저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

- 저작자표시. 귀하는 원저작자를 표시하여야 합니다.
- 비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.
- 변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.
공학박사학위논문

비순환 방향 그래프에서의 동적 프로그래밍을 이용한 효율적인 부분그래프 동형 알고리즘

An Efficient Algorithm for Subgraph Isomorphism using Dynamic Programming on Directed Acyclic Graphs

2018년 2월

서울대학교 대학원
전기·컴퓨터공학부

한 명 지
Abstract

An Efficient Algorithm for Subgraph Isomorphism using Dynamic Programming on Directed Acyclic Graphs

Myoungji Han
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Subgraph isomorphism (or subgraph matching) is one of the fundamental problems in graph analysis. It is an NP-hard problem, and extensive research has been done to develop practical solutions for subgraph isomorphism. Although a great deal of progress has been made, the existing solutions still show a limited scalability in handling large real graphs. The state-of-the-art algorithms are based on the general framework of backtracking which recursively finds all embeddings of a query graph in a data graph. In this thesis we introduce three new techniques: dynamic programming on directed acyclic graphs, the adaptive matching order with DAG-ordering, and pruning by failing sets, which together lead to a much faster and more scalable algorithm DP-iso for subgraph isomorphism. Extensive experiments with real datasets show that DP-iso outperforms the fastest existing solution by up to 4 orders of magnitude with respect to the running time and up to 6 orders of magnitude with respect to the number of recursions.

Keywords: Subgraph Isomorphism, Subgraph Matching, Dynamic Programming, Directed Acyclic Graph, DAG, NP-hard
Student Number: 2010-23297
# Contents

Abstract i

Contents ii

List of Figures v

List of Tables vi

Chapter 1 Introduction 1
  1.1 Background ........................................ 1
  1.2 Challenges ......................................... 3
  1.3 Our Ideas .......................................... 4
  1.4 Organization ...................................... 6

Chapter 2 Preliminaries 7
  2.1 Notations ............................................ 7
  2.2 Problem Definition .................................. 7
  2.3 Directed Acyclic Graph ............................. 8
  2.4 Related Work ....................................... 8
    2.4.1 Subgraph Matching ............................. 8
    2.4.2 Other Work on Subgraph Matching ............... 10
    2.4.3 Dynamic Programming .......................... 11
Chapter 3 Overview of DP-iso

3.1 Algorithm Outline .................................. 12
3.2 Leaf Decomposition .................................. 13

Chapter 4 Dynamic Programming on DAG

4.1 CS Structure ........................................... 14
   4.1.1 Definition of CS Structure ...................... 14
   4.1.2 Properties of CS Structure ...................... 15
4.2 DP on DAG ............................................ 16
   4.2.1 Weak Embedding ................................ 16
   4.2.2 CS Refinement using DP on DAG ................ 17
   4.2.3 Time Complexity ................................ 19
4.3 Optimizing CS ........................................ 19

Chapter 5 DAG-Ordering and Adaptive Matching Order

5.1 Backtracking Based on DAG-Ordering ................. 21
   5.1.1 DAG-Ordering and Extendable Candidates ....... 21
   5.1.2 Backtracking Framework ......................... 24
5.2 Adaptive Matching Order ............................. 24
   5.2.1 Infrequent-Path-First Strategy and Tree-like Paths . . 24
   5.2.2 Weight Array .................................. 25
   5.2.3 Adaptive Matching Order using Weight Array ...... 26
5.3 Backtracking Process ................................ 27

Chapter 6 Pruning by Failing Sets

6.1 Search Tree Node ................................... 29
6.2 General Idea .......................................... 30
6.3 Failing Set .......................................... 31
   6.3.1 Definition of Failing Set ....................... 31
   6.3.2 Computing Failing Sets ......................... 32
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Subgraph isomorphism</td>
<td>2</td>
</tr>
<tr>
<td>4.1</td>
<td>Query DAGs</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Dynamic programming on DAG</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>New running example</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Search tree</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Redundant partial embeddings</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Pruned search tree</td>
<td>36</td>
</tr>
<tr>
<td>7.1</td>
<td>Running time for $Q_{20\text{min}}$ of Human</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>DP-iso versus CFL-Match</td>
<td>42</td>
</tr>
<tr>
<td>7.3</td>
<td>DP-iso versus Turboiso</td>
<td>44</td>
</tr>
<tr>
<td>7.4</td>
<td>Evaluating boosting technique</td>
<td>46</td>
</tr>
<tr>
<td>7.5</td>
<td>Effects of pruning technique</td>
<td>47</td>
</tr>
<tr>
<td>7.6</td>
<td>Additional experiments</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables

Table 7.1 Characteristics of datasets .................................. 38
Chapter 1

Introduction

1.1 Background

In recent years, graphs have been playing an increasingly important role in various domains, e.g., social networks, bioinformatics, chemistry, software engineering, etc. One of the most fundamental problems in graph analysis is subgraph isomorphism (or subgraph matching). Given a data graph $G$ and a query graph $q$, subgraph isomorphism is the problem of finding all distinct embeddings of $q$ in $G$. For example, for query graph $q$ and data graph $G$ in Figure 1.1, there are two embeddings of $q$ in $G$, i.e., $\{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_{10})\}$ and $\{(u_1, v_1), (u_2, v_4), (u_3, v_5), (u_4, v_{10})\}$. In practice, subgraph isomorphism has a wide range of applications including protein interaction analysis [22], chemical compound search [33], and social network analysis [27]. In theory, it is one of the well-known NP-hard problems [12].

Extensive research has been done to develop practical solutions for subgraph isomorphism. Most practical solutions (including VF2 [7], QuickSI [25], GraphQL [14], GADDI [36], SPath [37], TurboISO [13], and CFL-Match [5]) are based on the backtracking approach [32, 18], which recursively extends a partial embedding.
by mapping the next query vertex to a data vertex. For example, $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_5)\}$ is a partial embedding, and $u_4$ is the next query vertex in Figure 1.1. If the mapping of the next query vertex is possible, it extends the partial embedding and recurses; otherwise, it backtracks.

The general framework of this approach consists of two stages.

- In the first stage, one adopts a filtering process to find a candidate set $C(u)$ for each vertex $u$ in $q$, where $C(u)$ is a subset of $V(G)$ which $u$ can be mapped to. (Without a filtering, $C(u)$ would be the whole set $V(G)$ of data vertices.) One wants to find smaller candidate sets, which lead to a smaller search space in the second stage.

- In the second stage, one chooses a linear order of the query vertices, called the matching order, and applies backtracking based on the matching order. One wants to choose a matching order that could minimize the search space of backtracking. One may use pruning methods to prune out some parts of the search space.

The state-of-the-art algorithms TurboIso [13] and CFL-Match [5] use a spanning tree $q_T$ of query graph $q$ for a filtering process which finds (potential) embeddings of $q_T$ in data graph $G$. CFL-Match additionally filters using non-tree edges (i.e., edges of $q$ that are not in $q_T$). The candidate sets computed by the filtering process are stored into an auxiliary data structure, i.e., CR [13].
and CPI [5]. Then, guided by the auxiliary data structure, they find all embeddings of \( q \) by checking non-tree edges during backtracking. The auxiliary data structure also helps to select an effective matching order.

### 1.2 Challenges

Although the above framework seems to be standard, even the state-of-the-art algorithms show a limited scalability in handling large real datasets. There are several challenges we need to deal with to solve subgraph isomorphism more efficiently.

1. **Limitations of Spanning Trees.** Although the use of a spanning tree \( q_T \) provides nice properties as above, it creates limitations at the same time. First, since \( q_T \) does not contain all the edges of \( q \), the resulting auxiliary data structure may contain many false positives, which leads to superfluous computations and also inaccurate matching order selection. Second, since the auxiliary data structure contains no information about non-tree edges, one has to probe data graph \( G \) frequently to check non-tree edges between the mapped data vertices during backtracking. Since data graphs are generally of large size, the frequent probing of \( G \) is quite an overhead.

2. **Sub-optimal Matching Order.** The state-of-the-art algorithms TurboISO and CFL-Match use path-ordering techniques which determine the matching order by ordering the root-to-leaf paths of spanning tree \( q_T \). It has two inherent problems. First, suppose that \( q_T \) has two root-to-leaf paths \( A \) and \( B \), \( A \) has 100 embeddings (i.e., matching paths) in \( G \) starting from a vertex \( v \), and \( B \) has 50 embeddings in \( G \) also starting from \( v \). Hence there are \( 100 \times 50 \) embeddings of paths \( A \) and \( B \) in \( G \). But if there are non-tree edges between \( A \) and \( B \), many of the 5000 embeddings can be eliminated by checking the existence of non-tree edges in the embeddings.
This phenomenon is called the redundant Cartesian products in [5]. That is, non-tree edges have a strong pruning power and so they should be processed as early as possible to reduce the redundant Cartesian products. To this end, CFL-Match decomposes the query graph into a dense subgraph (i.e., core) which contains all non-tree edges and a forest, and then matches the core first (i.e., put the core vertices forward in the matching order). However, since the core-matching process of [5] still applies the path-ordering to select the matching order of the core vertices, the same problem may arise inside the core.

Second, the path-ordering techniques select a global matching order so that the precomputed matching order is used throughout the whole search process. Though TurboISO [13] computes a different matching order for each region, it still runs in a global fashion inside each region. Only the path-at-a-time strategy of SPath [37] dynamically selects the next matching vertex considering the current partial embedding, but its matching order selection is found to be far from optimal [18].

3. Redundant Computations in Search. By the nature of backtracking, there could be many redundant computations in the search process. This gives us the possibility of utilizing the knowledge gained from past computations to prune out redundant computations in the future.

1.3 Our Ideas

To deal with the above challenges, we introduce the following new ideas that more fully exploit the general framework of backtracking, which together lead to a much faster and more scalable algorithm DP-iso for subgraph isomorphism.

1. Dynamic Programming on DAG. We use a directed acyclic graph (DAG) \( q_D \) of \( q \) in the filtering process instead of a spanning tree, and we find potential embeddings of \( q_D \) in data graph \( G \). This process is
computed by dynamic programming between DAG $q_D$ and data graph $G$, which is a new technique and we call it *dynamic programming on DAG*. Since $q_D$ contains all the edges of $q$, the candidate sets computed by dynamic programming on DAG are smaller than those in previous algorithms. We store the candidate sets into an auxiliary data structure called the *CS (candidate space) structure*. Since we use all the edges of $q$ in constructing the CS structure, it serves as a complete search space, i.e., finding all embeddings of $q$ in $G$ is equivalent to finding all embeddings of $q$ in the CS. Hence it eliminates the need to probe $G$ during the backtracking process.

2. **Adaptive Matching Order with DAG-ordering.** We first present a new backtracking framework based on *DAG-ordering* in which the matching order always follows a topological order of DAG $q_D$. In this way all non-tree edges are checked as early as possible so that we can reduce redundant Cartesian products. We also propose the *adaptive matching order*, where we select the next matching vertex dynamically during the backtracking process considering the current partial embedding in such a way that the final matching order follows the *infrequent-path-first* strategy, which has verified its effectiveness in previous works [37, 13, 5].

3. **Pruning by Failing Sets.** We introduce a new notion of *failing sets* to prune out some parts of the search space. A failing set is a subset of $V(q)$ combined with a partial embedding, which can never lead to a full embedding of $q$. Once we find a failing set, we can safely remove some partial embeddings in the search space. We propose a method to find a failing set for each partial embedding and prune out redundant partial embeddings based on the failing sets.

To evaluate DP-iso, we conducted extensive experiments on real datasets. In our experiments we use six real graph datasets as data graphs, which are
commonly used in previous works. The query graphs in our experiments are of larger scales than those in previous works. Our experiments show that DP-iso outperforms the fastest existing algorithm CFL-Match by up to 4 orders of magnitude with respect to the running time and up to 6 orders of magnitude with respect to the number of recursions. Also, DP-iso shows a better robustness than CFL-Match, where robustness means the percentage of solved queries in a query set. We also compare DP-iso with TurboISO and apply the boosting technique in [23] to DP-iso.

1.4 Organization

The rest of the paper is organized as follows. Chapter 2 gives the problem definition and related work. Chapter 3 presents a brief overview of DP-iso. Chapter 4 describes dynamic programming on DAG by which we compute the CS structure. Chapter 5 presents the backtracking framework based on DAG-ordering and the adaptive matching order. Chapter 6 gives the definition of failing sets and our redundancy pruning technique. Chapter 7 presents the results of performance evaluation, and we conclude in Chapter 8.
Chapter 2

Preliminaries

2.1 Notations

In this thesis we focus on undirected, connected, and vertex-labeled graphs. All techniques in this thesis can be readily extended to handle more general cases (e.g., directed/disconnected graphs, multiple labels on a vertex/edge). A graph $g = (V(g), E(g), L_g)$ consists of a set $V(g)$ of vertices, a set $E(g)$ of undirected edges, and a labeling function $L_g : V(g) \to \Sigma$, where $\Sigma$ is a set of labels. We say that $u \in V(g)$ is adjacent to $v \in V(g)$ if edge $(u, v)$ is in $E(g)$. The degree of a vertex $v \in V(g)$, denoted by $\text{deg}_g(v)$, is the number of vertices that are adjacent to $v$ in $g$. Let $S$ be a subset of $V(g)$. The induced subgraph $g[S]$ is the subgraph of $g$ whose vertex set is $S$ and whose edge set consists of all the edges in $E(g)$ that have both endpoints in $S$.

2.2 Problem Definition

Given a query graph $q = (V(q), E(q), L_q)$ and a data graph $G = (V(G), E(G), L_G)$, an embedding of $q$ in $G$ is a mapping $M : V(q) \to V(G)$ such that

1. $M$ is injective (i.e., $M(u) \neq M(u')$ for $u \neq u'$ in $V(q)$),
(2) \( L_q(u) = L_G(M(u)) \) for every \( u \in V(q) \), and

(3) \((M(u), M(u')) \in E(G)\) for every \((u, u') \in E(q)\).

A mapping that satisfies (2) and (3) is called a graph homomorphism (i.e., it may not be injective). An embedding of an induced subgraph of \( q \) in \( G \) is called a partial embedding. We will sometimes call an embedding a full embedding as opposed to a partial embedding.

**Problem statement.** Given a query graph \( q \) and a data graph \( G \), the subgraph isomorphism problem is to find all distinct embeddings of \( q \) in \( G \).

The subgraph isomorphism problem is NP-hard [12], and thus it is often difficult in practice to find all embeddings in a reasonable time due to the huge search space. Consequently, most algorithms for subgraph isomorphism stop after finding \( k \) embeddings for some \( k \) in practical settings.

### 2.3 Directed Acyclic Graph

A directed acyclic graph (DAG) \( g \) is a connected directed graph that contains no cycle. A DAG \( g \) is a rooted DAG if there is only one vertex \( r \in V(g) \) (i.e., root) that has no incoming edges. The parent-child relationship also holds in a DAG, i.e., \( u \) is a parent of \( v \) (\( v \) is a child of \( u \)) if a directed edge \((u, v)\) is in \( E(g) \). A vertex is a leaf in a DAG if it has no outgoing edges. A sub-DAG of \( g \) rooted at \( u \), denoted by \( g_u \), is the induced subgraph of \( g \) whose vertices are \( u \) and all the descendants of \( u \).

### 2.4 Related Work

#### 2.4.1 Subgraph Matching

The study of practical subgraph matching algorithms was initiated by Ullmann’s backtracking algorithm [32]. It finds all embeddings of a query graph \( q \)
in a data graph $G$ by iteratively mapping a query vertex to a data vertex. Henceforth, a great deal of efforts have been devoted to speed up the backtracking algorithm.

VF2 [7] improves Ullmann’s algorithm by selecting a vertex connected from the already matched query vertices as next vertex. Also, to reduce the number of candidates at each step, they adopt three pruning rules by testing adjacencies. To process edges having infrequent vertex/edge labels as early as possible, QuickSI [25] computes the frequencies of a triple (source vertex label, edge label, target vertex label) in the data graph. After that, they build a minimum spanning tree of the query graph, and then perform backtracking according to the order in which an infrequent edge is matched earlier. GraphQL [14] uses two pruning rules to reduce the candidate sets, local pruning and joint reduction. In the local pruning, they prune candidates by an extensive probe on the label frequencies around the candidates. In the joint reduction, they reduce the overall search space by iteratively removing candidates that a semi-perfect matching does not exist. GADDI [36] first pre-processes the data graph to select frequent subgraphs by subgraph mining. They reduce the size of candidate sets by checking the NDS distance, the numbers of frequent subgraphs between two vertices, between the mapped vertices and the candidates. SPath [37] matches a path of the query graph rather than a vertex at each time. To select the next path, they define a selectivity function, which estimates the join cost of the next path.

The algorithms mentioned above are extensively compared in [18]. Now we give detailed descriptions of the two state-of-the-art algorithms TurboISO [13] and CFL-Match [5] because not only they are the fastest algorithms, but also our algorithm uses the same framework as they do. Both TurboISO and CFL-Match use the general framework of backtracking in Chapter 1, where they use a spanning tree $q_T$ of query graph $q$ in the filtering process.

TurboISO [13] uses the idea that an optimal matching order varies with each region of the data graph, and thus TurboISO finds embeddings of query graph
$q$ in data graph $G$ region by region. By the candidate region exploration, all embeddings of the root-to-leaf paths of spanning tree $q_T$ are found and materialized into an auxiliary data structure called the CR structure. Based on the CR, an effective matching order for each region is computed by the path-ordering technique. A technique of compressing similar query vertices, called the neighborhood-equivalence-class, is also proposed.

CFL-Match [5] is motivated by the inherent problems of TurboISO. The first problem is that the path-ordering technique may delay the checking of non-tree edges and generate many redundant Cartesian products of partial embeddings during the search. To tackle this problem, CFL-Match proposes the core-forest-leaf decomposition, where the query graph is decomposed into a core, a forest, and leaves. They showed that the core-forest-leaf ordering effectively reduces redundant Cartesian products. To tackle the second problem of TurboISO, the exponential size of the CR structure, CFL-Match proposes a more compact auxiliary structure CPI.

2.4.2 Other Work on Subgraph Matching

Though not proposing a complete solution for subgraph matching, some papers focus on revising existing solutions for the sake of performance improvement or generalization. BoostIso [23] proposes a boosting technique that speeds up the existing subgraph matching algorithms by compressing similar vertices in the data graph. MQOsubiso [24] proposes a multi-query optimization technique which revises existing algorithms to handle multiple query graphs efficiently. There are an increasing number of studies on the problem of subgraph matching in a distributed environment [16, 26, 28, 30], where the join paradigm has been shown as the most popular strategy. Also, there are many interesting results on related problems [34, 38, 21, 11].
2.4.3 Dynamic Programming

Dynamic programming is a general method for optimization problems which solves a problem by solving subproblems and combining the solutions to subproblems. Dynamic programming has been one of the most popular methods to solve string problems such as string edit distance [31], longest common subsequence [3], approximate string matching [17], and sequence alignment [2], and it is also used to solve the tree edit distance problem [9, 20].

Dynamic programming has been shown to be an effective tool for subgraph isomorphism when graphs are restricted to some special classes. For example, Bodlaender [6] showed that the decision problem of subgraph isomorphism (i.e., "Does $G$ contain a subgraph isomorphic to $q$?") is solvable in polynomial time for data graphs with bounded tree-widths and degrees. Alon et al. [1] proposed a polynomial-time subgraph isomorphism algorithm when query graph $q$ has a bounded tree-width and $|V(q)| = O(\log |V(G)|)$. Eppstein [10] proposed a linear-time algorithm for subgraph isomorphism when both data graph and query graph are planar and the query graph is fixed. Also dynamic programming can be used to solve some NP-hard graph problems in polynomial time for some subclasses of graphs (e.g., trees, series-parallel graphs, graphs with bounded tree-width, etc) [29, 4, 6]. A main technique in these results is dynamic programming between a tree and a graph. To the best of our knowledge, our dynamic programming between a DAG and a graph is a new notion.
Chapter 3

Overview of DP-iso

In this chapter we give an overview of our algorithm DP-iso and the leaf decomposition strategy.

3.1 Algorithm Outline

Algorithm 1 shows the outline of DP-iso, which takes a query graph $q$ and a data graph $G$ as input, and finds all embeddings of $q$ in $G$.

Algorithm 1: DP-iso

| Input: query graph $q$, data graph $G$ |
| Output: all embeddings of $q$ in $G$ |

1. $q_D \leftarrow \text{BUILD-DAG}(q, G)$;
2. $CS \leftarrow \text{BUILDCS}(q, q_D, G)$;
3. $M \leftarrow \emptyset$;
4. \text{BACKTRACK}$(q, q_D, CS, M)$;

DP-iso consists of the following three procedures:

1. Initially, \text{BUILD-DAG} is invoked to build a rooted DAG $q_D$ from $q$. In
BUILD DAG, we first select a root. Since the root is the first query vertex to match, we prefer the root to have a small number of candidates in $G$ and to have a large degree for better pruning. The root $r$ of $q_D$ is selected as $r \leftarrow \arg\min_{u \in V(q)} \frac{|C_{ini}(u)|}{\deg_G(u)}$, where the initial candidate set $C_{ini}(u)$ is the set of vertices $v \in V(G)$ such that $L_G(v) = L_q(u)$ and $\deg_G(v) \geq \deg_q(u)$.

In order to build $q_D$, we traverse $q$ in a BFS order from $r$ and direct all edges from earlier to later visited vertices. For example, Figure 4.1a shows a rooted DAG built from query graph $q$ in Figure 1.1a when $u_1$ is the root.

2. BUILDCS is invoked to build the CS structure by using dynamic programming on DAG (Chapter 4). We will show that finding all embeddings of $q$ in $G$ is equivalent to finding all embeddings of $q$ in the CS structure.

3. Finally, BACKTRACK is invoked to find all embeddings of $q$ in the CS structure. In order to reduce the search space, BACKTRACK applies the adaptive matching order based on DAG-ordering (Chapter 5). BACKTRACK also uses a pruning technique by failing sets (Chapter 6).

### 3.2 Leaf Decomposition

We adopt the leaf decomposition strategy of [5], which is a conventional technique in graph optimization problems [19]. That is, we decompose the query vertices into the set of degree-one vertices and the set $V'$ of the remaining vertices. We first find embeddings of $q[V']$ in data graph $G$, and then we find full embeddings of $q$ in $G$ by matching the degree-one vertices, for which we use the leaf-matching algorithm of [5]. For simplicity of presentation, we assume that query graphs in this thesis are $q[V']$. 
Chapter 4

Dynamic Programming on DAG

In this chapter we describe a technique called dynamic programming on DAG, by which we construct the CS structure that serves as a compact search space for all embeddings of $q$ in $G$.

4.1 CS Structure

4.1.1 Definition of CS Structure

Given a query graph $q$ and a data graph $G$, the CS structure consists of the candidate set $C(u)$ for each $u \in V(q)$ and edges between the candidates as follows.

1. For each $u \in V(q)$, there is a candidate set $C(u)$, which is a set of vertices in $G$ that $u$ can be mapped to.

2. There is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E(q)$ and $(v, v') \in E(G)$.

An important difference of CS from CPI in [5] is that while CPI does not have an edge between $v \in C(u)$ and $v' \in C(u')$ for non-tree edges $(u, u')$ of $q$, CS can...
have an edge between \( v \in C(u) \) and \( v' \in C(u') \) for every edge \((u, u')\) in \( q \).

### 4.1.2 Properties of CS Structure

A CS structure is *sound* if it satisfies the following requirement:

- If there is an embedding \( M \) of \( q \) in \( G \) such that \( M(u) = v \), then \( v \) must be in \( C(u) \) in the CS.

If there is an embedding of \( q \) in \( G \), then the embedding also exists in a sound CS because of the soundness requirement and Condition (2) in the definition of the CS structure. Also, an embedding that is not in \( G \) cannot be created in a CS because the CS does not have an edge \((v, v')\) that is not in \( E(G) \). Therefore, we have the following property.

- Finding all embeddings of \( q \) in \( G \) is equivalent to finding all embeddings of \( q \) in a sound CS.

This property does not hold in the auxiliary data structure CR of TurboISO [13] or CPI of CFL-Match [5] due to non-tree edges of \( q \), where frequent probings of data graph \( G \) are required to check non-tree edges. Once we compute a compact sound CS, data graph \( G \) is no longer necessary for the remaining part of our algorithm by this property.

Since the edges in a CS are immediate from \( E(q) \) and \( E(G) \) once the candidate sets are decided, the key to the CS construction is to compute the candidate sets that are small as well as sound. For example, the CS structure in Figure 4.2d is the optimal CS for finding all embeddings of \( q \) (Figure 1.1a) in \( G \) (Figure 1.1b). Note that we can find the two embeddings \( \{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_{10})\} \) of \( q \) in the CS in Figure 4.2d.
4.2 DP on DAG

4.2.1 Weak Embedding

We first define the weak embedding, which is a key notion in our dynamic programming.

Definition 4.2.1. The path tree of a rooted DAG $g$ is defined as the tree $g'$ such that each path in $g'$ from the root to a leaf corresponds to a distinct path from the root to a leaf in $g$, and $g'$ shares common prefixes of all its paths from the root to leaves (e.g., Figure 4.1c is the path tree of $q_D$ in Figure 4.1a). For a rooted DAG $g$ with root $u$, a weak embedding of $g$ at $v \in V(G)$ is defined as a graph homomorphism $M'$ of the path tree of $g$ such that $M'(u) = v$.

Note that a weak embedding $M'$ of a rooted DAG $g$ may not be injective (because it is a graph homomorphism), and a vertex of $g$ may be mapped to two or more distinct vertices of $G$ in $M'$ (due to the path tree of $g$). For example, $\{(u_1, v_1), (u_2, v_4), (u_4, v_{10}), (u_3, v_5), (u_4', v_{10}), (u_4', v_6), (u_4', v_{10})\}$ is a weak embedding of $q_D$ (Figure 4.1a) in $G$ (Figure 1.1b), where $u_3$ in $q_D$ is mapped to two different vertices $v_5$ and $v_6$ of $G$ via the path tree. Every embedding of $g$ in $G$ is a weak embedding of $g$ in $G$, but the converse is not true (i.e., a weak embedding may not be an embedding). Hence a weak embedding is a necessary condition for an embedding.
Figure 4.2: Dynamic programming on DAG

4.2.2 CS Refinement using DP on DAG

Now we describe our strategy to compute a compact CS. Initially, $C(u)$ is set to $C_{ini}(u)$ for every $u \in V(q)$, i.e., $v \in C(u)$ if and only if $L_G(v) = L_q(u)$ and $\deg_G(v) \geq \deg_q(u)$. Clearly, the initial CS is sound. Figure 4.2a shows the initial CS for the example in Figure 1.1. A query DAG is defined as a DAG that is built from query graph $q$ by assigning directions to the edges in $q$ (e.g., $q_D$ and $q_D^{-1}$ in Figure 4.1 are query DAGs). We will refine the initial CS by using query DAGs.

Suppose that we are given a query DAG $q'$ and a sound CS. We have the following observation:

- If there is an embedding $M$ of $q$ in the CS such that $M(u) = v$ for a vertex
In $q$, there must be a weak embedding of $q'_u$ at $v$ in the CS (where $q'_u$ is the sub-DAG of $q'$ rooted at $u$).

Consequently, if such a weak embedding does not exist, the CS is still sound after we remove $v$ from $C(u)$ (and the incident edges) in the CS.

From the above observation, we design a CS refinement algorithm based on dynamic programming. For each candidate set $C(u)$, we define the refined candidate set $C'(u)$ as follows:

$v \in C'(u)$ iff $v \in C(u)$ and there is a weak embedding of $q'_u$ at $v$ in the CS.

For example, Figure 4.2c shows the result of the refinement of the CS in Figure 4.2b by using the rooted DAG in Figure 4.1a as query DAG $q'$. Note that $v_8$ is removed from $C(u_2)$ since there is no weak embedding of $q'_{u_2}$ at $v_8$ in the CS in Figure 4.2b. Then, we obtain the following recurrence:

$$v \in C'(u) \text{ iff } v \in C(u) \text{ and } \exists v_c \text{ adjacent to } v \text{ such that } v_c \in C'(u_c)$$

for every child $u_c$ of $u$ in $q'$. (4.1)

One can prove by induction that Recurrence (4.1) computes $C'(u)$ correctly. Based on the recurrence, we can compute $C'(u)$ for all $u \in V(q)$ by dynamic programming in a bottom-up fashion in DAG $q'$. Specifically, we compute $C'(u)$ for $u \in V(q)$ in a reverse topological order of $q'$, i.e., $u$ is processed after all its children in $q'$ are processed. This refinement technique will be called dynamic programming on DAG.

Remark. Suppose that we define $D[u, v] = 1$ if $v \in C(u)$; $D[u, v] = 0$ if $v \notin C(u)$, and we refine $D$ into $D'$ (i.e., $D'[u, v] = 1$ if and only if $D[u, v] = 1$ and there is a weak embedding of $q'_u$ at $v$ in the CS). Then Recurrence (4.1) would be

$$D'[u, v] = 1 \text{ iff } D[u, v] = 1 \text{ and } \exists v_c \text{ adjacent to } v \text{ such that } D'[u_c, v_c] = 1$$

for every child $u_c$ of $u$ in $q'$. 

18
$D$ is a typical dynamic programming table \cite{8} between query DAG $q'$.

4.2.3 Time Complexity

\begin{align*}
\forall v \in C(u) \\
(v, v_c) \quad v_c \in C'(u_c) \\
\end{align*}

$CS$

$u, u' \in V(q)$ \quad $O(|E(G)|)$

$CS$

$C(u) \quad C(u')$

$O(|E(G)| \times |E(q)|)$

4.3 Optimizing $CS$

\begin{align*}
\forall q_D \quad q_D^{-1} \\
\end{align*}

$CS$

$q_D \quad q_D^{-1}$

$CS$

$q_D \quad q_D^{-1}$

$CS$

$q_D \quad q_D^{-1}$

$CS$

$q_D \quad q_D^{-1}$
possible using either $q_D$ or $q_D^{-1}$.

Our empirical study showed that three steps are enough for optimization (the filtering rate after the first 3 steps was below 1% in almost all experiments). Thus, we set the number of refinements to 3 in our experiments.

The edges in the CS are immediately identified from query graph $q$ and data graph $G$, once candidate sets are given. During the refinement process, therefore, we maintain candidate sets $C(u)$ only. The edges depicted in Figure 4.2 are only for presentational purposes. After we compute the final candidate sets, however, we materialize the edges to obtain the complete CS structure (i.e., Figure 4.2d). The edges are stored as an adjacency list $N_{uc}(v)$ for each $v \in C(u)$ and each edge $(u, uc) \in E(q_D)$ in the CS, where $N_{uc}(v)$ is the list of vertices $v_c$ adjacent to $v$ in $G$ such that $v_c \in C(u_c)$. Once the CS is computed, data graph $G$ is no longer necessary, and we can free the memory allocated for the data graph. Since data graphs are generally of large size, it helps to alleviate the memory overhead during the backtracking.
Chapter 5

DAG-Ordering and Adaptive Matching Order

In this chapter we present our matching algorithm to find all embeddings of query graph $q$ in the CS structure. Our matching algorithm also exploits the rooted DAG $q_D$ in selecting the matching order.

Example 5.0.1. As a new running example, we use the rooted DAG $q_D$ of $q$ in Figure 5.1a ($q$ is easily inferred from $q_D$) and the CS in Figure 5.1b, which is constructed by our CS optimization technique in Chapter 4. There are 3 embeddings of $q$ in the CS, i.e., $\{(u_1, v_1), (u_2, v_3), (u_3, v_6), (u_4, v_2), (u_5, v_7), (u_6, v_9), (u_7, v_{12}), (u_8, v_{13-15}), (u_9, v_{19})\}$. Note that there are no embeddings such that $u_2$ is mapped to $v_2$ because, if so, $u_4$ cannot be mapped to its only candidate $v_2$.

5.1 Backtracking Based on DAG-Ordering

5.1.1 DAG-Ordering and Extendable Candidates

We propose a new backtracking framework for subgraph matching based on the DAG-ordering. Since the edges of query graph $q$ are the source of the pruning
Figure 5.1: New running example

power, we want the edges to be checked as early as possible. That is, a query vertex \( u \) should be processed after all its adjacent vertices that are closer (in terms of the number of edges) to a start vertex (i.e., the root of \( q_D \)) have been
processed. This is ensured by the DAG-ordering below and the fact that rooted DAG \( q_D \) is built by BFS in BuildDAG.

**Definition 5.1.1.** In the DAG-ordering of \( q_D \), a query vertex can be processed only after all its parents in \( q_D \) have been processed. Specifically, an unvisited query vertex \( u \) is called extendable regarding a partial embedding \( M \) if all parents of \( u \) are matched in \( M \). The DAG-ordering always selects an extendable vertex as the next vertex.

Based on the DAG-ordering and the CS structure, we can obtain the candidates of an extendable vertex by using the set intersection.

**Definition 5.1.2.** Suppose that we are given a partial embedding \( M \) and an extendable vertex \( u \). Let \( p_1, \ldots, p_k \) be the parents of \( u \) in \( q_D \). Since \( u \) is extendable, all \( p_1, \ldots, p_k \) are matched in \( M \). The set of extendable candidates of \( u \) regarding \( M \) is defined as \( C_M(u) = \bigcap_{i=1}^{k} N_{p_i}^u(M(p_i)) \).

**Example 5.1.1.** Given a partial embedding \( M = \{(u_1, v_1), (u_2, v_2), (u_3, v_5), (u_5, v_7), (u_6, v_8)\} \) in Figure 5.1, the extendable vertices are \( \{u_4, u_7\} \). The extendable candidates of \( u_4 \) are computed as \( C_M(u_4) = N_{u_4}^{u_1}(v_1) = \{v_2\} \). Also, the extendable candidates of \( u_7 \) are computed as \( C_M(u_7) = N_{u_7}^{u_5}(v_5) \cap N_{u_7}^{u_6}(v_8) = \{v_{11}\} \cap \{v_{10}, v_{11}\} = \{v_{11}\} \).

**Lemma 5.1.1.** Suppose that we are given a partial embedding \( M \) and an extendable vertex \( u \). For every unvisited candidate \( v \in C_M(u) \), \( M \cup \{(u, v)\} \) is a valid partial embedding.

In Example 5.1.1, \( M \cup \{(u_7, v_{11})\} \) is a valid partial embedding, i.e., \( \{(u_1, v_1), (u_2, v_2), (u_3, v_5), (u_5, v_7), (u_6, v_8), (u_7, v_{11})\} \). For extendable vertex \( u_4 \), we cannot extend \( M \) to \( u_4 \) because the only extendable candidate \( v_2 \in C_M(u_4) \) is already visited.
5.1.2 Backtracking Framework

Based on Lemma 5.1.1, our new backtracking framework of subgraph matching finds all embeddings of $q$ in the $CS$ as follows.

1. Select an extendable vertex $u$ regarding the current partial embedding $M$.

2. Extend $M$ by mapping $u$ to each unvisited $v \in C_M(u)$ and recurse.

Note that this framework based on the DAG-ordering will always choose a matching order that corresponds to a topological order of $q_D$.

A natural question arises: Among all extendable vertices regarding $M$, which one should be extended first? In Example 5.1.1, we have two extendable vertices regarding $M$, i.e., $u_4$ and $u_7$, and we need to decide which one we extend first. Our answer to this question is the key to our adaptive-matching-order technique in the following section.

5.2 Adaptive Matching Order

5.2.1 Infrequent-Path-First Strategy and Tree-like Paths

The common strategy of the path-ordering techniques [13, 5] is to match a root-to-leaf path in the spanning tree of $q$ that is infrequent in $G$ first, in order to reduce the search space. This infrequent-path-first strategy has been shown to be effective in previous works [37, 13, 5]. We wish to adopt the same strategy, but it is not clear what in $q_D$ should be considered as basic units of the strategy under the DAG-ordering (e.g., root-to-leaf paths are basic units for the path-ordering [13, 5]). To address this issue, we propose the notion of tree-like paths.

**Definition 5.2.1.** Given a rooted DAG $q_D$, a path $p$ in $q_D$ starting from $u \in V(q)$ is called tree-like if all vertices on $p$ except its leading vertex $u$ have exactly one parent in $q_D$. A tree-like path $p$ is maximal if there is no tree-like path that has $p$ as a proper prefix.
For example, the maximal tree-like paths starting from \( u_1 \) of \( q_D \) in Figure 5.1a are \( \{(u_1, u_2, u_5), (u_1, u_2, u_6), (u_1, u_3), (u_1, u_4, u_8)\} \).

The vertices along a tree-like path can always be matched continuously in backtracking while not violating the DAG-ordering. In backtracking, therefore, we match maximal tree-like paths one at a time. Following the infrequent-path-first strategy, we aim to match a maximal tree-like path in \( q_D \) that is infrequent in the CS first. To achieve this goal, we construct a weight array.

### 5.2.2 Weight Array

The weight array assigns a weight to each candidate in the CS. We denote by \( W_u(v) \) the weight of the candidate \( v \in C(u) \). For a path \( p = (u_1, u_2, \ldots, u_k) \) in \( q_D \), a path \( (v_1, v_2, \ldots, v_k) \) such that \( v_i \in C(u_i) \) is called a path in the CS corresponding to \( p \). Let \( P \) be the set of all maximal tree-like paths starting from \( u \) in \( q_D \). For each path \( p \in P \), we define \( n(p, v) \) as the number of paths starting from \( v \) in the CS corresponding to \( p \). In Figure 5.1, \( n(p_1, v_1) = 4 \) for \( p = (u_1, u_2, u_5) \). Note that it serves as an upper bound of the number of path embeddings, since vertex overlaps may occur in the paths in the CS (e.g., \( p \) actually has 2 embeddings when \( u_1 \) is mapped to \( v_1 \), i.e., \( (v_1, v_{2-3}, v_7) \)). Now we define \( W_u(v) = \min_{p \in P} n(p, v) \). That is, \( W_u(v) \) is an upper bound of the number of embeddings of the most infrequent maximal tree-like path starting from \( u \) in \( q_D \), when \( u \) is mapped to \( v \).

**Example 5.2.1.** Figure 5.1c shows the weight array for the CS in Figure 5.1b. The weights are written on the corresponding positions of the candidates. \( W_{u_1}(v_1) = 2 \) because for the set of all maximal tree-like paths starting from \( u_1 \), i.e., \( p_1 = (u_1, u_2, u_5), p_2 = (u_1, u_2, u_6), p_3 = (u_1, u_3), p_4 = (u_1, u_4, u_8) \), we have \( n(p_1, v_1) = 4, n(p_2, v_1) = 2, n(p_3, v_1) = 3, \) and \( n(p_4, v_1) = 4 \).

The weight array can be computed in time proportional to the size of the CS by dynamic programming in a bottom-up fashion. If \( u \in V(q) \) has no child
in $q_D$ that has only one parent, $W_u(v) = 1$ for all $v \in C(u)$. Otherwise, let the children of $u$ in $q_D$ that has only one parent (i.e., $u$) be $c_1, \ldots, c_k$. For each $v \in C(u)$, we compute $W_u(v)$ as follows. For each $c_i$, we compute $W_{u,c_i}(v) = \sum_{v' \in N^+_c(v)} W_{c_i}(v')$. Then, we set $W_u(v) = \min_{1 \leq i \leq k} W_{u,c_i}(v)$.

### 5.2.3 Adaptive Matching Order using Weight Array

In the adaptive matching order, we select the next matching vertex by using the weight array. Suppose that we are trying to extend a partial embedding $M$. The weight of each extendable vertex $u$ is computed as $w_M(u) = \sum_{v \in C_M(u)} W_u(v)$ (recall that $C_M(u)$ denotes the extendable candidates of $u$ regarding $M$). We always select the extendable vertex with the minimum weight as the next vertex. Since the weight is computed regarding the current partial embedding, the next vertex selected may be different for different partial embeddings.
Example 5.2.2. Consider the running example in Figure 5.1 and the search tree in Figure 5.2. For a partial embedding $M = \{(u_1, v_1), (u_2, v_2), (u_6, v_8), (u_5, v_7), (u_3, v_4)\}$, we have $w_M(u_4) = 4$ and $w_M(u_7) = 2$ (due to $C_M(u_7) = \{v_{10}\}$) for the extendable vertices $\{u_4, u_7\}$ regarding $M$, and so $u_7$ is selected as the next vertex. On the other hand, for $M' = \{(u_1, v_1), (u_2, v_2), (u_6, v_8), (u_5, v_7), (u_3, v_5)\}$, we have $w_{M'}(u_4) = 4$ and $w_{M'}(u_7) = 5$ (due to $C_{M'}(u_7) = \{v_{11}\}$), and so $u_4$ is selected as the next vertex.

5.3 Backtracking Process

Algorithm 2: BACKTRACK($q,q_D,CS,M$)

1 if $|M| = |V(q)|$ then
   2 Report $M$;
3 else if $|M| = 0$ then
   4 foreach $v \in C(r)$ do
      5 $M \leftarrow \{(r,v)\}$; Mark $v$ as visited;
      6 BACKTRACK($q,q_D,CS,M$); Mark $v$ as unvisited;
7 else
   8 $u \leftarrow$ extendable vertex with minimum weight $w_M(u)$;
   9 foreach $v \in C_M(u)$ do
      10 if $v$ is unvisited then
         11 $M' \leftarrow M \cup \{(u,v)\}$; Mark $v$ as visited;
         12 BACKTRACK($q,q_D,CS,M'$); Mark $v$ as unvisited;
13 return;

Algorithm 2 (i.e., BACKTRACK) shows the backtracking process of DP-iso. It finds all embeddings of $q$ in the CS by extending a partial embedding $M$. If $|M| = |V(q)|$, $M$ is an embedding and so we report it. If $|M| = 0$, we map the
root vertex $r$ of $q_D$ to one of its candidates $v \in C(r)$, and invoke Backtrack recursively (line 6). Otherwise (i.e., $0 < |M| < |V(q)|$), we select the extendable vertex $u$ with the minimum weight as the next vertex. For each unvisited $v \in C_M(u)$ (lines 9-10), we extend the current partial embedding $M$ by adding $(u, v)$ to $M$ and recursively invoke Backtrack with the extended partial embedding $M'$ (line 12). We maintain a min priority queue to get the vertex with the minimum weight. The weights of extendable vertices are computed immediately when they become extendable due to an extension of $M$.

The search tree in Figure 5.2 illustrates the search process of Backtrack for the running example in Figure 5.1, where a pair $(u, v)$ on a node represents the mapping of $u$ to $v$, ‘!’ means a mapping conflict, e.g., the leftmost leaf $(u_5, v_1)$! means that $v_1$ is already matched and so $u_5$ cannot be mapped to $v_1$, and $(u, \emptyset)$ means that there are no extendable candidates of $u$. 
Chapter 6

Pruning by Failing Sets

In this chapter we develop a new technique to prune out some parts of the search space by making use of the notion of failing sets. Since our technique runs on the search tree, we first define the notations for nodes of the search tree.

6.1 Search Tree Node

A node in the search tree in general corresponds to a partial embedding. Consider the search tree in Figure 5.2. The root of the search tree corresponds to partial embedding $M_1 = \{(u_1, v_1)\}$, its left child corresponds to $M_2 = \{(u_1, v_1), (u_2, v_2)\}$, and so on. Thus, we use the mapping function $M$ to represent a node as well as a partial embedding. For the sake of traceability, we enumerate the mapping pairs in $M$ in the order in which they are added to $M$. Although some leaf nodes in Figure 5.2 may not correspond to partial embeddings, we represent them by mapping functions with ‘!’ and $\emptyset$, e.g., $M = \{(u_1, v_1), (u_2, v_2), (u_6, v_8), (u_5, v_1)\}$, $M' = \{(u_1, v_1), (u_2, v_2), (u_6, v_8), (u_5, v_7), (u_3, v_6), (u_7, \emptyset)\}$. 
6.2 General Idea

Since we traverse the search tree in a DFS order, once we visit a new node $M$ (by an extension from its parent), we are supposed to explore the subtree rooted at $M$ and come back to node $M$. Our goal is to make use of some knowledge gained from the exploration of the subtree rooted at $M$ to prune out some partial embeddings, especially among the siblings of node $M$.

Suppose that we have explored the subtree rooted at $M$ and found no embeddings in the subtree. At this point, we have a trivial knowledge that partial embedding $M$ cannot lead to a full embedding of $q$. Rather than this trivial knowledge, we want to find a more specific knowledge as to what caused such
a failure.

**Example 6.2.1.** Figure 6.1c is a search tree for query graph \( q \) in Figure 6.1a and data graph \( G \) in Figure 6.1b. Suppose that we have just finished exploring the subtree rooted at \( M = \{ (u_1, v_1), (u_2, v_2), (u_4, v_{1003}) \} \). We get to know that partial embedding \( M \) cannot lead to a full embedding of \( q \). A more specific knowledge we can get, however, is that a subset \( M' = \{ (u_1, v_1), (u_2, v_2) \} \) of \( M \) cannot lead to an embedding of \( \{ u_1, u_2, u_3, u_5 \} \) (i.e., the subgraph of \( q \) induced by \( \{ u_1, u_2, u_3, u_5 \} \)). This is because we have tried all possible extensions to map \( u_3 \) and \( u_5 \) below node \( M \) and all attempts have failed by the conflict in the mappings of \( u_2 \) and \( u_5 \), while \( u_4 \) has not been relevant to any of the failures. It follows that all the siblings of node \( M \) (depicted by a dashed box in Figure 6.1) will not lead to an embedding of \( \{ u_1, u_2, u_3, u_5 \} \) since they also contain \( M' = \{ (u_1, v_1), (u_2, v_2) \} \) as a subset.

In Example 6.2.1, identifying the set \( \{ u_1, u_2, u_3, u_5 \} \) is a key to prune partial embeddings. We call this set a **failing set**. A formal definition of the failing set and how to compute it follow in the next section.

### 6.3 Failing Set

#### 6.3.1 Definition of Failing Set

We give a formal definition of the failing set. We say that a set of query vertices \( S \subseteq V(q) \) is **parent-closed** in \( q_D \) if for any \( u \in S \), all the parents of \( u \) in \( q_D \) are also in \( S \). Consider a search tree for finding a query graph \( q \). For a node \( M \) in the search tree, we define a failing set \( F_M \subseteq V(q) \) as a parent-closed set satisfying the following **failure property**:

- The partial embedding \( M[F_M] \) cannot lead to an embedding of \( F_M \), where \( M[F_M] \) is the largest subset of \( M \) that is a partial embedding of \( F_M \), i.e., \( (u, v) \in M[F_M] \) if and only if \( (u, v) \in M \) and \( u \in F_M \).
If the subtree rooted at $M$ has an embedding of $q$, $F_M$ cannot be defined. In this case, we simply let $F_M = \emptyset$.

In Example 6.2.1, $F_M = \{u_1, u_2, u_3, u_5\}$ is a failing set of node $M$. Note that $F_M$ is parent-closed and it satisfies the failure property with $M[F_M] = \{(u_1, v_1), (u_2, v_2)\}$.

A merit of the failing set is that once we obtain a failing set of a node, we can prune some unexamined nodes of the search tree. We say that a node of the search tree is redundant if it cannot lead to an embedding of $q$ due to a failing set. Intuitively, a query vertex that is not in the failing set is irrelevant to the failures, and so partial embeddings obtained by changing the mapping of the irrelevant vertex are redundant.

**Lemma 6.3.1.** Suppose that we are given a search tree node $M = \{\ldots, (u, v)\}$ and a non-empty failing set $F_M$. If $u \notin F_M$, then all siblings of node $M$ (including itself) are redundant.

**Proof.** By definition of the failing set, $M[F_M]$ cannot lead to an embedding of $F_M$. Since $u \notin F_M$, every sibling of $M$ has $M[F_M]$ as a subset, and thus it cannot lead to an embedding of $F_M$ due to $M[F_M]$.

In Example 6.2.1, since $u_4 \notin F_M$, we can immediately conclude that all siblings of node $M$ are redundant by Lemma 6.3.1.

### 6.3.2 Computing Failing Sets

Now we describe how to compute the failing set $F_M$ of each node $M$ of the search tree. The failing sets are computed in the search tree in a bottom-up fashion.

We first define the base case, i.e., leaves of the search tree. The leaves are categorized into the following three classes.

1. A leaf belongs to the conflict-class if a conflict of mapping occurs in the leaf, i.e., represented by $(u, v)$! in Figure 5.2. For a conflict-class node
\[ M = \{(u', v), \ldots, (u, v)\} \text{, we set } F_M = \text{anc}(u) \cup \text{anc}(u') \text{, where } \text{anc}(u) \text{ denotes the set of all ancestors of } u \text{ in } q_D \text{ including } u \text{ itself. Clearly, } F_M \text{ is parent-closed and satisfies the failure property due to } M[F_M]. \]

(2) A leaf belongs to the emptyset-class if \( C_M(u) = \emptyset \) for the current query vertex \( u \), i.e., represented by \((u, \emptyset)\) in Figure 5.2. For an emptyset-class node \( M = \{(u, \emptyset)\} \), we set \( F_M = \text{anc}(u) \). Clearly, \( F_M \) is parent-closed and satisfies the failure property.

(3) A leaf belongs to the embedding-class if it is a full embedding of \( q \). For an embedding-class node \( M \), we cannot define a failing set. Thus, we set \( F_M = \emptyset \).

Now we compute the failing set of an internal node by using the failing sets of its children. Suppose that an internal node \( M \) has \( k \) children \( M_1, \ldots, M_k \) that are all extensions of \( M \) to the next vertex \( u_n \), i.e., \( M_i = M \cup \{(u_n, v_i)\} \) for all \( v_i \in C_M(u_n) \). Assume that we have computed the failing sets \( F_{M_1}, \ldots, F_{M_k} \) of \( M_1, \ldots, M_k \), respectively. We compute the failing set \( F_M \) of node \( M \) according to the following cases:

Case 1. If there exists a child node \( M_i \) such that \( F_{M_i} = \emptyset \), we set \( F_M = \emptyset \).

Case 2. Otherwise,

   Case 2.1. If there exists a child node \( M_i \) such that \( u_n \notin F_{M_i} \), we set \( F_M = F_{M_i} \).

   Case 2.2. Otherwise, we set \( F_M = \bigcup_{i=1}^k F_{M_i} \).

6.3.3 Correctness of Failing Set Computation

We now prove the correctness of the procedure to compute failing sets.

Lemma 6.3.2. The set \( F_M \) computed as above is a failing set of search tree node \( M \).
Proof. It is easy to see that the parent-closure property is always satisfied for \( F_M \). Hence it suffices to prove that the set \( F_M \) computed in Section 6.3 satisfies the failure property, which we prove by induction. The base case is already addressed. We now prove the inductive step. Assuming that \( F_{M_1}, \ldots, F_{M_k} \) satisfy the failure property, we prove that \( F_M \) satisfies the failure property in Cases 1 and 2. Note that Case 1 is trivial.

Case 2.1: Since \( F_{M_i} \) satisfies the failure property, \( M_i[F_{M_i}] \) cannot lead to an embedding of \( F_{M_i} \). We need to prove that \( F_M (= F_{M_i}) \) also satisfies the failure property for node \( M \). Since \( u_n \notin F_{M_i}, M[F_M] = M_i[F_{M_i}] \). Consequently, \( M[F_M] \) cannot lead to an embedding of \( F_{M_i} \).

Case 2.2: We will prove that \( M[F_M] \) cannot lead to an embedding of \( F_M \) by trying all extensions of \( M[F_M] \) to \( u_n \). Since \( u_n \in F_M \) and \( F_M \) is parent-closed, \( M[F_M] \) maps all the parents of \( u_n \), and the mapping of the parents of \( u_n \) in \( M[F_M] \) is the same as that in \( M \) because \( M[F_M] \) is a subset of \( M \). It means that the extendable candidates of \( u_n \) regarding \( M[F_M] \) are identical to the extendable candidates of \( u_n \) regarding \( M \), i.e., \( C_{M[F_M]}(u_n) = C_M(u_n) \). For every \( v_i \in C_{M[F_M]}(u_n) \), the corresponding extension \( M[F_M] \cup \{(u_n, v_i)\} \) contains \( M_i[F_{M_i}] \) (where \( M_i = M \cup \{(u_n, v_i)\} \)) as a subset since \( F_{M_i} \subseteq F_M \). Since \( M_i[F_{M_i}] \) cannot lead to an embedding of \( F_{M_i} \) by induction hypothesis, \( M[F_M] \cup \{(u_n, v_i)\} \) also cannot lead to an embedding of \( F_{M_i} \), and so it cannot lead to an embedding of \( F_M \). Since none of the extensions of \( M[F_M] \) to \( u_n \) lead to an embedding of \( F_M \), \( M[F_M] \) cannot lead to an embedding of \( F_M \).

\[ \square \]

6.4 Refined Backtracking Process

Exploiting the notion of failing sets, we can prune a lot of redundant partial embeddings. Algorithm 3 (i.e., BACKTRACK+) outlines the refined search process. When the failing set \( F_M \) is \( \emptyset \), we set \( F_M = V(q) \) in Algorithm 3 (line 3) because \( \emptyset \) and \( V(q) \) make no difference in implementation, i.e., no pruning occurs at all.
Algorithm 3: \textsc{Backtrack+}(q,q_D,CS,M)

1. \textbf{if} $|M| = |V(q)|$ \textbf{then}
   2. Report $M$;
   3. return $V(q)$;
4. \textbf{else if} $|M| = 0$ \textbf{then}
   5. \textbf{foreach} $v \in C(r)$ \textbf{do}
     6. $M \leftarrow \{(r,v)\}$; Mark $v$ as visited;
     7. \textsc{Backtrack+}(q,q_D,CS,M); Mark $v$ as unvisited;
     8. return $\emptyset$; /* return value that is not used */
9. \textbf{else}
10. $u \leftarrow$ extendable vertex with minimum weight $w_M(u)$;
11. \textbf{if} $C_M(u) = \emptyset$ \textbf{then}
    12. $F_M \leftarrow \text{anc}(u)$;
13. \textbf{else}
    14. $F_M \leftarrow \emptyset$;
15. \textbf{foreach} $v \in C_M(u)$ \textbf{do}
16. \textbf{if} $v$ is unvisited \textbf{then}
    17. $M' \leftarrow M \cup \{(u,v)\}$; Mark $v$ as visited;
    18. $F_{M'} \leftarrow \textsc{Backtrack+}(q,q_D,CS,M')$; Mark $v$ as unvisited;
    19. \textbf{if} $u \notin F_{M'}$ \textbf{then}
    20. $F_M \leftarrow F_{M'}$;
    21. break;
    22. \textbf{else}
    23. $F_{M'} \leftarrow \text{anc}(u) \cup \text{anc}(M^{-1}(v))$;
    24. $F_M \leftarrow F_M \cup F_{M'}$;
25. return $F_M$;

35
along the path to the root, but using \( V(q) \) makes the algorithm simpler. Now the recursive procedure computes and returns a failing set \( F_M \) (lines 3, 25) of the current node \( M \). Lines 11-12 correspond to the emptyset-class. In line 14, \( F_M \) is initialized to an empty set. Then, for each \( v \in C_M(u) \), we compute \( F_{M'} \) as the failing set of the child node \( M' = M \cup \{(u, v)\} \). If \( v \) is unvisited, \( F_{M'} \) is computed by the recursive invocation of BACKTRACK + (line 18), and otherwise \( F_{M'} \) is computed in lines 22-23 as a conflict-class. \( F_M \) is computed as the union of all \( F_{M'} \)'s (line 24) according to Case 2.2. In the case of \( u \notin F_{M'} \) (Case 2.1), we set \( F_M = F_{M'} \) and stop extending the current node (lines 19-21) by Lemma 6.3.1.

Figure 6.2 shows the result of redundancy pruning for the left half of the search tree in Figure 5.2. Note that the redundant nodes enclosed by dashed boxes are successfully pruned out.
Chapter 7

Performance Evaluation

In this chapter we present experimental results to show the effectiveness of our algorithm DP-iso. For performance evaluation, we compare state-of-the-art algorithms TurboISO [13] and CFL-Match [5] against DP-iso. Since CFL-Match significantly outperformed the other existing algorithms including TurboISO [5], our main experiments compare the performances of DP-iso and CFL-Match. Then the comparison between DP-iso and TurboISO will be presented. We also present the result of applying the boosting technique in [23] to DP-iso, and we evaluate the effects of our pruning technique described in Chapter 6. Finally, we perform additional experiments to study the correlation between the algorithms’ behaviors and the number of edges of query graphs.

7.1 Experimental Settings

7.1.1 Environment

The source codes of CFL-Match and the boosting technique, and the executable file of TurboISO were obtained from their authors. DP-iso is implemented in C++ as CFL-Match is. Experiments are conducted on a machine with two Intel
Xeon E5-2680 v3 2.50GHz CPUs and 256GB memory running CentOS Linux, except for the experiments comparing with TurboISO. Since the executable file of TurboISO is Windows-based, these experiments are conducted on a machine with an Intel i7-4790K 4.00GHz CPU and 8GB memory running Windows 8.1 Pro.

| Dataset (G) | |V(G)| | |E(G)|| | | |Σ| Avg degree |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Yeast | 3,112 | 12,519 | 71 | 8.04 |
| Human | 4,674 | 86,282 | 44 | 36.91 |
| HPRD | 9,460 | 37,081 | 307 | 7.83 |
| Email | 36,692 | 183,831 | 20 | 10.02 |
| Wordnet | 82,670 | 121,307 | 5 | 2.93 |
| DBLP | 317,080 | 1,049,866 | 20 | 6.62 |

Table 7.1: Characteristics of datasets

7.1.2 Datasets

We use six real datasets in our experiments: Yeast, Human, HPRD, Email\(^1\), Wordnet\(^2\), and DBLP\(^3\), which are commonly used in previous works [28, 18, 23, 13, 5]. As no label information is available for Email and DBLP, we randomly assigned a label out of 20 distinct labels to each vertex. The characteristics of the datasets are given in Table 7.1. We can expect that Yeast and HPRD are easy data graphs for subgraph matching due to moderate average degrees and plentiful distinct labels, and the rest are relatively harder data graphs due to higher average degrees and/or fewer distinct labels.

\(^1\)http://snap.stanford.edu/data/com-DBLP.html, \(^2\)http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.htm

\(^3\)http://snap.stanford.edu/data/com-DBLP.html, \(~\)data/email-Enron.html

38
7.1.3 Query Graphs

For each data graph, we generate 12 query sets $Q_{N\text{min}}$, $Q_{Navg}$, $Q_{N\text{max}}$, where $N \in \{100, 200, 300, 400\}$ for Yeast and HPRD, and $N \in \{10, 20, 30, 40\}$ for Human, Email, Wordnet, and DBLP. Each query set contains 100 query graphs of the same size $N$ in terms of the number of vertices, e.g., $Q_{N\text{min}}$ contains 100 query graphs with $N$ vertices. A query graph is generated as a connected subgraph of the data graph to ensure that every query graph has at least one embedding. To extract a subgraph, we first perform a random walk on the data graph until we visit $N$ distinct vertices. From the $N$ visited vertices, we generate three query graphs $q_{N\text{min}}$, $q_{Navg}$, $q_{N\text{max}}$, which are then inserted to $Q_{N\text{min}}$, $Q_{Navg}$, $Q_{N\text{max}}$, respectively, where

- $q_{N\text{min}}$ consists of all the vertices and edges visited during the random walk.
- $q_{N\text{max}}$ consists of the $N$ visited vertices and all the edges between the vertices, i.e., $q_{N\text{max}}$ is an induced subgraph.
- $q_{Navg}$ consists of $q_{N\text{min}}$ and additional randomly chosen edges between the $N$ vertices such that the total number of edges is the average of the numbers of edges in $q_{N\text{min}}$ and $q_{N\text{max}}$.

7.1.4 Performance Measurement

To evaluate an algorithm, we measure the time in milliseconds to process each query graph in a query set. The processing time for a query graph consists primarily of the preprocessing time (i.e., time to build auxiliary data structures) and the search time (i.e., time to enumerate the first $k = 10^5$ embeddings). Since the search is the most time-consuming part, we simply use the total running time as a performance measure. An issue is that since we process query graphs of larger scales than previous works, some queries may be bad instances, i.e., processing a query cannot finish in a reasonable time. Note that since subgraph
isomorphism is an NP-hard problem, any algorithm will have bad instances. To address this issue, we set a time limit of 10 minutes for each query graph. We say that a query graph is solved if it terminates within the time limit. Our empirical study showed that the time limit of 10 minutes is reasonable; we also tried one hour as the time limit and additional solved queries were few.

With the existence of unsolved queries, comparing two algorithms fairly is not easy. Figure 7.1 shows the distribution of the running times of CFL-Match and DP-iso for query set $Q_{20min}$ of Human. The $x$-axis represents the query instances sorted in terms of running time for each respective algorithm and the $y$-axis shows the running time for a query instance (running times of unsolved queries are capped at 10 minutes). We can see that for easy queries, CFL-Match behaves slightly better than DP-iso, but for hard queries, DP-iso behaves orders-of-magnitude better than CFL-Match. Moreover, CFL-Match has more unsolved queries. This shows that DP-iso is superior to CFL-Match in two respects. To evaluate an algorithm regarding a query set, therefore, we report
the average running time and the average number of recursions (i.e., examined nodes in the search tree) for the $n$ least time-consuming queries where $n$ is the minimum among the numbers of solved queries in the compared algorithms. We also report the percentage of solved queries to show the robustness of the algorithms. A similar method is usually employed for experiments on large-scale graphs [35, 15].

7.2 Experimental Results

7.2.1 Comparing with CFL-Match

We compare DP-iso against CFL-Match. Figure 7.2 shows the experimental results for DP-iso and CFL-Match. For the query sets where CFL-Match fails to solve all 100 queries (i.e., $Q_{200\min}, Q_{300\min}, Q_{400\min}$ of Yeast), the running time is not presented. We can see that DP-iso consistently outperforms CFL-Match with respect to the percentage of solved queries and the average number of recursions. Specifically, CFL-Match solves 81% of queries while DP-iso solves all queries for $Q_{40\max}$ of Email (see Figure 7.2i). Also, CFL-Match solves 60% of queries while DP-iso solves 93% of queries for $Q_{40\max}$ of DBLP (see Figure 7.2j). In the average number of recursions, DP-iso outperforms CFL-Match by up to 6 orders of magnitude for DBLP (see $Q_{40\min}$ in Figure 7.2k) and up to 5 orders of magnitude for Human (see $Q_{40\min}$ in Figure 7.2b), Yeast (see $Q_{200\avg}$ in Figure 7.2e), Email (see $Q_{30\min}$ in Figure 7.2h), and Wordnet (see $Q_{40\min}$ in Figure 7.2n). In the average running time, DP-iso outperforms CFL-Match in most cases. Specifically, DP-iso outperforms CFL-Match in the average running time by up to 4 orders of magnitude for Email (see $Q_{40\avg}$ in Figure 7.2g) and up to 3 orders of magnitude for Human (see $Q_{20\min}$ in Figure 7.2a), Yeast (see $Q_{100\min}$ in Figure 7.2d), and DBLP (see $Q_{40\max}$ in Figure 7.2j). For Human, Yeast, Email, and DBLP, DP-iso runs consistently faster than CFL-Match. For Wordnet and HPRD, however, DP-iso runs slightly slower than CFL-Match for
Figure 7.2: DP-iso versus CFL-Match.
some query sets. Nevertheless, DP-iso consistently outperforms CFL-Match in the average number of recursions. This discrepancy between the running time and the number of recursions is due to the overhead of additional computations during the search of DP-iso (e.g., computing weights for the adaptive matching order and failing sets), resulting in a higher average cost per recursion. We observe that in the query sets for which DP-iso runs slower than CFL-Match (e.g., $Q_{10min}$ of Wordnet and $Q_{400max}$ of HPRD), all the 100 queries are answered within average 100 ms, which means that these queries are easy instances.

7.2.2 Comparing with TurboISO

To compare with TurboISO, we communicated with the authors, but we could obtain only the executable file of TurboISO. Since the program has different environmental requirements (i.e., Windows-based), we conducted independent experiments. In the experiments comparing DP-iso with TurboISO, we set $k = 1000$ as the number of embeddings to report, since it is fixed to 1000 in TurboISO. Since TurboISO always crashes for queries of size greater than 25, we conduct experiments with query sizes up to 25. Note that the query graphs in our main experiments comparing DP-iso and CFL-Match are of larger sizes. The results are shown in Figure 7.3. TurboISO contains a preprocessing (i.e., EXPLORECR) which is a recursive process with exponential time bounds, and thus the number of recursions of the backtracking process (which the program outputs) may not reflect its performance (note that EXPLORECR itself finds all embeddings for path queries, and so the number of recursions in backtracking is 0). Hence, we omit the results for the average number of recursions in Figure 7.3.

For Yeast, DBLP, and HPRD, DP-iso consistently outperforms TurboISO (by up to 2 orders of magnitude in the average running time for DBLP; see $Q_{25avg}$, $Q_{25max}$ in Figure 7.3g). For Human and Email, DP-iso outperforms TurboISO for avg and max queries (by up to 3 orders of magnitude in the average running time for Human; see $Q_{25max}$ Figure 7.3a). For min queries of
Figure 7.3: DP-iso
Human and Email, however, DP-iso and TurboISO are comparable. For Wordnet, TurboISO outperforms DP-iso. These cases can be explained by the average degree of query graphs. We observe that all query sets for which TurboISO shows competitive performances against DP-iso have low average degrees, i.e., approximately 2. Such query graphs have few non-tree edges, in which case the advantages of DP-iso are not fully expressed. From the comparison between DP-iso and CFL-Match in our main experiments and the comparison between CFL-Match and TurboISO in [5], however, we expect that DP-iso will significantly outperform TurboISO as the sizes of query graphs increase.

7.2.3 Evaluating Boosting Technique

We applied the boosting technique in [23] to DP-iso. Their boosting technique considers equivalence relationships (i.e., SE/QDE) and containment relationships (i.e., SC/QDC) in data vertices to speed up a subgraph matching algorithm. However, we have found that their method of exploiting containment relationships (called dynamic candidate loading in [23]) may miss some embeddings. Thus, our boosted version of DP-iso, called DP-iso-Boost, considers only the equivalence relationships. The results are shown in Figure 7.4. The effect of the boosting technique varies with data graphs. For Human, DP-iso-Boost shows a considerable improvement over DP-iso. For Email, DP-iso and DP-iso-Boost are comparable. For HPRD, the boosting technique seems not helpful. We remark that these differences are due to the compression ratio of the data graph by the boosting technique. The compression ratios of Human, Email, and HPRD are about 53.1%, 16.5%, and 1.4%, respectively. As expected, the boosting technique can be highly effective when the data graph has many similarities in itself.
Figure 7.4: Evaluating boosting technique
Figure 7.5: Effects of pruning technique
7.2.4 Evaluating Pruning Technique by Failing Sets

We evaluate the effects of our pruning technique by failing sets in Chapter 6. For this purpose, we run DP-iso-w-o-Prune, which is a variant of DP-iso without the pruning technique, and compare it with CFL-Match and DP-iso. The results are shown in Figure 7.5. We can see that DP-iso-w-o-Prune outperforms CFL-Match in most cases except for too easy instances. It shows that our framework with dynamic programming on DAG and the adaptive matching order (Algorithm 2) is effective in itself. Our pruning technique further improves DP-iso-w-o-Prune by pruning redundant partial embeddings. The improvement by the pruning technique is the most significant on Yeast due to larger query sizes (i.e., more redundancies in the search tree).

7.2.5 Additional Experiments

To study the correlation between algorithms’ behaviors and the number of edges of query graphs, we performed experiments with additional query sets that have more edges than max queries (i.e., $Q_{N_{max}}$). For each data graph, we generate a query set $Q_{N_{max}+p\%}$ (with varying $p > 0$) containing 100 query graphs with $N$ vertices, where each query graph $q_i' \in Q_{N_{max}+p\%}$ $(1 \leq i \leq 100)$ is generated by adding random edges to each query graph $q_i \in Q_{N_{max}}$ $(1 \leq i \leq 100)$ until $q_i'$ has $p$-percent more edges than $q_i$. It is clear that $q_i'$ has less or equal number of embeddings compared to $q_i$. Also, now we have no guarantee that each query graph in $Q_{N_{max}+p\%}$ has at least one embedding in the data graph, while it is the case for $Q_{N_{max}}$.

With the new query sets, we compared CFL-Match and DP-iso. The experimental settings (e.g., $k = 10^5$, time limit of 10 minutes) are the same as in the main experiment. For Human, we test $Q_{20_{max}+p\%}$ for $p \in \{1.25, 2.5, 3.75, 5, 6.25\}$. For Email, we test $Q_{40_{max}+p\%}$ for $p \in \{10, 20, 30, 40, 50\}$. For Wordnet and DBLP, we test $Q_{40_{max}+p\%}$ for $p \in \{20, 40, 60, 80, 100\}$. For Yeast and HPRD, we test $Q_{100_{max}+1}$ and $Q_{100_{max}+2}$, where each query graph in $Q_{100_{max}+1}$ (resp.,
\( Q_{100\max} + 2 \) is obtained by adding one (resp., two) random edge(s) to the query graphs in \( Q_{100\max} \).

To evaluate an algorithm, for each query set, we measure the following three metrics:

- \( \%\text{filtered} \) is the percentage of queries that are filtered. We say that a query graph is filtered if the filtering process (i.e., CPI/CS construction) reveals that there is no embedding of the query graph.

- \( \%\text{finished} \) is the percentage of queries that are finished. We say that a query graph is finished if all embeddings of the query graph in the data graph has been found (and it is confirmed) within the time limit. If a query is finished, it means that the whole search space is explored. All finished queries have less than \( 10^5 \) embeddings.

- \( \%\text{solved} \) is the percentage of solved queries. This is the same as before.

There are containment relationships among the three groups. That is, the finished queries contain the filtered queries and the solved queries contain the finished queries. Also, we measure the average running time in the same way as in Section 7.1.4.

Figure 7.6 shows the experimental results. For DBLP and Email, \( \%\text{filtered} \) is not presented since it was 0 for all query sets for both algorithms. For Yeast and HPRD, almost all queries are filtered out even in the case when we add only one edge to each query graph. Since DP-iso does more work in the filtering stage and the running time is dominated by the preprocessing time in this case, DP-iso runs slower than CFL-Match. For Human, as the number of edges increases, \( \%\text{filtered} \) consistently increases for both algorithms. Note that \( \%\text{filtered} \) and \( \%\text{solved} \) of DP-iso are always slightly higher than those of CFL-Match. Also, in the average running time, DP-iso always significantly outperforms CFL-Match (this is always the case except for Yeast/HPRD). For DBLP, very few queries
Figure 7.6: Additional experiments

(a) Human (3 Metrics)  
(b) Human (running time)  
(c) Yeast (3 Metrics)  
(d) Yeast (running time)  
(e) Email (3 Metrics)  
(f) Email (running time)  
(g) DBLP (3 Metrics)  
(h) DBLP (running time)  
(i) Wordnet (3 Metrics)  
(j) Wordnet (running time)  
(k) HPRD (3 Metrics)  
(l) HPRD (running time)
are finished for both algorithms. This is because DBLP has a relatively large search space (i.e., many vertices and small $|\Sigma|$, see Table 7.1). As the number of edges increases, $\%solved$ tends to decrease for both algorithms. This is because as the number of edges increases, the number of embeddings decreases (i.e., the embeddings are more sparsely distributed in the search space), and thus it takes longer time to find the first $10^5$ embeddings. Nevertheless, DP-iso always solves more queries than CFL-Match. For Email, however, as the number of edges increases, $\%finished$ rapidly increases for both algorithms. From this we can infer that the search space of Email is relatively small. Intuitively, as more edges are added to a query graph, the whole search space for finding all embeddings of it will become smaller, and more finished queries may appear (as shown in Figure 7.6e). It is remarkable that DP-iso finishes most queries for $Q_{40max+50\%}$. For Wordnet, both $\%filtered$ and $\%finished$ consistently increase as the number of edges increases in both algorithms. For $Q_{40max+80\%}$, $Q_{40max+100\%}$, especially, all solved queries are filtered or finished by both algorithms. We can observe that as the number of edges increases, $\%solved$ decreases to a certain point ($Q_{40max+60\%}$), however, it increases after that point for both algorithms due to the increase of the number of finished queries. We can see that DP-iso always finishes and solves much more queries than CFL-Match.
Chapter 8

Conclusion

In this thesis we have proposed an efficient and robust subgraph matching algorithm that addresses the limitations of existing techniques and the challenges of subgraph matching. Our algorithm more fully exploits the general framework of backtracking by using novel techniques. First, we proposed dynamic programming on DAG to compute the CS structure that serves as a complete and compact search space. Second, we proposed a backtracking framework based on DAG-ordering and the adaptive matching order, which together lead to a good matching order. Finally, we proposed an additional pruning technique based on failing sets to reduce redundant computations in the search tree. Extensive experiments on real datasets show that our algorithm outperforms the state-of-the-art algorithms by up to several orders of magnitude. Applying our techniques to some other problems related to subgraph isomorphism would be an interesting future work.
Bibliography


