embedding in increasing order of their node IDs. An embedding code *B* is the concatenation of the graph ID of *g'* and the sorted embedding. The first element in an embedding is the graph ID and the remaining elements are node IDs.

For example, in Figure 1, Pattern 1 has two embeddings in Graph 1, which are {1, 2, 3} and {1, 5, 6} in the form of sorted embeddings, and one embedding in Graph 2, which is {1, 2, 5}. In total, Pattern 1 has three embedding codes: <1, 1, 2, 3>, <1, 1, 5, 6> and <2, 1, 2, 5>.

**DEFINITION 4 (Adjacency Matrix).** Given an embedding code *B* of pattern *p* based on a subgraph isomorphism injection *f*, the adjacency matrix *M* of *p* is a $|V| \times |V|$ matrix, where *V* is the node set of pattern *p* and each entry of *M* [1] satisfies:

$$M[i,j] = \begin{cases} label(B[i]), & i = j \\ label(B[i], B[j]), & i \neq j \end{cases}$$

always exists Figure 2 shows the three adjacency matrices corresponding to the three different embedding codes of pattern *p*.

**DEFINITION 5 (Matrix Code).** The matrix code of a subgraph pattern *p* is the sequence formed by row-wise concatenation of the lower triangle entries of an adjacency matrix *M* of *p*.

For example, the matrix codes corresponding to Matrices 1, 2, 3 in Figure 2 are N1C01C, N0C11C and C1C10C, respectively.

**DEFINITION 6 (Conditional Canonical Adjacency Matrix).** Given a graph database, where each graph has a unique graph ID, the conditional canonical adjacency matrix of a subgraph pattern *p* is the adjacency matrix corresponding to the lexicographically smallest embedding code of *p*.

For example, in Figure 1, given the graph database composed of Graph 1 and Graph 2, the conditional canonical adjacency matrix of Pattern 1 is Matrix 1 in Figure 2.

It is "conditional" because only when a graph database is given can the canonical adjacency matrix be defined and generated. It is "canonical" because as long as a graph database is given, two isomorphic subgraph patterns must have the same conditional canonical adjacency matrix since two isomorphic subgraph patterns must have the same embeddings and therefore the same lexicographically smallest embedding code.

**DEFINITION 7 (CCAM Code).** Given a graph database, where each graph has a unique graph ID, the CCAM Code of a subgraph pattern *p* is the matrix code corresponding to the conditional canonical adjacency matrix of *p*.

For example, in Figure 1, given the graph database composed of Graph 1 and Graph 2, the CCAM code of Pattern 1 is N1C01C.

Given a graph database, two isomorphic subgraph patterns must have the same CCAM code because they have the same conditional canonical adjacency matrix.

[1] All subscripts are indexed starting at 1.

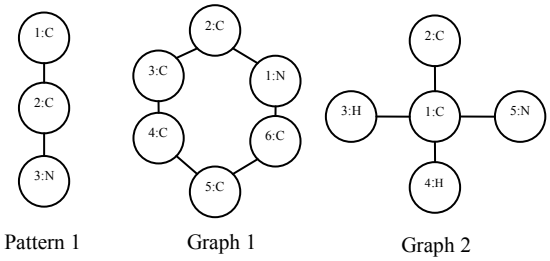


**Figure 1: an example of graph and subgraph pattern**

| | | |
|---|---|---|
| N | 1 | 0 |
| 1 | C | 1 |
| 0 | 1 | C |

Matrix 1: <1, 1, 2, 3>

| | | |
|---|---|---|
| N | 0 | 1 |
| 0 | C | 1 |
| 1 | 1 | C |

Matrix 2: <1, 1, 5, 6>

| | | |
|---|---|---|
| C | 1 | 1 |
| 1 | C | 0 |
| 1 | 0 | N |

Matrix 3: <2, 1, 2, 5>

**Figure 2: three adjacency matrices of pattern C-C-N**

Previous subgraph pattern encoding methods, such as minimum DFS code [Yan, 2002] and CAM code [Huan, 2003], only look at the structural information of the pattern, but do not take advantage of the embedding information. The complexity of computing any type of canonical code from a graph is NP-complete because it is proven to be equivalent to solving subgraph isomorphism which is NP-complete. However, in all efficient subgraph pattern mining algorithms, such as FFSM [Huan 2003], SPIN [Huan 2004] and COM [Jin, 2009], all embeddings of a pattern are actually already maintained and sorted in increasing order of graph IDs by the algorithms in order to calculate pattern frequency efficiently. Therefore, the embedding information is available when a subgraph pattern mining algorithm computes canonical codes. Given embeddings of a pattern *p* in a graph database sorted by graph IDs, the complexity of computing CCAM code of *p* can be reduced to $O(|V|^2)$, where $|V|$ is the number of nodes in *p*. The computation can be completed in three steps:

1. Retrieve embeddings with the smallest graph ID
2. For each embedding, sort the node IDs in ascending order and keep track of the lexicographically smallest embedding code *B*
3. Construct the conditional canonical adjacency matrix according to *B* and generate the CCAM code

The complexity of the first step can be considered as $O(1)$ because the embeddings are already sorted and the number of embeddings with the smallest graph ID can be upper-bounded by a small constant in most applications. The complexity of the sorting step is $O(|V| * lg\ |V|)$ where $|V|$ is the number of nodes in *p* because the number of embeddings from Step 1 is considered as a constant. The complexity of the third step is $O(|V|^2)$ because the size of the matrix is $O(|V|^2)$. Therefore, we can compute CCAM code in $O(|V|^2)$ time by taking advantage of embedding information that has been calculated already. This significant improvement in time efficiency is essential to GAIA because GAIA does not require a frequency threshold and therefore cannot prune subgraph patterns based on frequency. Most other subgraph

mining algorithms, such as gSpan [Yan, 2002], FFSM [Huan, 2003], SPIN [Huan, 2004], Leap [Yan, 2008], gPLS [Saigo, 2008] and COM [Jin, 2009], use a frequency threshold to limit the examination to only frequent subgraph patterns. In addition, using CCAM code allows arbitrary edge extensions to a subgraph pattern while previous encoding methods only allows certain types of edge extensions in order to maintain canonical codes of patterns efficiently.

One potential challenge of this encoding method is that the number of embeddings in Step 2 may be large especially when the patterns are small. We solve this problem by encoding patterns differently according to their sizes: one-edge patterns are encoded with their minimum matrix codes and larger patterns are encoded with their CCAM codes.

# 3. MINING DISCRIMINATIVE SUBGRAPH PATTERNS WITH EVOLUTIONARY COMPUTATION

The first goal of our work is to find a set of discriminative patterns among which each positive graph can have at least one representative pattern for graph classification and we achieve this goal by exploring candidate patterns in a process resembling biological evolution, a.k.a. evolutionary computation, implementing some of the evolutionary mechanisms such as competition and migration. Evolutionary computation can be viewed as a generic search process for solutions of high quality or fitness, which begins with a set of sample points in the search space and gradually biases to regions of high fitness. In the problem of discriminative pattern mining, we define a fitness function to measure the potential classification power of a pattern and larger patterns that can be generated by subgraph extension. As a result, our evolutionary search process here is directed toward subgraph patterns with high classification power.

## 3.1 Fitness Function

We use two types of scores to measure the fitness of a subgraph pattern: log ratio score and score per edge.

**DEFINITION 8 (Positive Frequency and Negative Frequency).** The positive (negative) frequency $r^+(p)$ ($r^-(p)$) of a subgraph pattern $p$ is the ratio of the number of positive (negative) graphs containing $p$ to the total number of positive (negative) graphs in the graph database.

**DEFINITION 9 (Log Ratio Score).** The log ratio score of a subgraph pattern $p$ is a function of the positive and negative frequencies of $p$ and is defined as:

$$log\ ratio\ score\ of\ pattern\ p = log\frac{r^+(p)}{r^-(p)}$$

The log ratio score is used to measure the fitness of patterns. Its rationale is that if two patterns have the same positive frequency (negative frequency) then the one with lower negative frequency (higher positive frequency) is better for discriminating positive graphs from negative graphs and is thus fitter to survive. To solve the problem of denominator being zero, when calculating the negative frequencies, we add an imaginary negative graph that has all subgraph patterns. Thus, the negative frequency of any pattern $p$ is never zero.

This function is asymmetric w.r.t. positive and negative frequencies. It focuses on subgraph patterns with high positive frequency and low negative frequency rather than patterns with low positive frequency and high negative frequency. In many applications, such as structure-based protein and chemical compound classification, positive graphs are much more likely to share common discriminative subgraph patterns than negative graphs. This is because that positive graphs typically have some common characteristics (e.g. a biological function) while negative graphs (e.g., those lacking a biological function) are often diverse. However, even if negative graphs share some common discriminative subgraph patterns, these patterns can be found easily by switching the roles of positive graphs and negative graphs.

**DEFINITION 10 (Score per Edge).** The score per edge of a subgraph pattern $p$ is a function of the positive frequency $r^+(p)$, negative frequency $r^-(p)$ and number of edges $m(p)$ and is defined as:

$$score\ per\ edge\ of\ pattern\ p = \frac{log\frac{r^+(p)}{r^-(p)}}{m(p)}$$

The score per edge function is used to alleviate the risk of potential overfitting. For patterns of competitive log ratio scores, the score per edge function favors patterns containing fewer edges. This is necessary in finding discriminative patterns for graph classification because large patterns tend to have low positive and negative frequencies and thus are likely to be useless in classification. Spending time in finding such patterns is not worthwhile if there exist some smaller patterns having similar log ratio scores. Therefore, when we measure the fitness of a pattern $p$, we consider not only its log ratio score but also its score per edge. By doing this, we can avoid many seemingly discriminative but useless patterns. This function is analogous to the Minimum Description Length model in information theory in the sense that we use the number of edges to measure the description length.

## 3.2 Framework of the Pattern Evolution: Organization and Resources

For each graph $g_i$ in the positive graph set $G^+$, we store a representative subgraph pattern and a list of up to $s$ candidate subgraph patterns, where $s$ is bounded (from above) by the available memory space divided by the number of graphs. Figure 3 illustrates the organization of candidate patterns and representative patterns. Only subgraphs of $g_i$ with positive log ratio scores can be its representative or in its candidate list. The representative pattern has the highest log ratio score among all patterns that are subgraphs of $g_i$ found during pattern evolution. Although one pattern can be subgraphs of several positive graphs, each pattern can only be in one candidate list at any time. The candidate lists are initialized with one-edge patterns. Whether and where a pattern will be placed in candidate lists will be discussed in Subsection 3.4.

The total number of subgraph patterns that the candidate lists can hold at any time is the product of $s$ and $|G^+|$. The motivation of the design of this framework is to cause selection pressure which can significantly speed up the convergence of evolutionary search. When the total size of candidate lists is less than the total number of patterns that can be found in positive graphs, not all patterns can be held in the candidate lists at the same time. As a result, one resource that candidate patterns need to compete for is a slot in candidate lists. In other words, patterns have to compete for survival and not all patterns are considered in the search process.

Generally speaking, the larger the candidate lists are, the less selection pressure there is and thereby the more patterns are considered in the search. When the candidate lists are infinitely large, the search process becomes an exhaustive search.

Another resource that candidate patterns compete for is the opportunity to extend or, analogous to biological evolution, to produce offspring. All subgraph pattern mining algorithms start with small subgraph patterns and then extend them into larger patterns. However, pattern extension is a costly operation and not every pattern extension leads to a discriminative pattern. In an evolutionary search process, candidate patterns compete for the opportunity of pattern extension according to their fitness, which enables the search process to focus on candidate patterns that are more likely to lead to discriminative patterns. Although it does not guarantee that it reaches the globally optimal solution faster because of the existence of local optimal solutions, our experiments show that in reality it has significant speed advantage over other methods.

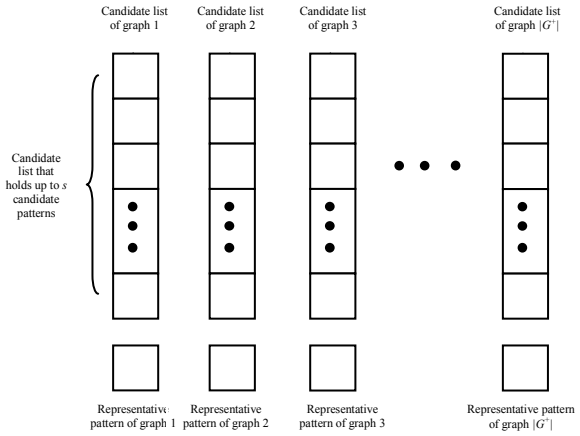


**Figure 3: organization of candidate patterns**

## 3.3 Pattern Extension

All candidate patterns currently in the candidate lists have a non-zero probability of being selected for pattern extension. To perform pattern evolution, GAIA runs for $n$ iterations, where $n$ is a parameter set by the user. During each iteration, we select one pattern from each candidate list for extension. The probability of pattern $p$ in candidate list of $g_i$ to be selected for extension is proportional to the log ratio score of $p$ and is calculated as follows:

$$Probability(pattern\ p\ gets\ extended) = \frac{log\_ratio\_score(p)}{\sum_{p' is\ in\ the\ candidate\ list\ of\ g_i} log\_ratio\_score(p')}$$

The probability is always between 0 and 1 because only patterns with positive log ratio scores are allowed in candidate lists as described in Subsection 3.2. This selection method is commonly used in evolutionary algorithms and an analysis on it can be found in [De Jong, 2006]. The intuition here is that candidate patterns with higher scores are more likely to be extended to patterns with high scores because structurally similar subgraph patterns have similar discrimination power [Yan, 2008]. Note that when $s$ = 1, each candidate list only holds 1 pattern. The probability of this pattern being selected for extension is 1. When $s$ > 1, multiple patterns may be held in a candidate list. A random number generator is used to determine which pattern is selected for extension according to their probabilities.

For an extension operation of pattern $p$, GAIA generates a pattern set $X(p)$ and each pattern $p'$ in $X(p)$ has one new edge attached to $p$. This new edge is not present in $p$ and it can be either between two existing nodes in $p$ or between one node in $p$ and a new node. Unlike many previous subgraph pattern mining algorithms that only extend patterns with certain types of edges in order to efficiently maintain their canonical codes, GAIA considers all one-edge extensions of pattern $p$ that occur in the positive graphs. This difference in extension operation is essential to GAIA because evolutionary computation is essentially a heuristic search for optimal solution. This difference enables GAIA to explore the candidate pattern space in any direction that appears promising.

Extensions of different patterns can produce the same pattern because a pattern $p$ with $k$ edges can be directly extended from all of its subgraphs with $k$-1 edges. Therefore, a lookup table is needed by GAIA to determine whether a pattern has already been generated to avoid repetitive examination of the same pattern. In our implementation of GAIA, we use *map* in *C++ STL* to implement the lookup table. The codes for pattern lookup are generated by the encoding method described in Section 2.

If pattern $p$ extends into pattern $p'$ and the log ratio score of $p'$ is less than that of $p$, then the edge extension is undesirable since it produces a "child" pattern that is less fit for survival than the parent. We consider such a decline in log ratio score as a sign of extending in a wrong direction and thus eliminate $p'$ from survival and further extension. It is also possible that no extensions of $p$ have better log ratio scores. In fact we can estimate a loose upper-bound of their scores before extending $p$. The log ratio score of any extension $p'$ reaches its upper-bound when $r^+(p) = r^+(p')$ and $r^-(p')=0$, where $r^+$ and $r^-$ are positive and negative frequencies respectively. If this upper-bound of log ratio score is no greater than that of $p$, then we do not need to extend $p$.

In addition to the log ratio score, we also use the "score per edge" measure to decide whether a new pattern should survive and have further extension. Assume that pattern $p$ extends into pattern $p'$. Even if the log ratio score of $p'$ is greater than that of $p$, the score per edge of $p'$ can still be less than that of $p$. If the score per edge function decreases steadily during successive extensions, then we want to prune these extensions. We quantify the change in score per edge between a pattern and its extensions by a variable called "momentum". If the score per edge value of a pattern $p'$ is greater than that of its parent pattern $p$, then the momentum value of $p'$ is the momentum value of $p$ added by 1; otherwise, the momentum value of $p'$ is the momentum value of $p$ subtracted by 1. All one-edge patterns whose log ratio scores are positive have momentum values being 1 (Patterns with non-positive log ratio scores are not allowed in the candidate lists). If a new pattern has its momentum value less than 0, this new pattern is pruned. We choose to prune patterns with decreasing score per edge values rather than patterns with low score per edge values because we consider a downward trend in score per edge as a stronger sign of unpromising extensions. In our experiments, using log ratio score to calculate momentum can result in 2-4 times longer runtime than using score per edge with the same classification accuracy because using log ratio scores considers more unpromising extensions.

## 3.4 Pattern Migration and Competition

In most cases, an extension operation on one pattern generates many new patterns and as a result the number of patterns found by the algorithm grows. Sooner or later the number of patterns will exceed the number of available positions in the candidate lists. It is also possible that the number of one-edge patterns already exceeds the number of available positions in the candidate lists at the very beginning if $s$ is small. Therefore some rules are needed to determine which patterns should survive in the candidate lists and which candidate list they should dwell in.

First, a pattern that has already been extended should not "live" in the candidate lists any longer because it has served its role in generating new patterns.

Second, some pattern in the candidate list may migrate to the candidate list of another graph if such migration will increase its chance of survival. Let $p$ be the candidate pattern for migration and $G(p)$ be the set of graphs containing $p$. Let $g_i$ be the graph in $G(p)$ which has the lowest value of $\sum_{p' is\ in\ the\ candidate\ list\ of\ g_i} log\_ratio\_score(p')$. $p$ will migrate to the candidate list of $g_i$. The rationale for this pattern migration is that if a pattern wants to survive then it should go to a candidate list with the least fierce competition. In GAIA, the fierceness of competition of a candidate list is measured by the sum of log ratio scores of patterns in the list.

If the candidate list of $g_i$ still has vacant positions, then $p$ can move into one vacant position directly. However, if the candidate list is already full, then $p$ has to compete with the "resident" patterns in the list. One straightforward approach to let $p$ compete with "resident" patterns is to compare the log ratio score of $p$ and the minimum log ratio score among "resident" patterns. If the score of $p$ is greater than the minimum score among "resident" patterns, then $p$ takes the position of pattern $p'$ with the minimum score and $p'$ no longer exists in any candidate list; otherwise, $p$ fails to survive and will not exist in any candidate list. The disadvantage of this greedy approach is that it ignores the fact that patterns with low log ratio scores may still have some potential to extend into patterns with high log ratio scores and patterns with high log ratio scores at the time may have reached their limits and will never extend to better patterns. Therefore, GAIA adopts a randomized method for pattern competition which is commonly used by evolutionary algorithms. The score of $p$ is compared against the score of a pattern $p'$, which is randomly selected with probability $1/s$ from the candidate list. If the score of $p$ is higher, then $p'$ is eliminated and $p$ takes the position of $p'$; otherwise, $p$ is eliminated. By doing so, GAIA can at least have a chance to protect some of the "weak" patterns and give them an opportunity to extend into "strong" patterns. The benefit of this randomized approach is more evident when $s$ is reasonably large. Note that when $s = 1$ the randomized strategy is essentially the same as the greedy strategy.

Again, the exhaustive extension operation is of great importance to allow pattern competition and elimination. When we eliminate a pattern $p$, the real loss is not only this pattern but also the patterns generated by extending $p$. In previous subgraph pattern mining algorithms, such as gSpan [Yan, 2002] and FFSM [Huan, 2003], a pattern $p$ can only be extended from one of its subpatterns, $p'$. If $p'$ is lost, then the algorithms will never find $p$. As a result, for these algorithms, allowing pattern elimination will surely lose many patterns, some of which are discriminative patterns. But in GAIA, eliminating $p'$ does not necessarily lead to the loss of $p$ because the exhaustive extension operation allows $p$ to be extended from many different patterns. As a result, the risk of missing discriminative patterns is much lower than other subgraph mining algorithms.

Figure 4 is an algorithmic description of pattern evolution in GAIA. The inputs are the positive graph set $G^+$, the negative graph set $G^-$, the (optional) maximum size of each candidate list, and the (optional) maximum number of iterations. The output is the representative patterns of the positive graphs.

```
Algorithm: Pattern_Evolution (G+, G-, n = INT_MAX,
                              s = available_space/number_of_positive_graphs)
G+: positive graph set
G-: negative graph set
s: maximum size of each candidate list,
   by default equal to available_space/number_of_positive_graphs
n: maximum number of iterations,
   by default the maximum interger value in the system
T: all candidate lists
H: lookup table of patterns that have already been found
1.    D = {all edges that occur in G+}
2.    for each edge e in D
3.       Migrate (e, T)
4.    for k = 1:n
5.       if (all candidate lists are empty)
6.          break
7.       for each g in G+
8.          randomly select a pattern p in the candidate list of g
9.          X (p) = {all patterns in G+ with one more edge attached to p}
10.            for each pattern p' in X (p)
11.               if (CCAM code of p' is in H)
12.                  continue
13.               insert p' into H
14.               Migrate (p', T)
15.               update representative patterns
```

**Figure 4: algorithm of pattern evolution**

```
Algorithm: Migrate (p, T)
p: a pattern
T: candidate lists
1.    g = argmin_g' (Σ_{p is in the candidate list of g'} log _ratio_score(p))
2.    if (the candidate list of g has vacant positions)
3.       insert p into the candidate list of g
4.    else
5.       randomly select a pattern p' in the candidate list of g
6.       if (log_ratio_score (p) > log_ratio_score (p'))
7.          replace p' with p
```

**Figure 5: algorithm of pattern migration**

# 4. GENERATING ASSOCIATION RULES FOR GRAPH CLASSIFICATION

Given a set of discriminative subgraph patterns, the second goal of our work is to generate a classifier to predict positive graphs and negative graphs. We choose association rule as the model of classifiers because of its success in COM [Jin, 2009] and its simplicity in computation and interpretation. However, different from COM, in each resulting association rule we only use one subgraph pattern instead of a combination of several patterns because (1) estimating combinations of disconnected patterns is time-consuming and (2) GAIA can locate large discriminative patterns far more efficiently than previous algorithms and thus does not need to approximate large patterns by combinations of small patterns.

## 4.1 Association Rules

**DEFINITION 11 (Association Rule).** In this paper, an association rule is defined as a classification rule in the form of $p \rightarrow L$, where $p$ is a subgraph pattern and $L$ is the class label. If an object has the subgraph pattern $p$, then it is classified as $L$. The output of GAIA is an association rule set $A$.

When the graph dataset only has two class labels, *positive* and *negative*, $L$ is always *positive* since we only look for patterns whose positive frequencies are higher than their negative frequencies (see Subsection 3.1) and thus a graph $g$ having the patterns in association rules indicates $g$ is a positive graph. When we use association rules generated by GAIA to classify a new graph $g$, as long as $g$ has any pattern in the association rules, we predict $g$ as positive. If $g$ does not have any pattern in the association rules, we predict it as negative.

**DEFINITION 12 (Sensitivity, Specificity and Normalized Accuracy).**

$$Sensitivity = \frac{\#\ of\ predictions\ that\ are\ positive\ and\ correct}{\#\ of\ graphs\ that\ are\ positive}$$

$$Specificity = \frac{\#\ of\ predictions\ that\ are\ negative\ and\ correct}{\#\ of\ graphs\ that\ are\ negative}$$

$$Normalized\ accuracy = \frac{Sensitivity + Specificity}{2}$$

Given the representative subgraph patterns from pattern evolution, we generate association rules by sequential coverage. First we sort the representative patterns by their log ratio scores in decreasing order. Then we traverse all representative patterns in the sorted order (that is we visit patterns with higher log ratio scores first). For each representative pattern $p$, we evaluate whether inclusion of a new association rule $p \rightarrow$ *positive* in the resulting association rule set $A$ can increase the normalized accuracy of classification of the training set. The new rule $p \rightarrow$ *positive* is to be included in $A$ if and only if the normalized accuracy increases. The generation algorithm terminates when all representative patterns have been tested. The algorithmic description is shown in Figure 6.

```
Algorithm: Sequential_Coverage (R, G)
R: a set of representative patterns from pattern evolution
G: training set
A: the resulting association rule set
1.    A is empty
2.    sort patterns in R by log ratio scores in decreasing order
3.    for i = 1 : |R|
4.       p = R[i]
5.       A' ← A
6.       A' ← A' ∪ {p → positive}
7.       t = normalized accuracy of classification on G using A
8.       t' = normalized accuracy of classification on G using A'
9.       if (t' > t)
10.         A ← A ∪ {p → positive}
```

**Figure 6: algorithm of generating association rules from representative patterns**

## 4.2 Generating Consensus Model with Parallel Computing

Because GAIA is a randomized algorithm (when $s > 1$), each single run of pattern evolution may generate different representative patterns and consume varying amount of CPU time. Some runs of pattern evolution may find better representative patterns than others and thus lead to classifiers with higher normalized accuracy. Therefore, if we run many instances of pattern evolution in parallel and generate an association rule set based on all representative patterns found by these instances of pattern evolution, it is very likely that we can build a better classifier than using representative patterns from one instance of pattern evolution alone. Therefore, by generating a consensus model based on many parallel instances of pattern evolution and only using the fastest instances of pattern evolution, we can improve the classification accuracy and expected response time by taking advantage of parallel computing. We choose to start association rule generation before all instances complete because, if we wait for all the pattern evolution instances, the runtime of GAIA will be determined by the runtime of the slowest instance (The time for sequential coverage is trivial). Let $c$ be the number of parallel instances of pattern evolution, which is a user specified parameter. We start generating association rules as soon as $\lceil c/2 \rceil$ instances of pattern evolution have terminated and only use the representative patterns found by these $\lceil c/2 \rceil$ instances. The association rule set is generated by the same algorithm *Sequential_Coverage* described in Figure 6 with all representative patterns found by the fastest $\lceil c/2 \rceil$ instances as input $R$.

# 5. EXPERIMENTS

The algorithm was implemented in C++ and compiled with g++. The experiments were performed on a 2.20 GHz dual core and 3.7 GB memory PC running Ubuntu Linux 9.10. We analyze the performance from two perspectives: runtime efficiency and normalized accuracy in graph classification applications. We evaluate two versions of GAIA: single-GAIA (when $c$=1) only performs one instance of pattern evolution to generate association rules and parallel-GAIA (when $c$>1 and $s$>1) runs in parallel $c$ instances of pattern evolution, where $c$ is a user-specified parameter, to generate association rules. For single-GAIA, we report the runtime of pattern evolution as the runtime of GAIA because the time for sequential coverage is trivial. For parallel-GAIA, we report the longest runtime of the first $\lceil c/2 \rceil$ instances of pattern evolution as the runtime of parallel-GAIA because it starts generating association rules as soon as $\lceil c/2 \rceil$ instances complete.

We use protein datasets and chemical compound datasets in our experiments. The protein datasets consist of protein structures from Protein Data Bank [2] classified by SCOP [3] (Structural Classification of Proteins). We use 16 protein datasets generated from all the large SCOP families with more than 25 members (listed in Table 1). In each dataset, protein structures in a selected family are taken as the positive set. Unless otherwise specified, we randomly select 256 other proteins (i.e., not members of the 16 families) as a common negative set used by all protein datasets. To generate a protein graph, each graph node denotes an amino acid, whose location is represented by the location of its alpha carbon. There is an edge between two nodes if the distance between the two alpha carbons is less than 11.5 angstroms. Nodes are labeled with their amino acid type and edges are labeled with the distances between the alpha carbons. On average, each protein graph has 250 nodes and 2700 edges. The chemical compound datasets consist of chemical compound structures from PubChem[4] classified by their biological activities, listed in Table 2. These are

[2] http://www.rcsb.org/pdb/

[3] http://scop.mrc-lmb.cam.ac.uk/scop/

[4] http://pubchem.ncbi.nlm.nih.gov

all the bioassays used in [Yan, 2008] and [Ranu, 2009]. Each compound can be either active or inactive in a bioassay. Unless otherwise specified, for each bioassay, we randomly select 400 active compounds as the positive set and 1600 inactive compounds as the negative set. The graph representation of compounds is straightforward. Each atom is represented by a node labeled with the atom type and each chemical bond is represented by an edge labeled with the bond type. On average, each compound graph has 55 nodes and 57 edges.

**Table 1: list of selected SCOP families**

| SCOP ID | Family name | Number of selected proteins |
|---|---|---|
| 46463 | Globins | 51 |
| 47617 | Glutathione S-transferase (GST) | 36 |
| 48623 | Vertebrate phospholipase A2 | 29 |
| 48942 | C1 set domains | 38 |
| 50514 | Eukaryotic proteases | 44 |
| 51012 | alpha-Amylases, C-terminal beta-sheet domain | 26 |
| 51487 | beta-glycanases | 32 |
| 51751 | Tyrosine-dependent oxidoreductases | 65 |
| 51800 | Glyceraldehyde-3-phosphate dehydrogenase-like | 34 |
| 52541 | Nucleotide and nucleoside kinases | 27 |
| 52592 | G proteins | 33 |
| 53851 | Phosphate binding protein-like | 32 |
| 56251 | Proteasome subunits | 35 |
| 56437 | C-type lectin domains | 38 |
| 88634 | Picornaviridae-like VP | 39 |
| 88854 | Protein kinases, catalytic subunit | 41 |

**Table 2: list of selected bioassays**

| Assay ID | Tumor description | Total number of actives | Total number of inactives |
|---|---|---|---|
| 1 | Non-Small Cell Lung | 2047 | 38410 |
| 33 | Melanoma | 1642 | 38456 |
| 41 | Prostate | 1568 | 25967 |
| 47 | Central Nerv Sys | 2018 | 38350 |
| 81 | Colon | 2401 | 38236 |
| 83 | Breast | 2287 | 25510 |
| 109 | Ovarian | 2072 | 38551 |
| 123 | Leukemia | 3123 | 36741 |
| 145 | Renal | 1948 | 38157 |
| 167 | Yeast anticancer | 9467 | 69998 |
| 330 | Leukemia | 2194 | 38799 |

For each experiment, we run GAIA (for both single-GAIA and parallel-GAIA) 5 times and report the average normalized accuracy and average runtime of the 5 runs. Note that GAIA is a randomized algorithm and each run may have slightly different classification accuracy and runtime even though the standard deviations are very small. For chemical datasets, standard deviations of normalized accuracies are less than 0.01 and standard deviations of runtimes are usually less than 1 second for single-GAIA and less than 0.1 second for parallel-GAIA. For protein datasets, standard deviations of normalized accuracies are usually less than 0.03 and standard deviations of runtimes are less than 0.1 seconds. Therefore, we only report the average in the following analysis.

## 5.1 GAIA Performance Analysis

In this subsection, we study the performance of GAIA with respect to three parameters: $s$ (maximum number of positions in a candidate list), $n$ (maximum number of iterations) and $c$ (number of instances of pattern evolution).

First, we run single-GAIA ($c$ = 1) with different $s$ and $n$ on the unbalanced chemical datasets and show the average normalized accuracy and average runtime in Table 3 and Table 4. In Table 3, $n$ is fixed at 4 and we can see the normalized accuracy is generally insensitive to the variation in $s$. When $n$ is large enough, larger $s$ enables GAIA to perform a more extensive search for discriminative patterns because it allows more candidate patterns to be stored in candidate lists and visited in the search process, which is why when the value of $s$ increases from 1 to 7 the normalized accuracy also increases. When the value of $s$ further increases, the normalized accuracy starts to decrease because although the size of candidate lists allows GAIA to perform a more exhaustive search, GAIA only runs for $n$ iterations and thus fails to take advantage of the large candidate lists. It can also be seen that although $n$ is fixed, the average runtime varies and is correlated with the normalized accuracy. This is because, generally speaking, the more discriminative a pattern is the more frequent and larger it is and thus the more time it takes to compute all of its embeddings and perform extensions. In Table 4, we fix $s$ at 10 and study the performance of single-GAIA with respect to $n$. means that pattern evolution terminates when all candidate lists are empty. We observe that increasing $n$ can effectively improve normalized accuracy when $n$ is small but only has marginal effect when $n$ is large. When $n$ is small, pattern evolution is far from convergence after $n$ iterations and larger $n$ can make the result closer to convergence. When $n$ is sufficiently large, pattern evolution is already near convergence and further increase in $n$ has little effect. Similarly, Table 4 also shows that larger $n$ results in longer runtime due to more iterations. We provide $n$ as an optional parameter for applications in which speed is crucial.

**Table 3: Normalized accuracy and average runtime of single-GAIA with different values of $s$, where $n$ = 4 (unbalanced chemical datasets)**

| $s$ | Normalized accuracy | Average runtime (sec) |
|---|---|---|
| 1 | 0.7295 | 2.3398 |
| 3 | 0.7329 | 2.7545 |
| 5 | 0.7310 | 2.8725 |
| 7 | 0.7330 | 2.8705 |
| 10 | 0.7298 | 2.7444 |
| 30 | 0.7311 | 2.4278 |
| 50 | 0.7300 | 2.4080 |

| | | |
|---|---|---|
| 70 | 0.7293 | 2.4207 |

**Table 4: Normalized accuracy and average runtime of single-GAIA with different values of *n*, where *s* = 10 (unbalanced chemical datasets)**

| *n* | Normalized accuracy | Average runtime (sec) |
|---|---|---|
| 1 | 0.7050 | 1.4192 |
| 2 | 0.7198 | 1.8795 |
| 4 | 0.7320 | 2.8066 |
| 8 | 0.7325 | 4.0611 |
| 16 | 0.7363 | 5.7075 |
| 32 | 0.7368 | 8.8772 |

Then we study the effect of *c* to the performance of GAIA. Figure 7 and Figure 8 show the average normalized accuracies of running GAIA with different *c* on chemical datasets and protein datasets respectively. Both figures illustrate that increasing *c* can result in higher average normalized accuracy. This positive correlation is due to the randomization in pattern evolution. Each instance of pattern evolution finds different representative patterns. One instance may be able to find a good representative pattern for $g_i$ but fail to find one for $g_j$ while another instance returns a good pattern for $g_j$ but not for $g_i$. Therefore when the representative patterns from different instances are merged together, the average quality of representative patterns can be improved. Generally, the larger the value of *c*, the better the normalized accuracy of GAIA. Figure 9 and Figure 10 show the average runtime of GAIA with different *c* on chemical datasets and protein datasets respectively. Both figures show the same trend of average runtime as *c* increases: runtime starts to converge when *c* is large. The runtime of GAIA is the sample median of the running times of *c* pattern evolution instances. When *c* is large enough, the runtime (sample median) should converge to the theoretical median of the runtime of all possible pattern evolution instances. Therefore, larger *c* leads to more stable runtime.

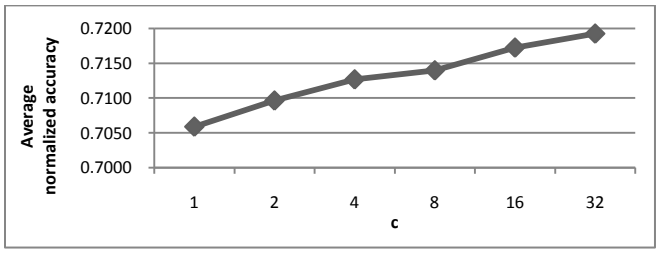


**Figure 7: average normalized accuracy vs. *c* (unbalanced chemical datasets)**

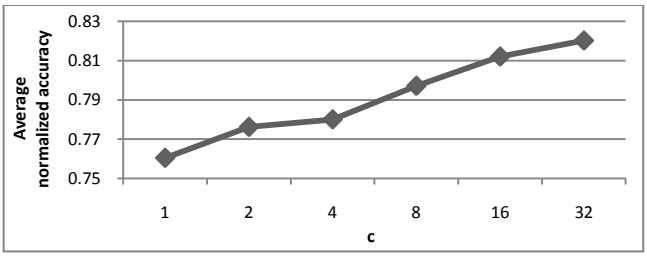


**Figure 8: average normalized accuracy vs. *c* (unbalanced protein datasets)**

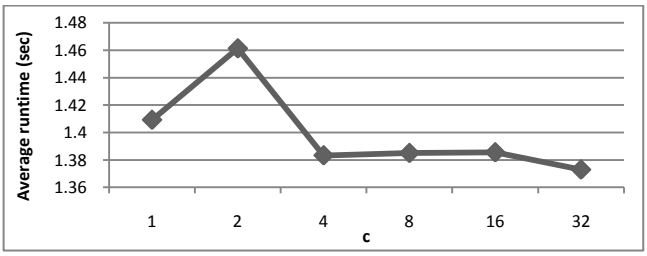


**Figure 9: average runtime vs. *c* (unbalanced chemical datasets)**

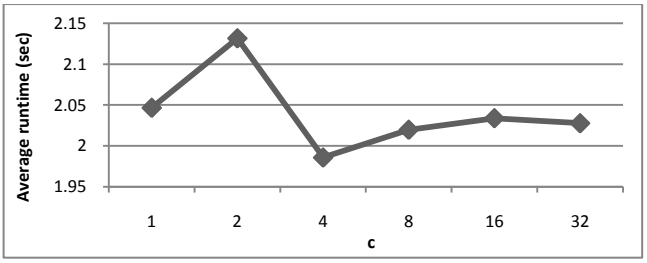


**Figure 10: average runtime vs. *c* (unbalanced protein datasets)**

Figure 11 and Figure 12 demonstrate the scalability of GAIA for chemical datasets as the number of positive graphs and number of negative graphs increase. In Figure 11, the number of negative graphs is fixed at 1600 and the number of positive graphs varies. The average runtime grows approximately linearly as the number of positive graphs increases. In Figure 12, the number of positive graphs is fixed at 400 and the number of negative graphs varies. We can see that the average runtime is linear to the number of negative graphs.

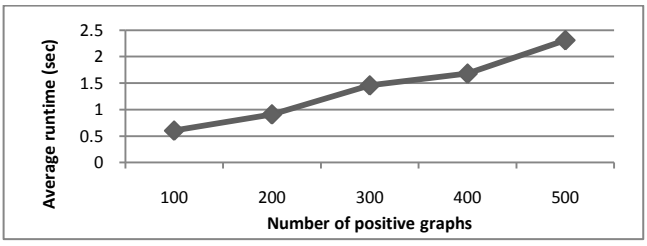


**Figure 11: average runtime vs. number of positive graphs (chemical datasets, number of negative graphs=1600)**

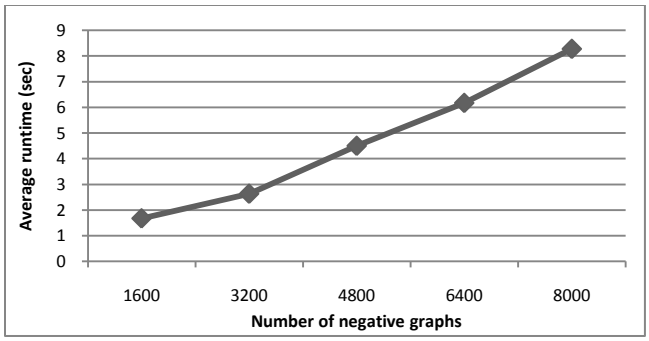


**Figure 12: average runtime vs. number of negative graphs (chemical datasets, number of positive graphs=400)**

## 5.2 Comparison with Other Methods

### *5.2.1 Parallel-GAIA*

We first compare parallel-GAIA (denoted as GAIA in this subsubsection for simplicity) with two other state-of-the-art graph classification methods: COM [Jin, 2009] and graphSig [Ranu, 2009], both of which claim to outperform Leap [Yan, 2008] in terms of runtime with competitive classification accuracy. COM also outperforms gPLS for the protein datasets. For GAIA, we use the parameters that give the best trade-off between runtime efficiency and classification accuracy. For COM and graphSig, we set the parameters that deliver the best results as suggested in [Jin, 2009] and [Ranu, 2009] respectively. The parameters for the three methods are summarized in Table 5. To compare with graphSig, we use the eleven chemical datasets and randomly sample 400 actives and 400 inactives from each dataset to form the training sets, because graphSig is implemented for balanced datasets. We also generate balanced protein datasets (number of negative graphs = number of positive graphs) to evaluate graphSig. We do not compare with Leap because graphSig outperforms Leap in chemical datasets and COM outperforms Leap in protein datasets with both faster speed and higher accuracy.

**Table 5: summary of parameters/environment**

| | Parameters for chemical datasets | Parameters for protein datasets |
|---|---|---|
| GAIA | $s$=10, $n$=4, $c$=32 | $s$=100, $n$=10, $c$=32 |
| COM | $t_p$=1%, $t_n$=0.4% | $t_p$=30%, $t_n$=0% |
| graphSig | maxPvalue=0.1, minFreq=0.1% | maxPvalue=0.1, minFreq=0.1% |

Figure 13 shows the normalized accuracy comparison between graphSig, GAIA and COM for balanced chemical datasets. GAIA delivers better normalized accuracy than graphSig on most datasets though not all of them. The average normalized accuracy of GAIA is 2.2% higher than that of graphSig. COM generally has lower normalized accuracy than GAIA and graphSig.

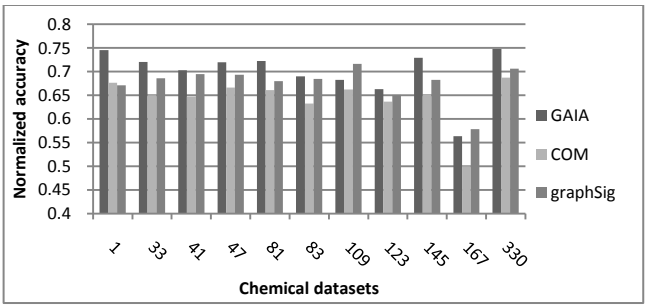


**Figure 13: normalized accuracy comparison for balanced chemical datasets between GAIA, COM and graphSig**

Figure 14 compares the runtime of graphSig, GAIA and COM for the balanced chemical datasets. GAIA demonstrates a huge advantage in runtime performance. For all datasets, GAIA is 20.44 times faster than graphSig on average. In addition, GAIA also outperforms COM considerably for every chemical dataset in terms of speed (on average 2.83 times faster).

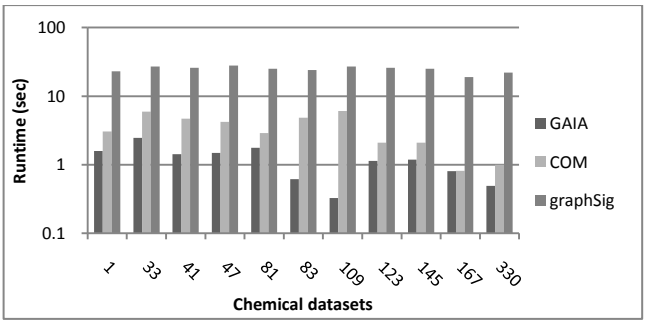


**Figure 14: runtime comparison for balanced chemical datasets between GAIA, COM and graphSig**

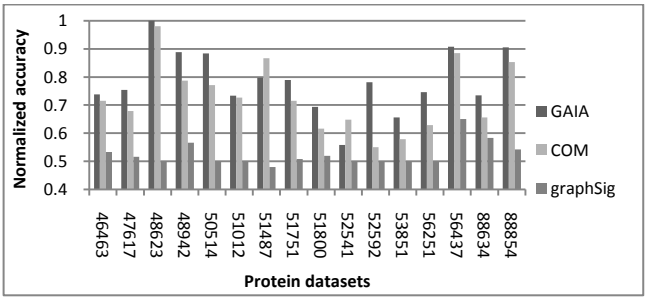


**Figure 15: normalized accuracy comparison for balanced protein datasets between GAIA, COM and graphSig**

Figure 15 and Figure 16 compare the normalized accuracy and average runtime performance respectively between graphSig, GAIA and COM. GraphSig is not comparable in processing protein datasets. Its speed is 2 orders of magnitude slower and its normalized accuracy is close to 0.5. This is mainly because graphSig is specifically designed and optimized for chemical compound datasets. Between GAIA and COM, GAIA is on average about 1.2 times faster and has a normalized accuracy 5.7% higher than that of COM.

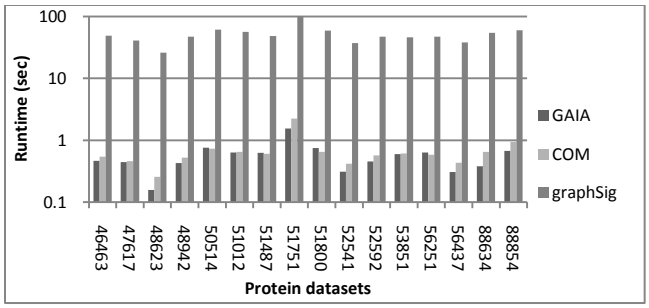


**Figure 16: runtime comparison for balanced protein datasets between GAIA, COM and graphSig**

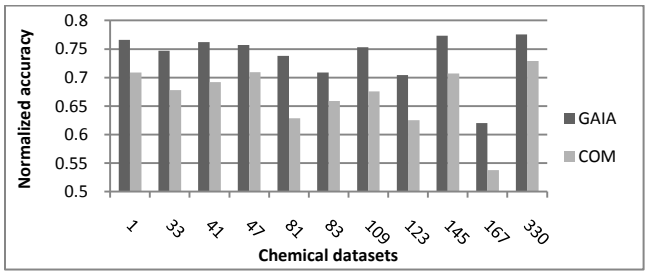


**Figure 17: normalized accuracy comparison for unbalanced chemical datasets between GAIA and COM**

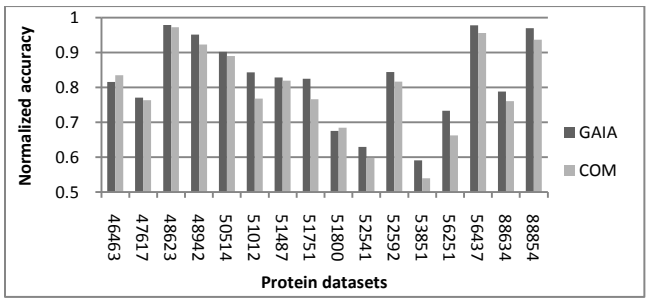


**Figure 18: normalized accuracy comparison for protein datasets between GAIA and COM**

Since both GAIA and COM can handle unbalanced datasets, we use the unbalanced chemical datasets and unbalanced protein datasets described at the beginning of this section to provide additional comparison. Figure 17 and Figure 18 show the normalized accuracy comparison. For chemical datasets, normalized accuracy of GAIA is 6.86% higher than that of COM on average. For protein datasets, normalized accuracy of GAIA is 2.7% higher than that of COM on average.

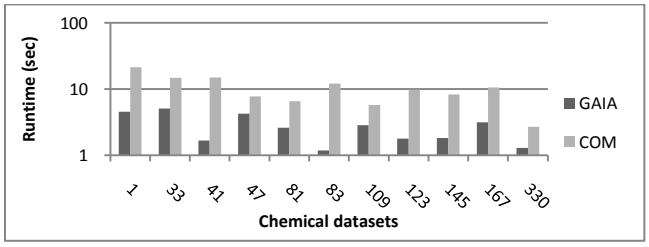


**Figure 19: runtime comparison for unbalanced chemical datasets between GAIA and COM**

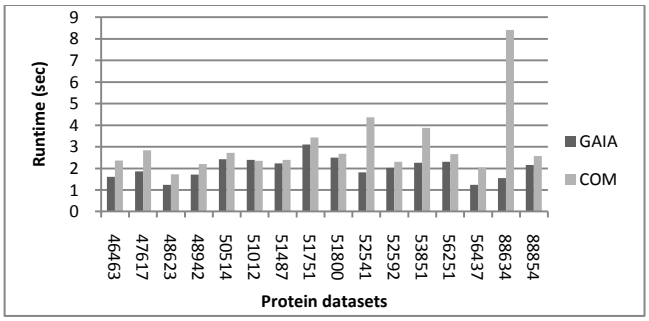


**Figure 20: runtime comparison for unbalanced protein datasets between GAIA and COM**

Figure 19 and Figure 20 compare the runtimes of the two methods for chemical datasets and protein datasets respectively. On average, GAIA is 3.8 times faster than COM for chemical datasets and 1.5 times faster than COM for protein datasets.

### *5.2.2 Single-GAIA*

Considering that parallel-GAIA demands more computation resources than COM and graphSig when outperforming them, we also compare single-GAIA with the other two methods to further demonstrate the advantage of GAIA. Table 6 summarizes the average runtime and average normalized accuracy comparisons between single-GAIA, parallel-GAIA, COM and graphSig using the datasets used in Subsubsection 5.2.1. The parameters are the same as listed in Table 5 except that for single-GAIA $c$=1. It can be seen in Table 6 that even without parallel computing, single-GAIA can excel COM and graphSig in terms of both average runtime and normalized accuracy. The only exception is that the average normalized accuracy of single-GAIA for the unbalanced protein datasets is 3.27% lower than that of COM.

**Table 6: summary of comparison between single-GAIA, parallel-GAIA, COM and graphSig**

| | | Single-GAIA | Parallel-GAIA | COM | graphSig |
|---|---|---|---|---|---|
| Balanced chemical datasets | Runtime (sec) | 1.296 | **1.210** | 3.430 | 24.73 |
| | Accuracy | **0.7029** [5] | 0.6988 | 0.6428 | 0.6768 |
| Balanced protein datasets | Runtime (sec) | 0.5996 | **0.5748** | 0.6788 | 51.13 |
| | Accuracy | 0.7665 | **0.7855** | 0.7285 | 0.5250 |
| Unbalanced chemical datasets | Runtime (sec) | 2.807 | **2.752** | 10.44 | N/A |
| | Accuracy | 0.7320 | **0.7368** | 0.6682 | N/A |
| Unbalanced protein datasets | Runtime (sec) | 2.047 | **2.028** | 3.059 | N/A |
| | Accuracy | 0.7605 | **0.8202** | 0.7932 | N/A |

[5] Most of the time parallel-GAIA has higher accuracy than single-GAIA, but single-GAIA may have slightly higher accuracy when single-GAIA alone can already produce high accuracy and thus parallel computing cannot improve it.

## 6. CONCLUSIONS

In this paper, we investigate the problem of efficiently finding discriminative subgraph patterns in graph databases for graph classification. We propose an efficient subgraph encoding approach that makes use of embedding information and supports arbitrary subgraph extensions. By using this encoding approach, we are able to adopt evolutionary computation in discriminative subgraph mining which explores candidate subgraph patterns efficiently in a randomized fashion. We also use parallel computation to further improve the quality of the resulting discriminative patterns by integrating the results from independent instances of pattern evolution. Experiments show that GAIA runs much faster and offers competitive or better classification accuracy than the state-of-the-art discriminative subgraph mining algorithms no matter whether with or without parallel computation, even when running on the datasets that the competitor algorithms are optimized for. In addition, GAIA shows linear scalability with respect to the size of graph database.

## 7. ACKNOLEDGMENTS

We thank Sayan Ranu for making implementation of graphSig available.

## 8. REFERENCES

1. D. Bandyopadhyay, J. Huan, J. Liu, J. Prins, J. Snoeyink, W.Wang, and A. Tropsha. Structure-based function inference using protein family-specific fingerprints, *Protein Science*, vol. 15, pp. 1537-1543, 2006.
2. C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, J. Han. Mining Graph Patterns Efficiently via Randomized Summaries, in *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, 2009.
3. H. Cheng, D. Lo, Y. Zhou, X. Wang and X. Yan, Identifying Bug Signatures Using Discriminative Graph Mining, Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA 09), Chicago, IL, July 2009.
4. K. A. De Jong, Evolutionary Computation: A Unified Approach. Cambridge, MA, USA: MIT Press, 2006, p 71-113, p 130-132.
5. M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent Sub-structure Based Approaches for Classifying Chemical Compounds. *IEEE Trans. Knowl. Data Eng. 17(8): 1036-1050*, *2005*.
6. M. A. Hasan and M. J. Zaki. Output Space Sampling for Graph Patterns, in *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, 2009.
7. C. Helma, T. Cramer, S. Kramer, and L.D. Raedt. Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds. *J. Chem. Inf. Comput. Sci*., 44:1402-1411, 2004.
8. J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism, *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pp. 549-552, 2003.
9. J. Huan, W. Wang, J. Prins, J. Yang. SPIN: Mining maximal frequent subgraphs from graph databases, in *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 581-586, 2004.
10. A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of 2000 European Symp. Principle of Data Mining and Knowledge Discovery*, pages 13-23, 2000.
11. N. Jin, C. Young and W. Wang, Graph Classification Based on Pattern Co-occurrence, *in Proceedings of the ACM 18th Conference on Information and Knowledge Management (CIKM)*, Hongkong, 2009.
12. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of ICDM*, pages 313-320, 2001.
13. S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases, in *Proceedings of the 25th International Conference on Data Engineering (ICDE),* April, 2009.
14. H. Saigo, N. Kraemer and K. Tsuda: Partial Least Squares Regression for Graph Mining, In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2008)*, 578-586, 2008.
15. M. Thoma, H. Cheng, A. Gretton, J. Han, H. Kriegel, A. Smola, L. Song, P. Yu, X. Yan, K. Borgwardt. "Near-optimal supervised feature selection among frequent subgraphs", In *SDM 2009*, Sparks, Nevada, USA.
16. J. R. Ullmann. An algorithm for Subgraph Isomorphism. *Journal of ACM, 23(1):31-42*, 1976.
17. X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 433–444, 2008.
18. X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, pages 721–724. IEEE Computer Society, 2002.