LTS: Discriminative Subgraph Mining by Learning from Search History

Ning Jin[#1], Wei Wang[#2]

[#]*Department of Computer Science, University of North Carolina at Chapel Hill*
Chapel Hill, NC USA

[1]`njin@cs.unc.edu`
[2]`weiwang@cs.unc.edu`

Abstract— **Discriminative subgraphs can be used to characterize complex graphs, construct graph classifiers and generate graph indices. The search space for discriminative subgraphs is usually prohibitively large. Most measurements of interestingness of discriminative subgraphs are neither monotonic nor anti-monotonic with respect to subgraph frequencies. Therefore, branch-and-bound algorithms are unable to mine discriminative subgraphs efficiently. We discover that search history of discriminative subgraph mining is very useful in computing empirical upper-bounds of discrimination scores of subgraphs. We propose a novel discriminative subgraph mining method, LTS (Learning To Search), which begins with a greedy algorithm that first samples the search space through subgraph probing and then explores the search space in a branch and bound fashion leveraging the search history of these samples. Extensive experiments have been performed to analyze the gain in performance by taking into account search history and to demonstrate that LTS can significantly improve performance compared with the state-of-the-art discriminative subgraph mining algorithms.**

I. Introduction

Complex structures in many scientific applications can be represented by graphs, and many data mining and database problems in graph databases, such as graph indexing [19], graph classification [3], [14] and graph clustering, need discriminative subgraph patterns. For example, biological structures can be stored as graphs, and in order to classify these structural graphs, discriminative subgraph patterns are demanded by graph classification algorithms to be used as features [1], [8], [9]; discriminative subgraphs in biological structures are also of wide utility in locating active sites and characterizing interesting structures for drug design [1]; program flow information can be represented as graphs and bug signatures can be identified by mining discriminative subgraphs in the graphs of failed programs and successful programs [2].

A. Related Work

SubdueCL [7] uses beam search to look for the most discriminative subgraph patterns successively until each positive graph can be covered by at least one subgraph pattern. However, its efficiency is limited since it calculates subgraph frequencies by computing subgraph-isomorphism many times. LEAP [17] is another algorithm for mining the most discriminative subgraph pattern. LEAP proposes two novel mining techniques, structural proximity pruning and frequency-descending mining. The structural proximity pruning takes into account the fact that subgraph patterns that are structurally alike tend to have similar discrimination power. This can be used to calculate a tight upper-bound of their discrimination scores. The frequency-descending mining takes advantage of the observation that subgraphs with higher frequency are more likely to be discriminative and thus may reach the optimal solution faster.

CORK [16] uses the number of correspondences to measure the discrimination power of a subgraph pattern and thereby achieves a theoretically near-optimal solution. Given a set of subgraph patterns, the number of correspondences for a set of subgraph patterns is the total number of pairs of graphs that these subgraphs cannot discriminate. Although the solution is near-optimal in terms of minimizing the number of correspondences, using the number of correspondences as the measurement of discrimination power may be problematic because subgraphs of significantly different discrimination power can have the same number of correspondences. GraphSig [13] tackles the problem of discriminative subgraph mining from a different perspective. One of the major challenges in discriminative subgraph mining is that discriminative subgraphs may have very low frequencies which results in a huge search space. Instead of looking for discriminative subgraph patterns in the whole dataset, graphSig clusters similar graphs into small groups and searches for discriminative subgraph patterns with high frequencies within each group. First, it uses random walk to generate feature vectors for all graphs and groups together graphs with similar feature vectors. Discriminative subgraph patterns with low frequencies among all graphs may therefore have high frequencies within these groups. Then, graphSig uses relatively high frequency thresholds to mine discriminative subgraph patterns from each group. COM [11] uses a heuristic subgraph exploration order to find discriminative patterns faster. It also takes into account co-occurrences of subgraph patterns to boost the discrimination power of features. Both COM and graphSig significantly outperform LEAP in speed. 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. GAIA [12] is the most recent development in efficient graph classification. It employs a novel subgraph pattern exploration order to enable the utility of evolutionary computation to search for discriminative subgraphs. GAIA has higher classification accuracy and faster speed than COM and graphSig for both chemical and protein graphs. However, GAIA needs parallel computing resource to have its best result.

For graph classification, an alternative solution to discriminative subgraph pattern mining is the graph kernel methods. [5] presents an optimal assignment kernel for chemical compound graphs. The idea is to compute an optimal assignment from nodes in one graph to those in another graph such that the overall matching score between two graphs is maximized. [15] defines a diffusion kernel for graph classification. They first identify frequent subgraphs from the graphs, map subgraphs to the graphs and then use a process called "pattern diffusion" to label nodes in the graphs. In the end, a graph alignment algorithm is used to compute the inner product of two graphs. [4] proposes an algorithm that incorporates the feature complexity in the learning process, using a kernel matrix weighted L_2 norm for regularization. It obtains improved regression performance over other machine learning methods that do not consider feature complexity.

B. Motivation and Our Contribution

We consider discriminative subgraph pattern mining as an optimization problem with a user-specified score function, whose search space includes all possible subgraphs. This search problem is typically solved in one of two ways: one is a greedy approach attempting to reach local optimal subgraph(s) as fast as possible; the other is a branch-and-bound approach that prunes the search space using an estimated upper-bound of the scores. We propose a new discriminative subgraph pattern mining algorithm, named LTS (Learn To Search), which integrates both approaches with novel probing and pruning techniques.

A tight estimated upper-bound of scores enables a branch-and-bound algorithm to prune branches that a loose estimated upper-bound is unable to and thereby leads to a smaller search space. COM [11] and GAIA [12] compute a very loose estimated upper-bound by assuming a constant positive frequency and zero negative frequency of "descendant" patterns in the subgraph enumeration tree. LEAP [17] proposes the prune-by-structural-proximity strategy, which is based on an observation that subgraph patterns with similar structures tend to have similar scores. It allows LEAP to calculate a tighter estimated upper-bound than COM and GAIA. We discover that a tight estimated upper-bound can also be achieved by learning from search history. We characterize a pattern by a sequence of scores of the "ancestor" patterns visited on the path to the pattern and we refer to this sequence as the score record of the pattern. We first sample the search space using a probing algorithm that locates a few discriminative subgraphs and acquires their score records. Then we explore the search space in a branch-and-bound fashion, using the score records to estimate an upper bound of scores along an exploration branch. When we compute the upper-bound for "descendants" of pattern p, we look for a pattern q probed during the first phase whose score record is the same as that of p and use the observed upper-bound of q as the estimated upper-bound for p. To facilitate this idea, the probing algorithm needs to be able to find subgraph patterns with high scores in a short amount of time. If the probing algorithm is unable to reach subgraph patterns with high discrimination scores, then the upper-bound based on the sample subgraphs from the probing algorithm tend to be underestimated and thus the optimal subgraph pattern may be missed. If the probing algorithm is very slow, then even if it finds subgraphs with high scores, it is still unable to improve the overall efficiency of the search. We propose an efficient probing algorithm that satisfies both requirements and also approaches the optimal score fast.

In a branch-and-bound algorithm that looks for the most discriminative subgraph(s), if the estimated upper-bound of scores along a search branch is not greater than the optimal score found so far, then the branch can be pruned without the risk of missing the optimal solution. Therefore, the higher the optimal score found so far, the more the pruned search space. Note that the optimal score found so far is always less than or equal to the true optimal score in the search space. So approaching the optimal score faster leads to more efficient pruning. LEAP achieves this goal by frequency-descending mining, which searches for discriminative subgraph patterns with high frequencies first to find a high score and then searches again with the help of the high score to prune. This search technique is based on an observation that subgraphs with high frequencies are more likely to have high scores. COM approaches the optimal score faster by exploring candidate subgraph patterns in the order of their changes in scores from their ancestors, which leverages the observation that patterns that have dramatic increase in scores are very likely to lead to patterns with even higher scores. GAIA utilizes evolutionary computation to approach the optimal score even faster, based on the assumption that patterns with higher scores are more likely to lead to the optimal pattern. In this paper, we propose a greedy probing algorithm, namely *fast-probe*, to efficiently locate sample subgraphs with high scores and compute their score records. These subgraphs and score records are used to achieve efficient pruning in the subsequent branch-and-bound search for discriminative subgraph patterns. *Fast-probe* only preserves and extends the best candidate subgraph patterns discovered so far. It uses multiple-lineage exploration instead of single-lineage exploration to compensate for its aggressiveness in pruning patterns and allow it to still find patterns with high discrimination scores. Our proposed algorithm LTS is a two-step method. First it invokes *fast-probe* to acquire search history and then performs a branch-and-bound search which utilizes the optimal scores from *fast-probe* as a starting point and estimates upper-bound based on search history. Experimental results show that *fast-probe* alone is a competitive discriminative subgraph mining algorithm compared with state-of-the-art competitors, and moreover, LTS as a whole can find even better subgraph patterns without prolonged runtime.

C. Organization

The remainder of this paper is organized as follows. In Section II we briefly review the preliminary knowledge for discriminative subgraph mining. Section III compares single-lineage exploration and multiple-lineage exploration and justifies why we adopt the latter. Section IV introduces the rationale of *fast-probe* and how it works. Section V explains how we model search history and how it is used to estimate upper-bounds. In Section VI, we demonstrate using extensive experiments the efficiency of the proposed algorithm and its advantage over other state-of-the-art competitors. Section VII concludes this paper.

II. Preliminaries

DEFINITION 1 (Graph). A graph is denoted as $g = (V, E)$, where V is a set of nodes and E is a set of edges connecting the nodes. Both nodes and edges can have labels.

Fig. 1 shows an example of two sets of graphs.

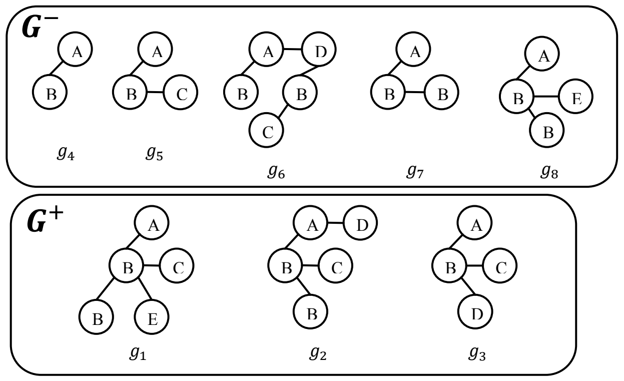


Figure 1: an example of two sets of graphs

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

For example, in Fig. 1, g_5 is a subgraph of g_1.

DEFINITION 3 (Frequency). Given a graph set G, the frequency of a subgraph pattern p is defined as:

$$freq(p|G) = \frac{|g \in G\ and\ g\ supports\ p|}{|G|}$$

The input of a discriminative subgraph pattern mining problem is usually composed of two sets of graphs: a positive set G^+ and a negative set G^-, as shown in Fig. 1. We denote $freq(p|G^+)$ by $pfreq(p)$ and $freq(p|G^-)$ by $nfreq(p)$.

For example, in Fig. 1, $pfreq(g_5)$ is 1 and $nfreq(g_5)$ is 0.2.

DEFINITION 4 (Discrimination Score). The discrimination score of a subgraph pattern p is the value of a user-specified objective function of its positive and negative frequencies. The more discriminative the pattern, the larger the discrimination score. In this paper, by default we define the discrimination score as:

$$s(p) = \log \frac{pfreq(p)}{nfreq(p)}$$

For example, in Fig. 1, $s(g_4) = \log \frac{1}{1} = 0$ and $s(g_5) = \log \frac{1}{0.2} = \log 5$.

We do not use the absolute value because it is very rare that the negative frequency is much higher than the positive frequency of p, as a result of highly diverse negative sets in most applications. Therefore, we only need to consider patterns with high positive frequencies and low negative frequencies.

To avoid the problem of denominator being zero in the objective function, when calculating the negative frequencies, we can replace zero with a small positive value ε. Thus, the negative frequency of any pattern p is never zero. For simple illustration, all examples in this paper do not have zero negative frequencies.

Given a positive set and a negative set of graphs, the goal of discriminative subgraph pattern mining is to find the subgraph pattern with the highest discrimination score for each positive graph. The output is a set of discriminative subgraphs. The resulting set may contain as few as only one pattern if all positive graphs share the same optimal pattern. We only consider discriminative subgraph patterns in the positive set because in most cases the negative set contains graphs that lack certain property or function and thus is too diverse to have highly discriminative subgraph patterns. If there is a need to find discriminative subgraph patterns in the negative set, users can simply switch the labels of the two input graph sets and the algorithm will treat the negative set as the positive set.

III. Pattern Exploration Order

Almost all efficient subgraph pattern exploration methods, such as gSpan [18] and FFSM [10], start with subgraphs having only one edge and extend them to larger subgraphs by adding one edge at a time. Each large subgraph pattern can be directly extended from more than one smaller subgraph patterns. For example, in Fig. 1, subgraph pattern *A-B-C* can be extended from either *A-B* or *B-C*.

DEFINITION 5 (Lineage). In a pattern exploration method M, a lineage of pattern p is a sequence of patterns: $l(p) = p_1 p_2 \dots p_{k-1} p_k$, where $p_k = p$, p_1 has only one edge and $\forall i \in [1, k-1]$, p_{i+1} can be directly extended from p_i by adding one more edge.

If a pattern exploration method allows a pattern to have multiple lineages, we call this exploration method a multiple-lineage exploration; otherwise, we call it a single-lineage exploration. Fig. 2 shows examples of multiple-lineage exploration and single-lineage exploration for the graph sets in Fig. 1. Each node represents a subgraph pattern (only patterns with less than 3 edges are shown for illustration) in the graph sets and there is a directed edge from node p to node q if in the exploration method pattern q is allowed to be reached by extending p.

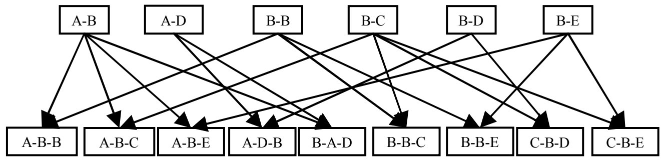
A subgraph pattern may have multiple possible lineages. Thus, multiple-lineage exploration is more natural than single-lineage exploration. To achieve single-lineage exploration, an algorithm needs to define an enumeration order $\prec$ on all subgraph patterns in the search space. If pattern $p \prec$ pattern q, then p is enumerated before q. The resulting lineages become the canonical lineages of the respective patterns. Both gSpan and FFSM are single-lineage exploration methods.

The major advantage of single-lineage exploration is that it is more efficient than multiple-lineage exploration in subgraph pattern enumeration without missing any pattern. In a single-lineage exploration method, each subgraph pattern is enumerated only once while a multiple-lineage exploration method may visit a pattern multiple times through different lineages. For example, in Fig. 2, the multiple-lineage exploration visits pattern A-B-B twice while the single-lineage exploration visits it only once. In addition, the average number of subgraph extensions performed for each subgraph pattern in single-lineage exploration is less than that in multiple-lineage exploration. For example, in Fig. 2, each subgraph pattern with one edge performs three extension operations on average[1] in multiple-lineage exploration while the average number of extension operations in single-lineage exploration is 1.5. Extension operation is the most costly operation in subgraph enumeration, thus algorithms requiring fewer extensions are highly favourable. In applications where subgraph patterns are much larger and more complex, the difference in number of extension operations becomes even larger. As a result, single-lineage exploration is preferred in most subgraph mining algorithms.

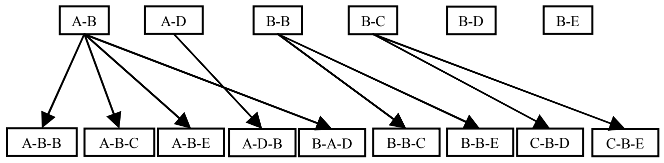
However, single-lineage exploration has the problem that its result is sensitive to subgraph pruning. Since each subgraph pattern can only be reached through a single lineage, we will miss a subgraph pattern if any subgraph on its lineage is pruned. On the contrary, multiple-lineage exploration is much more tolerant of subgraph pruning because a subgraph pattern can be reached through more than one lineage. This difference does not create any problem for using single-lineage exploration in frequent subgraph mining because of the antimonotonicity property of pattern frequency. In frequent subgraph mining, if pattern p is in the lineage of pattern q and q is a wanted (i.e. frequent) pattern, then p must also be wanted by the mining algorithm. However, in discriminative subgraph pattern mining, the redundancy in multiple-lineage exploration becomes its advantage over single-lineage exploration. Objective functions to measure discrimination power of subgraphs are usually not antimonotonic. If pattern p is in the lineage of pattern q and q is a wanted (i.e. discriminative) pattern, p is not necessarily wanted. Under such circumstances, multiple-lineage exploration can be aggressive in pruning patterns with low discrimination scores while single-lineage exploration cannot afford to prune any pattern unless it is absolutely certain that the pattern will not lead to any discriminative pattern.

For example, in Fig. 1, A-B-C is a highly discriminative subgraph pattern in the positive set while A-B is not discriminative as it appears in every positive and negative graph. The single-lineage exploration shown in Fig. 2 cannot prune A-B because otherwise A-B-C will be missed. The multiple-lineage exploration in Fig. 2 can afford to prune A-B since A-B-C can also be reached from B-C.

In our proposed algorithm, LTS, we adopt multiple-lineage exploration to reduce the risk of missing the most discriminative subgraph patterns due to pruning. We use CCAM code [12] to encode subgraph patterns and maintain a lookup table for subgraph patterns that have been extended to avoid extending a subgraph pattern repeatedly. Embeddings of subgraph patterns in the graph sets are also maintained to facilitate subgraph extension and frequency calculation.



Multiple-lineage exploration for the example in Fig. 1



Single-lineage exploration for the example in Fig. 1

Figure 2: an example of multiple-lineage exploration and single-lineage exploration for the two graph sets in Fig. 1

IV. Fast Probing Subgraph Pattern Space

As mentioned in Section I, a greedy algorithm can often reach a (relatively) discriminative subgraph quickly. Even though it may not be the optimal one, its score can be used to prune the search space. The higher the score, the better the pruning power. For example, let the estimated upper-bound for descendants of p be 1.0. By the time p is visited, if the best score so far is 1.2, all descendants of p can be pruned. But if the best score found so far is only 0.5, we are not able to perform any pruning.

We propose a greedy algorithm called *fast-probe* to generate a good sample of discriminative subgraphs to facilitate the subsequent branch-and-bound search. *Fast-probe* maintains a list of candidate subgraph patterns to be processed. The candidate list is initialized with all single-edge subgraph patterns in G^+. It repeatedly draws and processes a candidate pattern p from the list as long as the list is not empty. If pattern p is the optimal pattern for any positive graph at the time it is processed, *fast-probe* computes all extensions of p in the positive set with one more edge and put an extension into the candidate list if the extension has not be generated before; otherwise, p is discarded. *Fast-probe* terminates when the candidate list becomes empty. This process is efficient since only the best subgraphs are extended to generate candidate patterns. The algorithm is described in Fig. 3.

[1] This is computed by taking the ratio of the number of arrows to the number of single-edge subgraphs.

```
Algorithm: fast-probe ($G^+$, $G^-$)
$G^+$: positive graph set
$G^-$: negative graph set
Candidate_list: the set of subgraph patterns to be extended
1.  Put all single-edge subgraph patterns into candidate_list
2.  while (candidate_list is not empty)
3.      $p$ ← get next pattern and remove it from candidate_list
4.      updated ← false
5.      for each graph $g$ in $G^+$
6.          if $s(p)$ > optimal score for $g$ so far
7.              update the optimal pattern and optimal score for $g$
8.              updated ← true
9.      if (not updated)
10.         continue
11.     $C$ ← all subgraph patterns with one more edge attached to $p$
12.     for each pattern $q$ in $C$
13.         if $q$ has not been generated before
14.             put $q$ into candidate_list
15. return the optimal pattern for each $g$ in $G^+$
```

Figure 3: algorithm for *fast-probe*

We define an indicator function for a subgraph pattern p as follows:

$$d(p) = \begin{cases} 1, \exists g \in G^+,\ \ s(p) > optimal\ score\ for\ g\ so\ far \\ 0, otherwise \end{cases}$$

If function d is antimonotonic as patterns are extended, then when a pattern p is visited and it fails to be the optimal pattern for any positive graph (i.e. $d(p)$ = 0), we can safely prune p and any lineages extended from p. No supergraph of p will be the optimal pattern for any positive graph because of the antimonotonicity property that once $d(p)$ = 0 no supergraph q of p will have $d(q)$ = 1. If this assumption is true, then the search process would become very efficient as only good patterns need to be considered. And single lineage exploration would have been sufficient.

However, this assumption is not always true because discrimination scores of patterns may increase as patterns become larger. Therefore, even if a pattern p is not the optimal pattern for any positive graph, a supergraph of p may be the optimal pattern for some positive graph because its score is greater than the score of p.

Nevertheless, the assumption does not have to hold for all subgraph patterns to make *fast-probe* work. In fact, for the most discriminative subgraph pattern, as long as the assumption holds for at least one of its lineages, the optimal pattern will be found. Using multiple-lineage exploration helps because the likelihood of the assumption being true for at least one lineage is much larger in multiple-lineage exploration than in single-lineage exploration. In addition, the most discriminative subgraph pattern will not be missed as long as patterns in its lineages are optimal patterns for one positive graph at the time they are visited. This is very likely to be true: it is typical that some positive graphs are covered by multiple highly discriminative subgraphs while others do not have highly discriminative subgraphs. We call the former as "rich" graphs and the latter as "poor" graphs. Ancestors for the highly discriminative subgraphs for "rich" graphs may cover "poor" graphs when their positive frequencies are still high. Let p be the most discriminative subgraph for a "rich" graph g and q be another highly discriminative subgraph for g. Let q be visited before any ancestor of p is visited. Patterns in the lineages of p may not be the optimal patterns for g when they are visited because they may not be as discriminative as q. However they may be the optimal patterns for some "poor" graphs and thus survive and produce a lineage to p. The most discriminative subgraphs for "poor" graphs may be missed when there are no "poorer" graphs for their ancestors to survive. In this case, a subsequent (branch and bound) search may be needed to recover the most discriminative subgraphs missed by *fast-probe*.

V. Upper-bound Estimation by Learning from Search History

A tight estimated upper-bound of scores may improve the efficiency of branch-and-bound algorithms. For example, when p is visited, let the optimal score of any patterns visited so far be 1.2. If the estimated upper-bound is 1.5 (loose), then the algorithm cannot prune any descendants of p; but if the estimated upper-bound is 1.1 (tight), then the algorithm can prune all descendants of p.

We first study a simple way for upper-bound estimation. According to the definition, the discrimination score increases as the negative frequency decreases, and decreases as the positive frequency decreases. A simple estimation of upper-bound for scores of descendants of p is achieved when the positive frequency remains the same as that of p and the negative frequency is zero:

$\hat{B}(p) = \log\frac{pfreq(p)}{\varepsilon}$, where ε is a small value to replace 0.

This is a very loose upper-bound especially when the positive and negative frequencies of p are high. In most cases, adding edges to p causes both positive and negative frequencies to decrease. If the negative frequency decreases faster than the positive frequency, then the pattern becomes more and more discriminative; otherwise, the pattern becomes less discriminative. If the negative frequency of p is high, many edges need to be added to it to achieve zero negative frequency and as a result the positive frequency drops significantly as well. For example, in chemical compound graphs, *C-C* has positive and negative frequencies almost equal to 100% as it is prevalent in chemical compounds. However, its most discriminative descendants typically have positive frequency less than 15% and negative frequency close to zero. Therefore, the optimal discrimination score is much lower than the estimated upper-bound, which results in inefficient pruning.

Previous discriminative subgraph mining algorithms takes advantage of the correlations between score (or frequency) of a pattern and the largest score that the descendants of the pattern may have in designing exploration orders, in order to approach the optimal score as fast as possible. However, such correlations are qualitative and can only serve as a heuristic guidance. We propose to learn quantitative correlations from search history and use them to estimate tight upper-bounds.

DEFINITION 6 (score record). Given a lineage of pattern p, $l(p) = p_1p_2 \dots p_{k-1}p_k$, the score record for $l(p)$ is a sequence of scores for the patterns in the lineage:

$$h(p) = s(p_1), s(p_2), \dots, s(p_{k-1}), s(p_k)$$

A discriminative subgraph mining process always generates many score records, which can be organized into a prefix tree, called *prediction tree*. Fig. 4 shows an example of score records and the corresponding *prediction tree*. Each tree node is labelled with score and the root node is labelled with 0.0, which is the score of an empty subgraph. In our implementation, we discretize scores evenly into 10 bins and use the discretized scores as labels. In the example, we use the original scores as labels for the sake of intuitive illustration. In addition to the score label, each tree node is also associated with the maximum score in the sub-tree rooted at this node. The score records and the corresponding *prediction tree* can be considered as a sample of the whole search space. Therefore, the maximum score at each tree node is an estimated upper-bound in the search space. For example, for a pattern p with score record (0.5, 0.7, 1.0), its maximum score in the *prediction tree* is 1.5 and thus its estimated upper-bound in the search space is 1.5. We organize the sample space by scores (rather than by subgraph structures in the search space) because it is much easier to compare scores than structures. Sometimes the score record of a pattern p is absent in the tree, so we additionally generate a lookup table, named *prediction table*, to aggregate the information in the tree. The key for each entry in the *prediction table* is composed of the number of edges in the pattern and the score of the pattern. The value stored at each entry is the maximum score of the descendants of the patterns with the corresponding size and score in the sample space. For example, if the score record of p is (0.4, 0.8), which cannot be found in the *prediction tree*, then we use the key <2, 0.8> to look for an upper-bound estimation in the *prediction table*, which returns 1.0. The search history H is composed of the *prediction tree* and the *prediction table*. If neither the score record nor the <size, score> pair of p can be found in H, then we use the loose upper-bound estimation discussed earlier in this section. Fig. 6 summarizes the algorithm for upper-bound estimation based on history H.

Using search history to estimate upper-bound bears the risk of underestimating upper-bound if the discriminative subgraph mining process, which provides the score records, fails to capture a good sample of high discrimination scores. This will result in inefficient pruning and thus prolonged execution time. However, there is little impact to the mining process if the greedy sampling misses many low discrimination scores because, although these score records may be absent in the *prediction tree*, the *prediction table* can still provide a reasonably tight upper-bound estimation and we always have the last resort to the loose estimation.

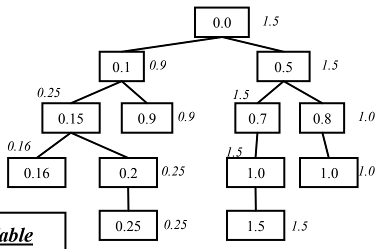
LTS first uses *fast-probe* to collect score records and generates search history H, which includes a *prediction tree* of score records and a *prediction table* aggregating the score records. LTS utilizes a vector F to keep track of the optimal pattern for each positive graph: $F[i]$ stores the optimal pattern for positive graph g_i. Vector F is updated with the optimal patterns found by *fast-probe*, which compose a better starting point than single-edge subgraphs, before the following branch-and-bound search. Then LTS performs a branch-and-bound search in the subgraph search space and uses a candidate list to keep track of candidate subgraph patterns. Its goal is to find the most discriminative subgraph for each positive graph. When the branch-and-bound search begins, the candidate list is initialized with all subgraphs with one edge. LTS repeatedly pops one subgraph from the candidate list at a time until the candidate list becomes empty. LTS uses CCAM code [12] to encode subgraphs and maintains a lookup table to keep track of processed subgraphs. For each subgraph p from the candidate list, LTS updates $F[i]$ if positive graph g_i supports p and $s(p)$ is greater than $s(F[i])$. Meanwhile, LTS estimates the upper-bound of p based on search history H and checks whether the upper-bound is greater than any $s(F[i])$ with g_i supporting p. If the upper-bound is not greater than the optimal score of any positive graph supporting p, then p is discarded from further extension. Note that for each pattern, we only consider the positive graphs supporting this pattern when updating optimal scores and pruning with the estimated upper-bound because we are looking for the optimal pattern for each positive graph. If p is preserved, LTS computes all of its extensions with one more edge in the positive set. The extensions that have not been visited before[2] are put into the candidate list.

Fig. 7 summarizes the algorithm for LTS.

Search Records
record 1: 0.5, 0.7, 1.0, 1.5
record 2: 0.5, 0.8, 1.0
record 3: 0.1, 0.15, 0.2, 0.25
record 4: 0.1, 0.9
record 5: 0.1, 0.15, 0.16

Corresponding *Prediction Table*

<# of edges, score>	bound
<2, 0.15>	0.25
<2, 0.7>	1.5
<2, 0.8>	1.0
<3, 0.16>	0.16
<3, 0.2>	0.25
<3, 1.0>	1.5
<4, 0.25>	0.25
<4, 1.5>	1.5



corresponding *prediction tree*

Figure 4: an example of search records and the corresponding *prediction tree* and *prediction table*

```
Algorithm: prediction_tree_search(h(p), par, cur)
p: a subgraph pattern
h(p): the score record for a lineage of p
par: a tree node in the prediction tree of search history
cur: an integer number indicating the current level in the prefix tree
1.  n ← the child node of par whose score equals $s(p_{cur})$ in h(p)
2.  if (n is empty)
3.      return empty
4.  if (cur = $|E(p)|$)
5.      return n
6.  else
7.      return prediction_tree_search(h(p), n, cur+1)
```

Figure 5: the algorithm for finding the tree node in the *prediction tree* of search history according to a score record

[2] That is, they are not present in the lookup table for subgraphs.

```
Algorithm: $U(p, H)$
$p$: a subgraph pattern
H: search history, including a prediction tree and a prediction table
1.  h(p) ← $s(p_1), s(p_2), \dots, s(p_{k-1}), s(p_k)$, where $l(p) = p_1 p_2 \dots p_{k-1} p_k$ is the current lineage through which $p$ is visited
2.  n ← prediction_tree_search(h(p), H.prefix_tree.root, 1)
3.  if (n is not empty)
4.      return n.empirical_upper_bound
5.  else if ($H.lookup\_table[|E(p)|, s(p)]$ exists)
6.      return $H.lookup\_table[|E(p)|, s(p)]$
7.  else
8.      return $\log \frac{pfreq(p)}{\varepsilon}$
```

Figure 6: algorithm for upper-bound estimation based on search history

```
Algorithm: LTS ($G^+$, $G^-$)
$G^+$: positive graph set
$G^-$: negative graph set
F: a vector maintaining the optimal pattern for each positive graph
Candidate_list: the set of subgraph patterns to be extended
H: search history of the patterns visited by fast-probe
1.  F ← fast-probe($G^+$, $G^-$) and store search history of the patterns visited by fast-probe in H
2.  Put all single-edge subgraph pattern into candidate_list
3.  while (candidate_list is not empty)
4.      $p$ ← get next pattern and remove it from candidate_list
5.      is_promising ← false
6.      for each graph $g$ in $G^+$ that supports $p$
7.          if $U(p, H)$ > s(F[i])
8.              is_promising ← true
9.          if $s(p)$ > s(F[i])
10.             F[i] ← p
11.     if (not is_promising)
12.         continue
13.     C ← all subgraph patterns with one more edge attached to $p$
14.     for each pattern $q$ in $C$
15.         if $q$ has not been generated before
16.             put $q$ into candidate_list
17. return the optimal pattern for each $g$ in $G^+$
```

Figure 7: algorithm for LTS

VI. 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 10.04 64-bit version. In the experiments, we use discriminative subgraphs found by *fast-probe* or LTS to perform graph classification and measure the performance of *fast-probe* or LTS by its runtime and classification accuracy. To build graph classifiers based on the discriminative subgraph patterns found by LTS, we generate a classification rule for each pattern p from LTS in the form of "if a graph g supports $p \rightarrow g$ is positive". Then we select classification rules to compose graph classifiers, optimizing normalized accuracy in the training sets. This classifier generation process is the same as that of GAIA [12]. The rationale for using classification accuracy to measure performance is that the simple classifiers described above demand highly discriminative subgraphs to achieve high accuracy and thus the classification accuracy directly reflects the discrimination power of the subgraphs. In the end of this section, we also compare the best score found by LTS and the best score found by LEAP [17].

All experiments involving classification are performed with 5-fold cross-validation unless otherwise specified. We measure the classification accuracy by normalized accuracy, which is defined as follows.

$$Sensitivity = \frac{\#\ of\ positives\ that\ are\ classified\ as\ positive}{\#\ of\ positives}$$

$$Specificity = \frac{\#\ of\ negatives\ that\ are\ classified\ as\ negative}{\#\ of\ negatives}$$

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

In this section, we are going to demonstrate:

1) Power of multiple-lineage exploration: *fast-probe* using multiple-lineage exploration achieves much higher classification accuracy than using single-lineage exploration with comparable speed.
2) High performance of *fast-probe*: *fast-probe* alone is sufficient to achieve competitive classification accuracy for some datasets and outruns competitors.
3) Effectiveness of using search history: when *fast-probe* produces inferior classification results, LTS can significantly improve classification accuracy and outperform competitors.
4) Efficiency in optimal pattern mining: LTS excels LEAP in mining the optimal subgraph pattern in terms of runtime with comparable or even better optimal scores. It demonstrates the efficiency of our upper-bound estimation based on search history since it represents the major difference between LTS and LEAP.

A. Description of the Datasets

We use protein datasets and chemical compound datasets in our experiments. The protein datasets consist of protein structures from Protein Data Bank [3] classified by SCOP [4] (Structural Classification of Proteins). As for protein datasets, we select all large SCOP families with more than 25 members (listed in Table I). In each dataset, protein structures in a selected family are taken as the positive set. Unless otherwise specified, we randomly select 256 outsider proteins (i.e., not members of the 16 families) as a common negative set used by all 16 protein datasets. The same datasets were used in [11] and [12]. 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 alpha carbons of two amino acids is less than 11.5 angstroms. Nodes are labeled with their amino acid type and edges are labeled with the discretized distance between the alpha carbons. On average, each protein graph has approximately 250 nodes and 2700 edges. The chemical compound datasets consist of chemical compound structures from PubChem [5] classified by their biological activities, listed in Table II. Each compound can be either active or inactive in a bioassay. The same datasets are used in [11]-[13], [17]. For each bioassay, we randomly select 400 active compounds as the positive set and 400 inactive

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

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

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

compounds as the negative set. We generate balanced chemical compound datasets in order to compare with graphSig [13] whose implementation can only process balanced datasets. In chemical compound graphs, each atom is represented by a graph node labeled with the atom type and each chemical bond is represented by a graph edge labeled with the bond type. On average, each compound graph has 54.76 nodes and 57.24 edges.

TABLE I.
LIST OF SELECTED SCOP FAMILIES

SCOP ID	Family name	# of selected proteins
46463	Globins	51
47617	Glutathione S-transferase (GST)	36
48623	Vertebrate phospholipase A2	29
48942	C1 set domains (antibody constant domain like)	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 II.
LIST OF SELECTED BIOASSAYS

Bioassay ID	Tumor description	# of actives	# 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

B. *Power of Multiple-lineage exploration*

First, we study the importance of using multiple-lineage exploration rather than single-lineage exploration as discussed in Section III. We implement two versions of *fast-probe*: one with multiple-lineage exploration and the other with single-lineage exploration, respectively. We refer to the former as *ME-fast-probe* and the latter as *SE-fast-probe*. Fig. 8 shows the normalized accuracy comparison between the two versions of *fast-probe* for chemical compound datasets. *SE-fast-probe* is obviously incompetent in finding discriminative subgraphs for its accuracy is around 50%, while *ME-fast-probe* has much better performance with an average accuracy around 70%. The low accuracy of *SE-fast-probe* is due to its intolerance of missing patterns and the aggressive pruning strategy of *fast-probe*. *ME-fast-probe* benefits from its redundancy in exploration and is able to find discriminative subgraph patterns in spite of the extremely aggressive pruning strategy. Fig. 9 compares the runtime of the two versions of *fast-probe*. Although *ME-fast-probe* is slower than *SE-fast-probe*, the difference is acceptable. The redundancy in exploration of *ME-fast-probe* does not lead to very poor runtime performance because many subgraphs are removed from extension by the pruning strategy. Therefore, the redundant exploration method and the highly aggressive pruning strategy complement each other well.

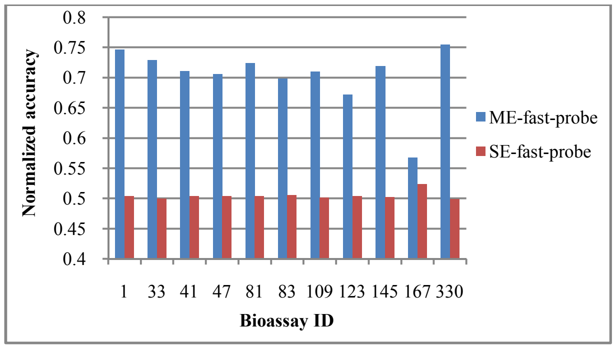


Figure 8: normalized accuracy comparison between multiple-lineage-exploration-based fast-probe and single-lineage-exploration based fast-probe using chemical datasets

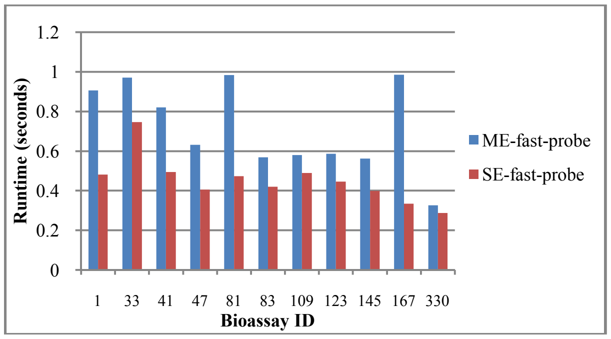


Figure 9: runtime comparison between multiple-lineage-exploration-based fast-probe and single-lineage-exploration based fast probe using chemical datasets

C. *Efficiency of fast-probe*

We also compare *fast-probe* with three state-of-the-art discriminative subgraph mining algorithms, GAIA [12], COM [11] and graphSig [13], using the chemical compound datasets. The parameter settings (listed in Table III) for GAIA, COM and graphSig are the same as those used in [11], [12] and [13] for chemical datasets since we are using the same datasets. The number of CPUs used by GAIA is set to be 1 in order to optimize its normalized accuracy. Using more than one CPU for chemical datasets does not lead to higher accuracy for GAIA. There is no parameter for *fast-probe*. We do not compare with frequent subgraph mining algorithms because it is computationally intractable to enumerate all the subgraph patterns and then find discriminative subgraphs among them.

Fig. 10 compares the normalized accuracy between the four algorithms. On average, the normalized accuracy of *fast-probe* is 6.07% higher than that of COM and 2.67% higher than that of graphSig. We perform paired t-test to evaluate the statistical significance of such difference in normalized accuracy. The lower the p-value is, the more statistically significant the difference is. The p-value for the improvement over graphSig is 0.003 and the p-value for the improvement over COM is $1.72 * 10^{-8}$. GAIA and *fast-probe* have similar normalized accuracy (GAIA's accuracy slightly lower by 0.06%). The difference is not statistically significant, but *fast-probe* runs faster than GAIA. Fig. 11 shows the runtime comparison between the four algorithms. On average, *fast-probe* is 4.76 times faster than COM and 34.35 times faster than graphSig. *Fast-probe* outruns GAIA in 9 out of 11 chemical compound datasets and its average speed is 1.8 times faster. The differences in runtimes are statistically significant at the 0.005 level.

This experiment shows that *fast-probe* alone is a highly competitive discriminative subgraph mining method for chemical compound datasets.

TABLE III.
PARAMETER SETTINGS FOR GAIA, COM AND GRAPHSIG FOR CHEMICAL DATASETS

algorithm	**Parameters for chemical datasets**
GAIA	Candidate list size = 10, maximal # of iterations = 4, # of CPUs used = 1
COM	Positive frequency threshold = 1%, Negative frequency threshold = 0.4% Maximal # of edges in a subgraph = 5
graphSig	maxPvalue=0.1, minFreq=0.1%

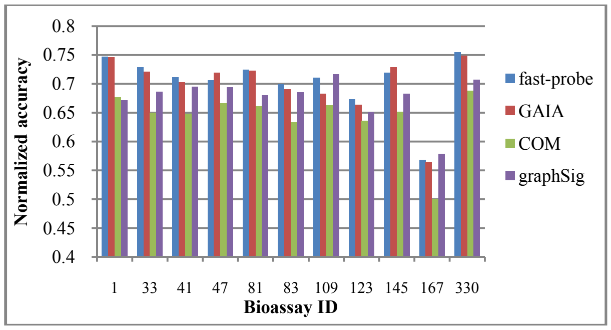


Figure 10: normalized accuracy comparison between *fast-probe*, GAIA, COM and graphSig using chemical datasets

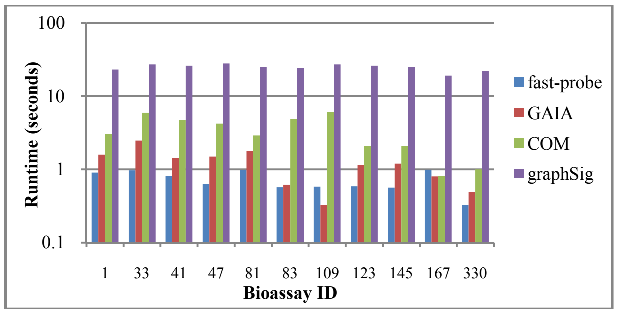


Figure 11: runtime comparison between *fast-probe*, GAIA, COM and graphSig using chemical datasets

D. Effectiveness of Using Search History

We compare the classification accuracy of *fast-probe* and LTS. *Fast-probe* is part of LTS and is invoked first to generate search history. Fig. 12 shows the normalized accuracy comparison between *fast-probe* and LTS for chemical datasets. It can be seen that LTS and *fast-probe* produce almost same classification accuracy in chemical compound datasets. LTS cannot further improve the classification accuracy of *fast-probe* in chemical datasets because *fast-probe* is already very efficient for these datasets and has comparable, if not better, performance compared with state-of-the-art algorithms. When the highly discriminative subgraph patterns are already found, further search in the pattern space cannot lead to better classification accuracy unless more sophisticated classifier generation methods are used. LTS has slightly worse result than *fast-probe* for some chemical datasets because higher discrimination scores[6] in the training set do not necessarily guarantee higher classification accuracy in the test set when the difference in discrimination scores is marginal. Discrimination score is calculated based on the training set while the classification is performed on the test set. If the score of p is only slightly greater than the score of q in the training set, it is possible that the score of p is slightly less than that of q in the test set. Selecting the pattern with higher score in the training set may sometimes produce slightly lower accuracy in the test set.

Fig. 13 compares the normalized accuracy of LTS and *fast-probe* using protein datasets. LTS outperforms *fast-probe* in terms of accuracy in 14 out of 16 protein datasets. In 5 protein datasets LTS improves accuracy by more than 10%. On average, LTS improve normalized accuracy by 8.06% compared with *fast-probe* and the p-value for this improvement is 0.003.

The results of the two comparisons in chemical datasets and protein datasets are consistent with the fact that protein graphs are much more complex than chemical graphs for protein graphs have more nodes, higher edge degrees and more diverse labels (chemical graphs have more labels, but the majority of nodes are labelled as Carbon, Oxygen or Nitrogen and most edge are labelled as single-bond). It is harder to mine the most discriminative subgraph patterns in protein graphs because of the much larger search space. As a result, *fast-probe* is relatively incompetent and LTS is able to improve accuracy significantly for protein graphs. This is contrary to the case of chemical graphs where *fast-probe* is highly efficient and LTS does not improve its accuracy.

For protein graphs, we compare LTS with GAIA and COM (graphSig is optimized for chemical graphs, so it achieves only ~50% normalized accuracy for protein graphs and takes significantly longer time than COM and GAIA). The parameter settings (listed in Table IV) are the same as those used in [11] and [12] for protein graphs, respectively. The number of CPUs is set to be 32 for GAIA to optimize the accuracy. When the number of CPUs used by GAIA is less

[6] We will show in next subsection that patterns found by LTS have higher scores than *fast-probe*.

than 8, the classification accuracy of GAIA is lower than that of COM. Note that LTS is parameter free.

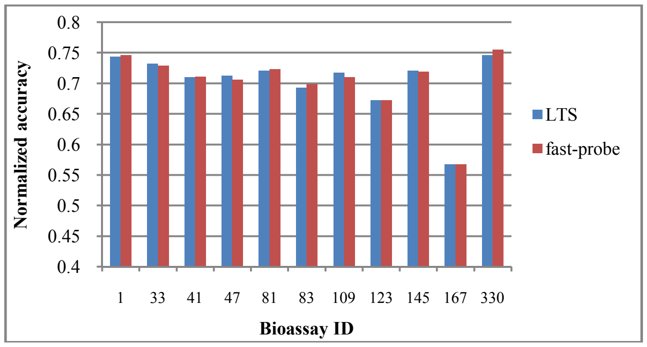


Figure 12: normalized accuracy comparison between *fast-probe* alone and LTS using chemical datasets

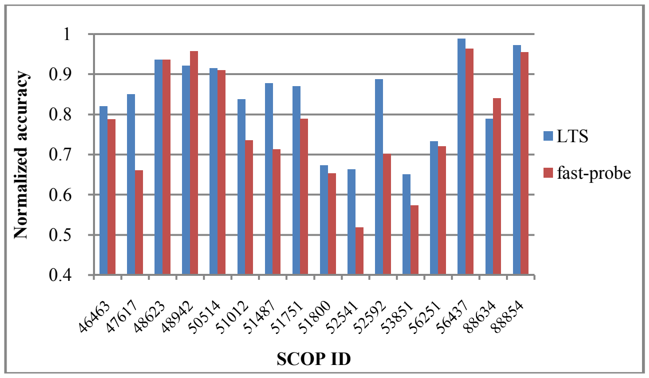


Figure 13: normalized accuracy comparison between *fast-probe* alone and LTS using protein datasets

TABLE IV.
PARAMETER SETTINGS FOR GAIA AND COM FOR PROTEIN DATASETS

algorithm	Parameters for chemical datasets
GAIA	Candidate list size = 100, maximal # of iterations = 10, # of CPUs used = 32
COM	Positive frequency threshold = 30%, Negative frequency threshold = 0.0%

Fig. 14 shows the normalized accuracy comparison between LTS, GAIA and COM using protein datasets. LTS outperforms GAIA in 11 out of 16 protein datasets and excels COM in 12 out of 16 datasets in terms of normalized accuracy. The average normalized accuracy of LTS is 1.63% higher than that of GAIA and 4.32% higher than that of COM. The p-value for LTS's improvement over GAIA is 0.033 and the p-value for LTS's improvement over COM is 0.0006.

Fig. 15 compares the runtime of LTS, GAIA and COM. In order to reflect the search efficiency, we multiply the runtime of GAIA by 32 for it uses 32 processes searching in parallel. Even though GAIA returns the result in a short amount of time *t*, its actual search time is much longer than what *t* indicates. It can be seen that after taking the total CPU cycles consumed as runtime, GAIA becomes an order of magnitude slower than LTS and COM. In most protein datasets, LTS has comparable speed with COM. On average, LTS is 11.3 times faster than GAIA when the actual computation time is considered and 1.88 times slower than COM with significantly higher accuracy. In spite of this, the runtime efficiency of LTS is still remarkable. We also implement a naive branch-and-bound search using the simplest upper-bound estimation (*fast-probe* is used to provide starting optimal scores). It runs for more than 10 minutes before it runs out of memory in the end on a protein dataset. Besides, COM requires user input in setting the suitable positive and negative frequency thresholds which determines the efficiency of the search space pruning. LTS does not require such information from users.

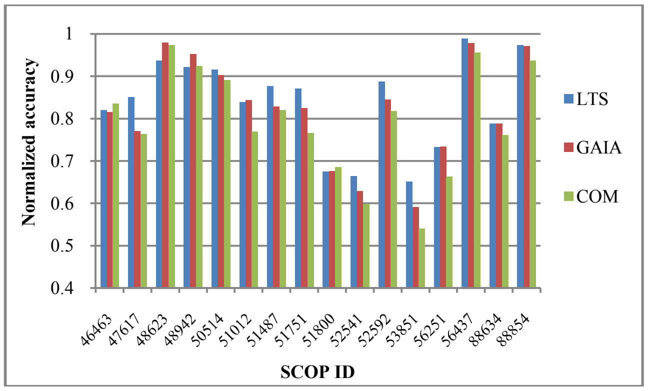


Figure 14: normalized accuracy comparison between LTS, GAIA and COM using protein datasets

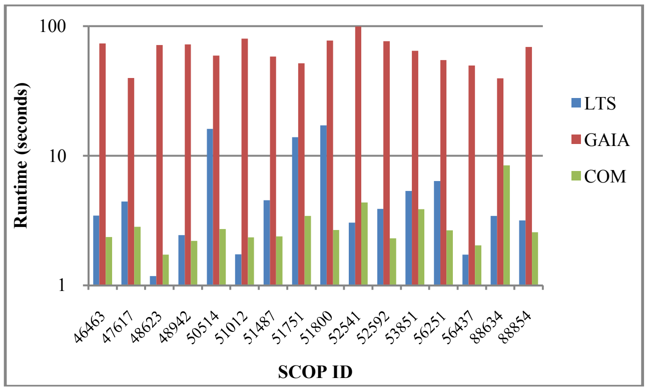


Figure 15: runtime comparison between LTS, GAIA and COM using protein datasets

E. *Quality of Individual Patterns*

Although LTS aims to find a set of discriminative subgraph patterns rather than the most discriminative subgraph pattern as LEAP does, the high classification accuracy of LTS is not a mere result of complementary weak features. In fact, the best subgraph pattern found by LTS is as good as, if not better than, the best pattern found by LEAP. We run both LTS and LEAP to look for the subgraph pattern with the highest g-test score for each dataset (We use g-test in this experiment because it is used in the original implementation of LEAP). The g-test score of a pattern p is defined as:

$$pfreq(p) * \log \frac{pfreq(p)}{nfreq(p)} + (1 - pfreq(p)) * \log \frac{1 - pfreq(p)}{1 - nfreq(p)}$$

LEAP has a parameter σ to control the leap length. The larger the value of σ, the faster the search. But a large σ often results in low optimal score found by LEAP. We set the leap length σ to be 0.05 which is the lowest leap length LEAP can handle without running out of memory. We also set the terminating positive frequency threshold equal to 20% for LEAP's frequency-descending mining[7] for protein datasets because all discriminative subgraph patterns found by LTS, GAIA and COM in protein datasets have positive frequency greater than 20%. For chemical compound datasets, we set the terminating positive frequency threshold to be 5% for LEAP's frequency-descending mining. LTS and *fast-probe* are parameter free.

Fig. 16 compares the optimal scores found by LTS and LEAP in protein datasets. Out of the 16 protein datasets, LTS has higher optimal score than LEAP in 10 datasets and has same optimal score with LEAP in 3 datasets. LEAP has better optimal score than LTS in only 3 out of 16 protein datasets. On average, the optimal score found by LTS is higher than that of LEAP by 0.4 and the p-value for this improvement is 0.03.

Fig. 17 shows the runtime comparison[8] between LTS and LEAP using protein datasets. LTS always outruns LEAP and the average speed of LTS is 50 times faster than that of LEAP. The p-value for this improvement in runtime is 0.018.

Fig. 18 compares the optimal scores found by LTS, LEAP and *fast-probe* in chemical datasets. Although *fast-probe* has the same classification accuracy as LTS (shown in Fig. 12), the optimal patterns found by *fast-probe* are not as good as those found by LTS. On average, the optimal scores found by LTS are higher than those from *fast-probe* by 0.076. The p-value for this difference is 0.0077. When comparing LTS and LEAP using chemical datasets, we can see that they have very similar performance in terms of optimal scores. Out of 11 chemical datasets, LTS has higher optimal scores than LEAP in 6 datasets, lower optimal sores in 4 datasets and same score in 1 dataset. The average optimal score of LTS is slightly higher than that of LEAP by 0.023. The p-value for this difference is 0.46, which is statistically insignificant.

Fig. 19 shows the runtime comparison between LTS and LEAP using chemical datasets. LTS is always faster than LEAP. The average speed of LTS is 6.5 times faster than that of LEAP. The p-value for this runtime improvement is 0.0007.

Since both LTS and LEAP use the branch-and-bound search framework and the major difference lies in their upper-bound estimation approaches, the advantage of LTS over LEAP can be attributed to the upper-bound estimation based on search history. The optimal scores from *fast-probe* do not play a main role in the high efficiency of LTS in protein datasets because the basic branch-and-bound search using optimal scores from *fast-probe* is still inefficient as mentioned in the previous subsection.

[7] It begins with a positive frequency threshold equal to 80% and terminates when the positive frequency threshold becomes lower than 20%.

[8] We do not run 5-fold cross-validation in this experiment, so the runtime in Fig. 17 is different from that in Fig. 15.

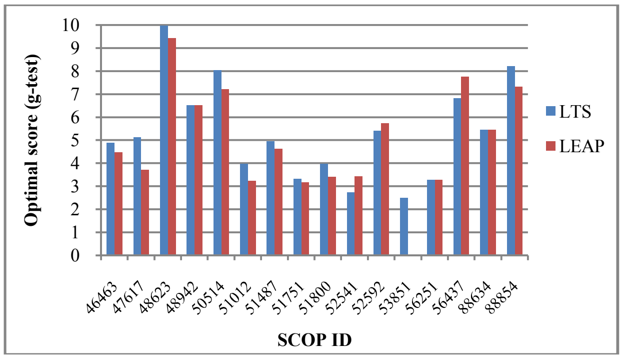


Figure 16: optimal score comparison between LTS and LEAP using protein datasets

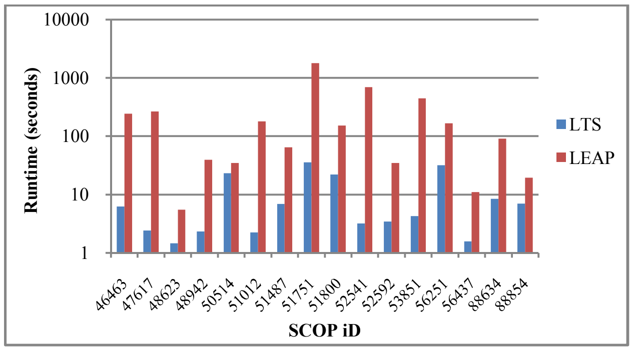


Figure 17: runtime comparison between LTS and LEAP using protein datasets

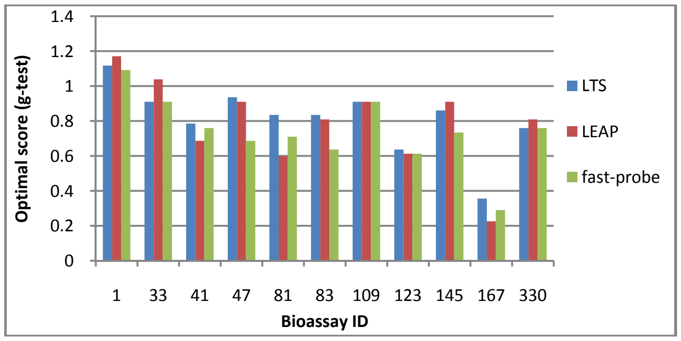


Figure 18: optimal score comparison between LTS, LEAP and *fast-probe* using chemical compound datasets

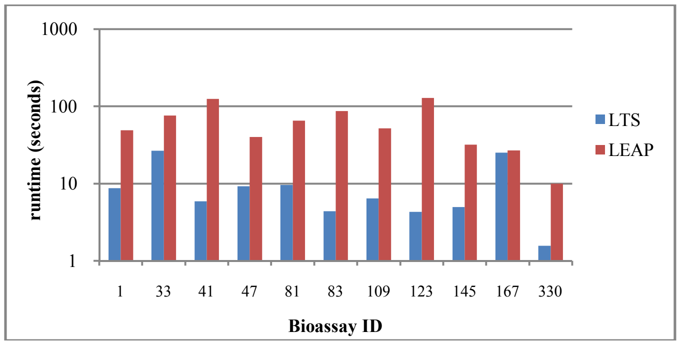


Figure 19: runtime comparison between LTS and LEAP using chemical compound datasets

VII. CONCLUSIONS

In this paper, we investigate the feasibility of estimating upper-bound for discrimination scores of subgraph patterns in discriminative subgraph mining by learning from search history. We devise a fast discriminative subgraph mining algorithm, namely *fast-probe*, to efficiently collect score records of subgraphs with high discrimination scores by using multiple-lineage exploration and an aggressive pruning strategy that only considers the optimal patterns. Then we organize the score records into a *prediction tree* and a *prediction table* to summarize upper-bound information in the sample search space visited by *fast-probe*. A subsequent branch-and-bound search is performed on the whole search space with upper-bound estimation based on the *prediction tree* and the *prediction table*. The overall algorithm LTS, which includes *fast-probe* and the consequent branch-and-bound search, searches for the most discriminative subgraph pattern for each positive graph. We first evaluate the performance of *fast-probe* and LTS by their runtime and the classification accuracy of graph classifiers built on the patterns found by them. Experiments demonstrate that *fast-probe* alone is already highly competitive in chemical datasets compared with state-of-the-art discriminative subgraph mining algorithms and thus LTS does not further improve classification accuracy. In the more complex protein datasets, LTS can significantly improve classification accuracy by the branch-and-bound search following *fast-probe*. LTS also has statistically significant improvement in normalized accuracy over state-of-the-art methods in protein datasets with comparable, if not higher, runtime efficiency. When used to mine the optimal subgraph pattern for the whole positive set, LTS outperforms LEAP dramatically in terms of runtime with comparable, if not better, optimal scores, which also demonstrates the value of learning from search history since both LTS and LEAP adopt the branch-and-bound framework but differ in that LEAP uses structural-proximity for upper-bound estimation and LTS uses search history for upper-bound estimation.

ACKNOWLEDGMENT

We thank Xifeng Yan and Sayan Ranu for their kindly providing implementation of LEAP and graphSig.

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] 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.

[3] H. Fei and J. Huan, Structure feature selection for graph classification, in *CIKM*, pages 991-1000, 2008.

[4] H. Fei and J. Huan, L2 Norm Regularized Feature Kernel Regression for Graph Data, in *CIKM*, pages 593-600, 2009.

[5] H. Fröhlich, J. K. Wegner, F. Sieker, A. Zell, Optimal assignment kernels for attributed molecular graphs, in *ICML*, pages 225-232, 2005.

[6] T. Gärtner, P. A. Flach, S. Wrobel, On graph kernels: hardness results and efficient alternatives, in *COLT*, pages 129-143, 2003.

[7] J. A. Gonzalez, L. B. Holder, D. J. Hook, Graph-based relational concept learning, in *proceedings of ICML*, pages 219-226, 2002.

[8] 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.

[9] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining spatial motifs from protein structure graphs, *Proceedings of the 8th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pp. 308-315, 2004.

[10] 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.

[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)*, pages 573-582, 2009.

[12] N. Jin, C. Young and W. Wang, GAIA: graph classification using evolutionary computation, in *Proceedings of the ACM SIGMOD International Conference on management of Data*, pages 879-890, 2010.

[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),* pages 844-855, 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)*, pages 578-586, 2008.

[15] A. Smalter, J. Huan, G. Lushington. A Graph Pattern Diffusion Kernel for Chemical Compound Classification. *In Proceedings of the 8th IEEE International Conference on Bioinformatics and BioEngineering* (*BIBE'08*), 2008.

[16] 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.

[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, 2002.

[19] X. Yan, P.S. Yu, J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems,* vol. 30, issue 4, pages 960-993, 2005.