Subsea: An efficient heuristic algorithm for subgraph isomorphism

V. Lipets, N. Vanetik, E. Gudes
Dept. of Computer Science, Ben-Gurion University
{ lipets, orlovn, ehud } @cs.bgu.ac.il

Abstract

We present a novel approach to the problem of finding all subgraphs and induced subgraphs of a (target) graph which are isomorphic to another (pattern) graph. To attain efficiency we use a special representation of the pattern graph. We also combine our search algorithm with some known bisection algorithms. Experimental comparison with other algorithms was performed on several types of graphs. The comparison results suggest that the approach provided here is most effective when all instances of a subgraph need to be found.

Keywords: graph algorithms, subgraph isomorphism, heuristic, bisection

1 Introduction

Subgraph isomorphism is an important and very general form of pattern matching that finds practical application in areas such as pattern recognition and computer vision, computer-aided design, image processing, graph grammars, graph transformation, bio computing, search operation in chemical structural formulae database, and numerous others. Theoretically, subgraph isomorphism is a common generalization of many important graph problems including finding Hamiltonian paths, cliques, matching, girth, and shortest paths. Subgraph isomorphism is also a basic component of any graph mining algorithm, an area which has become important in many data mining applications of unstructured and complex data. Graph mining algorithms often require finding not one but all subgraphs of the database graph isomorphic to a given small graph in order to compute the measure of statistical significance (also called 'support') of that small graph in the database. The subgraph isomorphism problem is generally NP-complete (see [19]) and computationally difficult to solve. But because of its wide applicability much effort was invested in finding algorithms and heuristics which can reduce the total search effort. Since the number of subgraphs isomorphic to a given small graph in a graph database can also be exponential in the size of the database, support computation is the bottleneck of all graph mining algorithms. Graph mining is an important and fast growing field of data mining with many applications such as mining biochemical structures, program flow analysis, mining XML structures and Web communities.

The most common technique to establish a subgraph isomorphism is based on backtracking in a search tree. In order to prevent the search tree from growing unnecessarily large, different refinement procedures are used. Best known is the one by Ullman (see [37]). Cordella et al have suggested another general graph matching algorithm in [27]. This algorithm is tailored for dealing with large graphs without using information about the topology of the graphs to be matched. It employs a recursive algorithm which grows a set of partial sub-graphs until the isomorphic sub-graph is found. The growing is done in such a way that reduces considerably the search space by providing pruning rules, and a dynamic ordering method for node matching. (The idea of pruning also appears in our algorithm as will be seen later.) The algorithm has become quite popular and the authors supply executables which can be used for evaluation [12].

On the other hand, a lot of work has been done to solve this problem in polynomial time for specific families of graphs, for example papers [15, 16, 26, 28]. Noga Alon has presented [2] a novel randomized method - color-coding - for finding simple paths and cycles of a specified length k, and other small subgraphs, within a given graph G.

The essential part of the recent research on subgraph isomorphism algorithms has been based on heuristic search techniques and was described in [1, 11, 13, 24, 29]. Some of the techniques work better in some special cases. For example, Akinniyi et al. in [1] use a tree searching procedure over the projections of the implicit product of the two graphs and utilize the minimum number of neighbors of the projected graphs to detect infeasible subtrees. Cheng and Huang in [11] use bitwise parallelism during the resolution process even though a sequential computer is used. Cortadella and Valiente [13] use a representation of relations and graphs by Boolean functions, which allows handling the combinatorial explosion in the case of small pattern graphs and large target graphs. Larrosa and Valiente [24] use the notion of neighborhood constraints. In particular, an efficient algorithm is described by Foggia et al. [18]. This algorithm uses a set of feasibility rules to significantly reduce the computational cost of the matching process.

Two additional algorithms have been published recently. Henrick and Krissinel [21] describe an algorithm that recursively enumerates all possible mappings of subgraphs of the two graphs. They improve the computational complexity of the algorithm by limiting the search area for possible matching. This is done by maintaining a static vertex matching matrix (VMM). The VMM stores all previous matchings, which the algorithm uses to determine if a certain mapping can or succeeded to cannot obtain a result with a sufficient size. If the mapping is present in the matrix, and is known to produce a result that is too small, the algorithm will simply skip calculating with that mapping and continue to the next one, thus saving computing the entire recursive branch.

Batz [5] describes a heuristic algorithm for finding a graph-subgraph mapping on directed labeled graphs, based on building a "plan graph" that helps model all available mappings and finds the mapping that takes the shortest time to build. First, a "search plan" is defined as an iterative mapping between two graphs built by atomic ("primitive") operations. on the primitive operations is defined, based on information from the host graph. This paper gives a heuristic algorithm of finding an optimal selection and ordering of the "search plan" that will give a matching between the two graphs. According to the analysis done by this paper, the algorithm runs in almost linear time when very few labels are repeated. However, cases where the graph is not labeled, or is labeled poorly (few different labels for many components) cause the algorithm to find more expensive search plans which take more time to calculate.

There are other variations of the subgraph isomorphism problem. An algorithm for subgraph isomorphism detection from a set of a priori known model graphs to an input graph that is given on-line is presented in [4]. The approach is based on a compact representation of the model graphs that is computed offline. A related problem is that of *Graph matching* where exact isomorphism is not required but some similarity measure is defined. One such measure is based on the "edit distance" [6], but other measures are possible. A recent paper by Sammoud et al. [35] describes an algorithm for finding an approximation to the maximal subgraph matching between two graphs. First, the authors define a function that measures the similarity of two graphs with respect to a certain mapping, and a function that measures the "desirability" of a single vertex matching for maximizing the similarity factor (a score function). They use these functions to iteratively build the best mapping (or close to it). mapping using a greedy algorithm. This greedy algorithm works by choosing the most desirable vertex matching available to it and adding it to its original graph mapping. The algorithm ends after the score function of the mapping no longer increases, or after a maximal number of iterations is reached.

Also, in some papers it is proposed to go a step further by introducing multi-vertex matchings, where a vertex in one graph may be matched with a set of vertices of the other graph [3, 7, 9]

Recently, the area of *graph mining* and especially finding frequent graph patterns, has become an important research area with applications in mining and discovering frequent patterns in semi-structured data such as Web and XML data [10, 39, 25], chemical compounds data such as [14], [32] and [31], or biological data [34]. All the algorithms for frequent graph mining require heavy use of subgraph isomorphism as a basic step; therefore, their performance is critically dependent on the efficiency of the subgraph isomorphism algorithm. Examples of graph mining algorithms which make heavy use of subgraph isomorphism can be found in [22, 38, 40].

Recall that the subgraph isomorphism problem is NP-complete in general. However, for any target graph with n vertices and any pattern graph with k vertices, where k is fixed, both the enumeration and decision problems can easily be solved in polynomial $O(n^k)$ time, and for some patterns an even better bound might be possible. However, for the general subgraph isomorphism problem, nothing better than the naive $O(n^k)$ bound is known. Thus one is led to the problem of finding heuristics for reducing the expected complexity of a subgraph isomorphism algorithm for a given fixed pattern graph.

In this paper, we propose an efficient heuristic method for finding all subgraphs and induced subgraphs of a target graph which are isomorphic to another connected pattern graph of a fixed size. The proposed algorithm is presented for non-directed labeled graphs but it has general validity, since no constraints are imposed on the topology of the pattern and the target graphs, and the method can be easily extended to the directed graphs case as well. To attain the efficiency we use a special representation of the pattern graph, which gives preference to checking the more "dense" parts first, thus obtaining "negative" answers as early as possible. We also combine our search algorithm with some known bisection algorithms and use them recursively which eases the search in large target graphs.

A special property of our algorithm is the fact that it finds *all* instances of the subgraph isomorphism and not just the first one. Our motivation was graph mining in the *single* graph-setting case, see e.g., [23, 20]. The presented algorithm is particularly useful when the target graph is much larger than the pattern graph, a case which is common in many applications, such as mining the Web or Social networks [8]. Recently the interest in mining all occurrences in a single large graph has increased. An example biological application was presented by Zhang et al [41]. The following is a citation from [41]: "For example, biological networks (protein-protein interaction networks, and gene regulatory networks, etc.) are often much larger than the graphs used in previous exact indexing methods. A single biological network may contain thousands or tens of thousands of vertices. Biologists may want to find all the occurrences of a particular pattern (subgraph), e.g., protein type A interacts with protein type B and C, and protein type C interacts with protein type A and D. In different occurrences of the pattern, the exact proteins involved may be different since multiple proteins may share the same type. As a result, all occurrences of a particular pattern need to be retrieved."

To evaluate our approach we implemented our algorithm and compared it to one of the most popular subgraph isomorphism algorithms, the one by Ullman [37], and to the algorithm of Cordella et al. [27], using latest publicly available implementations at [36]. Given a graph and a database graph, both of these algorithms are capable of finding and reporting all subgraph isomorphisms the given graph and the database graph, either as induced graph or not. The main conclusions of the evaluation is that our Subsea algorithm outperforms both these algorithms in finding all the instances of a subgraph in a graph, while finding a first instance of a subgraph in a graph may be slower than either one of them. Thus Subsea is particularly attractive for graph mining in a single transaction setting case ([23, 41]).

The rest of this paper is structured as follows: Section 2 is devoted To definitions and notations, and also contains the high-level outline of the algorithm. Sections 3, 4, and 5 present the necessary auxiliary algorithms; in Section 3 we describe some well known bisection algorithms, in Section 4 we give some heuristic methods to represent a “small” pattern graph, and Section 5 describes our search strategy. In section 6 we present our main algorithm to find all occurrences of the given subgraph in the target graph and which uses the above auxiliary algorithms. In Section 7, we present the experimental comparison of our algorithm with the algorithms presented in [37] and [27], and we show the effectiveness of the proposed heuristic method. We conclude and propose further research in Section 8.

2 Definition and notations

2.1 General definitions

A graph $G = (V, E)$ is called *vertex-labeled* (or simply *labeled*) if a mapping $l : V \to \mathbb{N}$ is given. $l(v)$ is called a label of a vertex v. Throughout this paper we always speak of a labeled graph unless otherwise specified. Two graphs which contain the same number of vertices with the same labels connected in the same way are said to be isomorphic. Formally, two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic*, denoted by $G_1 \cong G_2$, if there is a (label-preserving) bijection $\varphi : V_1 \longrightarrow V_2$ such that, for every pair of vertices $v_i, v_j \in V_1$, $(v_i, v_j) \in E_1$ if and only if $(\varphi(v_i), \varphi(v_j)) \in E_2$. Bijection φ is said to be an *isomorphism* between two graphs. An isomorphism which maps $v \in V$ into $v' \in V$ we denote by $(v \to v')$-isomorphism.

A graph G' is a subgraph of a given graph G if vertices and edges of G' form subsets of the vertices and edges of G. If G' is a subgraph of G, then G is said to be a *supergraph* of G'. We denote this relationship by $G' \subseteq G$.

A graph $G_1 = (V_1, E_1)$ is *isomorphic to a subgraph* of a graph $G_2 = (V_2, E_2)$ if there exists a subgraph of G_2, say G_2', such that $G_1 \cong G_2'$. In this case the corresponding bijection between vertices of G_1 and G_2' is said to be a *subisomorphism* between two graphs. Note that for labeled graphs, this bijection must be label-preserving.

An *induced subgraph* is a subset of the vertices of a graph G together with all edges whose endpoints are both in this subset. Formally, let G be a graph and $V' \subset V(G)$. We call the graph $G' = (V', E(G) \cap \{(u, v) | u, v \in V'\})$ the *subgraph of G induced by V'* and we denote it by $G(V')$. The relationship between G' and G in this case we denote by $\sqsubseteq$.

Thus, an induced subgraph isomorphism is an isomorphism with an induced subgraph of a given graph, i.e., a graph $G_1 = (V_1, E_1)$ is isomorphic to an induced subgraph of a graph $G_2 = (V_2, E_2)$ if there exists an induced subgraph of G_2, say G_2', such that $G_1 \cong G_2'$. In this case a corresponding bijection between vertices of G_1 and G_2' is said to be an *induced subisomorphism* between two graphs.

Note that for induced subgraph isomorphism all edges of G_1 must be mapped into edges of G_2', but also all non-edges of G_1 must be mapped to non-edges of G_2'. Thus without loss of generality we assume that the number of non-edges of the pattern graph is larger than the number of edges (for the induced subgraph isomorphism problem only), otherwise we can solve the inverse graph problem (recall that the inverse graph is obtained by replacing non-edges by edges). An *automorphism* of a graph $G = (V, E)$ is a graph isomorphism with itself.

The neighborhood of a vertex v in graph G, denoted by $N_G(v)$, is the set of vertices in G that are adjacent to v, i.e., $N_G(v) = \{u \in V | (u, v) \in E\}$. For any $e \in E(G)$, we define $G - e = (V(G), E(G) \setminus \{e\})$.

A *cut* in G is a partition of V into two disjoint sets, say $(A, \bar{A})$. For $A \subseteq V$, denote by $e(A, \bar{A})$ the set of all A-$\bar{A}$ edges:

$$e(A, \bar{A}) = \{(u, v) \in E \mid u \in A, v \notin A\}\,.$$

The *size* of the cut $(A, \bar{A})$, denoted by $c(A, \bar{A})$, is the number of edges having exactly one vertex in A and the other in $\bar{A}$, namely $|e(A, \bar{A})|$.

The *minimum bisection* is a cut $(A, \bar{A})$ minimizing $c(A, \bar{A})$ over all sets with A of size $\lceil |V|/2 \rceil$. For arbitrary graphs G, the problem of determining the minimum bisection is NP-hard [19].

For any graph $G = (V, E)$ we assume that there exists a total order on the set of vertices V which enables us to compare any two vertices formally for each $u \neq v \in V$ either $v < u$ or $u < v$. We use the notation $w = \langle w_1, w_2, \ldots, w_k \rangle$ to represent a sequence of k elements and denote the i-th element by $w[i]$ for all $i \in \{1, 2, \ldots, k\}$. Two sequences of numbers can be compared according to the lexicographical order.

In order to test bisection algorithms, we need a way to generate graphs with known best bisections. We use the following graph generation methods:

The probability distribution $G_{n,p,r}$ is commonly used in the literature on minimum bisection because the minimum bisection of a generated graph is known with high probability, and the simplicity of the distribution leads to the possibility of determining analytically the probability of success of some algorithms.

$G_{n,p,r}$ is a probability distribution on graphs with vertex set $\{1, 2, \ldots, n\}$ in which the presence of each possible edge is independent, with probability p for edges within $\{1, 2, ...n/2\}$ or $\{n/2 + 1, \ldots, n\}$ and probability $r < p$ for other edges. If $p - r$ is sufficiently large, then with high probability, the minimum bisection is the obvious one.

Throughout this paper we use notations of "*small*" and "*large*" graphs, denoted by G_S and G_L respectively. We assume that $|E(G_S)| << |E(G_L)|$, where $E(G_S)$ is the pattern graph and $E(G_L)$ is the target graph. The number of edges of a "large" graph was usually three or more orders of magnitude more than that of the "small" graph.

2.2 Outline of the Subsea algorithm

The algorithm, called **Subsea**, relies on the fact that the target graph is much larger than the pattern graph. Therefore it uses *bisection* to decompose the "large" graph and search for isomorphisms in the bisected parts (recursively). Clearly, if the bisection is minimal the amount of time which is spent on checking edges belonging to the bisection is also small, and the work can proceed on the two decomposed parts which are considerably smaller. This, of course, relies on the fact that the target graphs are not highly connected and that the bisection is not very large, something which is true in most of the applications we mentioned in the Introduction. The isomorphism check itself uses a canonical numbering scheme called *traverse history* constructed in such a way that the isomorphism check will find non-matching subgraphs as early as possible. Since the isomorphism check can start at any pair of adjacent vertices, a traverse history may be generated at most twice for every edge in the pattern graph and, in the worst case, $2e$ traverse histories are generated for e edges.

The main steps of the algorithm are as follows:

1. In a preprocessing step generate all the traverse histories of the pattern graph.

2. Decompose the target graph by finding an approximate minimum bisection (heuristically).

3. Check all possible isomorphisms using edges belonging to the bisection.

4. Apply step 2 again recursively on the two parts of the bisected graph until the target graph becomes comparable in size to the "small" graph.

Section 3 describes steps 1 and 2. Section 4 describes the isomorphism check using traverse histories. Section 5 describes the subgraph search algorithm and Section 6 discusses the entire algorithm.

There are two main reasons why Subsea performs as it does. First, traverse histories for the small graph are generated once in the pre-processing stage of the algorithm and the small graph is never traversed later during the search. We 'pay' in running time for that, but since the number of database subgraphs isomorphic to given graph can be exponential in the size of the database, a lot of running time is saved later. This is also the reason our approach has shown an advantage over dynamic re-ordering implemented in the algorithm of Cordella et al ([27]). The second reason is that bisecting the large graph each time decreases search space dramatically each time it is applied.

3 Bisection algorithms

In this section we describe two well-known approximation algorithms for finding a minimum bisection of a given graph G. Note that since the minimum bisection is only a tool to decompose the problem efficiently, we are not required to find the actual minimum bisection (which is a hard problem), but it is enough to provide an approximation for it.

We ignore vertex labels in this construction, because minimum bisection size depends only on the number of edges and not on vertex labels.

3.1 Black Holes Bisection algorithm

This simple method which we devised works well on graphs with a very thin minimum bisection, and has the advantage of running very fast.

Given a graph $G = (V, E)$, the algorithm runs as follows. Initialize $B_1 = B_2 = \emptyset$. These are the black holes. Choose uniformly at random an edge from $V \setminus (B_1 \cup B_2)$ to B_1, and add the first endpoint to B_1. If no such edge exists, choose uniformly at random among all vertices in $V \setminus (B_1 \cup B_2)$ for a vertex to add to B_1. Do the same for B_2. Repeat until $|B_1 \cup B_2| = |V|$. (See algorithm 3.1.)

The motivation for this algorithm is that, assuming that the black holes are currently contained in opposite sides of a minimal bisection, we are likely to add to each hole a vertex from the correct side because there will be more edges from this side.

Algorithm 3.1. Black Holes Bisection

Input: Graph $G = (V, E)$
Output: Cut $(B, \bar{B})$ of V which approximates a minimal bisection

```
1: $B_1 \longleftarrow B_2 \longleftarrow \emptyset$
2: $B_0 \longleftarrow V \setminus (B_1 \cup B_2)$
3: repeat
4:     $Add2Hole(1)$
5:     $Add2Hole(2)$
6: until $B_0 = \emptyset$
```

```
7: return $(B_1, \bar{B}_1)$
```

Algorithm 3.1 :: **Procedure** *Add2Hole(i)*

```
1: if $B_0 = \emptyset$ return
2: $E_0 \longleftarrow \{(u,v) : u \in B_i, v \in B_0\}$
3: if $E_0 \neq \emptyset$ then
4:     chose randomly $e = (u,v) \in E_0$ with $v \in B_0$
5: else
6:     chose randomly $v \in B_0$
7: end if
8: $B_i \longleftarrow B_i \cup \{v\}$
9: $B_0 \longleftarrow B_0 \setminus \{v\}$
```

Let G be a graph chosen at random according to the distribution $G_{n,p,r}$. The probability that randomized black holes will find the minimum bisection on G is $(p/p+r)^{n-2}/2$.

In fact, in the first step, we will simply be picking two vertices uniformly at random; the probability that they will lie on opposite sides of the canonical bisection is $\frac{1}{2}$ (or close to it). Assume that after k steps $H_1 \in P_1$ and $H_2 \in P_2$, where P_1 and P_2 are the two halves of the canonical bisection. The expected number of edges from H_1 to $P_1 \setminus H_1$ is $pk(n/2-k)$, while the expected number from H_2 to $P_2 \setminus H_2$ is $rk(n/2-k)$. Hence the probability that we pick an edge from the correct side is $p/(p+r)$. Since we must make $n-2$ such correct choices, our overall probability of success is as stated.

Obviously this success rate becomes quite bad as n becomes large if p and r are constant. But consider the case where $r = c/n$ for some constant c, or equivalently where the number of edges across the minimum bisection is $O(n)$. Now the probability of success is greater than

$$\frac{1}{2}(1-\frac{r}{p+r})^n > \frac{1}{2}(1-\frac{r}{p})^n = \frac{1}{2}(1-\frac{c}{np})^n$$

This last expression approaches the non-zero constant, say q, as n approaches infinity. Hence by running randomized black holes $1/q$ times, a number depending only on c and p, we achieve a success rate of at least $1/e$.

3.2 Simple Greedy Bisection method

The obvious greedy algorithm for the graph bisection problem consists of starting with any bisection $(B, \bar{B})$ of V (in particular, we can use a bisection obtained by Black Holes algorithm) and computing a new bisection by swapping the pair of elements $x \in B,\ y \in \bar{B}$ which maximizes the gain (number of edges in $(B, \bar{B})$ before the swap minus the number after the swap). This process is repeated until the maximum gain is less than zero or until the maximum gain is zero and another heuristic has determined that it is time to stop swapping "zero gain" pairs. The issue of breaking ties is resolved by choosing the pair to swap uniformly at random from the set of pairs for which the gain is maximum.

To efficiently determine the cost of a swap, we store for each vertex $x \in B$ its internal cost $I(x)$, which is the number of edges $(x, x') \in E$ with $x' \in B$, and its external cost $E(x)$, which is the number of edges $(x, y) \in E$ with $y \in \bar{B}$. This can be done in $O(m + n)$ time, and can be updated after swapping x and y in $O(|N_G(x)| + |N_G(y)|)$ time. The gain in swapping x and y is then $gain = E(x) - I(x) + E(y) - I(y) - 2w(x, y)$, where $w(x, y) = 1$ if $(x, y) \in E$ and $w(x, y) = 0$ otherwise. We can therefore determine the best pair to swap in $O(n^2)$ time by simply running through all $(n/2)^2$ candidate pairs, keeping track of the leaders for the random choice at the end.

4 Traverse history

In this section we give two heuristic methods to represent a "small" pattern graph. Each such representation enumerates the vertices of the pattern graph in a particular order. This order will determine the order in which the isomorphism check is done. In the experiments that we carried out, these simple methods essentially improved our search technique by reducing the run time of the corresponding search algorithm (when compared to the case when some induced traverse history was chosen randomly). The motivation for using these representations will become clearer in Section 5.

Let $d : V \longrightarrow \mathbf{N}$ be a numbering of vertices of graph G. Let l_i denote the label of the vertex that has number i in numbering d, i.e., $l_i := l(v),\ d(v) = i$; let $N_i := \{d(u) < i : u \in N_G(v),\ d(v) = i\}$. The sequence $\langle (l_1, N_1), (l_2, N_2), \ldots, (l_{|V|}, N_{|V|}) \rangle$ is called a *traverse history* of graph G induced by numbering d. In particular, we say that traverse history started on vertices $v_1, v_2, \ldots, v_i$, if $d(v_i) = i$, where $i \leq |V|$. Informally, N_i is the set of adjacent vertices to i with numbering smaller than i. The traverse history $\langle (l_1, N_1), (l_2, N_2), \ldots, (l_{|V|}, N_{|V|}) \rangle$ is *connected* if $N_i \neq \emptyset$ for each $2 \leq i \leq |V|$. See Figure 1 for an example of N_i on given graph. For traverse history H, $H[i]$ denotes (l_i, N_i), $H[i].l$ denotes l_i and $H[i].N$ denotes N_i.

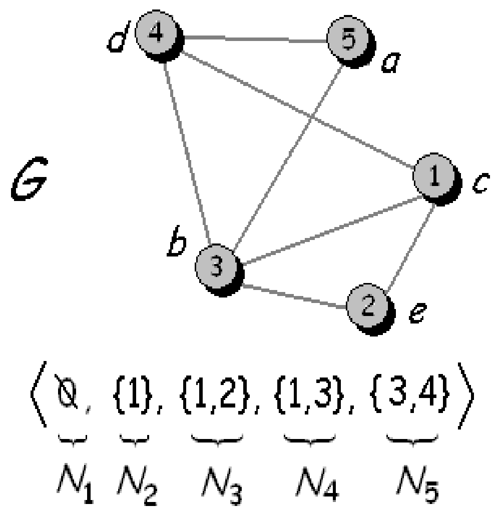

Figure 1: *Traverse history of graph G (labels omitted)*

We have two heuristic methods, Algorithm 4.1 - the DFS method, and 4.2 - the "black holes" method, for finding a traverse history for "small" pattern graph with given starting vertices. Algorithm 6.1 finds such traversals for all pairs of starting with adjacent vertices. This is needed since the starting edge of the isomorphism check may be arbitrary. The traverses obtained in this stage are used then as an input for the search algorithm which is described in Section 5.

4.1 The DFS approach

Our first approach is based on a modification of the well known DFS (Depth-First Search) algorithm which provides a general technique for traversing a graph (traversing a graph means visiting all of its vertices in some systematic order). Recall that the DFS traversing is not deterministic, i.e., for any graph G a number of traverses is possible. We extend the traversing strategy by some heuristic rules, to provide a "fastest" return to the visited nodes.

In Algorithm 4.1 we present our modification of the DFS algorithm. This algorithm produces a traverse history of a connected graph G starting at given vertices v_1, v_2. The same as the DFS algorithm, Algorithm 4.1 visits all vertices of G in some schematic order, thereby producing the numbering of vertices of $V(G)$. Note that the obtained traverse history is induced by the above numbering.

Algorithm 4.1. Traverse History

```
Input: Graph $G = (V, E)$, starting vertices $v_1, v_2 \in V$, with $(v_1, v_2) \in E$
Output: Traverse history $H$ started on $v_1, v_2 \in V$.
1: for all $v \in V$ do
2:     $d(v) \longleftarrow 0$
3: end for
4: $vtime \longleftarrow etime \longleftarrow 1$
5: $Visit(v_1)$
6: return $H$
```

Algorithm 4.1 :: **Procedure** $Visit(v)$

```
1: $d(v) \longleftarrow vtime$
2: $H[vtime++] = (l(v), \{0 < d(u_1) \leq ... \leq d(u_m) : u_1, ..., u_m \in N_G(v)\})$
3: if $v = v_1$ then $Visit(v_2)$
4: $N_0 \longleftarrow \{u \in N_G(v) : d(u) = 0\}$
5: while $N_0 \neq \emptyset$ do
6:     choose $w \in N_0$ with lexicographically minimal $EstimateNext(w, v)$
       pair
7:     if $d(w) = 0$ then $Visit(w)$
8:     $N_0 \longleftarrow N_0 \setminus \{w\}$
9: end while
```

Algorithm 4.1 :: **Procedure** $EstimateNext(w, v)$

```
1: $S \longleftarrow \{w\}$
2: $len \longleftarrow 1$
3: repeat
4:     $N_S \longleftarrow \cup_{z \in S} N_{G-(v,w)}(z)$
5:     $p = |\{y \in N_S : d(y) > 0\}|$
6:     if $p > 0$ then return $\langle len, -p \rangle$
7:     $len++$
8:     $S \longleftarrow S \cup N_S$
9: until $N_S \neq \emptyset$
10: return $\langle \infty, -|S| \rangle$
```

The main goal of this approach is the choice of the next vertex to visit. To implement our strategy, we define BFS (Breadth first search) based procedure *EstimateNext*, which helps to choose a next vertex to visit, by finding for any unvisited vertex (adjacent to the last visited vertex) an estimation pair using the following heuristic method, which prefers to visit first nodes which have high "proximity" to the current node: Let v be the last visited vertex and let $w \in N_G(v)$, be an unvisited vertex adjacent to v. We distinguish between two cases (illustrated in Fig. 2):

a) there exists a path in $G - (v, w)$ started in w and leading to a visited vertex,

b) such a path does not exist.

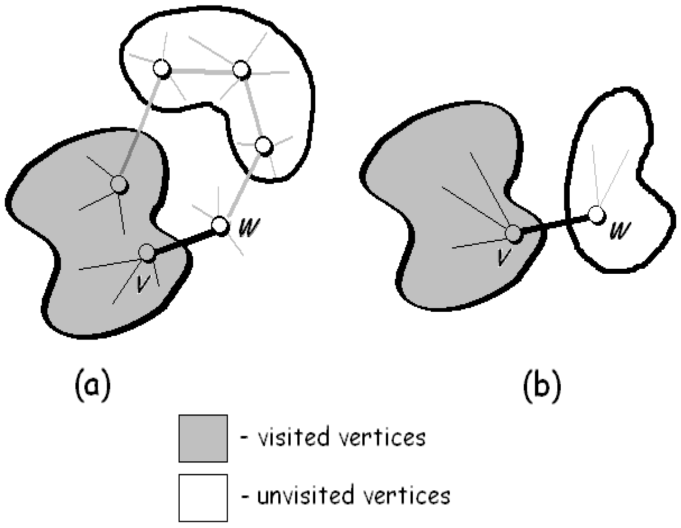

Figure 2: *Estimation cases*

The first coordinate of the estimation pair is at least 1 in case (a) and equal to ∞ in case (b). This provides highest priority to case (a) and lowest to (b), since the minimal value returned by *EstimateNext* procedure is chosen by the lexicographical order. Actually, in case (a) the first coordinate is the length of the shortest path (paths) leading to the visited vertices and the second coordinate equals the number of visited vertices that the above path (paths) is leading to (multiplied by -1). This forces the choice of the next vertex which belongs to the shortest path (paths) leading to the maximal number of visited vertices. In case (b) the second coordinate equals to the size of the connected component of $G - (v, w)$ containing w (multiplied by -1). This forces the choice of the next vertex which belongs to the smallest connected non-visited component, when the above path leading to the visited vertices does not exist.

Remark In fact, the run time $O((|E| + |V|)^2)$ of Algorithm 4.1 can be essentially improved. But since we apply this algorithm on the "small" graph only at the preprocessing stage, it is not necessary, and we present the simplest version here.

4.2 Black Holes approach

In this section we give another approach for finding induced traverse history for "small" pattern graph. In the experiments that we carried out, this method essentially improved the run time when the main algorithm was applied on randomly generated graphs.

For any traverse history $H = \langle (l_1, N_1), (l_2, N_2), \ldots, (l_{|V|}, N_{|V|}) \rangle$ of graph G we define a *traverse integrality* as follows:

$$I_G(H) = \sum_{i=1}^{|V|} (|V| + 1 - i)|N_i|.$$

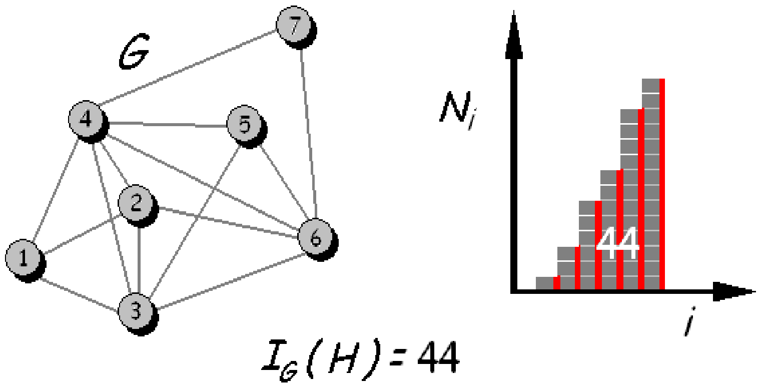

Figure 3: *Traverse integrality*

Actually this value gives a measure about the number of edges of subgraphs of some given graph G induced by the first $i = 1, 2, \ldots, V(G)$ vertices of any numbering d.

We provide a simple (and very fast) randomized method for finding the induced traverse history with the largest (or the smallest) traverse integrality. This method is very similar to the Black Holes Bisection algorithm, presented in Section 3.

Given a graph $G = (V, E)$ and starting vertices v_1, $v_2 \in V$, the algorithm runs as follows: Initialize $B = \{v_1, v_2\}$. This is the black hole. Choose uniformly at random an edge from $V \backslash B$ to B, and add the first endpoint to B (in the Case when we are seeking the smallest integrality, choose Randomly a vertex in $\{v \in \bar{B} : N_G(v) \cap B \neq \emptyset\}$ and add it to B). Enumerate v and for each vertex in $N_G(v) \cap B$ add $d(u)$ to $N_{|d(v)|}$. Repeat until $|B| = |V|$.

The motivation for this algorithm is the same as for the bisection algorithm: assuming that the black holes are currently contained in B, we are likely to add to a hole a vertex with the largest number of neighbors in B, thereby increasing the corresponding $|N_i|$ value.

Algorithm 4.2. Hole Integration

```
Input: Graph $G = (V, E)$, starting vertices $v_1, v_2 \in V, (v_1, v_2) \in E$
Output: Traverse history $H$ started on $v_1, v_2 \in V$.
 1: for all $v \in V$ do
 2:     $d[v] \longleftarrow 0$
 3: end for
 4: $d(v_1) = 1$
 5: $d(v_2) = 2$
 6: $B \longleftarrow \{v_1, v_2\}$
 7: while $|B| \neq |V|$ do
 8:     $Add2HoleAndVisit()$
 9:     $I \longleftarrow I + etime$
10: end while
11: return $(A_H, I)$
```

```
Algorithm 4.2 :: Procedure $Add2HoleAndVisit$
 1: chose randomly $e = (u, v) \in e(B, \bar{B})$, where $v \in \bar{B}$
 2: $d(v) = vtime$
 3: $H[vtime + +] = (l(v), \{d(u) : u \in N_G(v) \cap B\})$
```

Note that since initially $B = v_1, v_2$ and each vertex added to B has a neighbor in B, the sequence of the set obtained by Algorithm 4.2 is connected traverse history of graph G started on vertices v_1,$v_2 \in V$.

Remark The problem of finding traverse history with largest integrality is not well studied. We suppose that it is NP-complete, and we will try to prove this fact in the future. In our approach only the "small" graph is used as an input for this algorithm, thus we can apply it a sufficiently large number of times to attain a high probability that largest (smallest) integrality is found.

5 Search technique

In this section we start with an algorithm which searches for the isomorphic subgraphs by given traverse history. In fact, the above algorithm seeks for subgraphs satisfying the condition of the following obvious lemma.

Lemma 5.1. *Let* $H_1 = \big\langle (l_1, N_1), (l_2, N_2), \ldots, (l_{|V_1|}, N_{|V_1|}) \big\rangle$ *be a traverse history of graph* $G_1 = (V_1, E_1)$ *with labeling* l*, induced by numbering* d_1 *of* V_1*. A subgraph (induced subgraph)* G_2' *a of a graph* $G_2 = (V_2, E_2)$ *with labeling* m *is isomorphic to* G_1 *if* $|V_2| \geq |V_1|$ *and there exists* $H_2 = \big\langle (m_1, M_1), (m_2, M_2), \ldots, (m_{|V_2|}, M_{|V_2|}) \big\rangle$*, a traverse history of graph* G_2 *induced by some numbering* d_2 *of* V_2 *such that* $l_i = m_i$ *and* $N_i \subseteq M_i$ $(N_i = M_i)$ *for each* $1 \leq i \leq |V_1|$*. Moreover* $d_2^{-1} \circ d_1$ *is an isomorphism between* G_1 *and* $G_2' \subseteq G_2$ $(G_2' \sqsubseteq G_2)$*.*

In particular, graph G_1 *is* $(v_1 \to v_1', v_2 \to v_2')$*-isomorphic to subgraph (induced subgraph) of* G_2 *if* $v_1, v_2 \in V_1$ *and* $v_1', v_2' \in V_2$ *are starting vertices of* H_1 *and* H_2*, respectively.* □

Our search technique is based on the DFS method and is depicted in Algorithm 5.2. As an input it receives "large" target graph $G_L = (V_L, E_L)$, starting vertices $v_1, v_2 \in V_L$ and traverse history H of a "small" pattern graph G_S, previously obtained by Algorithms 4.1 or 4.2. It finds all subgraphs (induced subgraphs) of G_L $(v_1 \to v_1', v_2 \to v_2')$-isomorphic to G_S, where v_1', v_2' are the first two vertices of the traverse history H. An example of such an isomorphism is given in Figure 4. All the labels are assumed to be the same in this example and are therefore omitted from traverse histories. Here, G_S is $(v_1 \to v_1', v_2 \to v_2')$-isomorphic to two subgraphs of G_L, induced by vertices v_1', v_2', v_3' and v_1', v_2', v_4', respectively. The traverse history of G_L shown in the picture starts with v_1' and v_2', and therefore both these isomorphisms are discovered by simple comparison of the traverse histories.

The extension of the above algorithm for the induced subisomorphism is the use of the "Black" edges, namely some edge is colored to "Black" instead of being removing from the graph by the main algorithm described in the next section. Note that only if we finished seeking for all subgraphs $(v_i \to v_1', v_j \to v_2')$-isomorphic to G_S where $i, j = 1, 2 \ldots, |V_S|$, then the

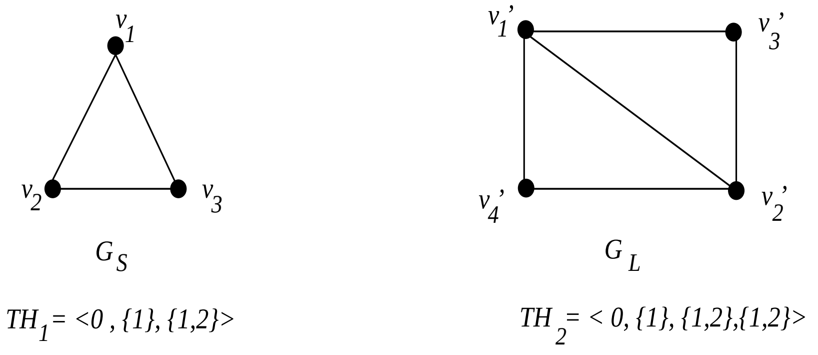

Figure 4: *Estimation cases*

edge (v_1', v_2') can just be removed from the graph. However, in an induced subgraph isomorphism case, this edge can be used as non-edge after being removed. Thus, in each stage of Algorithm 5.2 we verify that the found subgraphs do not contain "*Black*" edges.

Notation: in algorithms throughout the rest of the chapter we write "$/* \ldots */$" for the induced subisomorphism case.

Algorithm 5.2. SEARCH TRAVERSE

Input: Graph $G_L = (V_L, E_L)$, starting vertices $v_1', v_2' \in V_L$, traverse history H of G_S, $/*$ set of "*Black*" edges $*/$

Output: All subgraphs $/*$ or induced subgraphs $*/$ of G_L $(v_1 \to v_1', v_2 \to v_2')$-isomorphic to G_S, where v_1, v_2 are starting vertices of H $/*$ and do not contain "*Black*" edges $*/$

```
1: for all $v' \in V_L$ do
2:     $g(v') \longleftarrow 0$
3: end for
4: $g(v_1') \longleftarrow 1$
5: $g(v_2') \longleftarrow 2$
6: return $SearchVisit(\{v_1', v_2'\}, \emptyset)$
```

Algorithm 5.2 :: **Procedure** $SearhVisit(V', E')$

```
1: $vtime \longleftarrow |V'|$
2: $v' = g^{-1}(vtime)$
3: if $H[vtime].N \not\subseteq \{g(u') > 0 : (v', u') \in E\}$ or $H[vtime].l \neq l(v')$ then
   return $false$
```

```
4: /* **if** $|H[vtime]| \neq |\{g(u') > 0 : (v', u') \in E\}|$ **or** $H[vtime].l \neq l(v')$
   **then return** $false$ */
5: $E' \longleftarrow E' \cup \{(u', v' : d(u') \in H[vtime].N\}$
6: **if** $|H| = vtime$ **then return** $\{(V', E')\}$
7: **choose** $v \in H[vtime + 1].N$
8: $L \longleftarrow \cap\{N_{G_L}(u) : u \in H[vtime + 1].N\}$
9: $S \longleftarrow \emptyset$
10: **for** each $w \in L$ **do**
11:   **if** $g(w) = 0$ /* **and** $color(v, w) \neq$ "$Black$" */ **then**
12:     $g(w) \longleftarrow vtime$
13:     $S \longleftarrow S \cup SearchVisit(V' \cup \{w\}, E')$
14:     $g(w) \longleftarrow 0$
15:   **end if**
16: **end for**
17: **return** $S$
```

The main idea of this algorithm is to perform recursively all possible traversals (starting on given vertices $v_1, v_2 \in V_L$) on the subgraphs of the target graph. Each such traverse visiting vertices in a certain order proceed by numbering the visited vertices $g : V' \longrightarrow \mathbf{N}$. In each stage of the procedure we verify that the condition of Lemma 5.1 holds for a given traverse history H and for traverse history H' induced by g.

To implement this strategy we use a $SearchVisit$ procedure. Logically it can be divided into two parts: in the first part we are trying to check that the condition of Lemma 5.1 holds (lines 3-4), namely that $H[i].N \subseteq H'[i].N$ and $l_i = l'_i$ (for induced subgraph isomorphism $H[i].N = H'[i].N$ and $l_i = l'_i$) for $i \in \{1, 2, \ldots |V_S|\}$. If this condition holds, we extend E' by the corresponding (not "$Black$" for induced subgraph isomorphism case) edges. In the case when the set of vertices V' is extended to the number of vertices of the pattern graph (namely to $|H|$), then isomorphic subgraph (induced subgraph) $G' = (V', E')$ is found (line 6). Otherwise, in the second part we are trying to extend V' by adding each unvisited vertex adjacent to the one of the visited vertices (recall that H is connected traverse history, then each newly visited vertex has a visited neighbor), thereby finding all possible subgraphs of G_L isomorphic to G_S (line 12).

Actually, the above procedure will work "faster" if it does not enter the second recursive part. Except for the case when the subisomorphism is found, it happens only when there is no edge between two visited vertices when the corresponding edge exists in the pattern graph (or in addition there is an edge between two visited vertices when the corresponding edge does not exist in the pattern graph for the induced subisomorphism case). If this mismatch is discovered in the earlier stage of Algorithm 5.2, then its run time is reduced. This observation explains the importance of using the heuristics presented in Algorithms 4.1 and 4.2, since this technique forces the above checking to be done as soon as possible, thereby decreasing the expected run time.

Remark Obviously, each isomorphic subgraph is found by Algorithm 5.2 exactly k times, where k is the number of $(v_1 \to v_1', v_2 \to v_2')$-automorphisms of graph G_S. Therefore the modification of Algorithm 5.2 to just count the number of such subgraphs of G_L which are isomorphic to G_S can be obtained as follows:

- $SearchVisit$ procedure should return 1 instead of (V', E') in the case when isomorphic subgraph is found.
- The value returned by Algorithm 5.2 should be divided by k. (Note that the corresponding number of automorphisms can be also calculated in precomputational stage by Algorithm 5.2.)

The traverse history with the smallest integrality approach can improve the run time of the main search algorithm for the induced subisomorphim problem in following cases:

- The number of edges of the "large" target graph is close to the number of non-edges (recall that we assume that the number of non-edges of the "large" pattern graph is larger than number of edges)
- The number of edges of the "small" pattern graph is significantly smaller than the number of non-edges.

6 Subsea: Subgraph Isomorphism Algorithm

In this section we combine all the previously defined algorithms to achieve the main goal: subgraph (induced subgraph) isomorphism algorithm.

First, we present an algorithm which collects all traverse histories of a "small" pattern graph in a precomputation stage. Then we present the main algorithm, which finds all the subgraphs (or induced subgraphs) of a given target graph G_L isomorphic to a pattern graph G_S. Note that since our patterns are small, the space required to store all the traverse histories is not prohibitive.

6.1 Precomputation stage

A pair of vertices $\langle v_1, v_2 \rangle \in V^2$ of graph G we will call *redundant* if there exists an $(v_1 \to v_1', v_2 \to v_2')$-automorphism of G such that $\langle v_1, v_2 \rangle > \langle v_1', v_2' \rangle$.

Algorithm 6.1 finds a corresponding traverse history for each non-redundant pair of adjacent vertices. Note that each edge of the pattern graph may derive 0, 1, or 2 traverse histories.

Algorithm 6.1. All Traverse Histories

```
Input: Graph $G = (V, E)$
Output: Set of traverse histories of $G$
1: $A \longleftarrow \emptyset$
2: for each $\langle v_1, v_2 \rangle \in V^2$ such that $(v_1, v_2) \in E$ do
3:     run Algorithm 4.1 on $G, v_1, v_2$ to obtain traverse history $H_{v_1,v_2}$
4:     if ! $IsRedundant(v_1, v_2)$ then $A \longleftarrow A \cup \{H_{v_1,v_2}\}$
5: end for
6: return $A$
```

```
Algorithm 6.1 :: Procedure $IsRedundant(v_1, v_2)$
1: for each $\langle v_1', v_2' \rangle \in V^2$ such that $\langle v_1', v_2' \rangle \in E$ and $\langle v_1', v_2' \rangle < \langle v_1, v_2 \rangle$ do
2:     run Algorithm 5.2 on $G, v_1', v_2', H_{v_1 v_2}$ to obtain set $S$ of graphs
3:     if $S \neq \emptyset$ then return $true$ else return $false$
4: end for
```

We use Algorithm 5.2 to find redundant edges, which also can be used to look for the corresponding automorphisms.

This approach enables us to minimize the number of stored traversals, when a set of automorphisms of G is non-empty, thereby reducing the running time of the main search algorithm.

6.2 Main algorithm

In general our technique can be described as follows.

1. Find the traverse history (using Algorithm 6.1) for each non-redundant pair of adjacent vertices of the pattern graph.

2. Divide vertices of a given "large" target graph into two parts using the bisection methods of Section 3.

3. For each edge with endpoints in distinct parts of the obtained bisection, find the set of all subgraphs (or induced subgraphs) containing this edge and isomorphic to a given pattern graph (Algorithm 5.2 is used).

After performing these steps, we continue to apply, in recursive manner, the same approach on two subgraphs of G_L induced by the two parts of bisection. We stop when we get a graph with fewer vertices than the pattern graph.

Algorithm 6.2. ALL SUBGRAPH ISOMORPHISMS

```
Input: Pattern graph $G_S = (V_S, E_S)$; target graph $G_L = (V_L, E_L)$; the set
    $A$ of all non-redundant traverse histories on graph $G_S$.
Output: All subgraphs (induced subgraphs) of $G_L$ which are isomorphic
    to $G_S$
1: Apply Algorithm 6.1 on $G_S$ to obtain set of traverse histories $A$ of $G_S$.
2: return $SubIso(A, G_L)$
```

```
Algorithm 6.2 :: Procedure $SubIso(A, G_L)$
1: if $|V_S| > |V_L|$ then return $\emptyset$
2: Apply several times Algorithm 3.1 on $G_S$ to obtain $(B, \bar{B})$ − bisection
   of vertices of $G_L$
3: for each $e = (v_1, v_2) \in (B, \bar{B})$ do
4:    for each $H \in A$ do
5:       Apply Algorithm 5.2 on $G_L, v_1, v_2, H$ to obtain $S$ − a set of sub-
         graphs of $G$
6:       $G \longleftarrow G - e$ /* or $color(e) \longleftarrow$ "$Black$" */
7:    end for
8: end for
9: Eliminate duplicates from $S$
10: return $S \cup SubIso(G_L(B)) \cup SubIso(G_L(\bar{B}))$
```

Theorem *Algorithm 6.2 applied on (G_L, G_S) finds all subgraphs of target graph G_L which are isomorphic to pattern graph G_S.*

Proof The correctness is assured by Lemma 5.1. Completeness is proved by induction in the number of vertices of the target graph. If $|V_L| < |V_S|$, then, obviously, no subgraph of G_L exists isomorphic to G_S.

Assume that $|V_L| = k \geq |V_S|$, where $k \in \mathbf{N}$. Let $G' = (E', V')$ be a subgraph of G_L isomorphic to G_S (if such subgraph does not exist, then we are done). Let $(B, \bar{B})$ be a partition of V_L obtained after applying Algorithm 3.1. If $E' \cap e(B, \bar{B}) = \emptyset$ then, obviously G' is a subgraph of $G_L(B)$ or $G_L(\bar{B})$ and is found by recursion. Otherwise, let $e' = (v'_1, v'_2) \in E'$ be an edge that was removed before any other edge in E'. Since $G' \cong G_S$ thereby exists an isomorphism φ which maps V' to V_S. Put $u_1 = \varphi(v'_1)$ and $u_2 = \varphi(v'_2)$, where $u_1, u_2 \in V_S$. Let w_1, w_2 be a smallest pair of vertices of G_S such that there exists an automorphism of G_S which maps u_1 to w_1 and u_2 to w_2. Then, clearly,

- there is an isomorphism between G' and G_S which maps v'_1 to w_1 and v'_2 into w_2,
- $\langle w_1, w_2 \rangle$ is not a redundant pair and consequently an induced traverse history of G_S starting in w_1, w_2 is contained in A.

Thus applying Algorithm 6.2 on v'_1 and v'_2 and on traverse history starting on w_1, w_2 we obtain all subgraphs of G_L and isomorphic to G_S, such that v_1 and v_2 mapped to w_1 and w_2, respectively. Since we assume, that e' was an edge that was removed before any other edge in E', all other edges of G' are not removed when the above algorith is applied and G' is found. $\square$

The proof for the case of induced subgraph isomorphism is quite similar.

6.3 Duplicate counting problem

Algorithm 6.2 actually finds a copy of G_S for every automorphism of G_S. We eliminate duplicates from the set of G_S instances in Procedure *SubIso* before the set is returned. This elimination is relatively easy since all we have to do is to compare vertex sets of instances.

If we aim to only obtain a count of instances rather than the instance set, we need to return the size of set $S \cup SubIso(G_L(B)) \cup SubIso(G_L(\bar{B}))$ instead of the set itself in Procedure *SubIso*. This size needs to be divided by the size of automorphism group of G_S. This size can be computed during Traverse History construction, thus adding no computational overhead to the algorithm.

7 Experimental Results

7.1 Settings

The Subsea algorithm was implemented in C++, on an Intel Pentium 4 CPU 2.4 GHz running Windows 2000 and with 512 MB of main memory.

We performed three sets of experiments: experiments on random synthetic graphs, experiments on real-life database called NCI [30] and experiments on benchmark graphs used by Cordella et al. in [27].

For the synthetic case, We generated random graphs of sizes 100 to 5000 nodes with various numbers of edges and labels to serve as database graphs. We used two types of random graphs: graphs with large diameter (which are not expander-like) and Erdös-Renyi graphs (which are expander-like, see [17]) where each edge in a graph exists with a probability p. Labels distribution was uniformly random in part of experiments, and normal in another part. We have also used unlabeled cliques, stars and lines for the experiments.

7.2 Algorithms

We compared our algorithm's results on these graphs with the results of Ullman's algorithm (see [37]) and Cordella et al. (see [27]) algorithm. Both algorithms are publicly available on the Web (see [36]). In all experiments we use the sub-algorithm of these algorithms that find all distinct subgraphs isomorphic to a given subgraph.

7.3 Detailed comparison

In all of the charts, the running time of all the algorithms is specified in seconds. Bar charts are used to present the results and names of the algorithms (Subsea, Ullman and VF2 for Cordella et al. algorithm) appear above their respective bars.

7.3.1 Present subgraphs

Here, we have compared Subsea, Ullman's and Cordella et al. algorithms on various random and real-life database graphs and subgraphs that have instances in these database graphs (sometimes, quite large number of instances). The X axis shows the number of nodes in the large graph. All randomly generated graphs have average node degree 3 or more. We use log-scale for the Y axis (running time) in order to see small running times better.

The first series of experiments refers to random database graphs with 10 and 5 labels and having uniform label distribution. Figures 5 and 6 show how the three algorithms perform when a subgraph has 15 and 100 nodes respectively and the database graph contains 10 labels. For a subgraph of 15 nodes, Ullman's and Cordella et al. algorithms did not finish in many

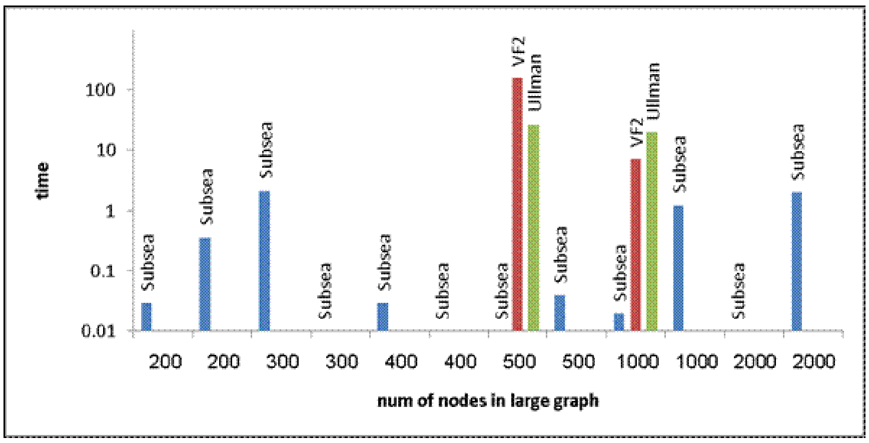

Figure 5: Subgraph with 15 nodes and 10 labels, uniform label distribution.

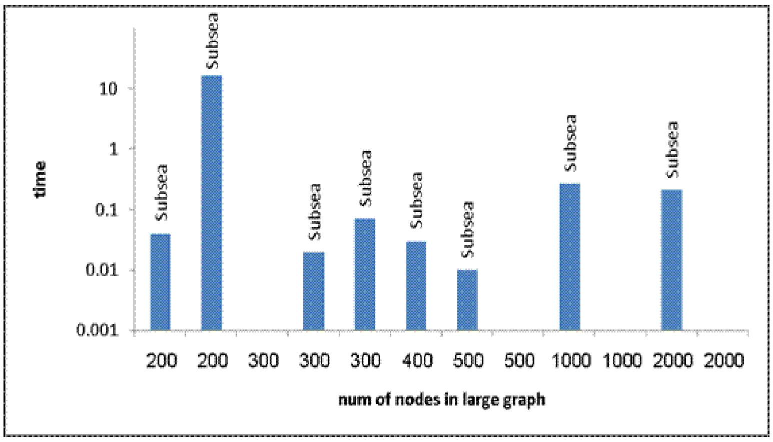

Figure 6: Subgraph with 100 nodes and 10 labels, uniform label distribution.

of the cases (denoted by missing bars). In Figure 6, we show times on Subsea algorithm only because the other two algorithms didn't finish for either database graph within 5 minutes (we counted pure application time, system time in these cases exceeded 30 minutes). We see that the Subsea algorithm has an advantage over the other two algorithms.

Figure 7 shows how the three algorithms perform when a subgraph has 15 nodes and the database graph has 5 different labels; the label distribution on both graphs is uniform. Smaller number of labels causes the number of instances to rise dramatically and the problem of finding all subgraph

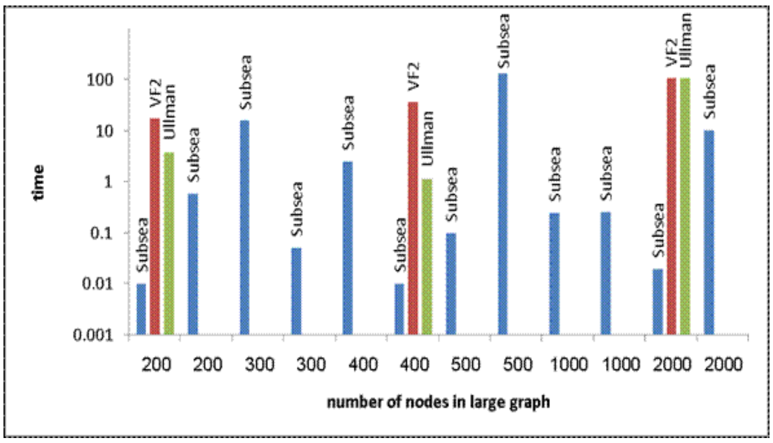

Figure 7: Subgraph on 15 nodes and 5 labels, uniform label distribution.

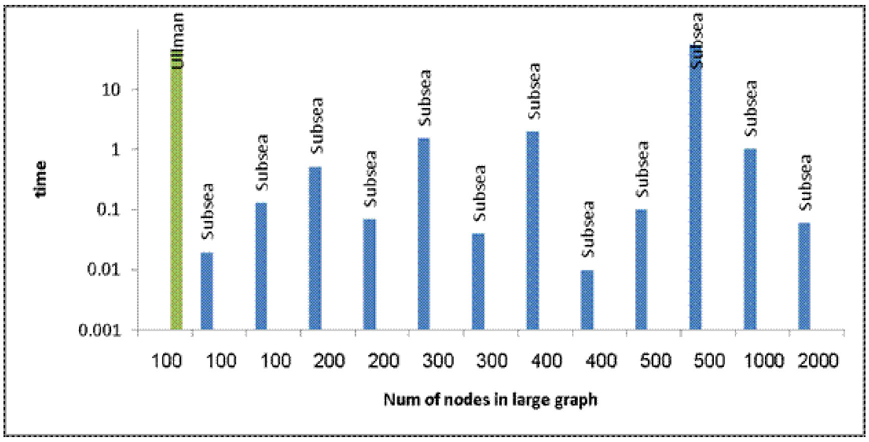

Figure 8: Subgraph of 15 nodes, normal label distribution.

instances quickly becomes unfeasible. Again, we see that Subsea algorithm performs better in these cases.

The second series of experiments was performed on labeled random Erdös-Renyi with 10 labels and normal label distribution (see Figures 8, 9 and 10). As expected, normal label distribution implies more appearances of a subgraph in the database graph and thus larger search space. We included in charts only the cases where at least one of the algorithms finishes. For smaller subgraphs (Figure 8), we see an advantage to the Subsea algorithm, but in other cases there are some subgraphs instances where

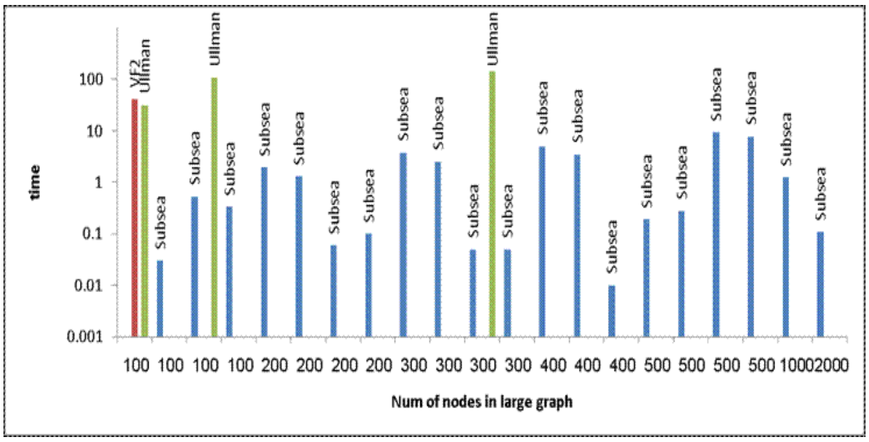

Figure 9: Subgraph of 50 nodes, normal label distribution.

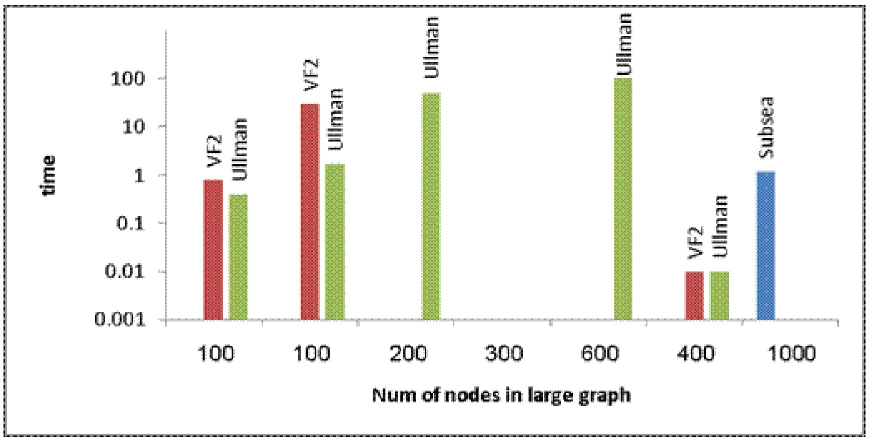

Figure 10: Subgraph of 100 nodes, normal label distribution.

Ullman's algorithm performs better.

The third series of experiments have been made on unlabeled graphs of specific structure in an attempt to understand the behavior of the Subsea algorithm. These cases, for which the number of subgraph instances can be easily found combinatorially, are in fact very hard for every algorithm tested. Double-digit running time (in seconds) occurs when the number of subgraph instances in a database graph reaches hundreds of thousands. The horizontal axis in the charts shows which small graphs were tested on the same database graph (*star* i denotes a star with i leaves, *line* i denotes a

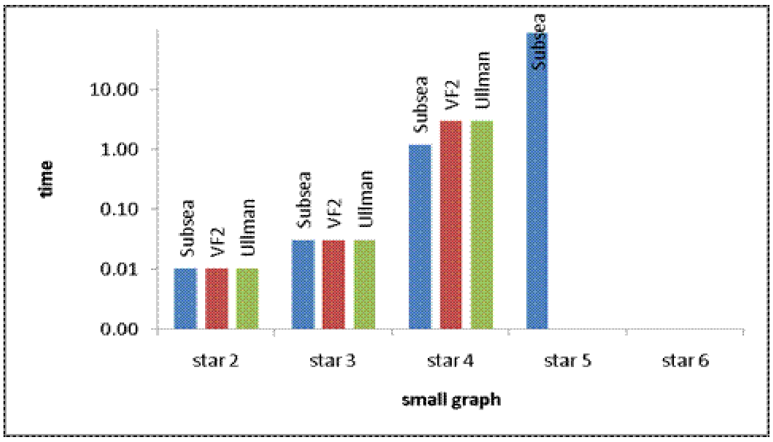

Figure 11: The database graph is an unlabeled star with 100 nodes.

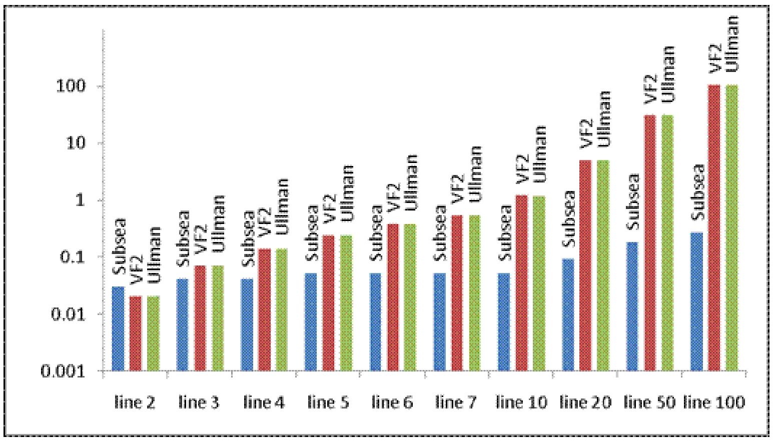

Figure 12: The database graph is an unlabeled line with 200 nodes.

simple path on i nodes, *clique* i denotes a complete graph on i nodes and *cycle* i denotes a simple cycle on i nodes). Figure 11 gives runtimes for the case when both database and subgraphs are stars, with the database graph having 100 nodes. Since Subsea algorithm uses divide-and-conquer approach based on cuts, and a well-chosen cut in the center of a star stops the recursion very quickly, we see that Subsea outperforms the other algorithms. In Figure 12, both the database and subgraphs are lines, and the database graph has 200 nodes. In this case, the Subsea recursion is deep (but not exponential), and still it performs better than other algorithms. For an

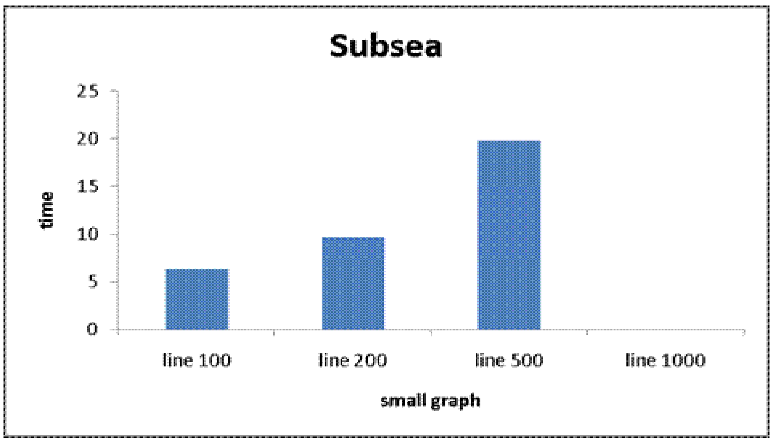

Figure 13: Unlabeled line on 2000 nodes as a large graph.

unlabeled line of 2000 nodes and larger subgraphs, both Ullman's and VF2 algorithms did not finish, while Subsea showed reasonable results (see Figure 13). Finally, Figures 14 shows how the algorithms behave when a database graph is an unlabeled complete graph and a subgraph is either a line, a cycle or a star. Complete graph is the worst case for any approach, and we see a clear dependence between the feasibility of a case and the number of nodes in a subgraph. It is not surprising that all algorithms fail when a subgraph size increases. However, since Subsea is based on finding-a-good-cut technique (which unlabeled clique definitely lacks), it performs worse than Ullman's and VF2 algorithms.

Our final series of experiments was performed on a real-life database obtained from National Cancer Institution (see [30]). We tested all three algorithms on subsets of the database graph, because the complete graph is too large; we present in Table 1 Subsea results only since other algorithms did not finish in any of the cases.

7.3.2 Absent subgraphs

We have compared all three algorithms on a series of cases where a subgraph in question is not present in the database graph. This is an important domain since in graph mining candidate subgraphs that are not present in database must be eliminated as quickly as possible. An absent subgraph detection can be also viewed as the time required to find the first instance of a subgraph in a database graph. In our charts X axis shows the number of nodes in a subgraph and the Y axis denoted the time in seconds on

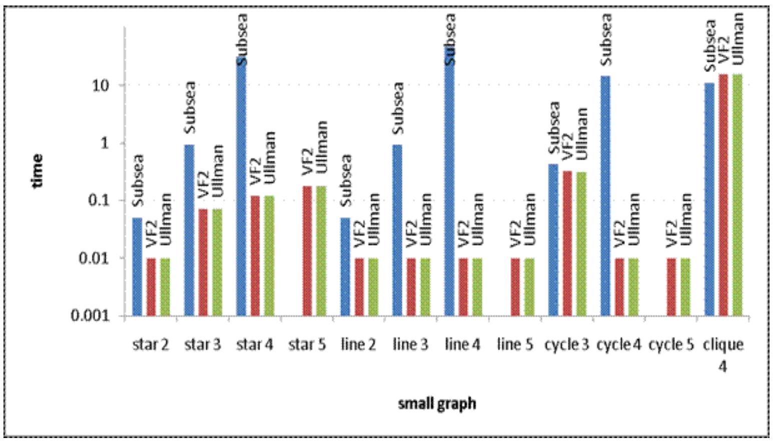

Figure 14: Complete graph on 50 nodes as a large graph.

Large graph	Small graph	Subsea time	Num of matches
532 nodes	20 nodes	0.23 secs	172
532 nodes	80 nodes	0.26 secs	696
532 nodes	50 nodes	0.22 secs	172
999 nodes	50 nodes	0.95 secs	257
999 nodes	50 nodes	0.61 secs	257
999 nodes	54 nodes	0.78 secs	257
999 nodes	41 nodes	0.86 secs	1158
999 nodes	50 nodes	0.85 secs	1157
999 nodes	50 nodes	0.69 secs	1158
1000 nodes	90 nodes	0.64 secs	290
1000 nodes	20 nodes	0.61 secs	290
1000 nodes	70 nodes	0.73 secs	1221
1000 nodes	99 nodes	0.81 secs	1221
1000 nodes	71 nodes	0.62 secs	290
2000 nodes	22 nodes	3.18 secs	642
2000 nodes	90 nodes	3.43 secs	2605

Table 1: NCI database tests.

logarithmic scale. Distribution of labels on the nodes of database graphs and small graphs in these tests is uniformly random.

Figure 15 shows the behavior of the three algorithms on large-diameter database graph with 1000 nodes and 10 labels. The chart in Figure 16 shows the behavior of the three algorithms on large-diameter database graph with 2000 nodes and 10 labels. We see that Subsea runs faster than Ullman's algorithm on these graphs, but slower than the VF2 algorithm. However, the number of cases where the latter algorithm was stuck is quite large,

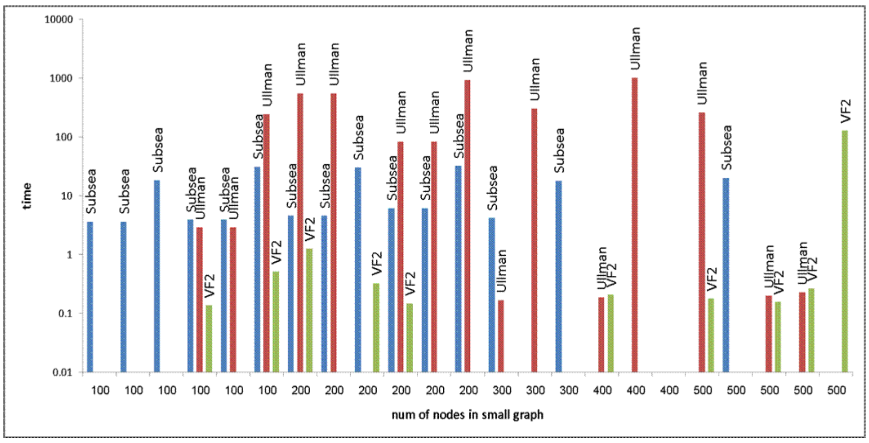

Figure 15: Database graph with 1000 nodes.

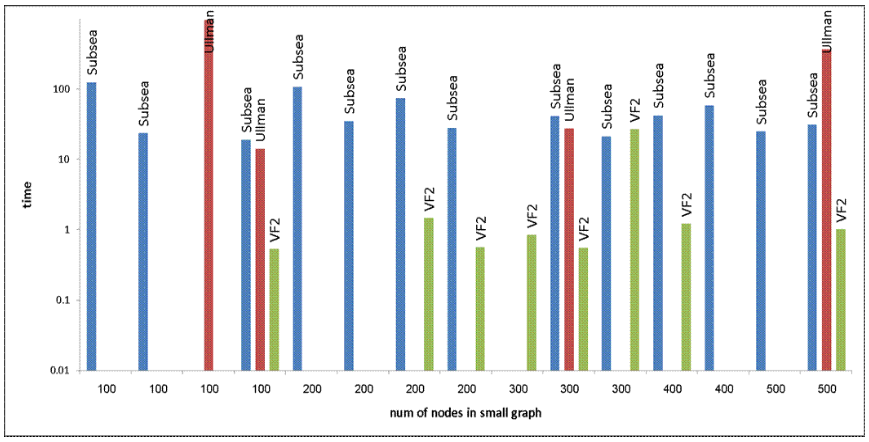

Figure 16: Database graph with 2000 nodes.

while Subsea provides an answer in most of the cases.

7.4 Benchmark graphs

For the last series of experiments we used the ARG Database (see [33]) by the Intelligent Systems and Artificial Vision Laboratory (SIVALab) of the University of Naples. We tested Subsea and VF2 algorithms on selection of unlabeled graphs from this database. We have run these experiments on cluster consisting of four Intel SMP servers with 2 Dual Core Xeon 5140 2.33GHz processors and 4G RAM. Table 2 summarizes obtained results and shows which algorithm ran faster for various types of graphs. For a random graph, η denotes the probability with which the edge is added and for a 2-dimensional mesh ρ indicates the ratio of extra edges added. Figures 17, 18 and 19 show a detailed comparison of runtimes of both algorithms on sample graphs of various size. On graph with bounded degree, VF2 runs faster than Subsea (see Figure 19).

	Random graphs			2D meshes			
Nodes	$\eta = 0.01$	$\eta = 0.05$	$\eta = 0.1$	regular	$\rho = 0.2$	$\rho = 0.4$	$\rho = 0.6$
up to 60 nodes	Subsea	Subsea	VF2	VF2	VF2	Subsea	Subsea
60 to 80 nodes	Subsea	Subsea	Subsea	VF2	Subsea	Subsea	Subsea
80 to 1000 nodes	Subsea	Subsea	Subsea	Subsea	Subsea	Subsea	Subsea

Table 2: Comparison of Subsea and VF2 on ARG random graphs and 2D meshes.

Detailed comparison shows that Subsea runs faster on randomly generated graphs and two-dimensional meshes, while VF2 shows better times on bounded degree graphs. In general, as the graph becomes larger and the number of isomorphic subgraph grows, the Subsea advantage becomes more prominent.

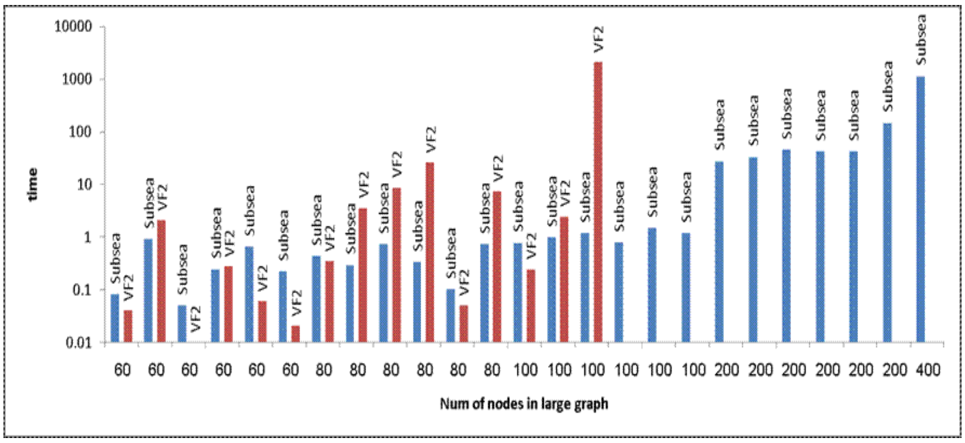

Figure 17: Randomly generated graphs with $\eta = 0.01$.

Finally, Figures 20-21 show how Subsea scales when graph density, graph size and number of labels change. In all these experiments, single small graph with 10 nodes is used. Figure 20 demonstrates how the runtime of Subsea changes when the number of edges of the large graph with 100 nodes

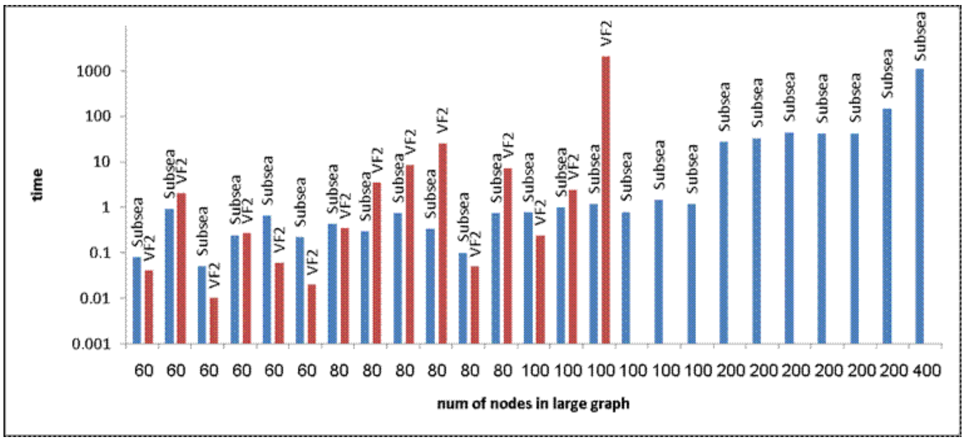

Figure 18: Randomly generated graphs with $\eta = 0.1$.

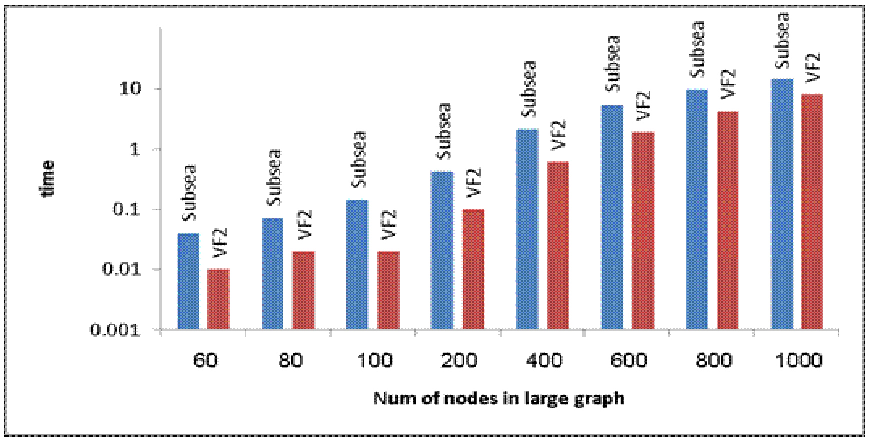

Figure 19: Bounded-valence graphs with valence 3.

increases. When large graph becomes denser, the number of isomorphisms rises to hundreds of thousands. Figure 21 shows the dependency of Subsea runtime on the large graph size when the density remains the same. Figure 22 demonstrates how Subsea scales when the number of labels increases in the same dense database graph of 100 nodes and 1200 edges. As expected, running times rise when density and graph size increase and drops dramatically when the number of labels increases.

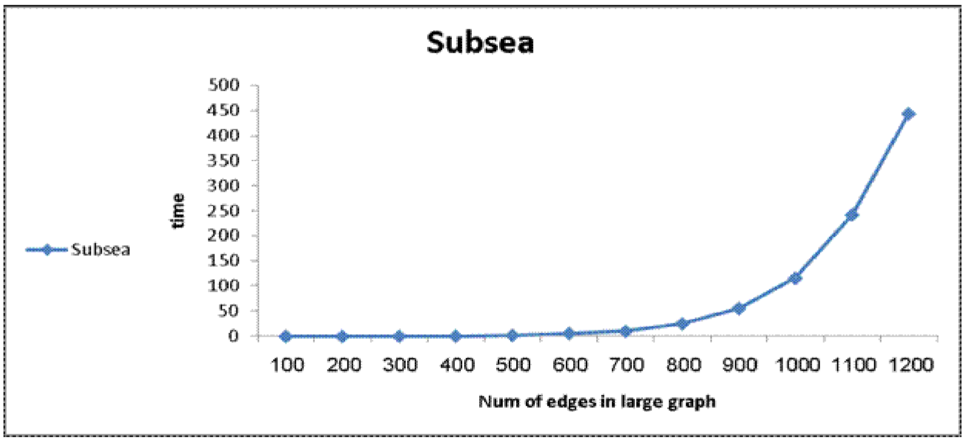

Figure 20: Subsea runtimes for database graphs of increasing density.

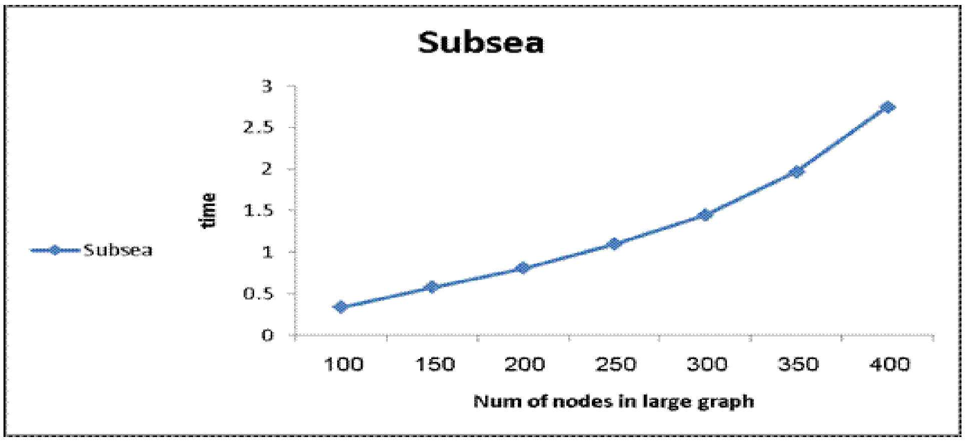

Figure 21: Subsea runtimes for database graphs of increasing size and the same density.

7.5 Summary

There is one distinct observation from all the above experiments. Subsea outperforms the other algorithm when there exist many isomorphic instances of the small graph within the database graph. We feel that the main reason is the bisection approach. Subsea starts the search only from "small" cuts, while in other algorithms the entire graph is searched for the next instance.

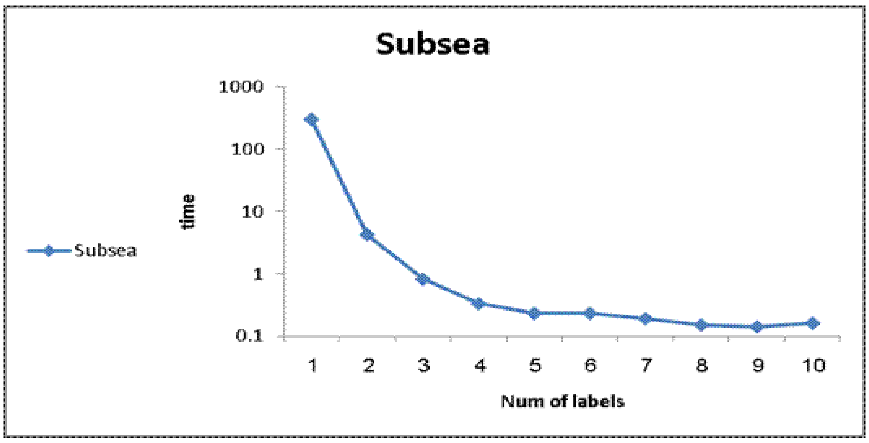

Figure 22: Subsea runtimes for database graphs of increasing label number.

8 Conclusions

We presented a new heuristic for finding all isomorphisms of a source pattern graph (or induced source graph) in a target graph. The new approach is based on a bisection algorithm and on a search strategy which is directed by an original traverse history notion. The experimental results show advantage of our approach over algorithms of [37] and [27] when all the instances of a subgraph need to be found. The conclusion from the above experiments is that Subsea is much better that the two other algorithms when multiple instances of a subgraph are searched for. The reason is that in Subsea in every bisection we test for multiple instances and never search that part of the graph later. When a single instance of a subgraph needs to be found, the algorithms of [37] and [27] have an advantage over Subsea.

The proposed algorithm has general validity, since no constraints are imposed on the topology of the pattern and the target graphs, and the method can be easily be extended to the case of directed graphs as well. An extension to the case of *edge-labeled* graphs is less trivial and will require some effort.

Another advantage of our algorithm is that the bisection procedure leads itself naturally to parallel computation. We plan to pursue this topic in the future. An important goal is to enable current researchers to compare their codes with each other, in hopes of identifying the more effective of the recent algorithmic innovations that have been proposed. Although many implementations do not include the most sophisticated speed-up tricks and thus may not be able to compete on speed with highly tuned alternatives, we may be able to gain insight into the effectiveness of various algorithmic ideas by comparing codes with similar levels of internal optimization. It will also be interesting to compare the run times of the best current optimization codes with those of the more complicated heuristics on instances that both can handle.

9 Acknowledgments

We thank anonymous reviewers for their useful comments on our paper. We thank Lior Becker and Arie Ohana for doing extraordinary job implementing the Subsea algorithm and making it even more efficient than we anticipated. We also thank Rosa Shlayzer for conducting the experiments. We thank Fraenkel Center for Computer Science and Paul Ivanier Center for Robotics and Production Management for partially supporting this work.

References

[1] F. A. Akinniyi, A. K. C. Wong, and D. A. Stacey. A new algorithm for graph monomorphism based on the projections of the product graph. *Trans. Systems, Man and Cybernetics*, SMC-16:740–751, 1986.

[2] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Electronic Colloquium on Computational Complexity (ECCC)*, 1(009), 1994. Full paper appears in J.ACM 42:4, July 1995, 844-856.

[3] R. Ambauen, S. Fischer, and H. Bunke. Graph edit distance with node splitting and merging. *IAPR-TC15 WKSP on Graph-based Representation in Pattern Recognition*, pages 95–106, 2003.

[4] H Bunke B. T. Messmer. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Trans. Knowl. Data Eng.*, 12:307–323, 2000.

[5] G. V. Batz. An optimization technique for subgraph matching strategies. Technical Report 2006-7, Universitat Karlsruhe, Faculty of Informatik, 2006.

[6] S. Berretti, A. Del Bimbo, and P. Pala. A Graph Edit Distance Based on Node Merging. In *Proc. CIVR2004*, pages 464–472, 2004.

[7] M. Boeres, C. Ribeiro, and I. Bloch. A randomized heuristic for scene recognition by graph matching. In *Proc. of WEA 2004*, pages 100–113, 2004.

[8] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Mining hidden community in heterogeneous social networks. In *Proceedings of the 3rd international workshop on Link discovery*, pages 58–65, 2005.

[9] P. Champin and C. Solnon. Measuring the similarity of labeled graphs. *Conference on Case-Based Reasoning (ICCBR)*, pages 100–113, 2003.

[10] M. S. Chen, J. S. Park, and P. S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.

[11] J. K. Cheng and T. S. Huang. A subgraph isomorphism algorithm using resolution. *Pattern Recognition*, 13:371–379, 1981.

[12] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings of ICIAP 1999*, pages 1172–1177, 1999.

[13] J. Cortadella and G. Valiente. A relational view of subgraph isomorphism. In *Proc. Fifth Int. Seminar on Relational Methods in Computer Science*, pages 45–54, 2000.

[14] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Da ta Mining (KDD-98)*, pages 30–36, 1998.

[15] A. Dessmark, A. Lingas, and A. Proskurowski. Faster algorithms for subgraph isomorphism of k-connected partial k-trees. *Algorithmica*, 27(3):337–347, 2000.

[16] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms & Applications*, 3(3):1–27, 1999.

[17] P. Erdoös and A. Renyi. On Random Graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959.

[18] P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations*, Lecture Notes in Computer Science. Ischia, 2001.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[20] E. Gudes, S. E. Shimony, and N. Vanetik. Discovering Frequent Graph Patterns Using Disjoint Paths. *IEEE Trans. Knowl. Data Eng*, 18(11):1441–1456, 2006.

[21] E. B. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software Practice and Experience*, 34(6):591–607, 2004.

[22] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of ICDM 2001*, 2001.

[23] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *Proceedings of SDM 2004*, 2004.

[24] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In *Proc. Joint APPLIGRAPH/GETGRATS Worksh. Graph Transformation Systems*, pages 189–196, 2000.

[25] X. Lin, Ch. Liu, Y. Zhang, and X. Zhou. Efficiently computing frequent tree-like topology patterns in a web environment. In *Proceedings of 31st Int. Conf. on Tech. of Object-Oriented Languages and Systems*, 1998.

[26] A. Lingas and M. M. Sysło. A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs. Technical Report LiTH-IDA-R-89-05, Linköping Univ., Dept. Computer and Information Science, 1989.

[27] Carlo Sansone Luigi P. Cordella, Pasquale Foggia and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[28] D. W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Ann. Discrete Math.*, 2:91–106, 1978.

[29] B. T. Messmer and H. Bunke. Subgraph isomorphism detection in polynomial time on preprocessed model graphs. In *Proc. 2nd Asian Conf. Computer Vision*, pages 373–382. Springer-Verlag, 1996.

[30] NCI. Database of interacting proteins. Technical report, 2008. Available at http://dtp.nci.nih.gov.

[31] Siegfried Nijssen and Joost N. Kok. Frequent graph mining and its application to molecular databases. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC 2004, Den Haag, Netherlands, October 10-13, 2004*. IEEE Press, 2004.

[32] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In Ronny Kohavi, Johannes Gehrke, William DuMouchel, and Joydeep Ghosh, editors, *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD2004, Seattle, USA, August 22-25, 2004*, pages 647–652. ACM Press, 2004.

[33] SIVALab of the University of Naples. Arg graph database. Technical report, 2003. Available at http://amalfi.dis.unina.it/graph/.

[34] X. Pennec and N. Ayache. A geometric algorithm to find small but highly similar 3d substructure s in proteins. *Bioinformatics*, 14(6):516–522, 1998.

[35] O. Sammoud, C. Solnon, and K. Ghdira. Ant algorithm for the graph matching problem. In *Proceedings of EvoCOP 2005*, pages 213–223, 2005.

[36] C. Sansone. Graph database library for subgraph isomorphism. Technical report, 2001. Available at http://amalfi.dis.unina.it/graph/.

[37] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.*, 23:31–42, 1976.

[38] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proceedings of ICDM 2002*, pages 458–465, 2002.

[39] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *Proceedings of SIGIR 1998*, pages 146–154, 1998.

[40] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of ICDM 2002*, pages 721–724, 2002.

[41] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of EDBT Conference, 2009*, 2009.