Rheinische
Friedrich-Wilhelms-Universität Bonn

Mathematisch-naturwissenschaftliche
fakultät

Master thesis

Frequent Subtree Mining From Locally Easy Graphs

Author:
Yakun Li

First Examiner:
Prof. Dr. Stefan Wrobel

Second Examiner:
Prof. Dr. Sören Auer

Advisor:
Pascal Welke

Submitted: December 14, 2016

Declaration of Authorship

I declare that the work presented here is original and the result of my own investigations. Formulations and ideas taken from other sources are cited as such. It has not been submitted, either in part or whole, for a degree at this or any other university.

Location, Date

Signature

Acknowledgements

First, I wish to express my sincere thanks to my thesis supervisor Pascal Welke. He is always patient with my questions, and responds to me timely. It would be very difficult to finish this thesis without his supervision.
I further appreciate that Prof. Dr. Stefan Wrobel and Prof. Dr. Sören Auer respectively provide me the opportunity to write thesis and work in their research group.
I also would like to thank Meilin Li, Meemansa Sood, Rania Briq and Vikramjit Singh for helping me with my thesis writing questions.

Abstract

In this thesis, we evaluated a levelwise and a depth first search algorithm to solve the frequent subtree mining problem. In order to solve the frequent subtree mining problem, we first evaluated a subtree isomorphism algorithm and an iterative version of it to solve the subtree isomorphism problem. The subtree isomorphism algorithm that we evaluated can decide subtree isomorphism in polynomial time if the input text graphs are locally easy. This positive result combined with the properties of subtrees implies that we can solve the frequent subtree mining problem on locally easy graphs in polynomial delay.

The easinesses of input text graphs influence the runtime of the algorithms that we evaluated. We developed some experiments to study the orders of polynomial of the runtime of the non iterative subtree isomorphism algorithm and the levelwise frequent subtree mining algorithm based on different easinesses of the input text graphs. Our evaluation results show that the non iterative subtree isomorphism algorithm has much lower order polynomial runtime on real datasets than the worst case. The levelwise frequent subtree mining algorithm has first order polynomial runtime on our test datasets. The runtimes of the binary and prefix tree search candidate elimination methods are evaluated based on different input text graphs. These two candidate elimination methods perform almost equally on some datasets even though prefix tree search has lower runtime complexity. In oder to construct synthetic graphs with a certain complexity for algorithm evaluation, we proposed one method to generate graphs with n^2 spanning trees, where n denotes the number of vertices of graphs.

Contents

List of Figures

List of Algorithms

1 Introduction

1.1 Motivation

The success of pharmaceutical research relies significantly on the continuously emerging information technologies. Drug discovery is one important field in pharmaceutical research which includes discovering some chemical entities to relieve some diseases. Target discovery plays an important role during drug discovery, and it has significant influence to the successful development of drugs (Butcher, 2003; Lindsay, 2003; Sams-Dodd, 2005). Molecular entities are one type of target in drug discovery (Yang et al., 2009), and molecule databases are usually large. It requires a lot of human effort or even could be impossible to manually find potential targets. Moreover, this process costs a huge amount of money. Data mining, which is the task of discovering hidden patterns from large datasets, is one of the information technologies which can speed up the target discovery process. Data mining plays a crucial role in Knowledge Discovery in Databases (KDD), which is the process of discovering knowledge from databases, as shown in figure 1.1.

Frequent Connected Subgraph Mining (FCSM) is one specific type of data mining which can be used to discover frequent patterns from graph structure data. For example, molecules can be represented in graph structures, and there are usually a huge number of patterns in graph databases. In order to reduce the number of

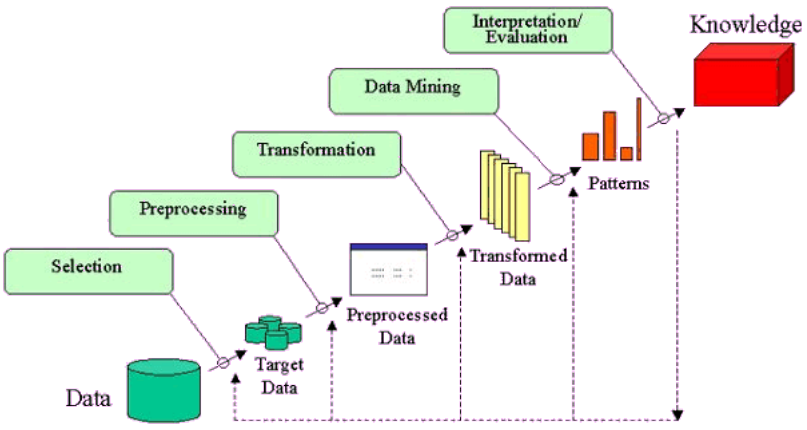


Figure 1.1: knowledge discovery in databases (*KDD diagram*)

patterns to be enumerated, we specifically focus on tree structure patterns in this thesis. Then this leads to Frequent Subtree Mining (FSM). FSM is not only applicable to drug target discovery, it can also be applied to frequent pattern mining form social network datasets, RDF graph datasets and many other graph structure datasets.

1.2 Formulization

The FCSM problem is defined in definition 1.2.1. Unfortunately, the FCSM problem cannot be solved in output polynomial time for arbitrary graph classes unless $P = NP$ (Horváth et al., 2007).

Definition 1.2.1. ***Frequent Connected Subgraph Mining (FCSM) problem**: Given a finite set $D \in \gamma$ for some graph class γ and an integer frequency threshold $t > 0$, list all graphs $p \in \zeta$ for some graph class ζ, called the pattern class, that are subgraph isomorphic to at least t graphs in D, and the patterns in the output must be pairwise non-isomorphic (Pascal Welke, 2015a).*

However, if the input text graphs are trees, then the FCSM problem can be solved in polynomial time. For example, the input text graphs are restricted to forests in (Chi et al., 2003). In this thesis, we restrict the pattern graphs to be trees and the input text graphs to be *locally easy* (see definition 1.2.2). These two conditions reduce the frequent subgraph mining problem to the problem of frequent subtree mining from locally easy graphs.

Definition 1.2.2. *A graph G with order n is locally easy if for any vertex $v \in V(G)$, the union of the biconnected components contains v has at most poly(n) spanning trees (Pascal Welke, 2015b).*

In order to compute frequent subtrees from locally easy graphs, we first need to solve the *subtree isomorphism* problem which is defined in definition 1.2.3. The subtree isomorphism problem is NP-complete for arbitrary graphs which generalizes the Hamiltonian path problem (Garey and Johnson, 1979). The subtree isomorphism problem is in P if we restrict the input text graphs to be trees (Shamir and Tsur, 1999). This positive result combined with the fact that we can generate the spanning trees of a graph in *polynomial delay* (Read and Tarjan, 1975) tells us that the subtree isomorphism problem is also in P if the text graph has polynomially many spanning trees. It is clear that locally easy graphs contain a relaxed constraint compared to graphs containing polynomially spanning trees. One example graph is given in figure 1.2 which is locally easy but has exponentially many

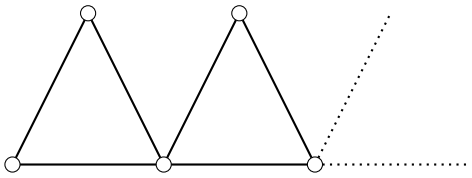


Figure 1.2: locally easy, but has $3^{\lceil \frac{n}{2} \rceil}$ spanning trees

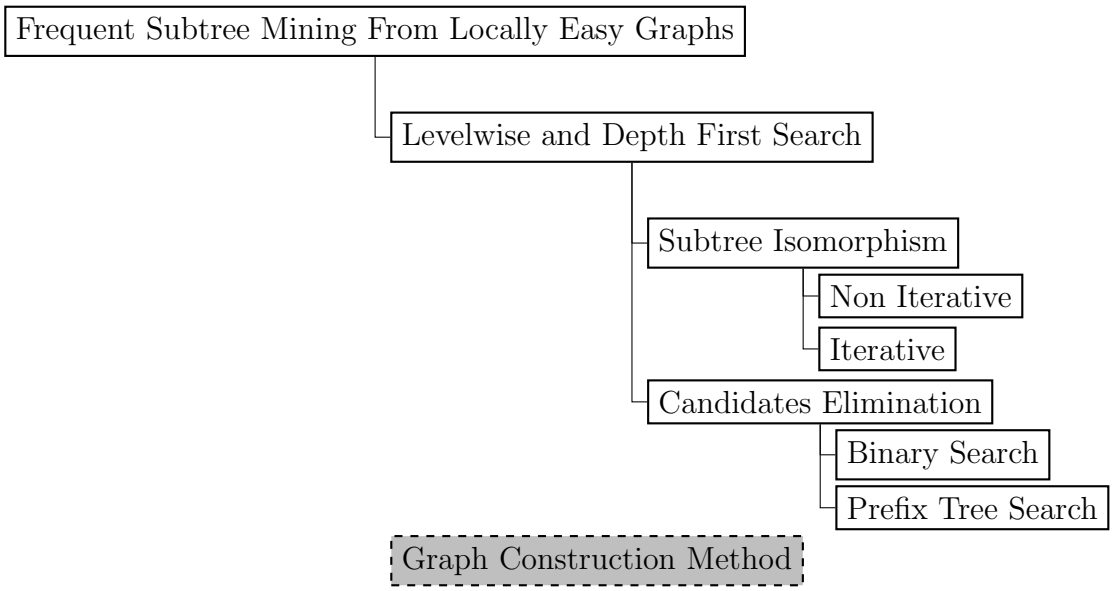


Figure 1.3: algorithms overview

spanning trees. One algorithm given in section 3.2 decides subtree isomorphism on locally easy graphs in polynomial time.

Definition 1.2.3. ***Subtree Isomorphism Problem****: Given a tree H (the pattern) and a graph G (the text graph), decide if there exists a subgraph isomorphism from H to G (Pascal Welke, 2015b).*

A levelwise and a depth first search frequent subtree mining algorithm are given in section 3.1 which compute frequent subtrees from locally easy graphs in polynomial delay by using the subtree isomorphism algorithms given in section 3.2.

1.3 Thesis Work and Research Result

In this thesis, we implemented and evaluated the algorithms mentioned in section 1.2. An overview of these algorithms is given in figure 1.3 which contains a levelwise (Pascal Welke, 2015a) and a depth first search frequent subtree mining algorithm. Levelwise search filters out more infrequent candidates but depth first search uses

less memory. Our evaluation results reveal this theoretical property. The frequent subtree mining algorithms invoke a subtree isomorphism algorithm (Pascal Welke, 2015b) or an iterative version (Welke, 2016) of it to decide subtree isomorphism. The iterative subtree isomorphism algorithm iteratively uses the already computed information of a frequent subtree $H - e$ and text graph G to decide whether the tree H is subtree isomorphism to text graph G or not. This speeds up the frequent subtree mining process in general, but it is not always faster than the non iterative version as it needs to compute all the information related to $H - e$ and text graph G. The details are given in section 3.3, and our evaluation results reveal this property. The frequent subtree mining algorithms use binary or prefix tree search candidates elimination algorithm as a subroutine. Prefix tree search algorithm has a lower runtime complexity than binary search, but our evaluation results show that they preform almost equally on some real datasets. We collected some statistical results from two real datasets, and the results show that our evaluated subtree isomorphism algorithms theoretically have lower orders of polynomial on most of the real graphs. This also implies that our evaluated frequent subtree mining algorithms have a lower complexity on most of the real graphs. We later studied the different orders of polynomial of the runtime of the levelwise frequent subtree mining algorithm and the non iterative subtree isomorphism algorithm based on different synthetic graph datasets. Our evaluation results show that the evaluated algorithms have much lower order of polynomial than the theoretically worst case runtime. The reason that we are using synthetic datasets is that graphs in real datasets do not always belong to a certain complexity. In order to construct graphs with a certain complexity, we proposed one method to generate graphs with exactly n^2 spanning trees.

1.4 Thesis Structure

The thesis contents are structured as follows. We first introduce the preliminaries on graphs in section 2. Then the theoretical background and implementation are presented in section 3. Some experiments are developed in section 4 to evaluate the algorithms given in section 3. In the end, in section 5, we conclude and propose some future works for further improvements.

2 Preliminaries on Graphs

2.1 Notions and Notations

We use the same notions and notations as in (Pascal Welke, 2015a). A graph G denotes an undirected graph, $V(G)$ denotes the vertices set of graph G and $E(G)$ denotes the edges set of graph G. For any vertex $v \in V(G)$, $u \in V(G)$, uv denotes an edge connecting u and v, and $\delta(v)$ denotes the set of neighbors of v. $G[V']$ denotes the subgraph of G induced by set of vertices in V', where $V' \subseteq V(G)$. The definition of *complete graph* is given in definition 2.1.1.

Definition 2.1.1. *A graph G is **complete** if all the vertices of G are pairwise adjacent, and a complete graph of n vertices is denoted as K^n (*Graph Theory*).*

2.2 Canonical Form and Canonical String

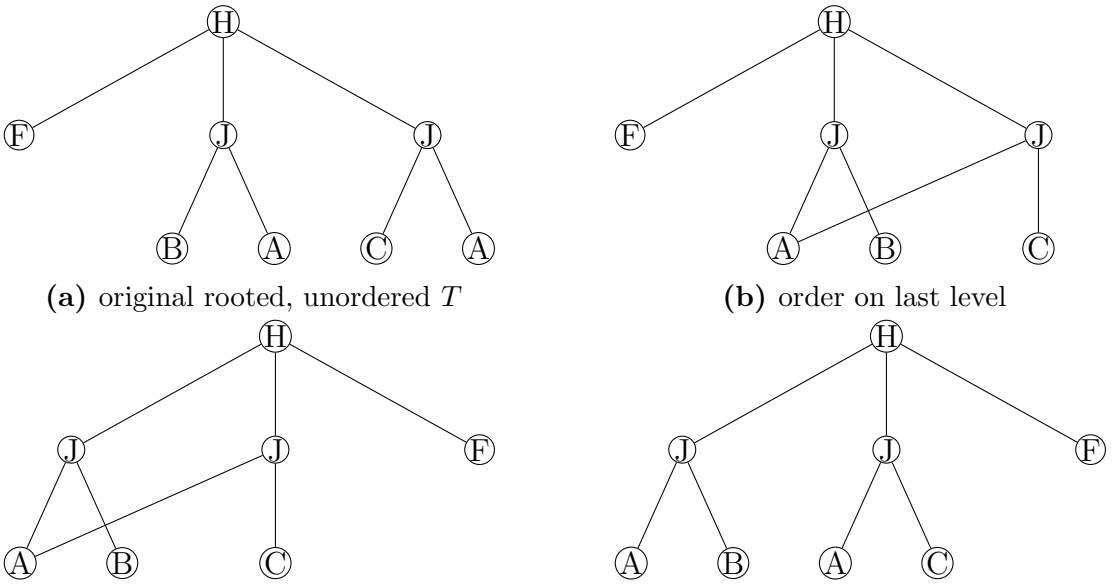
Canonical string is used intensively during our evaluation of the frequent subtree mining algorithms to remove duplicated candidates. It is an easier representation of canonical form, and canonical form is a unique form of rooted trees.

2.2.1 Canonical Form

One method is given in (Chi et al., 2003) to generate canonical form for *free trees*. *Free trees* are connected, acyclic and undirected graphs. Some examples are given in (Chi et al., 2003), like *evolutionary tree* (Hein et al., 1996), *shape axis tree* (Liu and Geiger, 1999) and *multicast tree* (Cui et al., 2005). Canonical form is in the same equivalence class as the relation defining the equivalence of tree isomorphism (Chi et al., 2003) given in definition 2.2.1.

Definition 2.2.1. *Two labeled trees T_1 and T_2 are isomorphic to each other if there is a one-to-one mapping from the vertices of T_1 to the vertices of T_2 that preserves vertex labels, edges labels, and adjacency (Chi et al., 2003).*

There are different ways to generate canonical form, and figure 2.1 demonstrates the method used in (Chi et al., 2003). This method is based on a tree isomorphism



(a) original rooted, unordered T **(b)** order on last level

(c) padding **(d)** canonical form

Figure 2.1: normalizing a rooted unordered tree T to it's canonical form

algorithm given in (Aho and Hopcroft, 1974). In the example, the label of an edge is combined with it's child vertex, then all edges are assumed to be identical. Figure 2.1a contains the original labeled, rooted and unordered tree T, then it is normalized at each level by using a bottom-up approach. In figure 2.1b, the vertices are ordered by their label, and the subtrees are padded in figure 2.1c. The canonical form representation is in figure 2.1d.

2.2.2 Canonical String

Canonical string representation is an easier (and equivalent) representation of canonical form representation. There are different ways to get canonical string from canonical form representation, and one example presented in (Zaki, 2005) uses a depth first search to get the unique string representations of rooted and ordered trees (Chi et al., 2003). We use the implementation from (Welke, 2014), and it uses a depth first traversal of the canonical form of the rooted tree T to obtain canonical string. Assuming all edges have label "1", and "(", ")" are not in the labels of any vertices and edges. "(" denotes searching by one level, and ")" denotes one backtrack. Then the canonical string representation of the canonical form in figure 2.1d by depth first traversal is *H(1 J(1 A)(1 B))(1 J(1 A)(1 C))(1 F)*.

3 Theoretical Background and Implementation

In this section, we first introduce the levelwise and depth first search frequent subtree mining algorithms. Then we introduce the subtree isomorphism algorithm and the iterative version of it. Afterwards, an initialization algorithm is given at section 3.4 and our graph construction method is given at section 3.5. The graph construction method is used during experimental evaluation to generate graphs with a certain complexity. Corresponding implementation is introduced after the presentation of each algorithm.

3.1 Levelwise and Depth First Search Frequent Subtree Mining for General Graphs

We present algorithms in this section to solve the FCSM problem. According to definition 1.2.1, we run into the listing problem to list the frequent connected subgraphs from an input graph set D. Denote an algorithm which solves the listing problem as $\mathfrak{A}$, and it outputs the frequent pattern set $O = [p_1, p_2, ..., p_n]$ (Pascal Welke, 2015a). According to (Pascal Welke, 2015a) and (Johnson et al., 1988), we can classify the complexity of the listing algorithms as follows:

1. *Output polynomial time (polynomial total time)*, if the time to output all frequent patterns in O is bounded by a polynomial of combined $size(O)$ and $size(D)$.

2. *Incremental polynomial time*, if the algorithm outputs p_1 in time bounded by a polynomial of $size(D)$, the time between outputting p_i and p_{i+1} is bounded by a polynomial of $size(D) + \sum_{j=1}^{i} size(p_j)$, and the time between the output of p_n and termination is bounded by a polynomial of $size(D) + size(O)$.

3. *Polynomial delay*, if the time before the output of p_1, between the output of consecutive elements of O, and between the output of p_n and the termination of $\mathfrak{A}$ is bounded by a polynomial of $size(D)$.

We cannot solve the FCSM problem in output polynomial time for general graphs unless $P = NP$ (Horváth et al., 2007). The delay of an output polynomial time algorithm may be exponential in $size(D)$ even before the output of the first element (Horváth and Ramon, 2008), and polynomial delay implies output polynomial time.

3.1.1 Levelwise Frquent Subtree Mining for General Graphs

The algorithm 1 (Pascal Welke, 2015a) is a levelwise search (Agrawal et al., 1996) frequent subtree mining algorithm for general graphs. A levelwise search mining algorithm has to enumerate all the frequent subtrees up to level l in order to list frequent subtrees in level $l + 1$.

Algorithm 1: A generic levelwise graph mining algorithm

Data: $D \subseteq \gamma$ for some graph class γ, a pattern class ζ, and an integer $t > 0$
Result: all frequent subgraphs of D that are in ζ
`MAIN()`
1 let $S_0 \in \zeta$ be the set of frequent pattern graphs consisting of a single vertex
2 **for** $(l := 0;\ S_l \neq \emptyset;\ l := l + 1)$ **do**
3 set $S_{l+1} := \emptyset$ and $C_{l+1} := \emptyset$
4 **for** ***all*** $P \in S_l$ **do**
5 print P
6 **for** ***all*** $H \in \rho(P) \cap \zeta$ *satisfying* $H \notin C_{l+1}$ **do**
7 add H to C_{l+1}
8 **if** `SUPPORTCOUNT(`H, D`)` $\geq t$ **then**
9 add H to S_{l+1}
Function `SUPPORTCOUNT(`H, D`)`
10 counter := 0
11 **for** ***all*** G *in* D **do**
12 **if** $H \preccurlyeq G$ **then**
13 counter := counter + 1
14 **return** counter;

Algorithm 1 is based on a modification of algorithm FCSM from (Horváth and Ramon, 2008). It discoveries the frequent subgraphs in class ζ from graphs in class γ. It takes the graph set D, pattern class ζ and frequency threshold $t > 0$ as input parameters, then it outputs all frequent subgraphs of D in pattern class ζ. The first line of algorithm 1 gets all the frequent vertices from D. Then it starts the levelwise mining from line 2, and it assigns graph set S_{l+1} and C_{l+1} to empty at line 3 to keep frequent subgraphs and possible frequent candidates. It

starts to iterate all the frequent subgraphs in S_l from line 4. $\rho(P)$ outputs all the extensible graphs from current graph P by a *node refinement* (Nijssen and Kok, 2004). A *node refinement* extends current graph P at each vertex v by all possible edges. Then the algorithm adds the extended graphs to C_{l+1} at line 7 if they are belong to pattern graph class ζ and not duplicated with any candidates already generated in C_{l+1}. Algorithm 1 adds the extended graph H to S_{l+1} at line 9 if it has a frequency support larger than equal to t. The `SUPPORTCOUNT` function checks the number of transaction graphs in D that pattern H is subtree isomorphic to.

Correctness and Runtime

Theorem 3.1.1. *Let γ and ζ be the transaction and pattern graph classes satisfying the following 4 conditions:*

1. *All graphs in ζ are connected, and ζ is closed downwards under taking subgraphs, i.e., for all $H \in \zeta$ and for all connected graphs H' we have $H' \in \zeta$ whenever $H' \preccurlyeq H$.*

2. *The membership problem for ζ can be decided efficiently, i.e., for any graph H it can be decided in polynomial time if $H \in \zeta$.*

3. *Subgraph isomorphism in ζ can be decided efficiently, i.e., for all $H_1, H_2 \in \zeta$, it can be decided in polynomial time if $H_1 \preccurlyeq H_2$.*

4. *Subgraph isomorphism between patterns and transactions can be decided efficiently, i.e., for all $H \in \zeta$ and $G \in \gamma$, it can be decided in polynomial time if $H \preccurlyeq G$.*

Then the FCSM problem can be solved with polynomial delay for ζ and for all finite subsets $D \in \gamma$ (Pascal Welke, 2015a).

Correctness The correctness of algorithm 1 is proved in (Pascal Welke, 2015a) under the above condition 1. The idea of soundness is that all the patterns added to S are frequent, pairwise non isomorphic and belong to pattern class ζ. These are checked at line 6 and 8 of the algorithm 1. The idea of completeness is that a pattern $H \in \zeta$ will be outputted at line 1 or 8 only if it is frequent. All the subgraphs of H are generated if H is frequent, because frequent patterns are connected and closed downwards.

Runtime The algorithm 1 runs in polynomial delay under the above conditions 1-4 (Pascal Welke, 2015a). The general ideas include:

1. The time between starting and outputting the first frequent pattern is linear in $size(D)$ by scanning each text graph $G \in D$.
2. The time between printing consecutive patterns, and between printing the last pattern and termination is polynomial in $size(D)$.
   a) $\rho(P)$ is polynomial in the size of P, so $\rho(P)$ is polynomial in $size(D)$. Therefore $\rho(P) \cup \zeta$ can be computed in polynomial time under condition 2.
   b) $H \notin C_{l+1}$ can be checked in linear time according to condition 3 by using canonical string representation of H and a prefix tree of canonical strings of patterns.
   c) Under condition 4, line 8 of algorithm 1 can be checked in polynomial time as it can be decided in polynomial time if $H \preccurlyeq G$ for each $G \in D$.

A subtree isomorphism algorithm is given in section 3.2 which decides subtree isomorphism in polynomial time for locally easy text graphs.

3.1.2 Depth First Search Frequent Subtree Mining for General Graphs

The algorithm 2 is a general depth first search frequent subtree mining algorithm. It computes all the frequent vertices from graph database D at line 1. It sets C_{all} to empty to keep the canonical strings of all the generated frequent patterns. It calls function `DP` at line 3, and the function `DP` sets C to empty at line 5 to keep possible frequent candidates at line 8. The function `DP` sets S_{next} to empty at line 6 to keep frequent patterns at line 10. It is required at line 7 of algorithm 2 that the extended graph H should be in graph class ζ. H should not be isomorphic to any already generated candidates in C and it should not be isomorphic to any of the already generated frequent patterns in S_{all}. The frequency of candidate H is calculated at line 9 by function `SUPPORTCOUNT`. The function `SUPPORTCOUNT` checks subgraph isomorphism between pattern H and each graph G contained in D and returns the frequency.

Correctness and Runtime

The proof of correctness and runtime of algorithm 2 is similar to the proof of correctness and runtime of algorithm 1 in (Pascal Welke, 2015a). The difference is

Algorithm 2: A depth first search frequent subgraph mining algorithm

Data: $D \subseteq \gamma$ for some graph class γ, a pattern class ζ, and an integer $t > 0$
Result: all frequent subgraphs of D that are in ζ

```
MAIN()
1   let S_0 ∈ ζ be the set of frequent pattern graphs consisting of a single
      vertex
2   let S_all = ∅
3   DP(S_0, S_all, t, ζ, D)
Function DP(S, S_all, t, ζ, D)
4   for all (P ∈ S) do
5       let C = ∅
6       let S_next = ∅
7       for all H ∈ ρ(P) ∩ ζ satisfying H ∉ C and H ∉ S_all do
8           add H to C
9           if SUPPORTCOUNT(H, D) ≥ t then
10              add H to S_next
11              add H to S_all
12      DP(S_next, S_all, t, ζ, D)
Function SUPPORTCOUNT(H, D)
13  counter := 0
14  for all G in D do
15      if H ≼ G then
16          counter := counter + 1
17  return counter;
```

that we need to keep all the generated frequent patterns in S_{all} to guarantee that there is no pairwise isomorphism in the output.

Correctness The algorithm 2 is sound because it adds H to S_{next} only when it is frequent, in pattern class ζ, not in C and S_{all}. These conditions are checked at line 7 and 9 of algorithm 2. Furthermore, S_{all} contains all the generated frequent patterns which is guaranteed at line 11 of the algorithm 2. The algorithm is complete because if a pattern $H \in \zeta$ is frequent, then all the subgraphs of H are already outputted under the condition 1 in theorem 14.

Runtime We keep the canonical string representations of the generated frequent trees in C and S_{all}. $H \notin C$ and $H \notin S_{all}$ both can be checked in linear time according to condition 3 in theorem 14. The runtime analysis of other parts is same as the analysis of the algorithm 1.

3.1.3 Implementation Details

We get the input graph database D at the beginning of the implementation for both algorithm 1 and 2. Then we obtain the frequent vertices and edges in D by scanning the database once. Afterwards we start the generation of frequent subtrees with order 3 from frequent edges with order 2. The steps of the algorithm 1 and 2 are clear. In this section, we mainly present the generation of candidates and elimination of non frequent candidates in our refinement operator.

Refinement Operator

Our refinement operator is summarized in algorithm 3, it gets frequent tree set gH in level l, frequent edges set es and frequency threshold integer t as input parameters. It sets cH to empty at line 1 to keep possibly frequent candidates. Algorithm 3 iterates through each frequent tree $H \in gH$ at line 2. For each vertex $v \in V(H)$, it iterates through each frequent edge $e \in es$. If vertex v is incident with e, then it generates H' from H by extending e at vertex $v \in V(H)$ at line 6, like the example shown in figure 3.1. Afterwards, it removes each leaf from the tree H' to get a list of subtrees H'' at line 7 and checks whether all of them are presented in gH or not at line 8, like the example shown in figure 3.2. If any subtree in H'' is not presented in gH, then we know that graph H' cannot be frequent. This property cannot be applied to algorithm 2, as it may not keep all the frequent trees in gH with order l during checking. If subtrees in H'' are all contained in gH and $H' \notin cH$, then we can calculate the intersection set I of their

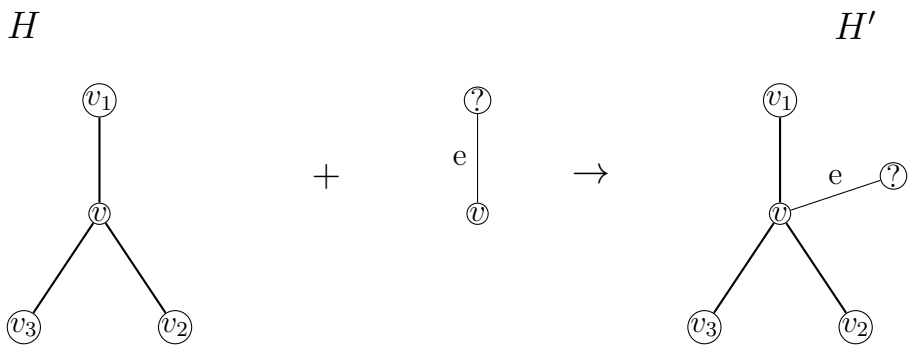


Figure 3.1: extend H

support text graph sets, like the example shown in figure 3.2. If $|I|$ is smaller than the threshold t, then H' cannot be frequent. For the algorithm 2, the intersection support set I is just the support set of frequent tree H.

Algorithm 3: Refinement operator

Data: frequent graphs gH from level l, frequent edges es, integer t
Result: a list of possible frequent candidates
`refinementOperator()`

```
1  candidates set cH
2  for frequent graph H ∈ gH do
3      for vertex v ∈ V(H) do
4          for edge e ∈ es do
5              if v is incident with e then
6                  extend edge e at vertex v in H → H′
7                  remove each leaf of graph H′ to a list graphs H″
8                  if all graphs in H″ appears in gH and H′ ∉ cH then
9                      get intersection support set I of H′ from H″, |I| = t′
10                     if t′ ≥ t then
                           add H′ to C
   return cH
```

In order to check whether all the subtrees in H'' appear in gH or not, we define an order on the canonical string representations of the trees in gH. Then we use binary search to check tree isomorphism. Two methods are provided to check whether $H' \in cH$ or not, one of them is binary search which is used to check whether elements in H'' appear in gH or not. The another one is prefix tree search, and the prefix tree is constructed from the canonical strings of the candidates in cH. One example is shown in figure 3.3, and it can be seen that prefix tree search is faster in computing whether canonical string *def* is in the candidates or not. However, binary search is faster in computing whether canonical string *abe* is in

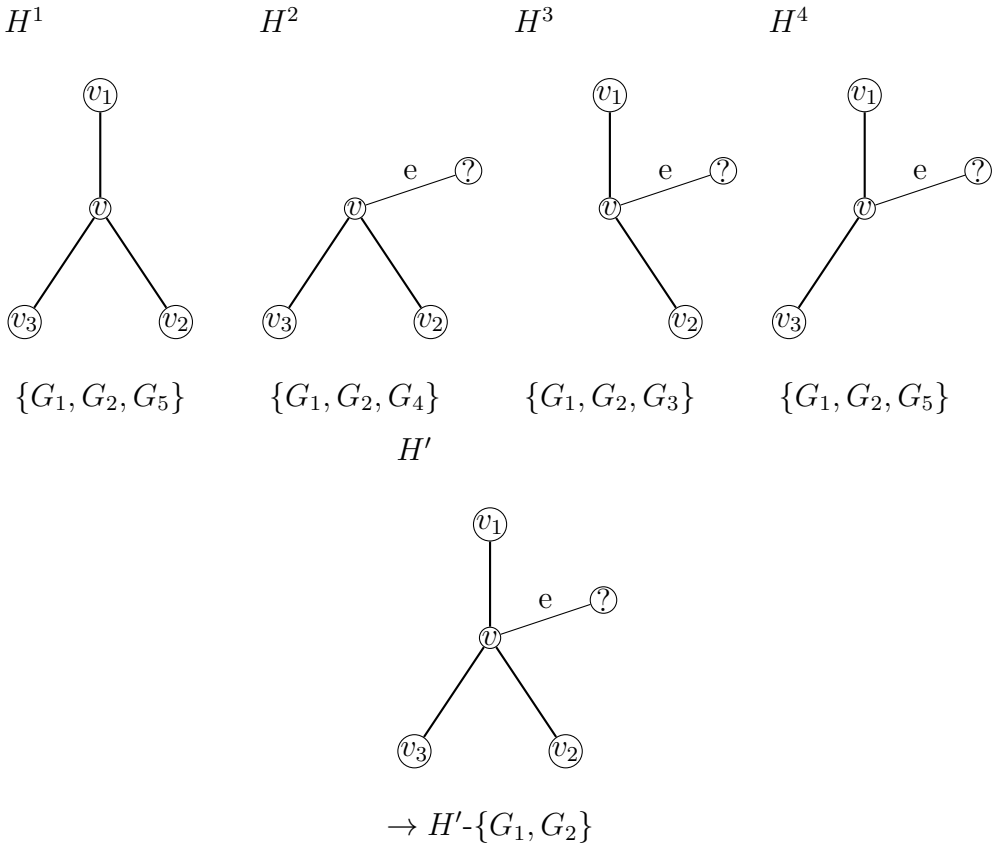
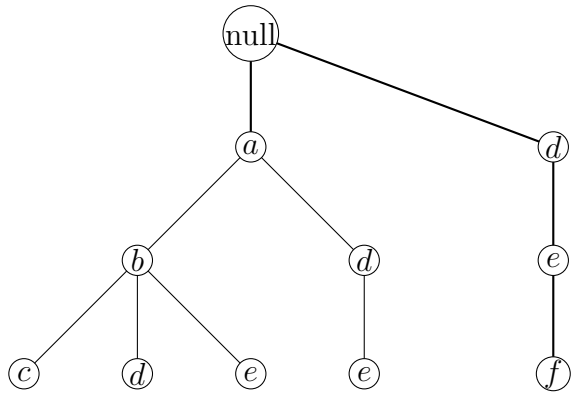


Figure 3.2: remove each leaf



ordered canonical strings $\{abc, abd, abe, ade, def\}$

Figure 3.3: prefix tree and ordered array of canonical strings

the candidates or not. Denote the length of the candidates array as L and the length of the canonical string as $|s|$, then the complexity of the binary search is $O(log(L) \times |s|)$. Denote the total number of vertex and edge labels of the candidates array as L', then the complexity of the prefix tree search is $O(L' \times |s|)$.

3.2 A Subtree Isomorphism Algorithm for General Graphs

The algorithm 4 is a dynamic programming subgraph isomorphism algorithm from a tree into a connected graph given in (Pascal Welke, 2015b). It deals with the subtree isomorphism problem defined in definition 1.2.3. The subtree isomorphism problem is a NP-complete problem and the subtree isomorphism problem is in P if we constrain the text graph class to be tree (Pascal Welke, 2015a). The algorithm 4 generalizes this positive result to text graphs in class locally easy which is defined in definition 1.2.2. It allows text graphs to have exponentially many spanning trees. The algorithm 4 uses a root tree T to iterate the text graph inspired by the idea from (Matoušek and Thomas, 1992), and the spanning trees generated in each block are combined carefully inspired by the idea from (Shamir and Tsur, 1999). The algorithm 4 gets a tree H and a text graph G as input, and it returns true if $H \preccurlyeq G$ otherwise it returns false.

Algorithm 4: Subgraph isomorphism from a tree into a connected graph

Data: tree H and an arbitrary connected graph G with $2 \leq |V(H)| \leq |V(G)|$
Result: TRUE if $H \preccurlyeq G$; o/w FALSE

```
MAIN()
1  | pick a vertex r ∈ V(G) and set C := ∅
2  | for all v ∈ V(T) in a postorder do
3  |   | let S_v be the set of spanning trees of 𝔅(v), each rooted at v
4  |   | for all T ∈ S_v do
5  |   |   | for all w ∈ V(T) in a postorder do
6  |   |   |   | for all u ∈ V(H) do
7  |   |   |   |   | for all y ∈ {u} ∪ δ(u) do
8  |   |   |   |   |   | C := C ∪ CHARACTERISTICS(v, u, y, T, w)
9  |   |   |   |   |   | if (H_u^u, T, w) ∈ C then
10 |   |   |   |   |   |   | return TRUE
11 | return FALSE
Function CHARACTERISTICS(v, u, y, T, w)
1  | for all θ ∈ Θ_vw(T) do
2  |   | let T' be the tree satisfying θ = T ∪ T'
3  |   | let C_T (resp. C_T') be the set of children of w in T (resp. T'), and
   |   |   C_θ := C_T ∪ C_T'
4  |   | let B = (C_θ ∪ δ(u), E) be the bipartite graph with cu' ∈ E
   |   | iff (c ∈ C_T ∧ (H_u'^u, T, c) ∈ C) ∨ (c ∈ C_T' ∧ (H_u'^u, T', c) ∈ C)
   |   | for all cu' ∈ C_θ × δ(u)
5  |   | if y ≠ u and B has a matching covering δ(u) \ {y} then
6  |   |   | return {(H_u^y, T, w)}
7  |   | if y = u and B has a matching covering δ(u) then
8  |   |   | return {(H_u^u, T, w)}
```

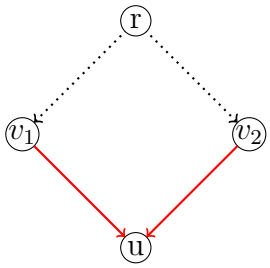


Figure 3.4: rooted tree T

3.2.1 More Concepts and Notations

In this section, we present the concepts and notations in details to understand the algorithm 4 given in (Pascal Welke, 2015b). In algorithm 4, we first need to pick an arbitrary vertex r as root as stated in line 1. Then the root v of each block B in the text graph G is defined as the vertex which has the smallest distance to vertex r. This block B is called the *v-rooted* block. A subgraph by joining all *v-rooted* blocks in G is denoted as $\mathfrak{B}(v)$. A vertex w is below another vertex v in graph G if every path from vertex r to w contains vertex v. A subgraph of G induced by the set of vertices below vertex v is denoted as G_v. The same notation is used for tree H, H_u^y denotes the subtree rooted at vertex u of the tree which is rooted at vertex y. A vertex w' is the child of vertex v if $vw' \in E(G)$ and $w' \in V(\mathfrak{B}(v))$. Let G' be an induced subgraph of G and τ' be a spanning tree of G', then $G\{G'/\tau'\}$ denotes the graph induced from G by replacing G' with τ'.

A directed tree T is defined on the set of roots of $\mathfrak{B}(v)$ in G in algorithm 4. For any vertex $u, v \in V(T)$ with $u \neq v$, $(u, v) \in E(T)$ if and only if there exists a block B with root v such that $u \in V(B)$ (Pascal Welke, 2015b). The directed tree T is rooted at vertex r, which is proved in (Pascal Welke, 2015b) by using a similar contradiction example shown in figure 3.4. The contradiction example shows that vertex $u \in V(T), u \neq r$ has indegree at most 1 otherwise v_1 and v_2 are in the same block of G. This violates the maximality of blocks B_1 and B_2 in $\mathfrak{B}(v_i)(i = 1, 2)$, you can check (Pascal Welke, 2015b) for more detail. The definitions of *iso-triple* and *v-characteristic* are given in definition 3.2.1 and 3.2.2.

Definition 3.2.1. *Let $v \in V(T)$ be a root, an **iso-triple** ξ of H relative to v is a triple (H_u^y, τ, w) such that $u \in V(H)$, $y \in \delta(u) \cup \{u\}$, τ is a spanning tree of $\mathfrak{B}(v)$, and $w \in V(\mathfrak{B}(v))$ (Pascal Welke, 2015b).*

Definition 3.2.2. *A **v-characteristic** is an iso-triple $\xi = (H_u^y, \tau, w)$ relative to v such that there exists a subgraph isomorphism φ from H_u^y to $(G\{\mathfrak{B}(v)/\tau\})$ with $\varphi(u) = w$ (Pascal Welke, 2015b).*

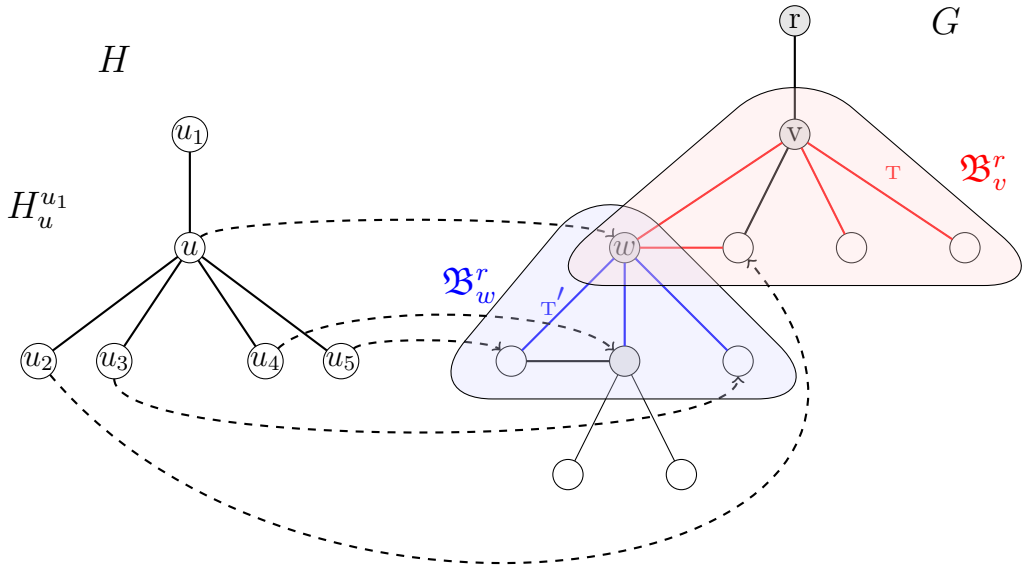


Figure 3.5: gluing spannings (Pascal Welke, 2015a)

In order to compute all the v-characteristics (H_u^y, T, w) related to a tree H and a text graph G, we iterate the directed and rooted tree T in post order. The number of (H_u^y, T, w) is bounded by a polynomial of the order of the locally easy text graph G (Pascal Welke, 2015b). If $w \in V(T)$, then it is necessary to combine the spanning tree T' rooted at vertex w and the spanning tree T rooted at vertex v like the example shown in figure 3.5. The following notations are given in (Pascal Welke, 2015b) to formalize these considerations: let $v \in V(T)$, and S_v be the set of spanning trees of $\mathfrak{B}(v)$, for any $\mathrm{T} \in S_v$ and $w \in V(\mathrm{T})$, and $\Theta_{vw}(\mathrm{T})$ is defined as

$$\Theta_{vw}(\mathrm{T}) = \begin{cases} \{\mathrm{T} \cup \mathrm{T}' : \mathrm{T}' \in S_w\}, \ if\ w \in V(T_G) \backslash \{v\} \\ \{\mathrm{T}\}, \ o/w\ (i.e., w \notin V(T_G)\ or\ v = w) \end{cases}$$

$\mathrm{T} \cup \mathrm{T}'$ denotes the tree consisting of vertex set $V(\mathrm{T}) \cup V(\mathrm{T}')$ and edge set $E(\mathrm{T}) \cup E(\mathrm{T}')$. For example, the tree by gluing T and T' at vertex w in figure 3.5 is one tree belonging to the tree set $\Theta_{vw}(\mathrm{T})$. $\Theta_{vw}(\mathrm{T})$ contains all possible gluing trees $\mathrm{T} \cup \mathrm{T}', \mathrm{T}' \in \mathfrak{B}_w^r$ and $w \in V(T) \backslash v$.

3.2.2 Algorithm Description

Finally, we are ready to explain the algorithm 4 after providing all the concepts and notations in section 3.2.1.

`MAIN()` function

The algorithm 4 receives a tree H and a connected graph G as input parameters under the condition $2 \leq |V(H)| \leq |V(G)|$. An initialization algorithm for the case $|V(H)| = 1$ is given in section 3.4. A vertex r is randomly picked from the graph G at line 1 and set C is empty. The vertex r is the root of tree T, and C is used to keep the generated v-characteristics. Next, the algorithm starts to iterate the vertices of $V(T)$ in post order from line 2, and gets the set of spanning trees S_v of $\mathfrak{B}(v)$ at line 3. It starts to iterate the spanning trees in set S_v from line 4, and begins to iterate the vertices of each spanning tree T from line 5. From line 6, the algorithm starts to iterate the vertices of the input tree H, and begins to iterate the neighbors of vertex $u \in V(H)$ and u from line 7. Then it unions the v-characteristics in C with the result returned from function `CHARACTERISTICS` at line 8. If it finds $(H_u^u, \mathrm{T}, w) \in C$ at line 9, then it returns TRUE indicating that it found a subtree isomorphism from tree H to graph G.

`CHARACTERISTICS` function

The function `CHARACTERISTICS` gets vertices v, u, y, w and spanning tree T as input parameters from the `MAIN()` function. It first computes θ by checking whether $w \in V(T)$ or not, and starts to iterate all the generated θ at line 1. It gets the T' satisfying $\theta = \mathrm{T} \cup \mathrm{T}'$ at line 2, and computes the set of children C_θ of vertex w in T and T' at line 3. Then it constructs a bipartite graph B at line 4 by combining the vertices from C_θ and $\delta(u)$. $(H_{u'}^u, \mathrm{T}, c) \in C$ and $(H_{u'}^u, \mathrm{T}', c) \in C$ are required for $c \in C_{\mathrm{T}}$ and $c \in C_{\mathrm{T}'}$. If the vertex y is not the same as the vertex u, then it returns (H_u^y, T, w) at line 6 when B contains a matching between $\delta(u) \backslash y$ and C_θ. If the vertex y is same as vertex u, then it returns (H_u^u, T, w) at line 8 when B contains a matching between $\delta(u)$ and C_θ.

3.2.3 Lemmas To Be Noticed

Some lemmas are important to be noticed, and their proofs can be found in (Pascal Welke, 2015b).

Lemma 3.2.1. *Let H be a tree and G be a graph (with root r). Then $H \preccurlyeq G$ iff there exists a v-characteristic (H_u^u, τ, w) for some $v \in V(T)$, $u \in V(H)$, spanning tree τ of $\mathfrak{B}(v)$, and $w \in V(\tau)$ (Pascal Welke, 2015b).*

Lemma 3.2.2. *An iso-triple (H_u^y, τ, w) of H is a v-characteristic for some $v \in V(T)$ and $y \in \delta(u)$ iff there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\delta(u)\backslash\{y\}$ to the children of w in θ such that for all $u' \in \delta(u)\backslash\{y\}$ there is a subgraph isomorphism $\varphi_{u'}$ from $H_{u'}^u$ to $(G\{\mathfrak{B}(v) \cup \mathfrak{B}(w)/\theta\})_{\psi(u')}$ mapping from u' to $\psi(u')$ (Pascal Welke, 2015b).*

Lemma 3.2.3. *An iso-triple (H_u^u, τ, w) of H is a v-characteristic for some $v \in V(T)$ iff there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\delta(u)$ to the children of w in θ such that for all $u' \in \delta(u)$ there is a subgraph isomorphism $\varphi_{u'}$ from $H_{u'}^u$ to $(G\{\mathfrak{B}(v) \cup \mathfrak{B}(w)/\theta\})_{\psi(u')}$ mapping u' to $\psi(u')$ (Pascal Welke, 2015b).*

3.2.4 Correctness and Runtime

Correctness

The correctness of the algorithm 4 can be shown by the completeness and soundness of the v-characteristics generation. This can be seen from the line 9 and 10 of the algorithm 4 and lemma 3.2.1. The correctness of the algorithm 4 is proved in (Pascal Welke, 2015b). The idea of completeness is that all the iso-triples are tested to be v-characteristics from line 4 and 8 of the algorithm 4. The idea of soundness is that every iso-triple is decided correctly whether it is a v-characteristics or not. This is done by showing that iso-triples are decided correctly to be v-characteristics for different height of $h_T(v)$ for $v \in V(T)$ and $h_\tau(w)$ for $w \in V(\tau)$. More detail explanations can be found in (Pascal Welke, 2015b).

Theorem 3.2.1. *Algorithm 4 is correct, i.e., for all trees H and connected graphs G with $2 \leq |V(H)| \leq |V(G)|$, it returns TRUE iff $H \preccurlyeq G$ (Pascal Welke, 2015b).*

Runtime

The runtime complexity of algorithm 4 is analyzed in (Pascal Welke, 2015b) and the analysis is summarized as follows:

1. Spanning tree generation in `Main` function
   a) Number of spanning trees in S_v is $O(n^k)$
   b) Linear delay generation of spanning trees (Read and Tarjan, 1975), $O(n^k) \to O(n^{k+1})$

c) $|V(T)|$ is bounded by n, $O(n^{k+1}) \rightarrow O(n^{k+2})$

2. Number of calls to `CHARACTERISTICS` in `Main` function
   a) Pair of v and w is $O(n)$
   b) Number of spanning trees in S_v is $O(n^k)$, $O(n) \rightarrow O(n^{k+1})$
   c) pair of u and y is $O(n)$, $O(n^{k+1}) \rightarrow O(n^{k+2})$

3. Runtime complexity of each call to `CHARACTERISTICS`
   a) Number of $\theta \in \Theta_{vw}(\mathrm{T})$ is the same as number of T', $O(n^k)$
   b) Bipartite graph B construction for $cu' \in C_\theta \times \delta(u)$, $O(n^k) \rightarrow O(n^k)O(n^2)$
   c) Maximum matching of B can be done in $O^{\frac{5}{2}}$ (Hopcroft and Karp, 1971), and $O(n^k)[O(n^2) + O(n^{\frac{5}{2}})] \rightarrow O(n^{k+\frac{5}{2}})$

The overall runtime complexity is $O(n^{k+\frac{5}{2}})O(n^{k+2}) + O(n^{k+2}) \rightarrow O(n^{2k+\frac{9}{2}})$.

3.2.5 Implementation Details

The implementation of algorithm 4 basically corresponds to the sequence of the algorithm processing steps.

`graphPreprocessing` function

The biconnected components, spanning trees and post orders of spanning trees are generated once for all the input text graphs $G \in D$ at the beginning, and this is summarized in algorithm 5. An empty list of *objs* is created at line 1 to keep the pointers of each generated *obj*. Then the algorithm 5 starts to iterate each input text graph $G \in D$ from line 2. It creates an empty object *obj* at line 3, and lists all the biconnected components at line 4. Next, it joins the generated biconnected components at line 5 by calling function `JoinBiconnectedComponents`, then it gets results in a list called *blocks*. From line 6 to line 12, it generates spanning trees from each $block \in blocks$ and post orders of each spanning tree. Then it lets *obj* points to different generated biconnected components, spanning trees and post orders of spanning trees at line 13. In the end, it appends *obj* to *objs* at line 14 and returns *objs* at line 15.

`MAIN` function

For the `MAIN` function of algorithm 4, it first takes an arbitrary vertex r from the input text graph G. Then it iterates the biconnected components of the text graph

Algorithm 5: Graph preprocessing

Data: text graph database D
Result: a list of objects, each object contains information of each $G \in D$.
`graphPreprocessing()`

```
1  a list objs = ∅
2  for G ∈ D do
3      initialize an object obj
4      list biconnected components in G → bComponents
5      JoinBiconnectedComponents(G, bComponents) → blocks //alg 6
6      a list sptLists = ∅
7      a list poLists = ∅
8      for block ∈ blocks do
9          list spanning trees in block → sptList
10         list post order of each spanning tree in sptList → poList
11         append sptList to sptLists
12         append poList to poLists
13     let obj contain pointers to bComponents, blocks, sptLists, poList, G
14     append obj to objs
15 return objs
```

G, and calculates the distance from vertices in the biconnected component to the picked vertex r. It fixes the vertex v in each biconnected component with shortest distance to vertex r as root. Afterwards it combines the biconnected components at vertex v or r to form blocks $\mathfrak{B}$. These procedures are formalized in algorithm 6.

The algorithm 6 fixes a vertex r of input text graph G at line 1. Then it gets post order of vertices in text graph G rooted at vertex r at line 2. In this case, the larger the index of the vertex v in the generated post order array $postG$ is, the smaller the distance between the vertex v and the vertex r is. It sets $array_v$ to empty to keep the vertex which has the closest distance to vertex r in each biconnected component. It starts to iterate each biconnected component from line 4, and sets variable max to -1 to keep the largest index at line 5. From line 6 to 10, it puts the vertex of each biconnected component having the closest distance to r to array $array_v$. The function `GenerateBlocksInPostOrder` takes the $postG$, $array_v$ and *bi-components* as input parameters, and sets the list *Blocks* to empty to keep the joined blocks. It starts to iterate the vertices of graph G in post order from line 13 to guarantee the generated blocks is in post order. If the vertex v is contained in the $array_v$, then v is a root of one block. It gets the indexes of vertices in $array_v$ which are same as v at line 15. These indexes correspond to the

Algorithm 6: Join biconnected components of text graph G

```
Data: text graph G, list of biconnected components of G bi-components
Result: a list of blocks 𝔅
JoinBiconnectedComponents()
1   | pick a vertex r ∈ V(G)
2   | get post order of vertices in text graph G in postG with root r
3   | set array_v = ∅
4   | for bcomponent ∈ bi-components do
5   |   | set max = −1
6   |   | for vertices v in V(bcomponent) do
7   |   |   | get index of v in postG → vi
8   |   |   | if vi > max then
9   |   |   |   | max = vi
10  |   | add the value of postG[max] to array_v
11  | return GenerateBlocksInPostOrder(postG, array_v, bi-components)
Function GenerateBlocksInPostOrder(postG, array_v, bi-components)
12  | Blocks = ∅
13  | for vertices v ∈ postG do
14  |   | if v ∈ array_v then
15  |   |   | get indexes of elements in array_v with value v → vis
16  |   |   | 𝔅 = null
17  |   |   | for bcomponent ∈ bi-components[vis] do
18  |   |   |   | 𝔅 = joinBcomponent(bcomponent, 𝔅)
19  |   |   | add 𝔅 to Blocks
20  | return Blocks
```

indexes of biconnected components rooted at v in *bi-components*. It sets blocks $\mathfrak{B}$ to empty at line 16 and joins these biconnected components from line 17 to 18. The function `joinBcomponent` at line 18 sets $\mathfrak{B}$ as *bcomponent* if $\mathfrak{B}$ is null otherwise it just appends $\mathfrak{B}$ at the end of *bcomponent* and returns the combined component. The blocks $\mathfrak{B}$ is appended to *Blocks* at line 19 and *Blocks* is returned at line 20. After getting all the joined blocks, we can list the spanning trees in each $\mathfrak{B}(v)$, and generate post order of vertices in each spanning tree by fixing root at vertex v. We now have all the necessary objects for the `MAIN` function.

`CHARACTERISTICS` function

For the `CHARACTERISTICS` function of algorithm 4, we need to decide θ first. If the vertex $w \in V(T)$, then we need to combine the spanning tree T' in spanning trees set T'' of $\mathfrak{B}(w)$ with T to constitute $\theta = \mathrm{T} \cup \mathrm{T}'$. Then we can get the set of children C_θ of w from θ by using algorithm 7.

Algorithm 7: Generate set of child of a vertex w in $\theta = \mathrm{T} \cup \mathrm{T}'$

```
   Data: tree T and T', vertex w, v
   Result: set of children of vertex w
   generateChild()
1  |  get post order of vertices in tree T → post_T with root v
2  |  get child C_T' = V(T')\w of w in tree T'
3  |  return getChild(post_T) ∪ C_T'
   Function getChild(post_order, w)
4  |  set C_T = ∅
5  |  for vertex w' ∈ post_order do
6  |  |  if w' = w then
7  |  |  |  break;
8  |  |  if w' is a neighbor of w then
9  |  |  |  add w' to C_T
10 |  return C_T
```

The algorithm 7 uses the fact that the children of the vertex w should have smaller index value than itself in the post order array *post_order* generated at line 1. The children of vertex w in T' is $V(\mathrm{T}')\backslash w$ at line 2. Then it calls the `getChild` function at line 3 to get the children of vertex w in T. It sets C_T to empty at line 4, and starts to iterate the vertices in the post order array from line 5. If it finds that vertex $w' \in post_{order}$ is same as vertex w at line 6, then it breaks at line 7. Otherwise vertex w' is added to C_T at line 9 if it is also a neighbor of vertex w.

Algorithm 8 outputs the vertex pairs of the edges in the bipartite graph B. The

Algorithm 8: Get vertices of edges of the bipartite graph B from $\delta'(u) \cup C_\theta$

Data: tree T and T', vertex w, v, child set C_T, $C_{\mathrm{T}'}$, characteristics set C
Result: vertices of edges in the bipartite graph B
`generateEdgesBipartite()`

```
1   set bv = ∅
2   for u' ∈ δ'(u) do
3       checking(C_T, u', T, u, bv)
4       checking(C_T', u', T', u, bv)
    return bv
```

Function `checking`(C_T, u', T, u, bv)

```
5   for vertex c ∈ C_T do
6       if (H^u_{u'}, T, c) ∈ C then
7           add (u', c) to bv
```

vertex y is moved already from $\delta'(u)$ if $y \neq u$ in algorithm 8. The algorithm 8 initializes bv to empty at line 1 to keep the generated vertex pairs. Then for the neighbors of vertex u at line 2, it uses `checking` function at line 3 and 4 to check whether the iso-triples are in C or not. The `checking` function checks iso-triples for each pair (u', c) with $u' \in \delta(u)$, $c \in C_\theta$, $C_\theta = C_\mathrm{T} \cup C_{\mathrm{T}'}$. It is clear that if not all the neighbors in $\delta'(u)$ are in the returned bv, then B cannot cover $\delta(u) \backslash \{y\}$ or $\delta(u)$. Hence, the algorithm simply returns in this case. We can construct a bipartite matrix from bv by using algorithm 9.

Algorithm 9: Construct bipartite matrix

Data: bipartite matrix $bi[m][n]$, $m = |\delta'(u)|$, $n = |C_\theta|$, bv
Result: bipartite matrix $bi[m][n]$, where $bi[i][j] = 1$ if there is a (u'_i, c_j)
`constructBipartiteMatrix()`

```
1   idx = 0
2   for i = 0; i < |bv|; i+ = 2 do
3       get index of bv[i + 1] in C_θ → idx_c
4       if (bv[i], u) and (bv[i + 1], w) have same edge label then
5           bi[idx][idx_c] = 1
6       if i + 2 < |bv| and bv[i] ≠ bv[i + 2] then
7           + + idx
8   return bi
```

In the algorithm 9, bv is returned from the algorithm 8. It also gets vertex u, vertex w and an empty matrix bi as input parameters. The number of rows of the matrix bi is the size of $\delta'(u)$, and the number of columns is the size of C_θ. The output of algorithm 9 is bipartite matrix bi, and $bi[i][j] = 1$ if the edge $u'_i u$ is same

as the edge $c_j w$. The algorithm 9 initializes $idx = 0$ at line 1 to index the row of bi. Then it starts to iterate the vertices array bv of the bipartite graph B from line 2. It gets the index idx_c of c in C_θ at line 3. Then it checks at line 4 whether the edge $u'_i u$ is same as the edge $c_{\{i+2\}} w$ or not, and sets the corresponding matrix value to 1 at line 5 if they are the same. At line 6, it first checks whether it would access out of the bound of bi with index $i+2$ or not. Then it checks whether $bv[i]$ equals to $bv[i+2]$ or not. If $bv[i]$ is not equal to $bv[i+2]$, then it increases the row index idx at line 7. It returns bi at line 8 in the end.

Algorithm 10: Maximum bipartite matching

Data: bipartite matrix $bi[m][n]$, $m = |\delta'(u)|$, $n = |C_\theta|$, y, u, w, bv
Result: True if there is a bipartite matching, O/W False
`maximumBipartiteMatching()`

```
1  | if y ≠ u and δ(u) = {y} and label(u) = label(w) and bv = ∅ then
2  |   return True
3  | initialize matchR[|C_θ|] with -1
4  | for j = 0; j < |δ'(u)|; ++j do
5  |   initialize seen[|C_θ|] with 0
6  |   if !checking(j, seen, matchR, |C_θ|, bi) then
7  |     return False
8  | return True
```

Function `checking(`*j, seen, matchR, $|C_\theta|$, bi*`)`

```
9  | for i = 0; i < |C_θ|; ++i do
10 |   if bi[j][i] and !seen[i] then
11 |     seen[i] = 1
12 |     if matchR[i] < 0 or checking(matchR[i], seen, matchR, |C_θ|,
   |      bi) then
13 |       matchR[i] = j
14 |       return True
15 | return False
```

The algorithm 10 determines whether the bipartite graph B covers $\delta'(u)$ or not. It gets bipartite matrix bi from the algorithm 9. It also gets vertex y, u, w and the array bv as input parameters. If the vertex y is the only neighbor of the vertex u, there are not children of vertex w, and vertex u has the same label with vertex w, then it is clear that $\emptyset$ covers $\emptyset$, so the algorithm 10 returns True at line 2. From line 3 to line 15 of the algorithm 10, we use one maximum bipartite matching algorithm (*Graph Theory: Network Flows and Matching*) to check whether it is possible to find a matching c for each $u' \in \delta'(u)$ or not. The algorithm 10 initializes an array $matchR$ of size $|C_\theta|$ with -1 at line 3 to indicate whether $c \in C_\theta$ is matched or

not. It starts to iterate each $u' \in \delta'(u)$ from line 4, and sets an array *seen* of size $|C_\theta|$ with 0 at line 5 to indicate whether it checks $c_i \in C_\theta$ already or not. It starts to find one match for u' at line 6. If it cannot find a match for u', then it returns False at line 7. If there is a match for each u', then True is returned at line 8.

The function `checking` iterates through each element $c \in C_\theta$ from line 9. If there is an edge between u' and c in bi and c has not been checked at line 10, then it sets $seen[i] = 1$ at line 11. If c has not been matched to any other elements in $\delta'(u)/u'$ or c is matched with u'' but it can find another assignment c' for $u'' \in \delta'(u)$ indexed by $matchR[i]$ at line 12, then it can set matching between current u' and c at line 13. Then it returns True at line 14 to indicate that it finds a matching for u'. If it cannot find a matching for u' by the above methods, then it returns False at line 15. This bipartite maximum matching algorithm is an adapted Ford-Fulkerson algorithm with runtime complexity $O(|V||E|)$ for graph $G = (V, E)$ (Cormen et al., 2009).

3.3 An Iterative Subtree Isomorphism Algorithm for General Graphs

Algorithm 4 computes v-characteristics for all possible subtrees of tree H to decide subtree isomorphism from H to text graph G. We can speed up this process during frequent subtree mining by using the already generated v-characteristics of tree $H - e$ and text graph G. The tree $H - e$ denotes a subtree of tree H. This idea benefits from the fact that we have to compute v-characteristics for all (or some) subtrees of tree H while doing frequent subtree mining using the levelwise (or depth first search) method.

Algorithm 11 (Welke, 2016) is an iterative subtree isomorphic algorithm for general graphs. It iteratively uses the already generated v-characteristics of tree $H - e$ and text graph G while computing the v-characteristics of tree H and text graph G. This idea has been used in the case of frequent rooted unordered subtree mining in databases of rooted unordered trees (Asai et al., 2003; Nijssen and Kok, 2003) to speed up the (polynomial delay) mining process (Welke, 2016). The idea is also used in (Horváth and Ramon, 2008) to discovery frequent subgraphs from graphs with bounded treewidth.

3.3.1 Transforming $(H - e)$-characteristics to H-characteristics

During frequent subtree mining, we denote tree $(H - e)$ as a subtree of tree H, where $e = \{a, b\} \in E(H)$, $\delta(b) = 1$. If tree $H - e$ is subtree isomorphism to input

Algorithm 11: Subgraph isomorphism from a tree into a connected graph reusing information from a subtree

Data: A tree H, an arbitrary connected graph G with $2 \leq |V(H)| \leq |V(G)|$, $r \in V(G)$, and the set C_{H-e} of all characteristics for some $e = \{a, b\}$ where b is a leaf

Result: TRUE if $H \preccurlyeq G$; o/w FALSE and the set C of all characteristics of H

```
MAIN()
1  set X := FALSE
2  set C := ∅
3  let p_a(u) be the unique parent in H^a of all u ∈ V(H)
4  for all v ∈ V(T) in a postorder do
5      let S_v be the set of spanning trees of 𝔅(v), each rooted at v
6      for all T ∈ S_v do
7          for all w ∈ V(T) in a postorder do
8              // Add new characteristics
9              add (H_b^a, ⊤, w) to C
10             if (H_a^a, T, w) ∈ C_{H-e} then
11                 add (H_b^a, ⊤, w) to C
12             C := C ∪ CHARACTERISTICS(v, b, b, ⊤, w)
13             if (H_b^b, T, w) ∈ C then
14                 set X := TRUE
15             // Filter existing characteristics (wrt. H − e)
16             for all (H_u^y, T, w) ∈ C_{H-e} do
17                 if y = p_a(u) then
18                     add (H_u^y, ⊤, w) to C
19                 else
20                     C := C ∪ CHARACTERISTICS(v, u, y, ⊤, w)
21                 if (H_u^u, T, w) ∈ C then
22                     set X := TRUE
23 return X and C
```

text G, then we can use the computed v-characteristics of subtree $H - e$ and text graph G to decide whether tree H is subtree isomorphism to text graph G or not. In order to combine the already computed v-characteristics of subtree $H - e$ and text graph G with the new characteristics after extending $(H - e)$ with vertex b, we need to clarify the positions where b is added. The following positions of new added vertex b are distinguished in (Welke, 2016):

1. $b \notin \{y, u\}$, then a characteristic with respect to H exists only when it also exists with respect to $H - e$.
   a) $y \neq u$ and a is not below u in H_u^y, like shown in figure 3.6a and 3.6b, then $(H - e)_u^y = H_u^y$ which implies that $(H_u^y, \intercal, w)$ is a v-characteristic if and only if $((H - e)_u^y, \intercal, w)$ is a v-characteristic.
   b) $y \neq u$ and a is below u in H_u^y, like shown in figure 3.6c and 3.6d, then $(H - e)_u^y$ is a proper rooted subtree of H_u^y which implies that $((H-e)_u^y, \intercal, w)$ is a v-characteristic only if $(H_u^y, \intercal, w)$ is a v-characteristic, then we need to check the existing characteristics w.r.t $H-e$ for validity when extending it by e to H.
   c) $y = u$, like shown in figure 3.7a and 3.7b, then a is below u in $H_u^y = H_u^u$, we need to check the existing v-characteristic $((H-e)_u^u, \intercal, w)$ for validity.
2. $b \in \{y, u\}$, a corresponding characteristic with respect to $H - e$ does not exist and new characteristics may arise
   a) $y = a$ and $u = b$, like shown in figure 3.6e, then there is always a new v-characteristic $(H_b^a, \intercal, w)$.
   b) $y = b$ and $u = a$, like shown in figure 3.6f, then there is always a new v-characteristic $(H_a^b, \intercal, w)$ if and only if there is a v-characteristic $((H - e)_b^a, \intercal, w)$.
   c) $u = y = b$, like shown in figure 3.7c, then there is a v-characteristic $(H_b^b, \intercal, w)$ if and only if there exist either a v-characteristic $(H_a^b, \intercal, c)$ for some child of w in $\intercal$, or a w-characteristic $(H_a^b, \intercal', c)$ for some spanning tree $\intercal'$ of $\mathfrak{B}(v)$ and some child of w in $\intercal'$.

3.3.2 Algorithm Description

All the possible positions that the new vertex b occupies in a v-characteristic with respect to H are shown in section 3.3.1. The procedures from line 1 to 7 of algorithm 11 are same as given in algorithm 4. The line 9 of algorithm 11 deals

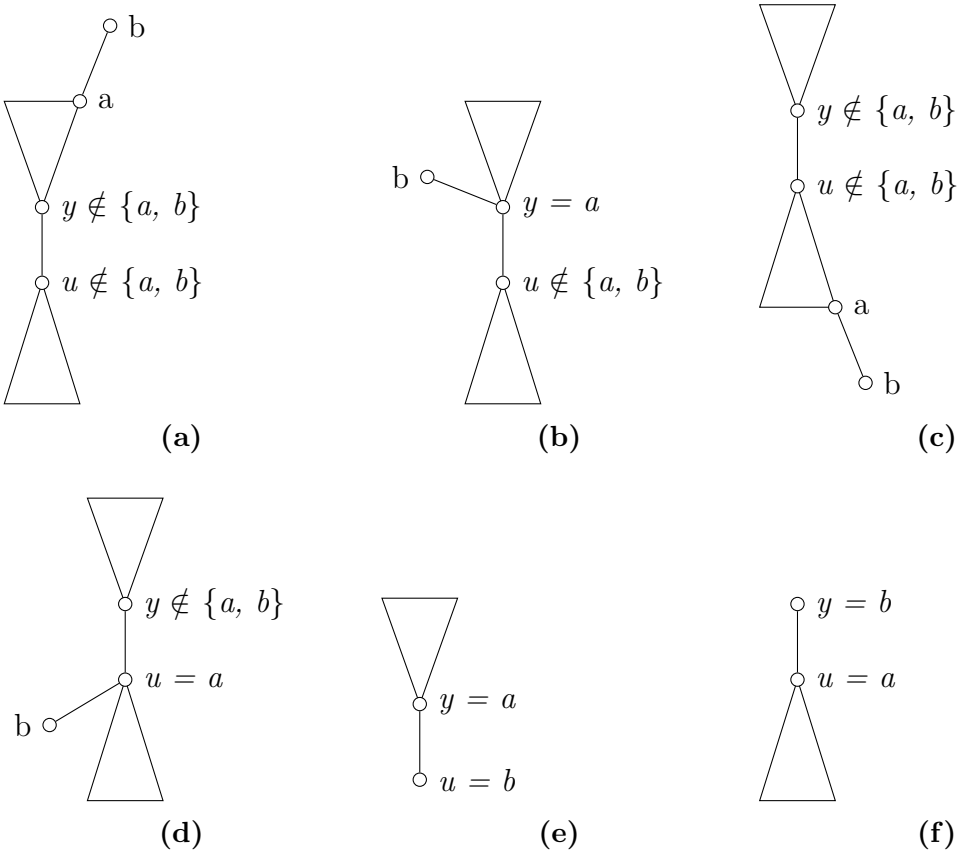


Figure 3.6: six cases of H_u^y for $u \neq y$ (Welke, 2016)

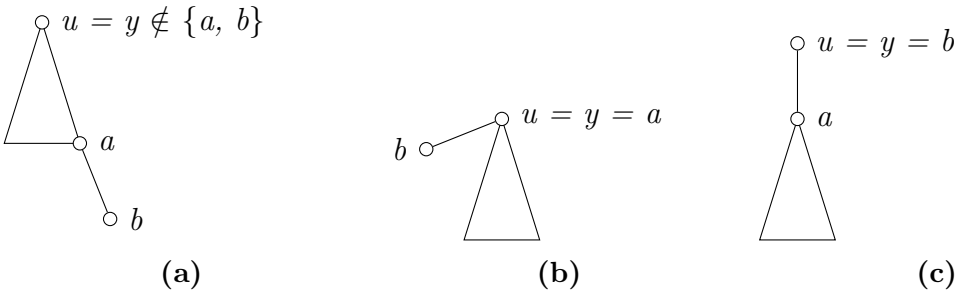


Figure 3.7: three cases of H_u^u for $u = y$ (Welke, 2016)

with the case shown in figure 3.6e. Line 10 and 11 deal with the case shown in figure 3.6f. From line 12 to 14, it deals with the case shown in figure 3.7c. If it finds $(H_b^b, \intercal, w)$ at line 13, then it finds subtree isomorphism from tree H to text graph G. From line 16 to 22, it deals with the other cases shown in figure 3.7 and figure 3.6. It starts to iterate each triple in C_{H-e} from line 16. It checks at line 17 whether vertex a is below vertex u in H_u^y or not, if vertex y is between vertex u and vertex a, then it knows that vertex a is above vertex u. So it deals with cases shown in figure 3.6a and 3.6b at line 17 and 18. If vertex a is below vertex u in H_u^y or H_u^u, like the cases shown in figure 3.6c, 3.6d, 3.7a and 3.7b, then it checks $(H_u^y, \intercal, w)$ and $(H_u^u, \intercal, w)$ at line 20. If it finds $(H_u^u, \intercal, w)$ in C at line 22, then it finds subtree isomorphism from tree H to text graph G.

3.3.3 Correctness and Runtime

Lemma 3.3.1. *Algorithm 11 computes the set of v-characteristics of H and G correctly, given correct input (Welke, 2016).*

The proof of correctness of the algorithm 11 is given in (Welke, 2016). The idea of completeness is that the algorithm 11 does not skip to check any iso-triple that might be a v-characteristic. The idea of soundness is that the added iso-triples by algorithm 11 are all v-characteristics. You can check (Welke, 2016) for more detail.

The algorithm 11 speeds up the frequent subtree mining process. It brings more benefits when the order of the frequent pattern is relatively large. The algorithm 11 has less calls to function `CHARACTERISTICS` as compared to algorithm 4.

3.3.4 Implementation Details

The implementation of the algorithm 11 is similar to the implementation of algorithm 4, and they use the same `CHARACTERISTICS` function. The algorithm 11 checks whether vertex y is a parent of vertex u or not in pattern tree H at line 15 by assuming that H is rooted at vertex a. This is done by using algorithm 12, the algorithm generates post order $post_a$ of vertices from pattern tree H rooted at vertex a at line 1, and then it gets the indexes of vertex y and u in $post_a$ from line 2 to 7. If the index of vertex y is larger than the index of vertex u, then the vertex y is between u and a, as y is neighbor of u, so vertex y is the parent of u, and then the algorithm 11 returns True at line 9. Otherwise, the algorithm returns False at line 10.

Algorithm 12: Check whether $y = p_a(u)$ in pattern graph H

Data: vertex u, y, a and pattern graph H
Result: True if $y = p_a(u)$; o/w False
`parentChecking()`

```
1  | get vertices in post order from H rooted at a → post_a
2  | u_idx = −1, y_idx = −1
3  | for i=0; i< |V(H)|; i++ do
4  |   | if post_a[i] = u then
5  |   |   | u_idx = i
6  |   | if post_a[i] = u then
7  |   |   | y_idx = i
8  | if y_idx > u_idx then
9  |   | return True
10 | return False
```

3.4 Initialization for Singleton Patterns

Since we have the condition $2 \leq |V(H)| \leq |V(G)|$ in algorithm 4 and 11, then one algorithm to process the singleton pattern trees is needed. The algorithm 13 (Welke, 2016) is the resulting algorithm which always returns True for unlabeled graphs as single vertex matches any vertex on input text graph G. For labeled graphs, it returns True only if the label of the single vertex H is the same with the label of the vertex $w \in V(G)$.

Lemma 3.4.1. *Let H be a tree containing the single vertex 0 and G be a connected graph. There there is a v-characteristic (H_0^0, τ, w) for all $v \in V(T)$, $w \in V(\mathfrak{B}(v))$ and $\tau \in S_v$ (Welke, 2016).*

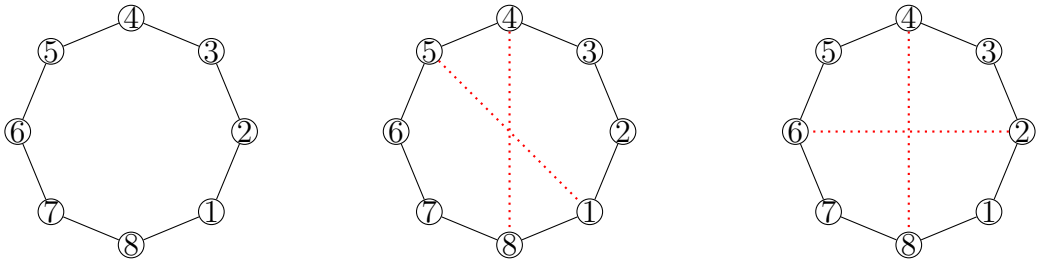
3.5 A Method to Construct Graphs with n^2 Spanning Trees

In this section we propose our method to construct graphs with exactly n^2 spanning trees. This class of graphs are important for the evaluations of the frequent subtree algorithms and subtree isomorphism algorithms. In order to introduce our method, definitions 3.5.1 and 3.5.2 are given. Like the examples shown in figure 3.8, the edges $\{1, 5\}$, $\{4, 8\}$, and $\{2, 6\}$ are half circle edges, but only $\{1, 5\}$ and $\{4, 8\}$ are adjacent half circle edges.

Algorithm 13: Subgraph isomorphism from a tree into a connected graph reusing information from a subtree - initialization algorithm

Data: A tree H, an arbitrary connected graph G with $1 = |V(H)| \leq |V(G)|$, $r \in V(G)$
Result: The set C of all characteristics of H, and TRUE if $H \preccurlyeq G$ otherwise FALSE

```
MAIN()
1   set X := FALSE
2   set C := ∅
3   for all v ∈ V(T) in a postorder do
4       let S_v be the set of spanning trees of B(v), each rooted at v
5       for all T ∈ S_v do
6           for all w ∈ V(T) in a postorder do
7               vertex u = V(H)
8               if u has the same label with w then
9                   add (H_0^0, T, w) to C
10                  set X := TRUE
11  return X and C
```



(a) C_8 **(b)** C_8^1 **(c)** C_8^2

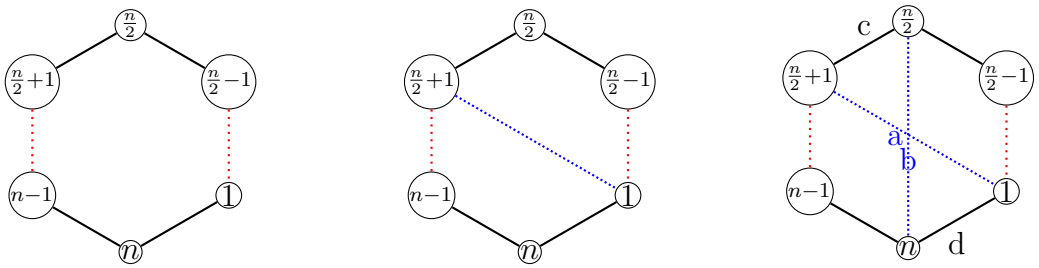
Figure 3.8: half circle graphs

Definition 3.5.1. ***Half Circle Edge****: an edge which connects any two vertices of a circle C. This edge conceptually divides the circle C into two parts, and each part contains half of the edges of C.*

Definition 3.5.2. ***Adjacent Half Circle Edges****: half circle edges which have adjacent start and end points in a circle C.*

Lemma 3.5.1. *For any circle C_n with n vertices, $n \geq 4$ and $n \bmod 2 = 0$. If we add any two arbitrary adjacent half circle edges to C_n, then we raise the number of spanning trees of C_n from n to n^2.*

Lemma 3.5.1 describes our method to generate graphs with exactly n^2 spanning trees. The correctness of it can be proved as follows:



(a) C_n **(b)** C'_n **(c)** C''_n

Figure 3.9: half circle graphs proof

Proof. Assume we have a graph C_n as shown in figure 3.9a, with $n \geq 4$ and $n \bmod 2 = 0$. Denote edge $a = \{\frac{n}{2}+1, 1\}$, $b = \{\frac{n}{2}, n\}$, $c = \{\frac{n}{2}+1, \frac{n}{2}\}$ and $d = \{n, 1\}$. The vertex $\frac{n}{2}+1$ and $n-1$ can be same vertex when $n = 4$. The same applies to vertex $\frac{n}{2}-1$ and 1. Two adjacent half circle edges a and b are added to C_n, then graph C''_n is constructed in figure 3.9c. The spanning trees of C''_n can be classified in the following four categories:

1. Spanning trees not contain edge a and b. This category contains $\binom{n}{1}$ spanning trees, and they are same as the spanning trees of circle C_n.

2. Spanning trees contain only one edge from a and b, one example is shown in figure 3.9b. In this case, we need to remove one edge from each part conceptually divided by a or b. Thus the number of spanning trees is $\binom{2}{1} \times \binom{\frac{n}{2}}{1} \times \binom{\frac{n}{2}}{1}$.

3. Spanning trees contain both edge a and b. In this case, the edges a, b, c and d form a circle. Then at least one of c and d has to be removed to generate spanning trees from C''_n.

   a) Remove one edge from c and d. One edge in path from vertex $\frac{n}{2}+1$ pass vertex $n-1$ to vertex n needs to be removed, and one edge in path from vertex $\frac{n}{2}$ pass vertex $\frac{n}{2}-1$ to vertex 1 needs to be removed. Each of these two paths contains $\frac{n}{2}-1$ edges, so there are $\binom{2}{1} \times \binom{\frac{n}{2}-1}{1} \times \binom{\frac{n}{2}-1}{1}$ spanning trees in this case.

   b) Remove both edge c and d. Only one edge from the remaining edges in C''_n except edge a and b needs to be removed, so there are $\binom{2}{2} \times \binom{(\frac{n}{2}-1)\times 2}{1}$ spanning trees in this case.

According to the above cases, there are

$$
\begin{aligned}
&= \binom{n}{1} + \binom{2}{1} \times \binom{\frac{n}{2}}{1} \times \binom{\frac{n}{2}}{1} + [\binom{2}{1} \times \binom{\frac{n}{2}-1}{1} \times \binom{\frac{n}{2}-1}{1} \\
&\quad + \binom{2}{2} \times \binom{(\frac{n}{2}-1)\times 2}{1}] \\
&= n + 2 \times \frac{n}{2} \times \frac{n}{2} + [2 \times (\frac{n}{2}-1)^2 + 2 \times (\frac{n}{2}-1)] \\
&= n + \frac{n^2}{2} + [\frac{n^2}{2} - n] \\
&= n^2
\end{aligned}
$$

spanning trees in total. As circle C_n is symmetric, adding any two arbitrary adjacent half circle edges to it raises the number of spanning trees of it from n to n^2. Thus, lemma 3.5.1 is correct. □

4 Experimental Evaluation

In this section we develop some experiments to evaluate:

1. Order of polynomial of the runtime of subtree isomorphism algorithm 4 based on different input graphs.
2. Order of polynomial of the runtime of frequent subtree mining algorithm 1 based on different input graphs. Algorithm 1 calls the iterative subtree isomorphism algorithm 11.
3. Runtime of binary and prefix tree search for candidates elimination based on different input graphs.
4. Runtime of our implemented algorithms, and comparison with other frequent subgraph mining tools.

Biconnected components are assumed to contain more than 2 vertices in this section. Denote n as the order of a graph G, and constants $c,\ k \geq 0$. Two conditions are given in (Pascal Welke, 2015b) to obtain locally easy graphs:

1. If each biconnected component of G contains at most $O(n^k)$ spanning trees, then G is a *k-easy* graph.
2. If each vertex v of G belongs to at most c distinct biconnected components, then G is of bounded cyclic block degree c.

The number of spanning trees containing a specific vertex v in graph G cannot be bounded by only one of these two conditions. It is bounded by $(n^k)^c = n^{k \times c}$, hence $k \times c$ is used as the parameter to generate locally easy graphs.

4.1 Statistical Collection of Graphs with Different Easiness

We collected some statistical results from two real graph datasets about the number of graphs with different easinesses. The number of spanning trees of all the

biconnected components containing vertex v in each connected graph G in the real datasets is counted. The maximum number of spanning trees in each biconnected component containing v is counted until 10^8, the counting for this graph is terminated if there are more than 10^8 spanning trees in any biconnected component. Let us denote the number of spanning trees of all the biconnected components containing vertex v by $|T|$. The value of $c' \times k'$ is calculated by

$$c' \times k' = \frac{log(|T|)}{log(n)}, c' \leq c,\ k' \leq k.$$

Then the value of $c' \times k'$ is recalculated by

$$c' \times k' = \lfloor c' \times k' + 0.5 \rfloor.$$

Then graphs with $n^{1.5} \leq |T| < n^{2.5}$ maximum number of spanning trees of all the biconnected components containing vertex v are assigned to graph class with n^2 maximum number of spanning trees of all the biconnected components containing vertex v.

The AIDS99 (*AIDS99*) graph dataset contains 39337 connected graphs, and 39323 out of 39337 connected graphs are counted under the above conditions. There are 14 graphs that have more than 10^8 spanning trees for at least one biconnected component containing vertex v. The statistical result is shown in figure 4.1. We further count the number of graphs with different easiness for ZINC (*ZINC*) graph dataset which has size 3.6Gb in disk. It is not easy to load all the graphs in memory, so we randomly sample 10^6 graphs from it. This gives a 407Mb file in disk, and ten files are sampled in this way. The average counting of graphs with different easiness in these ten files is also shown in figure 4.1. It can be seen from figure 4.1 that the maximum number of spanning trees of all the biconnected components containing vertex v in most graphs are n^0, n^1 and n^2. Therefore, we evaluate algorithms based on graphs with these easinesses.

We generate some synthetic graph datasets with n^0, n^1 and n^2 spanning trees and the vertices and edges of the generated graphs are randomly labeled. The vertices of the generated graphs are randomly labeled from integer value 1 to 4, and the edges are randomly labeled from integer value 1 to 3 unless otherwise noted.

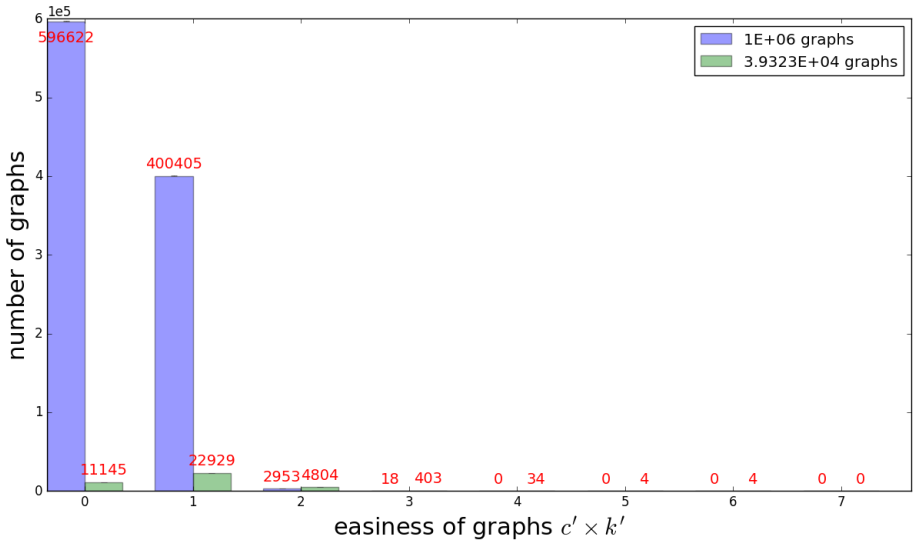


Figure 4.1: number of graphs with different easinesses, by allowing $n^{c' \times k'} = 10^8$

4.2 Subtree Isomorphism Evaluation

In this section, we evaluate subtree isomorphism algorithm 4 based on different input graphs. We first present subtree isomorphism evaluations based on locally easy graphs, then give the evaluations based on non locally easy graphs. The runtime of the algorithm is tested 2 or 3 times against different numbers of vertices of graphs. Then a polynomial fitting is generated based on the averaged runtime. The runtime data is not shown in all the figures to avoid messy plots.

4.2.1 Evaluation on Locally Easy Graphs

Subtree isomorphism evaluations based on some constant values of c and k are shown in this section. Different orders of polynomial of the runtime of the subtree isomorphic algorithm 4 are shown based on different input graphs.

Runtime on 0-*easy* and 0 Cyclic Block Degree Graphs

A tree t is a 0-*easy* and 0 cyclic block degree graph, because any biconnected component of a tree contains no more than 2 vertices. Based on the runtime analysis of the subtree isomorphism algorithm 4 at section 3.2.4, its runtime complexity on trees is $O(n^{\frac{9}{2}})$. In this section, we present runtime results of the subtree isomorphism algorithm 4 based on trees. We generate trees in star shape from vertices 6 to 2000 like the example shown in figure 4.2, and check subtree isomorphism

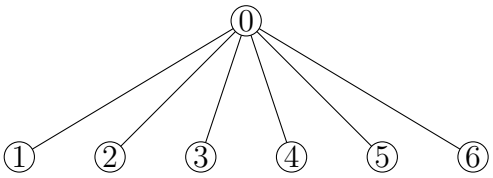


Figure 4.2: star tree

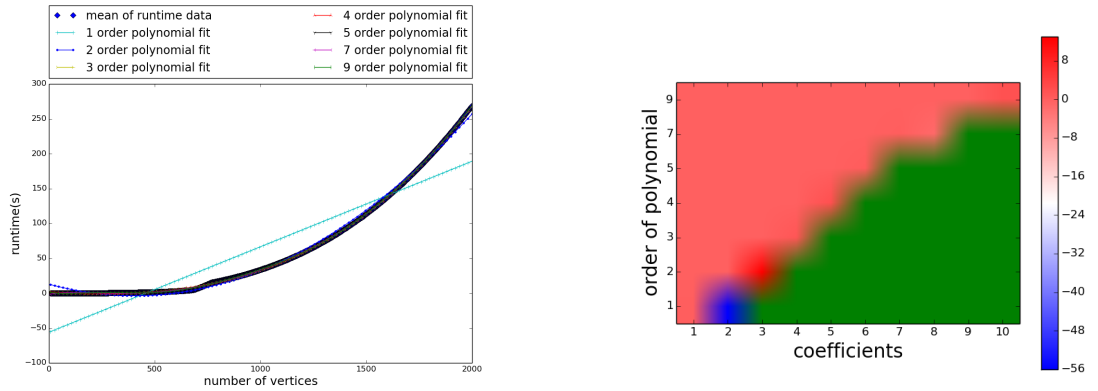
Order of polynomial	Polynomial function
9	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0$
7	$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0$
5	$x^5 + x^4 + x^3 + x^2 + x^1 + x^0$
4	$x^4 + x^3 + x^2 + x^1 + x^0$
3	$x^3 + x^2 + x^1 + x^0$
2	$x^2 + x^1 + x^0$
1	$x^1 + x^0$

Table 4.1: different order of polynomial and corresponding function

from the star tree with 6 vertices to each of the generated star trees. The result is shown in figure 4.3a, and it can be seen that polynomial fittings with orders larger than equal to 2 coincide. Hence, the 2nd order of polynomial fits well already. The coefficients of different orders of polynomial are plotted in figure 4.3b. For example, we can extract coefficients in table 4.2 from the different order polynomial functions in table 4.1. The green area in figure 4.3b corresponds to the non existing coefficients.

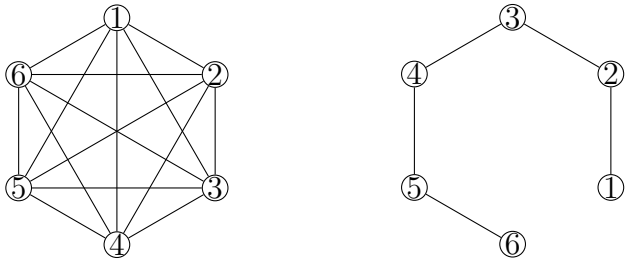
$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \\ 1 & 1 & 1 & 1 & 1 & 1 & & & & \\ 1 & 1 & 1 & 1 & 1 & & & & & \\ 1 & 1 & 1 & 1 & & & & & & \\ 1 & 1 & 1 & & & & & & & \\ 1 & 1 & & & & & & & & \end{bmatrix}$$

Table 4.2: coefficients of different polynomial functions



(a) polynomial fit **(b)** coefficients

Figure 4.3: start tree



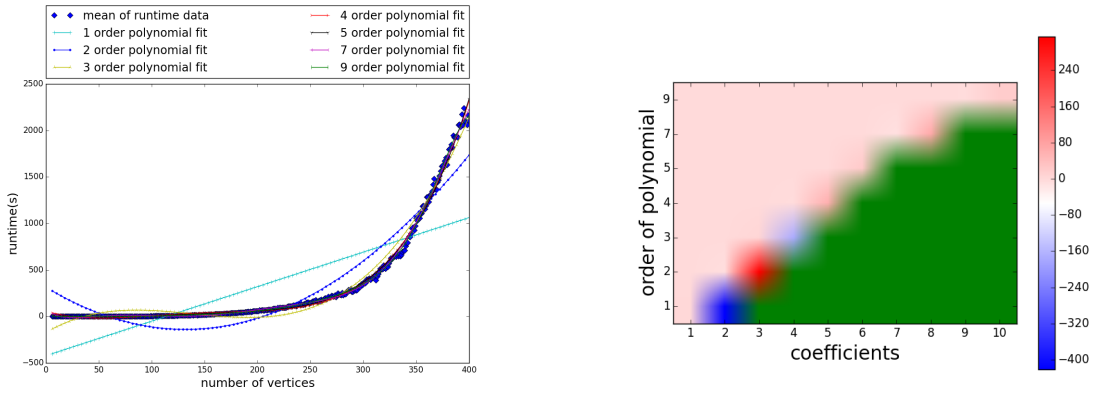
(a) K^6 **(b)** K_p^6

Figure 4.4: K^6 and one spanning tree of K^6

Runtime on 1-*easy* and 1 Cyclic Block Degree Graphs

The runtime complexity of the subtree isomorphism algorithm 4 on 1-*easy* and 1 cyclic block degree graphs is $O(n^{2+\frac{9}{2}})$ according to the analysis in section 3.2.4. A complete graph has bounded cyclic block degree 1, and the number of spanning trees of the only biconnected component is n^{n-2}. This can be calculated by Caley's formula (Aigner and Ziegler, 2010). The number of spanning trees of complete graphs grows exponentially with the number of vertices. The spanning tree of a complete graph K^n is a path with n vertices, and it is denoted as K_p^n. Examples of K^6 and K_p^6 are shown in figure 4.4a and 4.4b.

Circles A circle C_n is a 1-*easy* and 1 cyclic block degree graph with n vertices, and the number of spanning trees of the only biconnected component is n. We generate circles with $6 \leq n \leq 200$, and check subtree isomorphism from K_p^6 to



(a) polynomial fit **(b)** coefficients

Figure 4.5: subtree isomorphism runtime on circles

each generated circle. The runtime and different orders of polynomial fitting are shown in figure 4.5a. It can be seen that the 3rd order polynomial fitting already generalizes the runtime well. The coefficients of different orders of polynomial fitting are shown in figure 4.5b.

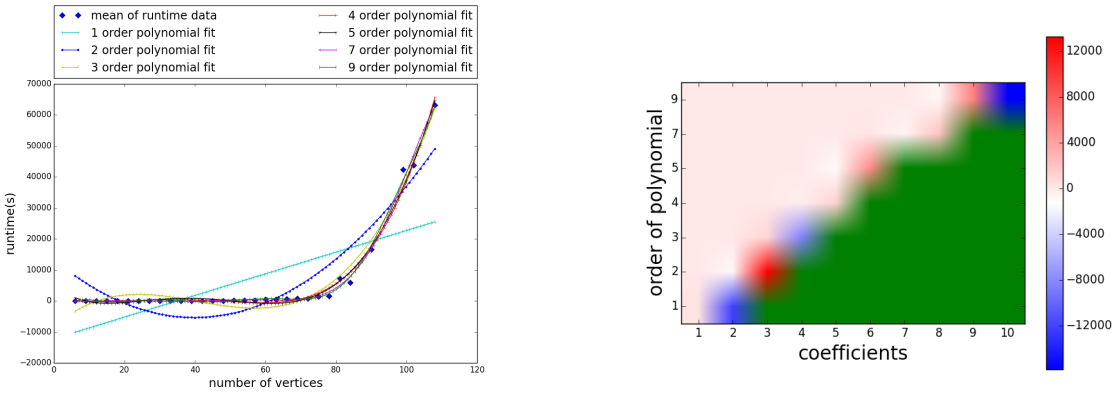
Runtime on 2-*easy* and 1 Cyclic Block Degree Graphs

The 2-*easy* and 1 cyclic block degree graphs are generated by the method introduced in lemma 3.5.1. The runtime of the subtree isomorphism algorithm 4 is shown in figure 4.6a. The corresponding coefficients of different orders of polynomial are shown in figure 4.6b. It can be seen that the 3rd order polynomial generalizes the runtime already well. Even though the runtime of the subtree isomorphism algorithm on circles is also 3rd order of polynomial, the coefficients of 3rd order polynomial of circles have much lower values than the coefficients in figure 4.6b.

4.2.2 Evaluation on Non Locally Easy Graphs

Runtime on Complete Graphs

It has been shown in section 3.2.4 that the runtime complexity of the algorithm 4 is $O(n^{2k+\frac{9}{2}})$. A complete graph K^n contains n^{n+2} spanning trees, then the complexity of algorithm 4 is $O(n^{2(n+2)+\frac{9}{2}}) = O(n^{2n+\frac{17}{2}})$ for complete graphs. We generate complete graphs from $3 \leq n \leq 7$, and check subtree isomorphic from K_p^3 to all the generated complete graphs. The number of spanning trees of complete



(a) polynomial fit **(b)** coefficients

Figure 4.6: subtree isomorphism runtime on 2-*easy* graphs

graphs is shown in 4.7a, and the corresponding subtree isomorphism runtime is shown in figure 4.7b. An exponential curve $a \times n^n + b \times n^c$ is generated based on the runtime data by using the implementation (Jones et al., 2001–) of the algorithm proposed in (Marquardt, 1963). It can be seen from figure 4.7b that the exponential fitting generalizes the data well.

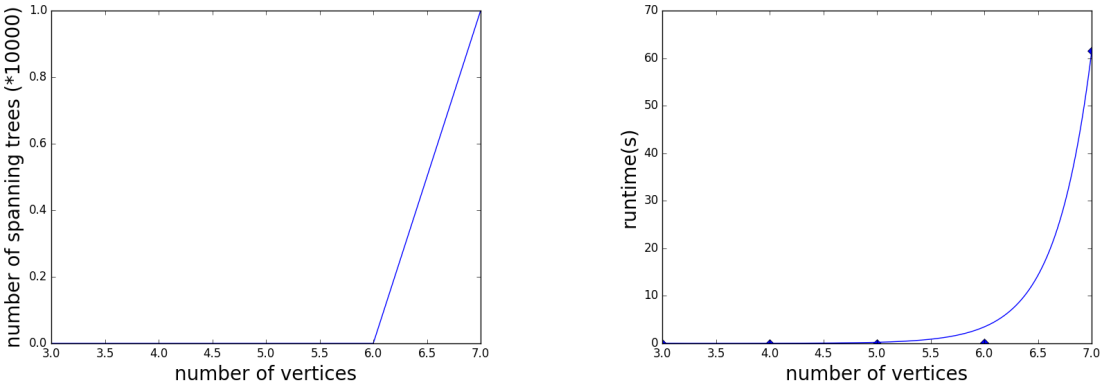
4.3 Frequent Subtree Mining Evaluation

This section describes the evaluations of the levelwise frequent subtree mining algorithm 1 with the binary search candidate elimination method as a subroutine. The iterative subtree isomorphic algorithm 11 is used to check subtree isomorphism. The labels of vertices and edges of synthetic graphs are randomly labeled with an integer value from $\{1, \ldots, 10\}$. The evaluation results on locally easy graphs are given first, then the evaluation results on non locally easy graphs are presented.

4.3.1 Evaluation on Locally Easy Graphs

Runtime on 0-*easy* and 0 Cyclic Block Degree Graphs

We already know that trees are 0-*easy* and 0 cyclic block degree graphs. We run the levelwise algorithm 1 with the iterative subtree isomorphism algorithm 11 on different sizes of input trees. Different numbers of trees are randomly sampled from 1000 AIDS trees (*1000 AIDS trees*), and different orders of polynomial fitting are generated based on the runtime. The runtime with different orders of polynomial



(a) spanning trees **(b)** run time

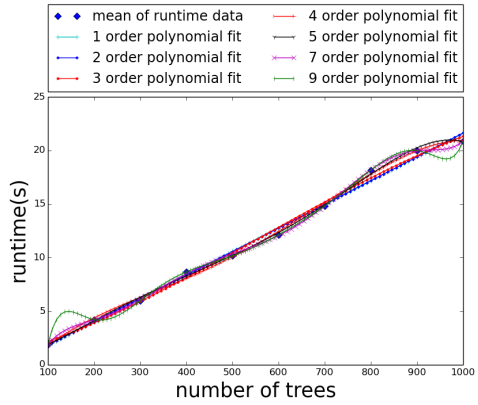
Figure 4.7: complete graph, spanning trees and runtime

fitting are shown in figure 4.8a, and the corresponding coefficients are plotted in figure 4.8b. It can be seen that the 1st order polynomial fitting generalizes the result well.

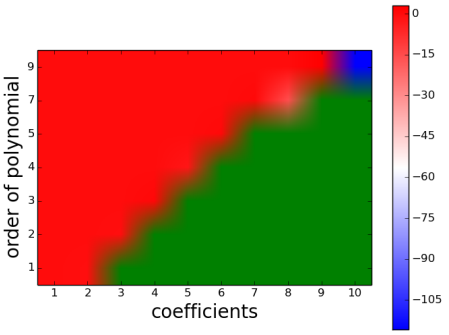
In order to check the generality about the 1st order polynomial of the runtime of frequent subtree mining algorithm 1 on trees, the algorithm is tested again on a larger dataset. The dataset contains 9000 spanning trees which are randomly generated from the AIDS99 graph dataset. The polynomial fitting results are shown in figure 4.9a. The corresponding coefficients of different orders of polynomial are plotted in figure 4.9b. It can be seen from figure 4.9a that the 1st order of polynomial does not coincide with other orders of polynomial. Furthermore, it is only slightly more general than other orders of polynomial. Thus, we can say that the 1st order polynomial fitting generalizes the runtime.

Runtime on 1-*easy* and 1 Cyclic Block Degree Graphs

We already know that circles are 1-*easy* and 1 cyclic block degree graphs, and algorithm 1 is tested on different sizes of input circles in this part. The different orders of polynomial fitting and the runtime are shown in figure 4.10a, and the corresponding coefficients of different orders of polynomial fitting are shown in figure 4.10b. It can be seen that the 1st order polynomial fits the runtime well already. The algorithm 1 is again tested on another circle dataset which contains circles with more vertices. This implies that in general the subtree isomorphism checking takes more time. The runtime result is shown in figure 4.11, and it can be seen that the first order of polynomial already generalizes the runtime. The slope of this first order of polynomial is in $(230, 300)$ which is much lager than

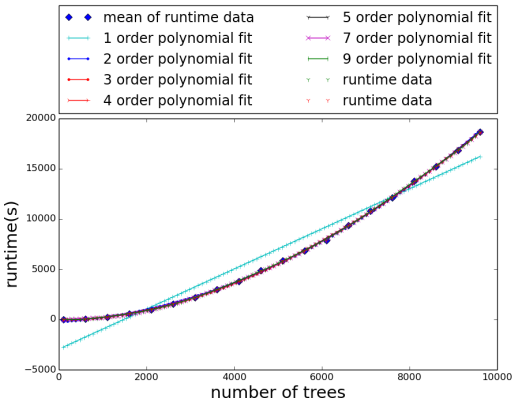


(a) mining runtime with maximum 5-*order* on different size of input

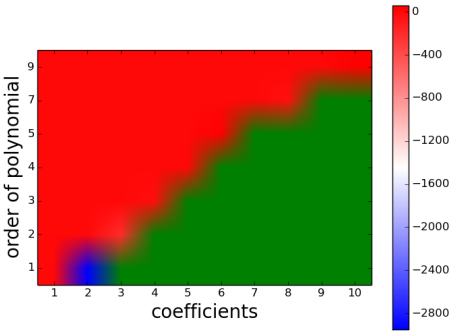


(b) coefficients of different orders of polynomial on trees

Figure 4.8: different orders of polynomial and coefficients, 1000 trees

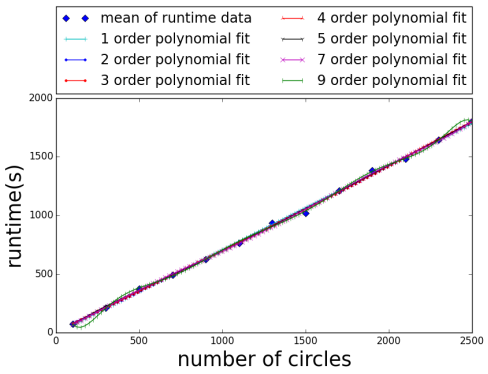


(a) mining runtime with maximum 5-*order* on different size of input

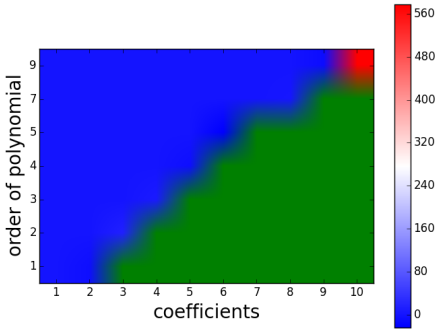


(b) coefficients of different orders of polynomial on trees

Figure 4.9: different orders of polynomial and coefficients, 9000 trees



(a) mining runtime with maximum 4-*order* on different size of input



(b) coefficients of different orders of polynomial on circles

Figure 4.10: different orders of polynomial and coefficients, circles

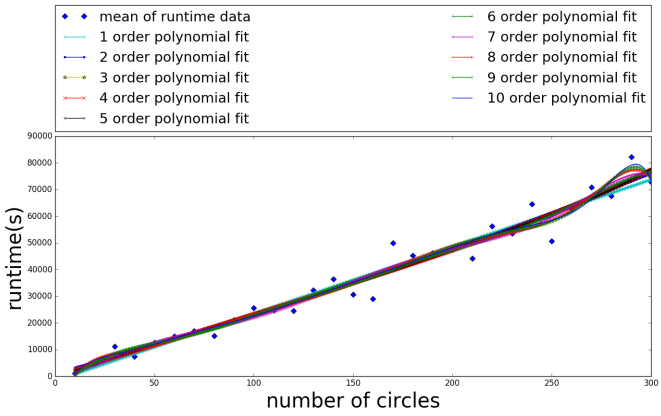
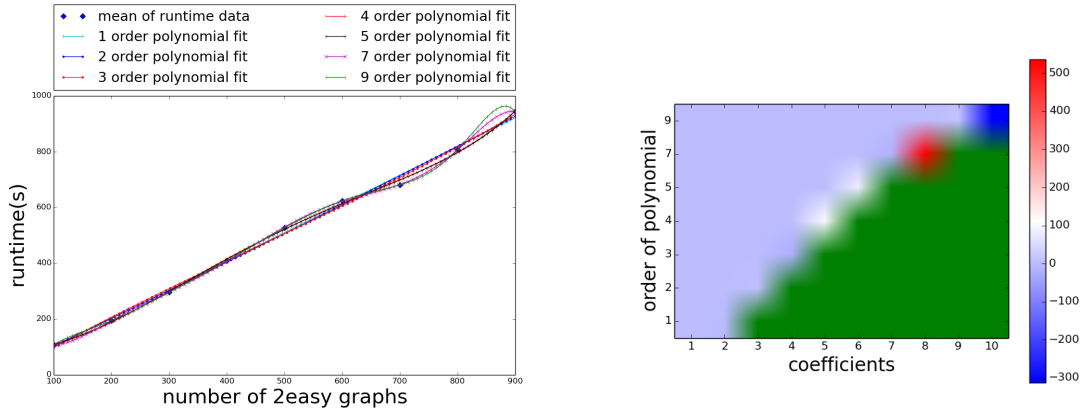


Figure 4.11: mining runtime on more complex circles

the slope of the first order polynomial shown in figure 4.10a. This is as expected because the graphs are more complex.

Runtime on 2-*easy* and 1 Cyclic Block Degree Graphs

The method to construct 2-*easy* and 1 cyclic block degree graphs was introduced in lemma 3.5.1. We use that method in this section to construct 2-*easy* and 1 cyclic block degree graphs, and evaluate the runtime of the levelwise frequent subtree mining algorithm on these graphs. The runtime and different orders of polynomial fitting are shown in figure 4.12. It can be seen from figure 4.12a that the 1st order polynomial fits the runtime well already.



(a) mining runtime with maximum 3-*order* on different size of input

(b) coefficients of different orders of polynomial on 2-*easy* graphs

Figure 4.12: different orders of polynomial and coefficients, 2-*easy* graphs

4.3.2 Evaluation on Non Locally Easy Graphs

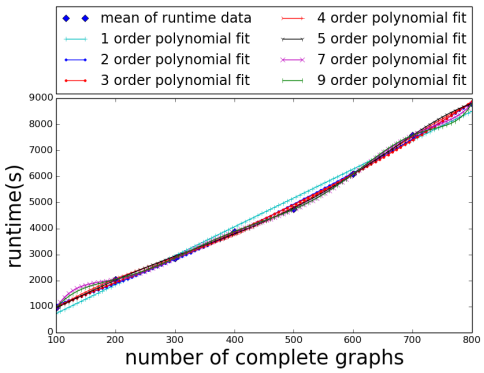
Complete graphs are used in this section to evaluate the runtime of the frequent subtree mining algorithm 1 on non locally easy graphs. The runtime result is shown in figure 4.13. It can be seen from figure 4.13a that the 1st order of polynomial fits the runtime well.

4.4 Comparison Between Different Candidate Elimination Methods

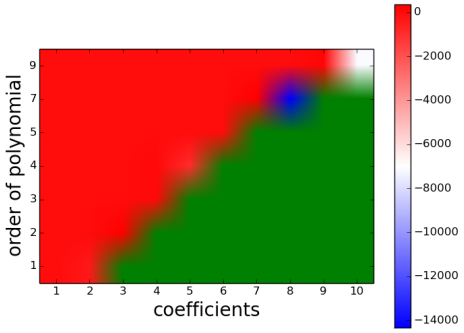
In this section, we compare the runtime of binary and prefix tree search methods for candidate elimination based on different inputs. The frequent subtree mining time is measured instead of the candidate elimination time. This is sufficient to see which method is faster based on different inputs as we only switch between these two methods during frequent subtree mining.

4.4.1 Runtime Comparison on 1000 Trees

The runtime of binary and prefix tree search methods based on different frequent subtree orders and frequency thresholds is presented in this section. The result shown in figure 4.14a is based on different orders of frequent subtrees. The result shown in figure 4.14b is based on different frequency thresholds. It can be seen



(a) mining runtime with maximum 3-*order* on different size of input



(b) coefficients of different orders of polynomial on complete graphs

Figure 4.13: different orders of polynomial and coefficients, complete graphs

that there is not much difference between their runtimes in general.

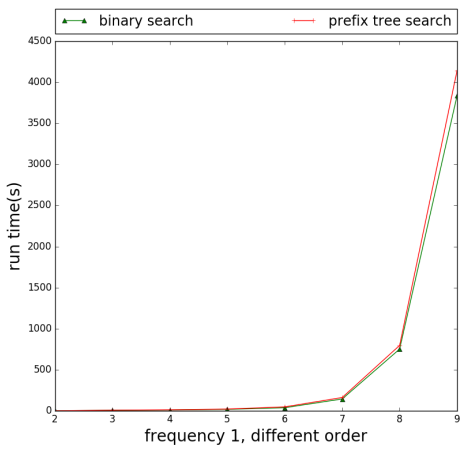
4.4.2 Runtime Comparison on 18028 Cactus Graphs

The runtime of binary and prefix tree search methods for candidate elimination on 18028 cactus graphs is given in this section. The 18028 cactus graphs are collected from the AIDS 99 dataset. The results are shown in figure 4.14c. The maximum order of frequent subtrees is set 4, and the runtime is tested based on different frequencies. It can be seen from figure 4.14c that binary search method is sometimes faster than the prefix tree search and slower at other times. This depends on the different number of vertex and edge labels in candidates and the number of candidates.

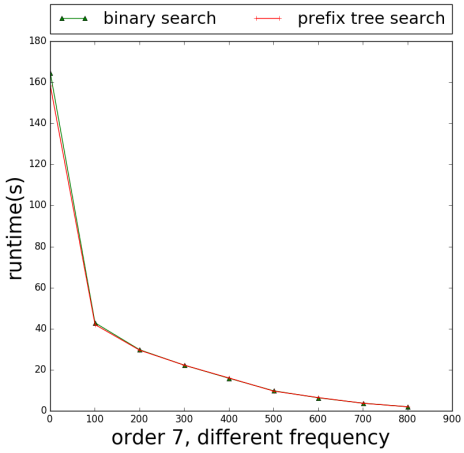
4.5 Comparison Between Implementations

4.5.1 Runtime of Our Implementations on 1000 Trees

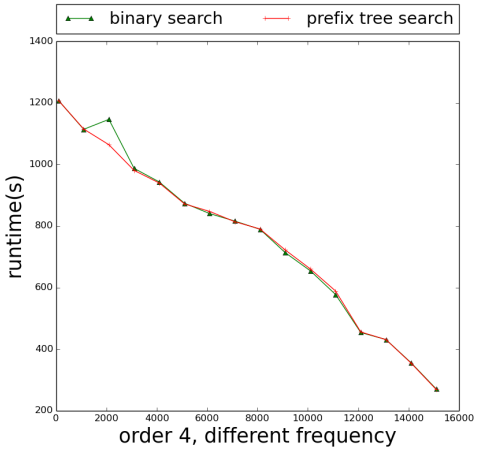
We compare our implementations of the frequent subtree mining algorithms and subtree isomorphism algorithms on 1000 AIDS trees. The frequency threshold of subtree is set to 1, and the largest number of vertices of the frequent subtrees is set to 7. Binary search is used for candidate elimination, and the mining results are show in table 4.3. The runtime of implementations is shown in figure 4.15, and each method runs for 3 times. It can be seen from figure 4.15 that the levelwise method runs faster than the depth first search method, as it filters more



(a) different order, on trees



(b) different frequency, on trees



(c) different frequency, on graphs

Figure 4.14: runtime comparison between binary and prefix tree search

Number of vertex of tree	Number of trees
1	32
2	87
3	284
4	881
5	2640
6	7472
7	19618

Table 4.3: Results

infrequent candidates and all duplicated candidates. The iterative subtree isomorphic algorithm runs faster than the non iterative one, as it iteratively uses already computed v-characteristics.

4.5.2 Runtime Comparison on 1000 Trees

We compare some frequent subtree mining tools in this section. Our tool is called *fst*. One tool (Welke, 2014) which computes frequent subtrees from trees is called *lwg*, and GASTON (Nijssen and Kok, 2004) is also compared. *fst* uses levelwise mining method by calling the two subtree isomorphism algorithms and the two candidates elimination algorithms. The comparison ran on 1000 AIDS trees, and the results are shown in figure 4.16a and 4.16b. Frequent subtree frequency threshold is set to 1 in figure 4.16a, and algorithms are compared on different orders of the frequent subtrees. It can be seen that the runtime increases with the order of frequent subtrees. In figure 4.16b, the maximum order of frequent subtrees is set to 7, and algorithms are compared on different frequency thresholds. It can be seen that the runtime decreases with increasing frequency thresholds. The iterative subtree isomorphism algorithm 11 is even slower than the non iterative version when the frequency is very large and the number of candidates is small. The reason is that the algorithm 11 computes all the v-characteristics, but the non iterative version immediately terminates once it finds subtree isomorphism.

4.5.3 Runtime Comparison on 18028 Cactus Graphs

In this section, we compare *fst* and GASTON on the 18028 cactus graphs collected from the AIDS 99 dataset. The runtime comparison between *fst* and GASTON based on different frequency thresholds is shown in figure 4.16c. The maximum order of frequent subtrees is set to 4. Our implementation can run on some large dataset, but it still runs slower than GASTON. It uses around 6.4Gb memory

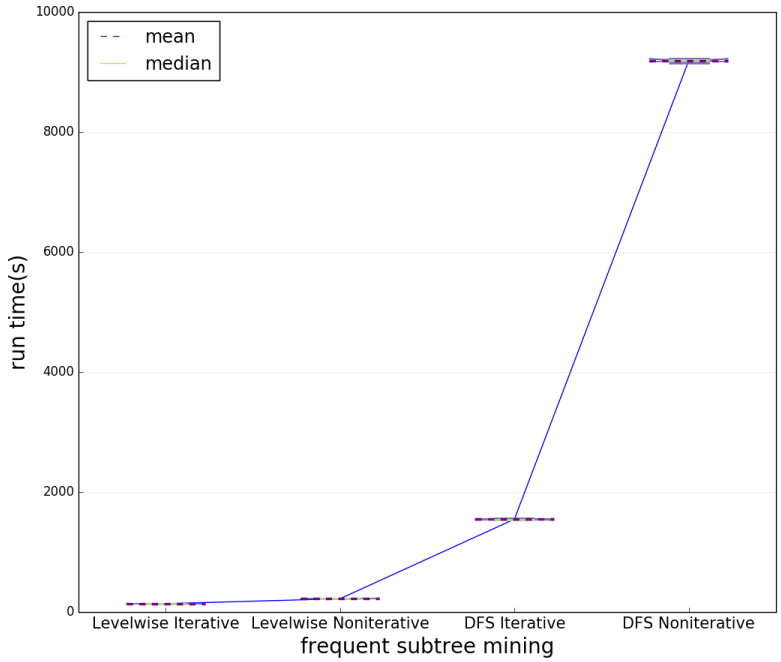
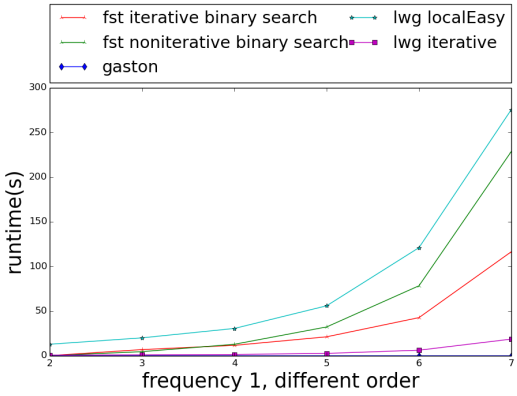
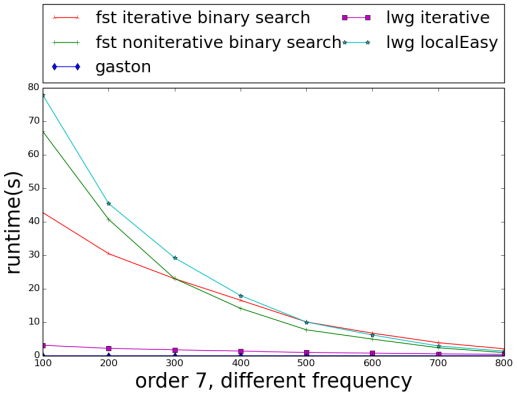


Figure 4.15: comparison on different methods

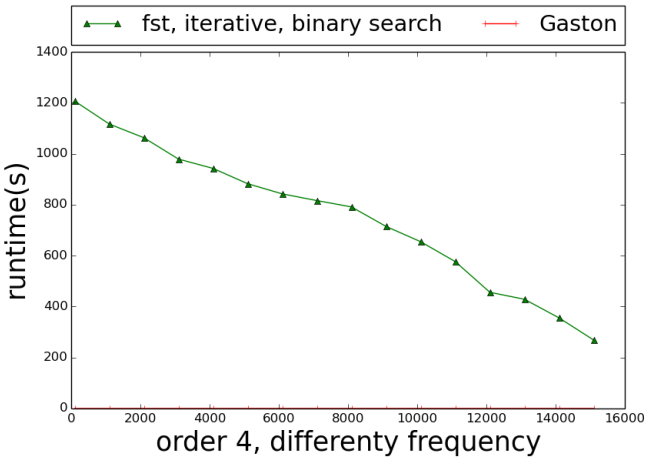
when the frequency threshold is 1 and the maximum subtree order is 4.



(a) different orders, on trees



(b) different frequency, on trees



(c) different frequency, on graphs

Figure 4.16: algorithms comparison

5 Conclusion and Future Work

5.1 Conclusion

In the previous sections, we described algorithms to solve the frequent subtree mining problem and subtree isomorphism problem. The subtree isomorphism algorithms run in polynomial time on locally easy graphs. In addition, the frequent subtree mining algorithms run in polynomial delay on locally easy graphs. Graphs with different easiness are collected and constructed to study different orders of polynomial of the runtime of the non iterative subtree isomorphism algorithm and levelwise frequent subtree mining algorithm in practice. The evaluation results are summarized in figure 5.1, and it can be seen that the subtree isomorphism algorithm has much lower orders of polynomial runtime than the worst case. The levelwise frequent subtree mining algorithm has first order of polynomial runtime on different easiness of graphs.

The evaluations of the binary and prefix tree search candidate elimination algorithms based on different inputs show that the runtimes of these two algorithms have not much difference in practice. In general, the binary search candidate elimination algorithm runs faster than the prefix tree search if the number of candidates is relatively small. The prefix tree search candidate elimination algorithm runs faster if the number of vertex and edge labels is relatively small. The levelwise frequent subtree mining algorithm runs faster than the depth first search frequent subtree mining algorithm. However, the depth first search frequent subtree mining algorithm uses less memory. In general, the iterative subtree isomorphism algorithm runs faster than the non iterative subtree isomorphism algorithm, but it runs slower when the number of candidates is relatively small.

5.2 Future Work

The value of $c \times k$ influences the runtime of the subtree isomorphism algorithms and frequent subtree mining algorithms that we evaluated. We did not find many graphs with $c' \times k' \geq 3$. If there are more graphs collected with $c' \times k' \geq 3$, then it is interesting to see the different orders of polynomial of the runtime of the non

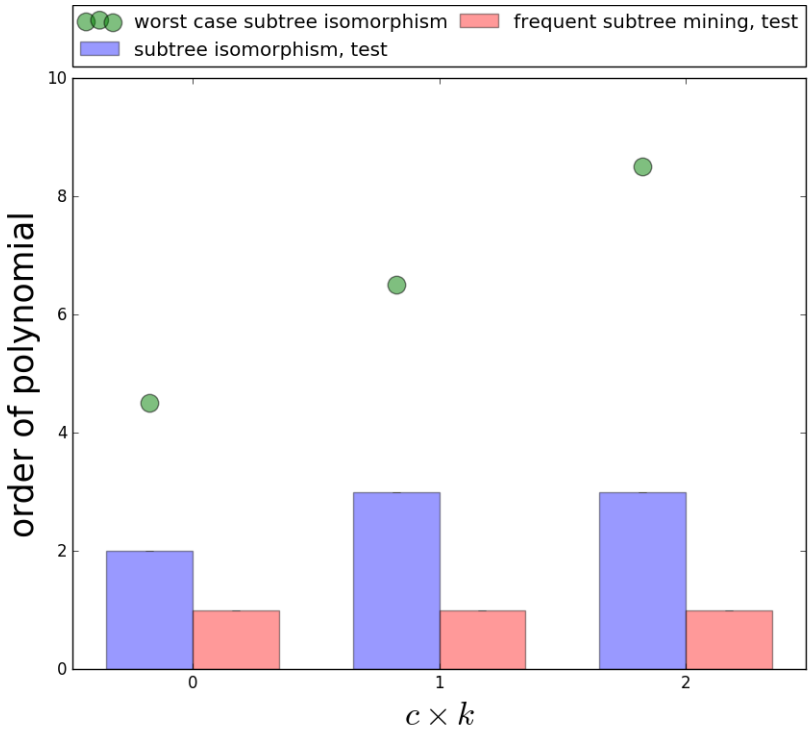


Figure 5.1: different orders of polynomial vs. different easinesses of graphs

iterative subtree isomorphism algorithm and frequent subtree mining algorithms on these graphs. Our implementation is still not efficient enough for large input and some data structures and implementation procedures can be improved further.

6 Appendix

Subtree Isomorphism in Graphs with Locally Polynomial Spanning Trees

Pascal Welke[a], Tamás Horváth[a,b], Stefan Wrobel[b,a]

[a] *University of Bonn, Germany*
[b] *Fraunhofer IAIS, Sankt Augustin, Germany*

Abstract

We show that deciding subgraph isomorphism from trees into *locally easy* graphs is in **P**, where a graph is locally easy if for any vertex v, the union of the biconnected components containing v has at most polynomially many spanning trees. This graph class properly generalizes almost $(k-1)$-trees and contains also other graphs of treewidth k for any $k > 0$. In contrast to the related positive results, we do not assume any constant upper bound on the vertex degree.

Keywords: Graph Algorithms, Subgraph Isomorphism

1. Introduction

We deal with the complexity of the following decision problem:

SubTreeIso Problem: *Given* a tree H (the *pattern*) and a graph G (the *text* graph), *decide* if there exists a subgraph isomorphism from H to G.

SubTreeIso is a well-known **NP**-complete problem (it generalizes e.g. the Hamiltonian path problem). If, however, the text graph is also a tree, it belongs to **P** (see, e.g., [7]). This positive result, together with that on generating the spanning trees of a graph with polynomial delay [6], implies that SubTreeIso is in **P** if G has polynomially many spanning trees only; just list all spanning trees τ of G and check if H is subgraph isomorphic to τ. We generalize this straightforward positive result to a broader class of text graphs that can have *exponentially* many spanning trees. More precisely, let G be a graph of order n. Then G is *locally easy* if for all vertices v of G, the union of the biconnected

components containing v has at most poly(n) spanning trees. We have the following main result for locally easy graphs:

Theorem 1. *For any tree H and for any locally easy graph G,* SubTreeIso *can be decided in polynomial time.*

We discuss this result in detail in Section 3. To prove the theorem above, in Section 2 we give an algorithm and show that it correctly solves SubTreeIso for *any* arbitrary text graph (Theorem 6) and that its runtime is polynomial for locally easy text graphs (Theorem 7). Our algorithm is inspired by the ideas in [5] and [7]. Though [5, 7] solve subgraph isomorphism for other graph classes, similarly to [5], we propose a dynamic programming algorithm directed by a rooted tree and, similarly to [7], it is based on a careful combination of the subtrees calculated in the previous levels.

Analogously to tree decompositions of bounded treewidth, our dynamic programming algorithm splits G into certain induced subgraphs defined later and evaluates partial subgraph isomorphisms from subtrees of H to such subgraphs. The bottom-up evaluation of the algorithm is, however, controlled by a rooted tree defined on the articulation vertices of G. For all nodes v of this tree, the biconnected components of v in G are replaced by a spanning tree in all possible ways and for each spanning tree τ, the subproblem corresponding to v is solved by carefully combining τ with the partial subgraph isomorphisms already computed. Iterating over all spanning trees of the biconnected components, we can correctly decide SubTreeIso for the part of G that is "below" v with respect to a vertex fixed at the beginning of the algorithm. By the condition of Theorem 1, polynomially many spanning trees must only be tested for each node v visited.

1.1. Notions and Notation

For standard concepts in graph theory, the reader is referred e.g. to [2]. Unless otherwise stated, by *graphs* we mean undirected graphs and denote the set of vertices (resp. edges) of a graph G by $V(G)$ (resp. $E(G)$). An edge $\{u, v\} \in E(G)$ will be denoted by uv and the set of neighbors of a vertex v by

$\delta(v)$. The subgraph of G induced by a set $V' \subseteq V(G)$ is denoted by $G[V']$. A *biconnected component* is a maximal subgraph of G with at least three vertices that is biconnected and a *bridge* is an edge that does not lie on any cycle. Finally, a *block* is a maximal connected subgraph that has no articulation vertex, i.e., it is either a biconnected component, or a bridge, or an isolated vertex. Two graphs G_1, G_2 are *isomorphic*, if there is a bijection $\varphi : V(G_1) \to V(G_2)$ such that $uv \in E(G_1)$ iff $\varphi(u)\varphi(v) \in E(G_2)$ for all $u, v \in V(G_1)$. G_1 is *subgraph isomorphic* to G_2, denoted $G_1 \preccurlyeq G_2$, if G_2 has a subgraph isomorphic to G_1.

2. Proof of Theorem 1

To prove Theorem 1, we give an algorithm and show that it is correct for all text graphs (Theorem 6) and that it solves the SubTreeIso problem for locally easy graphs in polynomial time (Theorem 7). The algorithm utilizes the facts that for any tree H and connected graph G, $H \preccurlyeq G$ iff H is subgraph isomorphic to a spanning tree of G and that for all $v \in V(G)$, the number of spanning trees of the subgraph induced by the set of biconnected components containing v is bounded by a polynomial of G's order. In what follows, H and G denote a tree and a locally easy graph with $|V(G)| = n$, respectively. We assume w.l.o.g. that G is connected and that $2 \leq n' \leq n$, implying that all blocks of G contain at least two vertices.

We fix an arbitrary vertex $r \in V(G)$ and will always refer to the pair of G and r, when talking about G. For a block B of G we define its *root* v to be the vertex of B with the smallest distance to r and will refer to B as a *v-rooted* block. For any $v \in V(G)$, the subgraph formed by the set of v-rooted blocks of G is denoted by $\mathcal{B}(v)$. Clearly, $\mathcal{B}(v)$ can be empty. On the set of roots of the blocks in G we define a directed graph $\mathcal{T}$ as follows (since G and r have been fixed, we omit them from the notation): For any $u, v \in V(\mathcal{T})$ with $u \neq v$, $(u, v) \in E(\mathcal{T})$ iff there exists a block B with root v such that $u \in V(B)$. In the proposition below we show that $\mathcal{T}$ is a rooted tree. This tree will be used to direct our dynamic subgraph isomorphism algorithm.

Proposition 2. *$\mathcal{T}$ is a tree rooted at r.*

Proof. It suffices to show that for all $u \in V(\mathcal{T})$ with $u \neq r$, u has outdegree at most 1; the claim then follows by noting that the outdegree of r is 0 and that $\mathcal{T}$ is connected, as G is connected. Suppose for contradiction that there exists $u \in V(\mathcal{T})$, $u \neq r$, with two different parents $v_1, v_2 \in V(\mathcal{T})$. Let $B_i \in \mathcal{B}(v_i)$ $(i = 1, 2)$. Then $B_1 \neq B_2$ and there is a path P_1 (resp. P_2) in G connecting r and v_1 (resp. v_2) that is edge disjoint with B_1 (resp. B_2). The union of P_1 and P_2 together with the paths connecting u with v_1 and u with v_2 contains a cycle intersecting both B_1 and B_2. But then u, v_1, and v_2 all belong to some block of G, contradicting the maximality of B_1 and B_2. □

We need some further concepts. Let $v, w \in V(G)$. Then w is *below* v if all paths connecting r and w in G contain v. A *rooted subgraph* G_v of G for v is the subgraph of G induced by the set of vertices below v. The same notation will be used consistently for the pair consisting of the tree pattern H and some vertex $y \in V(H)$, i.e., for any $u, y \in V(H)$, H_u^y is the tree obtained from the tree H rooted at y by keeping the subtree rooted at u. The definitions and the connectivity of G imply that G_v is connected, $G_r = G$, and G_w is a single vertex iff $w \notin V(\mathcal{T})$. A vertex $w' \in V(G)$ is called a *child* of v, if $vw' \in E(G)$ and $w' \in V(\mathcal{B}(v))$.

Let $v \in V(\mathcal{T})$ be a root. An *iso-triple*[1] ξ of H relative to v is a triple (H_u^y, τ, w) such that $u \in V(H)$, $y \in \delta(u) \cup \{u\}$, τ is a spanning tree of $\mathcal{B}(v)$, and $w \in V(\tau)$. Let G' be an induced subgraph of G and τ' be a spanning tree of G'. Then $G\{G'/\tau\}$ denotes the graph obtained from G by removing all edges of G' that are not in τ (i.e., by substituting G' with τ). Now we are able to define the partial subgraph isomorphisms we are interested in. A *v-characteristic* is an iso-triple $\xi = (H_u^y, \tau, w)$ relative to v such that there exists

[1] Though our terminology is similar to that of [3] (which, in turn, is based on the concepts in [5]), the definitions of iso-triples and characteristics in this paper are semantically different from those in [3].

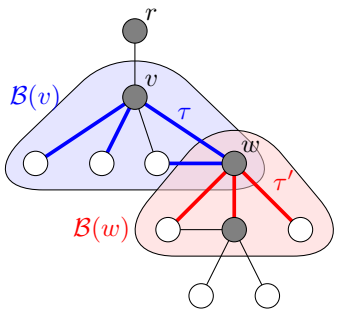


Figure 1: This figure shows a small graph G with its subgraphs $\mathcal{B}(v)$ and $\mathcal{B}(w)$ (depicted by the rounded triangles). One spanning tree τ of $\mathcal{B}(v)$ and τ' of $\mathcal{B}(w)$ are shown in red and blue, respectively.

a subgraph isomorphism φ from H_u^y to $(G\{\mathcal{B}(v)/\tau\})_w$ with $\varphi(u) = w$. In the lemma below we provide a characterization of subgraph isomorphisms from H to G in terms of v-characteristics. Its proof follows directly from the definitions.

Lemma 3. *Let H be a tree and G be a graph (with root r). Then $H \preccurlyeq G$ iff there exists a v-characteristic (H_u^u, τ, w) for some $v \in V(\mathcal{T})$, $u \in V(H)$, spanning tree τ of $\mathcal{B}(v)$, and $w \in V(\tau)$.*

Notice that the number of v-characteristics (H_u^y, τ, w) is bounded by $\mathrm{poly}(n)$ for all $w \in V(G)$ if G is a locally easy graph. We will show how these characteristics can be computed recursively by a post-order traversal of $\mathcal{T}$. In order to recover all v-characteristics, the spanning trees of the w-rooted blocks must carefully be combined with τ when w itself is also a root (i.e., $w \in V(\mathcal{T})$). To formalize these considerations, we introduce the following notation. Let $v \in V(\mathcal{T})$ and $\mathcal{S}_v$ be the set of spanning trees of $\mathcal{B}(v)$. Then for any $\tau \in \mathcal{S}_v$ and $w \in V(\tau)$ we define $\Theta_{vw}(\tau)$ by

$$\Theta_{vw}(\tau) := \begin{cases} \{\tau \cup \tau' \ : \ \tau' \in \mathcal{S}_w\} & \text{if } w \in V(\mathcal{T}_G) \setminus \{v\} \\ \{\tau\} & \text{o/w (i.e., if } w \notin V(\mathcal{T}_G) \text{ or } v = w), \end{cases}$$

where $\tau \cup \tau'$ is the graph with vertex set $V(\tau) \cup V(\tau')$ and edge set $E(\tau) \cup E(\tau')$.

That is, for the case that $w \in V(\mathcal{T}) \setminus \{v\}$, $\Theta_{vw}(\tau)$ is the set of trees obtained by "gluing" τ and τ' at vertex w, for all spanning trees τ' of $\mathcal{B}(w)$. The definition is correct, as $V(\tau) \cap V(\tau') = \{w\}$ for this case. Note that if w is a root vertex different from v then it always has at least one child in $\mathcal{B}(w)$, i.e., τ' is always a tree with at least one edge. As an example, the combination of the blue and the red tree in Fig. 1 denotes an element of $\Theta_{vw}(\tau)$. In Lemma 4 below we first provide a characterization of v-characteristics for subtrees H_u^y with $y \in \delta(u)$.

Lemma 4. *An iso-triple (H_u^y, τ, w) of H is a v-characteristic for some $v \in V(\mathcal{T})$ and $y \in \delta(u)$ iff there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\delta(u) \setminus \{y\}$ to the children of w in θ such that for all $u' \in \delta(u) \setminus \{y\}$ there is a subgraph isomorphism $\varphi_{u'}$ from $H_{u'}^u$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\psi(u')}$ mapping u' to $\psi(u')$.*

Proof. "$\Rightarrow$" Suppose (H_u^y, τ, w) is a v-characteristic for some $v \in V(\mathcal{T})$ and $y \in \delta(u)$. Then, by definition, there is a subgraph isomorphism φ from H_u^y to $(G\{\mathcal{B}(v)/\tau\})_w$ with $\varphi(u) = w$. Let R be an arbitrary spanning tree of $(G\{\mathcal{B}(v)/\tau\})_v$ containing the image $\varphi(H_u^y)$ as a subtree. Then $R[V(\mathcal{B}(w))] \in \mathcal{S}_w$ and $R[V(\mathcal{B}(v))] = \tau$ and hence $\theta = R[V(\mathcal{B}(v))] \cup R[V(\mathcal{B}(w))] \in \Theta_{vw}(\tau)$ implying that for all $u' \in \delta(u) \setminus \{y\}$, φ maps $H_{u'}^u$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\varphi(u')}$. As φ is injective we can set ψ to be the restriction of φ to $\delta(u) \setminus \{y\}$ and φ' to be the restriction of φ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\varphi(u')}$.

"$\Leftarrow$" Let $\varphi : V(H_u^y) \to V(G(\{\mathcal{B}(v)/\tau\})_w)$ with $\varphi : u \mapsto w$ and $v' \mapsto \varphi_{u'}(v')$ for all $u' \in \delta(u) \setminus \{y\}$ and $v' \in V(H_{u'}^u)$. Since for all u', $\varphi_{u'}$ is at the same time a subgraph isomorphism from $H_{u'}^u$ to $G(\{\mathcal{B}(v)/\tau\})_w$, $\varphi_{u'}(u') = \psi(u')$. But then, as ψ is injective, φ is a subgraph isomorphism, implying the claim. □

In Lemma 5 we formulate an analogous characterization for the entire pattern H (i.e., for $y = u$). The proof of this lemma is similar to that of Lemma 4.

Lemma 5. *An iso-triple (H_u^u, τ, w) of H is a v-characteristic for some $v \in V(\mathcal{T})$ iff there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\delta(u)$ to the children of w in θ such that for all $u' \in \delta(u)$ there is a subgraph isomorphism*

φ' *from* $H^u_{u'}$ *to* $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\psi(u')}$ *mapping* u' *to* $\psi(u')$.

Theorem 6 below is concerned with the correctness of Algorithm 1. Notice that the claim holds for any connected text graph.

Theorem 6 (Correctness)**.** *Algorithm 1 is correct, i.e., for all trees* H *and connected graphs* G *with* $2 \leq |V(H)| \leq |V(G)|$*, it returns* True *iff* $H \preccurlyeq G$.

Proof. Algorithm 1 first fixes a root r of G (Line 1) and, by traversing the tree $\mathcal{T}$ (rooted at r) in a postorder manner (Line 2), it calculates the set $\mathcal{C}$ of v-characteristics for all $v \in V(\mathcal{T})$ (Lines 4–8). We only need to show that $\mathcal{C}$ is correct (i.e., complete and sound); the correctness of the algorithm then follows directly from Line 9 together with Lemma 3.

The completeness of $\mathcal{C}$ holds by the fact that all possible iso-triples $\xi = (H^y_u, \tau, w)$ relative to v are tested for being v-characteristics (Lines 3–7 of Main). Thus, it remains to show that it is decided correctly whether or not $\xi = (H^y_u, \tau, w)$ is a v-characteristic. We prove this by double induction on the height $h_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ and on the height $h_\tau(w)$ of w in τ. Depending on whether or not $h_\tau(w) = 0$ and $h_{\mathcal{T}}(v) = 0$, four cases can be distinguished. We only show the base case (i.e., $h_{\mathcal{T}}(v) = h_\tau(w) = 0$) and the most general case (i.e., $h_{\mathcal{T}}(v) > 0$ and $h_\tau(w) > 0$) by noting that the proofs of the other two cases can be shown by an argumentation similar to the one used for the most general case.

For the base case $h_{\mathcal{T}}(v) = h_\tau(w) = 0$ we have $C_\theta = \emptyset$ and hence $B = \emptyset$ (Lines 3 and 4 of Characteristics). Applying Lemma 4 to this case, ξ is a v-characteristic iff $\delta(u) = \{y\}$, which, in turn, holds iff there is a matching covering $\delta(u) \setminus \{y\}$ in B (Lines 5–6).

If $h_{\mathcal{T}}(v) > 0$ and $h_\tau(w) > 0$ then $C_\tau \neq \emptyset$. Two cases can be distinguished: (i) If $w \notin V(\mathcal{T})$ then $C_{\tau'} = \emptyset$ and thus $C_\theta = C_\tau$. Applying Lemma 4 to this case, ξ is a v-characteristic iff there exists an injective function $\psi : \delta(u) \setminus \{y\} \to C_\tau$ such that for all $u' \in \delta(u) \setminus \{y\}$, there exist a child c of w in τ (i.e., $c \in C_\tau$) and a subgraph isomorphism $\varphi_{u'}$ from $H^u_{u'}$ to $(G\{\mathcal{B}(v)/\tau\})_c$ with $\varphi_{u'}(u') = \psi(u') = c$ (i.e., a v-characteristic $(H^u_{u'}, \tau, c)$). By the induction hypothesis, the bipartite graph B is constructed correctly in Line 4, and hence ψ exists

iff there exists a matching in B covering $\delta(u) \setminus \{y\}$. (ii) If $w \in V(\mathcal{T})$ then $C_\theta = C_\tau \cup C_{\tau'}$ with $C_\tau, C_{\tau'} \neq \emptyset$. Then, by Lemma 4, ξ is a v-characteristic iff for all $u' \in \delta(u) \setminus \{y\}$ there exist a child c of w in θ and an injective function $\psi : \delta(u) \setminus \{y\} \to C_\tau \cup C_{\tau'}$ such that there is a subgraph isomorphism $\varphi_{u'}$ from $H^u_{u'}$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\tau \cup \tau'\})_c$ with $\varphi_{u'}(u') = \psi(u') = c$. Such a subgraph isomorphism either corresponds to a v-characteristic $(H^u_{u'}, \tau, c)$ for $c \in C_\tau$, which has already been computed by the induction hypothesis on $h_\tau(w)$, or to a w-characteristic $(H^u_{u'}, \tau', c)$ for $c \in C_{\tau'}$, which has already been computed by the induction hypothesis on $h_{\mathcal{T}}(v)$. Hence ψ exists iff a matching in B (constructed in Line 4) covering $\delta(u) \setminus \{y\}$ exists (Lines 5–6). The proof for the v-characteristics (H^u_u, τ, w) using Lemma 5 is analog. □

Theorem 7 (Runtime)**.** *For locally easy text graphs, Algorithm 1 runs in polynomial time.*

Proof. By condition, there is a constant $k \geq 0$ such that $|\mathcal{S}_v|$ is $O\left(n^k\right)$ for all $v \in V(\mathcal{T})$. Thus, as the spanning trees of a graph can be generated with linear delay [6], $\mathcal{S}_v$ can be computed in $O\left(n^{k+1}\right)$ time. Hence Main spends altogether

$$O\left(n^{k+2}\right) \tag{1}$$

time on calculating the sets of spanning trees for the nodes in $\mathcal{T}$. Furthermore, it calls subroutine Characteristics

$$O\left(n^{k+2}\right) \tag{2}$$

times. This is because H is a tree with $O(n)$ vertices and therefore, the number of pairs (u, y) (Lines 6–7) is $O(n)$ and, because of a similar argument, the number of pairs (v, w) (Lines 2 and 5) is also $O(n)$. Regarding the complexity of Characteristics, note that $|\Theta_{vw}(\tau)|$ is bounded by $O\left(n^k\right)$ (see Line 1 of Characteristics) and that the bipartite graph B constructed in Line 4 has at most $|\delta(w)| + |\delta(u)|$, i.e., $O(n)$ vertices for any $\theta \in \Theta_{vw}(\tau)$. The edges of B can be constructed by $O\left(n^2\right)$ membership queries to $\mathcal{C}$. We can implement the set $\mathcal{C}$ of characteristics found by the algorithm as a multidimensional array of

polynomial size such that each look-up and storage operation can be performed in constant time. A maximum matching of B can be found in $O\left(n^{5/2}\right)$ time [4]. (Note that we can check for the existence of a matching covering all but u' by deleting u' from B in Line 5.) Thus, each call of CHARACTERISTICS needs $O\left(n^k\left(n^2+n^{5/2}\right)\right)$, i.e.,

$$O\left(n^{k+5/2}\right) \tag{3}$$

time. Putting (1–3) together and using that (3) dominates (1), the overall runtime of Algorithm 1 is bounded by $O\left(n^{2k+9/2}\right)$. □

3. Remarks

A sufficient condition of local easiness can be obtained by combining the following two conditions: Let G be a graph of order n and $c, k \geq 0$ be constants. Then (i) G is a *k-easy* graph if each biconnected component of G has at most $O(n^k)$ spanning trees and (ii) G is of bounded *cyclic block degree* if each vertex v of G belongs to at most c distinct biconnected components. We note that none of the conditions alone suffices to resolve the intractability of the SUBTREEISO problem. In particular, if only (i) holds then the problem remains **NP**-complete even for 0-easy text graphs; this can be shown by the same reduction used in [1] to prove that SUBTREEISO is **NP**-complete even for almost 0-trees. If only (ii) holds then SUBTREEISO is **NP**-complete even for $c = 1$; this follows directly from the intractability of the Hamiltonian path problem for biconnected graphs.

The class of locally easy graphs properly contains that of almost k-trees of bounded degree. Thus, Theorem 1 generalizes the positive result of [1] on almost k-tree text graphs for tree patterns. Furthermore, there are locally easy graphs (in fact even 1-easy graphs with cyclic block degree 1) that neither have a bounded treewidth nor are almost k-trees of bounded degree, as shown in the example below.

Example 8. *Let $k \geq 4$ be a fixed integer and K_{k+1} be the complete graph on $k+1$ vertices. We create a graph G by adding $(k+1)^{k-1} - k - 1$ new bridges to an arbitrary vertex v of K_{k+1}. The only biconnected component K_{k+1} of G has*

treewidth k and $|E(K_{k+1}| = \binom{k+1}{2} > 2k+1$. Thus, on the one hand, G is only an almost k-tree. The degree of v is $(k+1)^{k-1} - 1 = |V(G)| - 1$. On the other hand, K_{k+1} has exactly $(k+1)^{k-1}$ spanning trees. Since $|V(G)| = (k+1)^{k-1}$, the number of spanning trees of the only biconnected component is $O(|V(G)|)$, implying that G is a 1-easy graph with cyclic block degree 1.

This example shows that local easiness neither implies a constant bound on the vertex degree (required, e.g., in [1, 5]) nor log-bounded fragmentation [3]. However, the vertex degree is an upper bound on the cyclic block degree and log-bounded fragmentation implies an $O(\log(n))$-bound on the cyclic block degree.

4. References

[1] T. Akutsu. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE trans. on fundamentals of electronics, communications and computer sciences*, 76(9):1488–1493, 1993.

[2] R. Diestel. *Graph Theory*, volume 173. Springer, 2012.

[3] M. Hajiaghayi and N. Nishimura. Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. *J. Comput. Syst. Sci.*, 73(5):755–768, 2007.

[4] J. E. Hopcroft and R. M. Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[5] J. Matoušek and R. Thomas. On the complexity of finding iso-and other morphisms for partial k-trees. *Discrete Mathematics*, 108(1):343–364, 1992.

[6] R. C. Read and R. Tarjan. Bound on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.

[7] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Theory of Computing and Systems, 1997*, pages 126–131. IEEE, 1997.

Fast Frequent Subtree Mining in Graph Databases with Locally Polynomial Spanning Trees

Pascal Welke

Informatik III, University of Bonn, Germany

1 Introduction

This document is based on our article that we submitted to Information Processing Letters. Its goal is to find a new version of the subgraph isomorphism algorithm for trees in graphs of locally bounded spanning trees that gives practical speedups in the use case of frequent subgraph mining in either an apriori- or FPgrowth-style mining algorithm.

2 Notation and Preliminaries

We use the same notation as in our paper [5]. Particularly, we also extend the algorithm mentioned there. Algorithm 1 is a slight modification of Algorithm 1 from [5], with the only change, that we added a variable X that stores if a subgraph isomorphism was found, instead of early termination. This does not change the asymptotic worst time complexity of the algorithm but might increase the runtime in the case that a subgraph isomorphism is found. However, to validate that a subgraph isomorphism does not exist always requires a complete evaluation of all characteristics / iso-tuples.

We further note that the requirement in [5] that $|V(H| > 1$ is not necessary for the correctness of the algorithm and hence can be omitted.

3 Unblackboxing the Subtree Isomorphism Algorithm

The question we want to address in this section is the following: How can we answer a subgraph isomorphism query for a given tree H pattern and locally easy transaction graph G fast, if we know that all or some subgraphs of H are subgraph isomorphic to H? The answer to this question can speed up frequent subtree mining in this class of graphs dramatically, as

1. all subtrees of H have already been processed once the mining algorithm evaluates H in the case of an apriori-style frequent subgraph mining algorithm, and
2. at least one subgraph of H with $|V(H)| - 1$ vertices has already been evaluated in the case of an FP-growth style frequent subgraph mining algorithm.

The dynamic programming subgraph isomorphism algorithm developed in our previous work [5] exactly does this kind of operation internally. It builds (partial) subgraph isomorphisms by combining partial subgraph isomorphisms from smaller subgraphs of the pattern into the transaction graph. Looking at the mining algorithm described in [4] we notice that we use the subgraph isomorphism algorithm as a blackbox to decide if an extension H (which is obtained by adding a single new edge e to a single additional vertex) of a frequent pattern $H' := H - e$ is subgraph isomorphic to some transaction graph G. For doing this, internally the subgraph isomorphism algorithm recomputes the – now partial – subgraph isomorphism

Algorithm 1 Subgraph Isomorphism from a Tree into a Connected Graph

Input : tree H and an arbitrary connected graph G with $|V(H)| \leq |V(G)|$
Output: TRUE if $H \preccurlyeq G$; o/w FALSE

MAIN:

1: set $X :=$ FALSE
2: pick a vertex $r \in V(G)$ and set $\mathcal{C} := \emptyset$
3: **for all** $v \in V(\mathcal{T})$ in a postorder **do**
4: let $\mathcal{S}_v$ be the set of spanning trees of $\mathcal{B}(v)$, each rooted at v
5: **for all** $\tau \in \mathcal{S}_v$ **do**
6: **for all** $w \in V(\tau)$ in a postorder **do**
7: **for all** $u \in V(H)$ **do**
8: **for all** $y \in \{u\} \cup \delta(u)$ **do**
9: $\mathcal{C} := \mathcal{C} \cup$ CHARACTERISTICS(v, u, y, τ, w)
10: **if** $(H_u^u, \tau, w) \in \mathcal{C}$ **then** set $X :=$ TRUE
11: **return** X

FUNCTION CHARACTERISTICS(v, u, y, τ, w):

1: **for all** $\theta \in \Theta_{vw}(\tau)$ **do**
2: let τ' be the tree satisfying $\theta = \tau \cup \tau'$
3: let C_τ (resp. $C_{\tau'}$) be the set of children of w in τ (resp. τ') and $C_\theta := C_\tau \cup C_{\tau'}$
4: let $B = (C_\theta \dot\cup \delta(u), E)$ be the bipartite graph with $cu' \in E$ iff $(c \in C_\tau \wedge (H_{u'}^u, \tau, c) \in \mathcal{C}) \vee (c \in C_{\tau'} \wedge (H_{u'}^u, \tau', c) \in \mathcal{C})$ for all $cu' \in C_\theta \times \delta(u)$
5: **if** $y \neq u$ and B has a matching covering $\delta(u) \setminus \{y\}$ **then**
6: **return** $\{(H_u^y, \tau, w)\}$
7: **if** $y = u$ and B has a matching that covers $\delta(u)$ **then**
8: **return** $\{(H_u^u, \tau, w)\}$

from H' to G to then combine it with the single additional edge. This – thinking about it this way – seems very inefficient, doesn't it? Hence, storing the already computed partial subgraph isomorphisms and reusing them when we need to decide subgraph isomorphism for an extension should yield an iterative subgraph isomorphism checker that ideally answers the questions $H_i \preccurlyeq G$ for some sequence of extensions $H_0 \preccurlyeq H_1 \preccurlyeq \ldots \preccurlyeq H_{n-1} = H' \preccurlyeq H_n = H$ in the same asymptotic time than just deciding $H \preccurlyeq G$.

The idea of unrolling a dynamic programming algorithm in such a way has already been used in the case of frequent rooted undordered subtree mining in databases of rooted unordered trees [3, 1] to speed up the (polynomial delay) mining process. It was also applied to mine frequent subgraphs in databases of bounded treewidth graphs [2], where it yields an incremental polynomial delay algorithm although subgraph isomorphism is NP-complete for bounded treewidth graphs. While the former two examples show that practical speedups can be gained by this idea, the latter shows that this technique also has benefits that render a mining problem "feasible", although the corresponding subgraph isomorphism check is infeasible.

3.1 Transforming $(H-e)$-characteristics to H-characteristics

In this section we analyze how the embedding information computed for a subgraph $H-e$ can be reused for the full graph H. This will finally result in an algorithm that can be applied in the breadth- and depth-first-search frequent subgraph mining algorithms, as it does not require the embedding information for all subgraphs of H, but only for a particular arbitrary one.

Let G be a graph, H be a tree, and $e = \{a, b\} \in E(H)$ some edge with $|\delta(b)| = 1$. Let (H_u^y, τ, w) be a v-characteristic with respect to H and G. We consider the situation of Lemma 5 and Lemma 6 in [5] and will obtain insight how the v-characteristics (H_u^y, τ, w) are related to the v-characteristics $((H-e)_u^y, \tau, w)$. Several cases arise that are depicted in Figure 1 (Lemma 5) and Figure 2 (Lemma 6). The two figures show only H and the vertices u, y, a, b in their positions. Triangles are meant to depict subtrees. Recall that a v-characteristic (H_u^y, τ, w) describes a partial isomorphism from H_u^y (the subgraph induced by all vertices below u in H^y) such that u is mapped to w in $(G\{\mathcal{B}(v)/\tau\})_w^v$.

When we extend $H-e$ by e to H, we add a single edge and a single new vertex. Hence, existing partial isomorphisms with respect to $H-e$ are good starting points to generate partial isomorphisms with respect to H: They tell us if there is a partial isomorphism from a subgraph of $H-e$ to some subgraph of G. However, characteristics containing the new vertex b were not considered for $H-e$ and need to be handled separately.

Hence, we distinguish two main cases, depending on whether the new vertex b explicitly occurs in one of the positions u or y of a v-characteristic.

1. In the case that $b \notin \{y, u\}$, a characteristic with respect to H always requires an existing characteristic with respect to $H-e$: if there is no subgraph isomorphism from $(H-e)_u^y$ to $(G\{\mathcal{B}(v)/\tau\})_w^v$, then there cannot be a subgraph isomorphism from its supergraph H_u^y to $(G\{\mathcal{B}(v)/\tau\})_w^v$.
   (a) Suppose $y \neq u$ and a is not below u in H_u^y (Figure 1a and Figure 1b). It follows that $H_u^y = (H-e)_u^y$, hence it holds that (H_u^y, τ, w) is a v-characteristic if and only if $((H-e)_u^y, \tau, w)$ is a v-characteristic.
   (b) Suppose $y \neq u$ and a is below u in H_u^y (Figure 1c and Figure 1d). In this case, $(H-e)_u^y$ is a proper rooted subtree of H_u^y and hence (H_u^y, τ, w) is a v-characteristic $\Rightarrow ((H-e)_u^y, \tau, w)$ is a v-characteristic. This means that we only need to check the existing characteristics (with respect to $H-e$) for validity when extending it by e to H. We will describe below how this can be done efficiently.
   (c) Suppose $y = u$. Then a is below u in $H_u^y = H_u^u$ (Figure 2a and Figure 2b) and – as in Case 1b – we need to check the existing v-characteristic $((H-e)_u^u, \tau, w)$ (resp. $((H-e)_a^a, \tau, w)$) for validity.
2. In the case that $b \in \{u, y\}$, a corresponding characteristic with respect to $H-e$ does not exist and new characteristics may arise:
   (a) $y = a$ and $u = b$: (Figure 1e) Then there is always a new v-characteristic (H_b^a, τ, w).
   (b) $y = b$ and $u = a$: (Figure 1f) There is a new v-characteristic (H_a^b, τ, w) if and only if there is a v-characteristic $((H-e)_a^a, \tau, w)$.
   (c) $u = y = b$: (Figure 2c) Then there is a v-characteristic (H_b^b, τ, w) if and only if there is either a v-characteristic (H_a^b, τ, c) for some child of w in τ or there is a w-characteristic (H_a^b, τ', c) for some spanning tree τ' of $\mathcal{B}(w)$ and some child of w in τ'. This is the counterpart to Case 2b directly above, as any of these two conditions can only hold if there is a v-characteristic $((H-e)_a^a, \tau, c)$ or a w-characteristic $((H-e)_a^a, \tau', c)$.

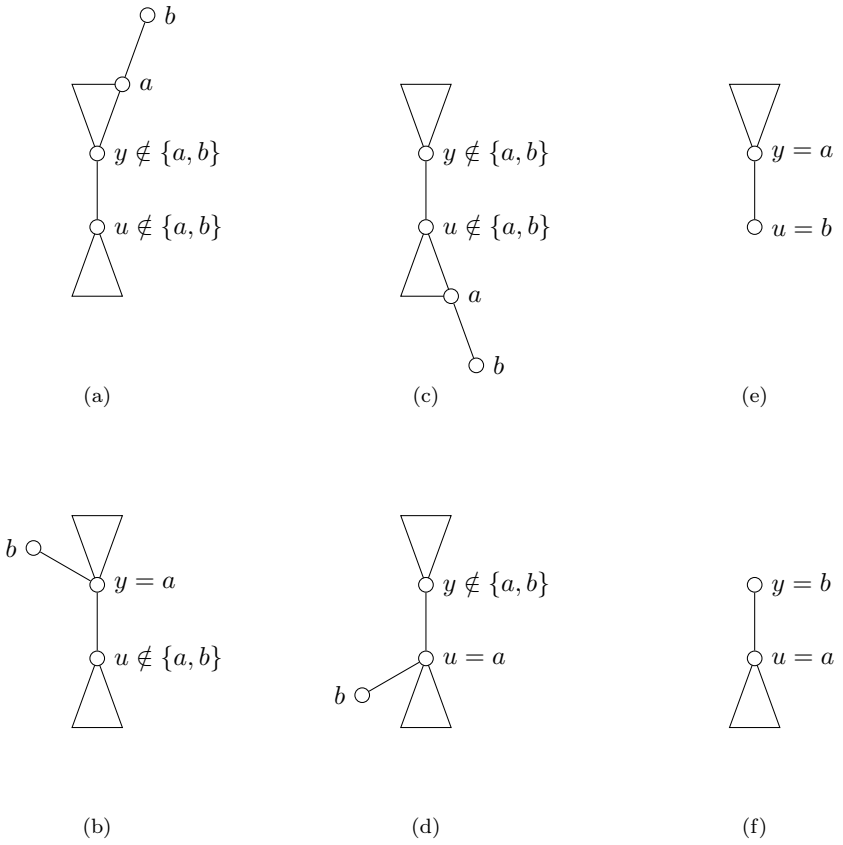


Figure 1: The six cases of H_u^y for $u \neq y$.

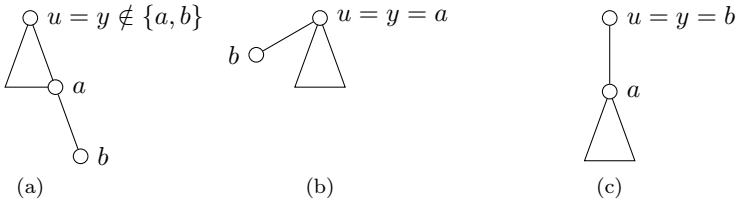


Figure 2: The three cases of H_u^u for $u = y$.

We have seen all the possible scenarios for a and b occupying the positions in a v-characteristic with respect to H. In the cases where b takes one of the positions of u or y, we know exactly when there is a v-characteristic with respect to H based on the v-characteristics based on $H-e$. The tests for Case 1a, Case 2a and Case 2b can be done in constant time per characteristic, as we will see below. It thus only remains to show how we can check the validity of a v-characteristic with respect to $H - e$ for the Cases 1b, 1c, and 2c.

3.2 A Less Naïve Frequent Subtree Mining Algorithm

The previous section gives several insights that can be used to speed up Algorithm 1 if the set of v-characteristics with respect to some $H - e$ is part of its input. In this section, we will describe this algorithm in detail. In Section 3.3, we will give an algorithm for the base case that the pattern graph H consists of a single vertex. Together, the two algorithms solve the subgraph isomorphism problem for a tree into any connected graph correctly, as we will see in Section **??**.

1. If $u, y \in V(H - e)$, then we only need to process iso triples (H_u^y, τ, w) for which a corresponding v-characteristic $((H - e)_u^y, \tau, w)$ exists. This might result in significantly less calls to Characteristics, as iso triples that have already been found not to be v-characteristics do not need to be checked again.

2. If we are in Case 1a, we can omit the call to Characteristics and just add (H_u^y, τ, w) if $((H - e)_u^y, \tau, w)$ exists. The question whether a is below u in H_u^y can be answered in constant time per vertex: We just compute the unique parent $p_a(u)$ of each vertex u in H^a up front. Now a is not below u in H_u^y if and only if $y = p_a(u)$.

3. If b appears in a iso triple, we can decide in constant time for Case 2a and Case 2b whether we add the characteristic or not. For Case 2c, we need to call Characteristics once.

Algorithm 2 implements these ideas. Hence we obtain the following result:

Lemma 1. *Algorithm 2 computes the set of v-characteristics of H and G correctly, given correct input.*

Proof. If the algorithm adds an iso-tuple to its output, this tuple is a v-characteristic: $H_b^a, \tau, w)$ (Line 9) is always a v-characteristic (see Case 2a). The addition in Line 10 is correct due to the argumentation in Case 2b, the addition in Line 15 is correct due to Case 1a. The possible additions in Line 11 and 16 are done using the method Characteristics from Algorithm 1. From the proof of the correctness of Algorithm 1 in [5] we know that Algorithm 1 is sound and complete. Characteristics checks for the existence of a characteristic by solving a matching instance in a bipartite graph where the edges correspond to already computed characteristics. Hence, by induction, it follows that Characteristics invoked by Algorithm 2 on a subset of characteristics cannot output an iso-tuple that was not output by some invocation of Characteristics in Algorithm 1.

For the other direction, we need to ensure that we do not skip a test for any iso-tuple that might be a characteristic. We note, that the outermost three loops of Algorithm 2 are identical to those of Algorithm 1. Hence, the new algorithm maintains the order of processing that ensures that all children of the current vertex w in $G\{\mathcal{B}(v)/\tau\}$ were already visited. The outer loops visit all possible combinations of v, τ, and w. Hence, all iso-tuples containing the new vertex b are checked in Line 9–11. For $u \neq b$ and $y \neq b$, we only consider those iso-tuples (H_u^y, τ, w) where there is a v-characteristic $((H - e)_u^y, \tau, w) \in \mathcal{C}_{H-e}$, which is sufficient as a

Algorithm 2 Subgraph Isomorphism from a Tree into a Connected Graph Reusing Information from a Subtree

Input : A tree H, an arbitrary connected graph G with $2 \leq |V(H)| \leq |V(G)|$, $r \in V(G)$, and the set $\mathcal{C}_{H-e}$ of all characteristics for some $e = \{a, b\}$ where b is a leaf
Output: TRUE if $H \preccurlyeq G$; o/w FALSE and the set $\mathcal{C}$ of all characteristics of H

MAIN:

set $X :=$ FALSE
set $\mathcal{C} := \emptyset$
let $p_a(u)$ be the uniqe parent in H^a of all $u \in V(H)$
for all $v \in V(\mathcal{T})$ in a postorder **do**
    let $\mathcal{S}_v$ be the set of spanning trees of $\mathcal{B}(v)$, each rooted at v
    for all $\tau \in \mathcal{S}_v$ **do**
        for all $w \in V(\tau)$ in a postorder **do**
            // Add new characteristics
            add (H_b^a, τ, w) to $\mathcal{C}$
            if $(H_a^a, \tau, w) \in \mathcal{C}_{H-e}$ **then** add (H_a^b, τ, w) to $\mathcal{C}$
            $\mathcal{C} := \mathcal{C} \cup$ CHARACTERISTICS(v, b, b, τ, w)
            if $(H_b^b, \tau, w) \in \mathcal{C}$ **then** set $X :=$ TRUE
            // Filter existing characteristics (wrt. $H - e$)
            for all $(H_u^y, \tau, w) \in \mathcal{C}_{H-e}$ **do**
                if $y = p_a(u)$ **then** add (H_u^y, τ, w) to $\mathcal{C}$
                else $\mathcal{C} := \mathcal{C} \cup$ CHARACTERISTICS(v, u, y, τ, w)
                if $(H_u^u, \tau, w) \in \mathcal{C}$ **then** set $X :=$ TRUE
return X and $\mathcal{C}$

characteristic with respect to H always requires an existing characteristic with respect to $H - e$ as we have seen in Case 1. □

3.3 Initialization for Singleton Patterns

Algorithm 2 requires the characteristics of a subtree $H - e$ of some tree H as part of its input. Hence, we require an additional algorithm that can compute characteristics for simple pattern trees. For singleton pattern graphs (i.e., a tree that only consists of a single vertex), this is rather easy. Lemma 2 gives the insight, Algorithm 3 the resulting algorithm.

Lemma 2. *Let H be a tree containing the single vertex 0 and G be a connected graph. Then there is a v-characteristic (H_0^0, τ, w) for all $v \in V(\mathcal{T})$, $w \in V(\mathcal{B}(v))$ and $\tau \in \mathcal{S}_v$.*

Proof. By definition [5], a v-characteristic (H_0^0, τ, w) is equivalent to a subgraph isomorphism φ from H_0^0 to $(G\{\mathcal{B}(v)/\tau\})_w$ with $\varphi(0) = w$. However, H_0^0 is a single vertex and hence can be mapped to w in any $(G\{\mathcal{B}(v)/\tau\})_w$, as this subgraph contains w. □

Note that Algorithm 3 always returns true for unlabeled graphs, as the singleton vertex of H matches any vertex in G. In the labeled case, this might not be the case: There might only exist characteristics for some vertices of G. However, the existence of a v-characteristic (H_0^0, τ, w) still only depends on 0 and w, not on v and τ.

Algorithm 3 Subgraph Isomorphism from a Tree into a Connected Graph Reusing Information from a Subtree – Initialization Algorithm

Input : A tree H, an arbitrary connected graph G with $1 = |V(H)| \leq |V(G)|$, $r \in V(G)$.
Output: TRUE if $H \preccurlyeq G$; o/w FALSE and the set $\mathcal{C}$ of all characteristics of H

MAIN:

set $X :=$ FALSE
set $\mathcal{C} := \emptyset$
let $p_a(u)$ be the uniqe parent in H^a of all $u \in V(H)$
for all $v \in V(\mathcal{T})$ in a postorder **do**
    let $\mathcal{S}_v$ be the set of spanning trees of $\mathcal{B}(v)$, each rooted at v
    for all $\tau \in \mathcal{S}_v$ **do**
        for all $w \in V(\tau)$ in a postorder **do**
            add (H_0^0, τ, w) to $\mathcal{C}$
return X and $\mathcal{C}$

References

[1] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *Discovery Science, 6th International Conference, DS 2003, Sapporo, Japan, October 17-19,2003, Proceedings*, pages 47–61, 2003.

[2] T. Horváth and J. Ramon. Efficient frequent connected subgraph mining in graphs of bounded tree-width. *Theor. Comput. Sci.*, 411(31-33):2784–2797, 2010.

[3] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First international workshop on mining graphs, trees and sequences*, volume 2003. Citeseer, 2003.

[4] P. Welke, T. Horváth, and S. Wrobel. On the complexity of frequent subtree mining in very simple structures. *Proceedings of Inductive Logic Programming 2014*, 2015.

[5] P. Welke, T. Horváth, and S. Wrobel. Subtree isomorphism in graphs with locally polynomial spanning trees. *under review for Information Processing Letters*, September 2015.

Bibliography

1000 AIDS trees. [Online; accessed 15-Oct-2016]. URL: https://github.com/liyakun/pattern-recognition/blob/master/1000aidstrees.txt (cit. on p. 43).

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo (1996). "Advances in Knowledge Discovery and Data Mining". In: ed. by U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. Menlo Park, CA, USA: American Association for Artificial Intelligence. Chap. Fast Discovery of Association Rules, pp. 307–328. ISBN: 0-262-56097-6. URL: http://dl.acm.org/citation.cfm?id=257938.257975 (cit. on p. 8).

Aho, A. V. and J. E. Hopcroft (1974). *The Design and Analysis of Computer Algorithms*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000296 (cit. on p. 6).

Aigner, M. and G. M. Ziegler (2010). "Cayley's formula for the number of trees". In: *Proofs from THE BOOK*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 201–206. ISBN: 978-3-642-00856-6. URL: http://dx.doi.org/10.1007/978-3-642-00856-6_30 (cit. on p. 41).

Asai, T., H. Arimura, T. Uno, and S. ichi Nakano (2003). "Discovering Frequent Substructures In Large Unordered Trees". In: *IN PROC. OF THE 6TH INTL. CONF. ON DISCOVERY SCIENCE*. Springer-Verlag, pp. 47–61 (cit. on p. 27).

Butcher, S. P. (2003). "Target Discovery and Validation in the Post-Genomic Era". In: *Neurochemical Research* 28.2, pp. 367–371. ISSN: 1573-6903. URL: http://dx.doi.org/10.1023/A:1022349805831 (cit. on p. 1).

Chi, Y., Y. Yang, and R. R. Muntz (2003). "Indexing and mining free trees". In: *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pp. 509–512 (cit. on pp. 2, 5, 6).

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press. ISBN: 0262033844, 9780262033848 (cit. on p. 27).

Cui, J.-H., J. Kim, D. Maggiorini, K. Boussetta, and M. Gerla (2005). "Aggregated Multicast – A Comparative Study". In: *Cluster Computing* 8.1, pp. 15–26. ISSN: 1573-7543. URL: http://dx.doi.org/10.1007/s10586-004-4433-8 (cit. on p. 5).

Diestel, R. *Graph Theory*. Vol. 173. Springer (cit. on p. 5).

Garey, M. R. and D. S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co. ISBN: 0716710447 (cit. on p. 2).

Hein, J., T. Jiang, L. Wang, and K. Zhang (1996). "On the complexity of comparing evolutionary trees". In: *Discrete Applied Mathematics* 71.1, pp. 153 –169. ISSN: 0166-218X. URL: http://www.sciencedirect.com/science/article/pii/S0166218X96000625 (cit. on p. 5).

Hopcroft, J. E. and R. M. Karp (1971). "A n5/2 algorithm for maximum matchings in bipartite". In: *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pp. 122–125 (cit. on p. 21).

Horváth, T. and J. Ramon (2008). "Efficient Frequent Connected Subgraph Mining in Graphs of Bounded Treewidth". In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I*. Ed. by W. Daelemans, B. Goethals, and K. Morik. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 520–535. ISBN: 978-3-540-87479-9. URL: http://dx.doi.org/10.1007/978-3-540-87479-9_52 (cit. on pp. 8, 27).

Horváth, T., B. Bringmann, and L. De Raedt (2007). "Frequent Hypergraph Mining". In: *Inductive Logic Programming: 16th International Conference, ILP 2006, Santiago de Compostela, Spain, August 24-27, 2006, Revised Selected Papers*. Ed. by S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 244–259. ISBN: 978-3-540-73847-3. URL: http://dx.doi.org/10.1007/978-3-540-73847-3_26 (cit. on pp. 2, 8).

Johnson, D. S., M. Yannakakis, and C. H. Papadimitriou (1988). "On generating all maximal independent sets". In: *Information Processing Letters* 27.3, pp. 119 –123. ISSN: 0020-0190. URL: http://www.sciencedirect.com/science/article/pii/0020019088900658 (cit. on p. 7).

Jones, E., T. Oliphant, P. Peterson, et al. (2001–). *SciPy: Open source scientific tools for Python*. [Online; accessed 14-Dec-2016]. URL: http://www.scipy.org/ (cit. on p. 43).

KDD diagram. [Online, accessed 24-Nov-2016]. URL: http://www2.cs.uregina.ca/~dbd/cs831/notes/kdd/kdd.gif (cit. on p. 1).

Lindsay, M. A. (2003). "Target discovery". In: *Nat Rev Drug Discov* 2.10, pp. 831–838. ISSN: 1474-1776. URL: http://dx.doi.org/10.1038/nrd1202 (cit. on p. 1).

Liu, T.-L. and D. Geiger (1999). "Approximate tree matching and shape similarity". In: *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. Vol. 1, 456–462 vol.1 (cit. on p. 5).

Marquardt, D. W. (1963). "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *Journal of the Society for Industrial and Applied Mathematics* 11.2, pp. 431–441. ISSN: 0368-4245. URL: http://dx.doi.org/10.1137/0111030 (cit. on p. 43).

Matoušek, J. and R. Thomas (1992). "On the Complexity of Finding Iso- and Other Morphisms for Partial K-trees". In: *Discrete Math.* 108.1-3, pp. 343–364. ISSN: 0012-365X. URL: http://dx.doi.org/10.1016/0012-365X(92)90687-B (cit. on p. 15).

Nijssen, S. and J. N. Kok (2003). "Efficient discovery of frequent unordered trees". In: *In First International Workshop on Mining Graphs, Trees and Sequences*, pp. 55–64 (cit. on p. 27).

— (2004). "A Quickstart in Frequent Structure Mining Can Make a Difference". In: *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '04. Seattle, WA, USA: ACM, pp. 647–652. ISBN: 1-58113-888-1. URL: http://doi.acm.org/10.1145/1014052.1014134 (cit. on pp. 9, 50).

Pascal Welke Tamas Horvath, S. W. (2015a). "On the Complexity of Frequent Subtree Mining in Very Simple Structures". In: Proceedings of Inductive Logic Programming 2014 (cit. on pp. 2, 3, 5, 7–10, 15, 18).

— (2015b). *Subtree isomorphism in graphs with locally polynomical spanning trees*. unpublished, see Appendix (cit. on pp. 2–4, 15, 17–20, 37).

Read, R. C. and R. E. Tarjan (1975). "Bounds on Backtrack Algorithms for listing Cycles, Paths, and Spanning Trees". In: *Networks* (cit. on pp. 2, 20).

Sams-Dodd, F. (2005). "Target-based drug discovery: is something wrong?" In: *Drug Discovery Today* 10.2, pp. 139 –147. ISSN: 1359-6446. URL: http://www.sciencedirect.com/science/article/pii/S1359644604033161 (cit. on p. 1).

Shamir, R. and D. Tsur (1999). "Faster Subtree Isomorphism". In: *J. Algorithms* 33.2, pp. 267–280. ISSN: 0196-6774. URL: http://dx.doi.org/10.1006/jagm.1999.1044 (cit. on pp. 2, 15).

Tools, C. G. C. and U. Services. *AIDS99*. [Online; accessed 20-September-2016]. URL: https://cactus.nci.nih.gov/ (cit. on p. 38).

Tseng, W.-L. D. *Graph Theory: Network Flows and Matching*. [Online; accessed 19-July-2016]. URL: http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part5.pdf (cit. on p. 26).

Welke, P. (2014). *smallgraphs*. https://bitbucket.org/pascalwelke/smallgraphs/. Bitbucket project; Online; accessed 14-Dec-2016 (cit. on pp. 6, 50).

— (2016). *Fast Frequent Subtree Mining in Graph Databases with Locally Polynomial Spanning Trees*. unpublished, see Appendix (cit. on pp. 4, 27, 29–32).

Yang, Y., S. J. Adelstein, and A. I. Kassis (2009). "Target discovery from data mining approaches". In: *Drug Discovery Today* 14.3–4, pp. 147 –154. ISSN: 1359-6446. URL: http://www.sciencedirect.com/science/article/pii/S1359644608004133 (cit. on p. 1).

Zaki, M. J. (2005). "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications". In: *IEEE Trans. on Knowl. and Data Eng.* 17.8, pp. 1021–1035. ISSN: 1041-4347. URL: http://dx.doi.org/10.1109/TKDE.2005.125 (cit. on p. 6).

ZINC. [Online; accessed 14-Nov-2016]. URL: `http://zinc.docking.org/` (cit. on p. 38).